http://algs4.cs.princeton.edu

ANALYSIS OF ALGORITHMS

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan
Computer Engineering Department
University of Jordan

http://algs4.cs.princeton.edu

Analysis of Algorithms

- *introduction*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

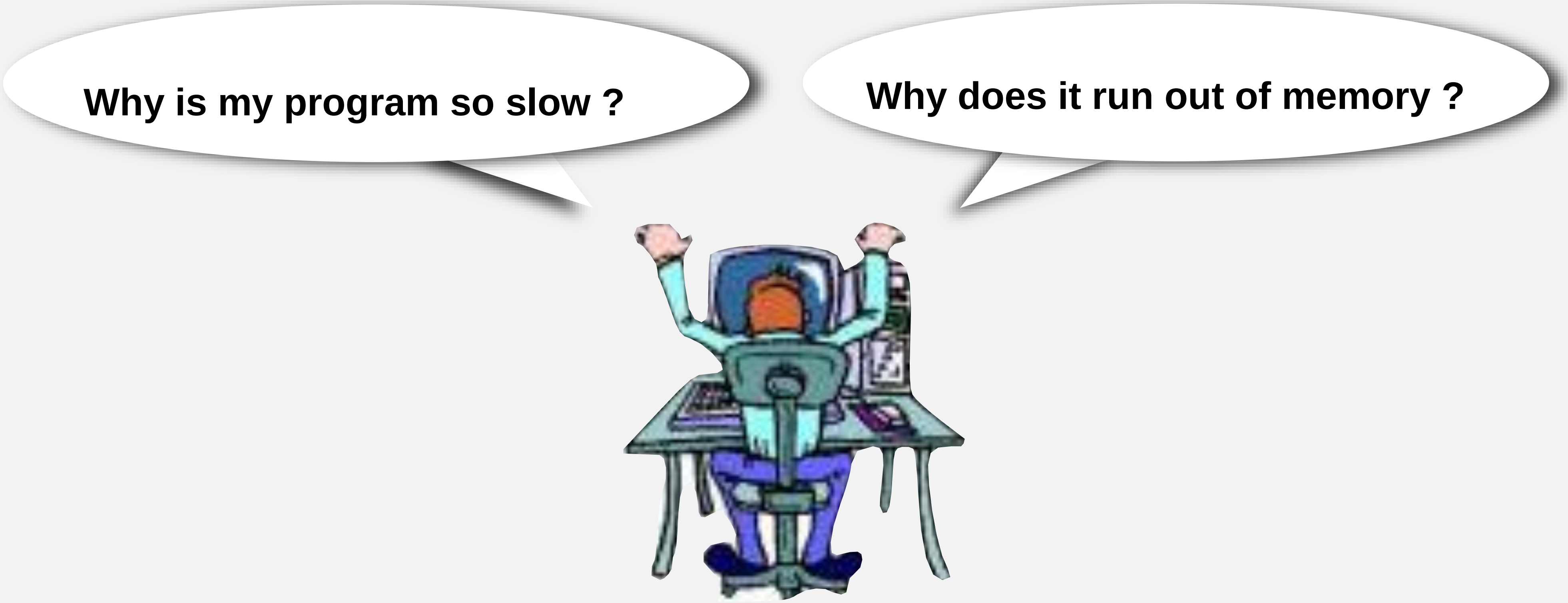
Provide guarantees.

Primary practical reason: avoid performance bugs.

client gets poor performance because programmer did not understand performance characteristics

The challenge

Q. Will my program be able to solve a large practical input?



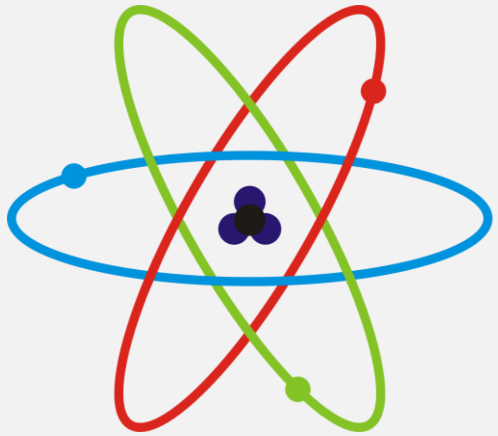
Scientific method applied to analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

- Observe some feature of the natural world.
- Hypothesize a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

Feature of the natural world. Computer itself.



Example: 3-SUM

3-SUM. Given N distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

Context. Deeply related to problems in computational geometry.

3-SUM: brute-force algorithm

```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int count = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int[] a = in.readAllInts();
      StdOut.println(count(a));
   }
}
```

check each triple

for simplicity, ignore integer overflow

Measuring the running time

`public class Stopwatch`	(part of `stdlib.jar`)
`Stopwatch()`	*create a new stopwatch*
`double elapsedTime()`	*time since creation (in seconds)*

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   int[] a = in.readAllInts();
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
   StdOut.println("elapsed time " + time);
}
```

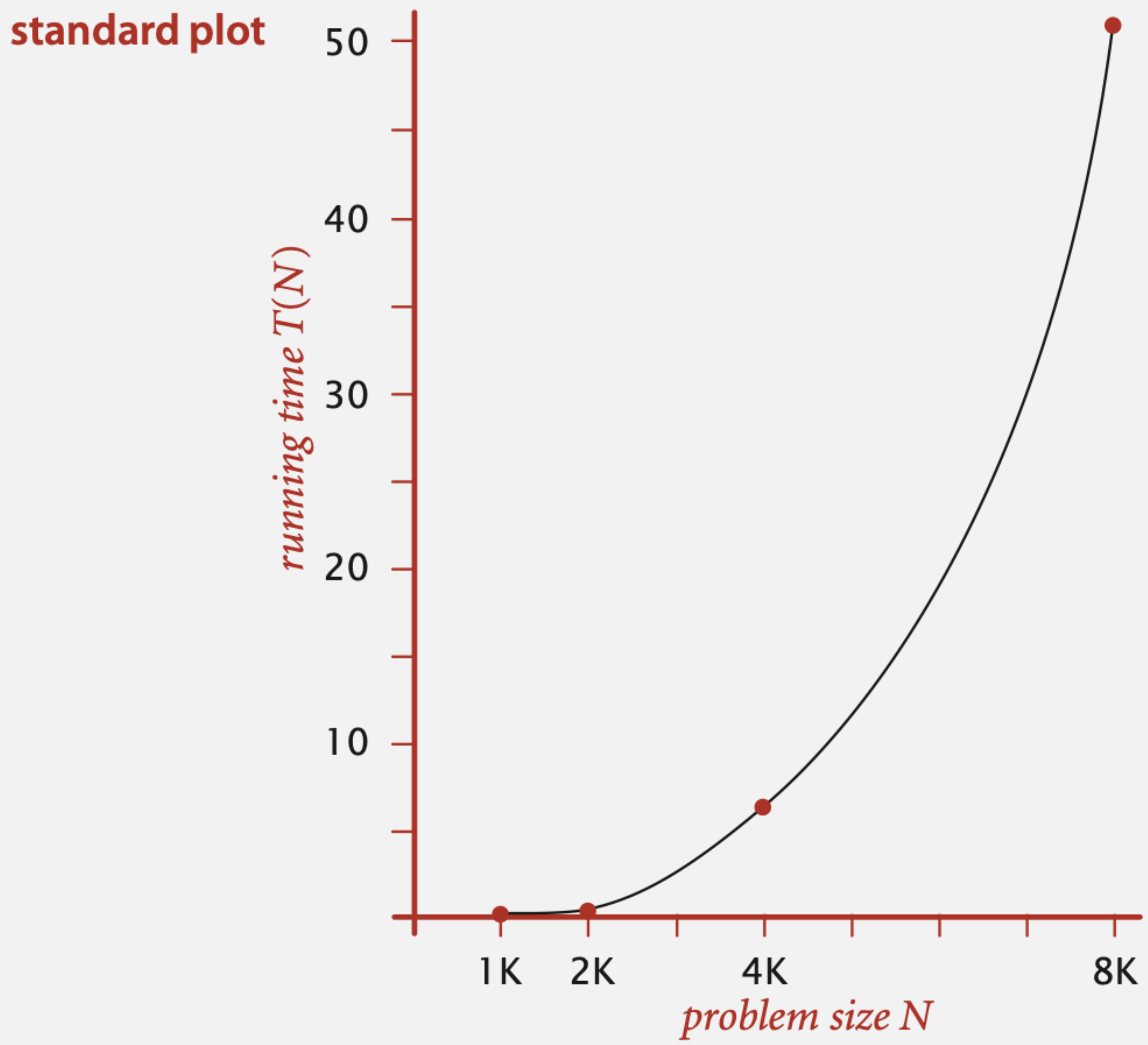
Empirical analysis

Run the program for various input sizes and measure running time.

N	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

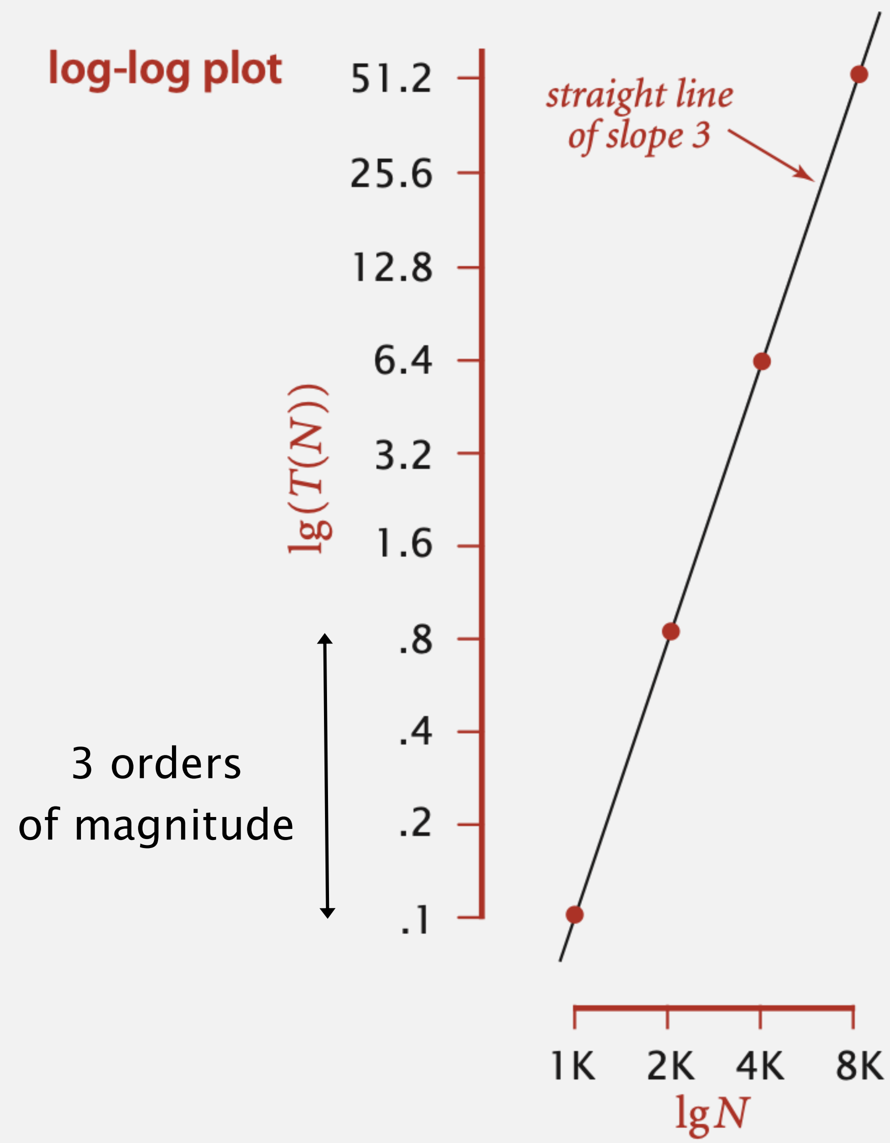
Data analysis

Standard plot. Plot running time $T(N)$ vs. input size N.



Data analysis

Log-log plot. Plot running time $T(N)$ vs. input size N using log-log scale.



$\lg(T(N)) = b \lg N + c$

$b = 2.999$

$c = -33.2103$

$T(N) = a N^b$, where $a = 2^c$

power law

slope

Regression. Fit straight line through data points: $a N^b$.

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Predictions.

- 51.0 seconds for $N = 8{,}000$.
- 408.1 seconds for $N = 16{,}000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1
16,000	410.8

validates hypothesis!

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent in power law

System dependent effects.

- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other apps, …

determines constant in power law

http://algs4.cs.princeton.edu

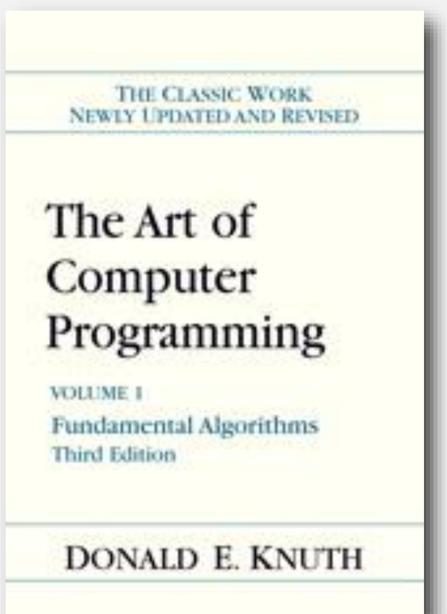
Analysis of Algorithms

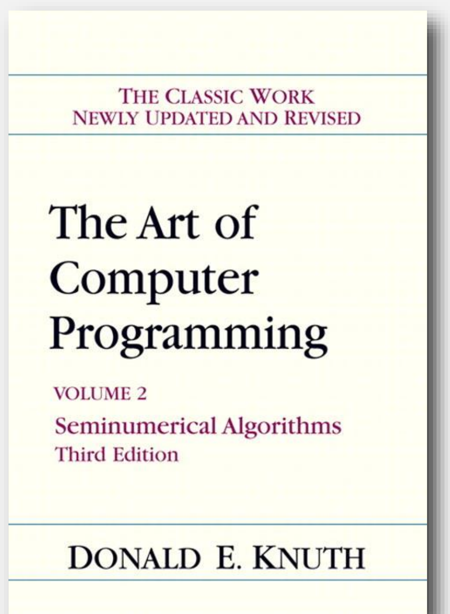
- *Introduction*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

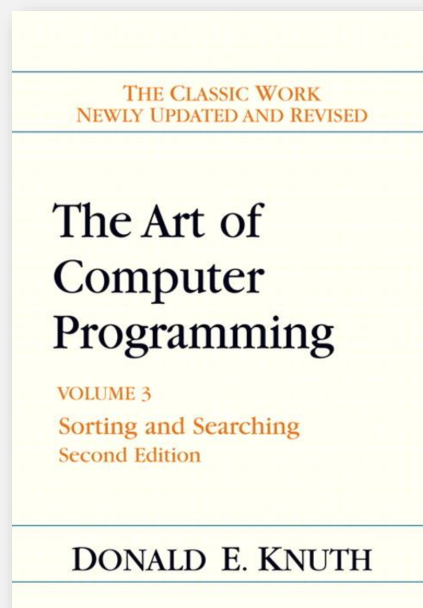
Mathematical models for running time

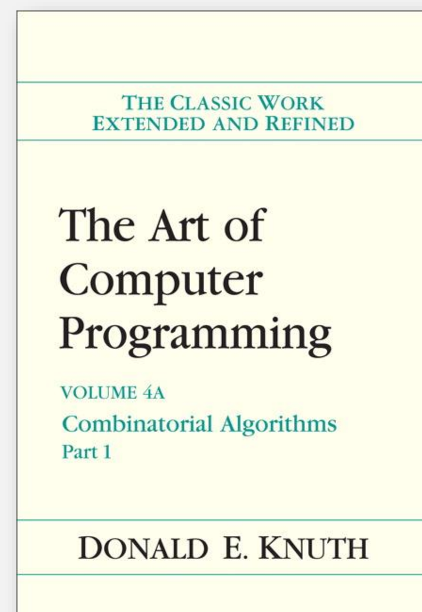
Total running time: **sum of cost × frequency for all operations.**

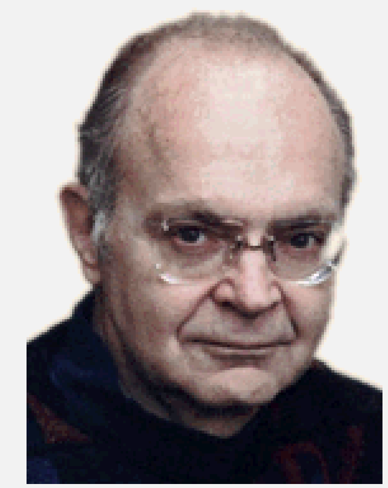
- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.











Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

Example: 1-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2\ N$

Example: 2-SUM

Q. How many instructions as a function of input size N ?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N(N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$½\,(N+1)\,(N+2)$
equal to compare	$½\,N\,(N-1)$
array access	$N\,(N-1)$
increment	$½\,N\,(N-1)$ to $N\,(N-1)$

tedious to count exactly

Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

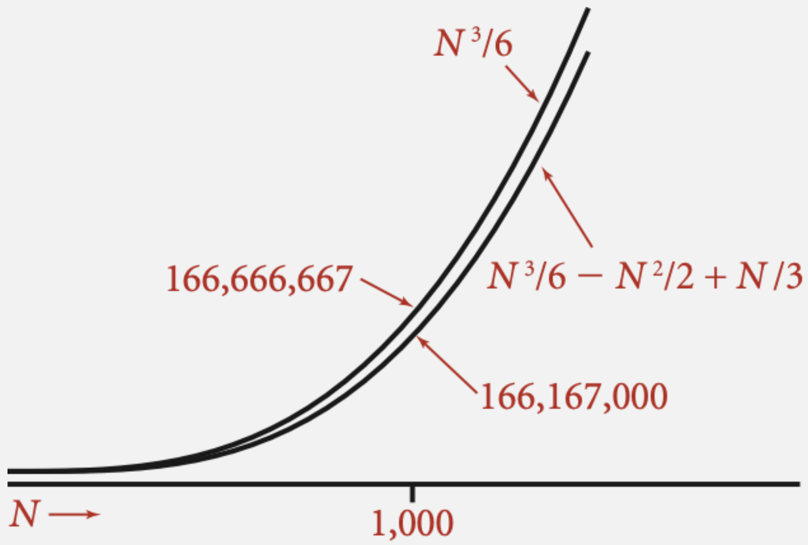
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N+2$
assignment statement	$N+2$
less than compare	$½ (N+1)(N+2)$
equal to compare	$½ N (N-1)$
array access	$N (N-1)$
increment	$½ N (N-1)$ to $N (N-1)$

cost model = array accesses

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care



Leading-term approximation

Ex 1. $\tfrac{1}{6} N^3 + 20 N + 16 \sim \tfrac{1}{6} N^3$

Ex 2. $\tfrac{1}{6} N^3 + 100 N^{4/3} + 56 \sim \tfrac{1}{6} N^3$

Ex 3. $\tfrac{1}{6} N^3 \underbrace{- \tfrac{1}{2} N^2 + \tfrac{1}{3} N}_{} \sim \tfrac{1}{6} N^3$

discard lower-order terms

(e.g., N = 1000: 166.67 million vs. 166.17 million)

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Example: 2-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

"inner loop"

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N(N-1) = \binom{N}{2}$$

A. $\sim N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

A. $\sim \frac{1}{2} N^3$ array accesses.

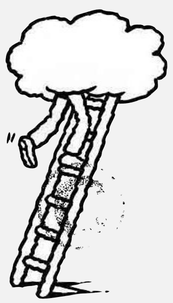
Bottom line. Use cost model and tilde notation to simplify counts.

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$A =$ array access
$B =$ integer add
$C =$ integer compare
$D =$ increment
$E =$ variable assignment

frequencies
(depend on algorithm, input)

Bottom line. We use approximate models in this course: $T(N) \sim c N^3$.

http://algs4.cs.princeton.edu

Analysis of Algorithms

- introduction
- mathematical models
- **order-of-growth classifications**
- theory of algorithms
- memory

Common order-of-growth classifications

Definition. If $f(N) \sim c\, g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is N^3.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

Typical usage. With running times.

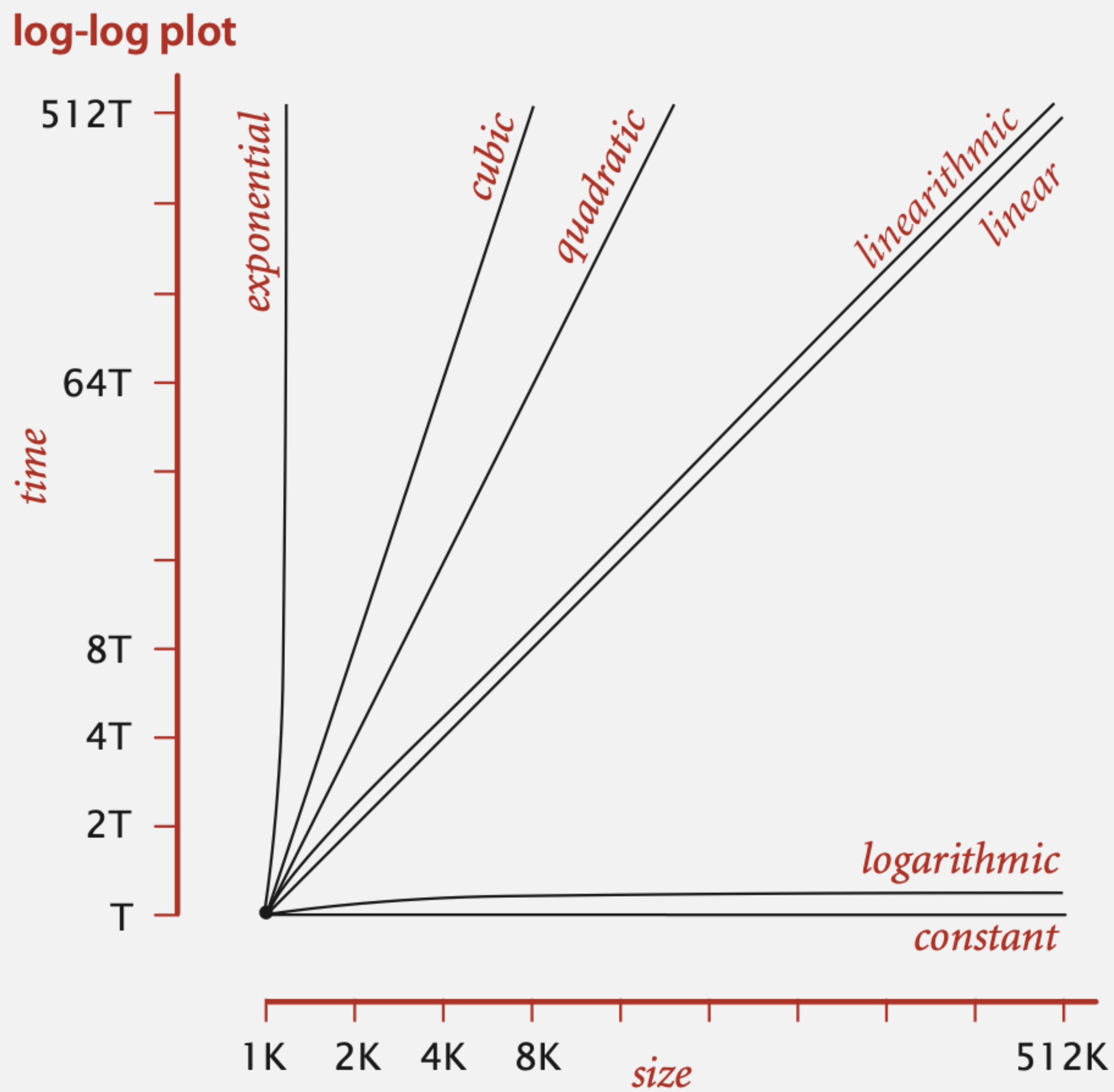
where leading coefficient depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

$$1, \ \log N, \ N, \ N \log N, \ N^2, \ N^3, \text{ and } 2^N$$

suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2

N^2	**quadratic**	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`{ ... }`
			double loop	check all pairs	4

N^3	**cubic**	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`for (int k = 0; k < N; k++)`
		`{ ... }`
			triple loop	check all triples	8

2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

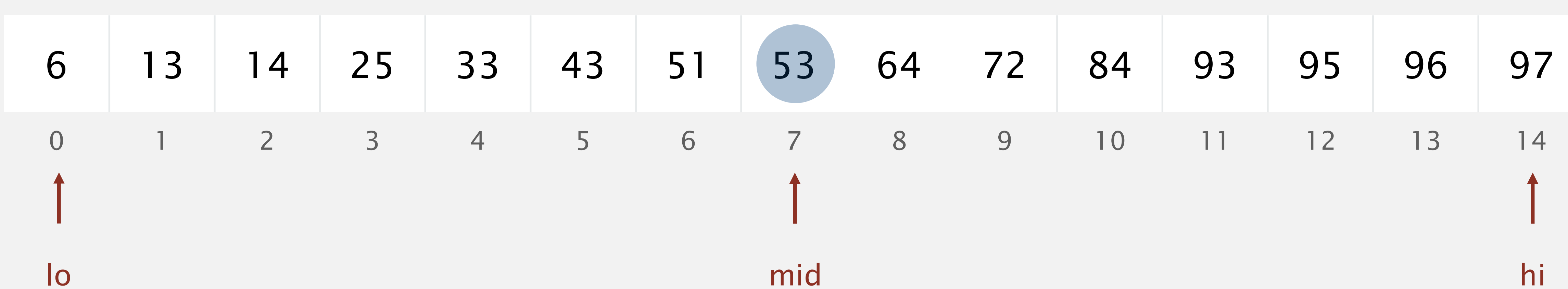
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



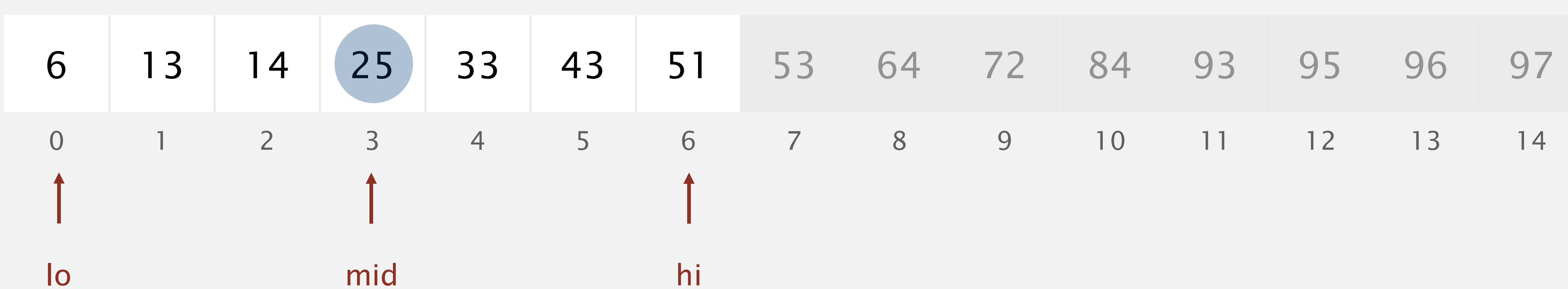
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



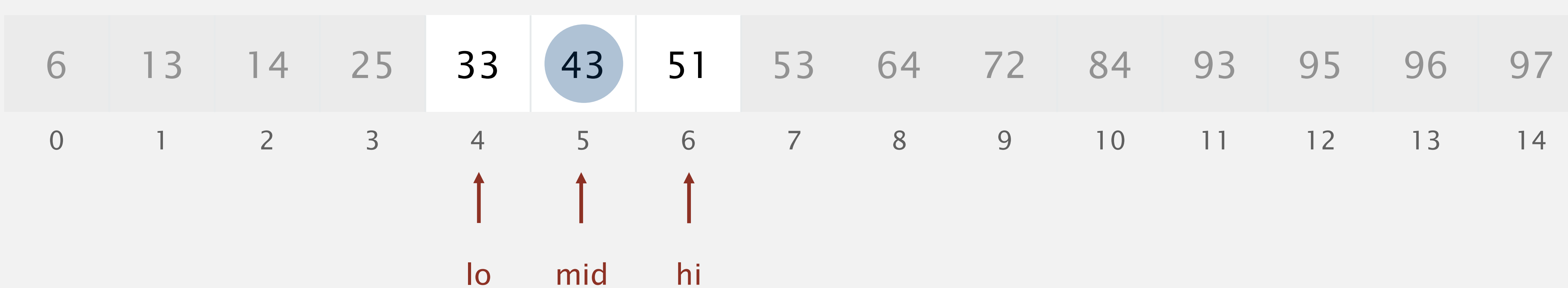
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



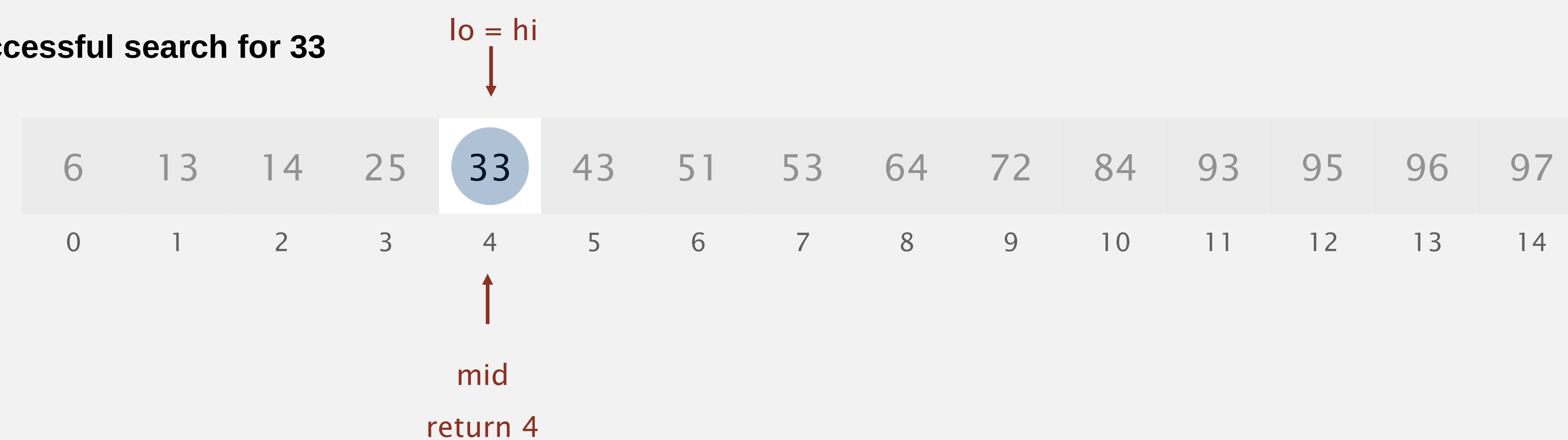
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;
}
```

one "3-way compare"

Invariant. If `key` appears in the array `a[]`, then `a[lo]` ≤ `key` ≤ `a[hi]`.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \log N$ key compares to search in a sorted array of size N.

Def. $T(N)$ = # key compares to binary search a sorted subarray of size ≤ N.

Binary search recurrence. $T(N) \le \underbrace{T(N/2)}_{\text{left or right half (floored division)}} + \underbrace{1}_{\text{possible to implement with one 2-way compare (instead of 3-way)}}$ for $N > 1$, with $T(1) = 1$.

Pf sketch. [assume N is a power of 2]

$$\begin{aligned}
T(N) &\le T(N/2) + 1 && \text{[given]} \\
&\le T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\
&\le T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\
&\ \ \vdots \\
&\le T(N/N) + 1 + 1 + \ldots + 1 && \text{[stop applying, } T(1) = 1 \text{]} \\
&= 1 + \log N
\end{aligned}$$

An N^2 log N algorithm for 3-SUM

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers `a[i]` and `a[j]`, binary search for `-(a[i] + a[j])`.

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

input

`30 -40 -20 -10 40 0 10 5`

sort

`-40 -20 -10 0 5 10 30 40`

binary search

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-20, -10)	30
⋮	⋮
(-10, 0)	10
⋮	⋮
(10, 30)	-40
(10, 40)	-50

only count if a[i] < a[j] < a[k] to avoid double counting

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

http://algs4.cs.princeton.edu

Analysis of Algorithms

- *Introduction*
- *mathematical models*
- *order-of-growth classifications*
- ***theory of algorithms***
- *memory*

Types of analyses

Best case. Lower bound on cost.

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

this course

Average case. Expected cost for random input.

- Need a model for "random" input.
- Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

Theory of algorithms: example 1

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 1-SUM = "*Is there a 0 in the array?*"

Upper bound.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound.

- Ex. Have to examine all N entries.
- Running time of the algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the improved algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound.

- Running time of the improved algorithm for 3-SUM is $\Omega(N^2)$.
- Running time of a **magical optimal** algorithm for solving 3-SUM is $\Omega(N)$. (why?)

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s-.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than "to within a constant factor" to predict performance.

http://algs4.cs.princeton.edu

Analysis of Algorithms

- *introduction*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Basics

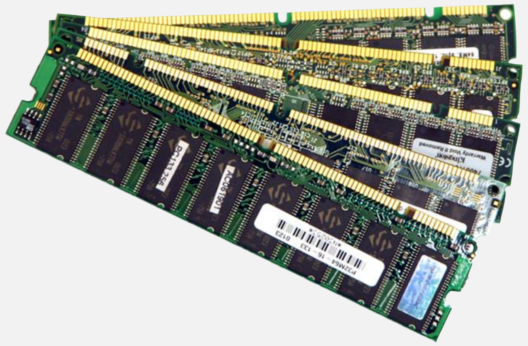
Bit. 0 or 1.

NIST most computer scientists

Byte. 8 bits.

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.



64-bit machine. We assume a 64-bit machine with 8-byte pointers.

- Can address more memory.
- Pointers use more space.

some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

primitive types

type	bytes
`char[]`	$2N + 24$
`int[]`	$4N + 24$
`double[]`	$8N + 24$

one-dimensional arrays

type	bytes
`char[][]`	$\sim 2MN$
`int[][]`	$\sim 4MN$
`double[][]`	$\sim 8MN$

two-dimensional arrays

Typical memory usage for objects in Java

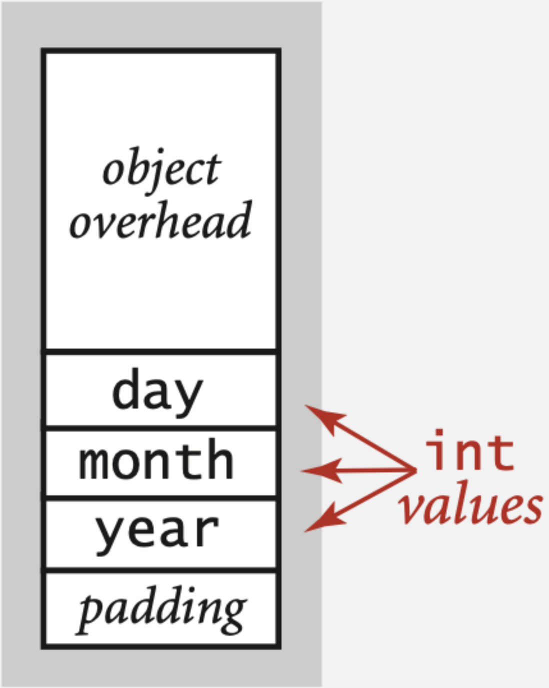
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A `Date` object uses 32 bytes of memory.

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```



16 bytes (object overhead)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

32 bytes

Typical memory usage for objects in Java

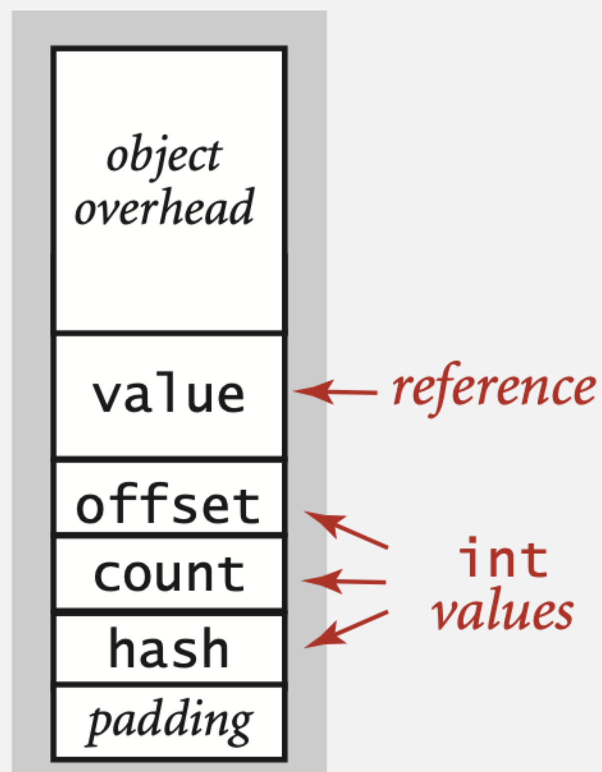
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 2. A virgin `String` of length N uses $\sim 2N$ bytes of memory.

```
public class String
{
   private char[] value;
   private int offset;
   private int count;
   private int hash;
...
}
```



16 bytes (object overhead)

8 bytes (reference to array)

2N + 24 bytes (char[] array)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

2N + 64 bytes

Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, …
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object
(for reference to enclosing class)

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, count memory (recursively) for referenced object.

http://algs4.cs.princeton.edu

Stacks and Queues

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

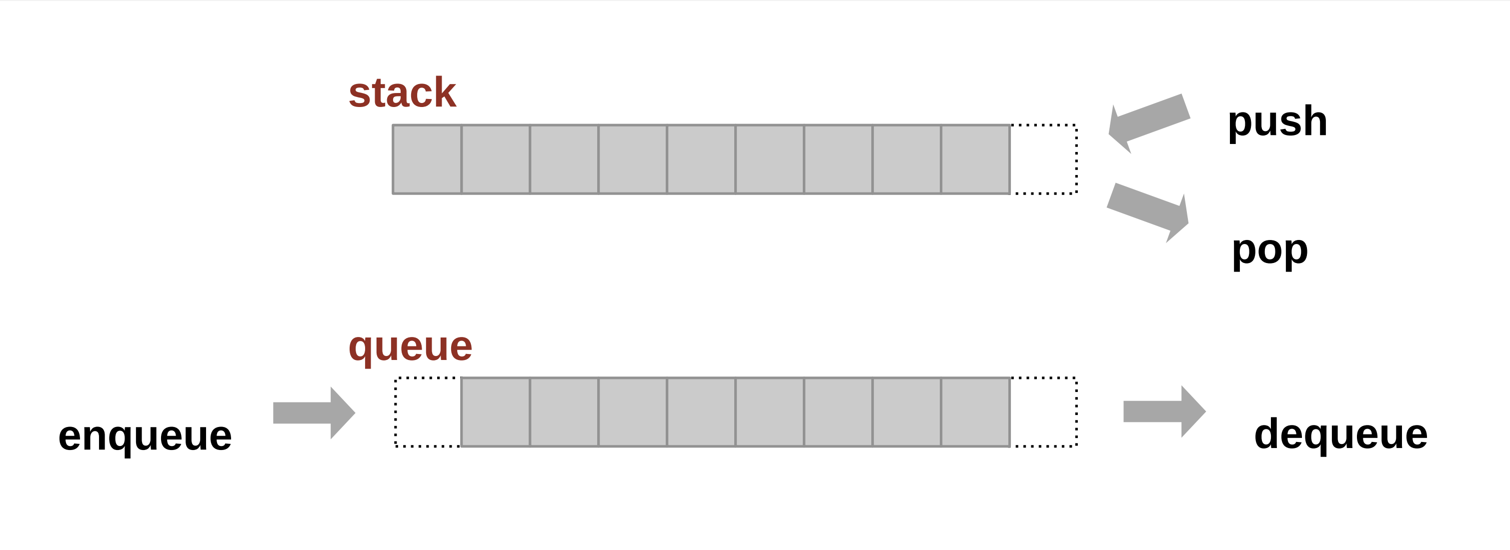
Computer Engineering Department

University of Jordan

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

http://algs4.cs.princeton.edu

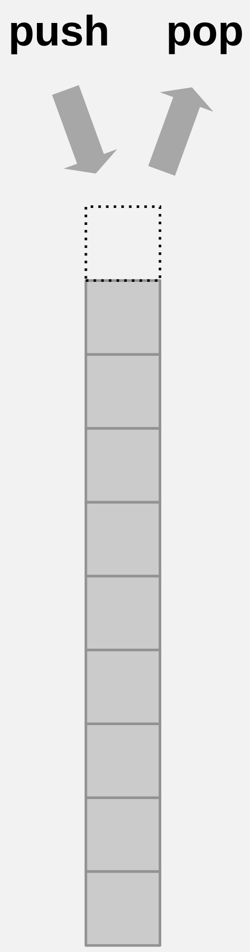
Stacks and Queues

- *stacks*
- *resizing arrays*
- *queues*
- *generics*

Stack API

Warmup API. Stack of strings data type.

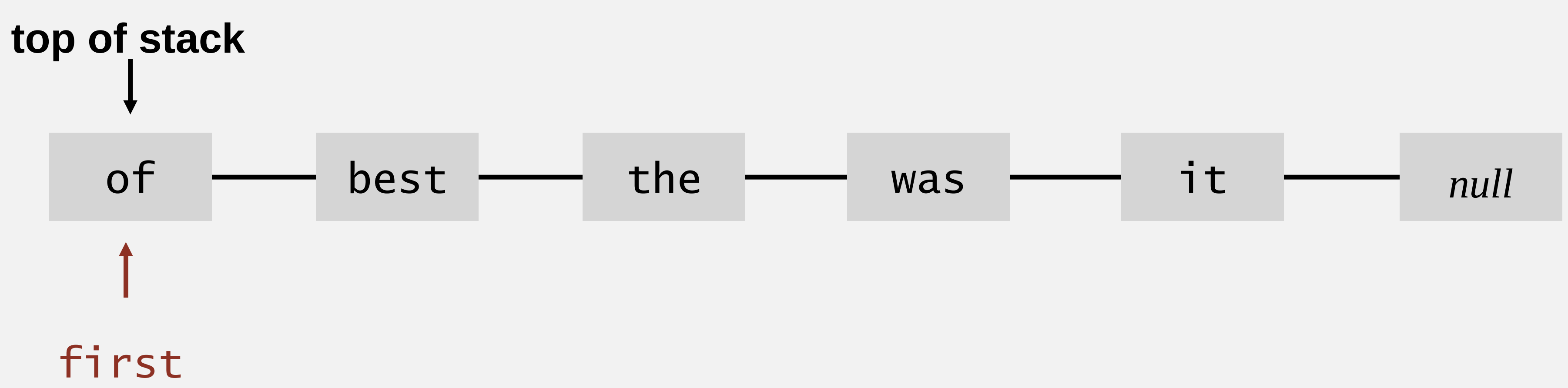
	`public class StackOfStrings`	
	`StackOfStrings()`	*create an empty stack*
`void`	`push(String item)`	*insert a new string onto stack*
`String`	`pop()`	*remove and return the string most recently added*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of strings on the stack*



Warmup client. Reverse sequence of strings from standard input.

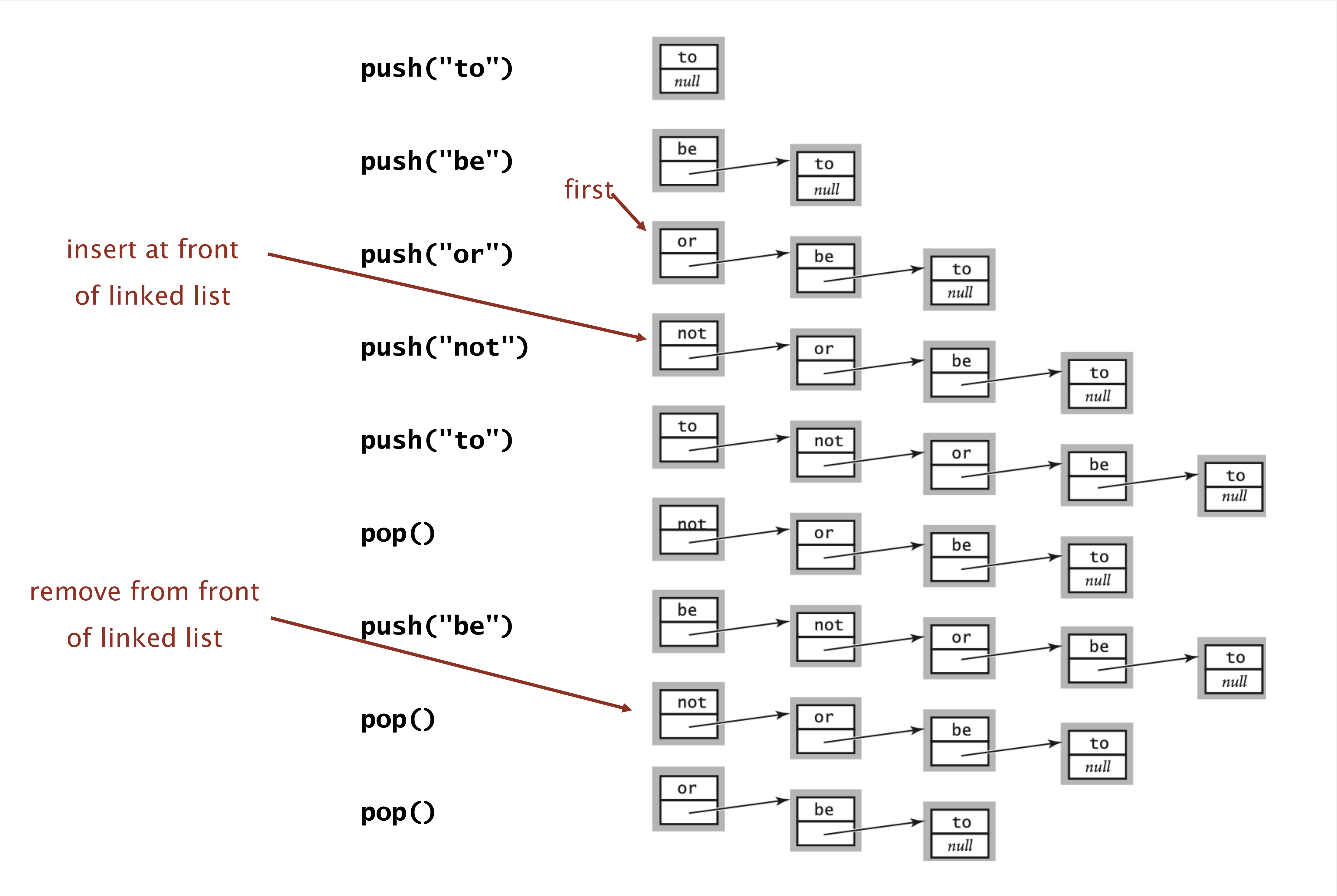
Stack: linked-list implementation

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.



Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.



Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

private inner class
(access modifiers for instance
variables don't matter)

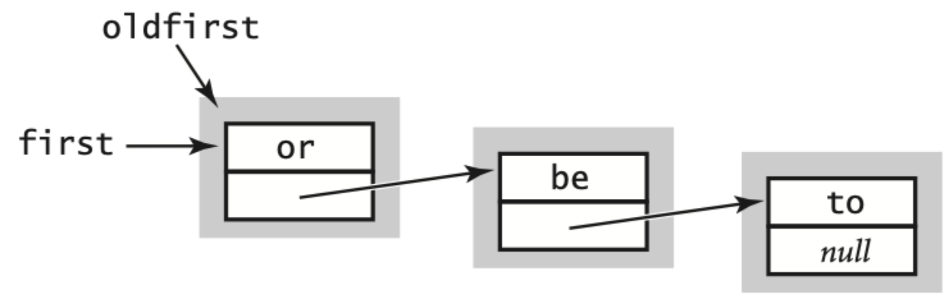
Stack push: linked-list implementation

inner class

```
private class Node
{
   String item;
   Node next;
}
```

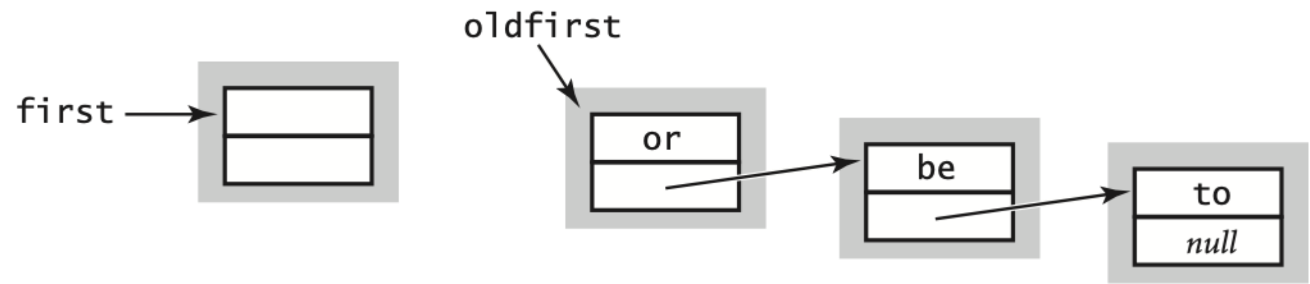
save a link to the list

```
Node oldfirst = first;
```



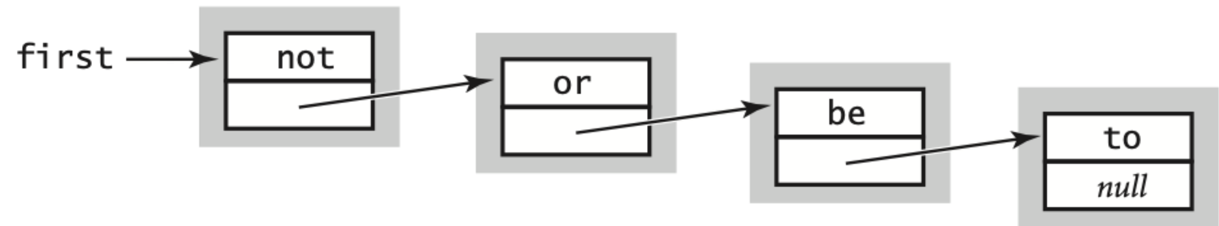
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```



Stack pop: linked-list implementation

inner class

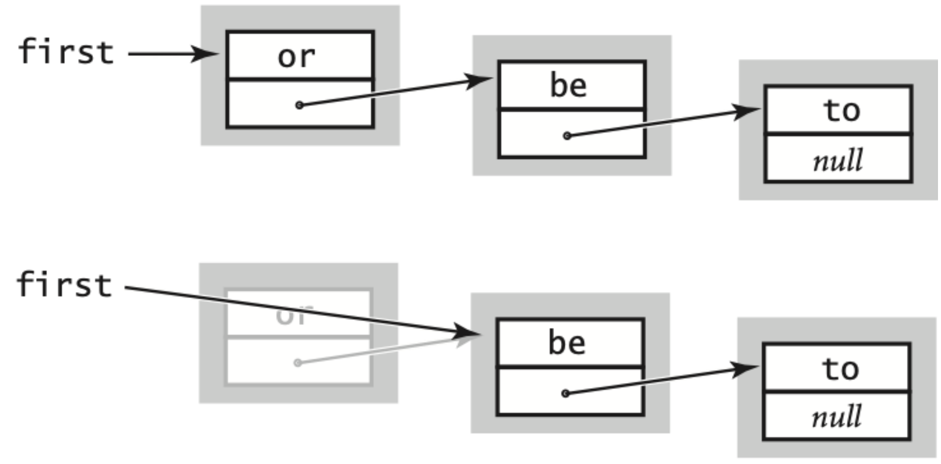
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

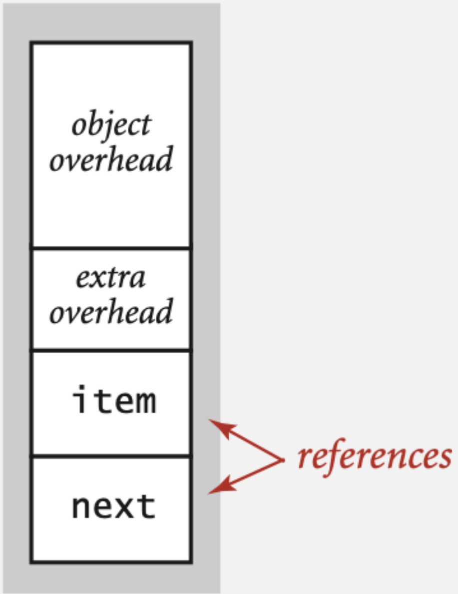
Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with N items uses $\sim 40\,N$ bytes.

inner class

```
private class Node
{
   String item;
   Node next;
}
```



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

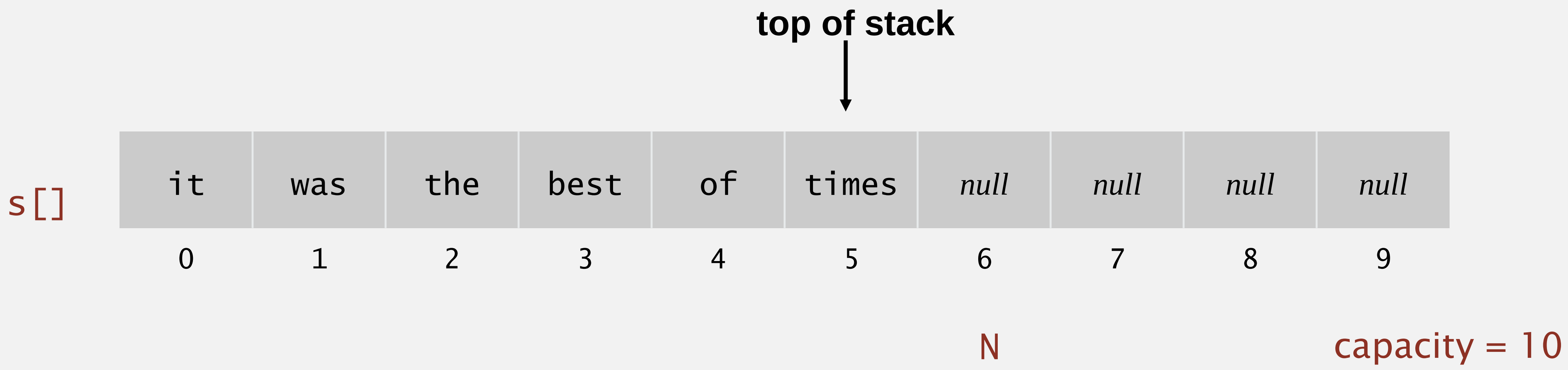
8 bytes (reference to Node)

40 bytes per stack node

Remark. This accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

Fixed-capacity stack: array implementation

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.



Defect. Stack overflows when `N` exceeds capacity. [stay tuned]

Fixed-capacity stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

a cheat
(stay tuned)

use to index into array;
then increment N

decrement N;
then use to index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

```
public String pop()
{  return s[--N];  }
```

loitering

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

this version avoids "loitering": garbage collector can reclaim memory for an object only if no outstanding references

http://algs4.cs.princeton.edu

Stacks and Queues

- *stacks*
- *resizing arrays*
- *queues*
- *generics*

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by `1`.
- `pop()`: decrease size of array `s[]` by `1`.

Too expensive.

infeasible for large N

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first N items = $N + (2 + 4 + \ldots + 2(N-1)) \sim N^2$.

1 array access per push

2(k–1) array accesses to expand to size k (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?

"repeated doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
public ResizingArrayStackOfStrings()
{   s = new String[1];   }
public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}
private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}
```

Array accesses to insert first $N = 2^i$ items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

1 array access per push

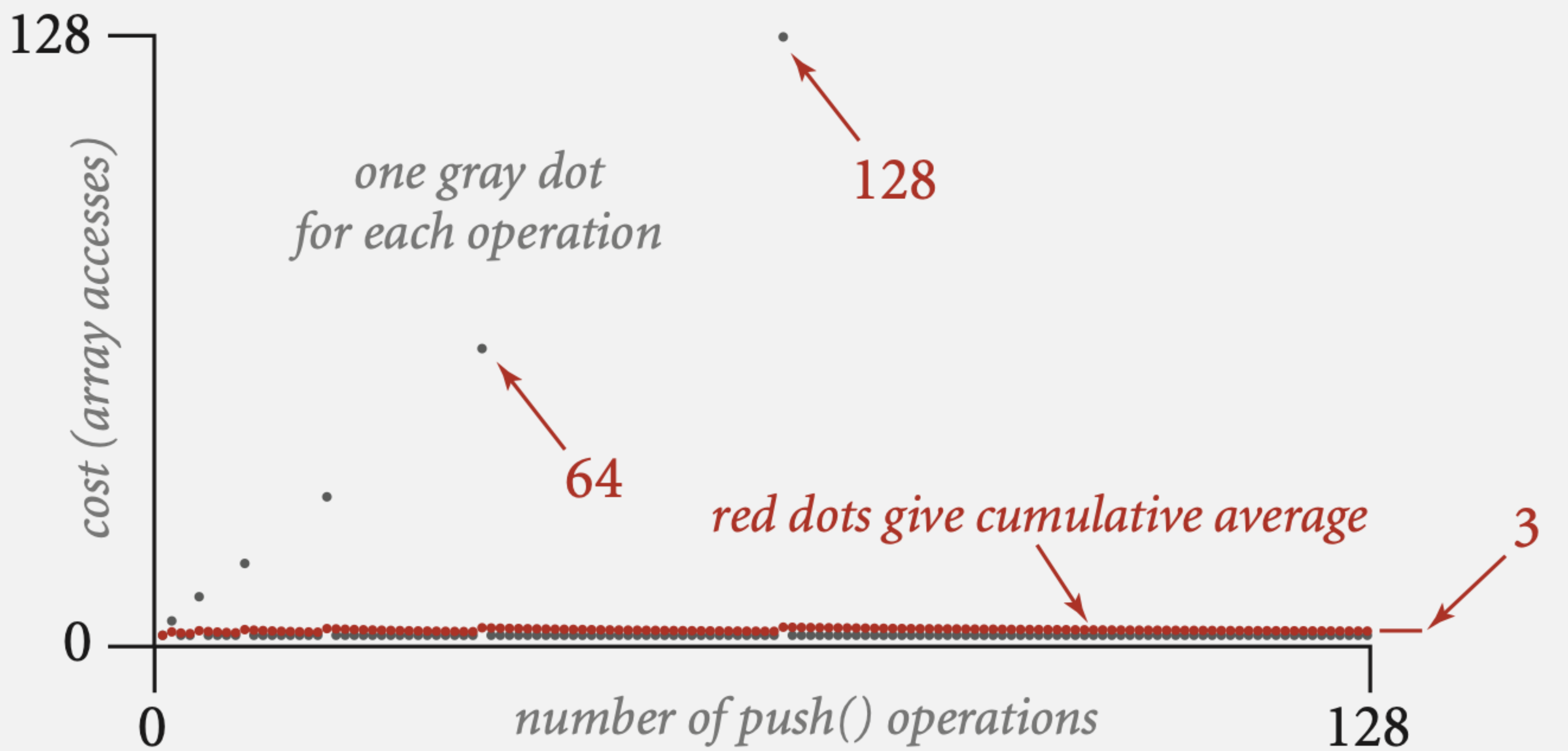
k array accesses to double to size k (ignoring cost to create new array)

Stack: amortized cost of adding to a stack

Cost of inserting first N items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

1 array access per push

k array accesses to double to size k (ignoring cost to create new array)



Stack: resizing-array implementation

Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-half full.

Too expensive in worst case.

- Consider push-pop-push-pop-… sequence when array is full.
- Each operation takes time proportional to N.

N = 5	to	be	or	not	to	*null*	*null*	*null*
N = 4	to	be	or	not				
N = 5	to	be	or	not	to	*null*	*null*	*null*
N = 4	to	be	or	not				

Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-quarter full.

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length/2);
   return item;
}
```

Invariant. Array is between 25% and 100% full.

Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.

	best	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

doubling and halving operations

order of growth of running time for resizing stack with N items

Stack resizing-array implementation: memory usage

Proposition. Uses between $\sim 8\,N$ and $\sim 32\,N$ bytes to represent a stack with N items.

- $\sim 8\,N$ when full.
- $\sim 32\,N$ when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
   private String[] s;        8 bytes × array size
   private int N = 0;
   …
}
```

Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).

Stack implementations: resizing array vs. linked list

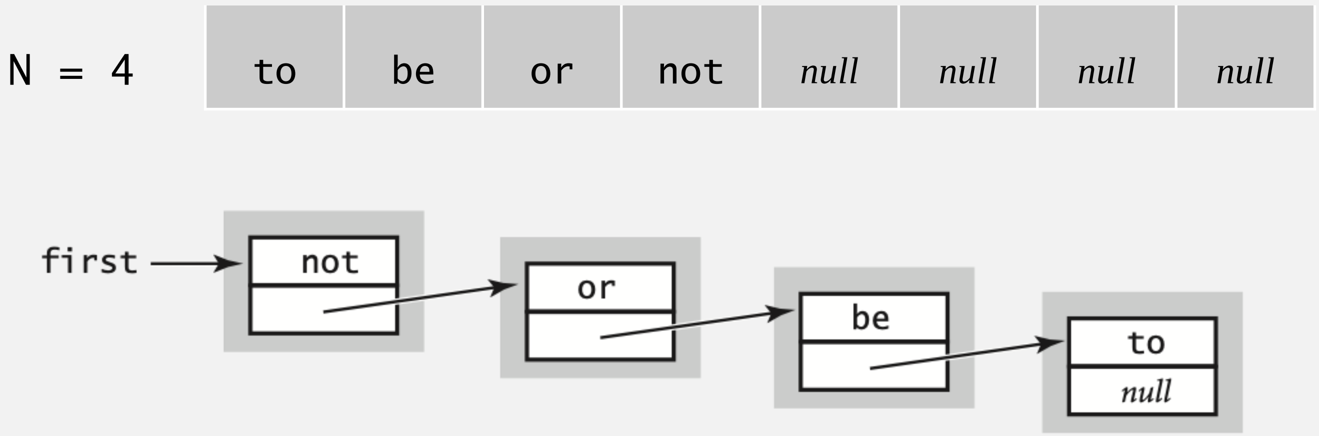
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant amortized time.
- Less wasted space.



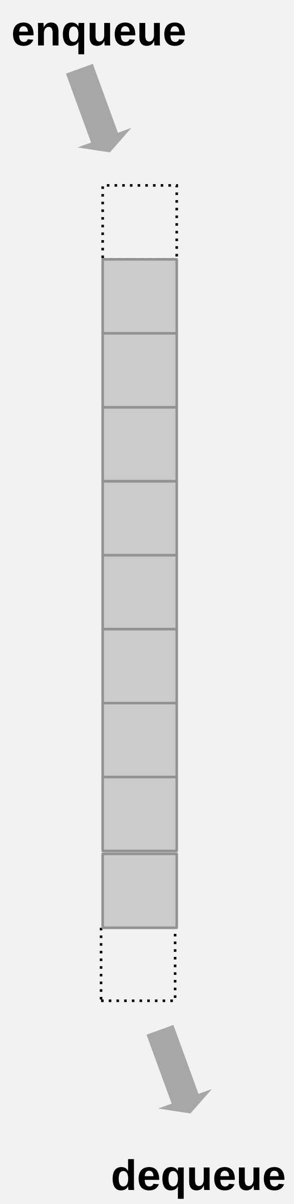
http://algs4.cs.princeton.edu

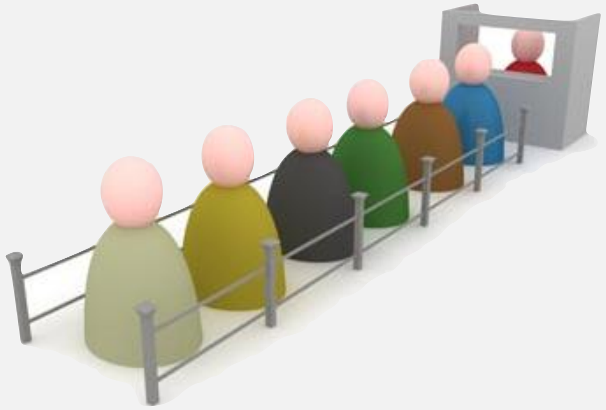
Stacks and Queues

- *stacks*
- *resizing arrays*
- ***queues***
- *generics*

Queue API

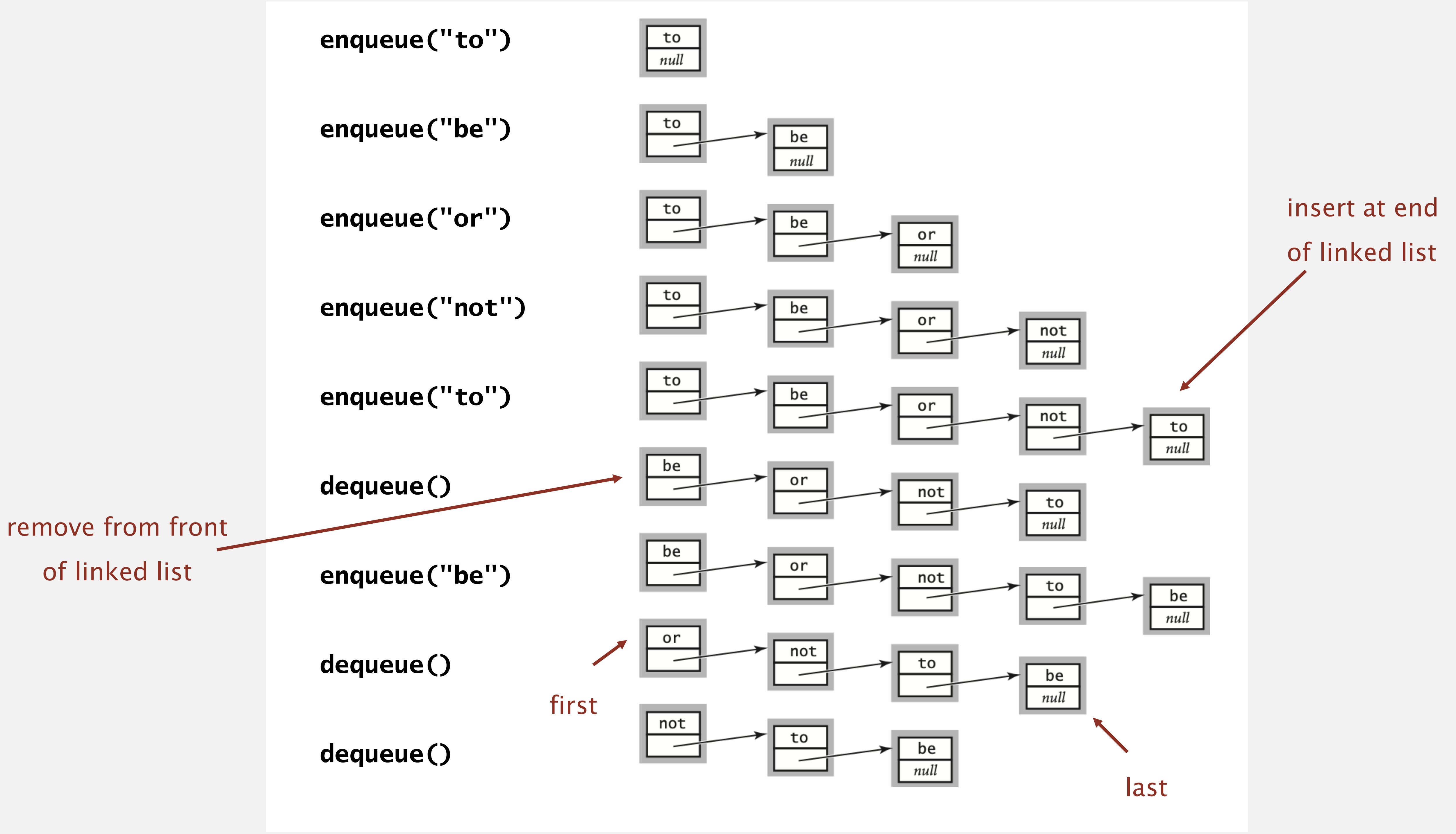
public class QueueOfStrings		
	QueueOfStrings()	*create an empty queue*
void	enqueue(String item)	*insert a new string onto queue*
String	dequeue()	*remove and return the string least recently added*
boolean	isEmpty()	*is the queue empty?*
int	size()	*number of strings on the queue*





Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list;
remove from front; insert at end.



Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in LinkedStackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

special cases for empty queue

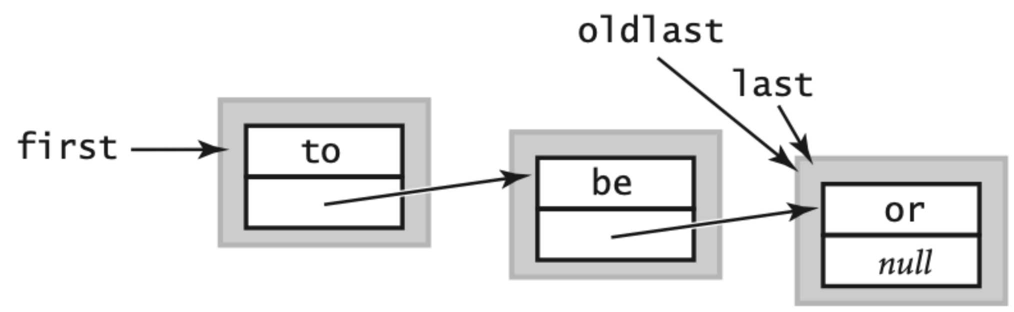
Queue enqueue: linked-list implementation

inner class

```
private class Node
{
   String item;
   Node next;
}
```

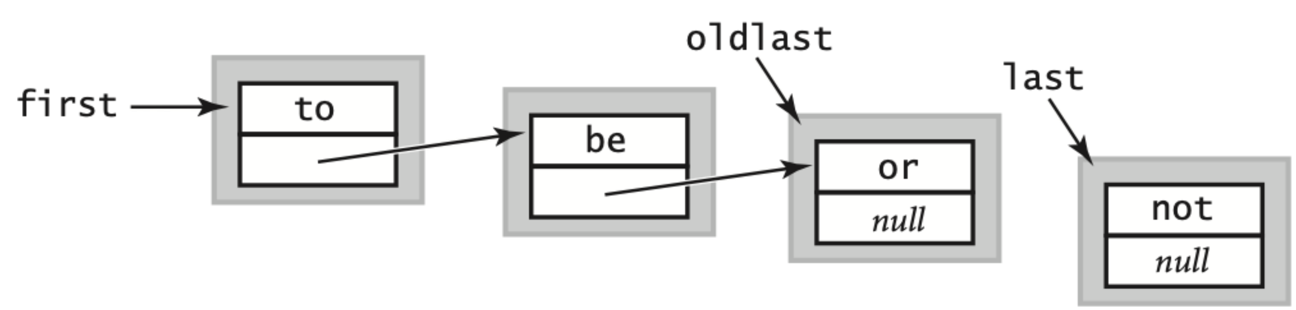
save a link to the last node

```
Node oldlast = last;
```



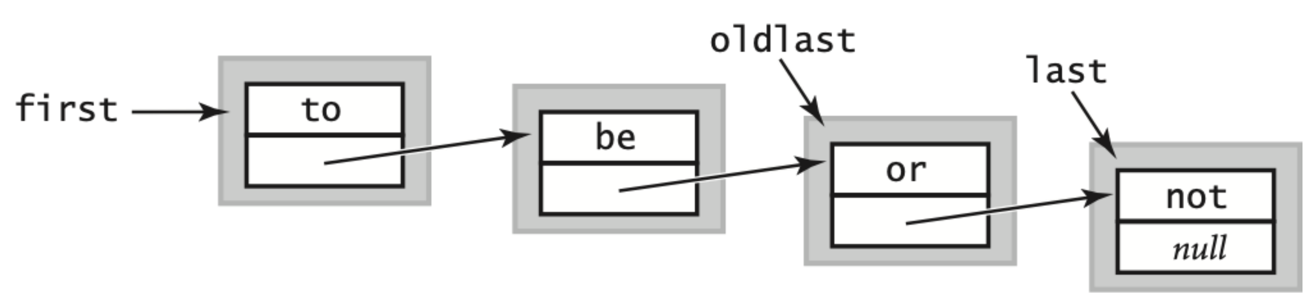
create a new node for the end

```
last = new Node();
last.item = "not";
```



link the new node to the end of the list

```
oldlast.next = last;
```



Queue dequeue: linked-list implementation

inner class

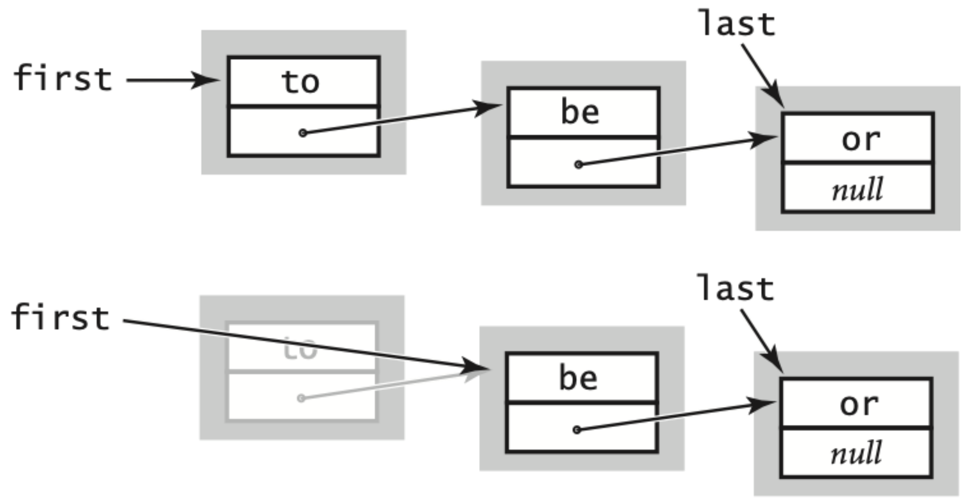
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



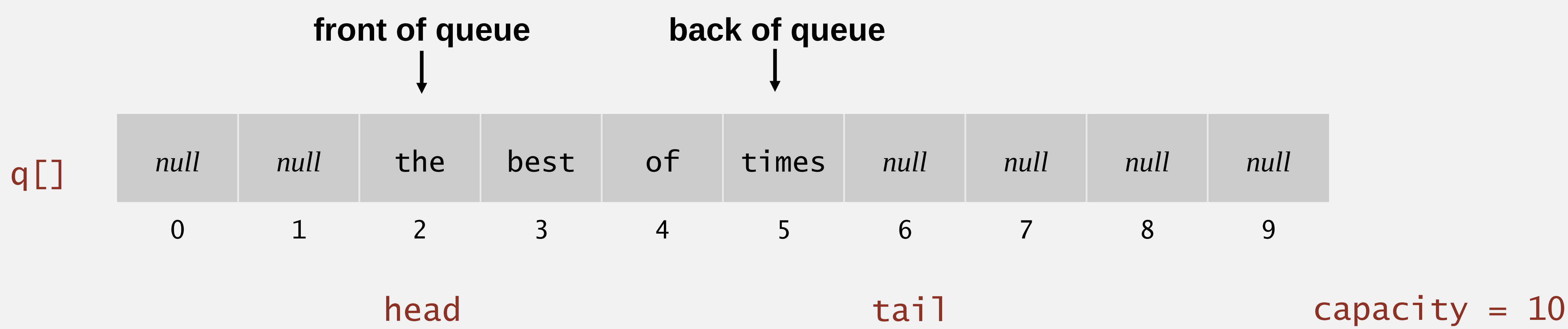
return saved item

```
return item;
```

Remark. Identical code to linked-list stack pop().

Queue: resizing-array implementation

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.
- Add resizing array.



Q. How to resize?

http://algs4.cs.princeton.edu

Stacks and Queues

- stacks
- resizing arrays
- queues
- **generics**

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ...

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5.

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

Attempt 2. Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

type parameter

```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

compile-time error

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
```

generic type name

Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public ..StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

the way it should be

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int N = 0;

   public FixedCapacityStack(int
capacity)
   {  s = new Item[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public ..StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

the way it is

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int N = 0;

   public FixedCapacityStack(int capacity)
   {  s = (Item[]) new Object[capacity];
}

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the ugly cast

Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);          // s.push(Integer.valueOf(17));
int a = s.pop();     // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.

Java collections library

List interface. `java.util.List` is API for a sequence of items.

`public interface List<Item> implements Iterable<Item>`		
	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*append item to the end*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
`Iterator<Item>`	`iterator()`	*iterator over all items in the list*
`...`		

Implementations. `java.util.ArrayList` uses resizing array;
`java.util.LinkedList` uses linked list.

caveat: only some
operations are efficient

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.

Java 1.3 bug report (June 27, 2001)

```
The iterator method on java.util.Stack iterates through a Stack from
the bottom up. One would think that it should iterate as if it were
popping off the top of the Stack.
```

status (closed, will not fix)

```
It was an incorrect design decision to have Stack extend Vector ("is-a"
rather than "has-a"). We sympathize with the submitter but cannot fix
this because of compatibility.
```

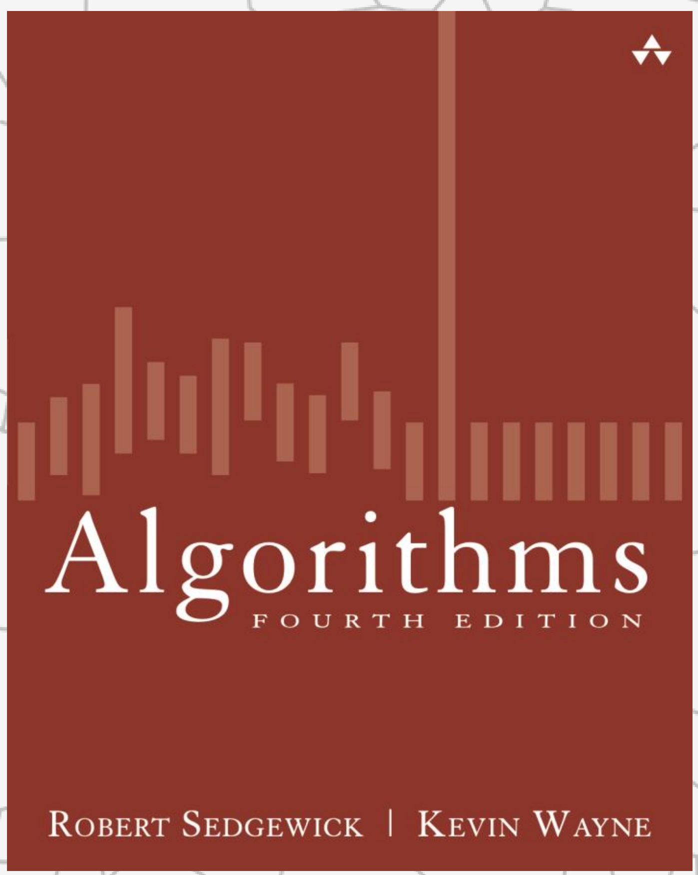
Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.

`java.util.Queue`. An interface, not an implementation of a queue.

ELEMENTARY SORTS



http://algs4.cs.princeton.edu

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

Sorting problem

Ex. Student records in a university.

item →

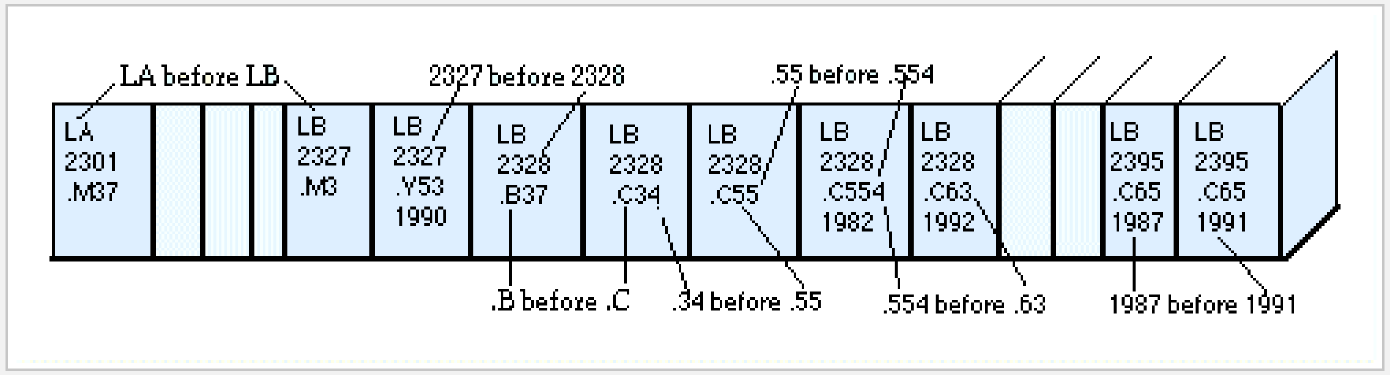
key →

Chen	3	A	991-878-4944	308 Blair
Rohde	2	A	232-343-5555	343 Forbes
Gazsi	4	B	766-093-9873	101 Brown
Furia	1	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman

Sort. Rearrange array of N items into ascending order.

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

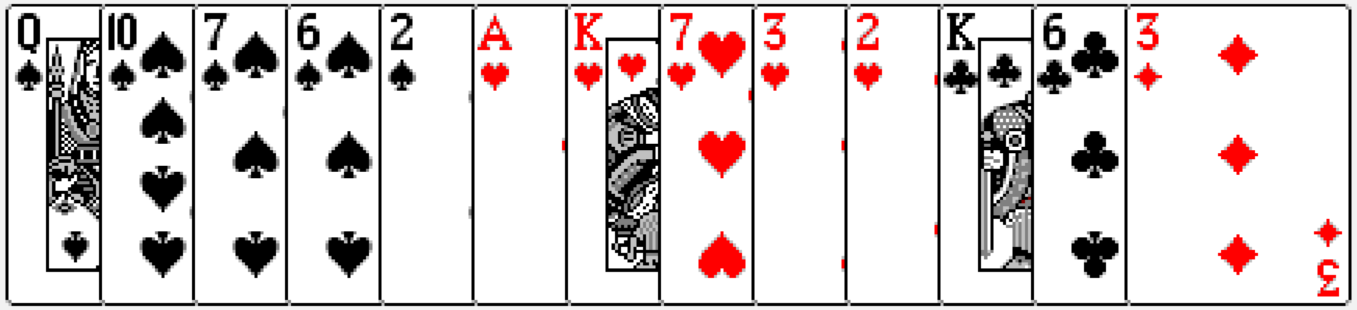
Sorting applications



Library of Congress numbers

FedEx packages

contacts



playing cards

Total order

Goal. Sort any type of data (for which sorting is well defined).

A total order is a binary relation $\leq$ that satisfies:

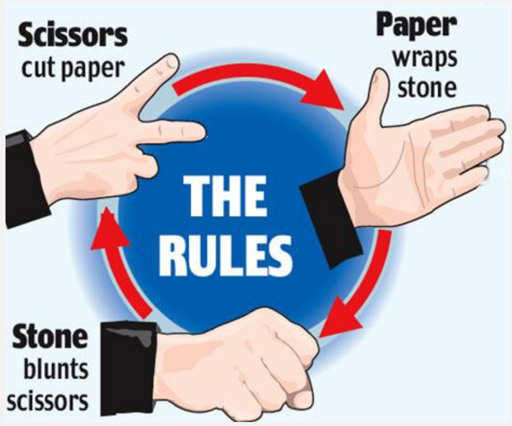
- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Ex.

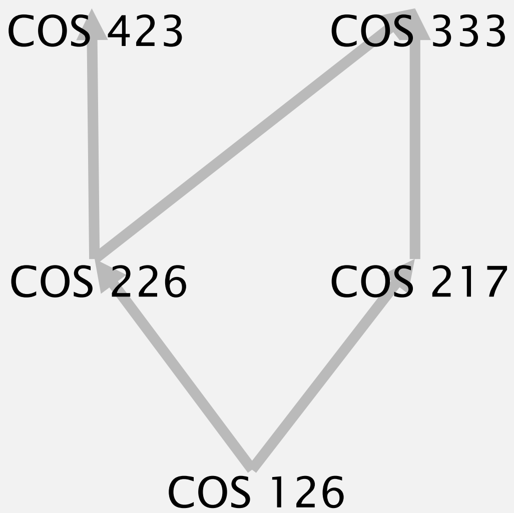
- Standard order for natural and real numbers.
- Chronological order for dates or times.
- Alphabetical order for strings.

No transitivity. Rock-paper-scissors.

No totality. PU course prerequisites.



violates transitivity



violates totality

Comparable interface

Comparable interface: sort using a type's natural order.

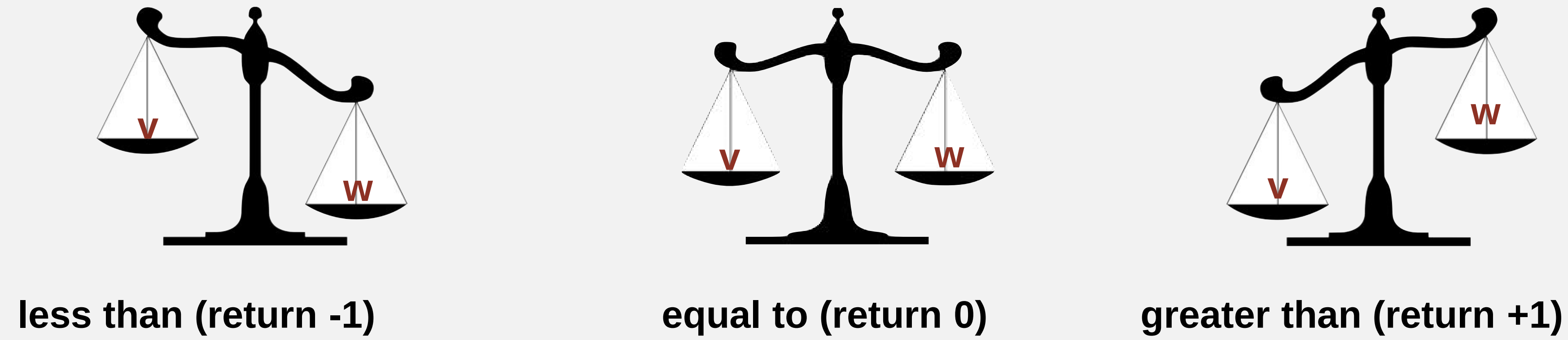
```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }
   …
   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
```

natural order

Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

- Defines a total order.
- Returns a negative integer, zero, or positive integer if v is less than, equal to, or greater than w, respectively.
- Throws an exception if incompatible types (or either is `null`).



Built-in comparable types. `Integer`, `Double`, `String`, `Date`, `File`, ...

User-defined comparable types. Implement the `Comparable` interface.

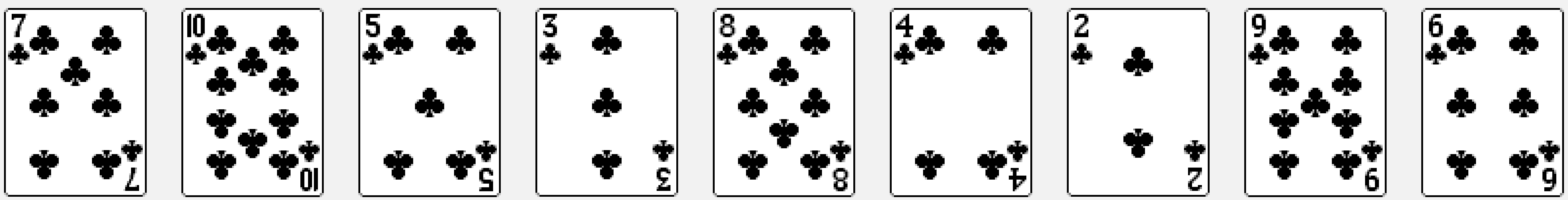
http://algs4.cs.princeton.edu

Elementary Sorts

- *selection sort*
- *insertion sort*
- *shuffling*

Selection sort demo

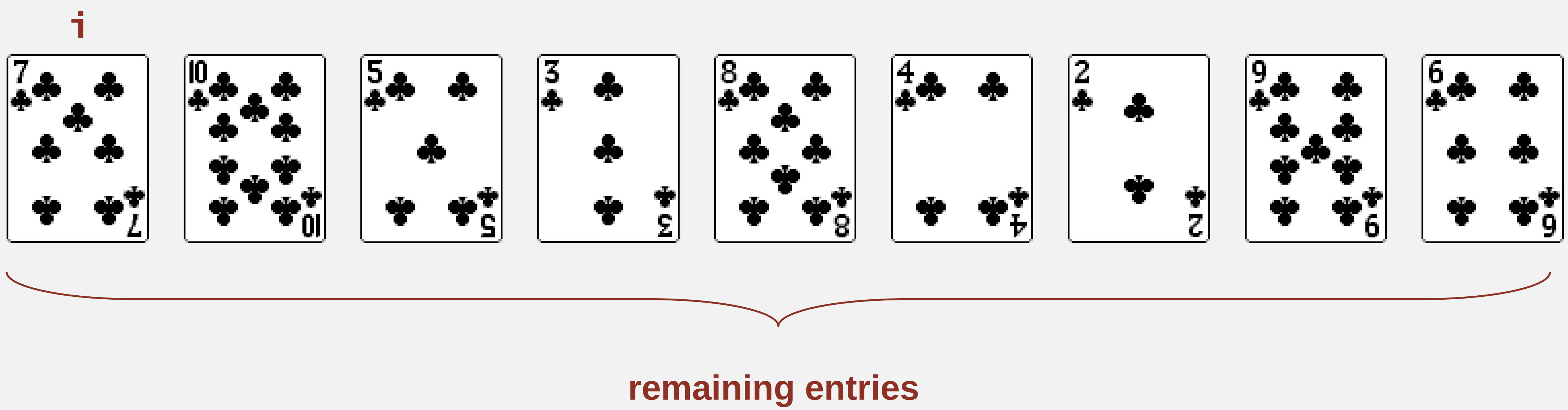
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



initial

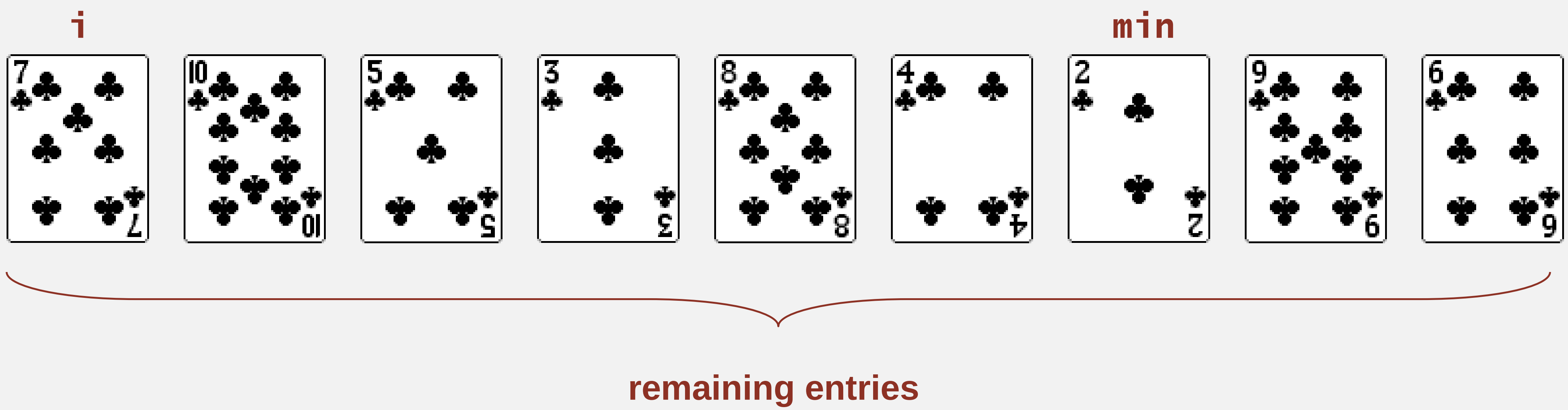
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



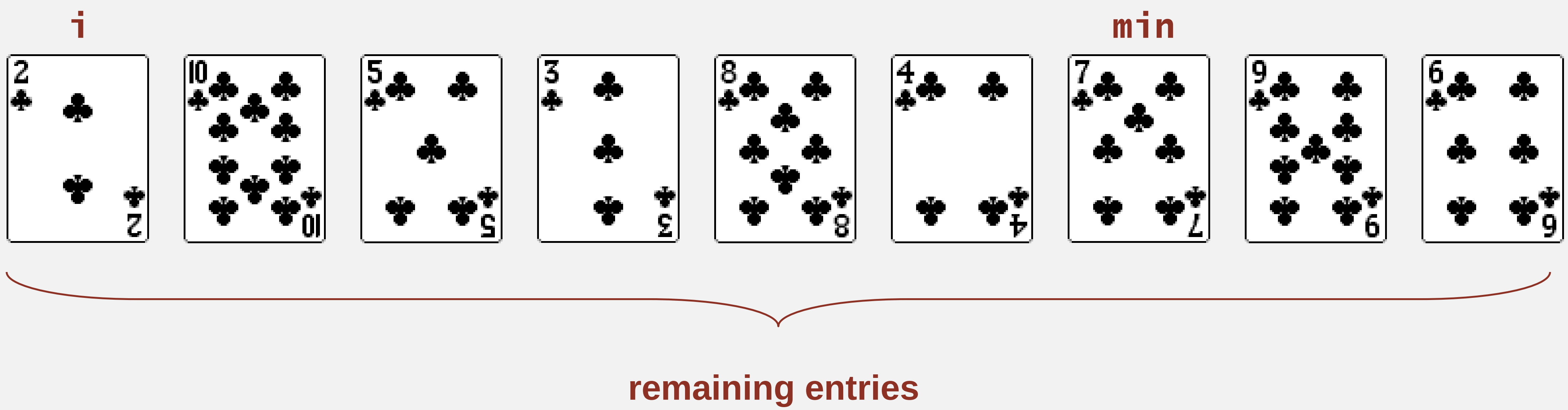
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



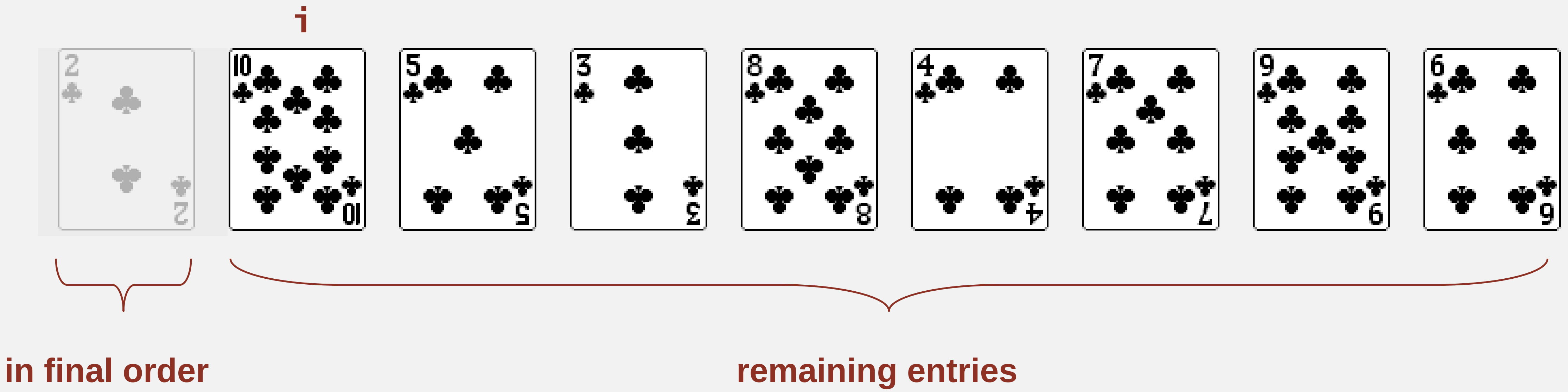
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



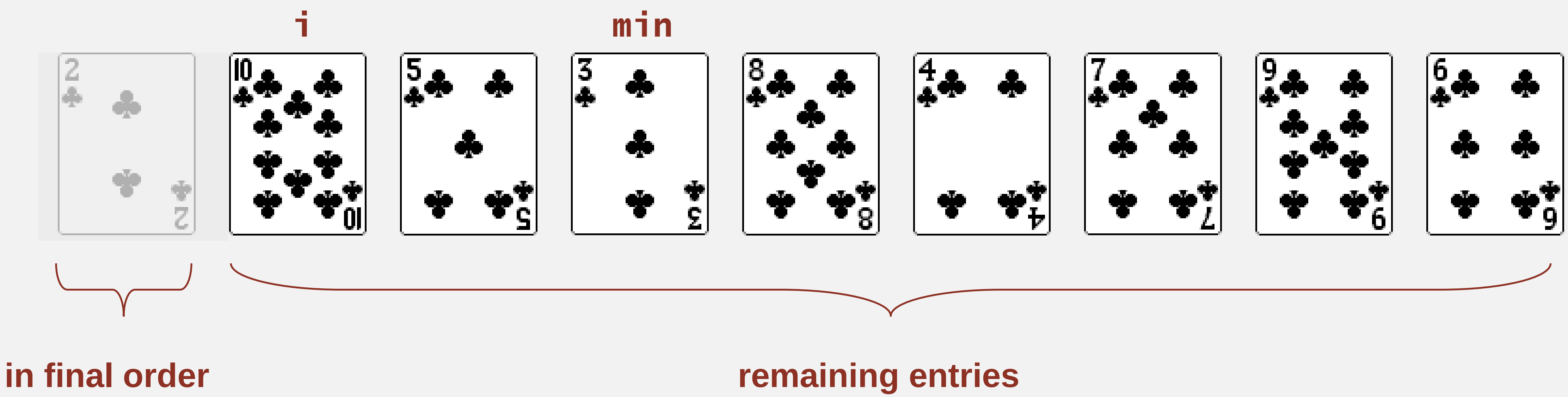
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



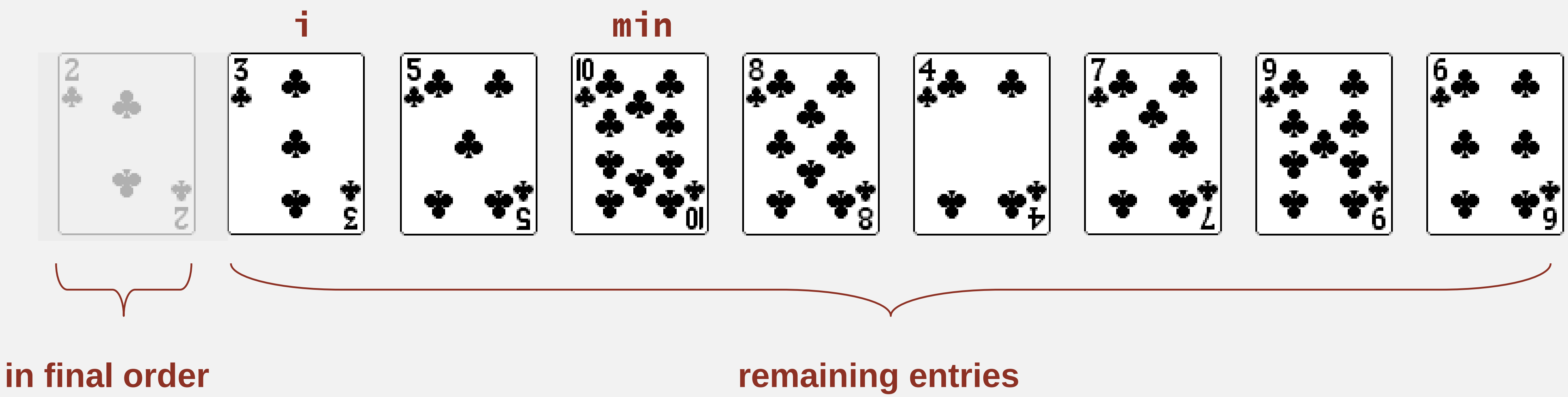
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



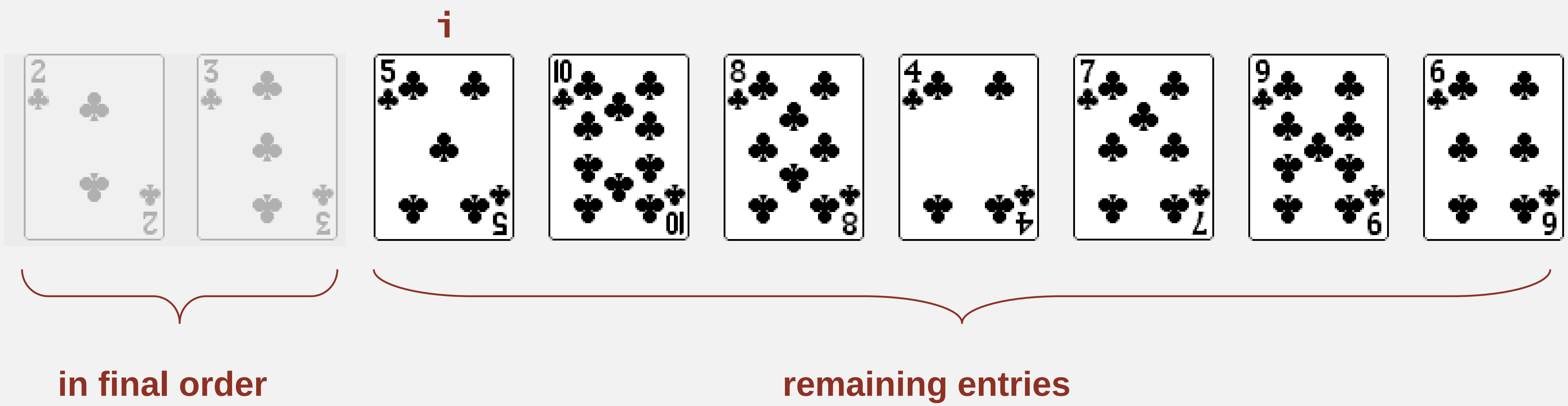
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



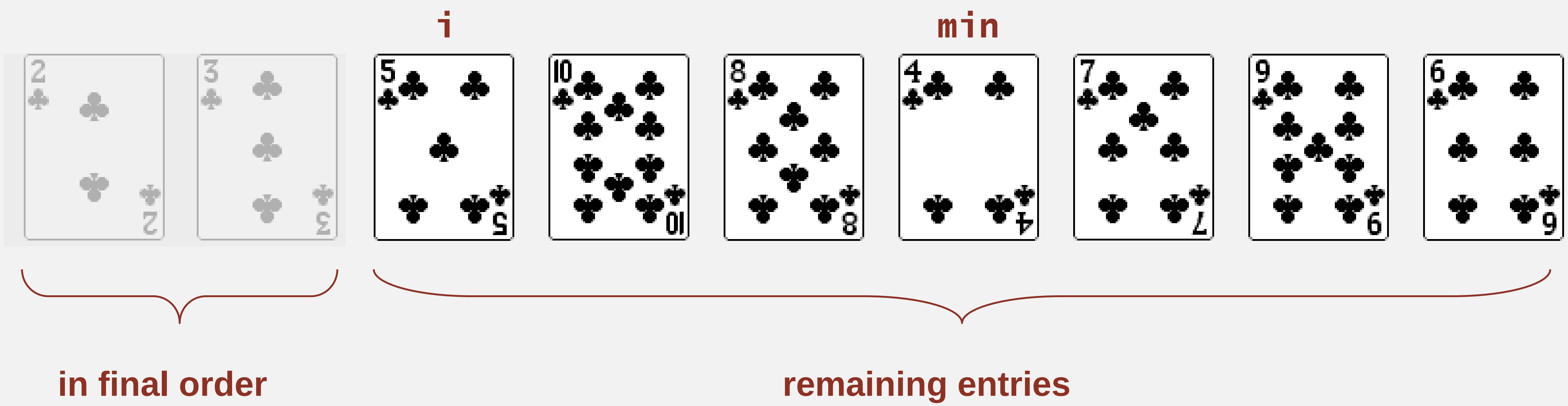
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



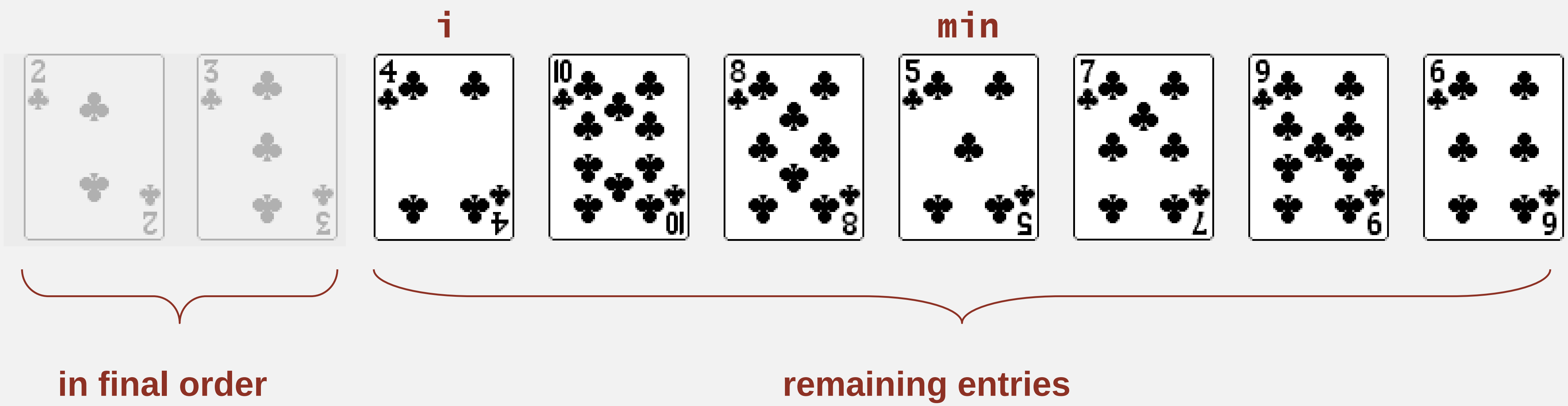
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



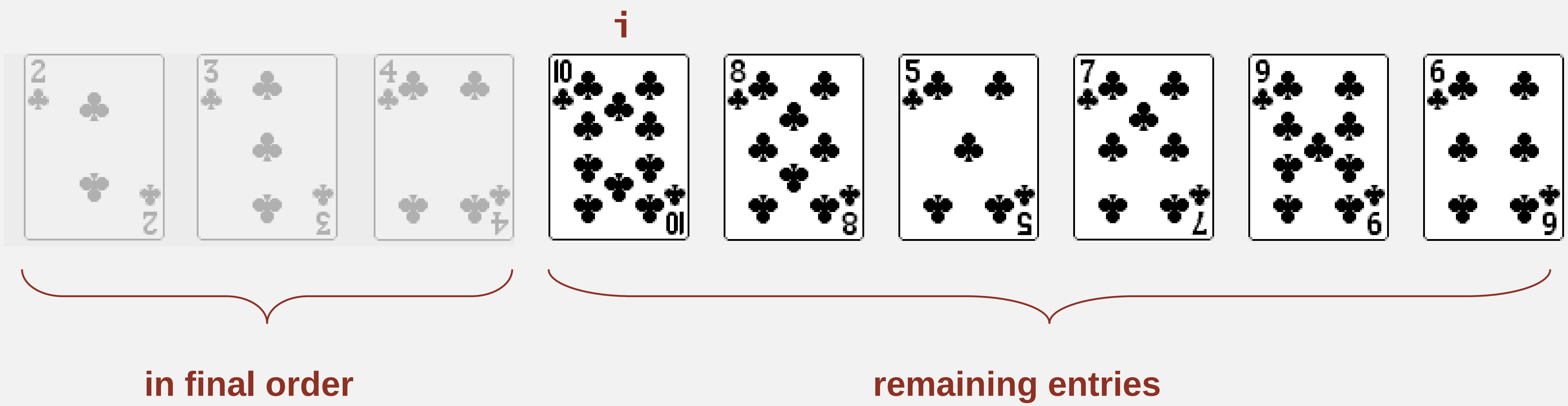
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



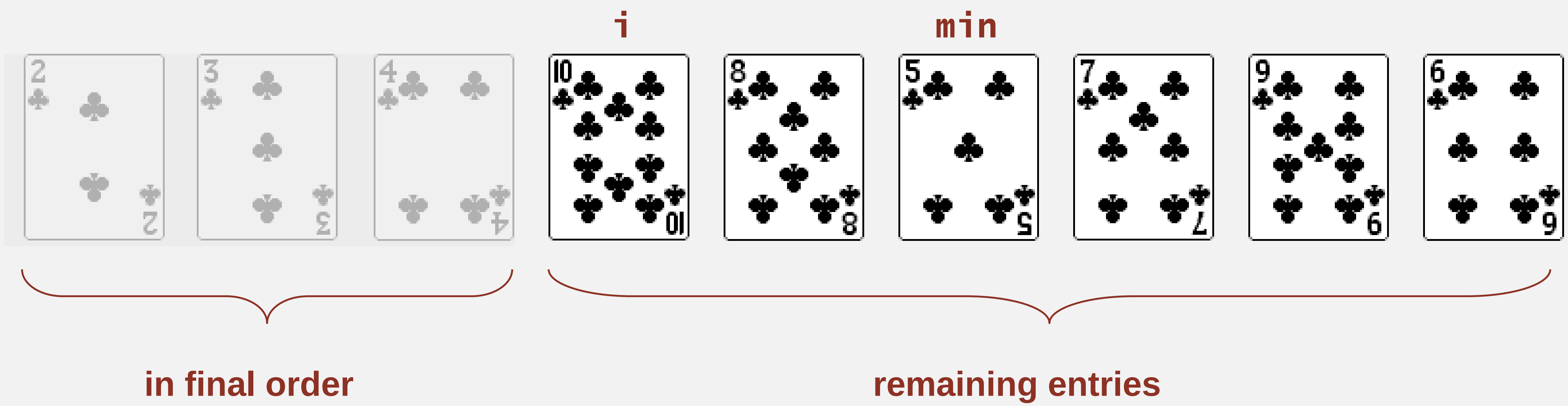
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



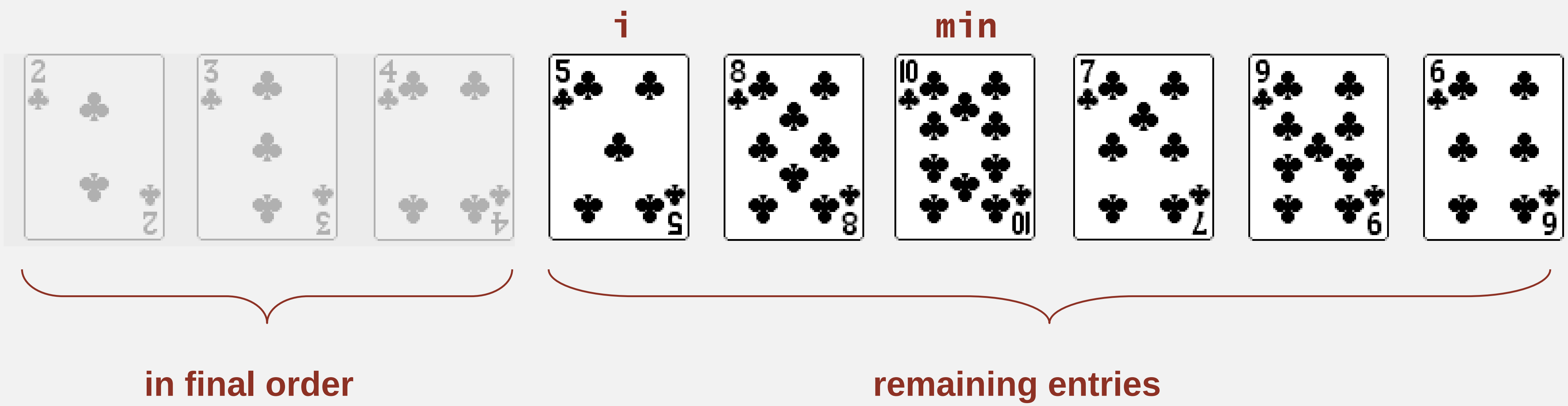
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



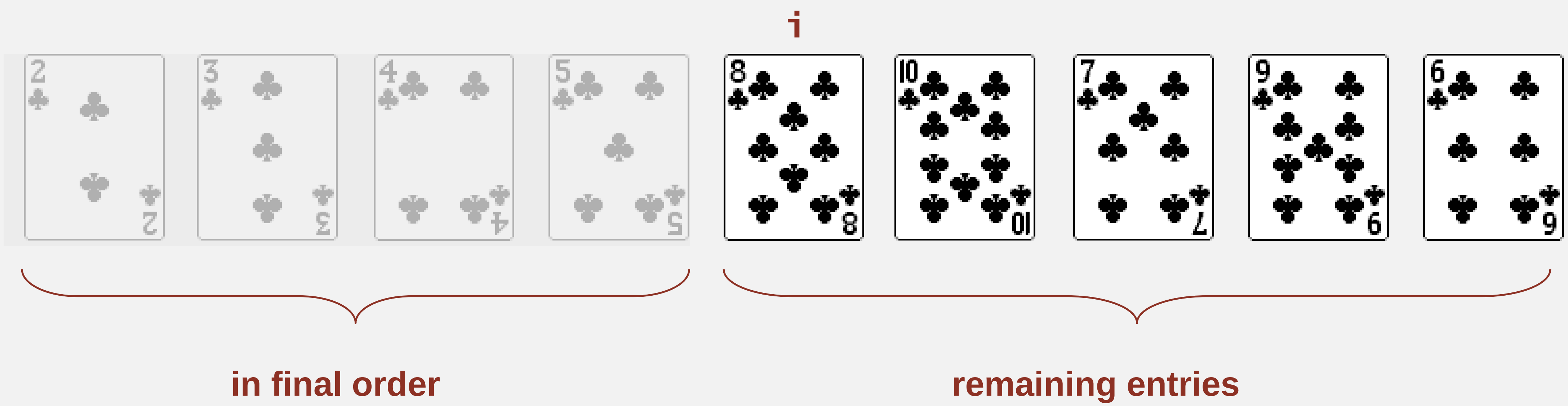
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



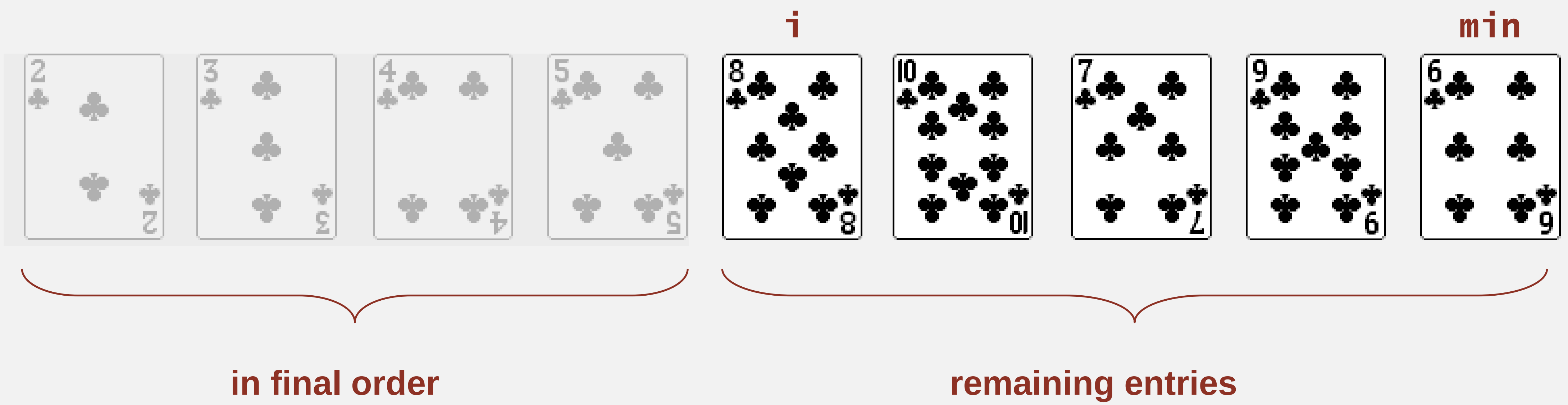
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



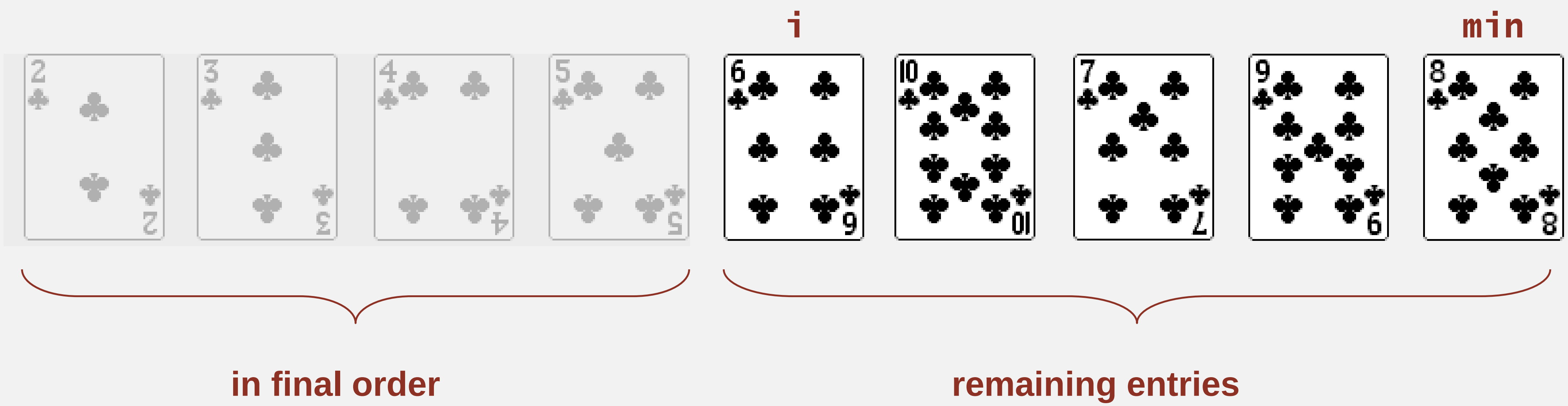
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



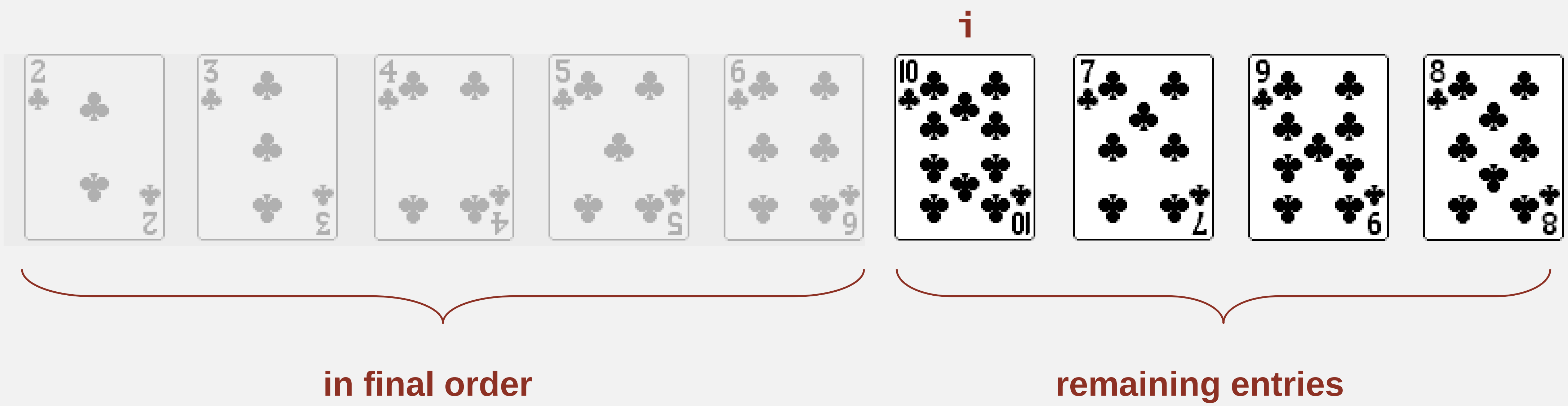
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



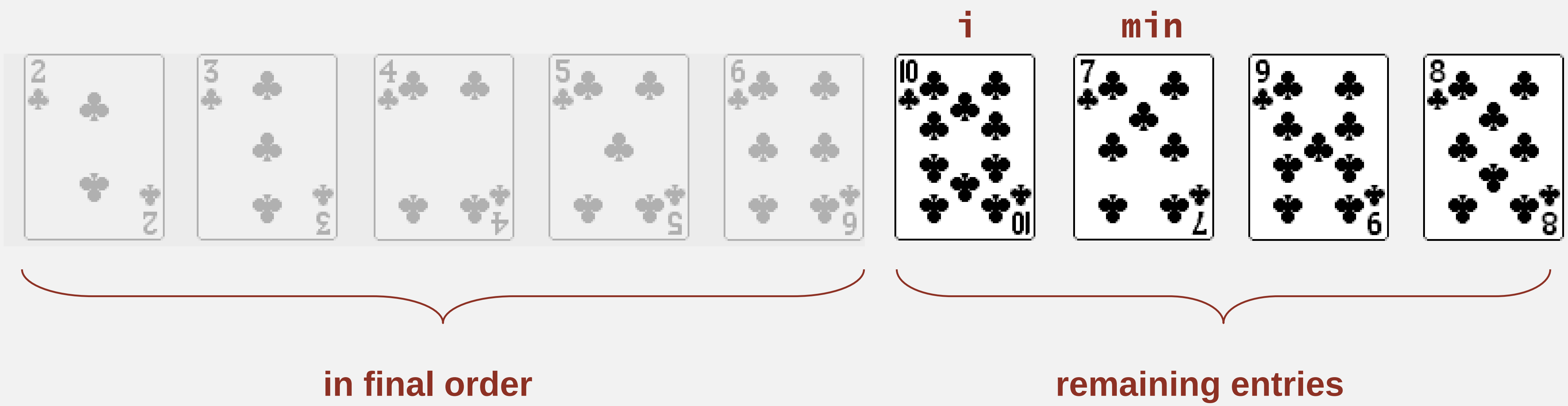
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



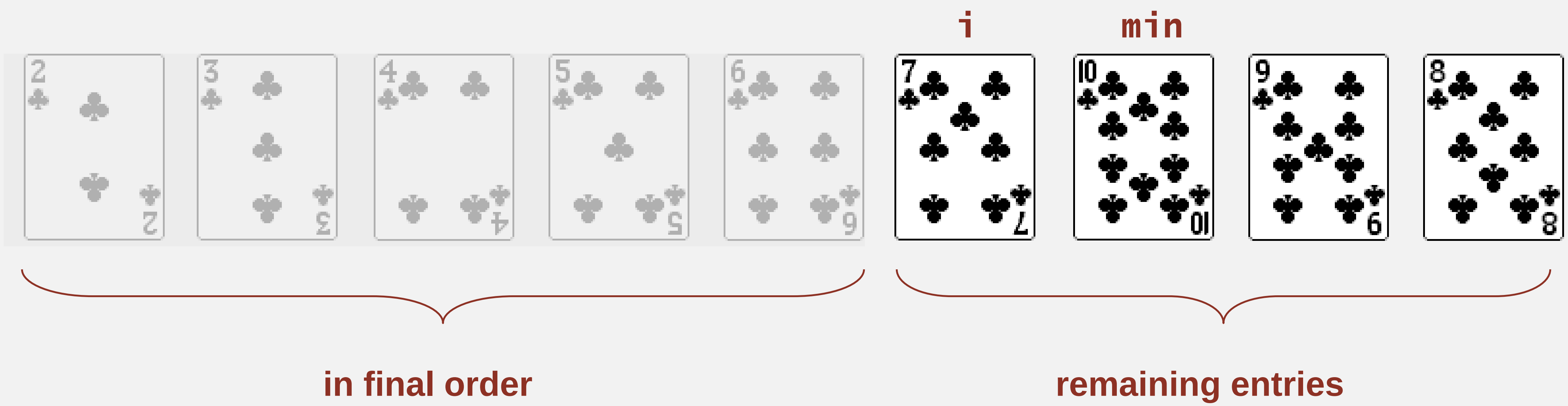
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



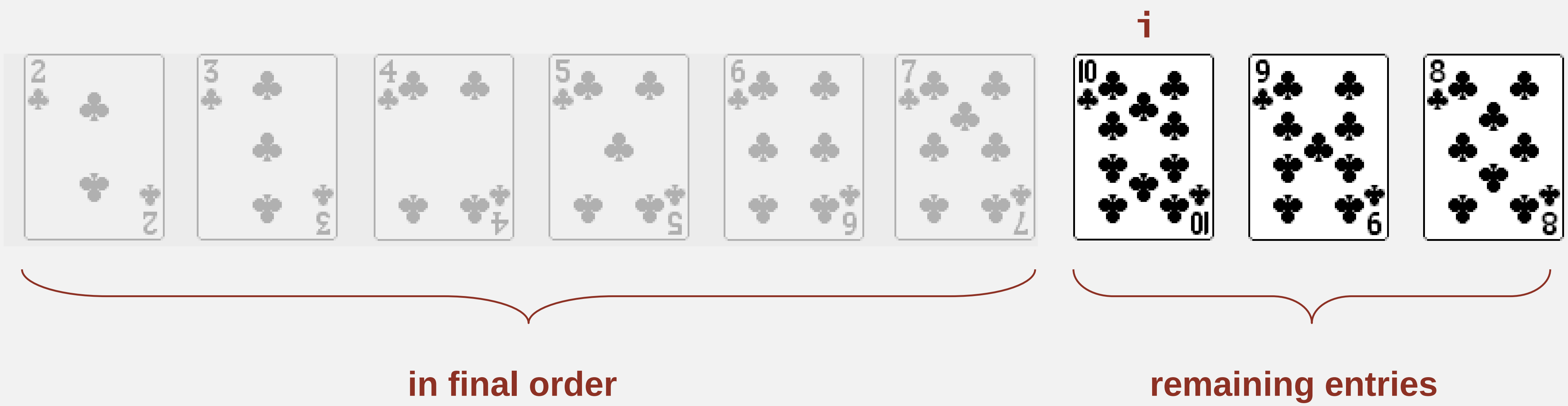
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



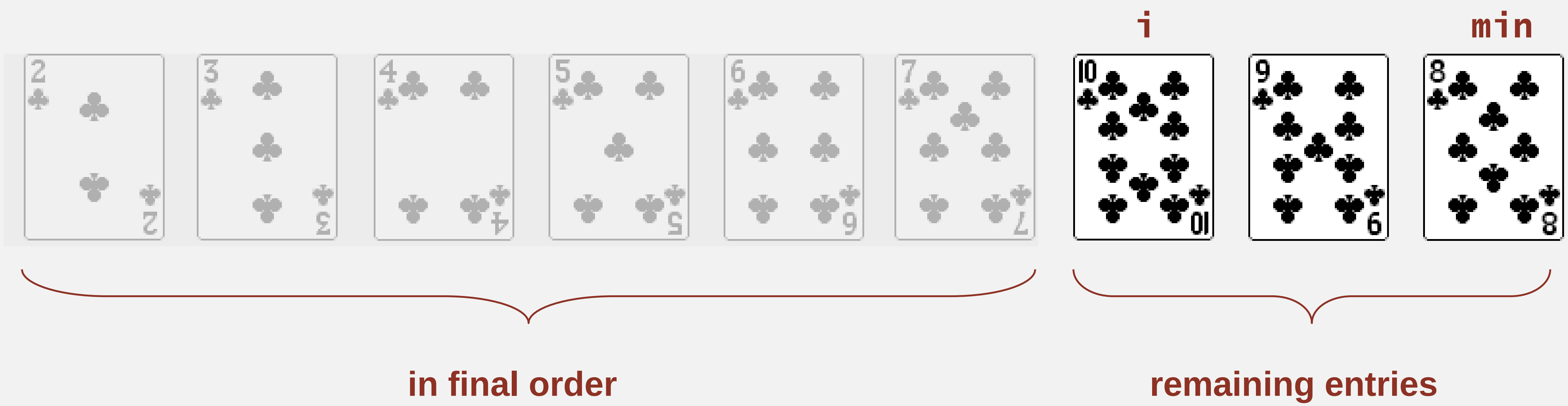
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



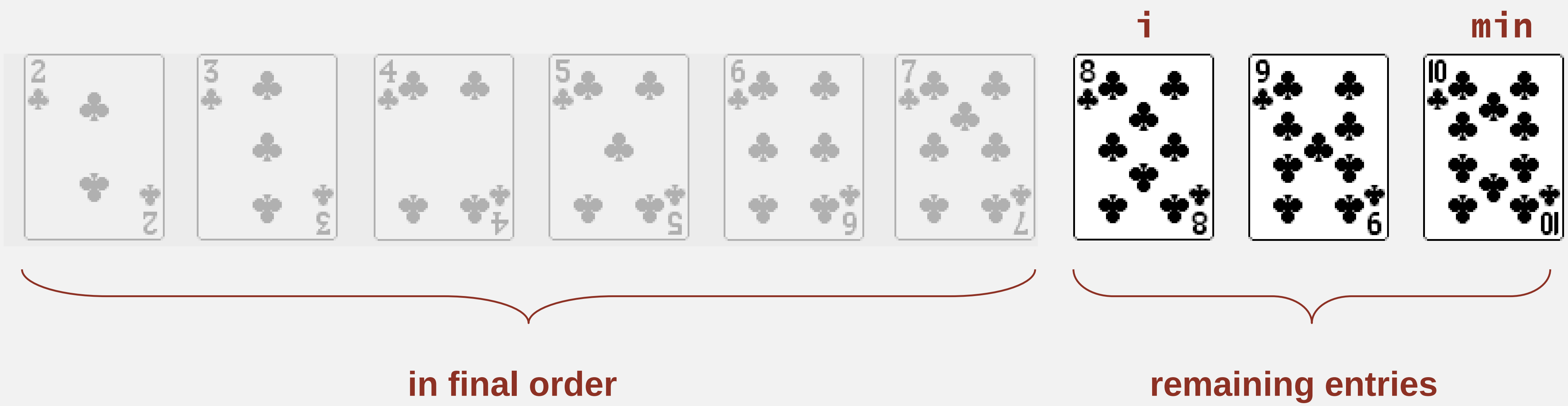
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



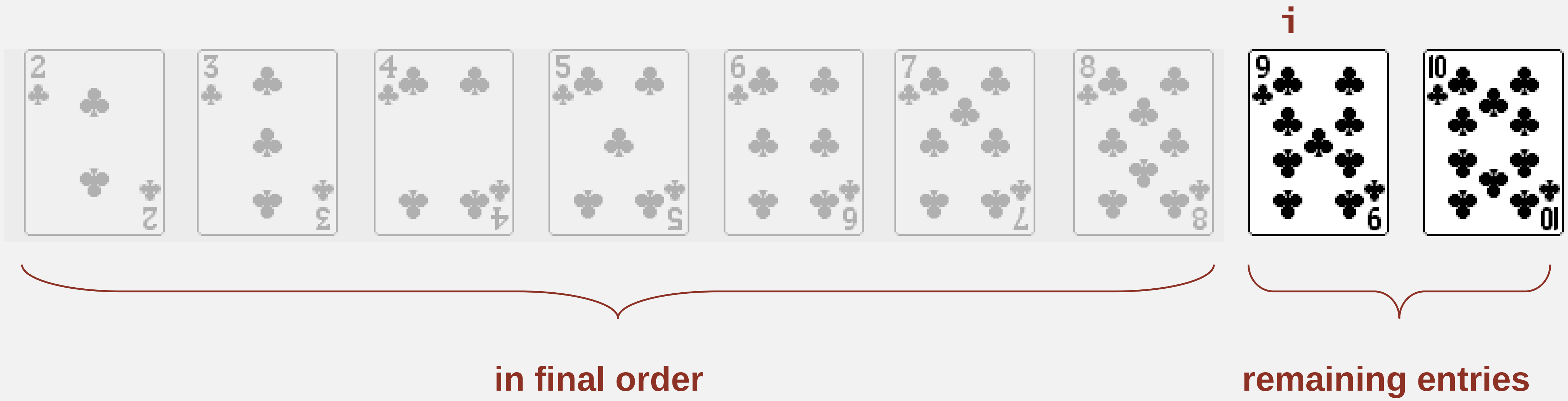
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



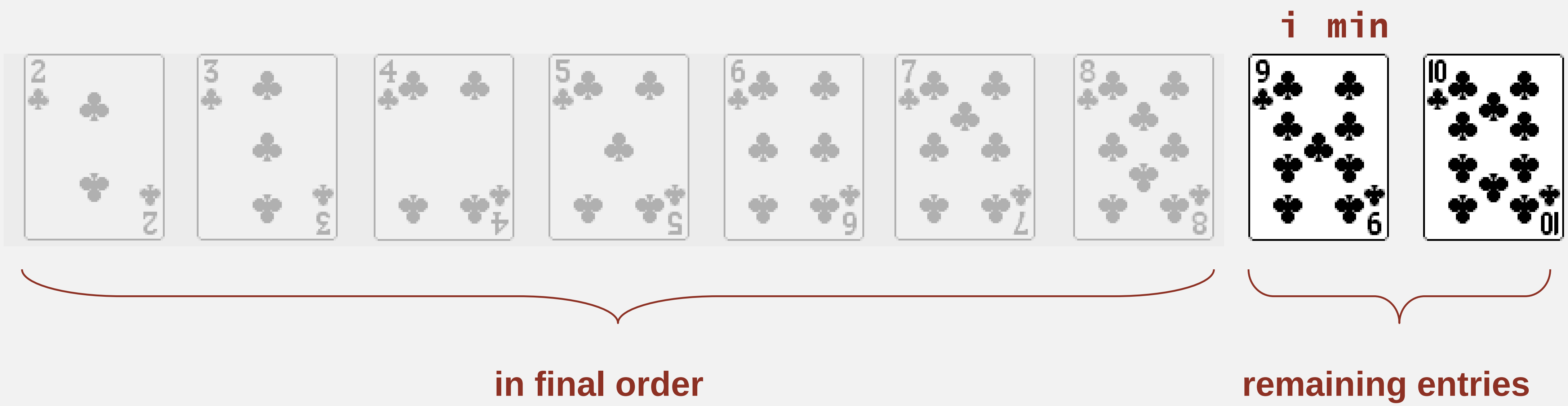
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



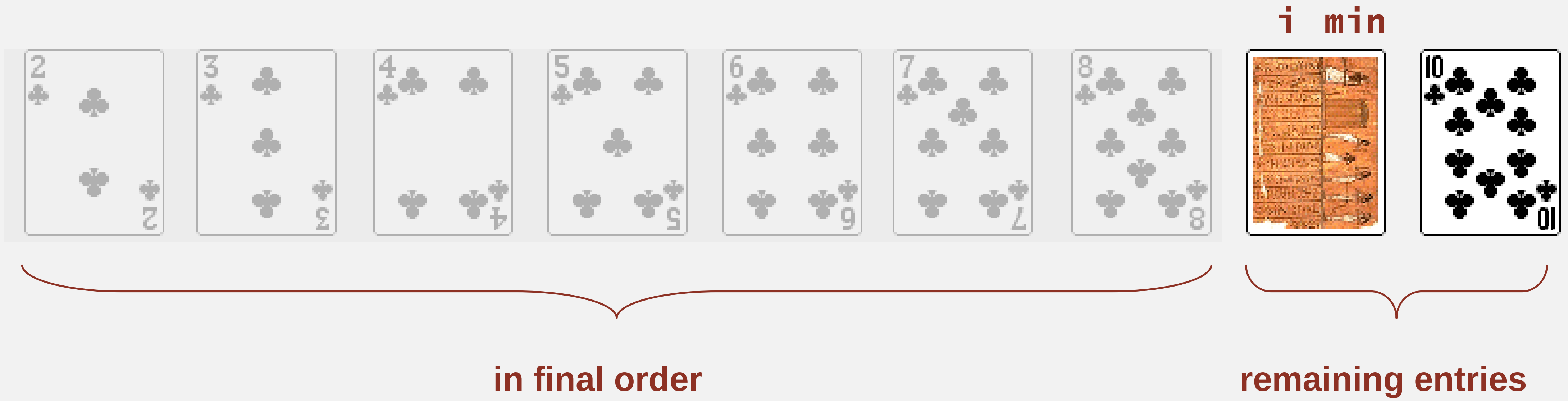
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



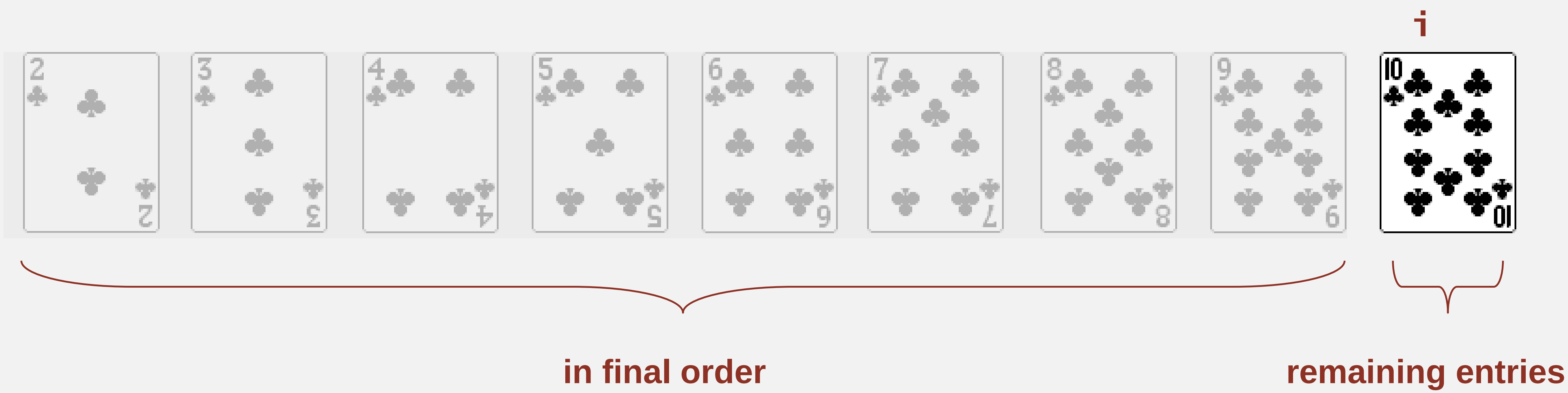
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



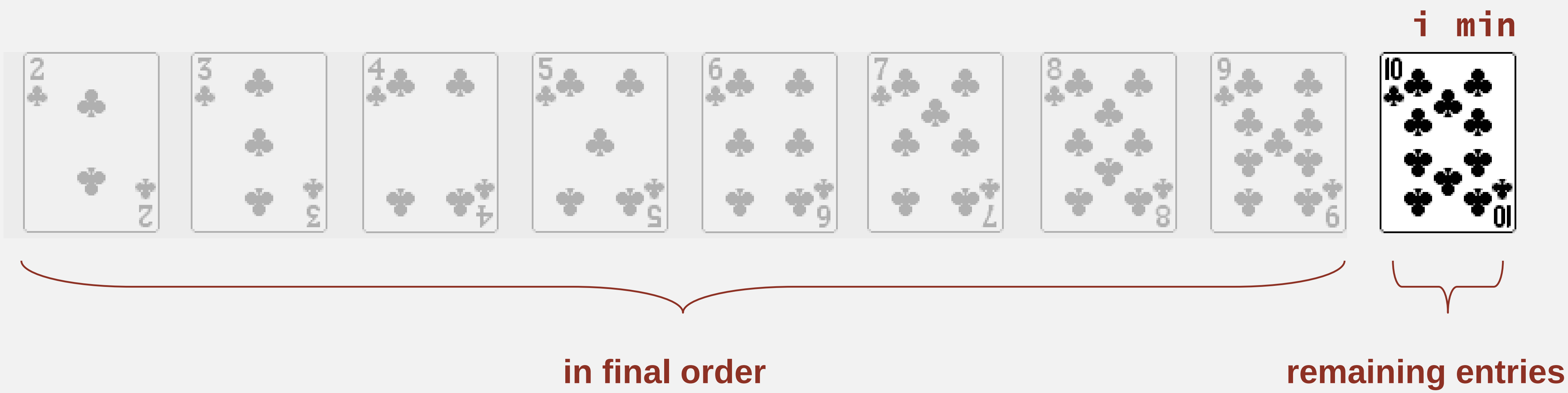
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



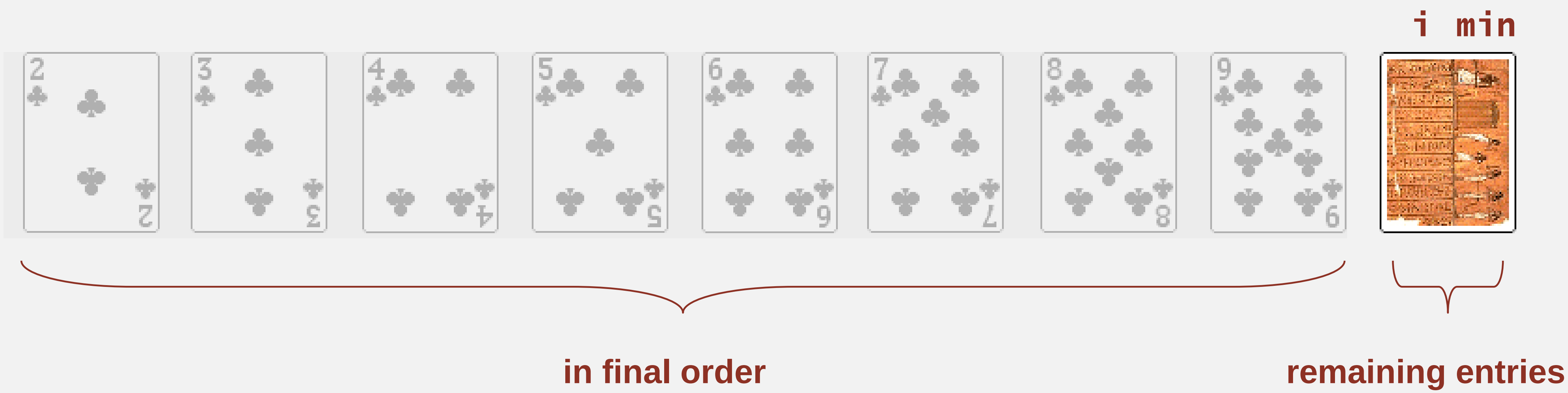
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



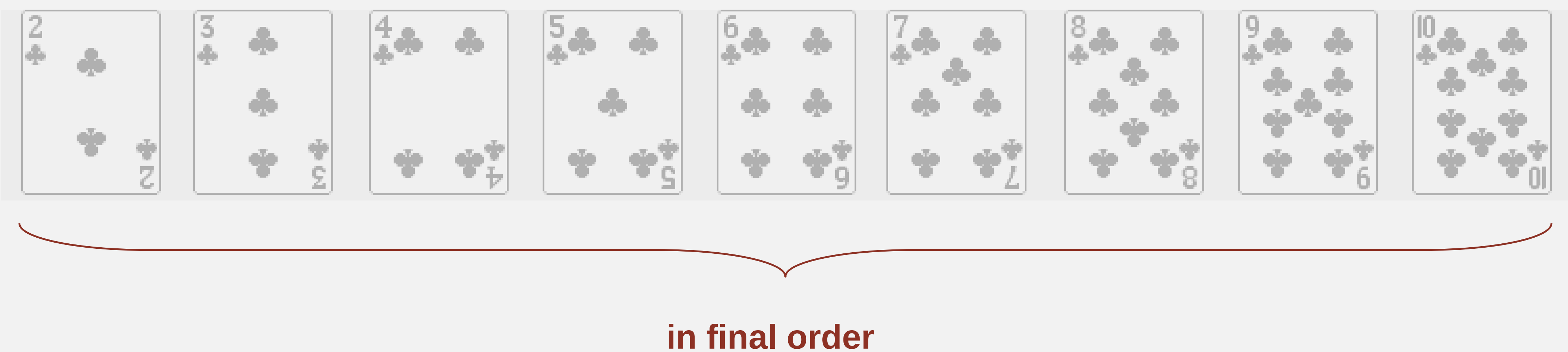
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



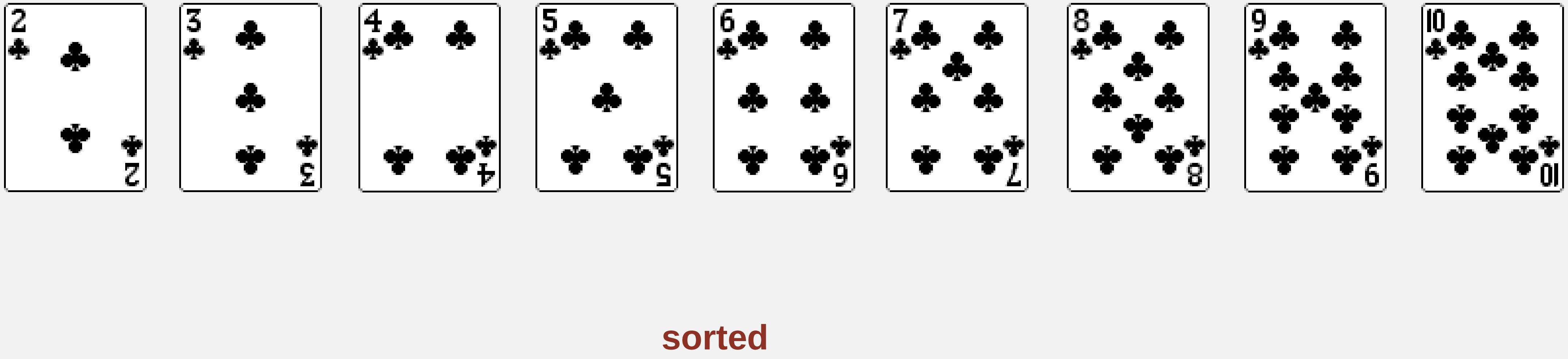
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is item `v` less than `w` ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array `a[]` at index `i` with the one at index `j`.

```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

Selection sort: Java implementation

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }
   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Selection sort: mathematical analysis

Proposition. Selection sort uses $(N-1) + (N-2) + \ \dots + 1 + 0 \ \sim \ N^2/2$ compares and N exchanges.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

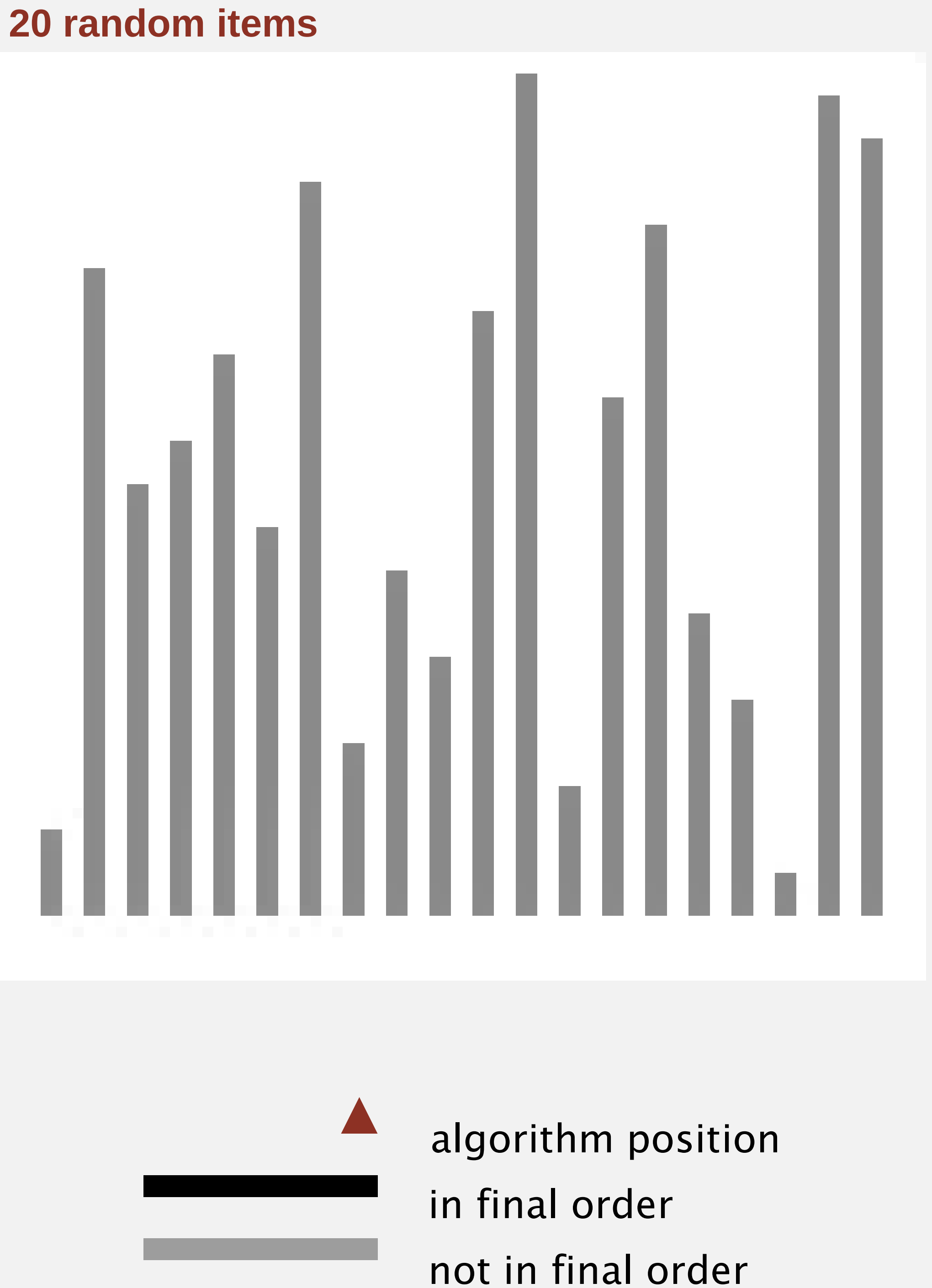
entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if input is sorted.

Data movement is minimal. Linear number of exchanges.

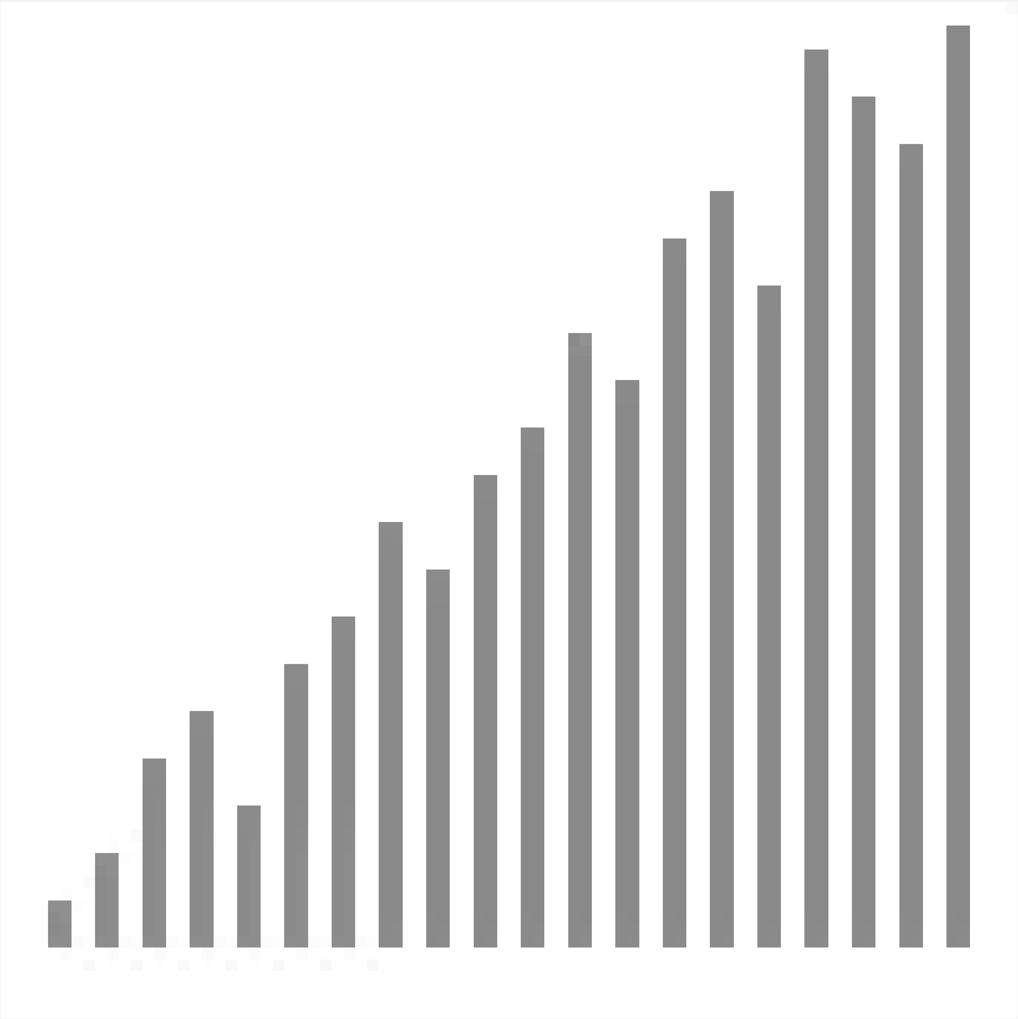
Selection sort: animations

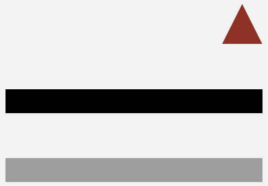


http://www.sorting-algorithms.com/selection-sort

Selection sort: animations

20 partially-sorted items





http://www.sorting-algorithms.com/selection-sort

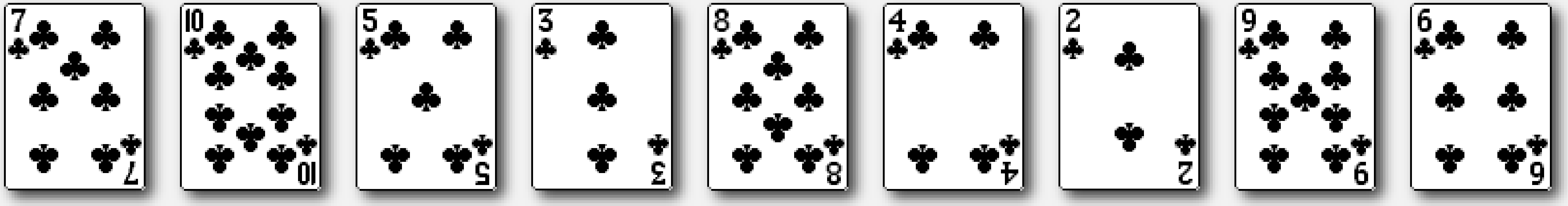
http://algs4.cs.princeton.edu

Elementary Sorts

- *selection sort*
- *insertion sort*
- *shuffling*

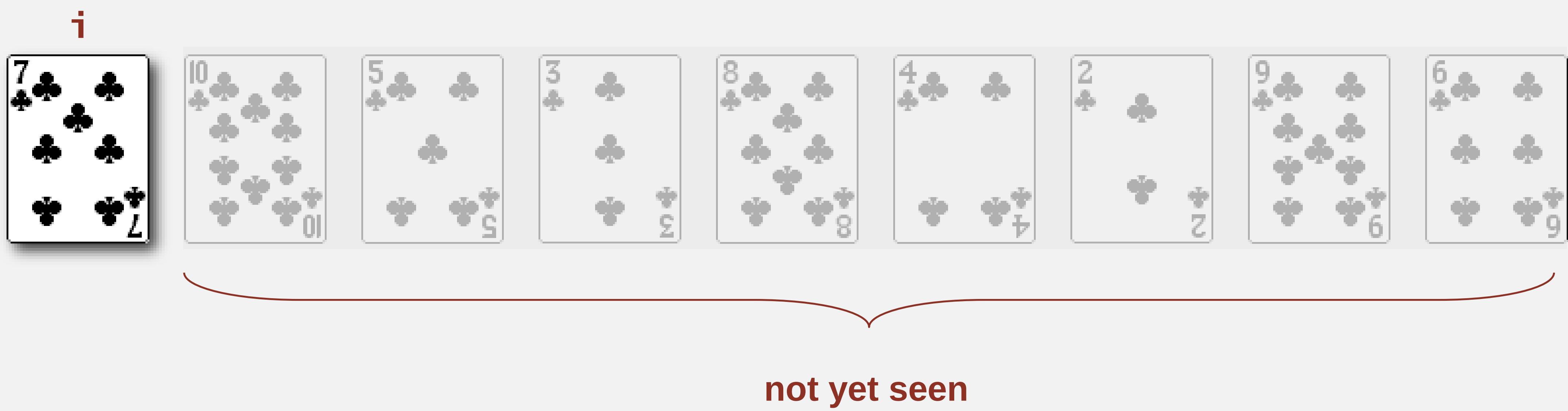
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



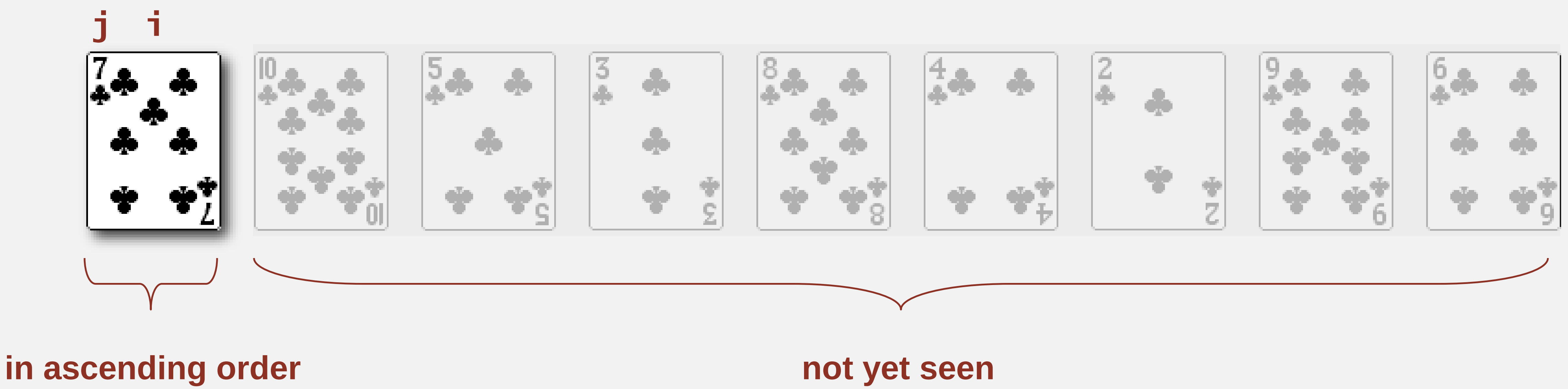
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



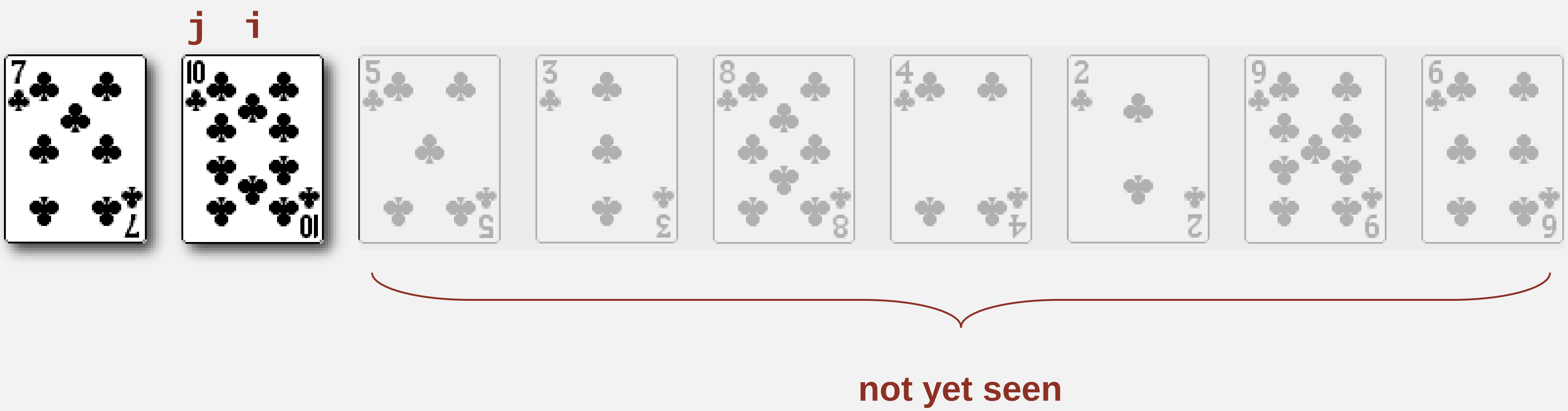
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



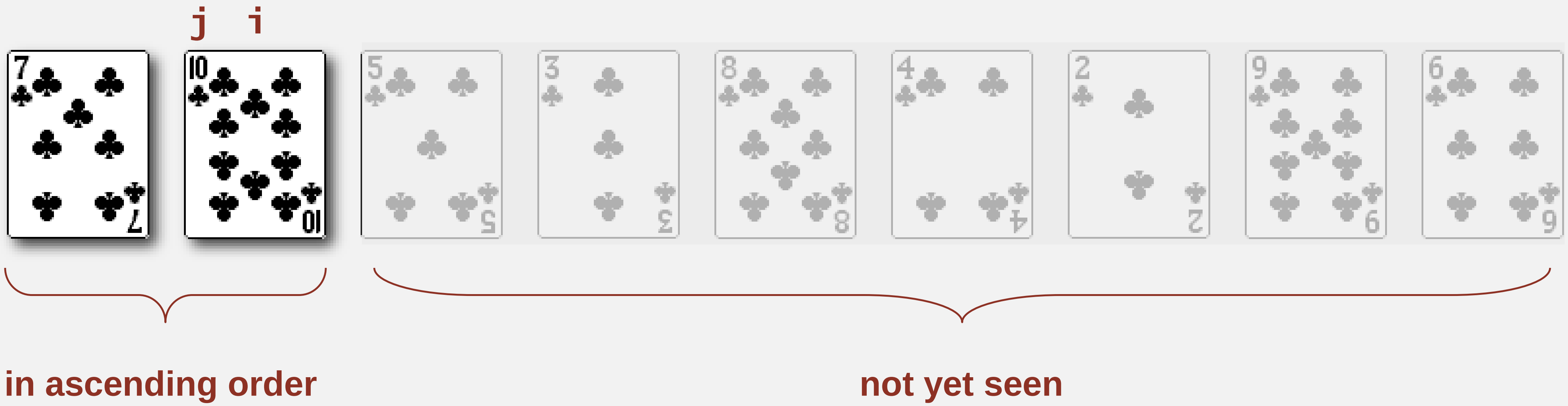
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



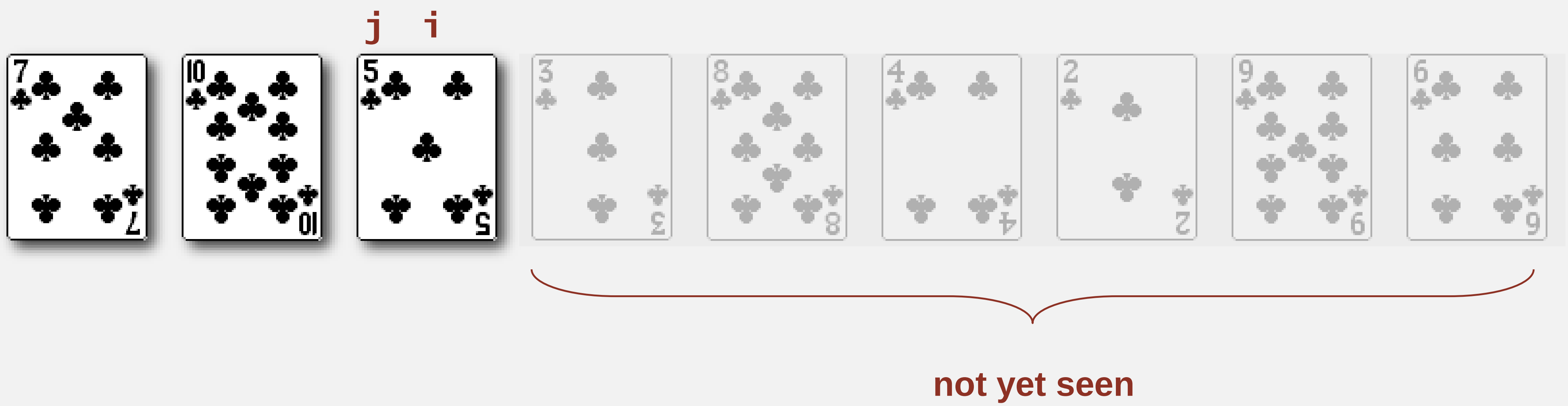
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



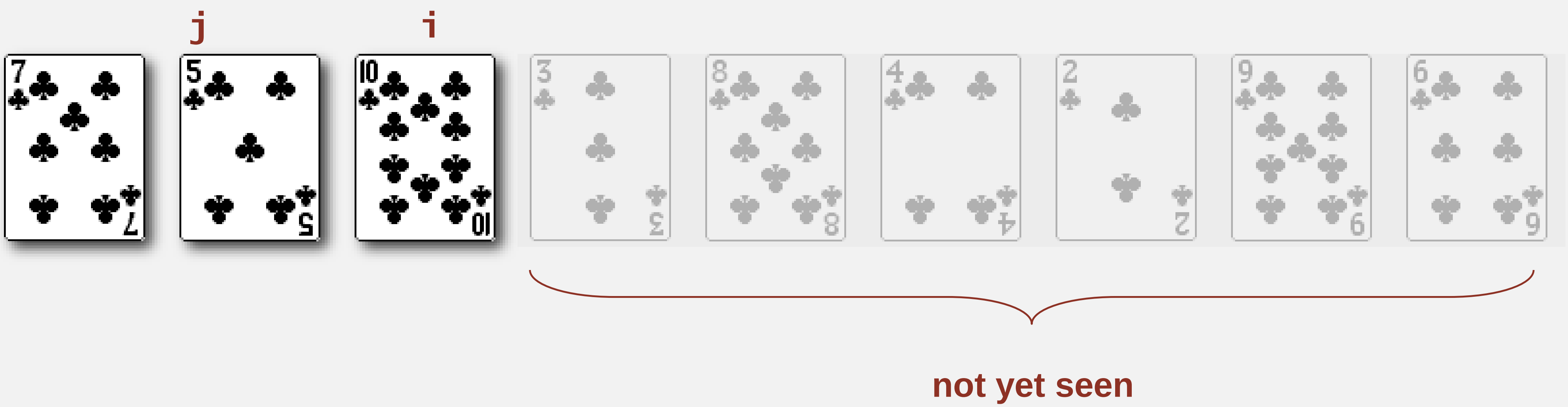
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



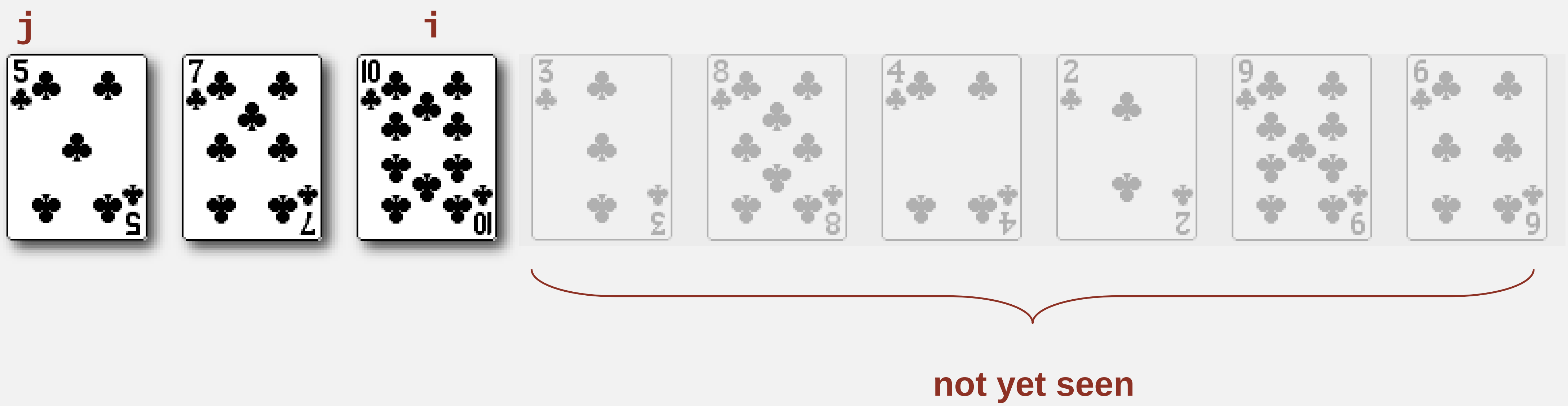
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



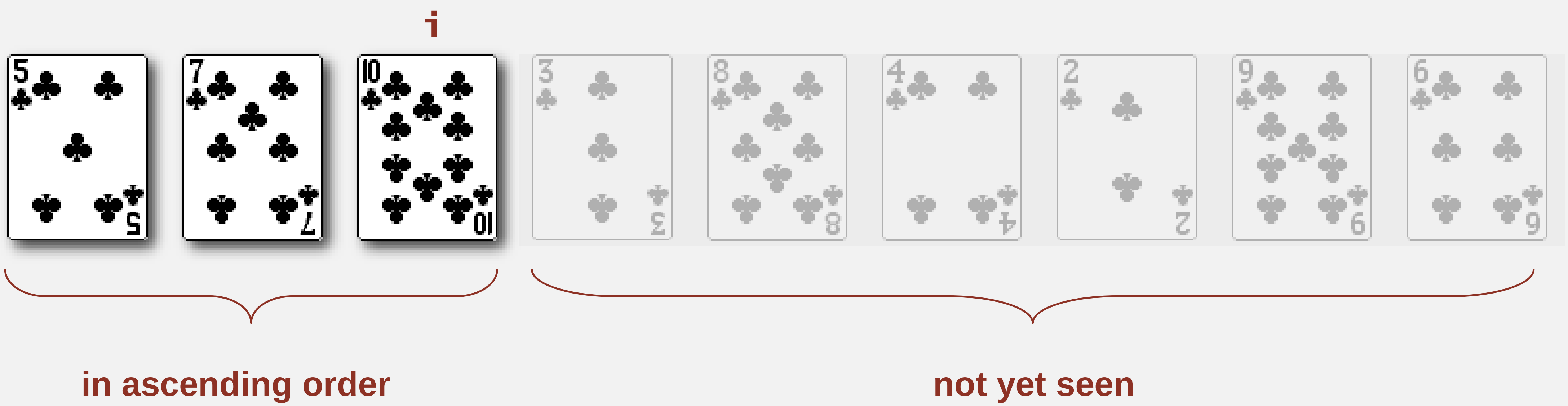
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



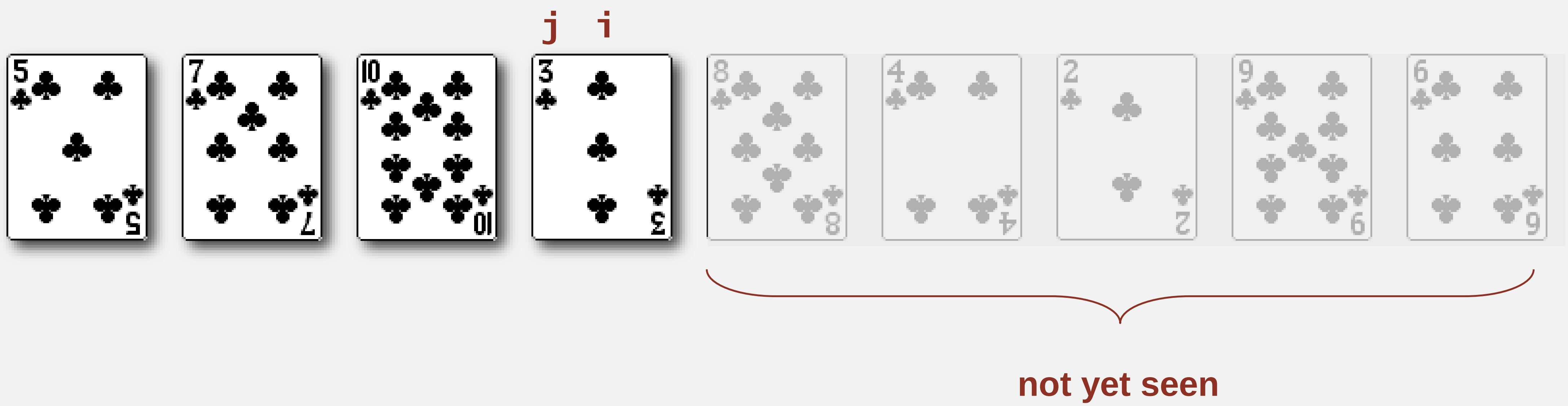
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



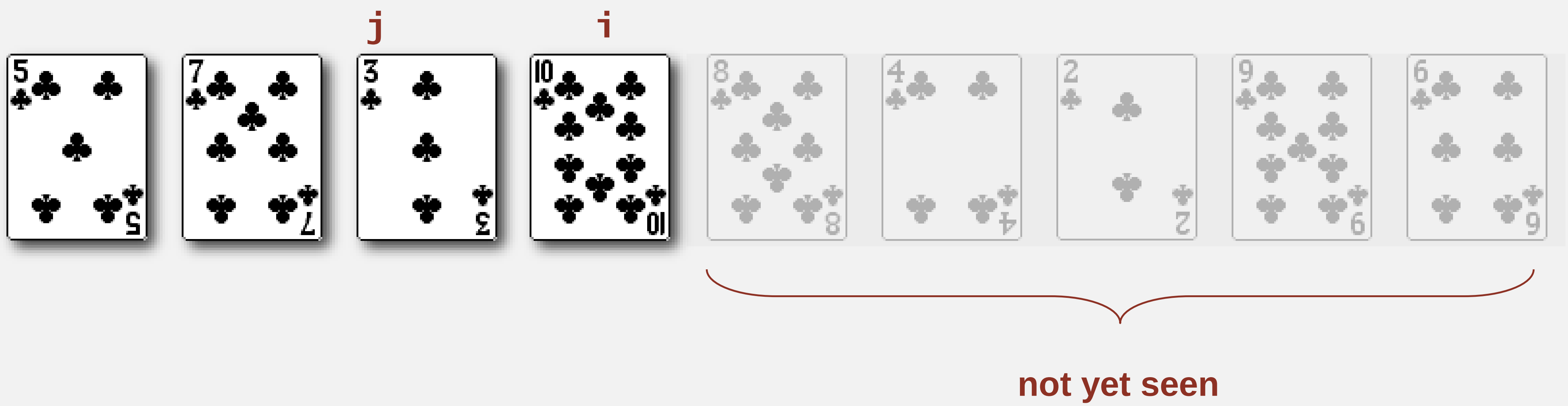
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



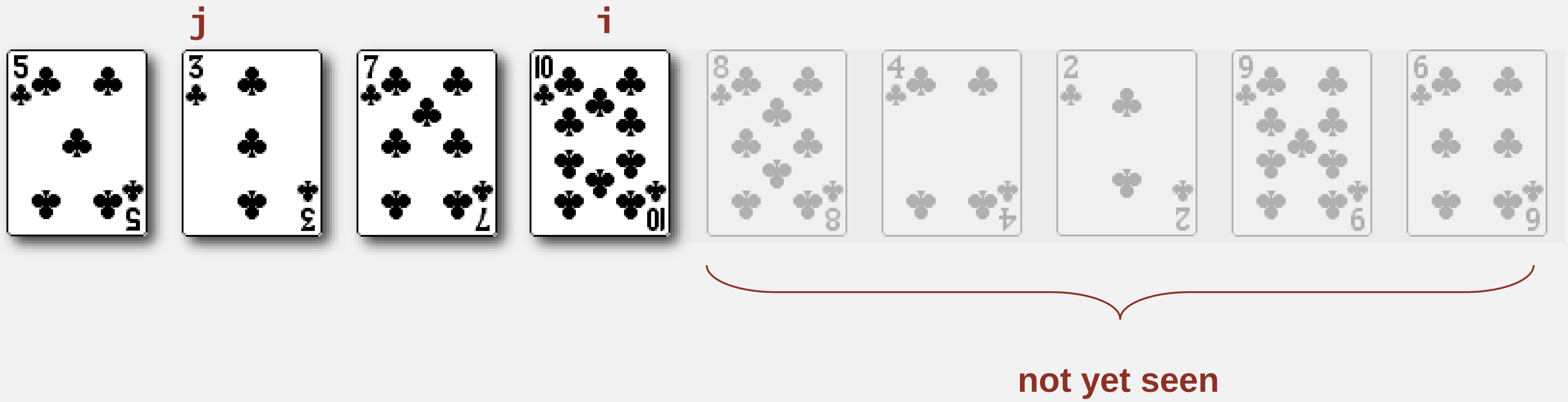
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



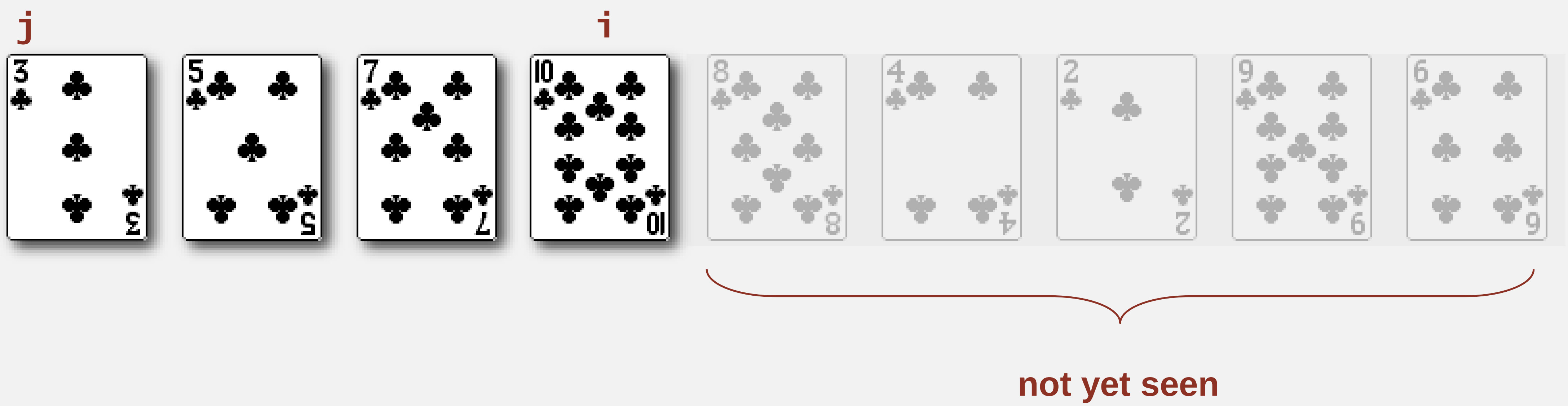
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



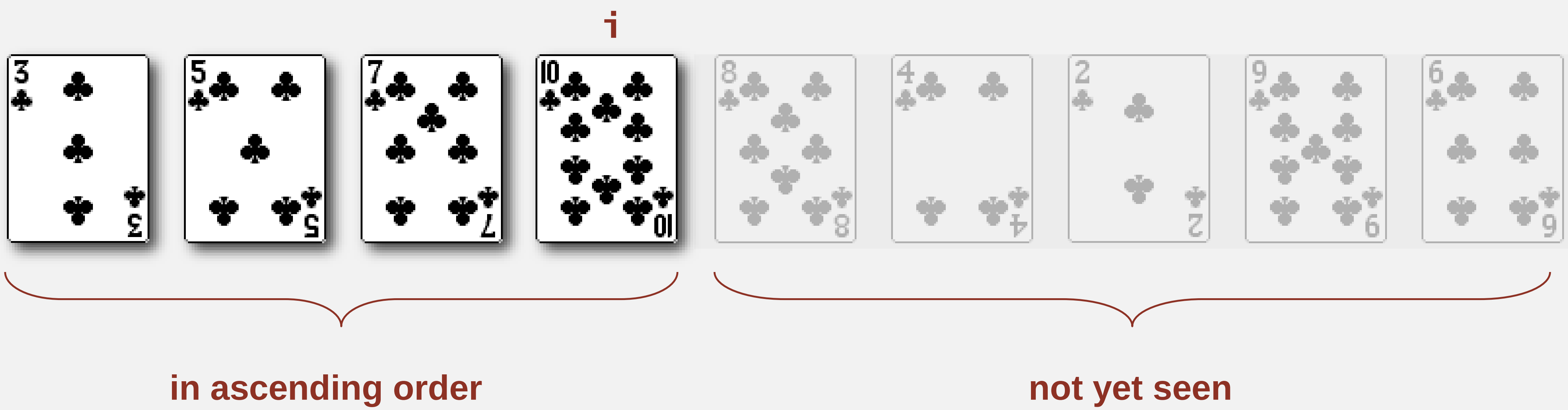
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



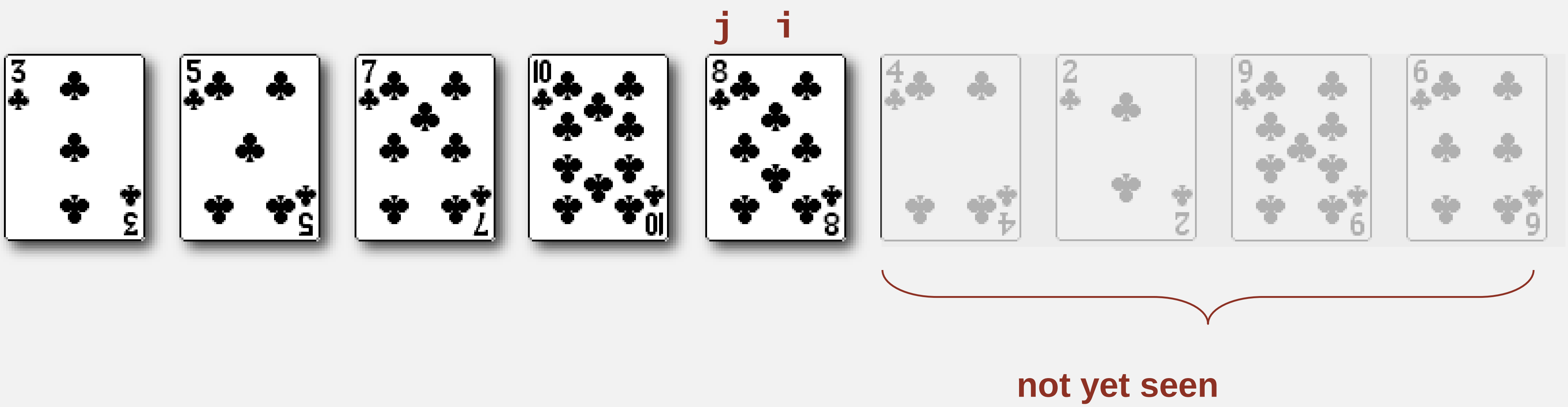
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



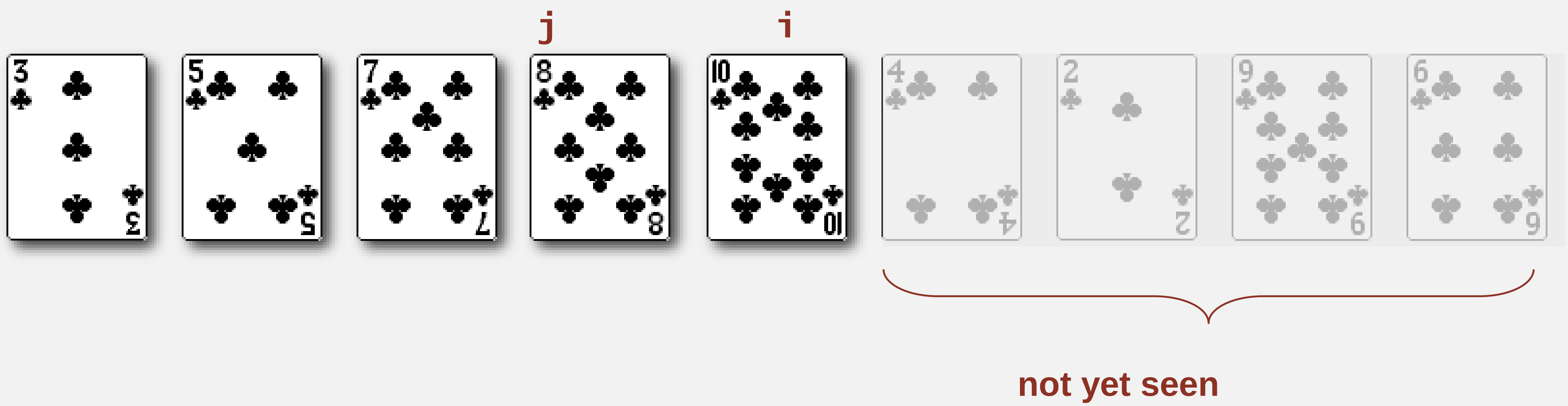
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



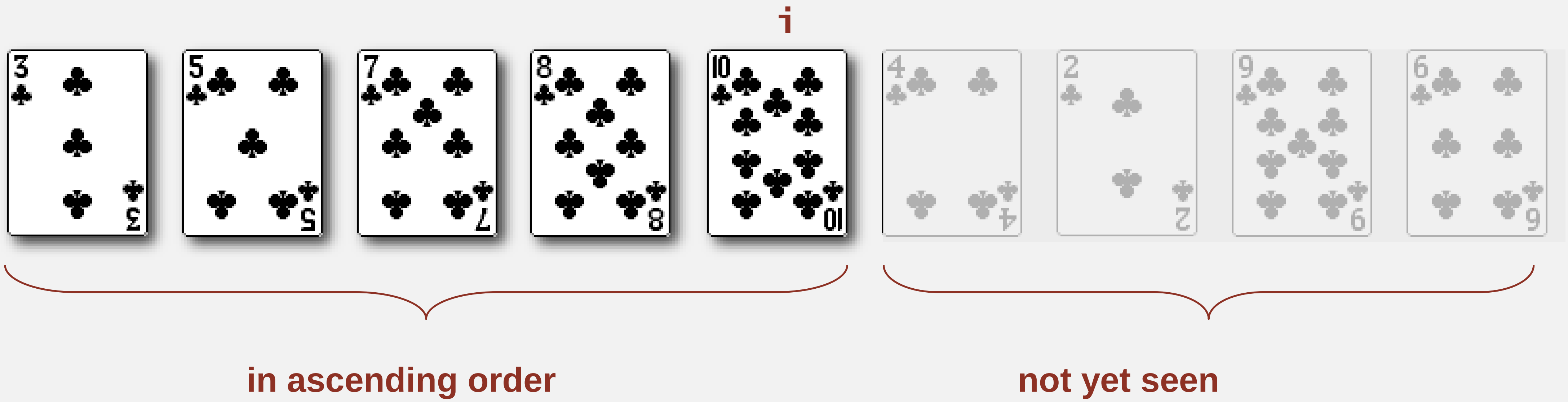
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



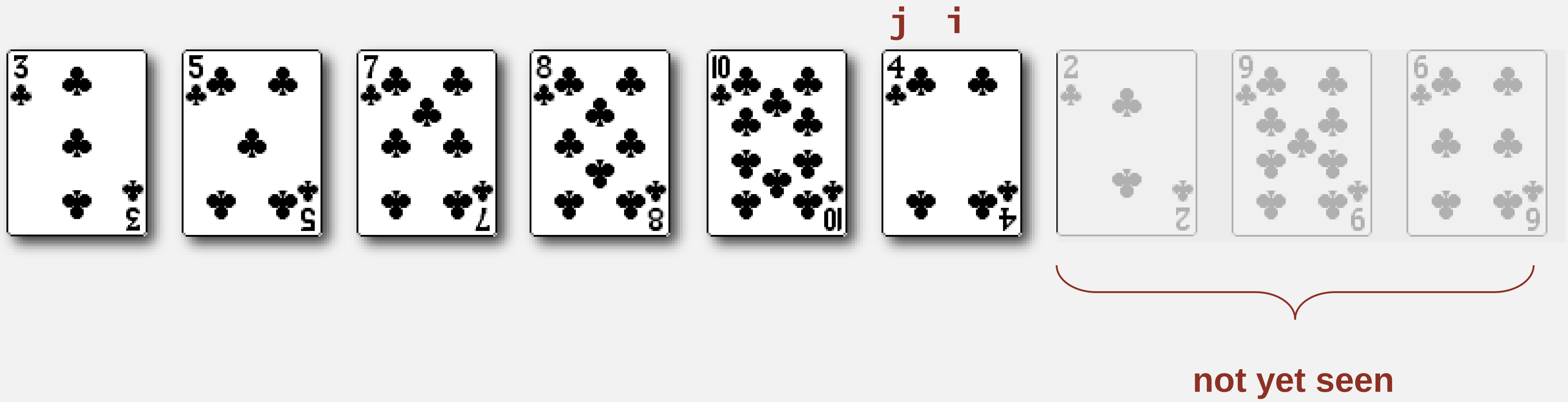
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



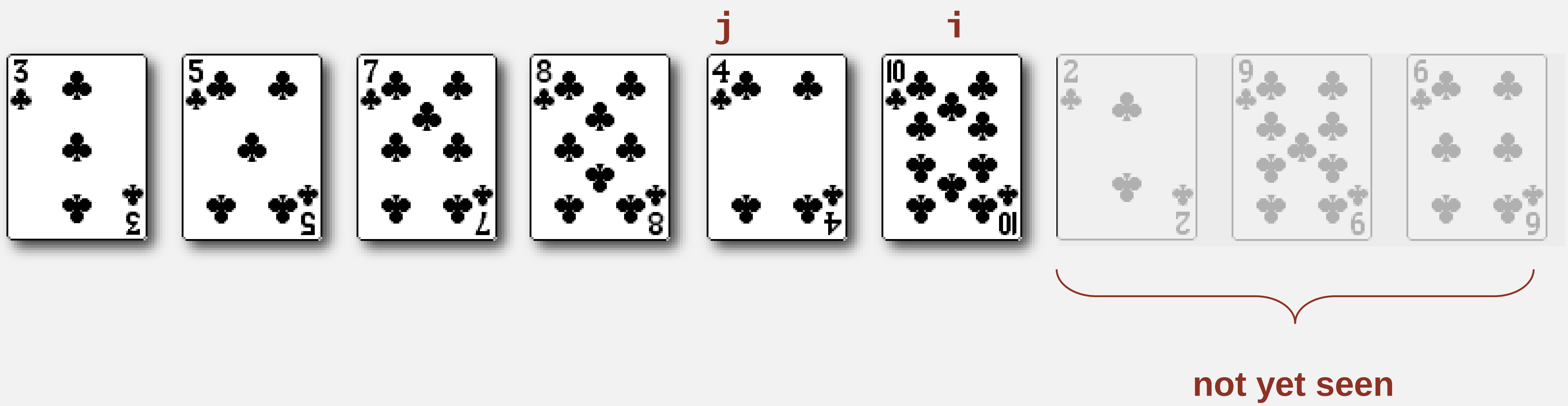
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



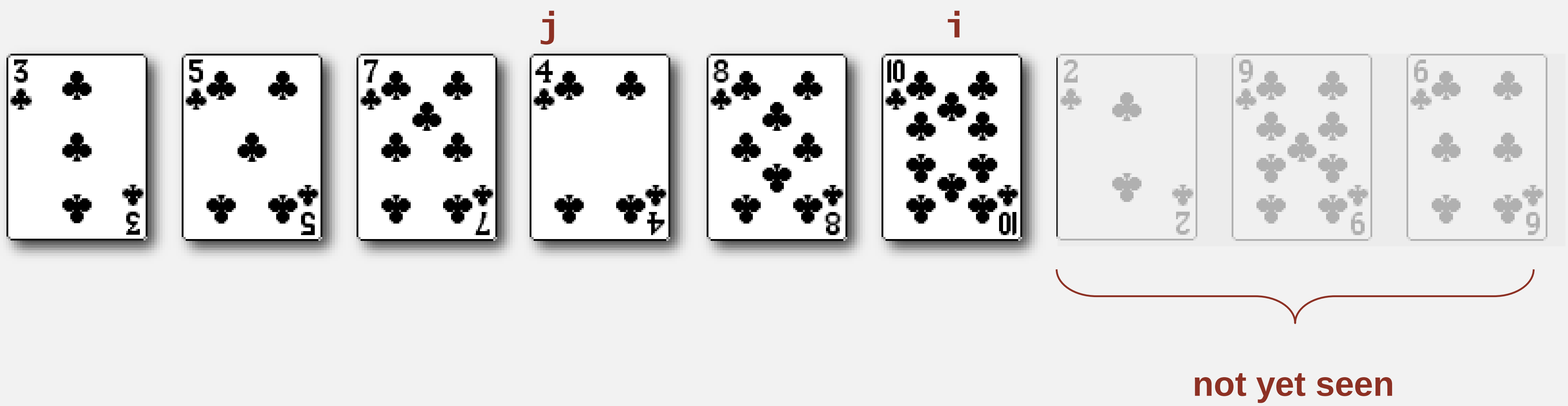
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



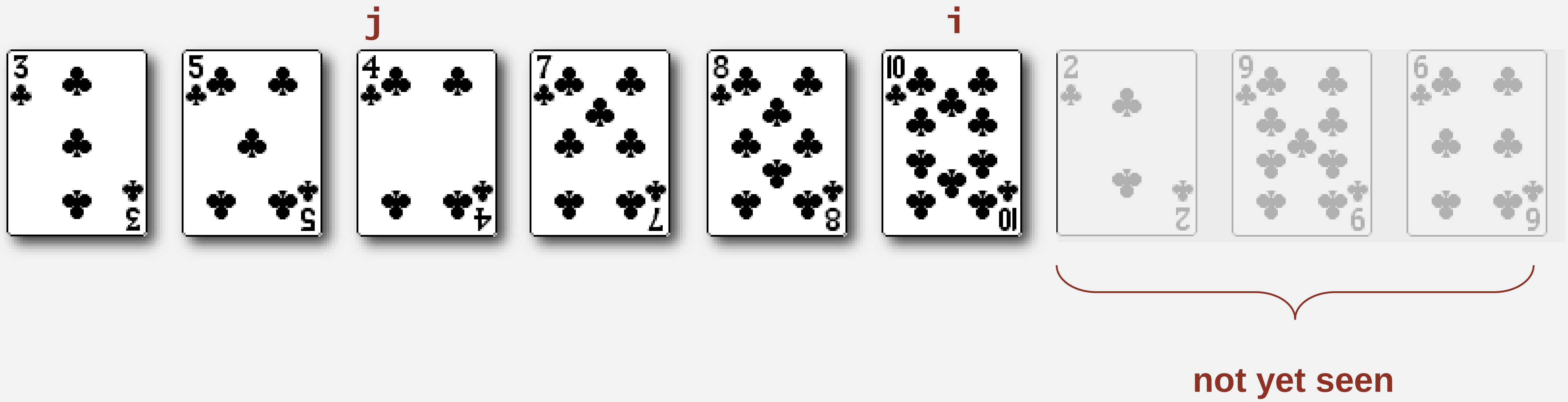
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



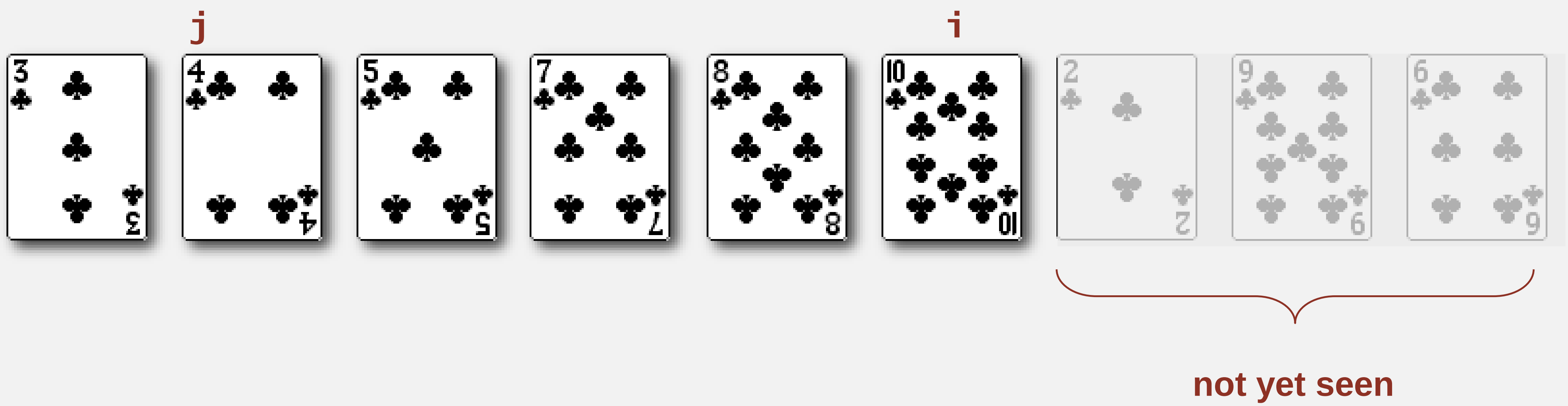
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



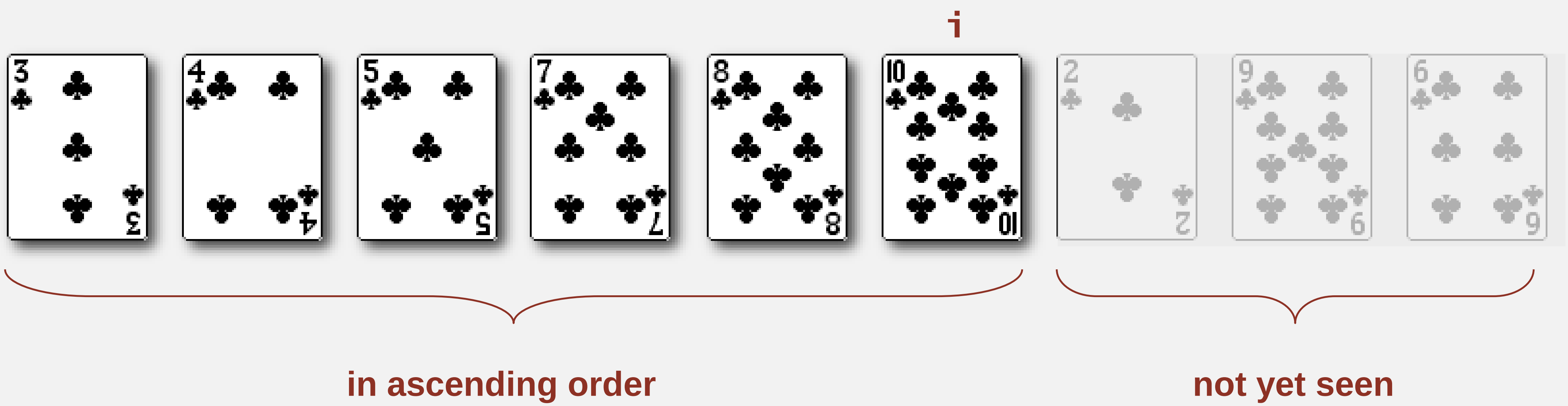
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



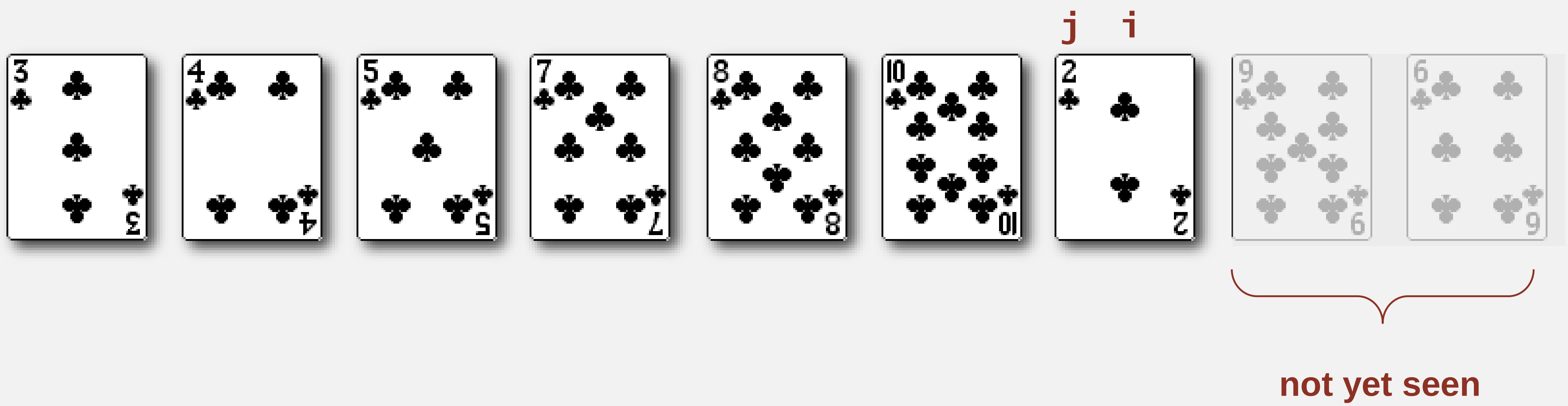
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



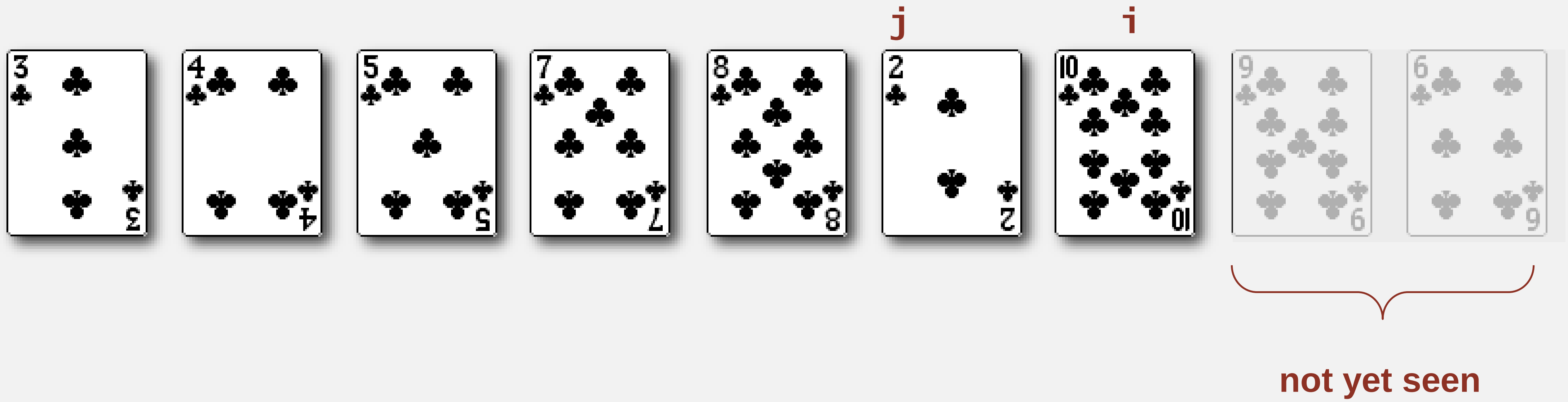
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



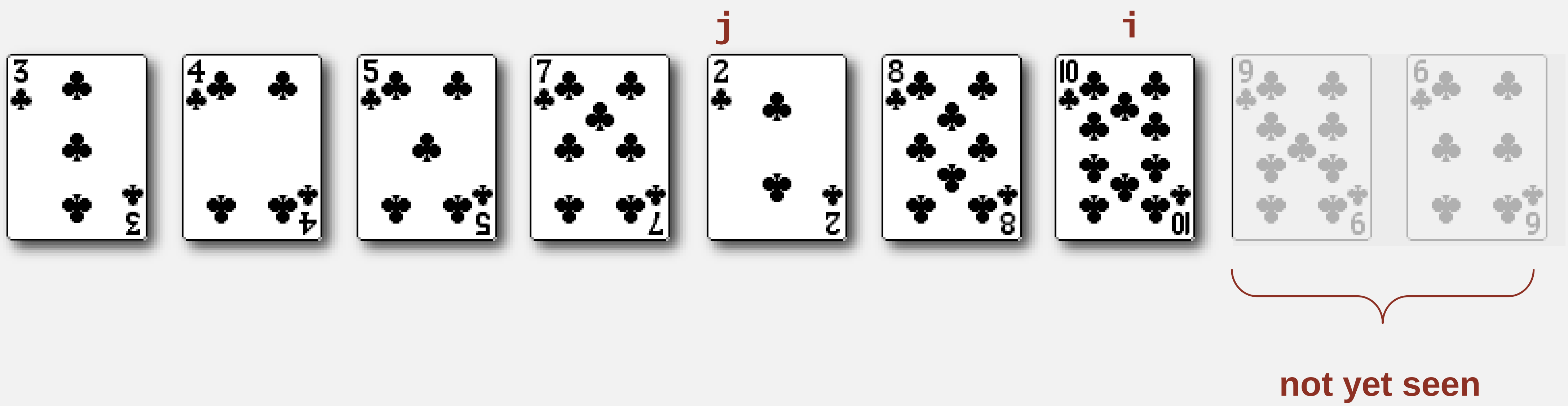
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



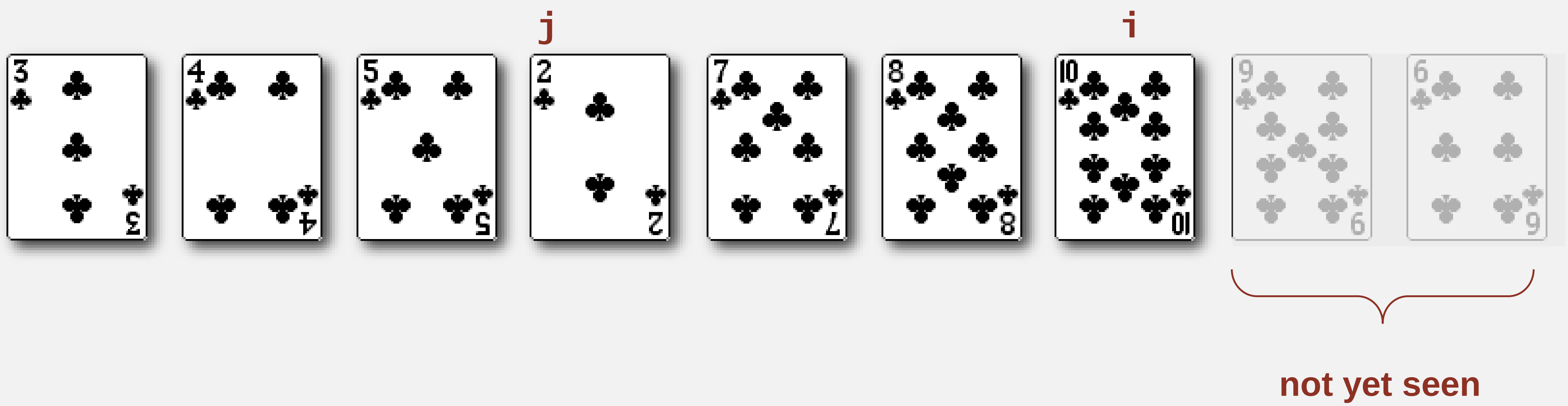
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



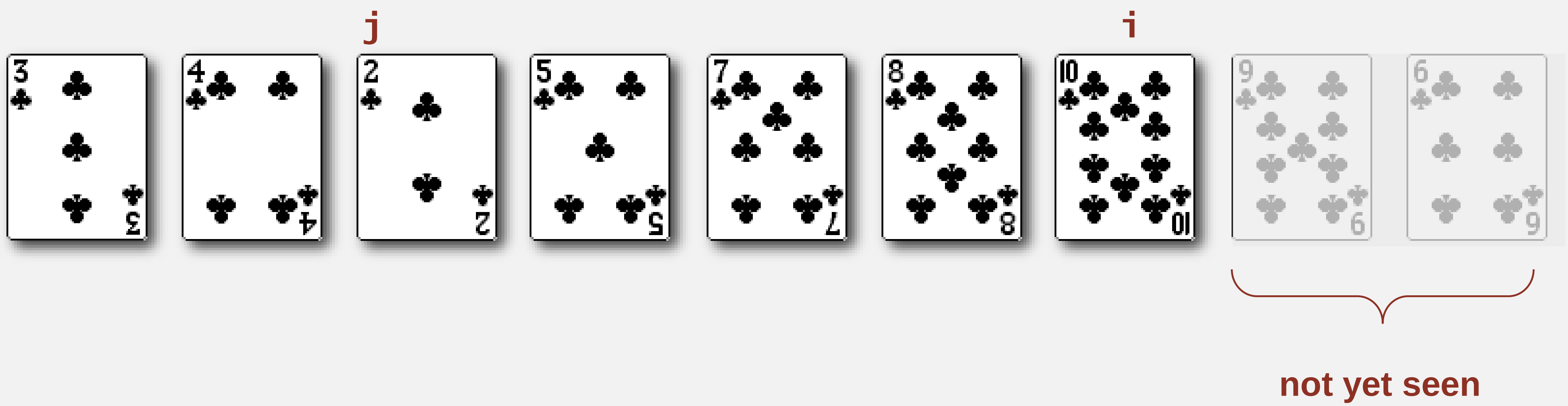
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



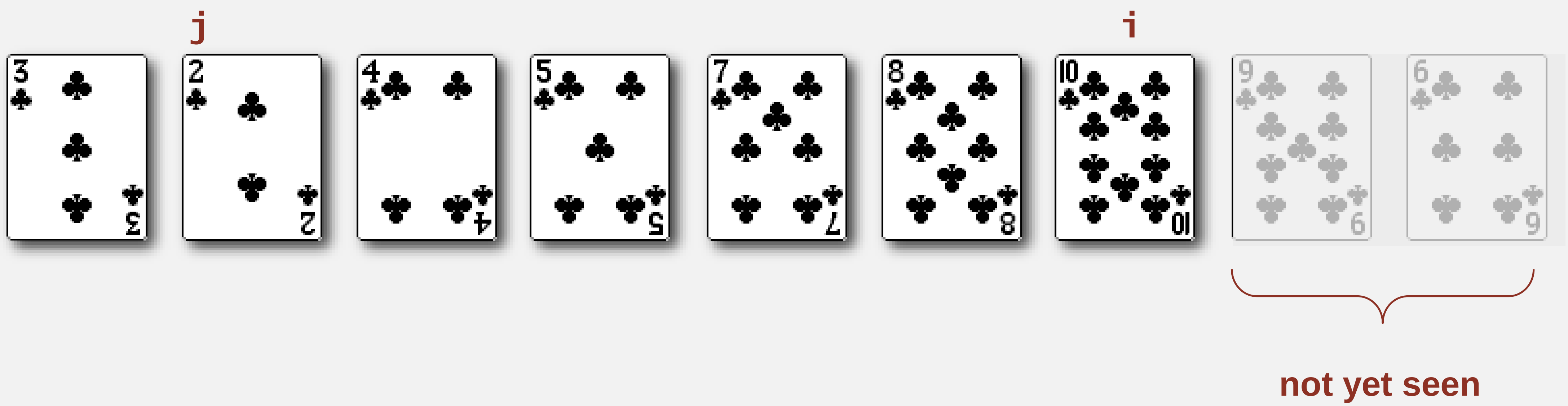
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



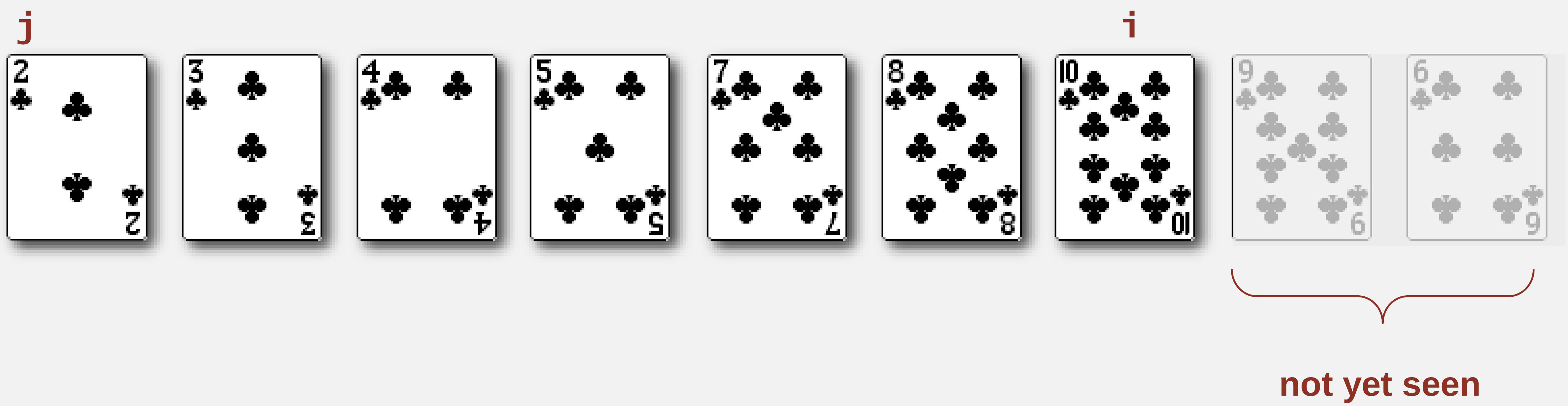
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



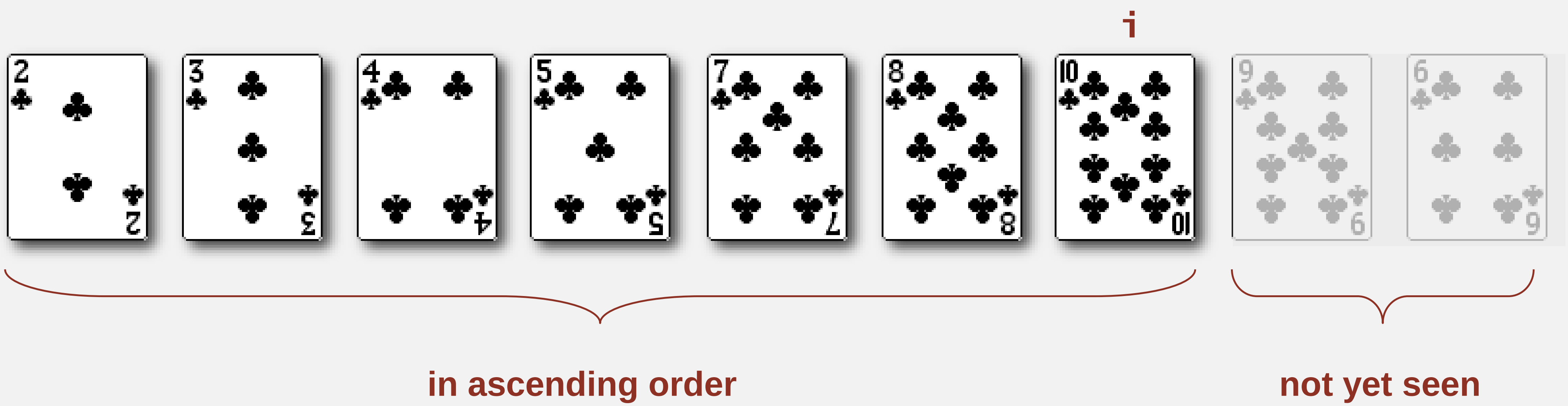
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



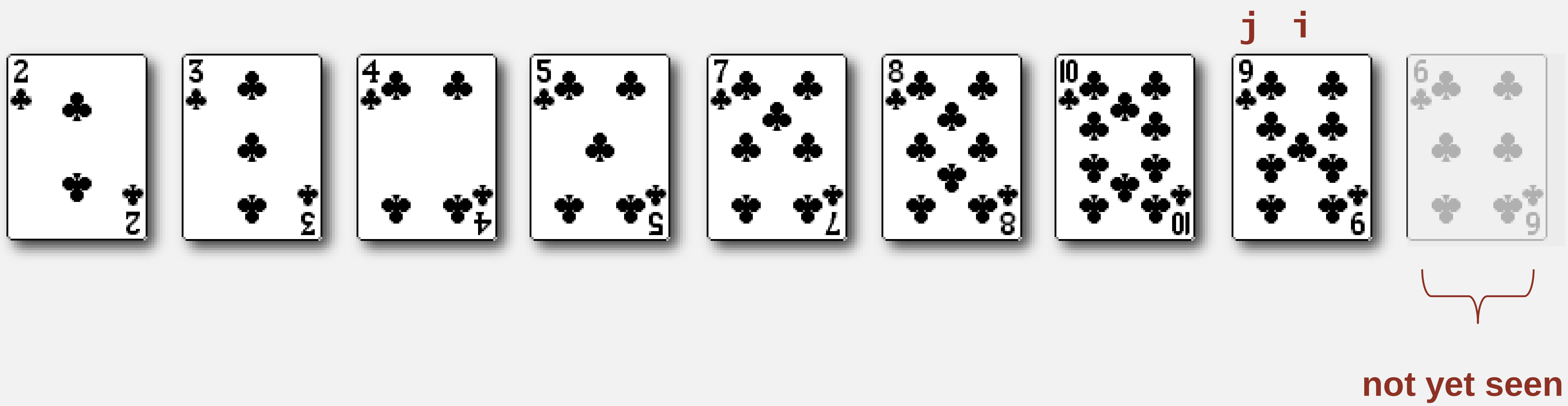
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



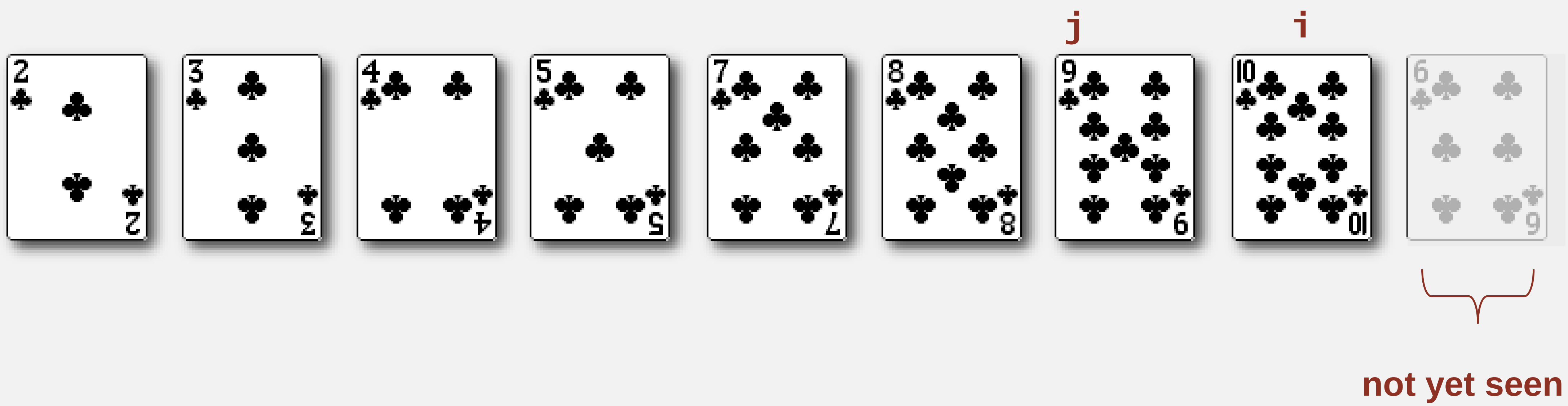
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



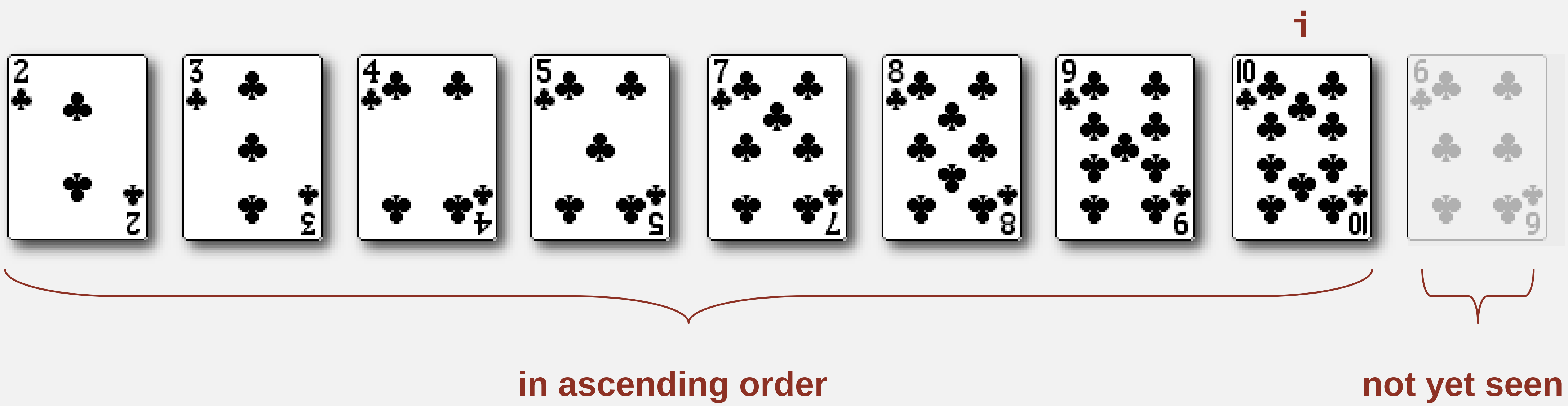
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



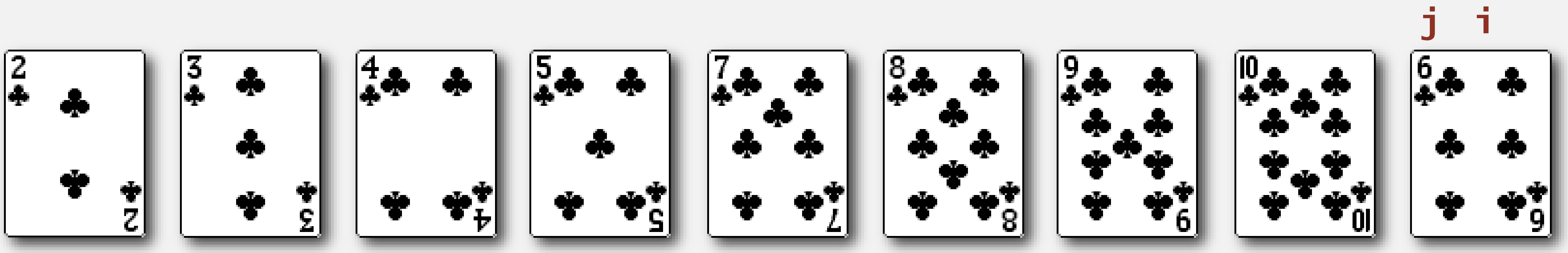
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



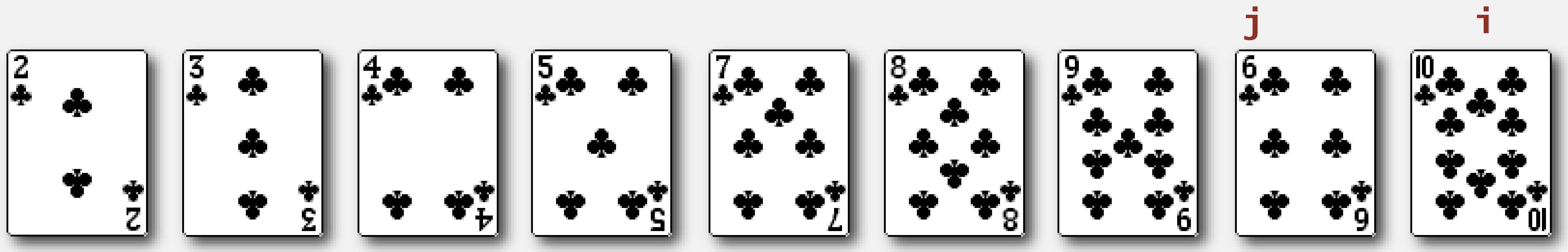
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



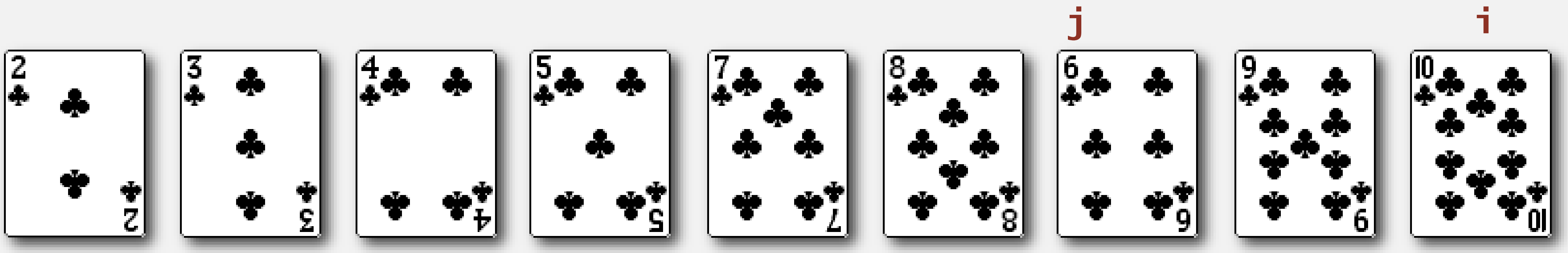
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



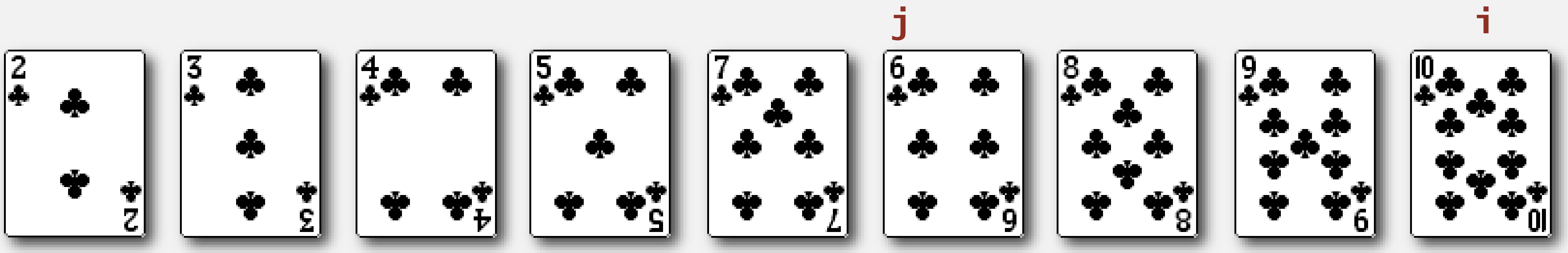
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



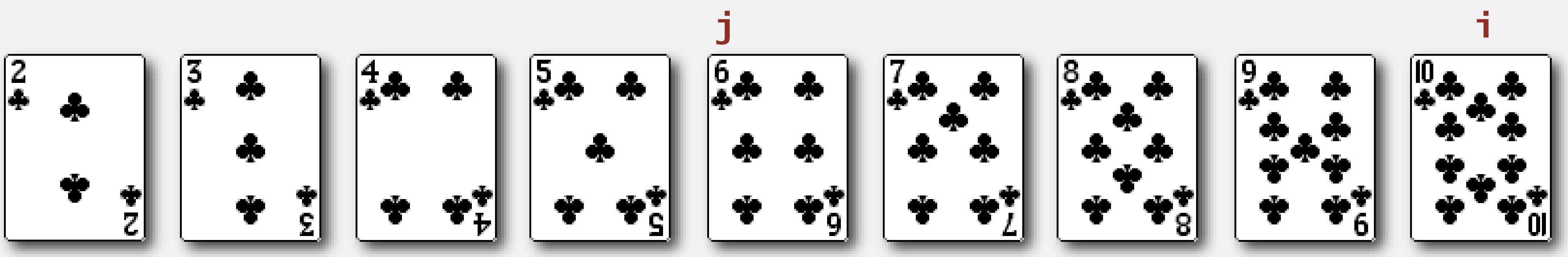
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



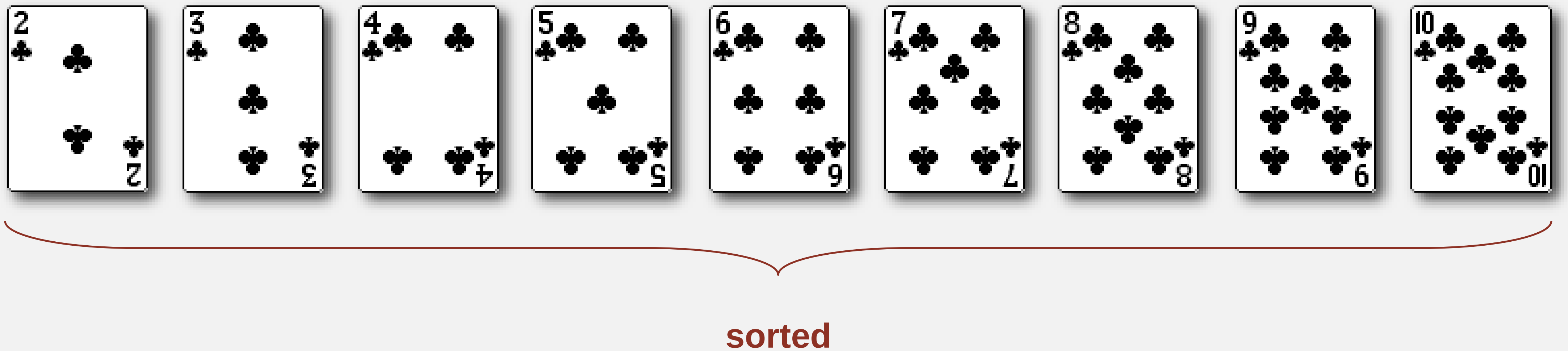
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion sort: Java implementation

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Insertion sort: mathematical analysis

Proposition. To sort a randomly-ordered array with distinct keys, insertion sort uses $\sim \frac{1}{4} N^2$ compares and $\sim \frac{1}{4} N^2$ exchanges on average.

Pf. Expect each entry to move halfway back.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

Insertion sort: analysis

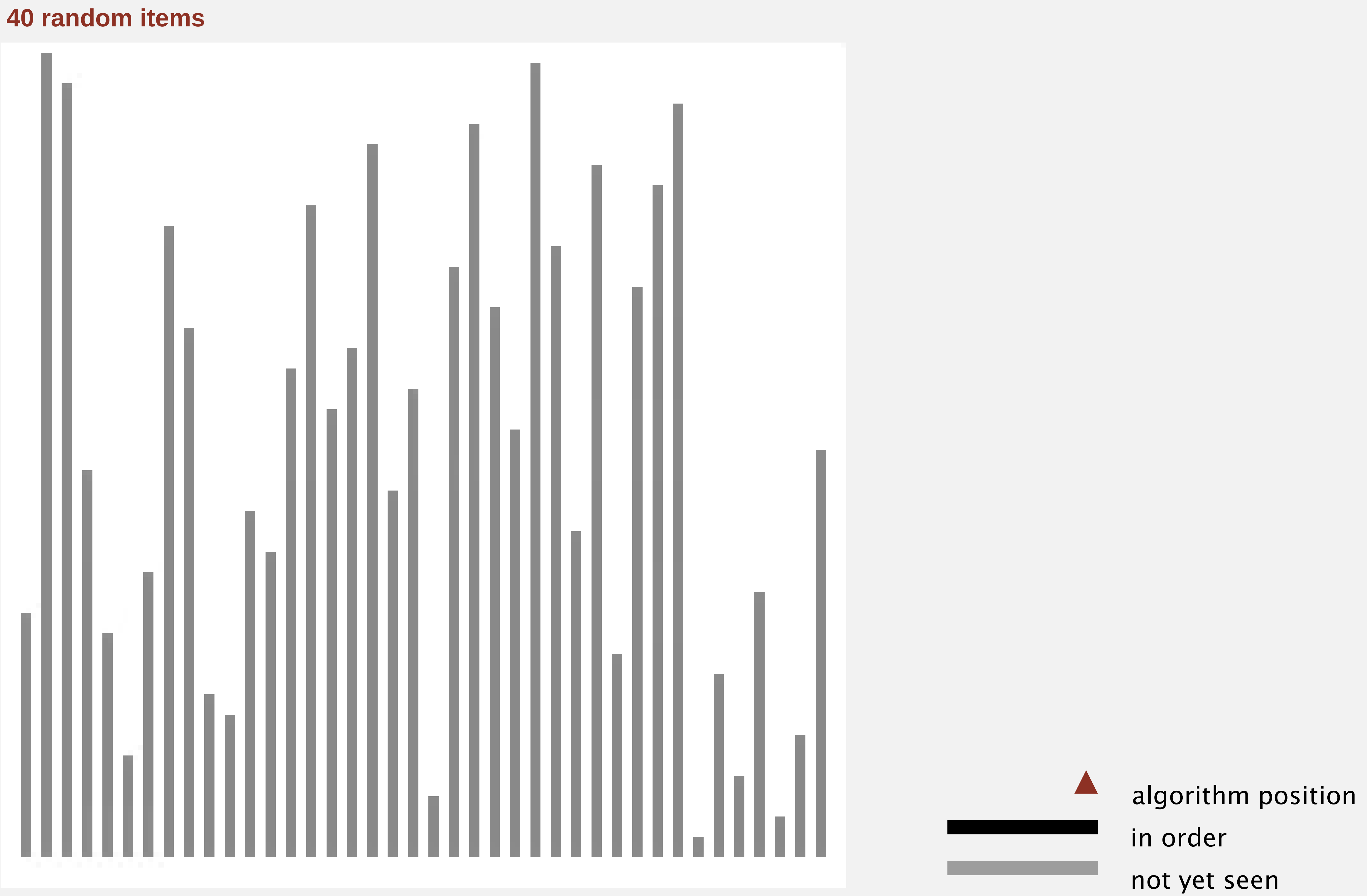
Best case. If the array is in ascending order, insertion sort makes $N-1$ compares and 0 exchanges.

A E E L M O P R S T X

Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

X T S R P O M L F E A

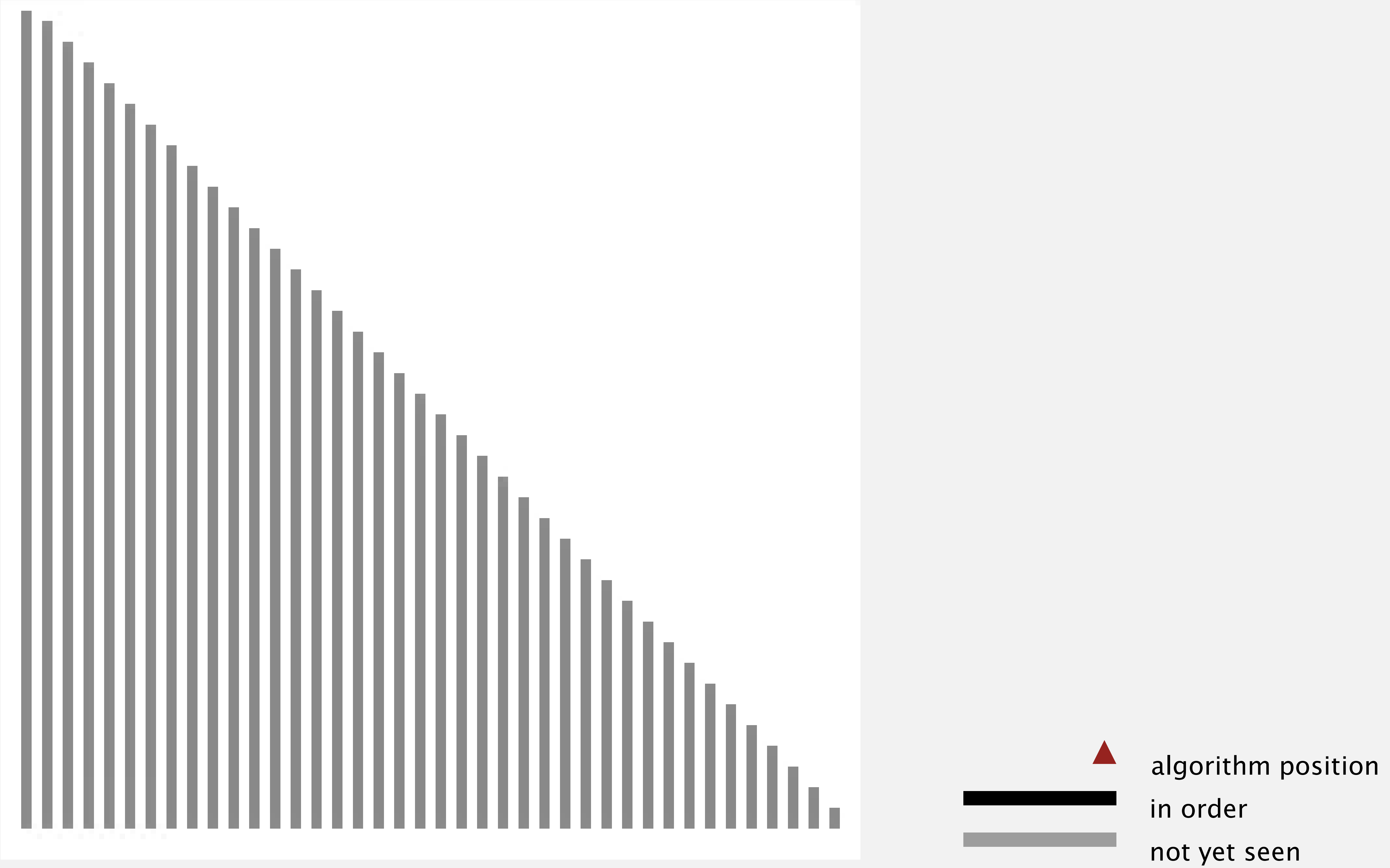
Insertion sort: animation



http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

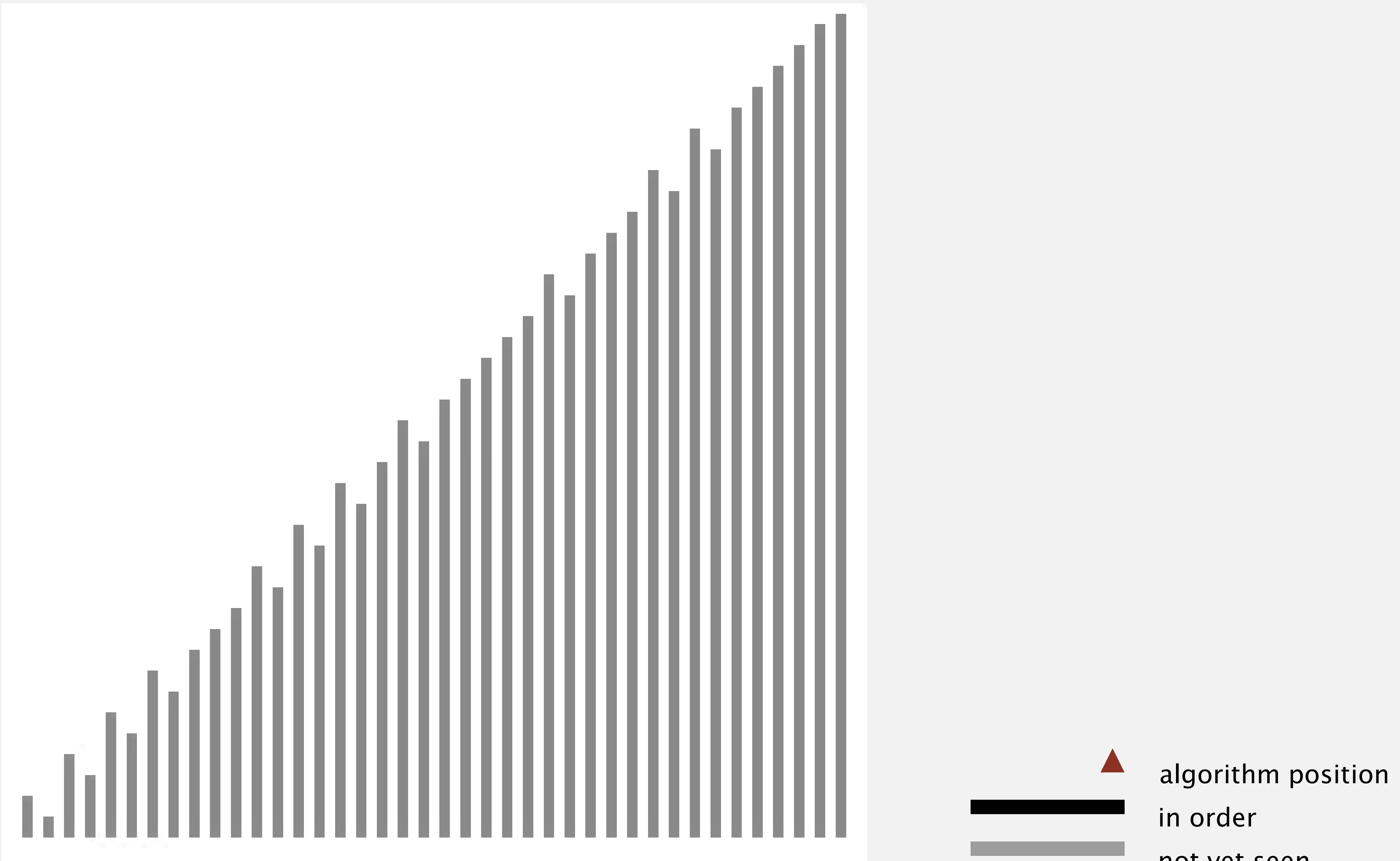
40 reverse-sorted items



http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 partially-sorted items



http://www.sorting-algorithms.com/insertion-sort

http://algs4.cs.princeton.edu

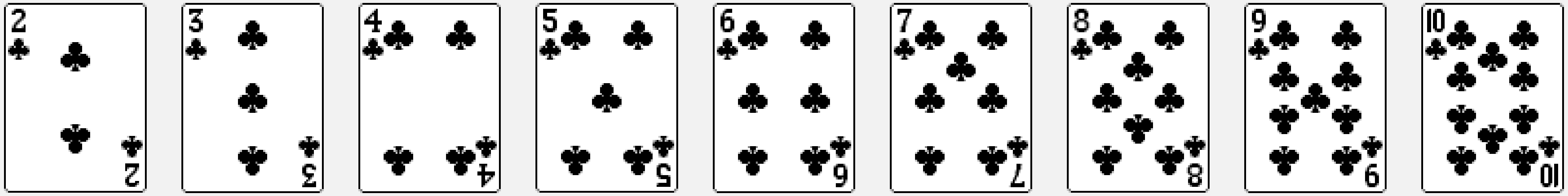
Elementary Sorts

- *selection sort*
- *insertion sort*
- *shuffling*

How to shuffle an array

Goal. Rearrange array so that result is a uniformly random permutation.

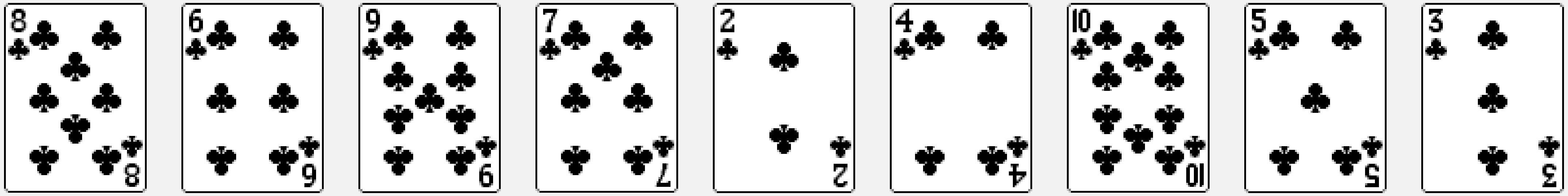
all permutations
equally likely



How to shuffle an array

Goal. Rearrange array so that result is a uniformly random permutation.

all permutations equally likely

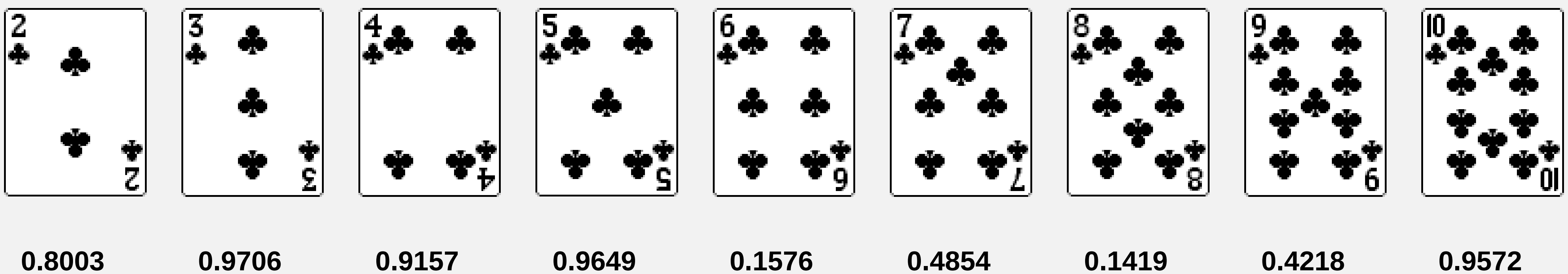


Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling

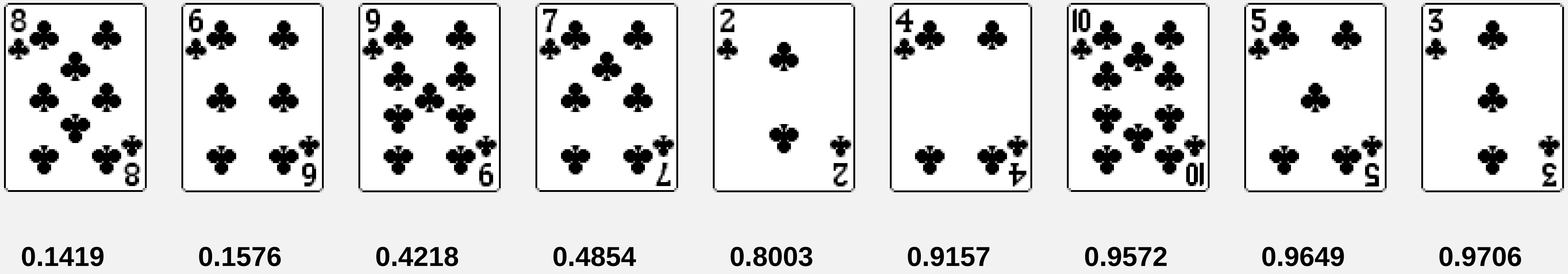
columns in a spreadsheet



Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.

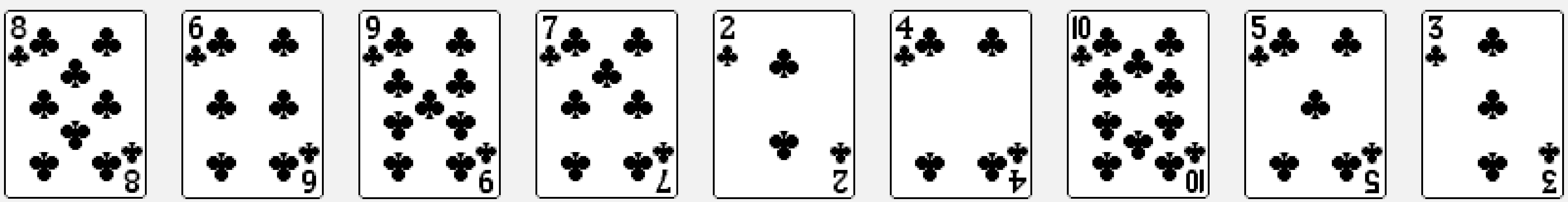
useful for shuffling columns in a spreadsheet



Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling columns in a spreadsheet



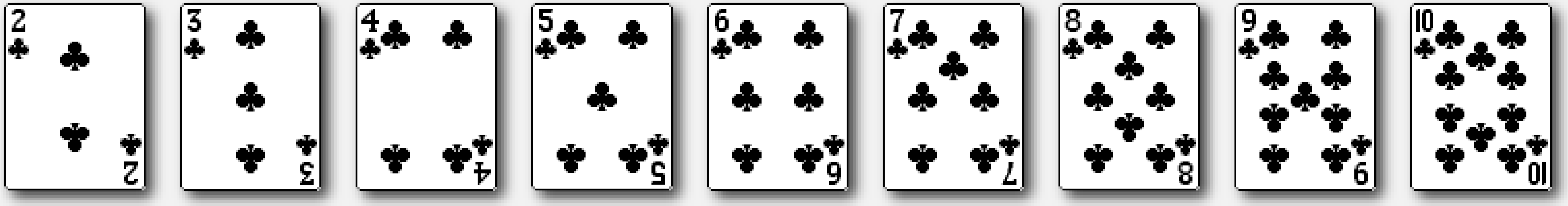
0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Proposition. Shuffle sort produces a uniformly random permutation.

assuming real numbers uniformly at random (and no ties)

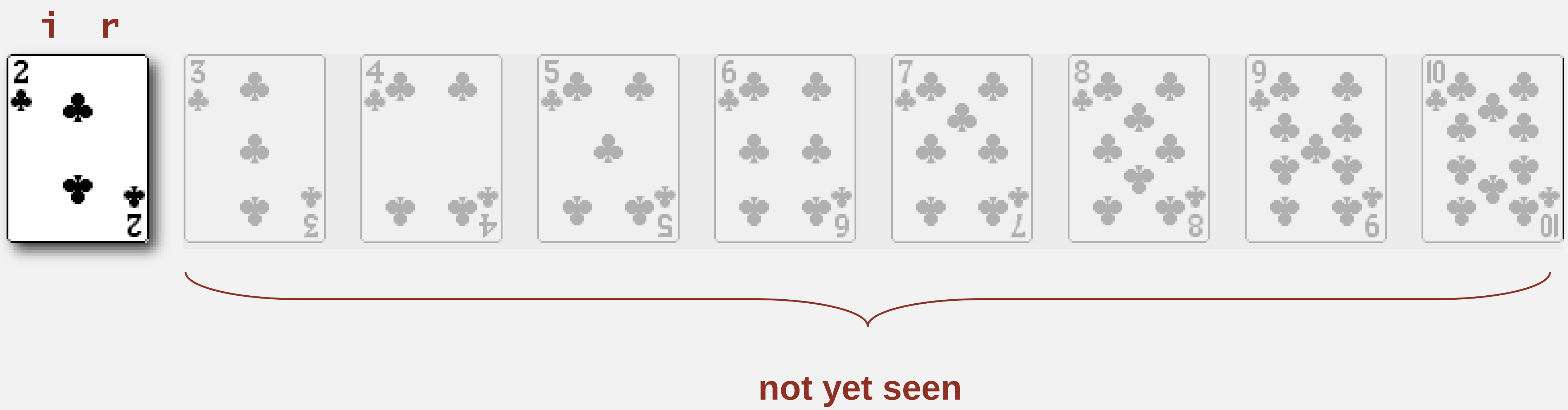
Knuth shuffle demo

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



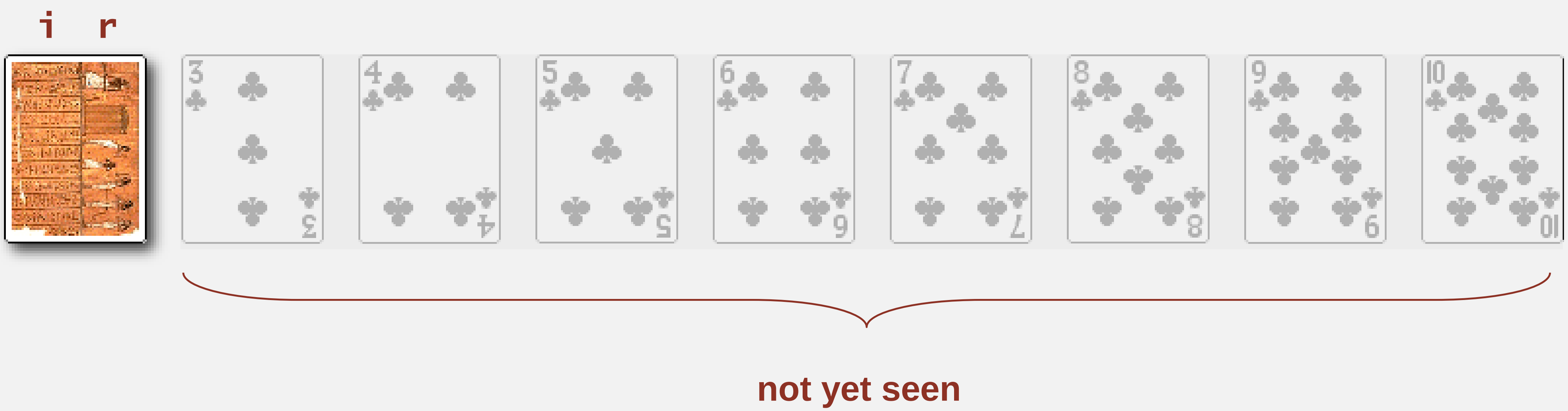
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



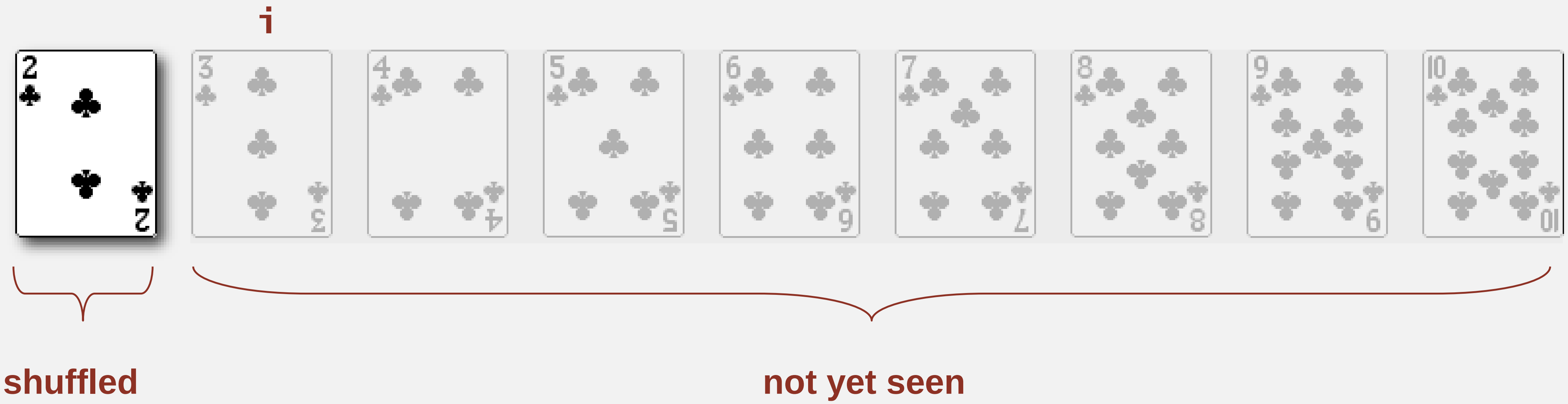
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



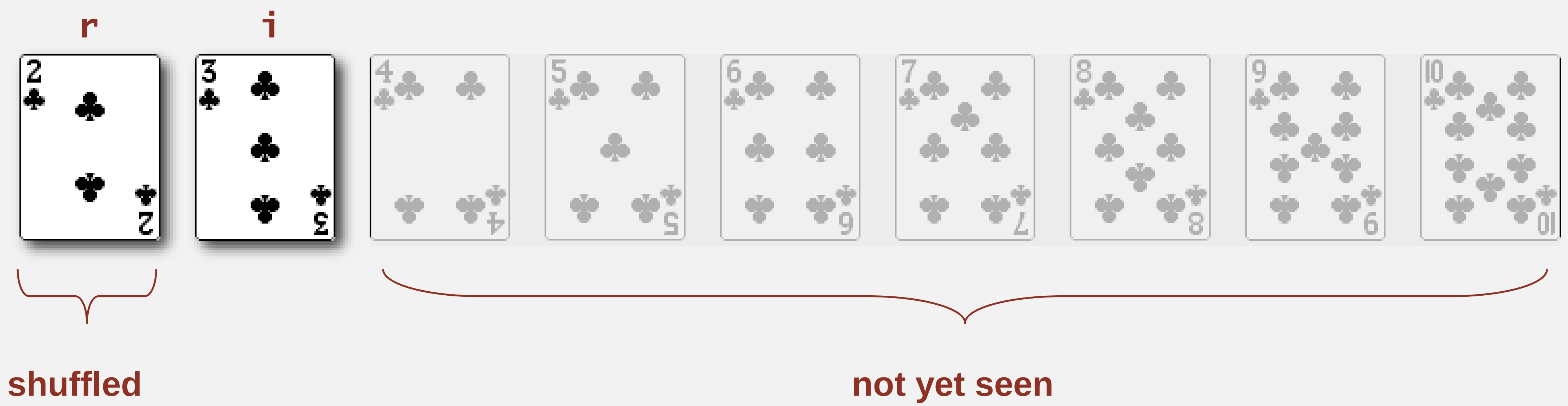
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



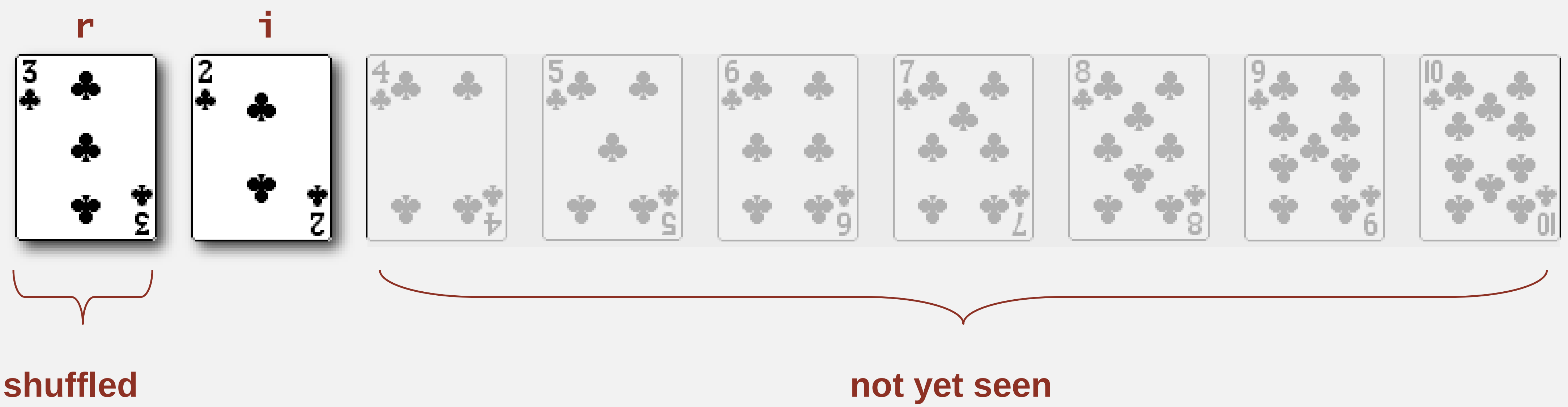
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



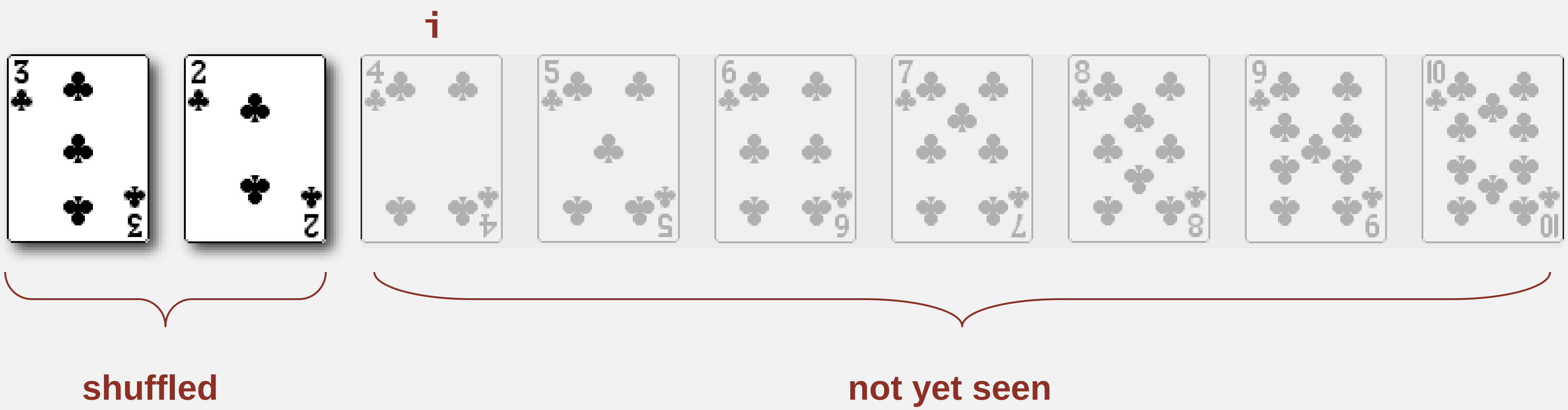
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



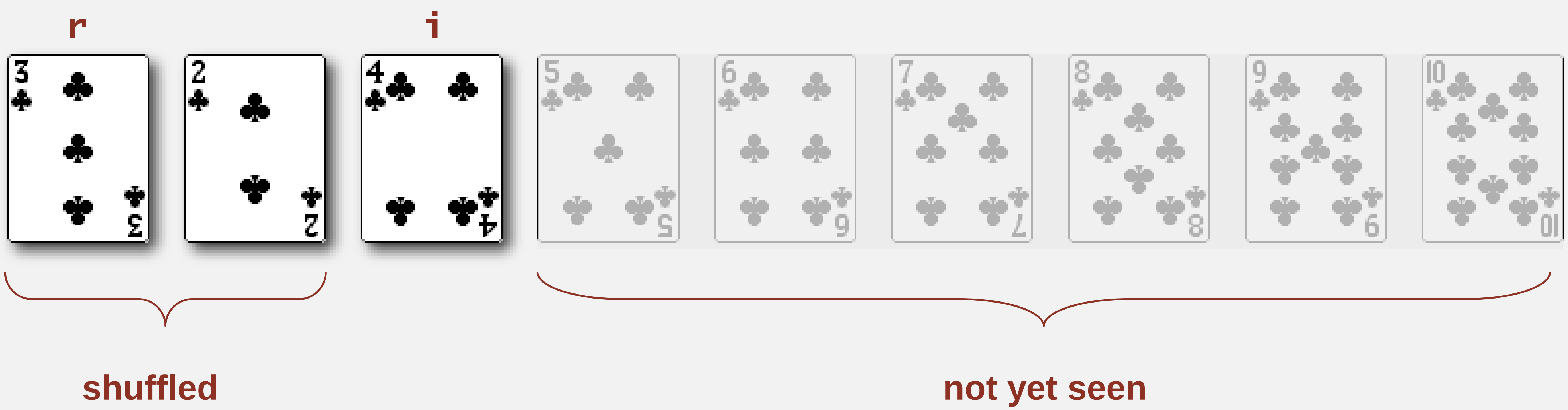
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



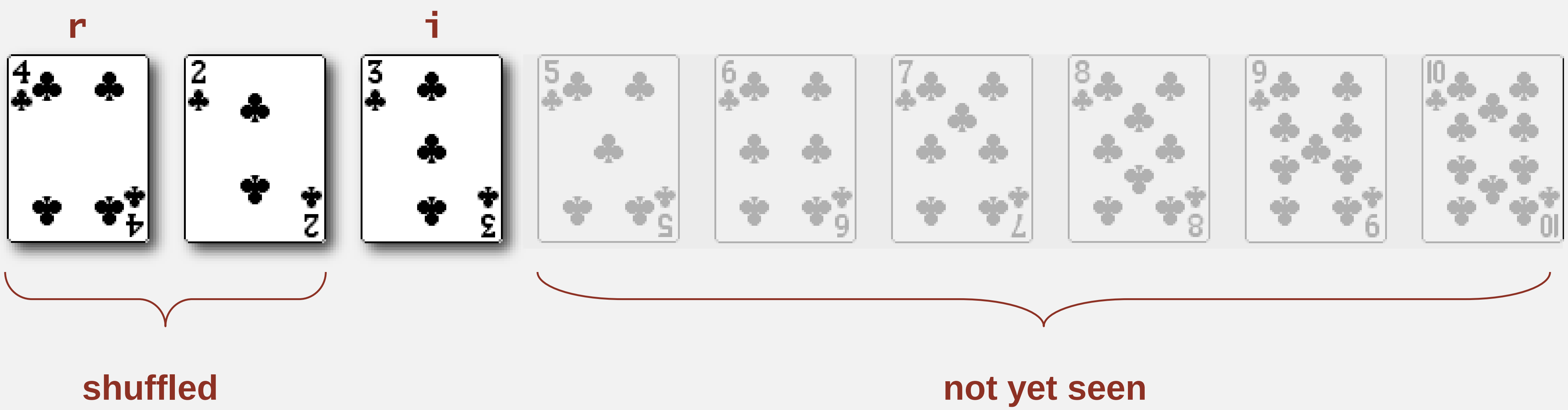
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



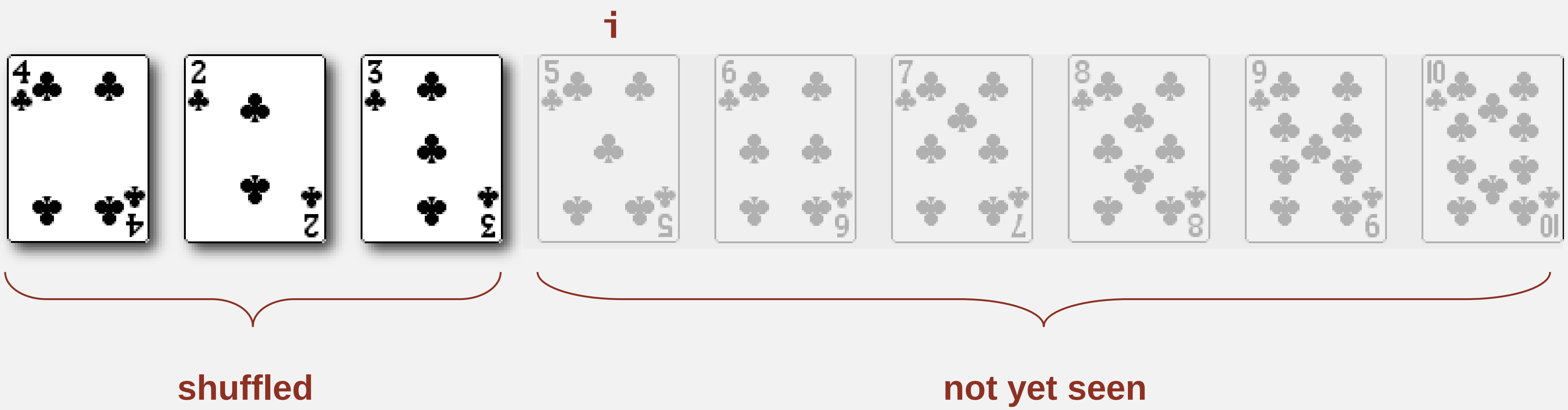
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



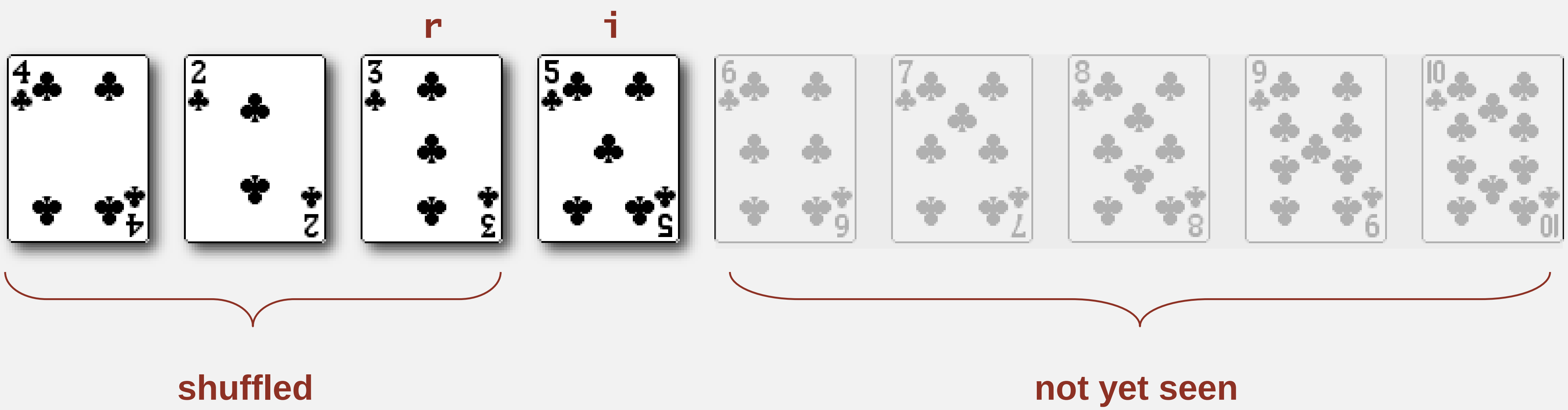
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



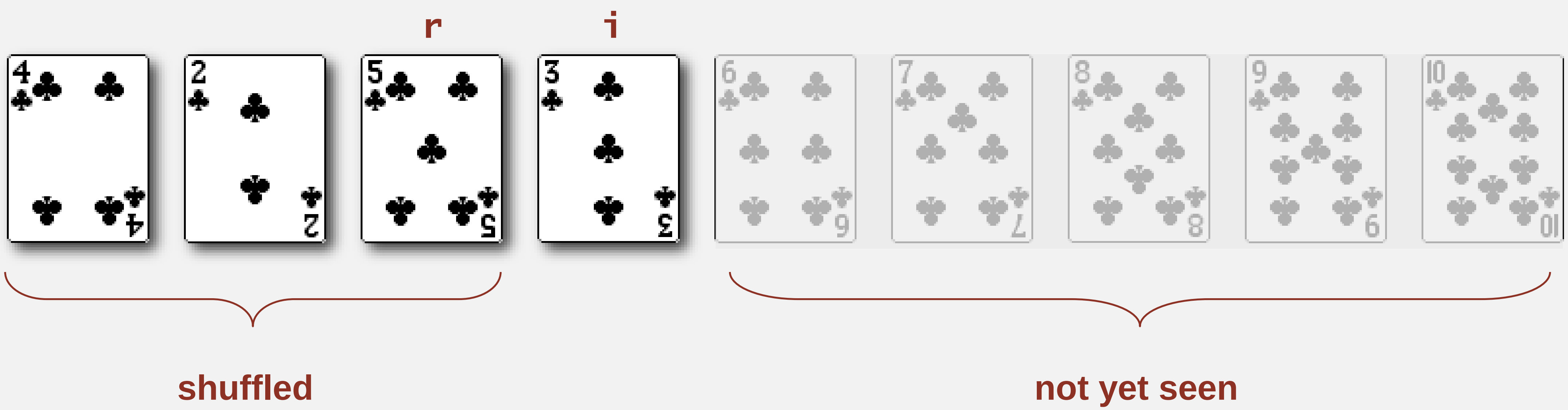
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



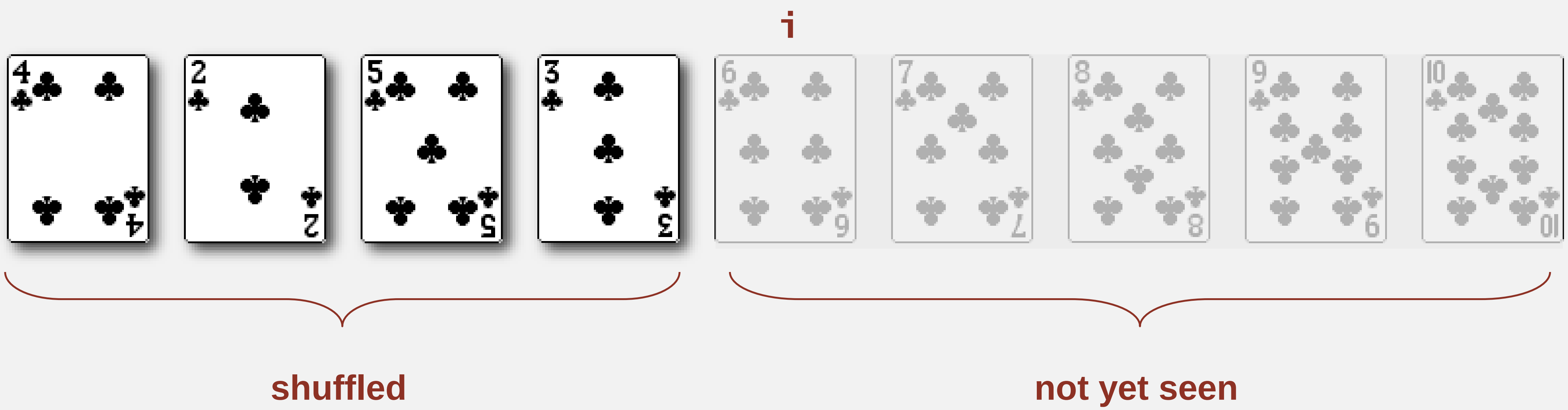
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



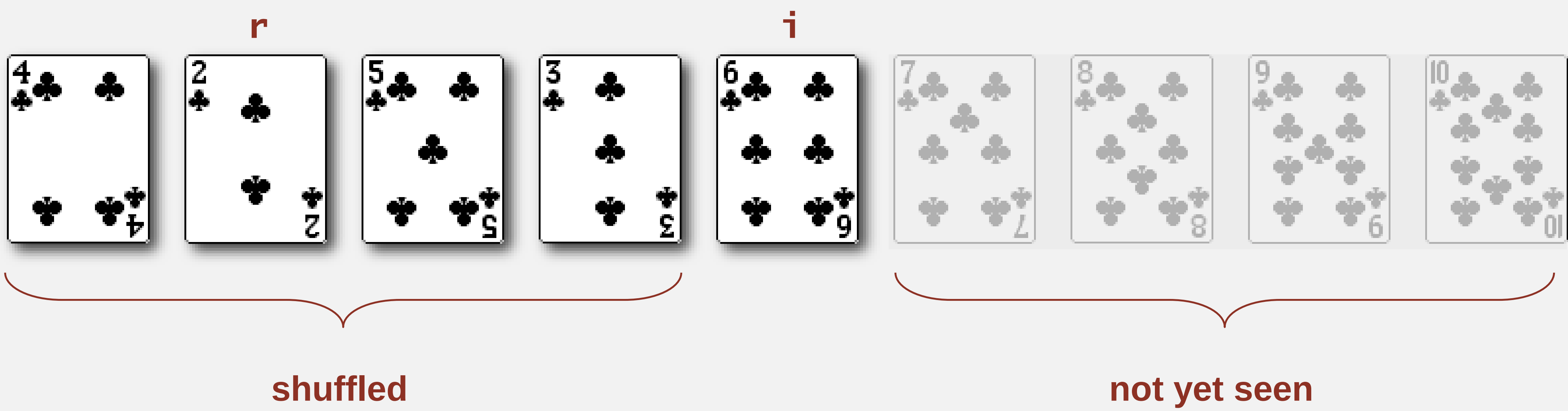
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



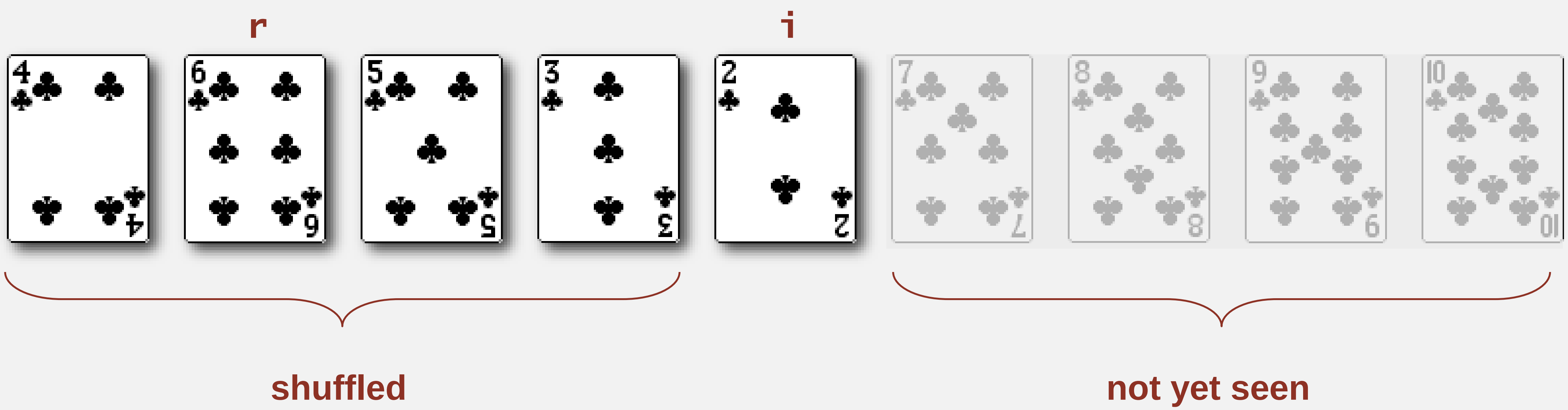
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



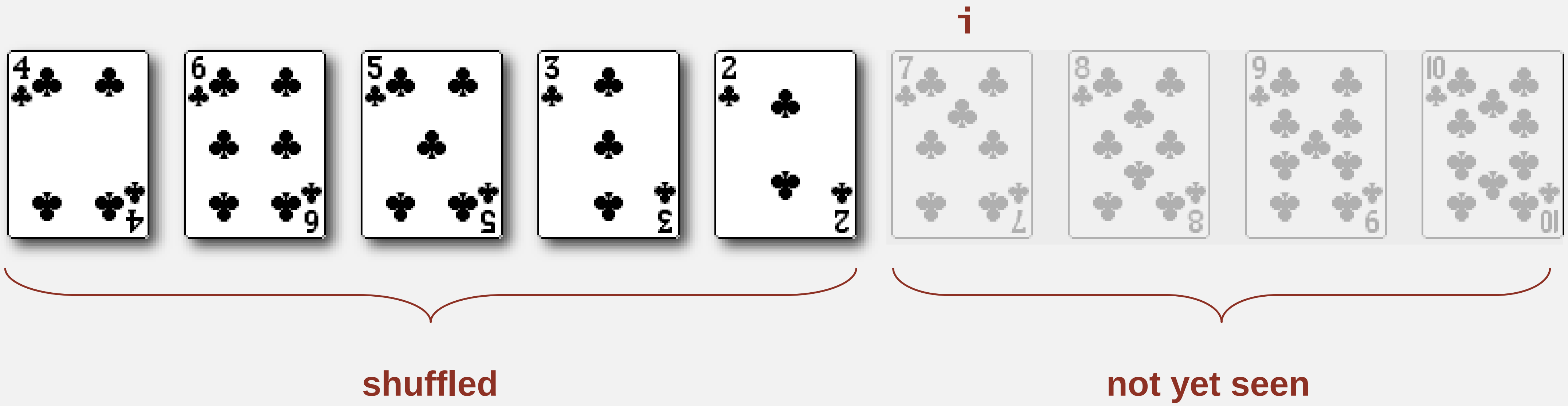
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



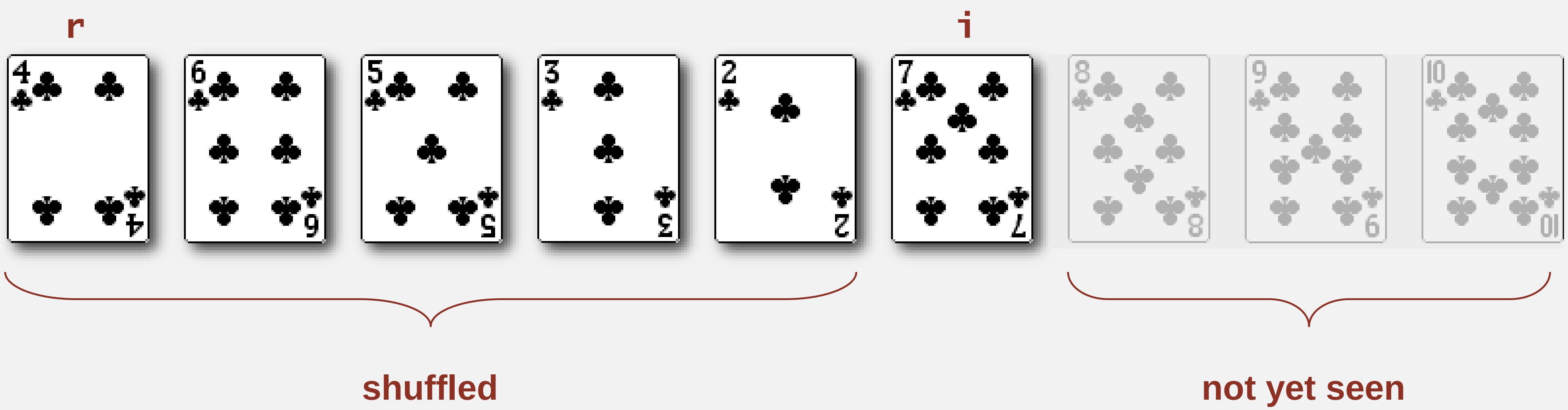
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



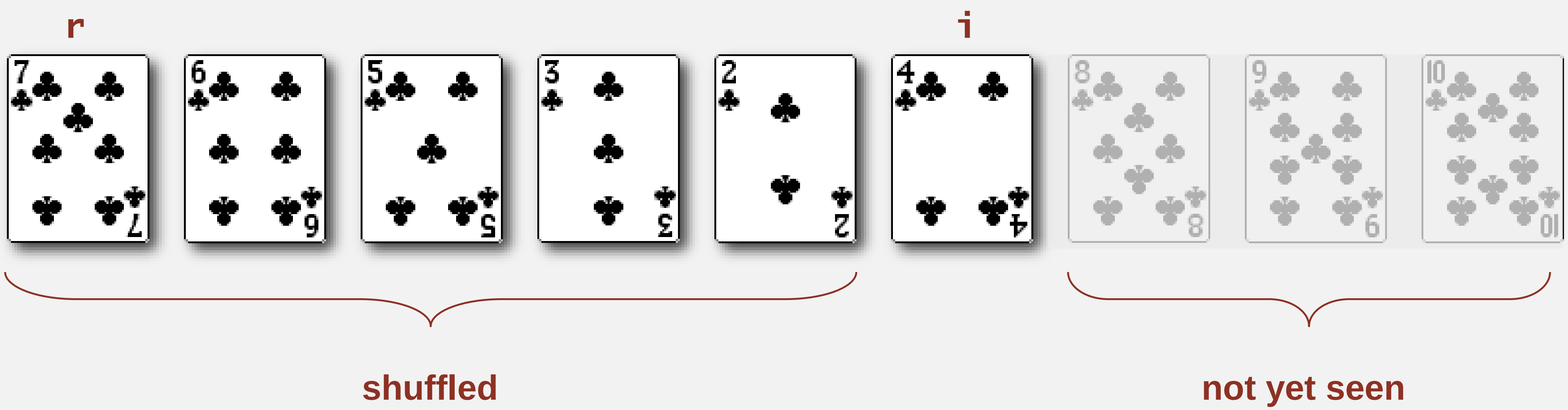
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



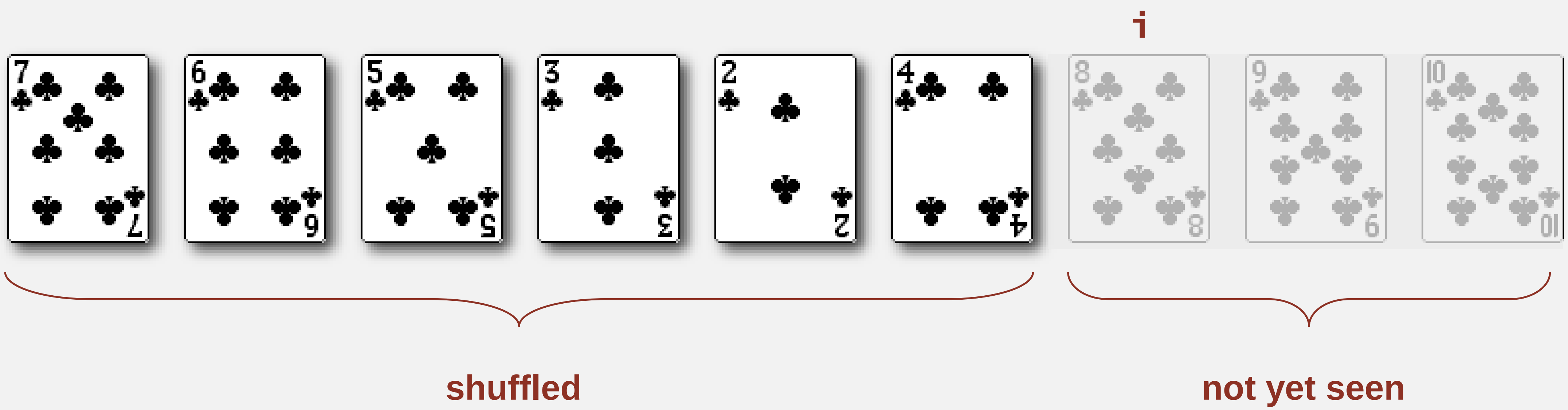
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



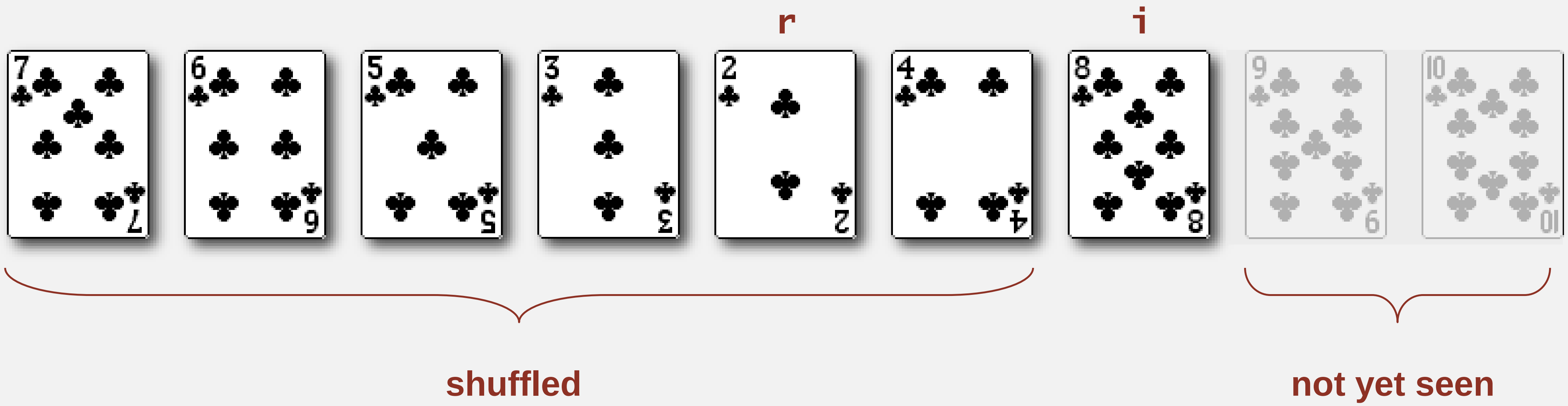
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



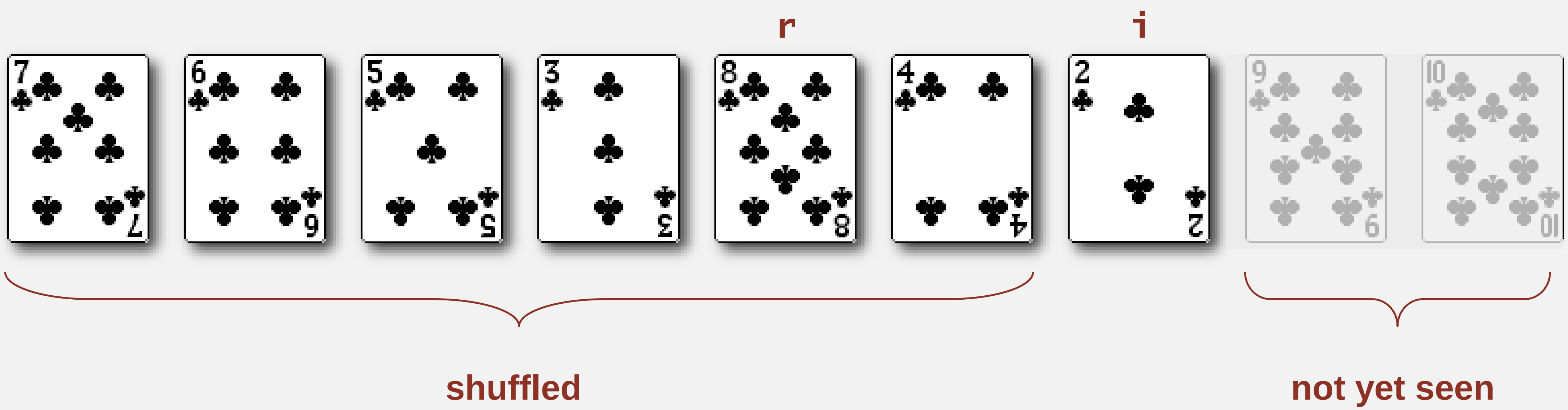
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



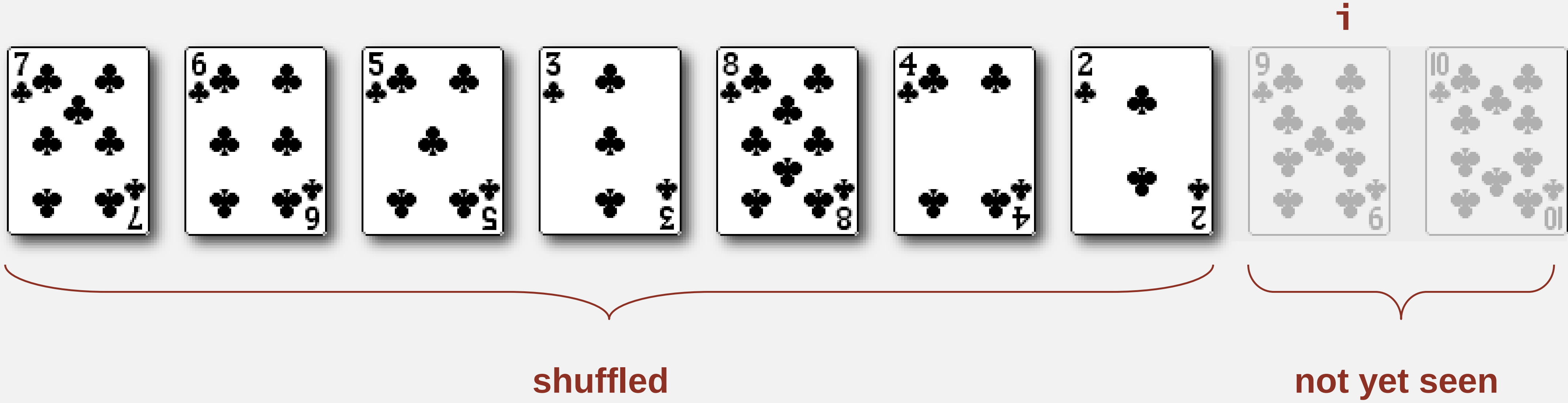
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



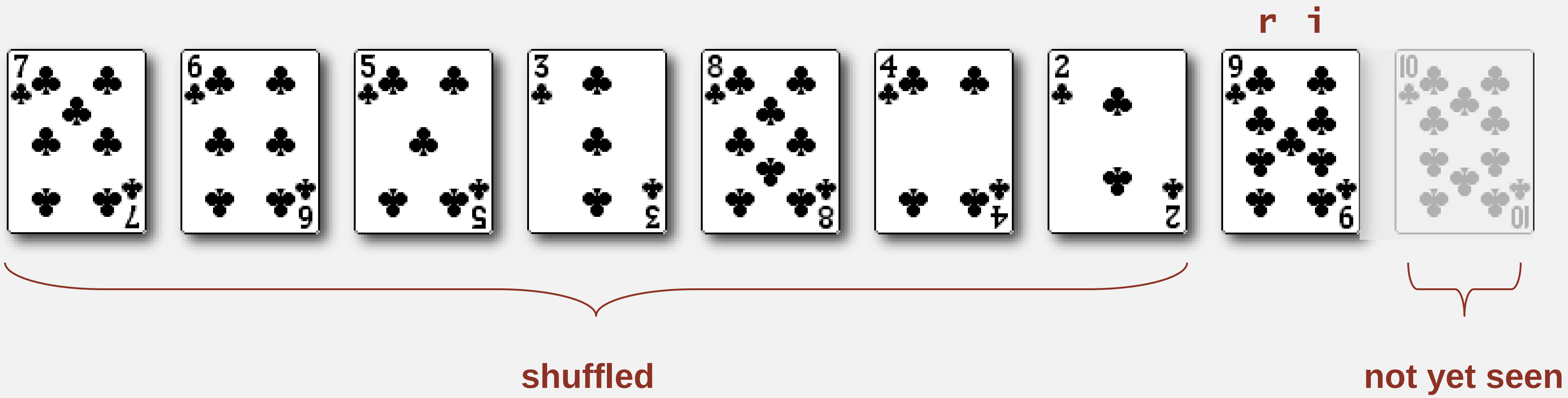
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



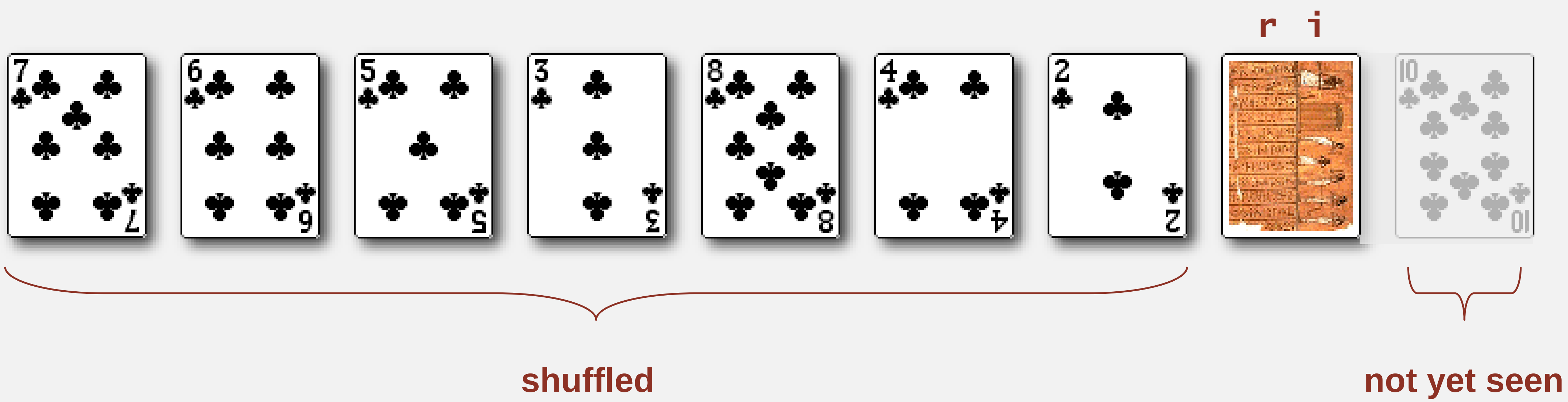
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



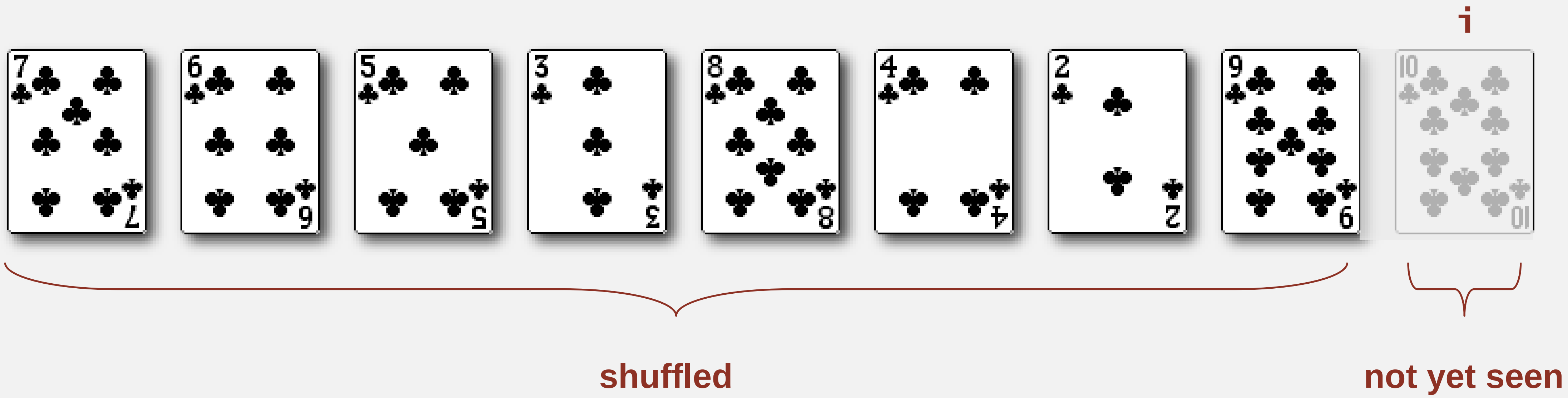
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



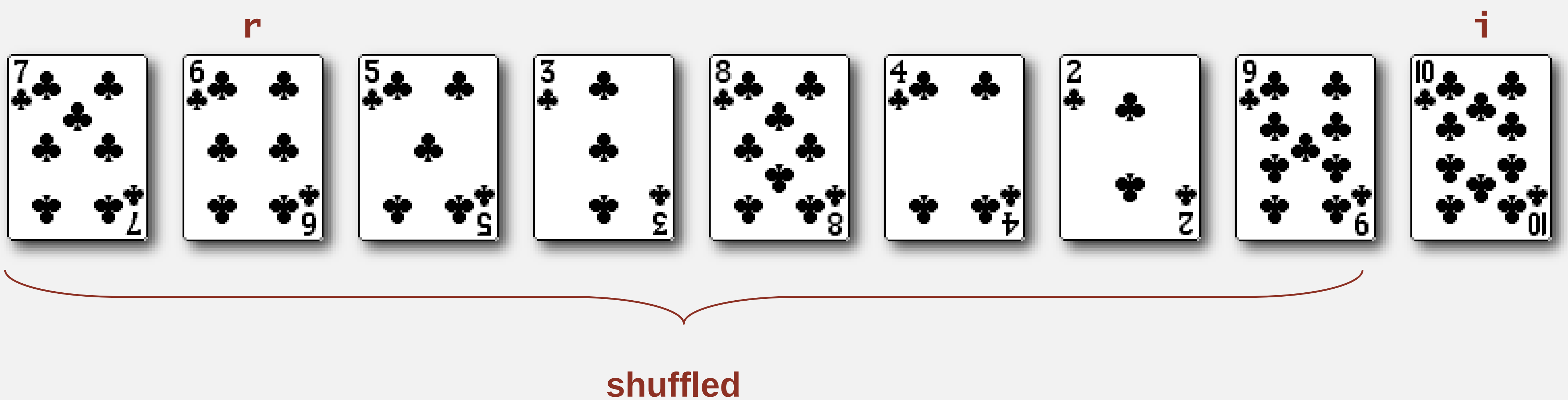
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



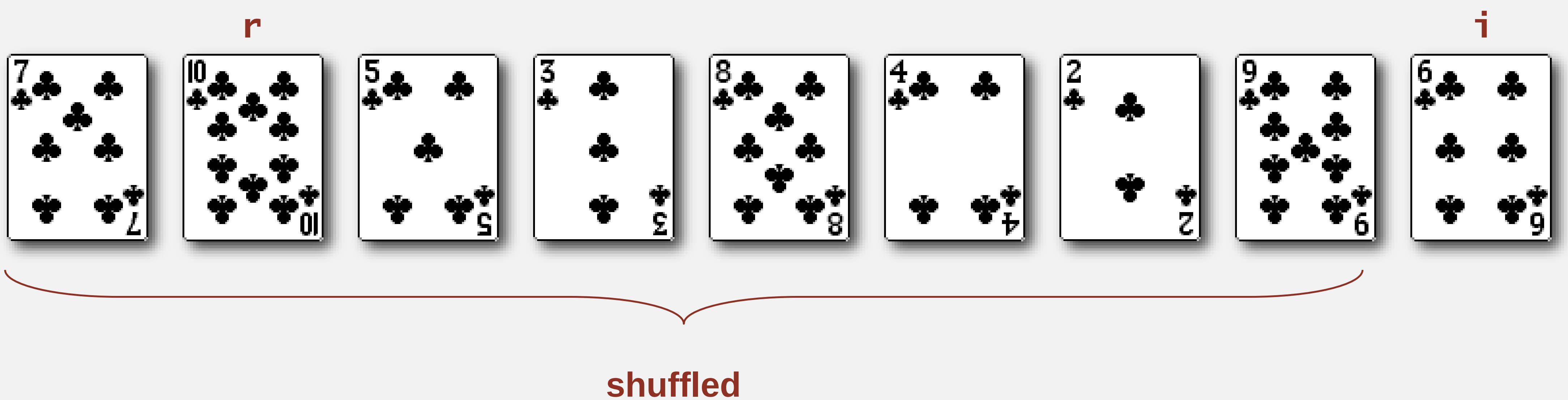
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



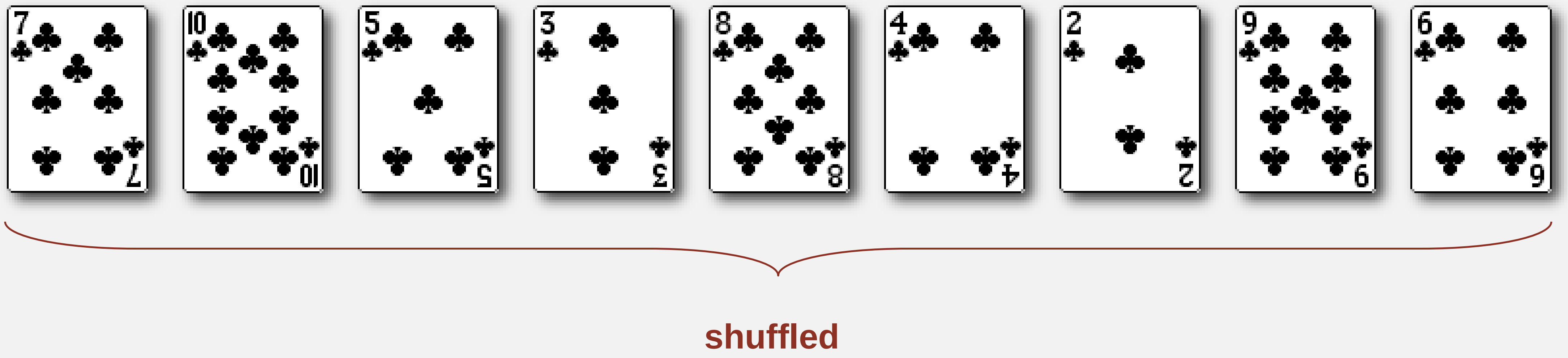
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



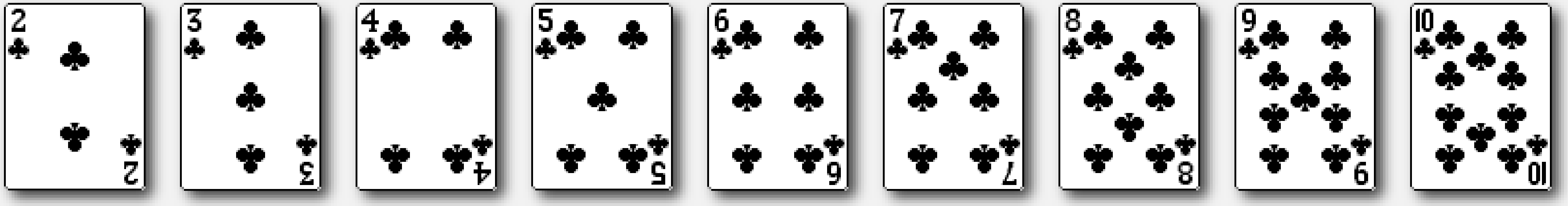
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Proposition. [Fisher-Yates 1938] Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time.

assuming integers uniformly at random

Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.

```
public class StdRandom
{
   ...
   public static void shuffle(Object[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int r = StdRandom.uniform(i + 1);
         exch(a, i, r);
      }
   }
}
```

between 0 and i

http://algs4.cs.princeton.edu

MERGESORT AND QUICKSORT

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [this lecture]

...

Quicksort. [next lecture]

...

http://algs4.cs.princeton.edu

Mergesort and quicksort

- *mergesort*
- *comparators*
- *stability*
- *quicksort*

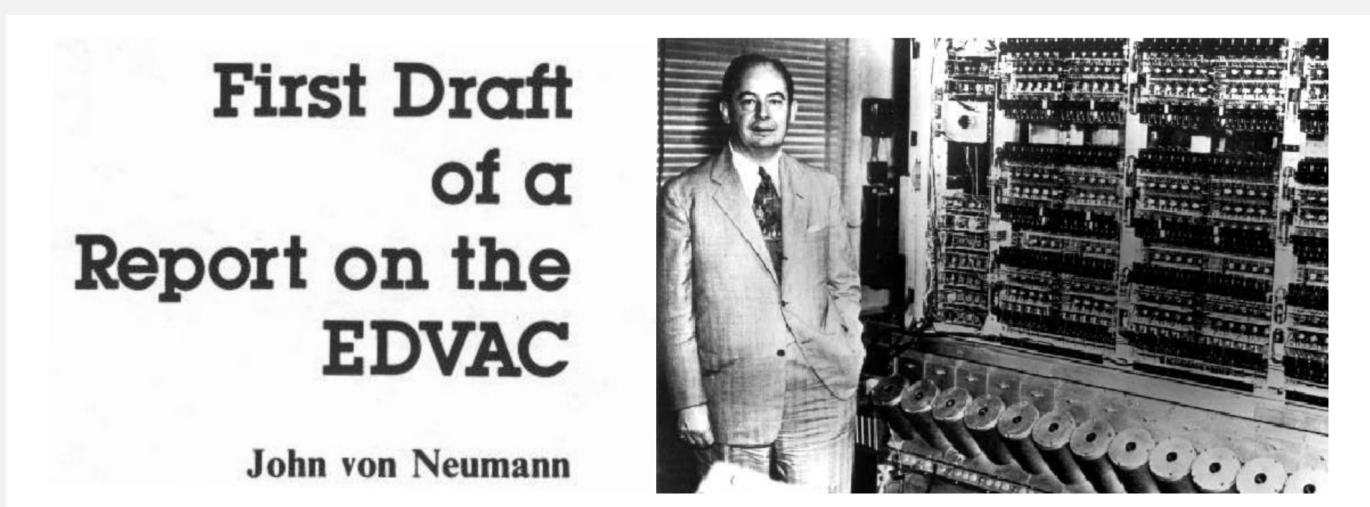
Mergesort

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

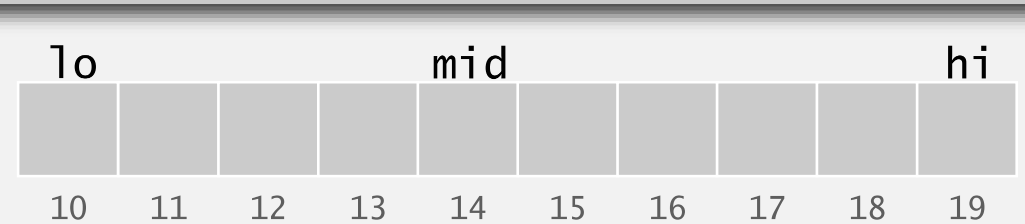
input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview



Mergesort: Java implementation

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }
     private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }
   public static void sort(Comparable[] a)
   {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

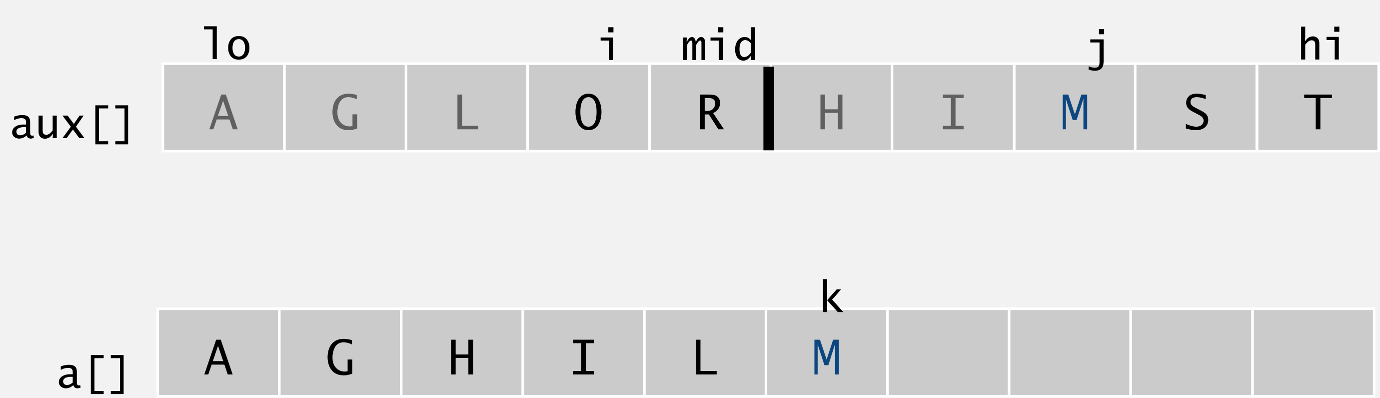


Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

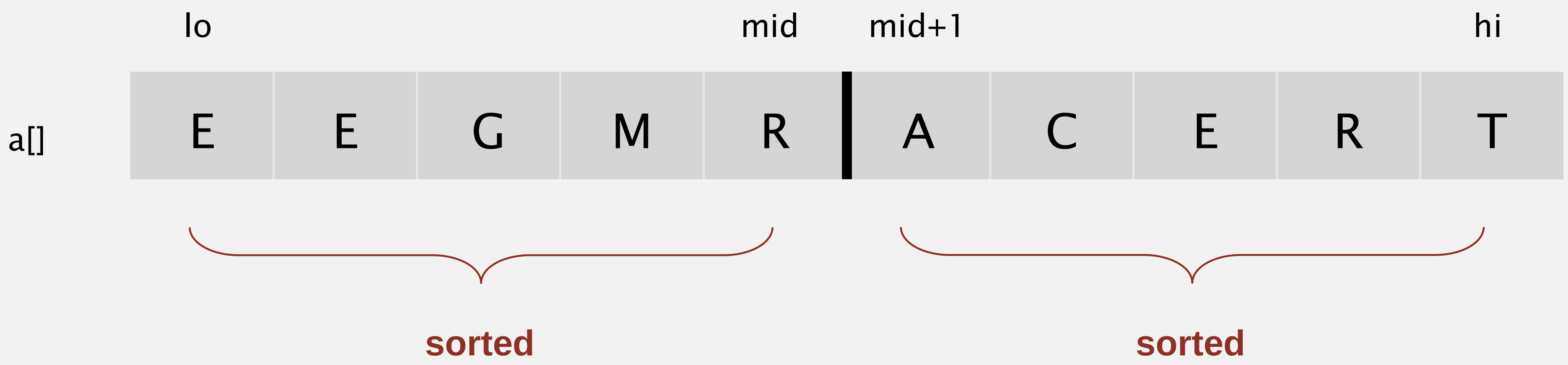
copy

merge



Abstract in-place merge demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

a[]	E	E	G	M	R	A	C	E	R	T

aux[]	E	E	G	M	R	A	C	E	R	T

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

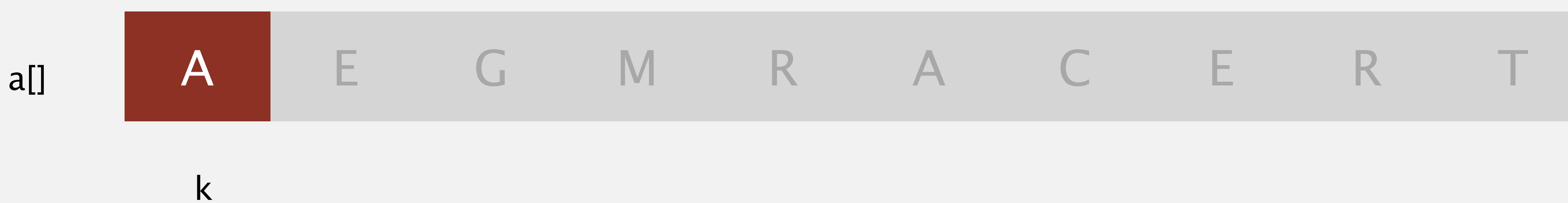
a[]	E	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



compare minimum in each subarray

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

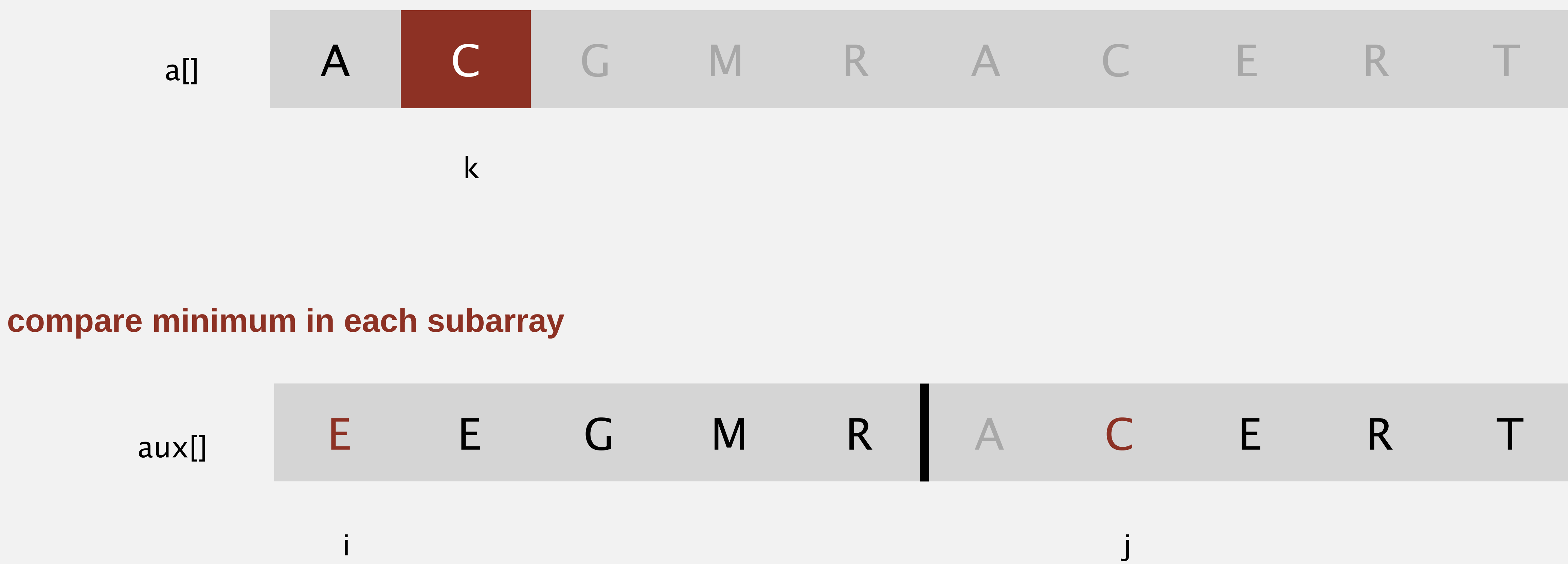
a[]										
	A	E	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]										
	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

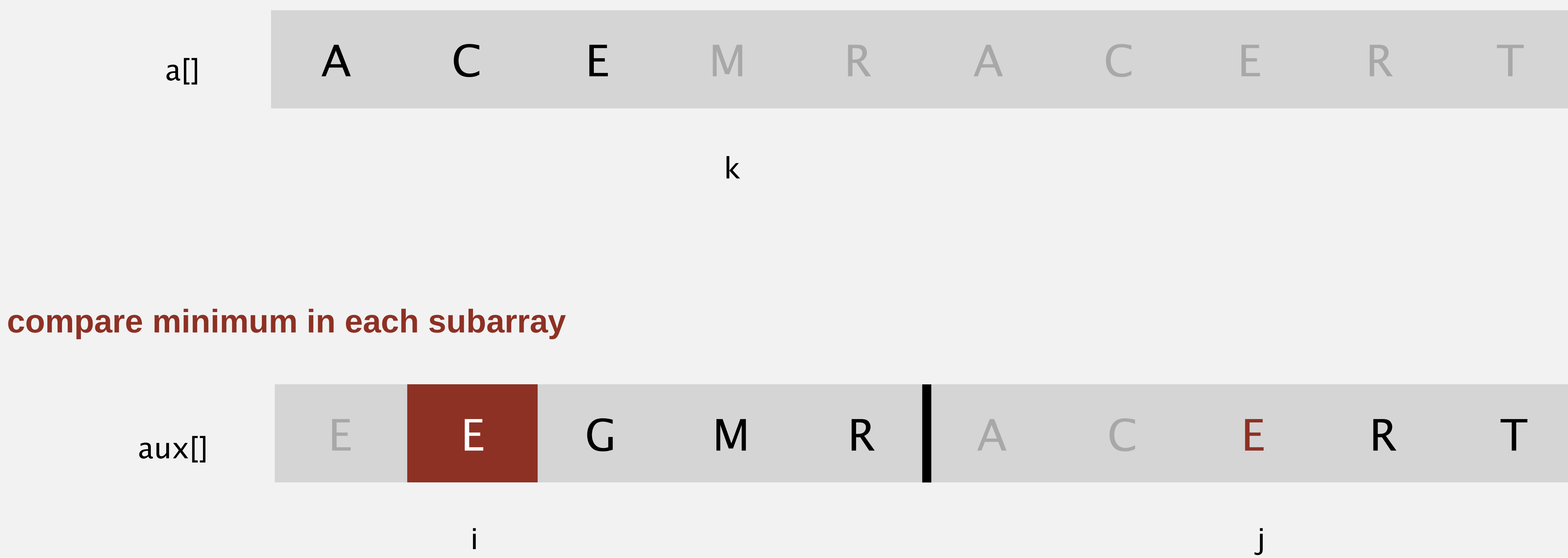
a[]	A	C	E	M	R	A	C	E	R	T
			k							

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i							j		

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

a[]	A	C	E	E	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

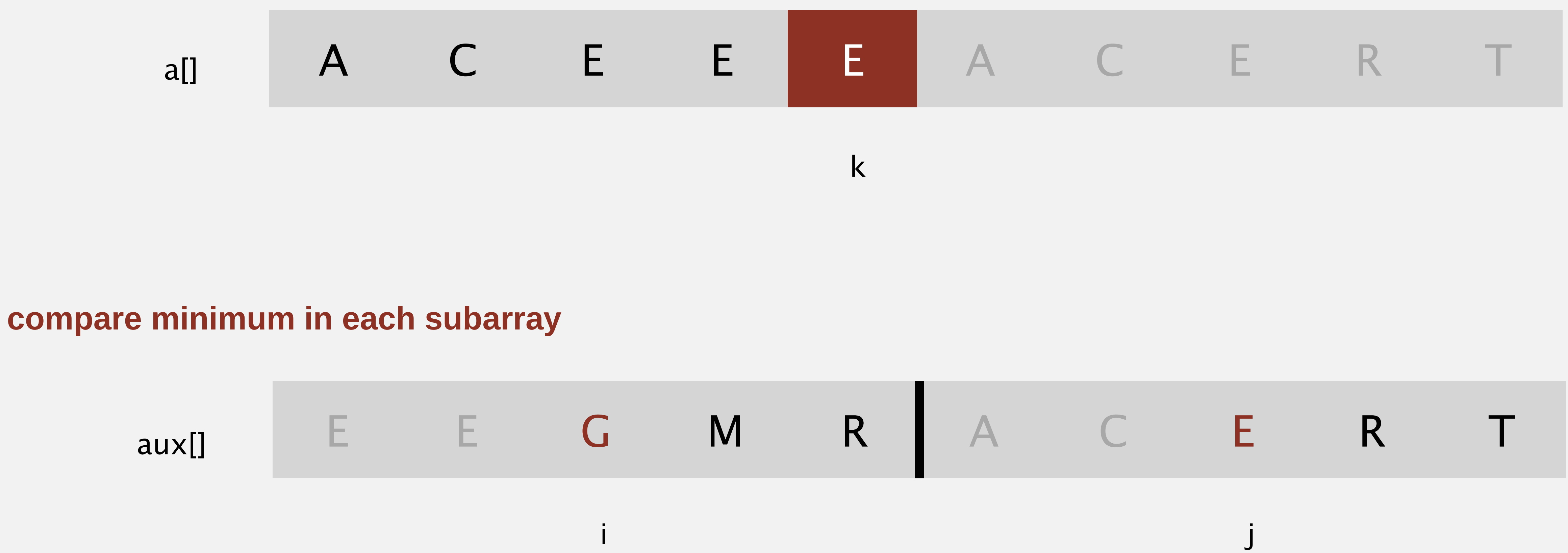
a[]	A	C	E	E	R	A	C	E	R	T
					k					

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i					j		

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

a[]	A	C	E	E	E	A	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

a[]	A	C	E	E	E	G	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

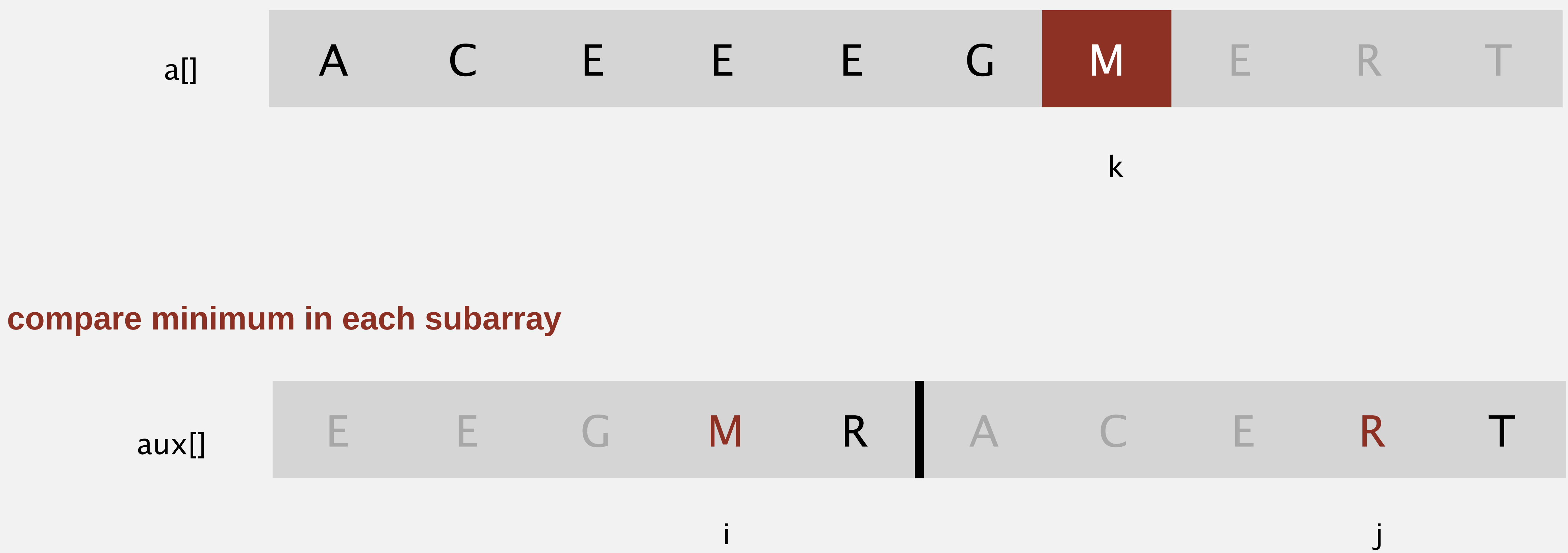
a[]	A	C	E	E	E	G	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

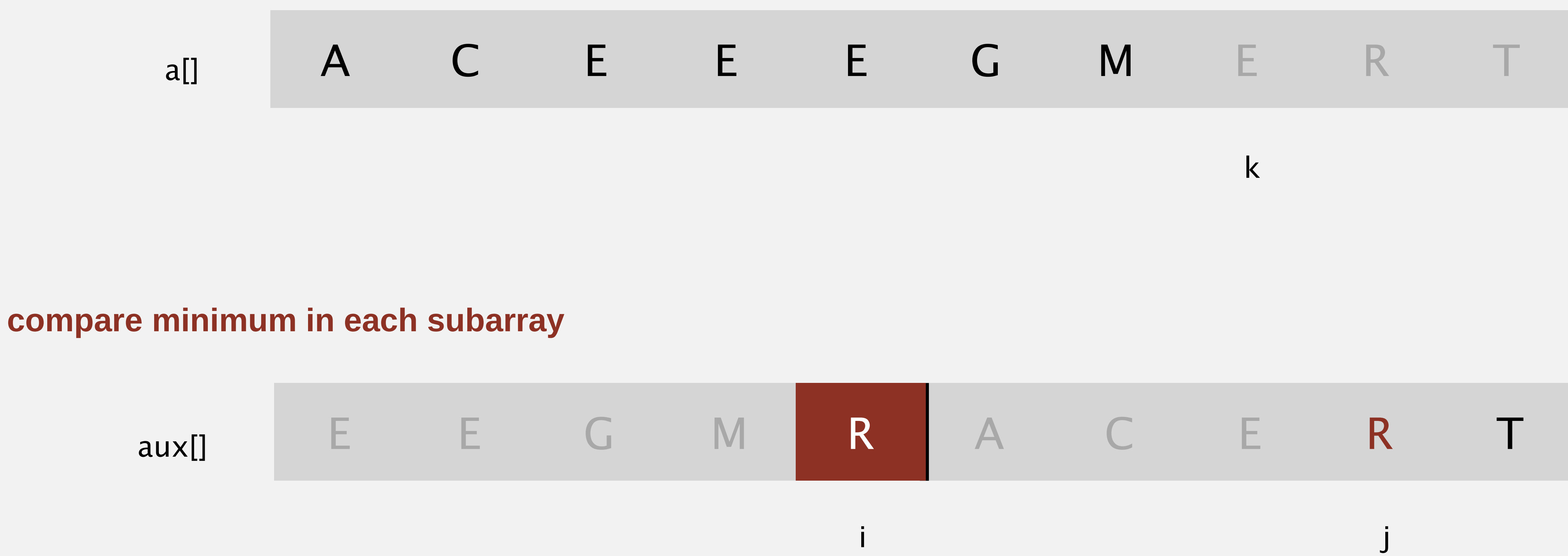
Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



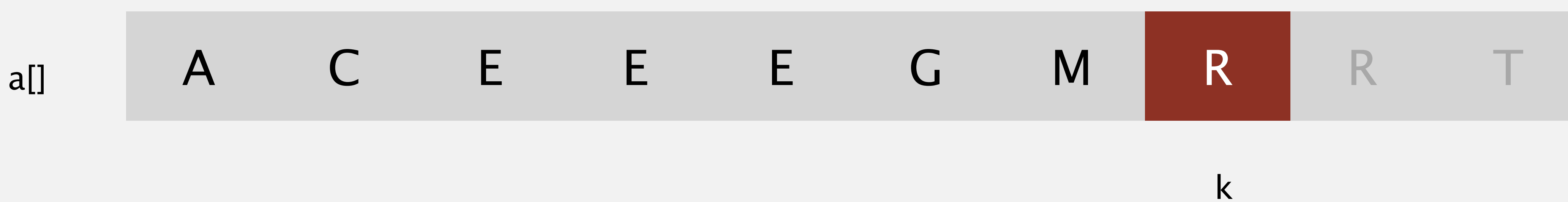
Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



compare minimum in each subarray

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

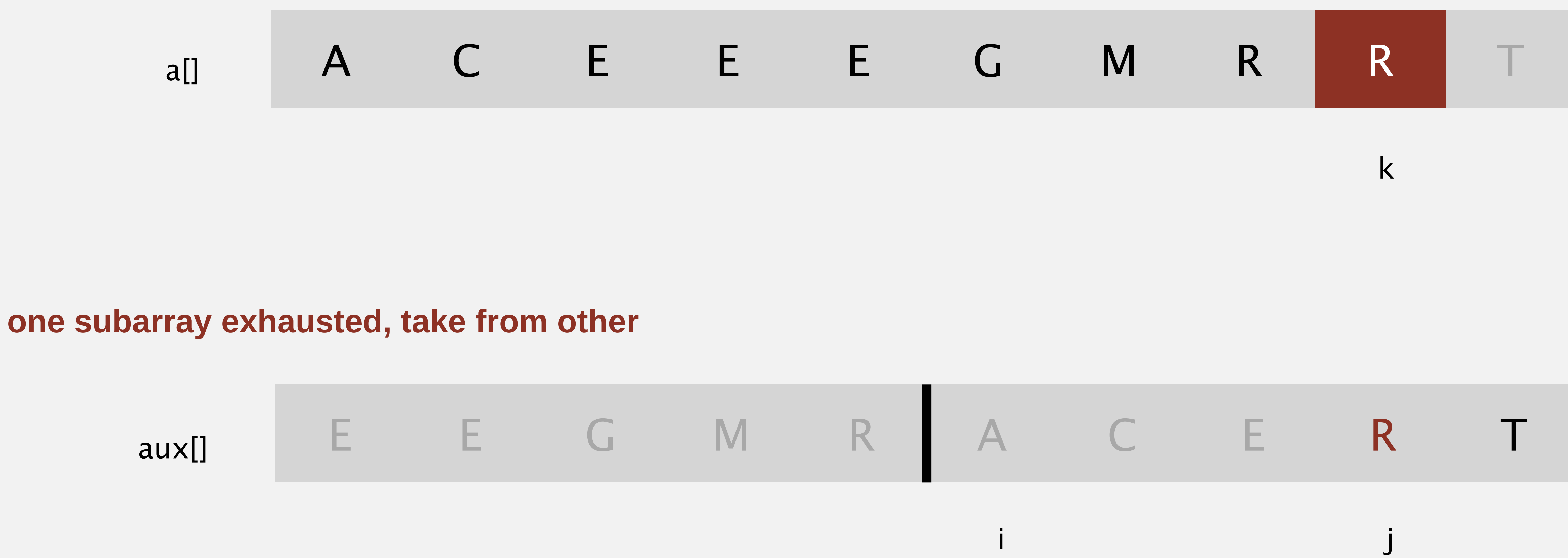
a[]	A	C	E	E	E	G	M	R	R	T
									k	

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i			j	

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

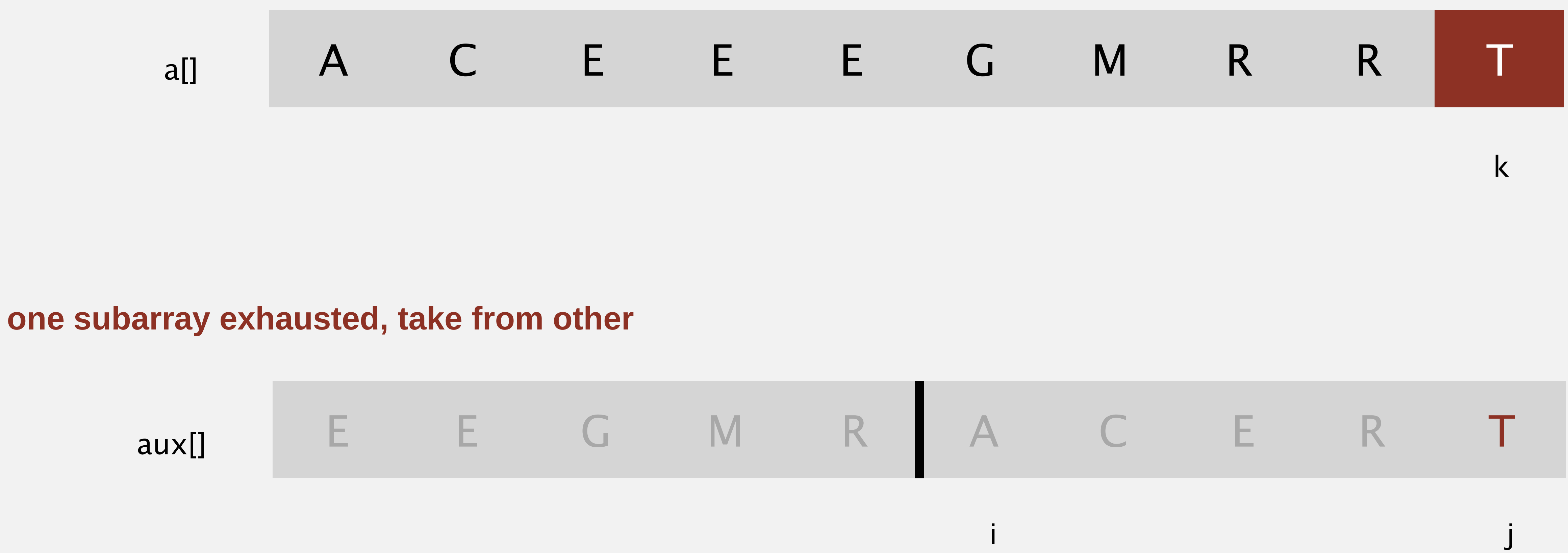
a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merging demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

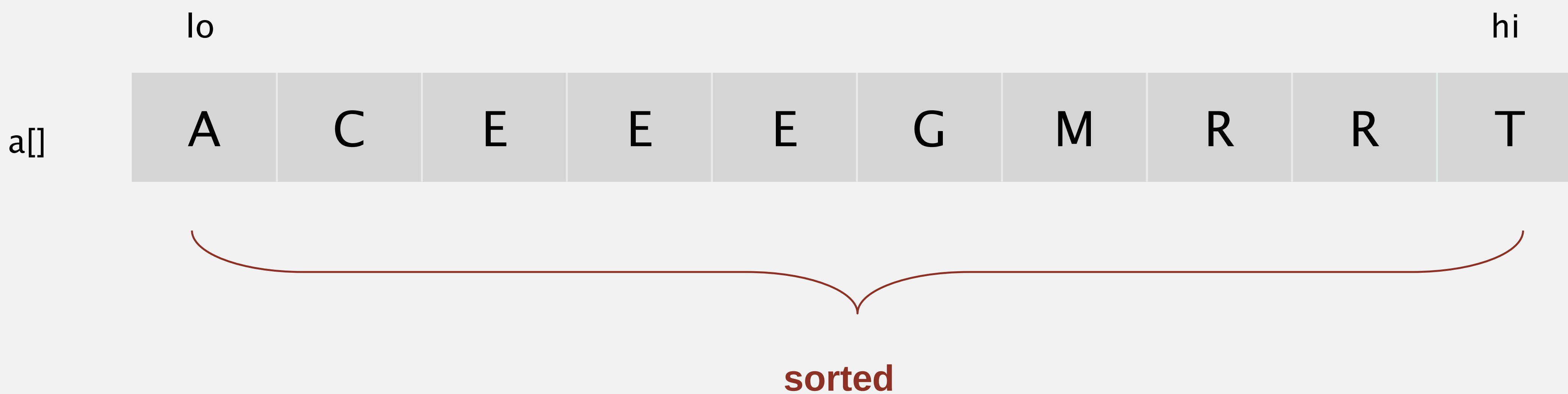
a[]	A	C	E	E	E	G	M	R	R	T	
											k

both subarrays exhausted, done

aux[]	E	E	G	M	R	A	C	E	R	T	
						i					j

Abstract in-place merge demo

Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

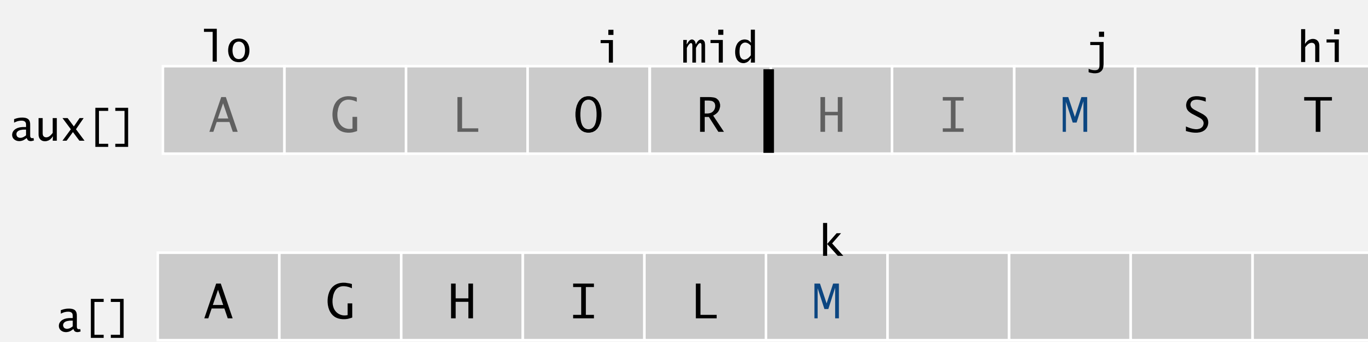


Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

}
```

copy

merge



Mergesort: trace

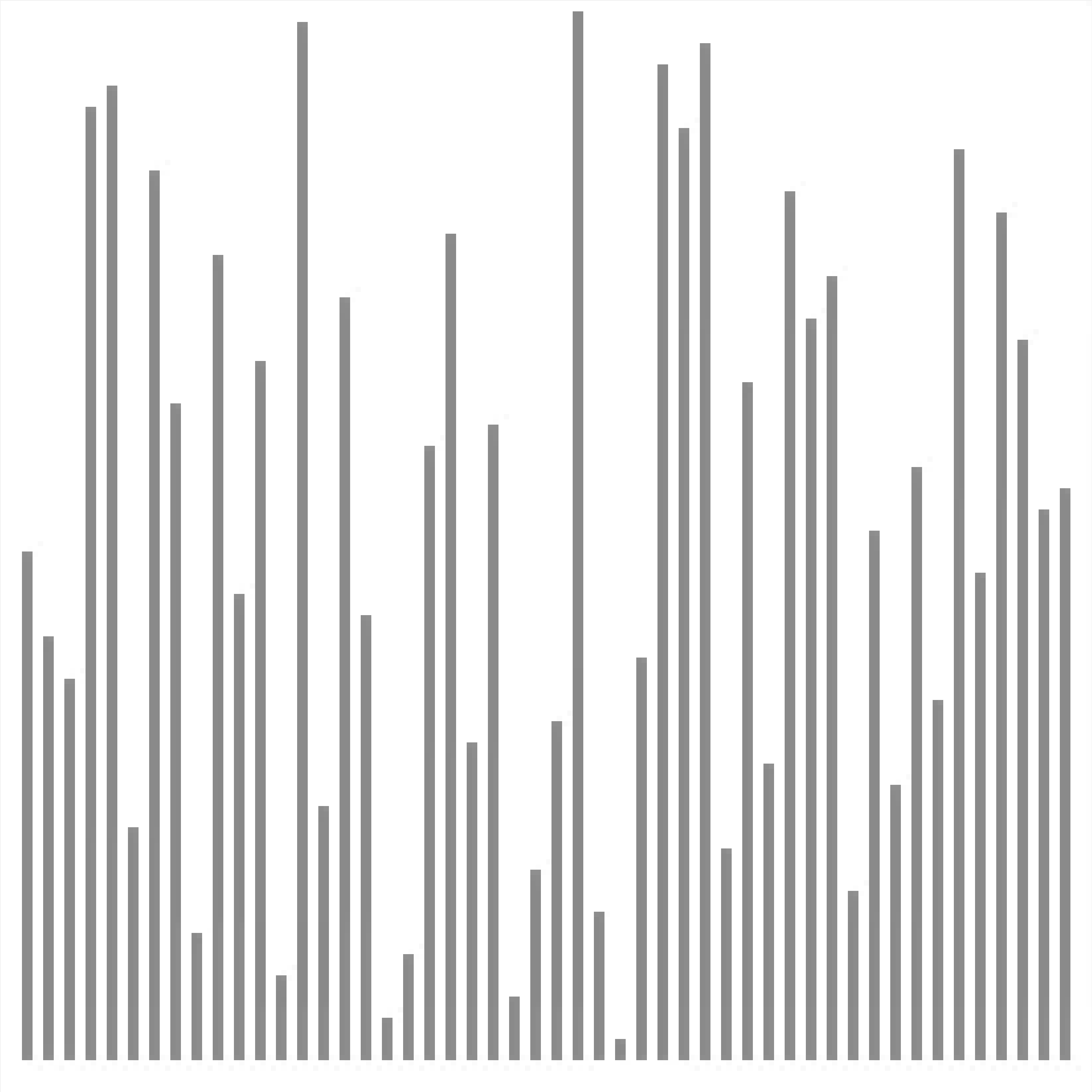
a[]

lo / hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
`merge(a, aux, 0, 3, 7)`	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
`merge(a, aux, 8, 9, 11)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, aux, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
`merge(a, aux, 0, 7, 15)`	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

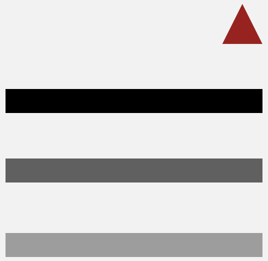
result after recursive call

Mergesort: animation

50 random items

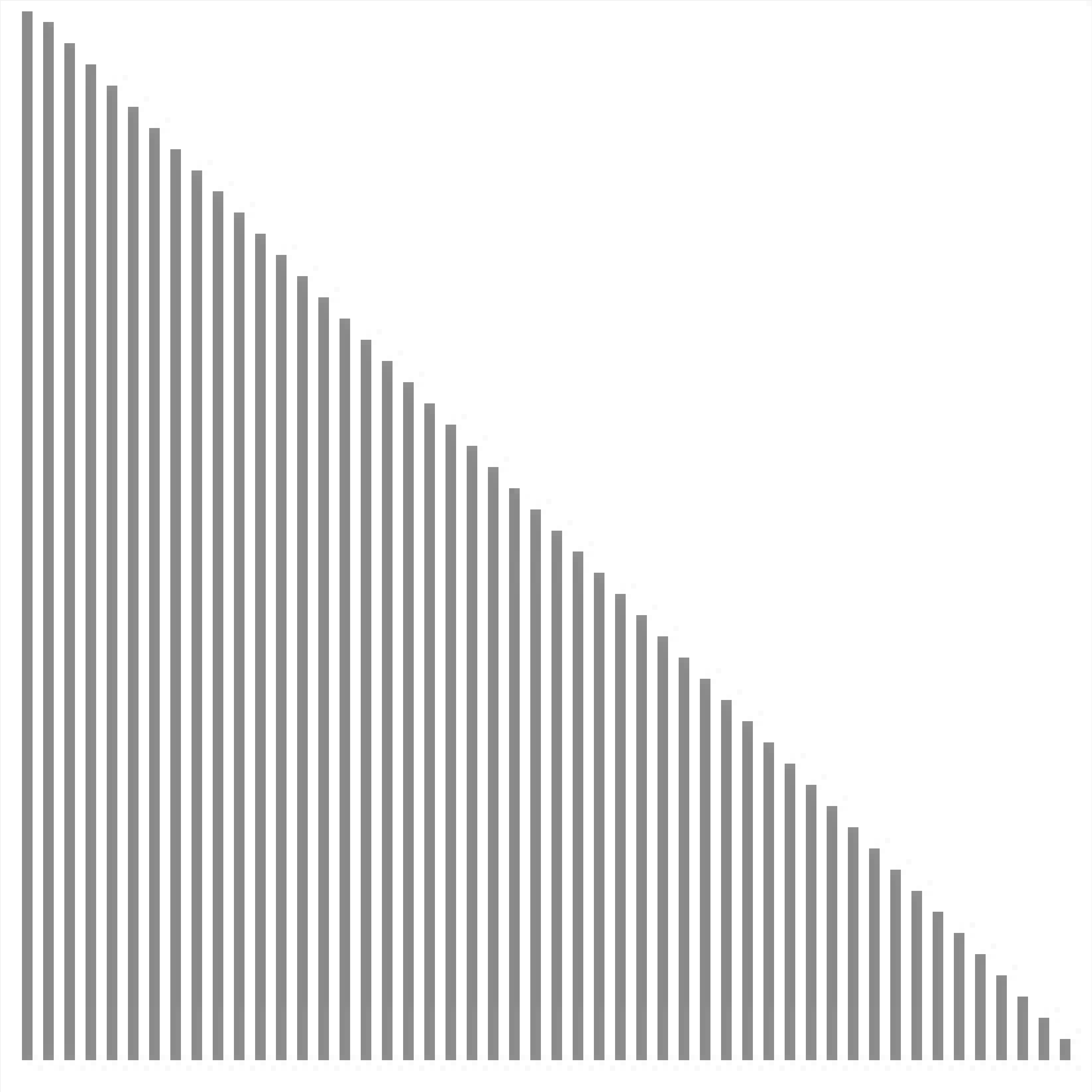


http://www.sorting-algorithms.com/merge-sort

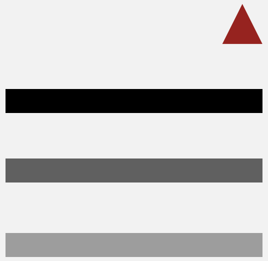


Mergesort: animation

50 reverse-sorted items



http://www.sorting-algorithms.com/merge-sort



Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N}_{\text{merge}} \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

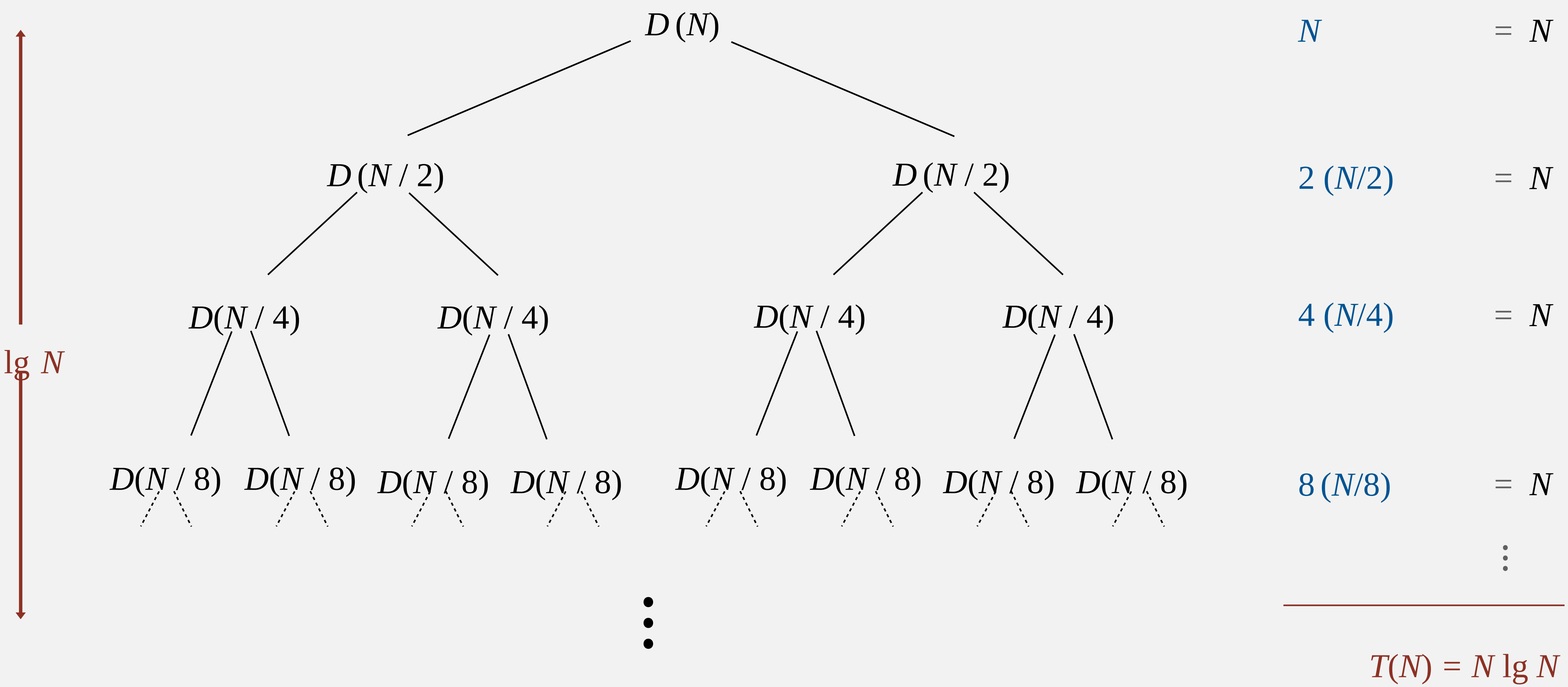
We solve the recurrence when N is a power of 2: ← result holds for all N (analysis cleaner in this case)

$$D(N) = 2\,D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of array accesses

Proposition. Mergesort uses $\leq 6 N \lg N$ array accesses to sort an array of length N.

Pf sketch. The number of array accesses $A(N)$ satisfies the recurrence:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0.$$

Key point. Any algorithm with the following structure takes $N \log N$ time:

```
public static void linearithmic(int N)
{
    if (N == 0) return;
    linearithmic(N/2);
    linearithmic(N/2);
    linear(N);
}
```

solve two problems of half the size

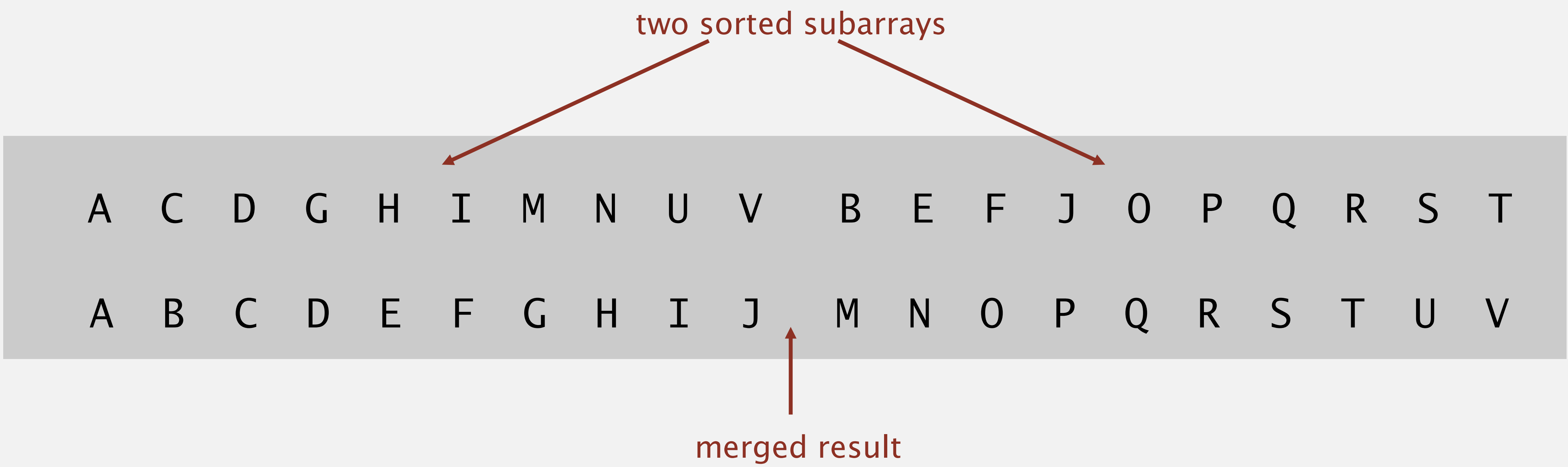
do a linear amount of work

Notable examples. FFT, hidden-line removal, Kendall-tau distance, …

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array `aux[]` needs to be of length N for the last merge.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort.

Challenge 1 (not hard). Use `aux[]` array of length $\sim \frac{1}{2} N$ instead of N.

Challenge 2 (very hard). In-place merge. [Kronrod 1969]

Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

Stop if already sorted.

- Is largest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

http://algs4.cs.princeton.edu

Mergesort and quicksort

- *mergesort*
- *comparators*
- *stability*
- *quicksort*

Sort countries by gold medals

NOC	Gold	Silver	Bronze	Total
United States (USA)	46	29	29	104
China (CHN)§	38	28	22	88
Great Britain (GBR)*	29	17	19	65
Russia (RUS)§	24	25	32	81
South Korea (KOR)	13	8	7	28
Germany (GER)	11	19	14	44
France (FRA)	11	11	12	34
Italy (ITA)	8	9	11	28
Hungary (HUN)§	8	4	6	18
Australia (AUS)	7	16	12	35

Sort countries by total medals

NOC	Gold	Silver	Bronze	Total
United States (USA)	46	29	29	104
China (CHN)§	38	28	22	88
Russia (RUS)§	24	25	32	81
Great Britain (GBR)*	29	17	19	65
Germany (GER)	11	19	14	44
Japan (JPN)	7	14	17	38
Australia (AUS)	7	16	12	35
France (FRA)	11	11	12	34
South Korea (KOR)	13	8	7	28
Italy (ITA)	8	9	11	28

Comparator interface

Comparator interface: sort using an alternate order.

public interface Comparator<Key>		
int	compare(Key v, Key w)	*compare keys v and w*

Required property. Must be a total order.

string order	example
natural order	`Now is the time`
case insensitive	`is Now the time`
Spanish language	`café cafetero cuarto churro nube ñoño`
British phone book	`McKinley Mackintosh`

pre-1994 order for digraphs ch and ll and rr

Comparator interface: system sort

To use with Java system sort:

- Create `Comparator` object.
- Pass as second argument to `Arrays.sort()`.

uses natural order

uses alternate order defined by Comparator<String> object

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
...
```

Bottom line. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
public class Student
{
   private final String name;
   private final int section;
   ...

   public static class ByName implements Comparator<Student>
   {
     public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }
   public static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
Arrays.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Arrays.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
public class Student
{
   public static final Comparator<Student> BY_NAME    = new ByName();
   public static final Comparator<Student> BY_SECTION = new BySection();
   private final String name;
   private final int section;
   ...
   private static class ByName implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }
   private static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

one Comparator for the class

this technique works here since no danger of overflow

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.
- Provide access to `Comparator`.

`Arrays.sort(a, Student.BY_NAME);`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

`Arrays.sort(a, Student.BY_SECTION);`

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

http://algs4.cs.princeton.edu

Mergesort and quicksort

- mergesort
- comparators
- *stability*
- quicksort

Stability

A typical application. First, sort by name; then sort by section.

```
Selection.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Selection.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

Stability

Q. Which sorts are stable?

A. Need to check algorithm (and implementation).

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Stability: selection sort

Proposition. Selection sort is not stable.

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }
}
```

i	min	0	1	2
0	2	B_1	B_2	A_3
1	1	A_3	B_2	B_1
2	2	A_3	B_2	B_1
		A_3	B_2	B_1

Pf by counterexample. Long-distance exchange can move one equal item past another one.

Stability: insertion sort

Proposition. Insertion sort is stable.

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
   }
}
```

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Pf. Equal items never move past each other.

Stability: mergesort

Proposition. Mergesort is stable.

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }
   public static void sort(Comparable[] a)
   {  /* as before */  }
}
```

Pf. Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is stable.

```
private static void merge(...)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

0	1	2	3	4
A_1	A_2	A_3	B	D

5	6	7	8	9	10
A_4	A_5	C	E	F	G

Pf. Takes from left subarray if equal keys.

http://algs4.cs.princeton.edu

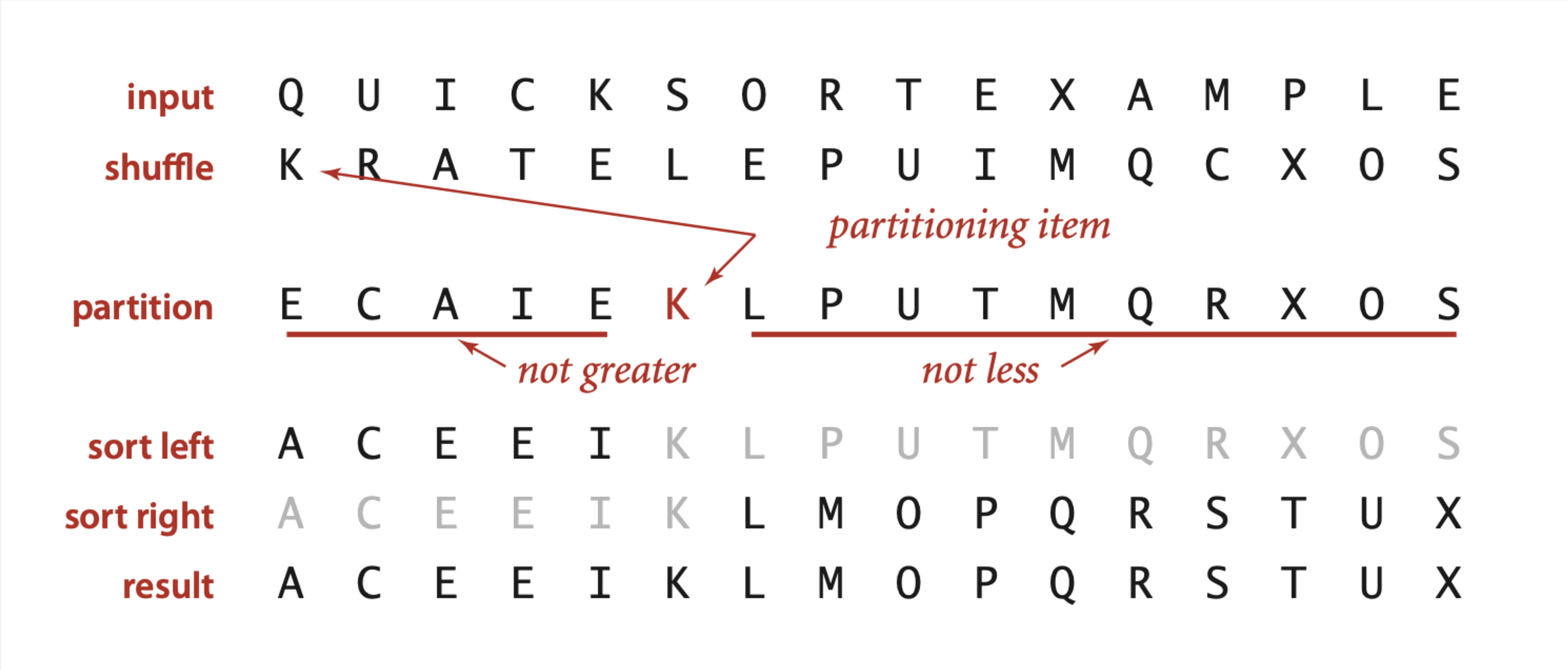
Mergesort and quicksort

- *mergesort*
- *comparators*
- *stability*
- *quicksort*

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some `j`
  - entry `a[j]` is in place
  - no larger entry to the left of `j`
  - no smaller entry to the right of `j`
- Sort each subarray recursively.



Quicksort: Java implementation

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
  }
}
```

shuffle needed for performance guarantee (stay tuned)

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
↑	↑														↑
lo	i														j

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

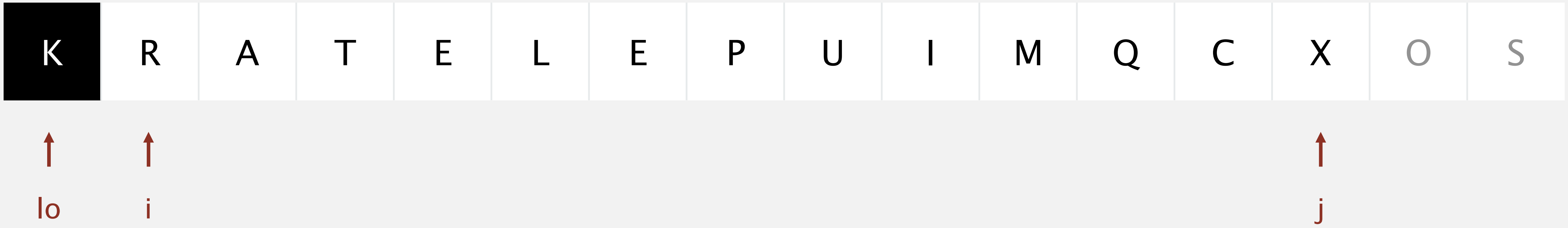
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
lo	i													j	

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

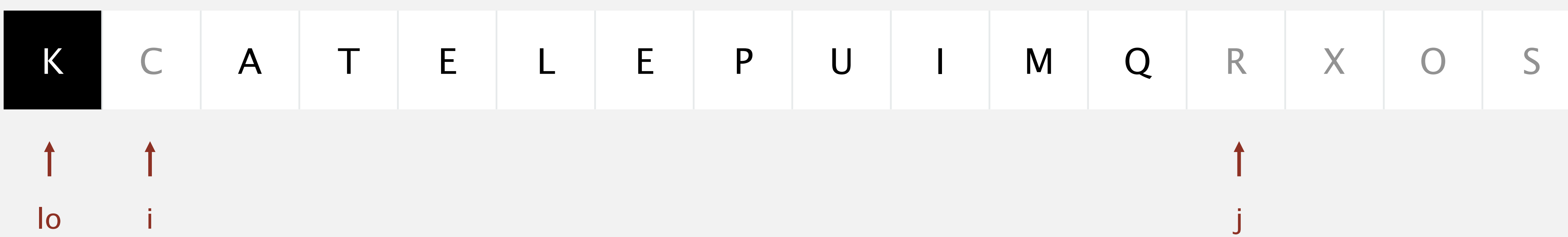
K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
lo	i											j			

stop j scan and exchange a[i] with a[j]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

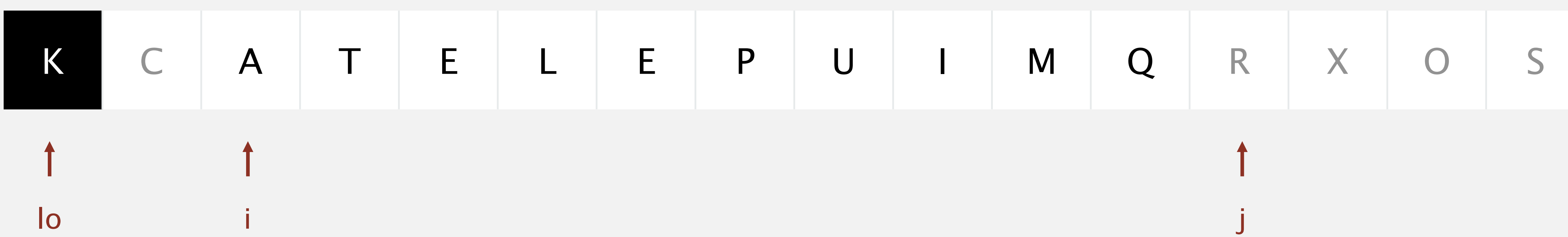
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

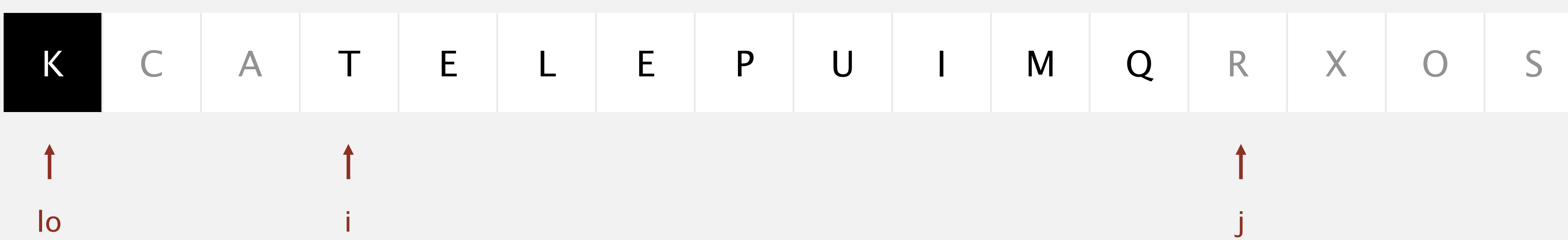
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

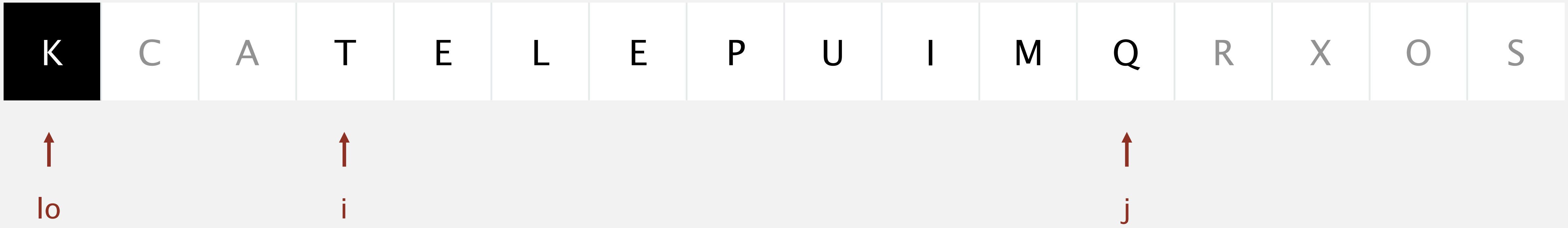


stop i scan because a[i] >= a[lo]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
lo			i							j					

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

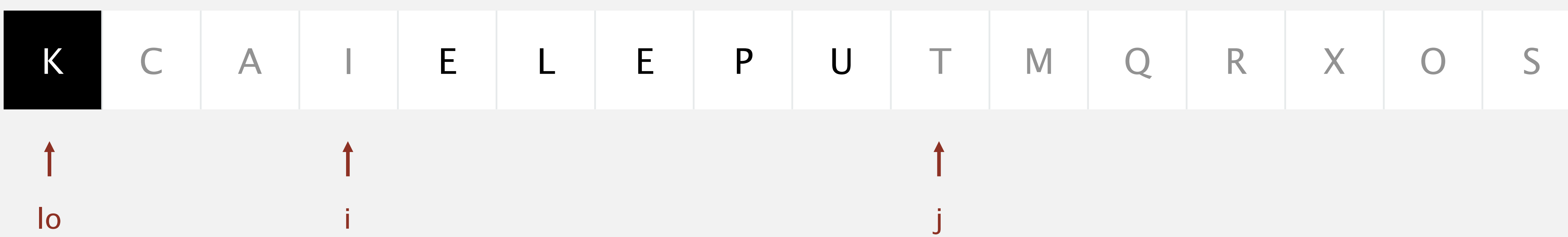
K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
↑ lo			↑ i						↑ j						

stop j scan and exchange a[i] with a[j]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

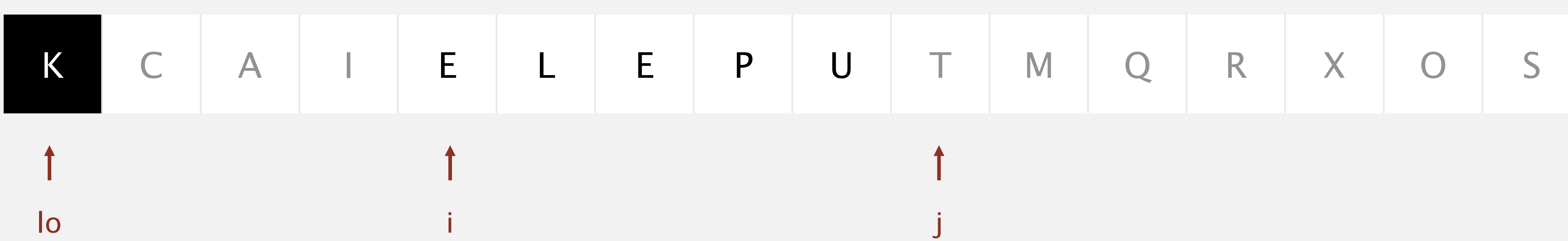
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

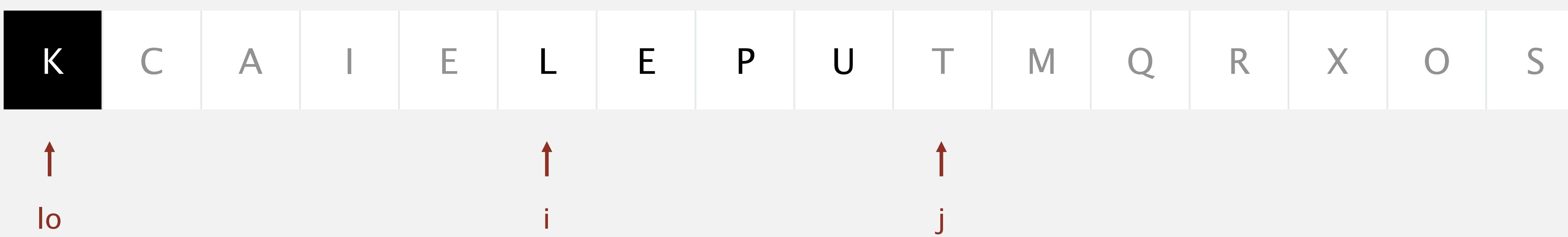
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



stop i scan because a[i] >= a[lo]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
lo					i			j							

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
↑					↑		↑								
lo					i		j								

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

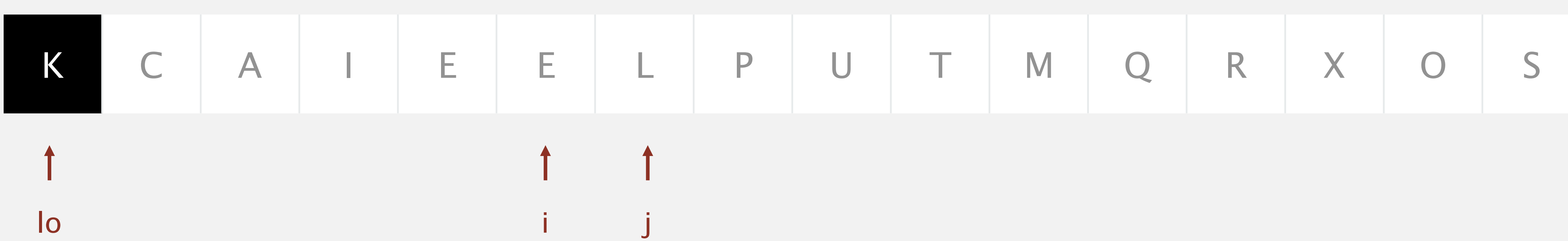
K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
lo					i	j									

stop j scan and exchange a[i] with a[j]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

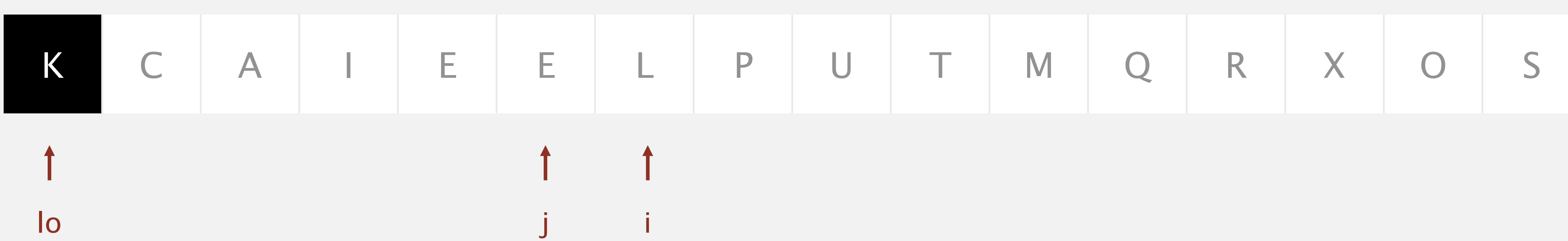
K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
lo						i j									

stop i scan because a[i] >= a[lo]

Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



stop j scan because a[j] <= a[lo]

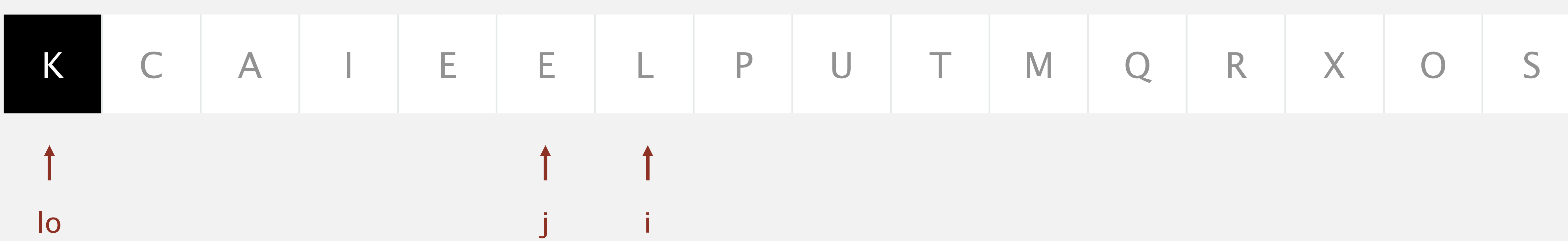
Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.



pointers cross: exchange a[lo] with a[j]

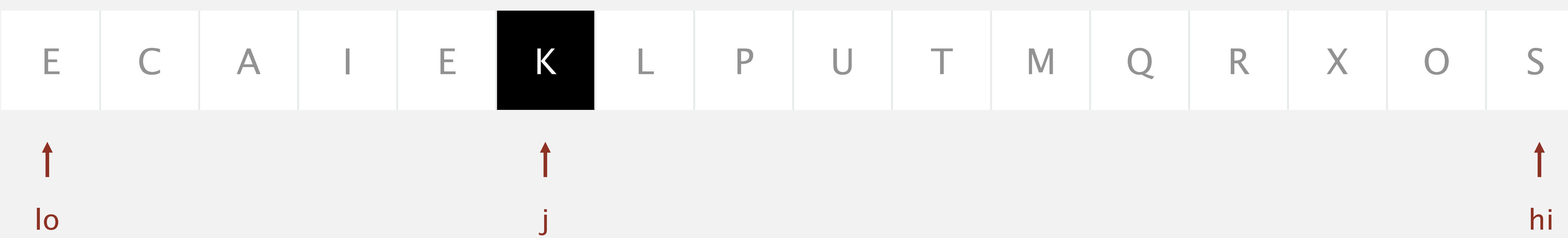
Quicksort partitioning demo

Repeat until i and j pointers cross.

- Scan i from left to right so long as (a[i] < a[lo]).
- Scan j from right to left so long as (a[j] > a[lo]).
- Exchange a[i] with a[j].

When pointers cross.

- Exchange a[lo] with a[j].



partitioned!

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

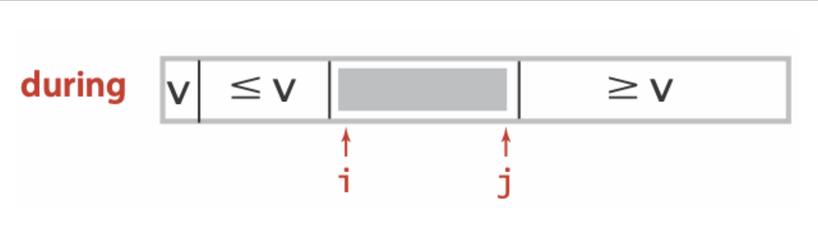
find item on right to swap

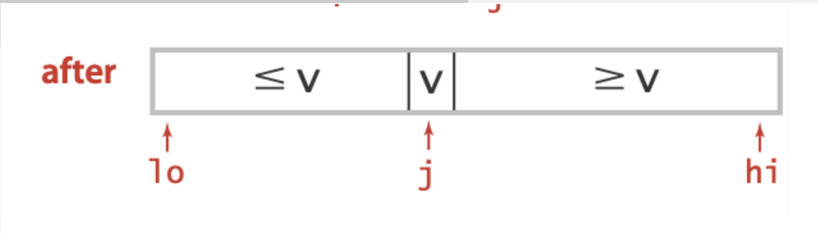
check if pointers cross

swap

swap with partitioning item

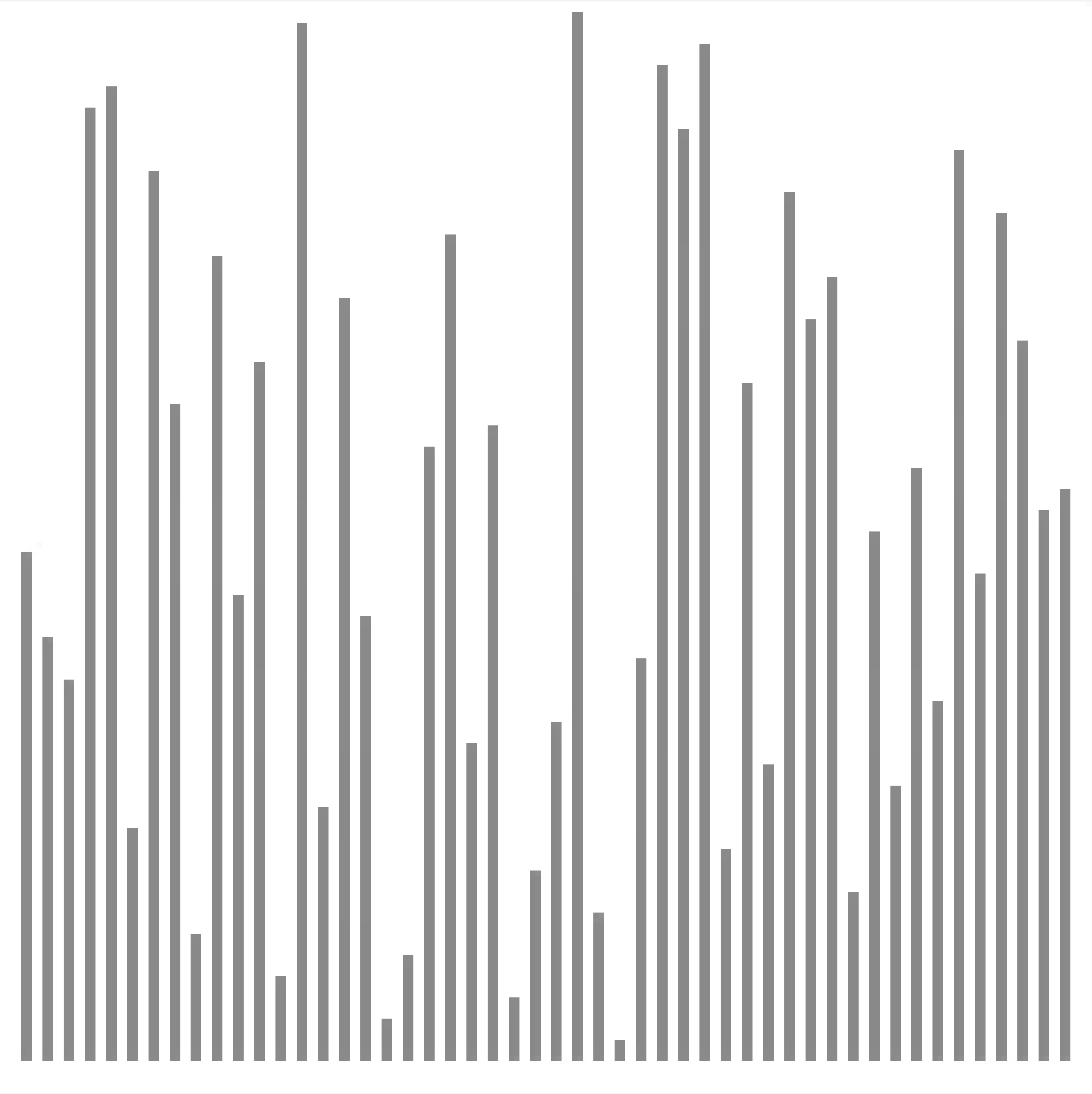
return index of item now known to be in place



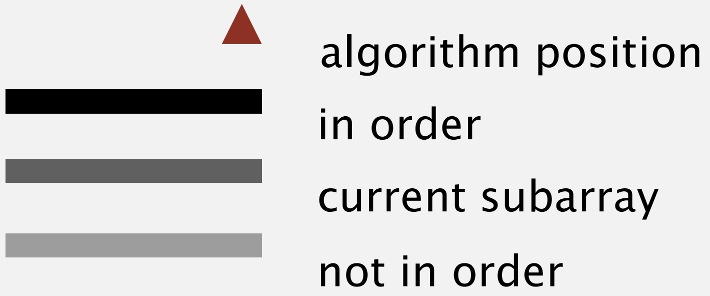


Quicksort animation

50 random items



http://www.sorting-algorithms.com/quick-sort



Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Preserving randomness. Shuffling is needed for performance guarantee.
Equivalent alternative. Pick a random partitioning item in each subarray.

Quicksort: best-case analysis

Best case. Number of compares is $\sim N \lg N$.

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
	0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
	0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
	0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
	8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
				A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: summary of performance characteristics

Quicksort is a randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39 N \lg N$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Best case. Number of compares is $\sim N \lg N$.

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

[but more likely that lightning bolt strikes computer during execution]

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (requires using an explicit stack)

Proposition. Quicksort is not stable.

Pf. [by counterexample]

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		½ N^2	½ N^2	½ N^2	N exchanges
insertion	✓	✓	N	¼ N^2	½ N^2	use for small N or partially ordered
merge		✓	½ $N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
quick	✓		$N \lg N$	$2 N \ln N$	½ N^2	$N \log N$ probabilistic guarantee; fastest in practice
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail

Interesting Problem: Selection

Goal. Given an array of N items, find the k^{th} smallest item.

Ex. Min $(k = 0)$, max $(k = N - 1)$, median $(k = N/2)$.

Applications.

- Order statistics.
- Find the "top k."

Use theory as a guide.

- Easy $N \log N$ upper bound. How?
- Easy N lower bound. Why?

Which is true?

- $N \log N$ lower bound? ← is selection as hard as sorting?
- N upper bound? ← is there a linear-time algorithm?

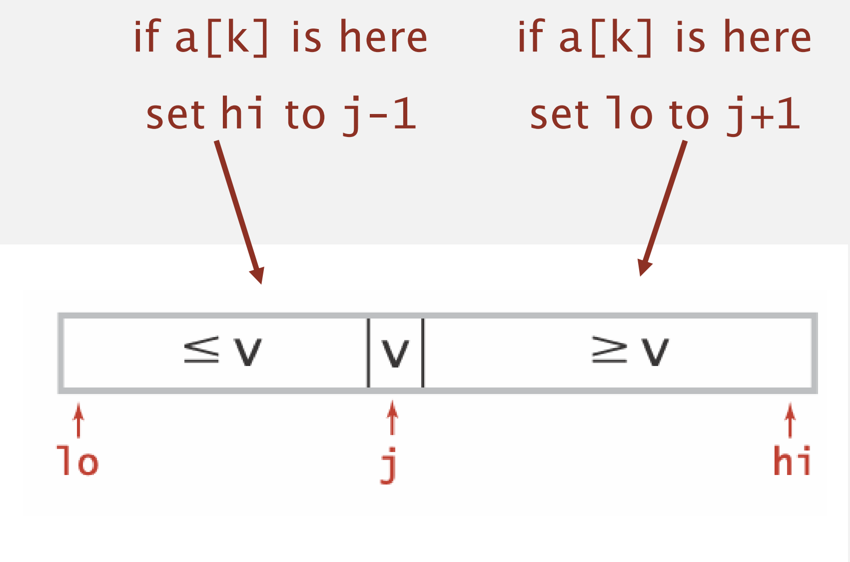
Quick-select

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in **one** subarray, depending on `j`; finished when `j` equals `k`.

```
public static Comparable select(Comparable[] a, int
k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
       int j = partition(a, lo, hi);
       if      (j < k) lo = j + 1;
       else if (j > k) hi = j - 1;
       else            return a[k];
    }
    return a[k];
}
```



Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

select element of rank k = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
50	21	28	65	39	59	56	22	95	12	90	53	32	77	33

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
50	21	28	65	39	59	56	22	95	12	90	53	32	77	33

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
22	21	28	33	39	32	12	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

can safely ignore right subarray

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
22	21	28	33	39	32	12	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
22	21	28	33	39	32	12	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	33	39	32	28	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

can safely ignore left subarray

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	33	39	32	28	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	33	39	32	28	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	32	28	33	39	50	95	56	90	53	59	77	65

k = 5

Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

stop: partitioning item is at index k

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	32	28	33	39	50	95	56	90	53	59	77	65

k = 5

Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

- Intuitively, each partitioning step splits array approximately in half:
  $N + N/2 + N/4 + \ldots + 1 \sim 2N$ compares

http://algs4.cs.princeton.edu

PRIORITY QUEUES

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

http://algs4.cs.princeton.edu

Priority Queues

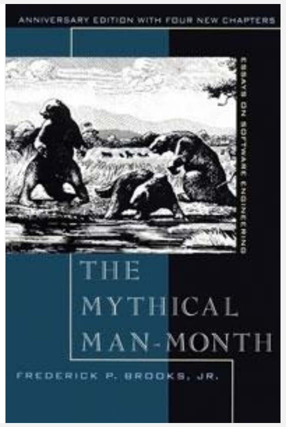
- *API and elementary implementations*
- *binary heaps*
- *heapsort*

Collections

A collection is a data types that store groups of items.

data type	key operations	data structure
stack	PUSH, POP	*linked list, resizing array*
queue	ENQUEUE, DEQUEUE	*linked list, resizing array*
priority queue	INSERT, DELETE-MAX	*binary heap*
symbol table	PUT, GET, DELETE	*BST, hash table*
set	ADD, CONTAINS, DELETE	*BST, hash table*

" Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious." — Fred Brooks



Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

Priority queue. Remove the largest (or smallest) item.

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [online median in data stream]
- Operating systems. [load balancing, interrupt handling]
- Computer networks. [web cache]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue API

Requirement. Generic items are `Comparable`.

Key must be Comparable
(bounded type parameter)

`public class MaxPQ<Key extends Comparable<Key>>`

	`MaxPQ()`	*create an empty priority queue*
	`MaxPQ(Key[] a)`	*create a priority queue with given keys*
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`delMax()`	*return and remove the largest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`max()`	*return the largest key*
`int`	`size()`	*number of entries in the priority queue*

Priority queue API

Requirement. Generic items are `Comparable`.

`public class MinPQ<Key extends Comparable<Key>>`

	`MinPQ()`	*create an empty priority queue*
	`MinPQ(Key[] a)`	*create a priority queue with given keys*
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`delMin()`	*return and remove the smallest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`min()`	*return the smallest key*
`int`	`size()`	*number of entries in the priority queue*

Priority queue: unordered array implementation

```
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;    // pq[i] = ith element on pq
   private int N;       // number of elements on pq

   public UnorderedArrayMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

   public boolean isEmpty(){  return N == 0; }

   public void insert(Key x)
   {  pq[N++] = x;  }

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic
array creation

should null out entry
to prevent loitering

less() and exch()
similar to sorting methods
(but don't pass pq[])

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	$\log N$	$\log N$	$\log N$

order of growth of running time for priority queue with N items

http://algs4.cs.princeton.edu

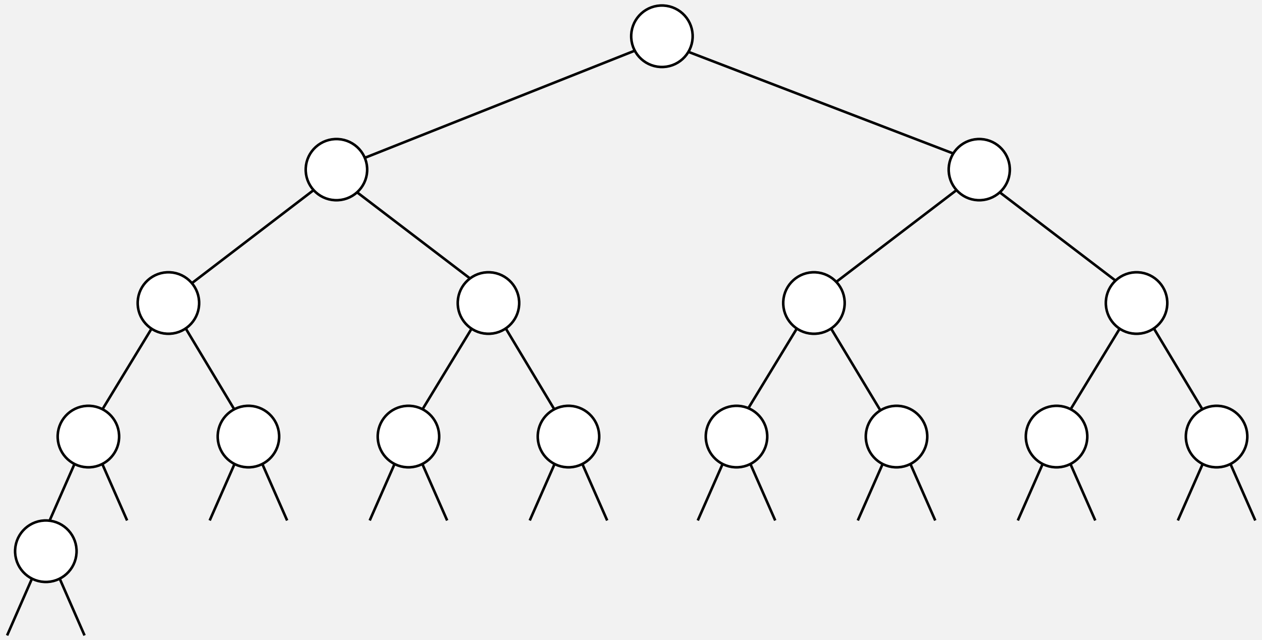
Priority Queues

- *API and elementary implementations*
- *binary heaps*
- *heapsort*

Complete binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.



complete tree with N = 16 nodes (height = 4)

Property. Height of complete tree with N nodes is $\lfloor \lg N \rfloor$.

Pf. Height increases only when N is a power of 2.

Binary heap representations

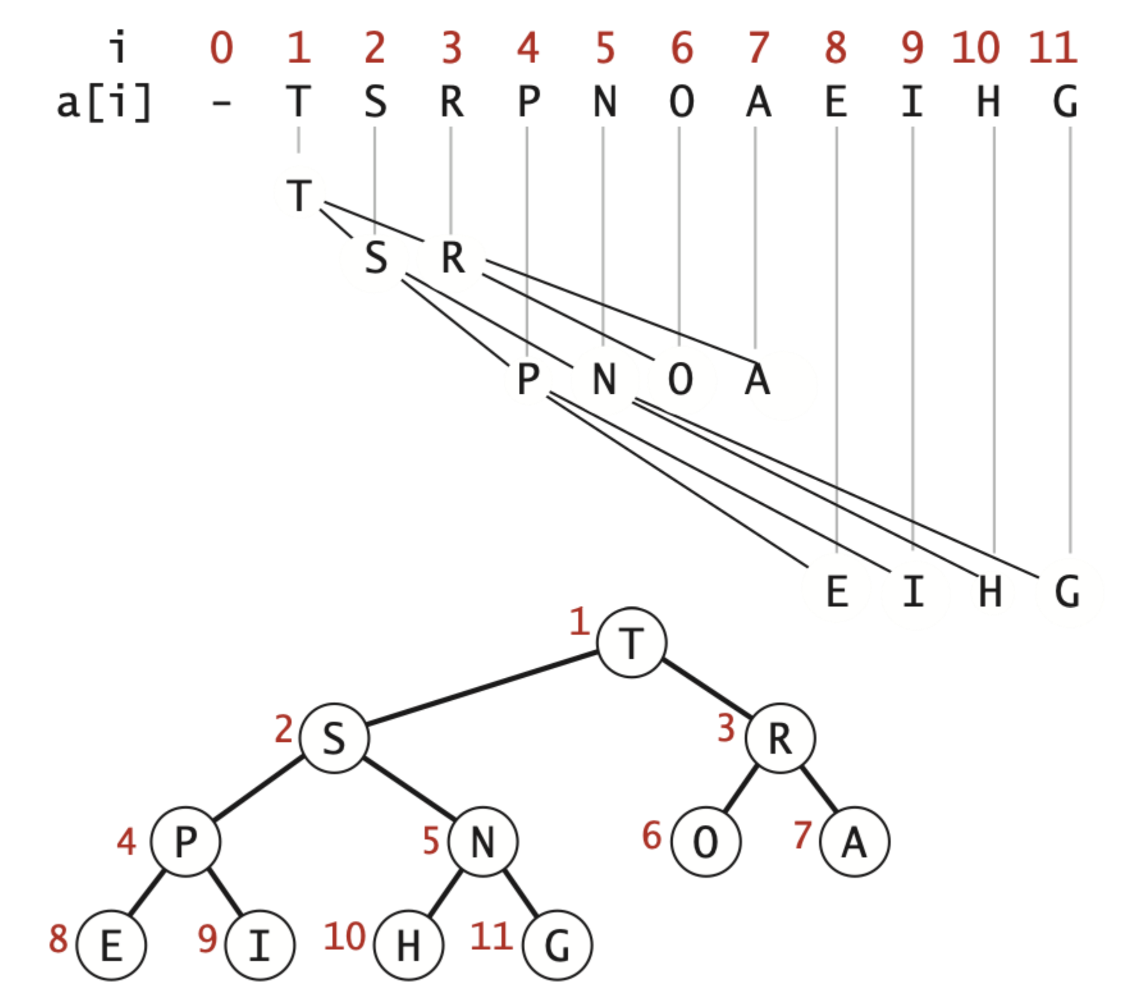
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no smaller than children's keys.

Array representation.

- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!



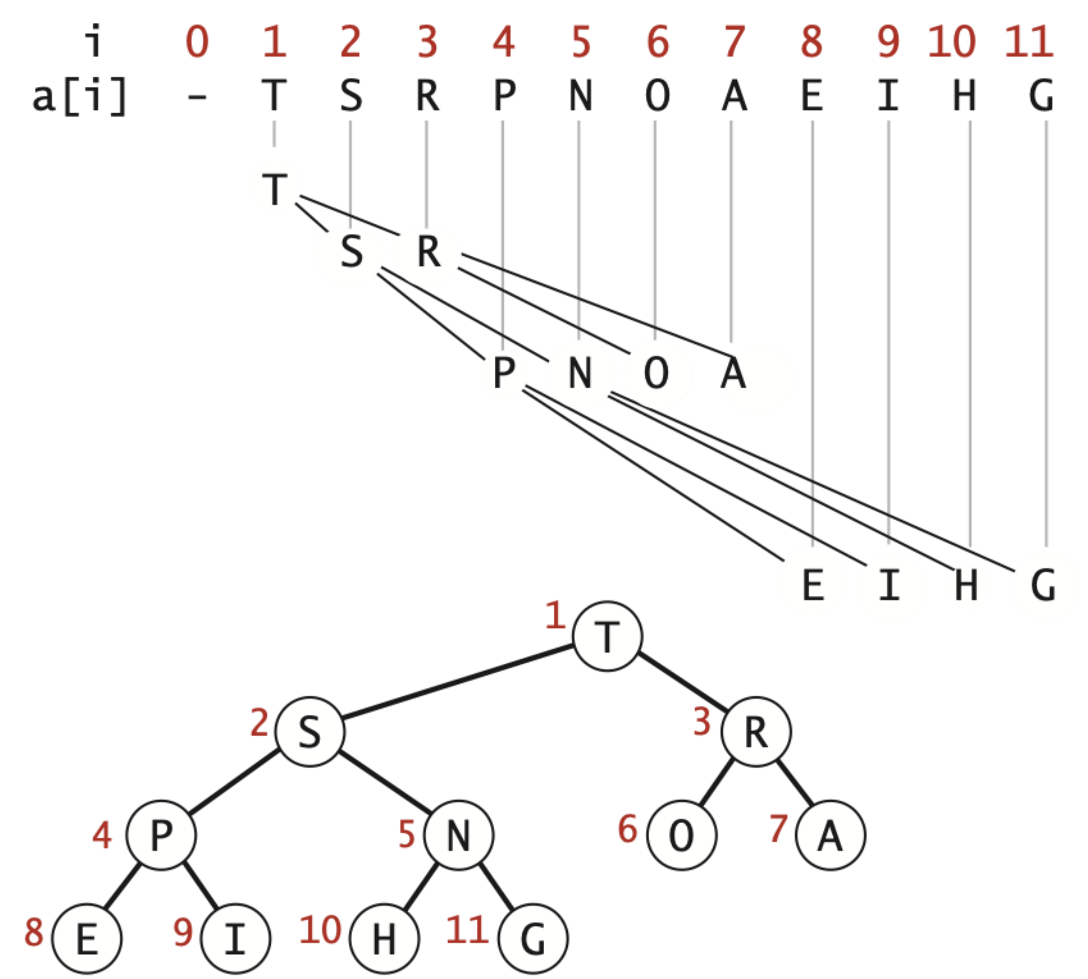
Heap representations

Binary heap properties

Proposition. Largest key is `a[1]`, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at `k` is at `k/2`.
- Children of node at `k` are at `2k` and `2k+1`.



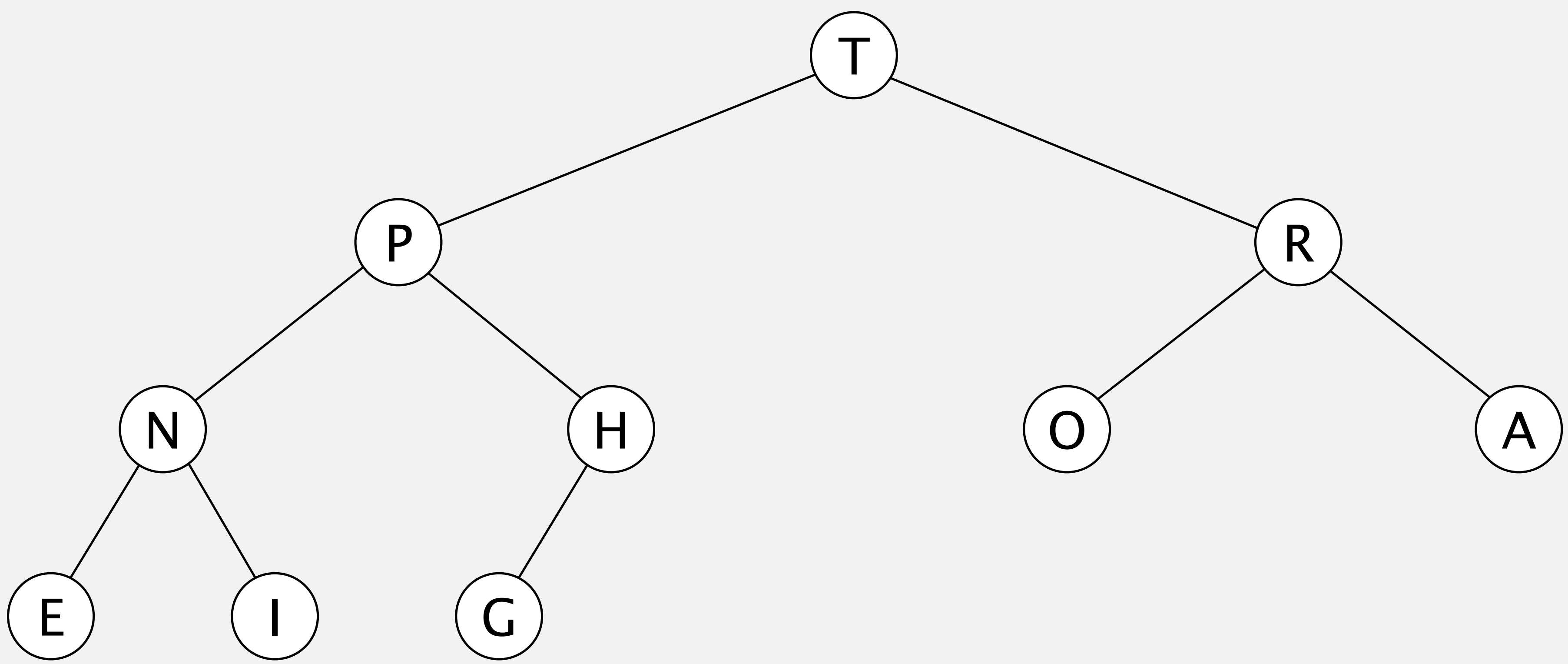
Heap representations

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

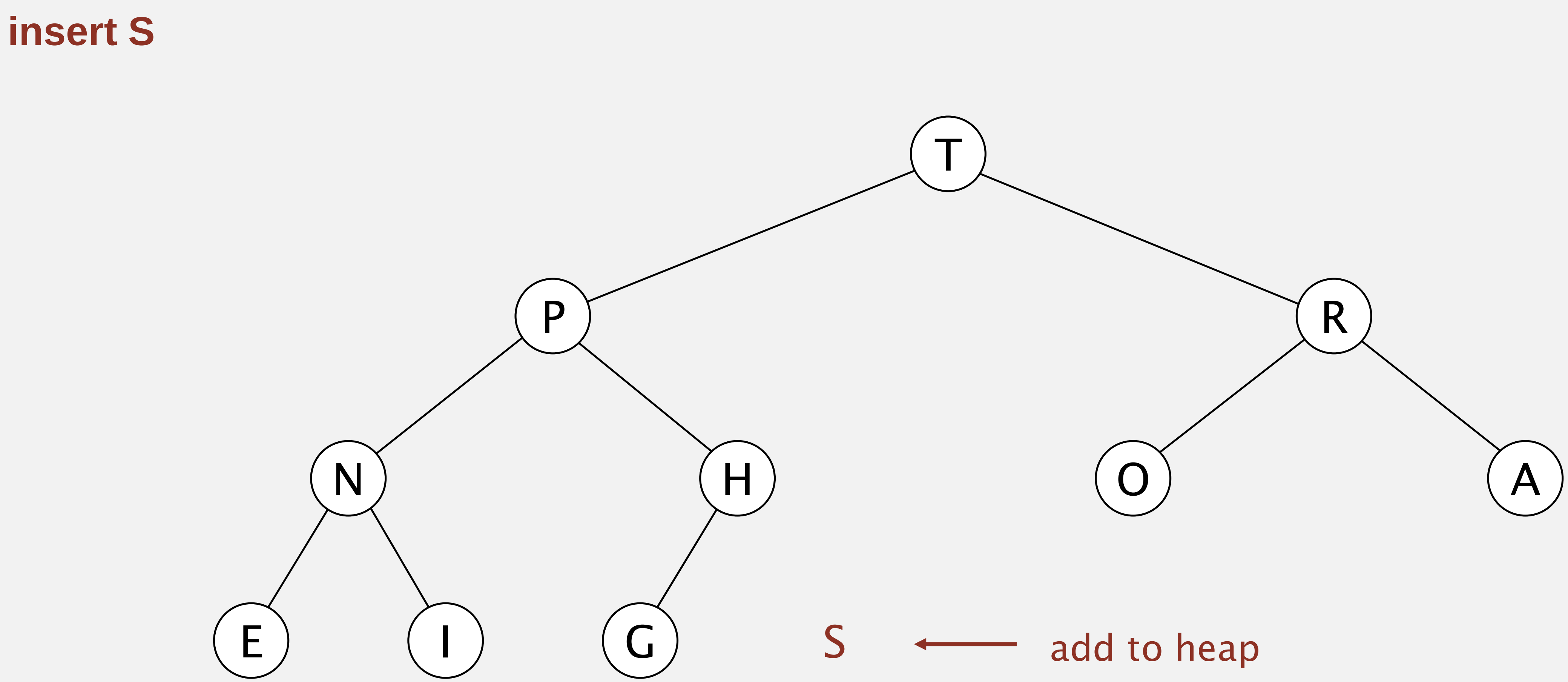


T	P	R	N	H	O	A	E	I	G	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

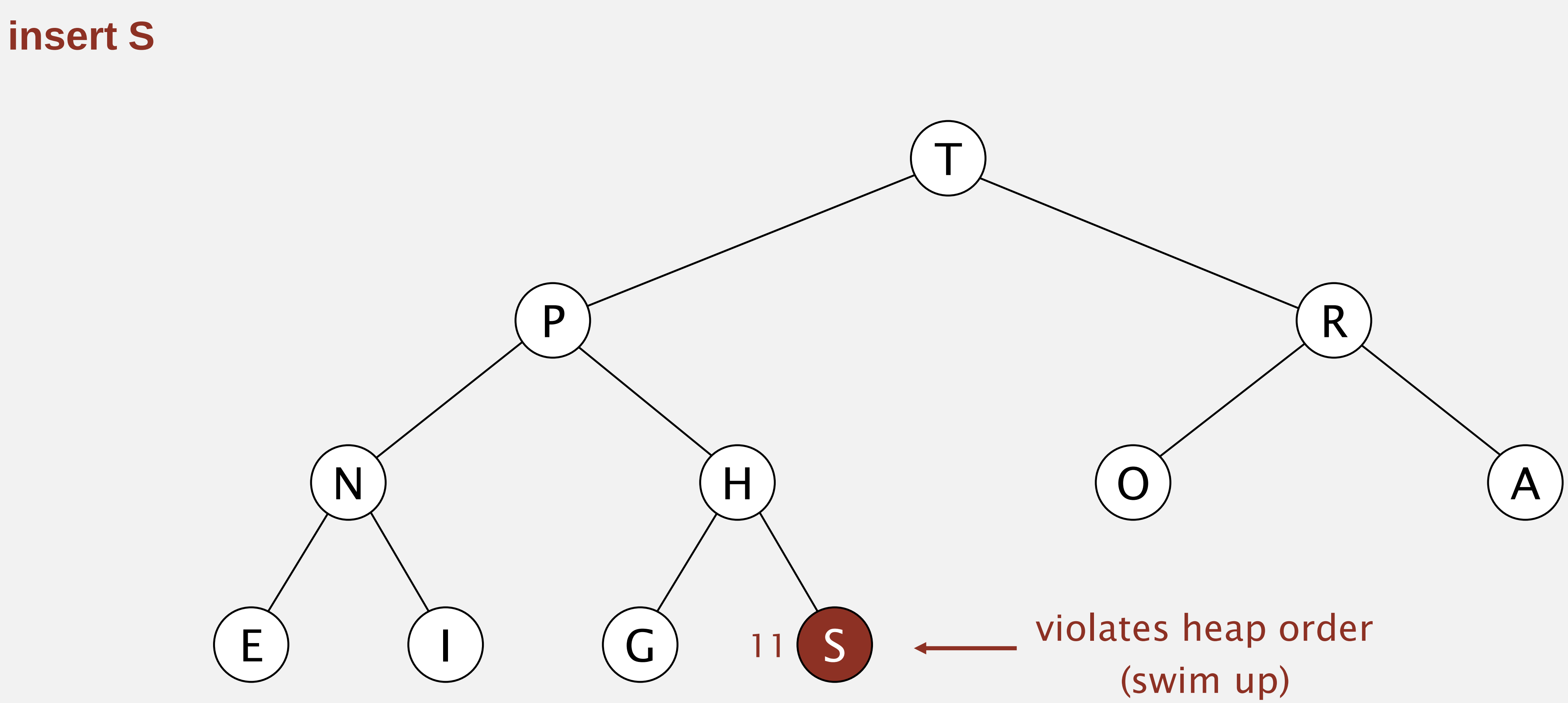


T	P	R	N	H	O	A	E	I	G	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.



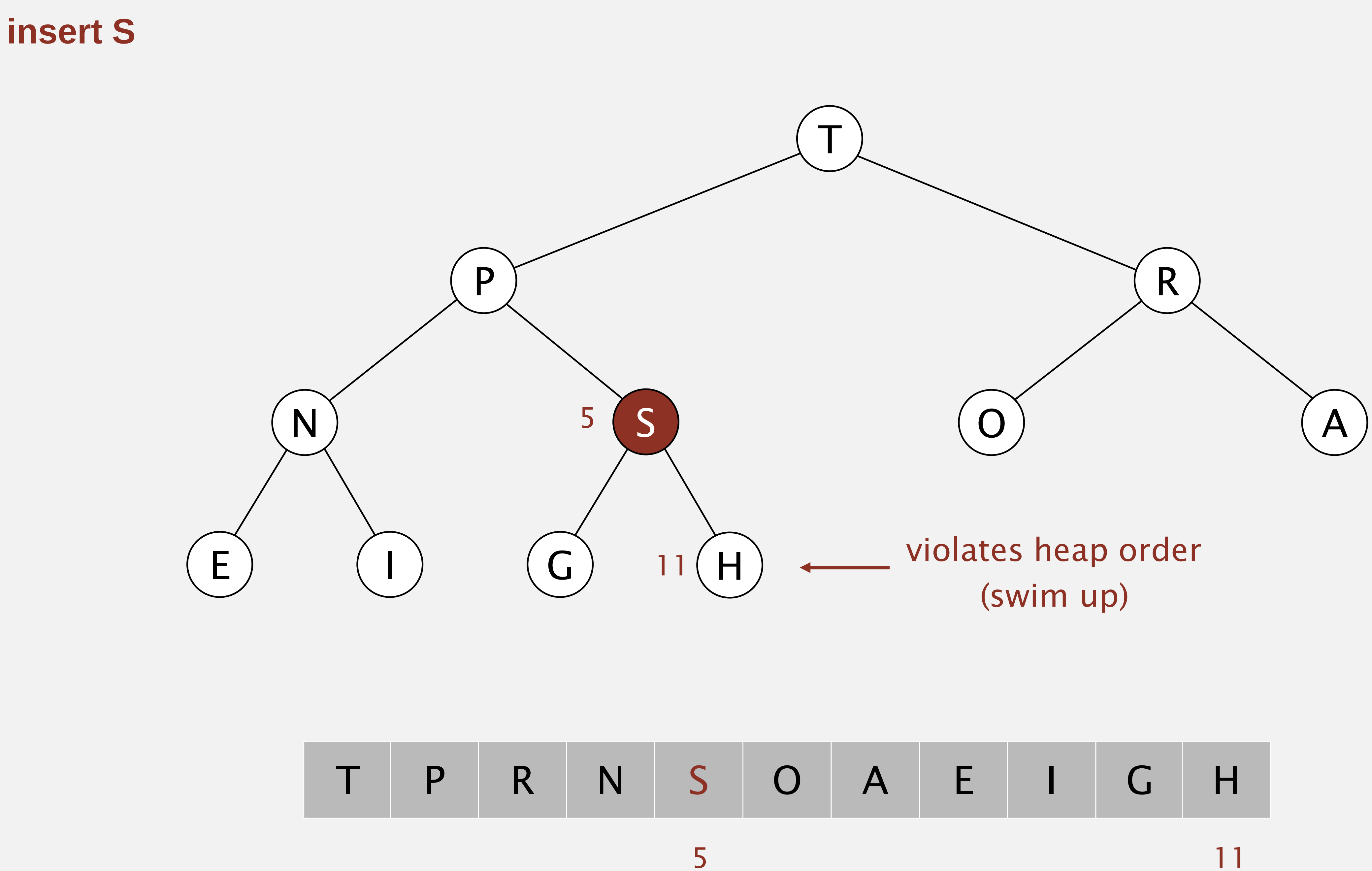
T	P	R	N	H	O	A	E	I	G	S

11

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

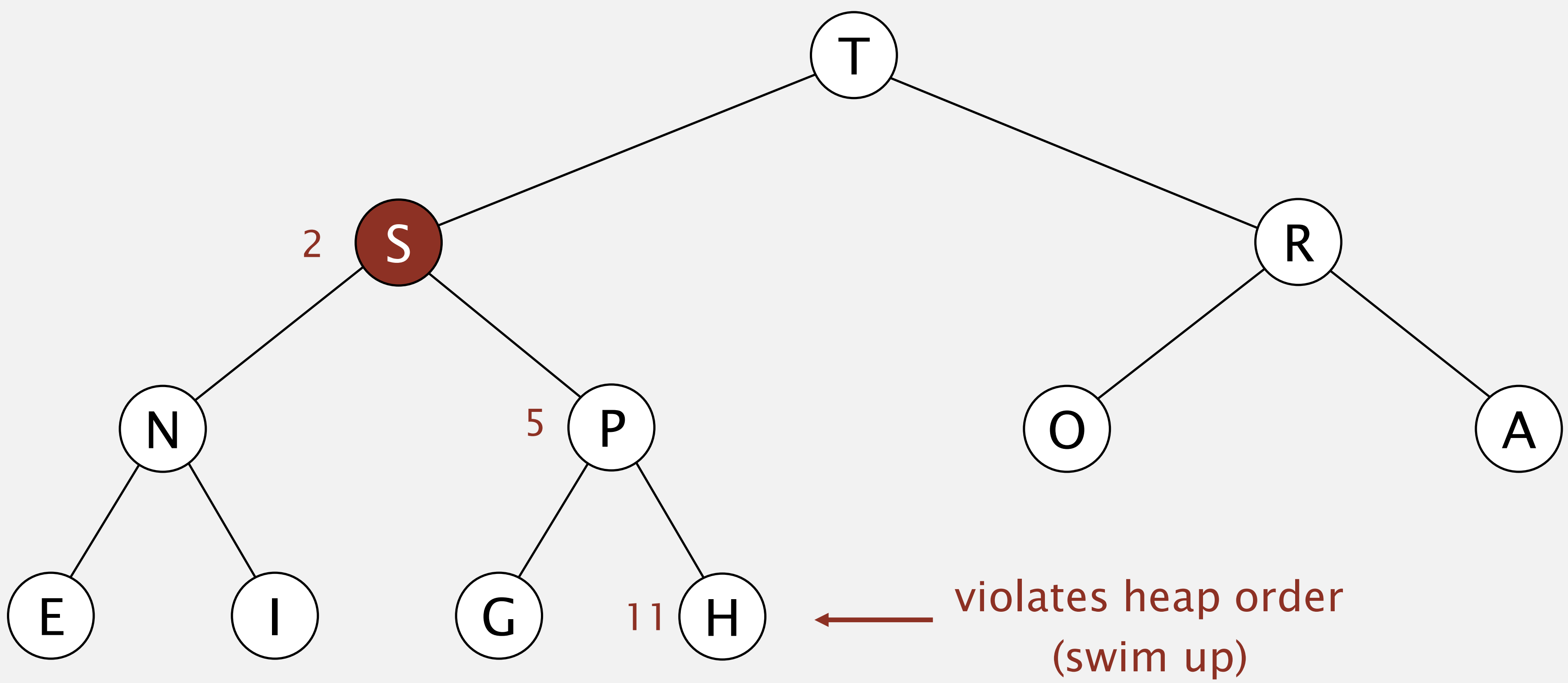


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



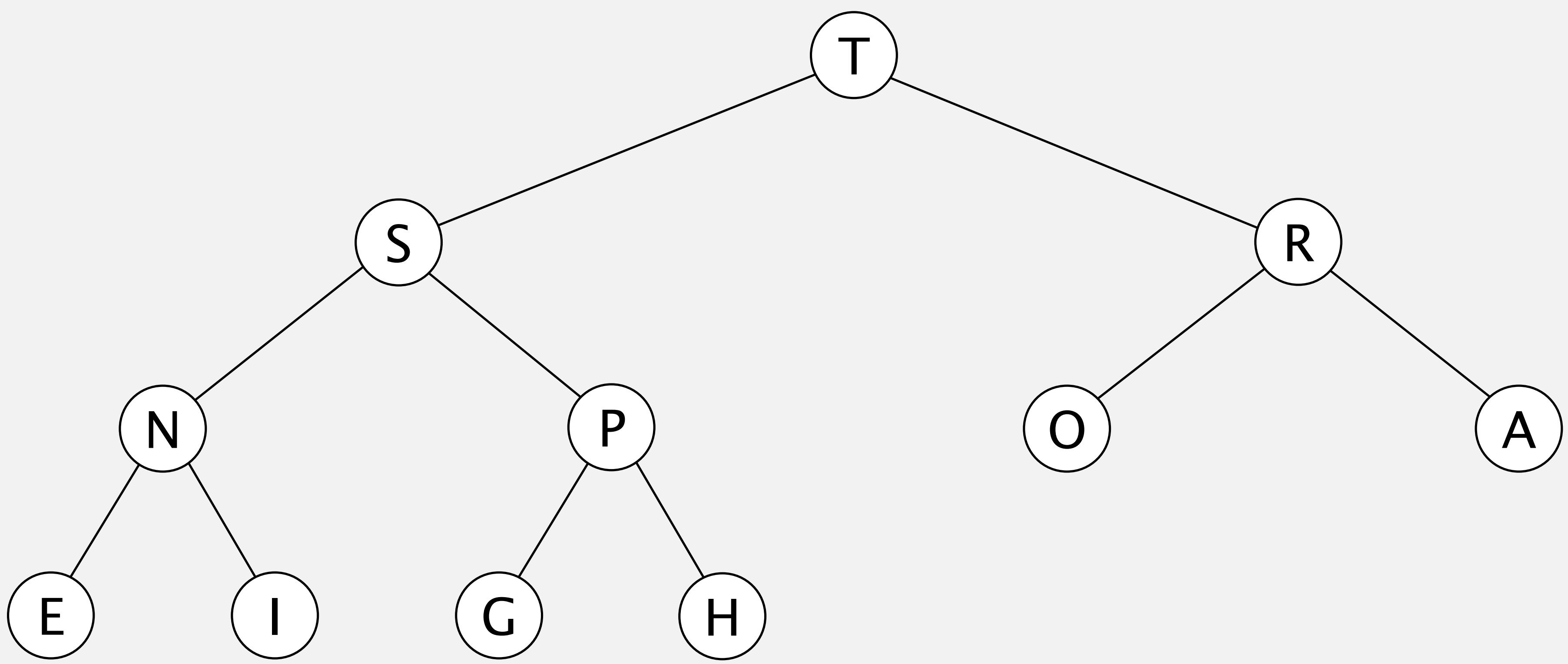
T	S	R	N	P	O	A	E	I	G	H
	2			5						11

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

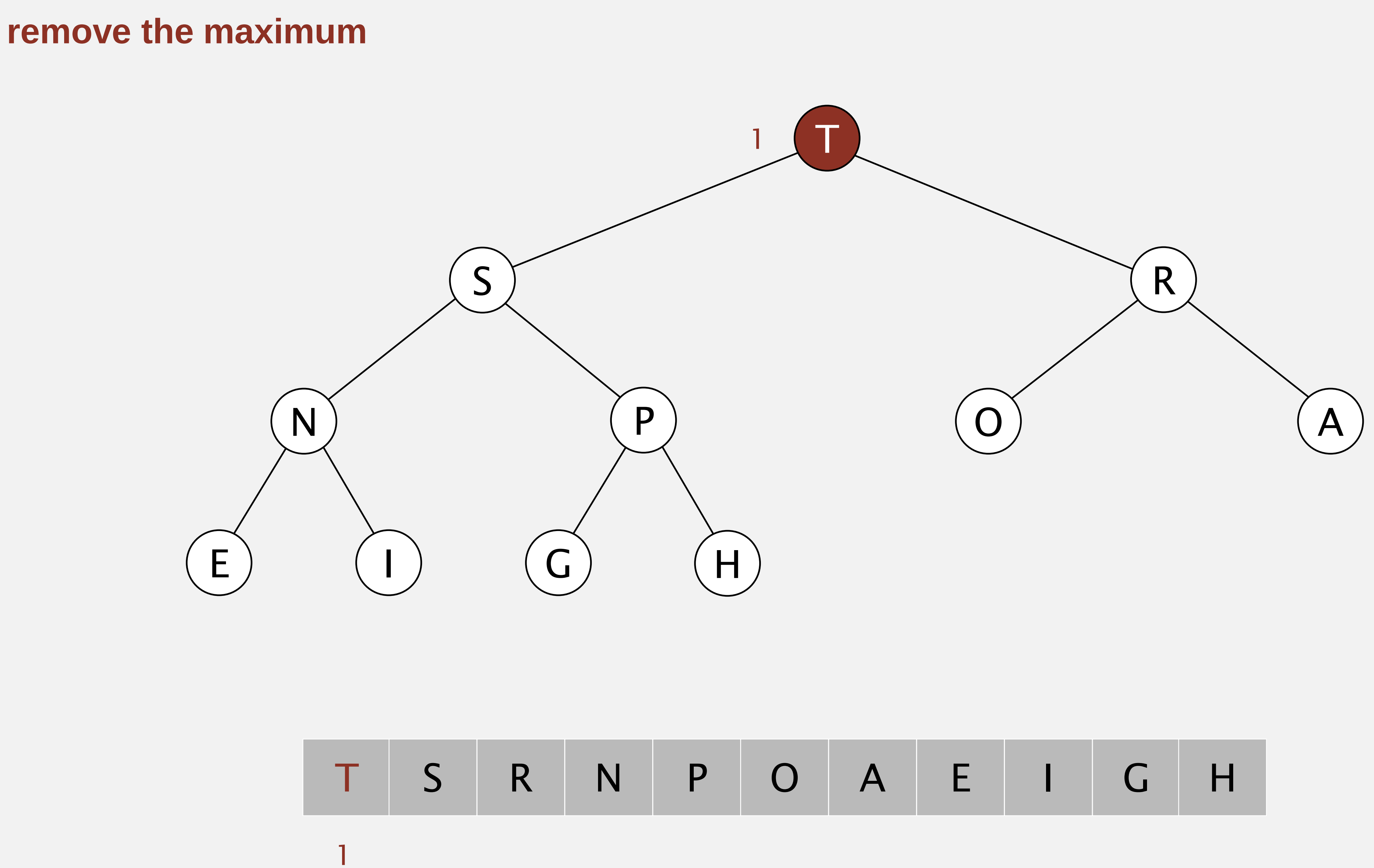


T	S	R	N	P	O	A	E	I	G	H

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

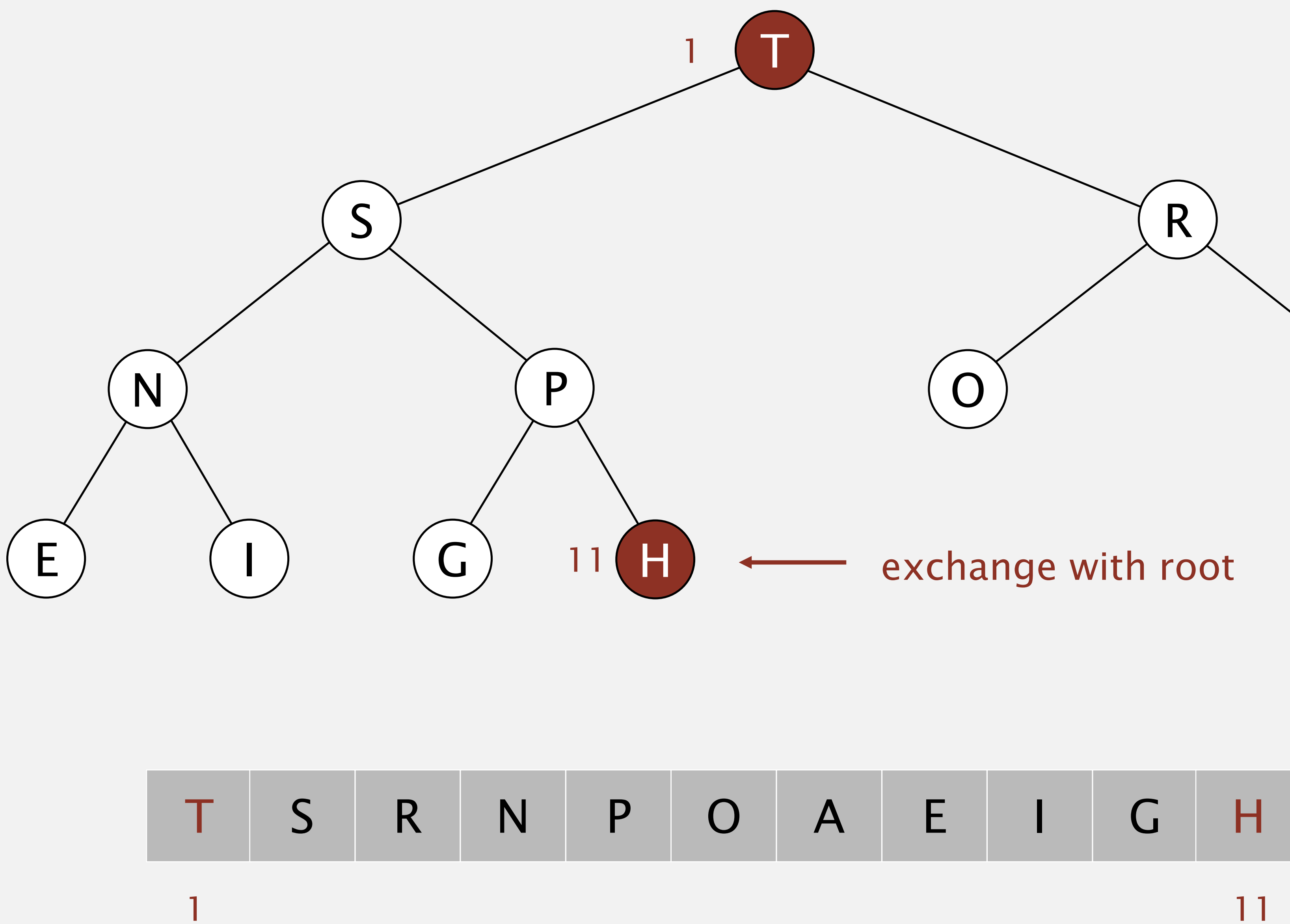


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

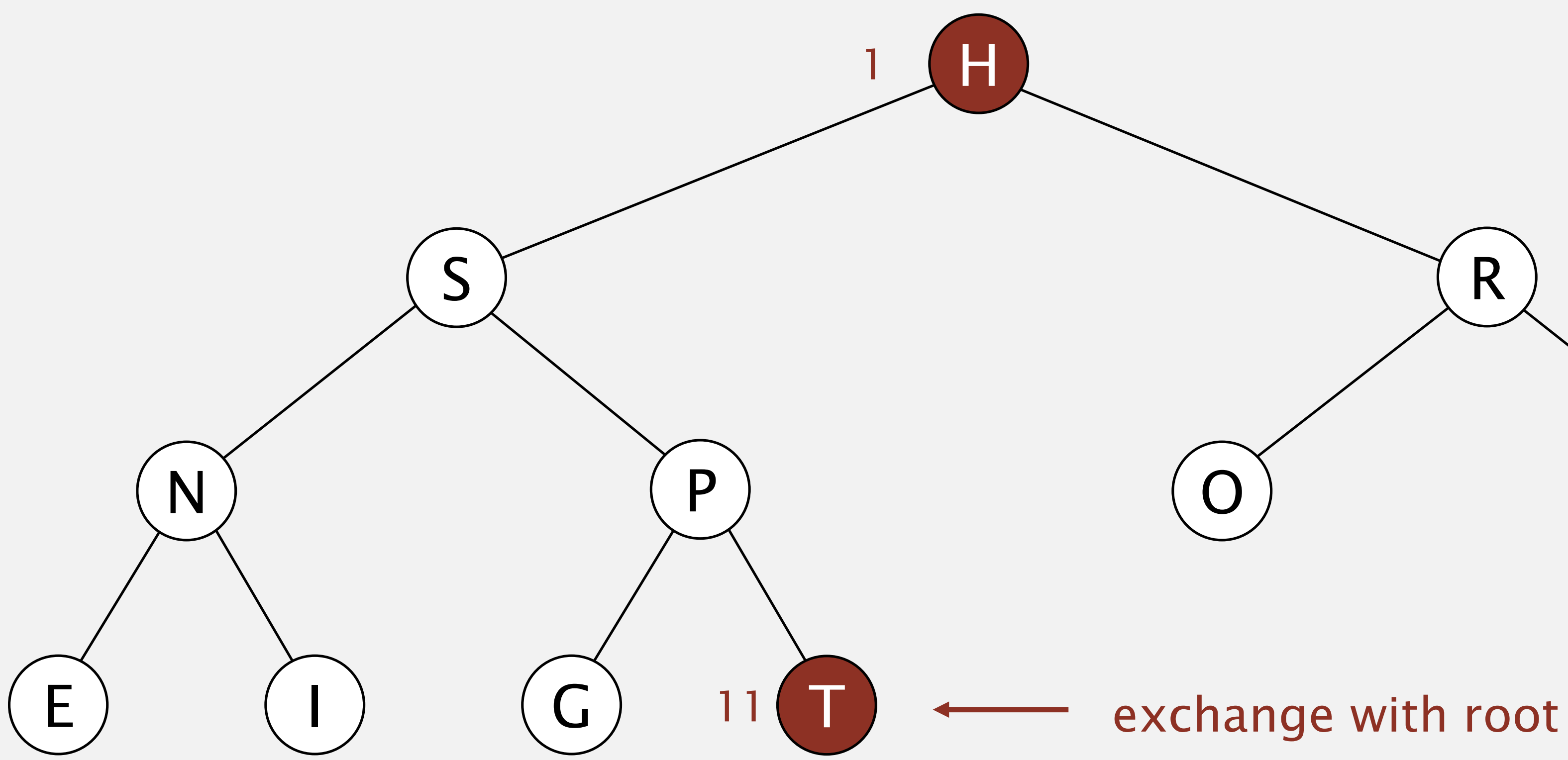


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



H	S	R	N	P	O	A	E	I	G	T

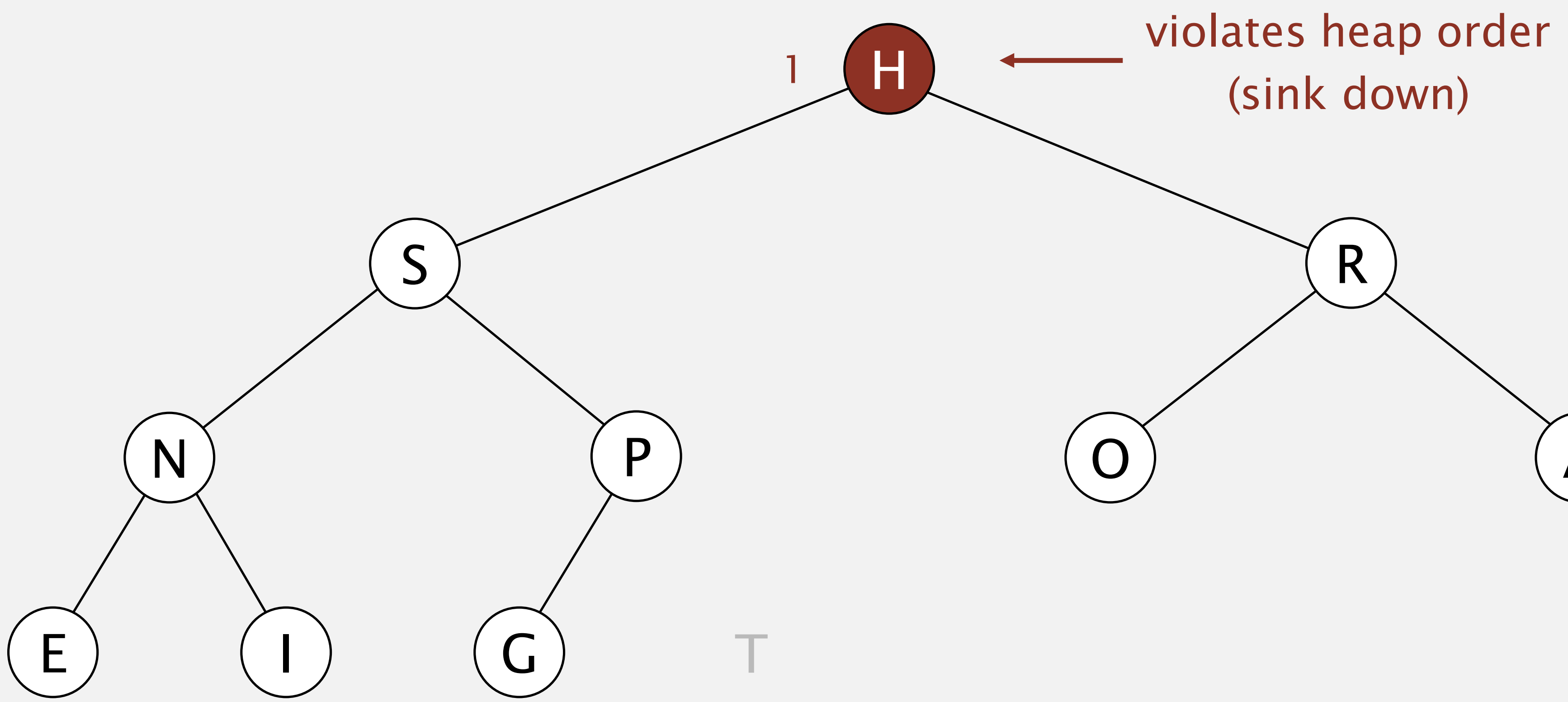
1 11

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



H	S	R	N	P	O	A	E	I	G	T

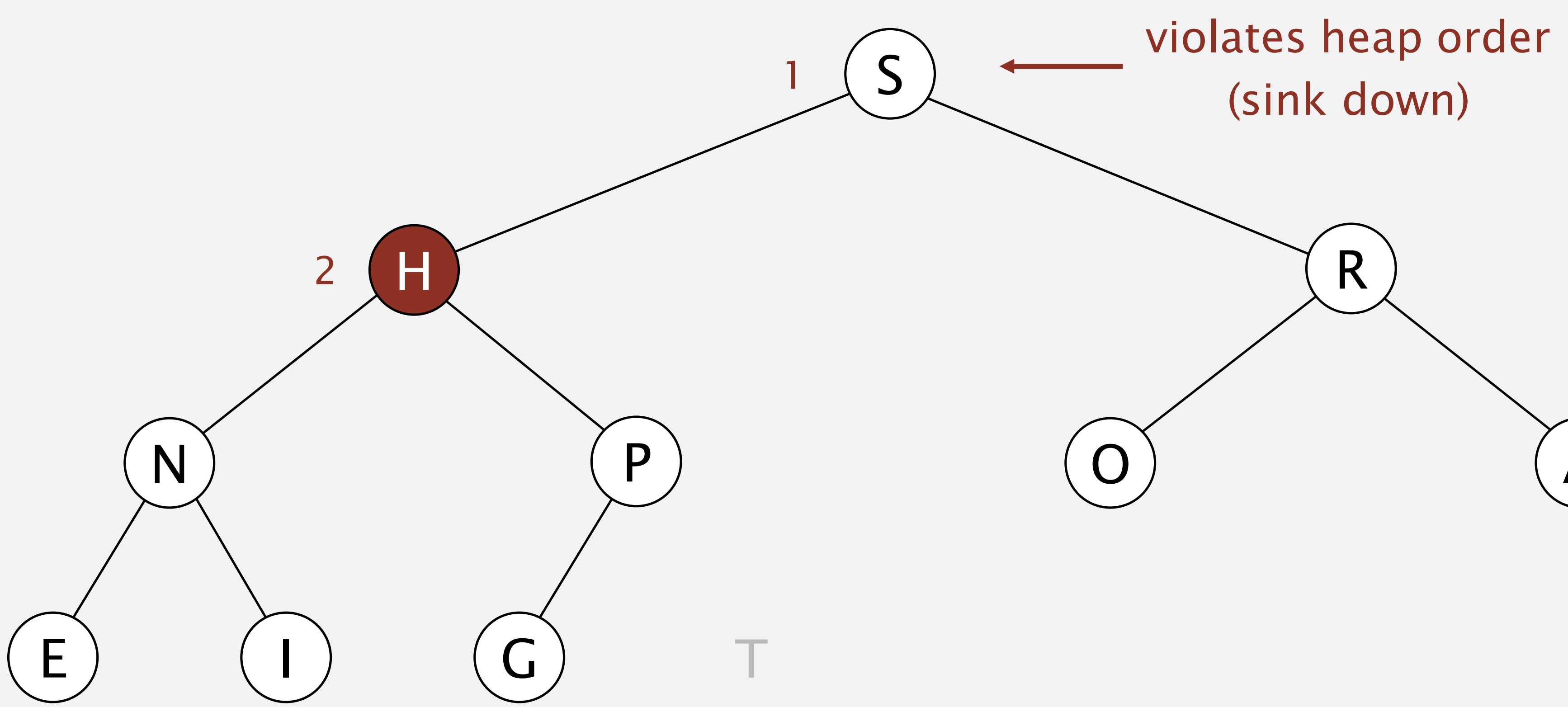
1

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



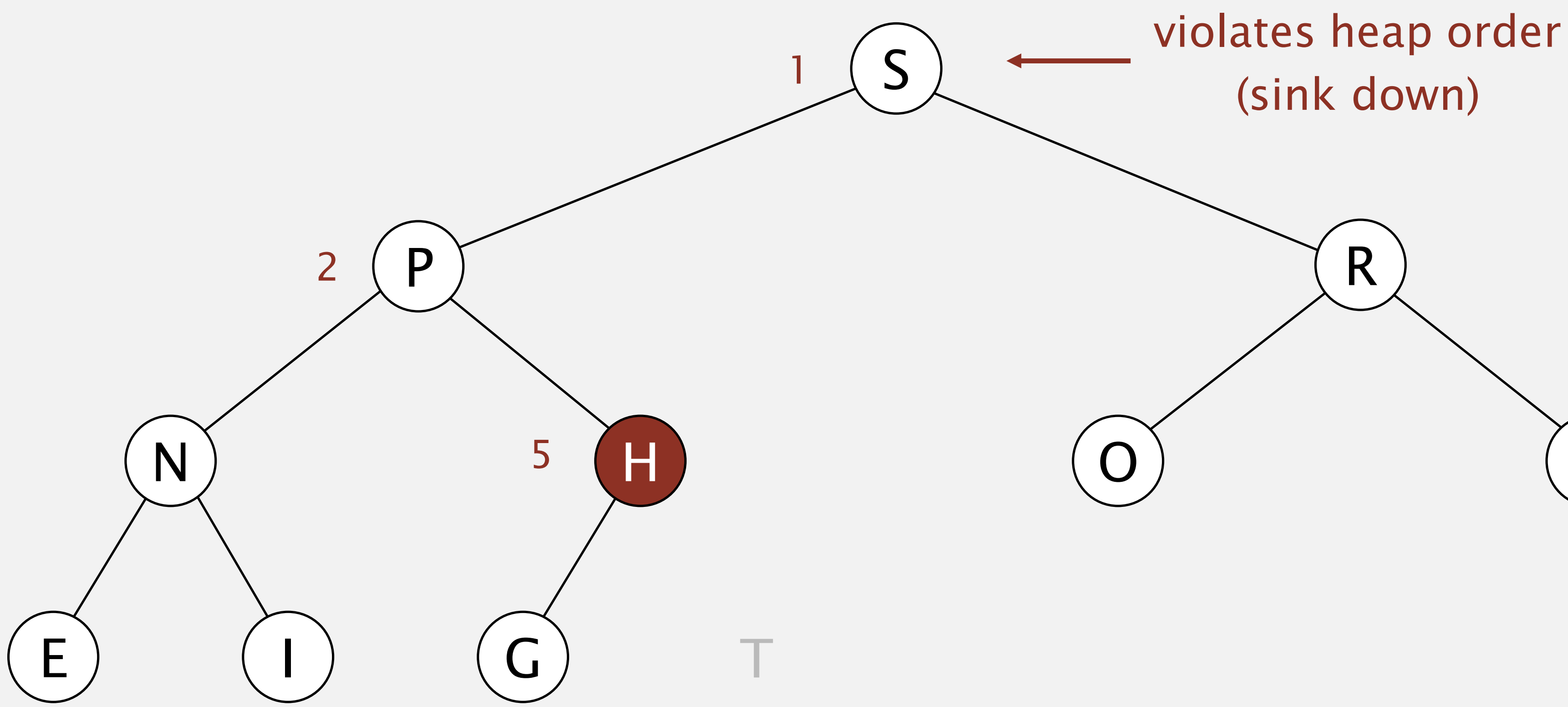
S	H	R	N	P	O	A	E	I	G	T
1	2									

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



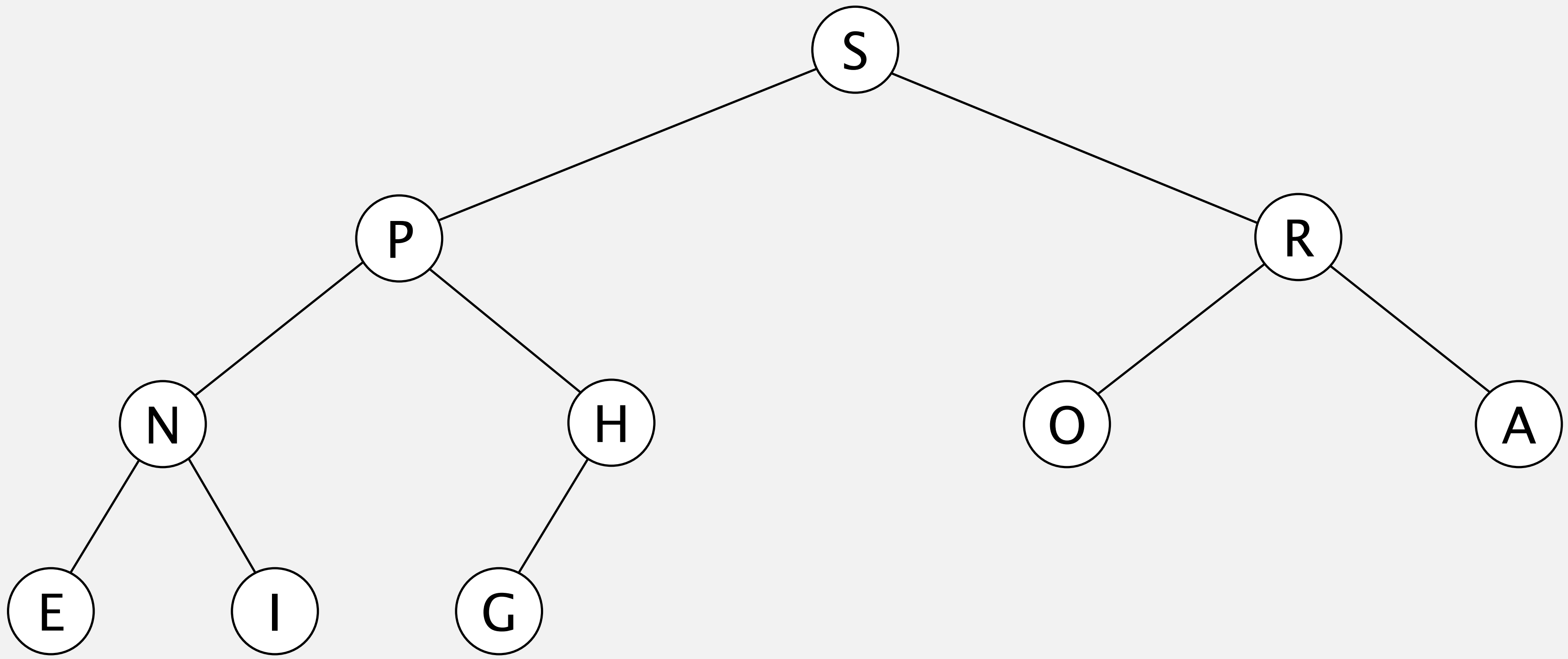
S	P	R	N	H	O	A	E	I	G	T
1	2			5						

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

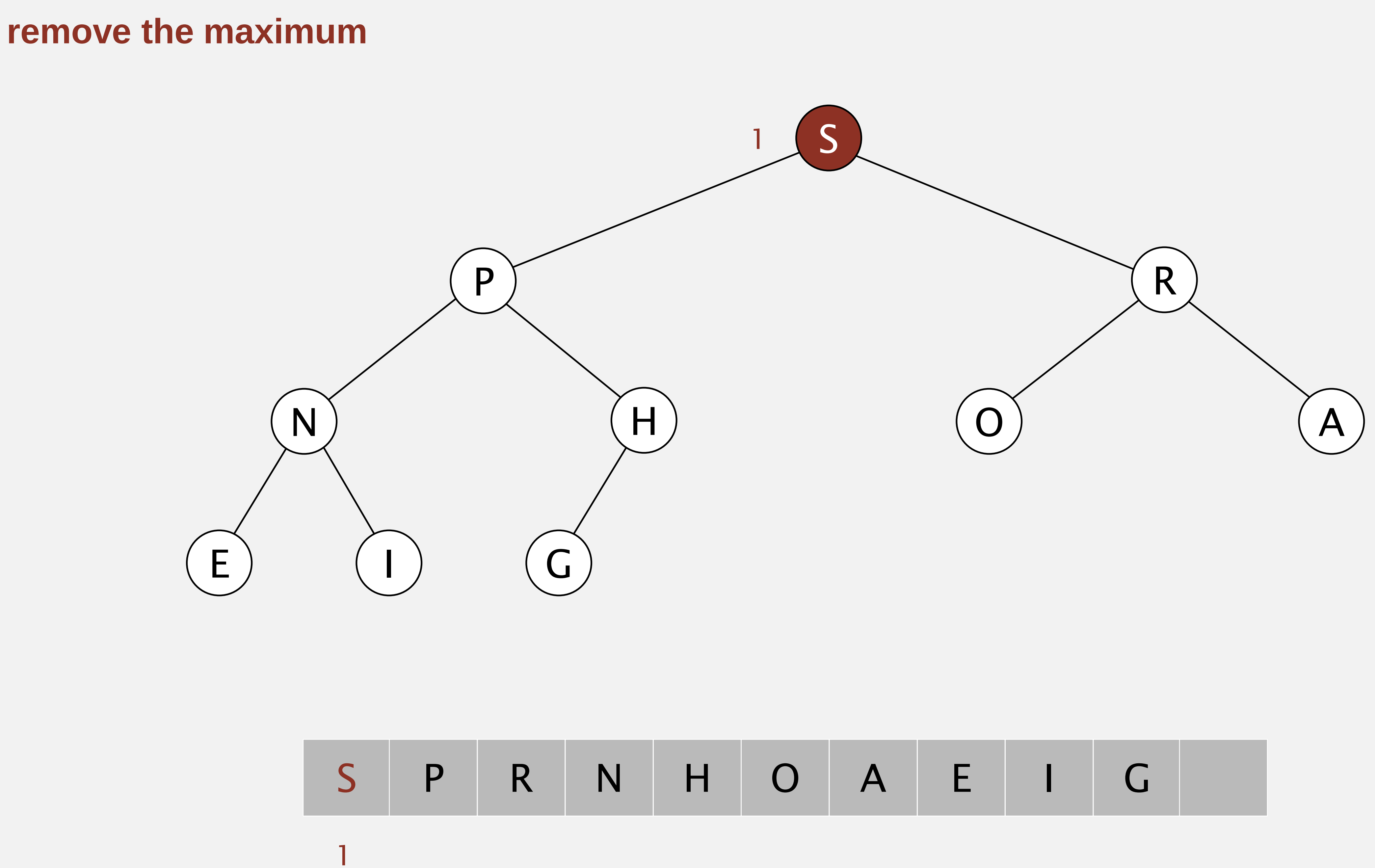


S	P	R	N	H	O	A	E	I	G	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

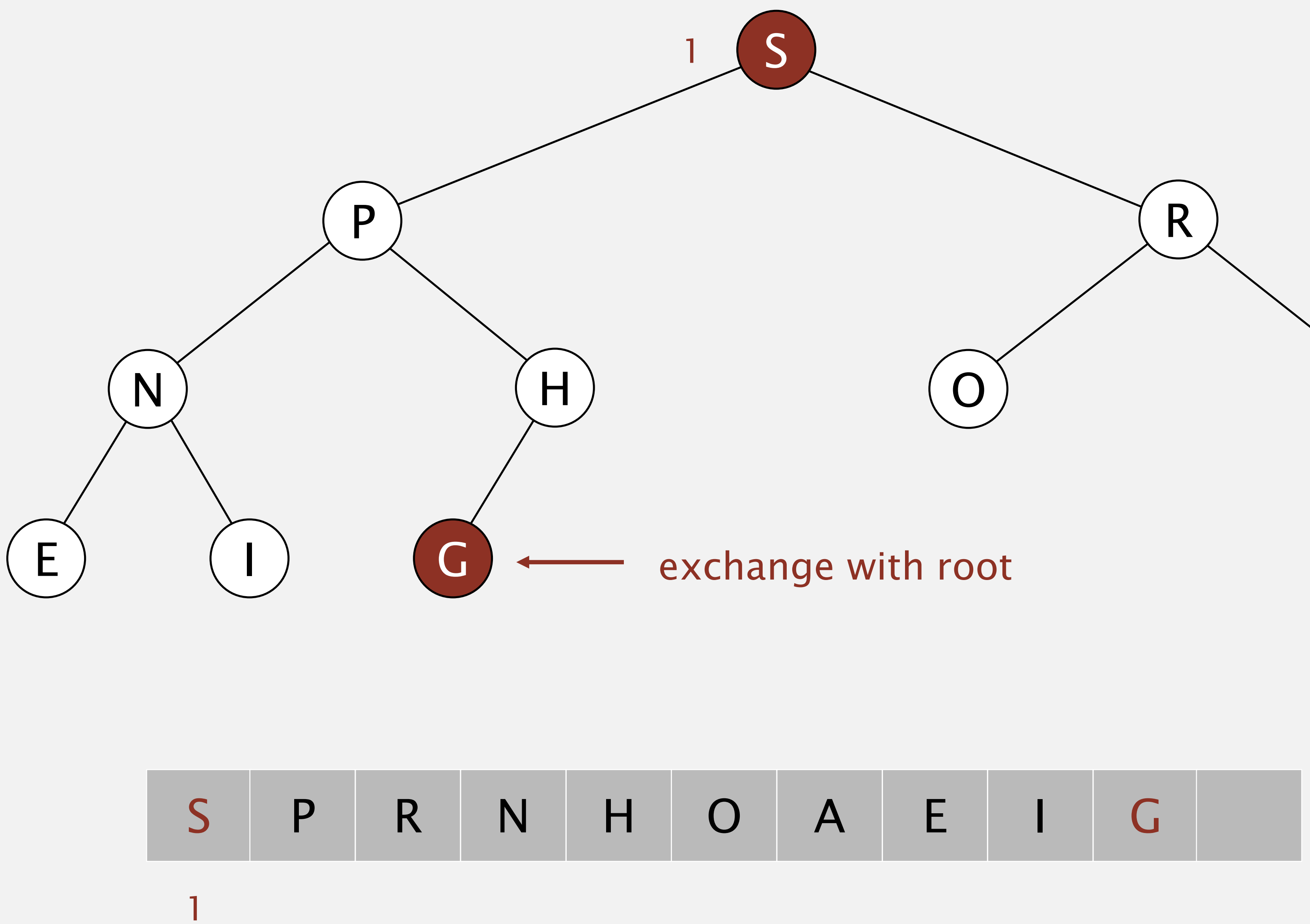


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

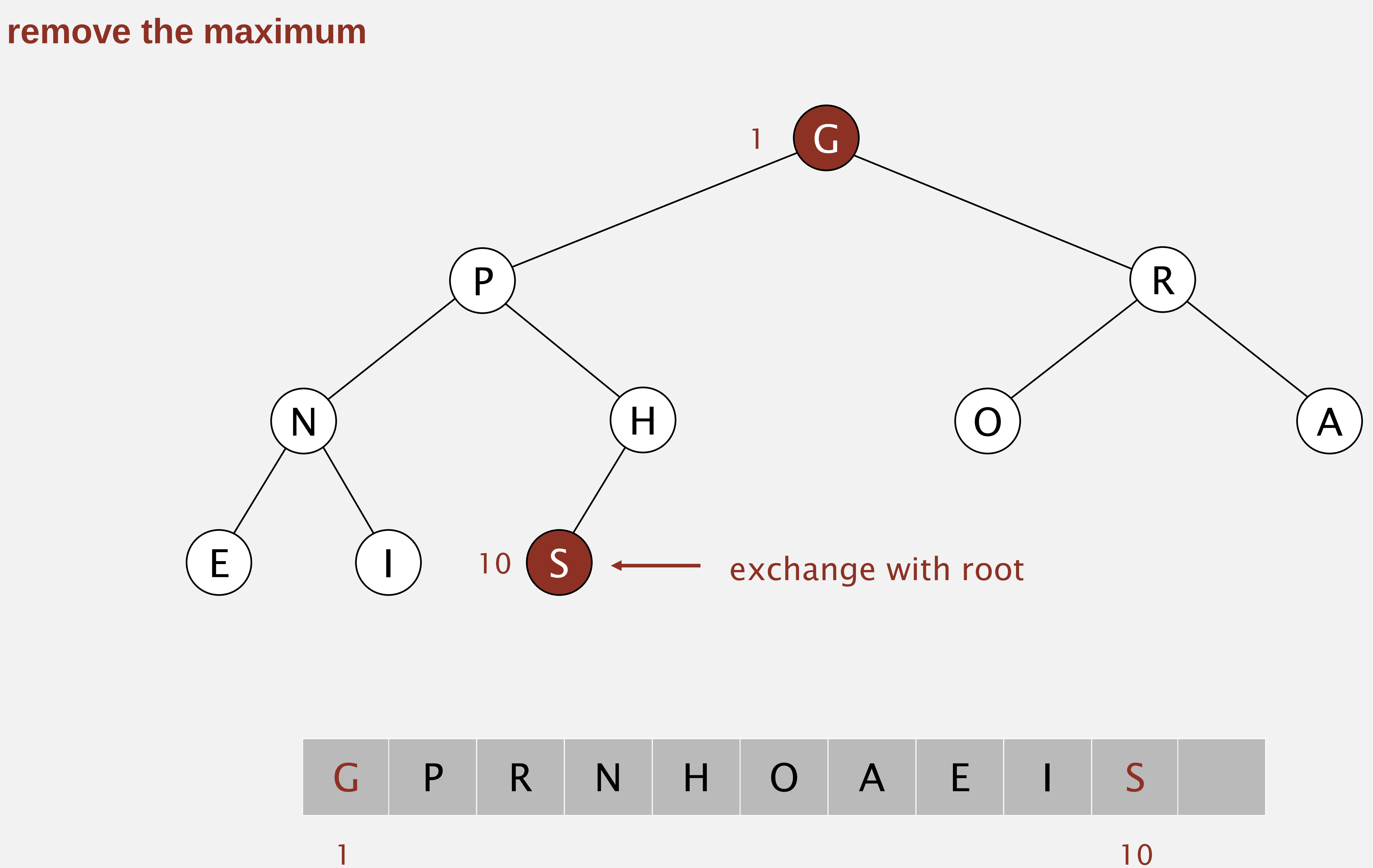
remove the maximum



Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

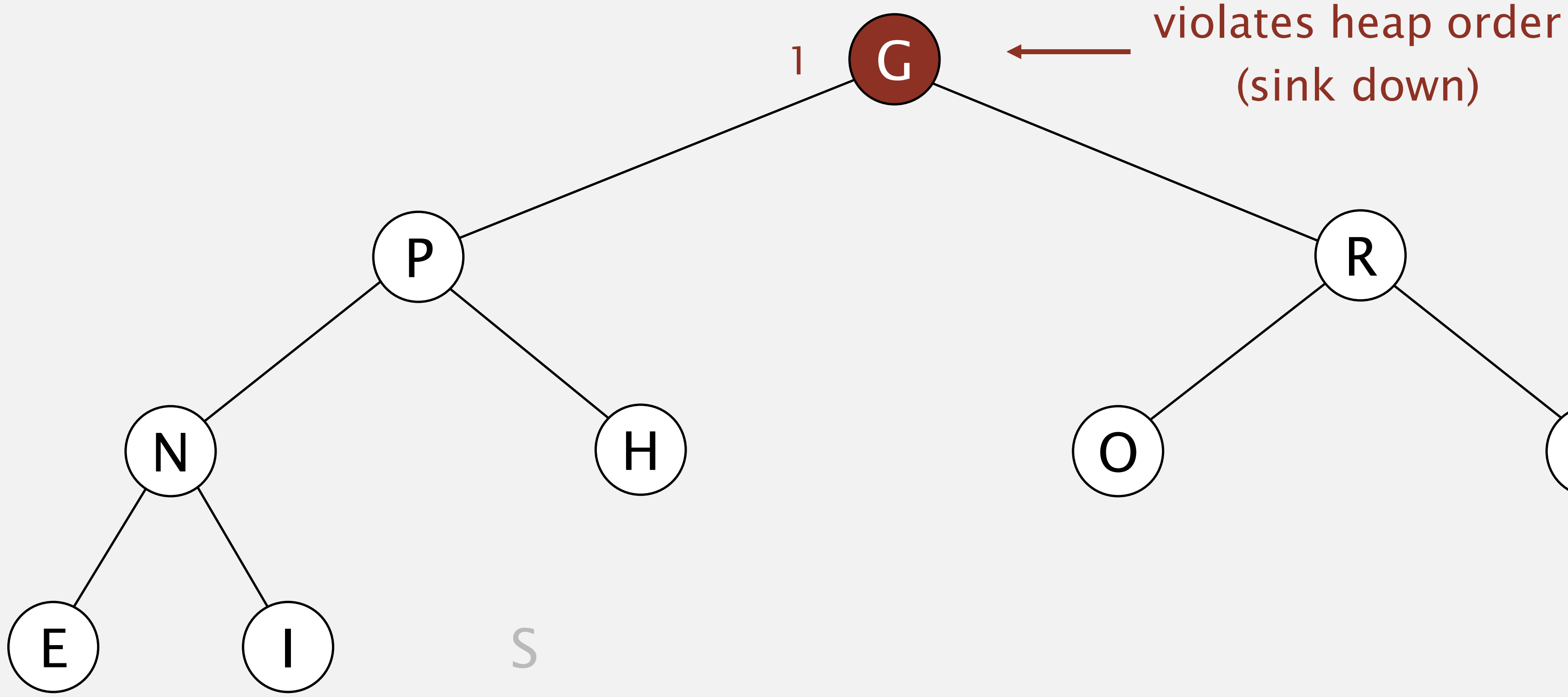


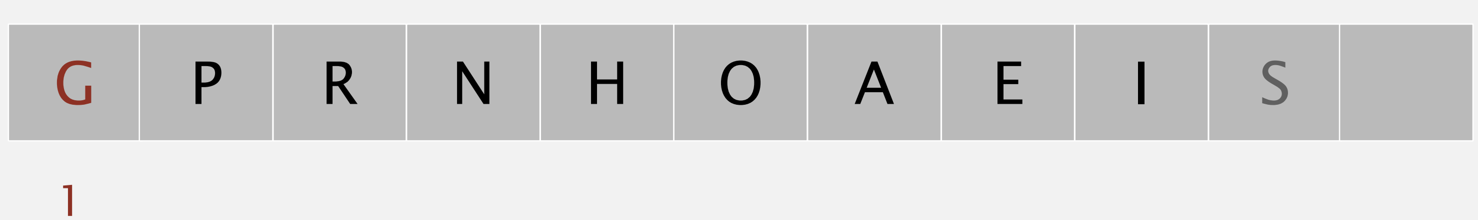
Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



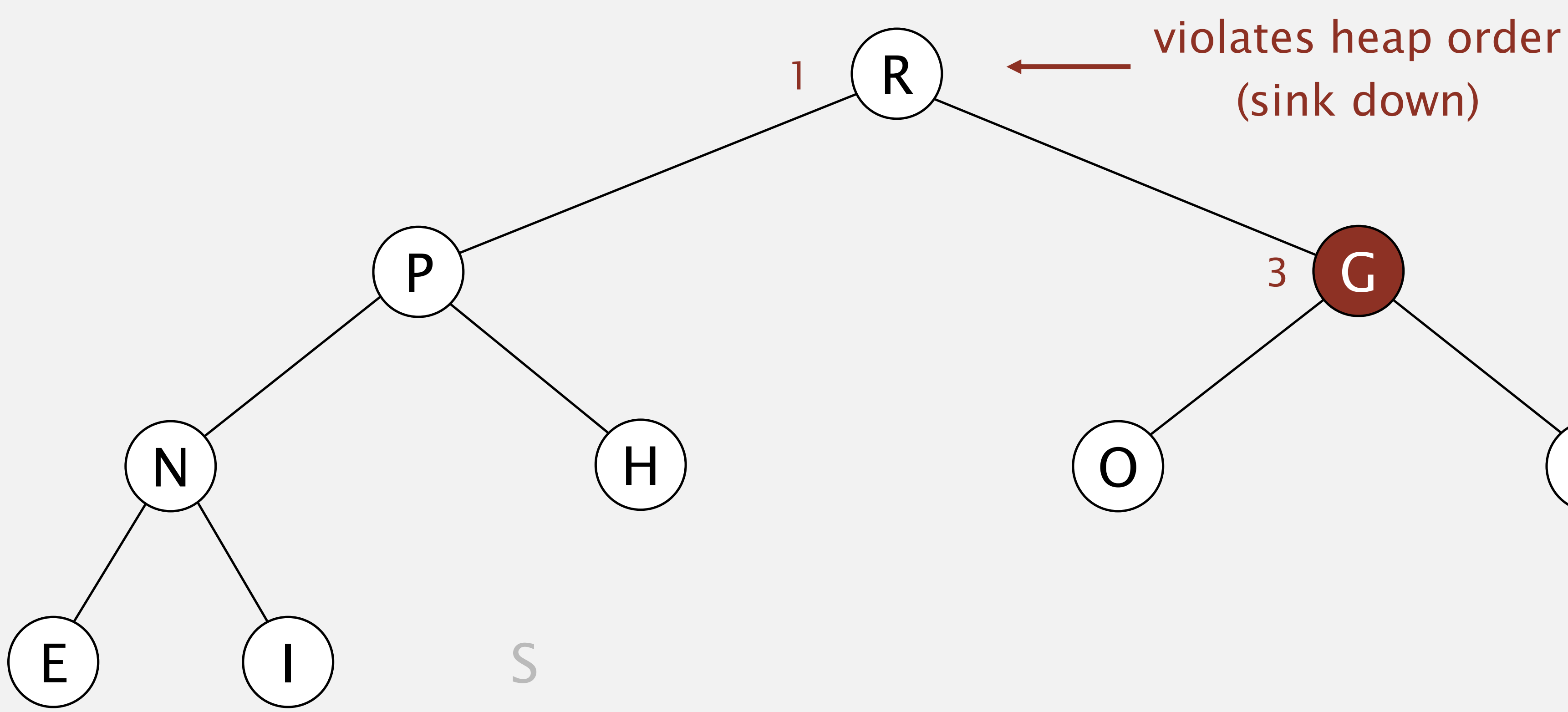


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



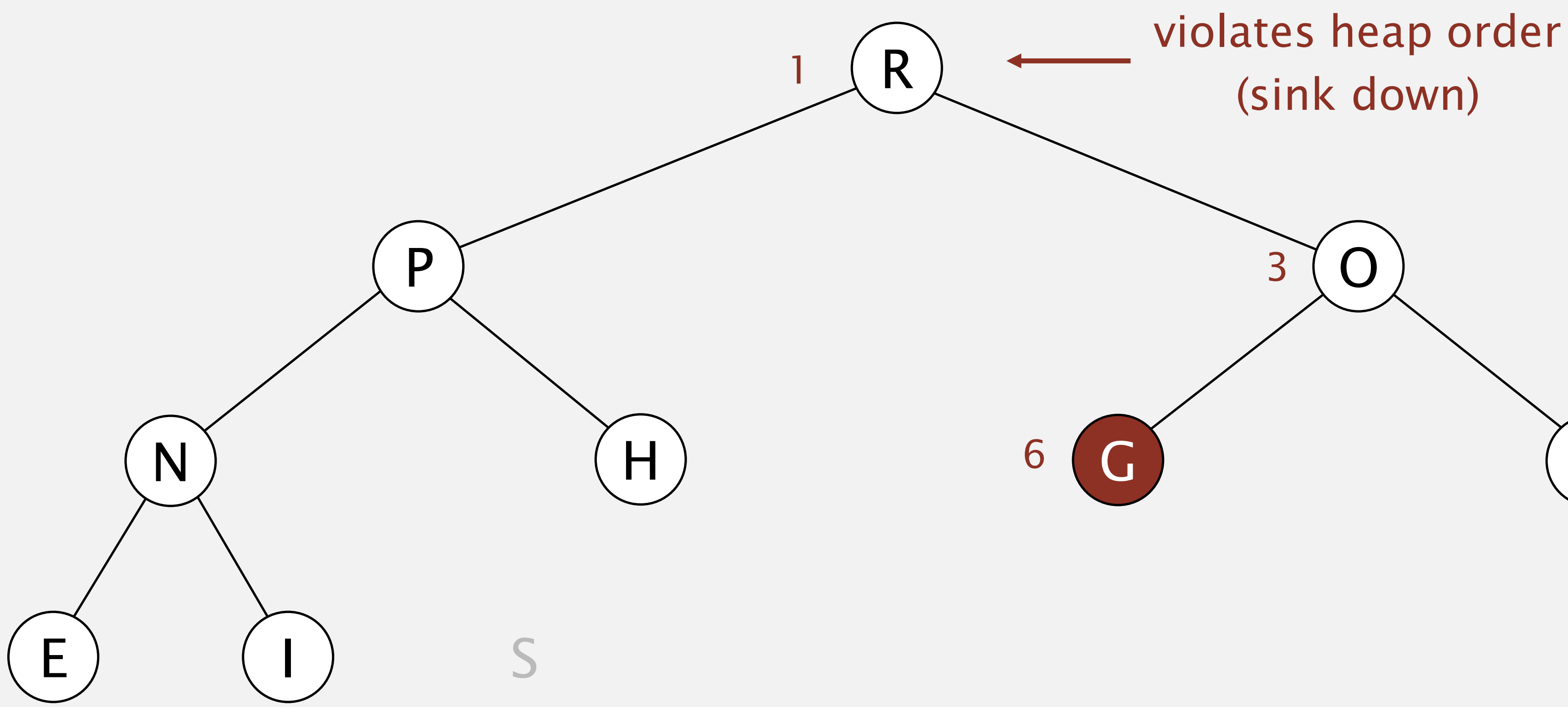
R	P	G	N	H	O	A	E	I	S	
1		3								

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



R	P	O	N	H	G	A	E	I	S	

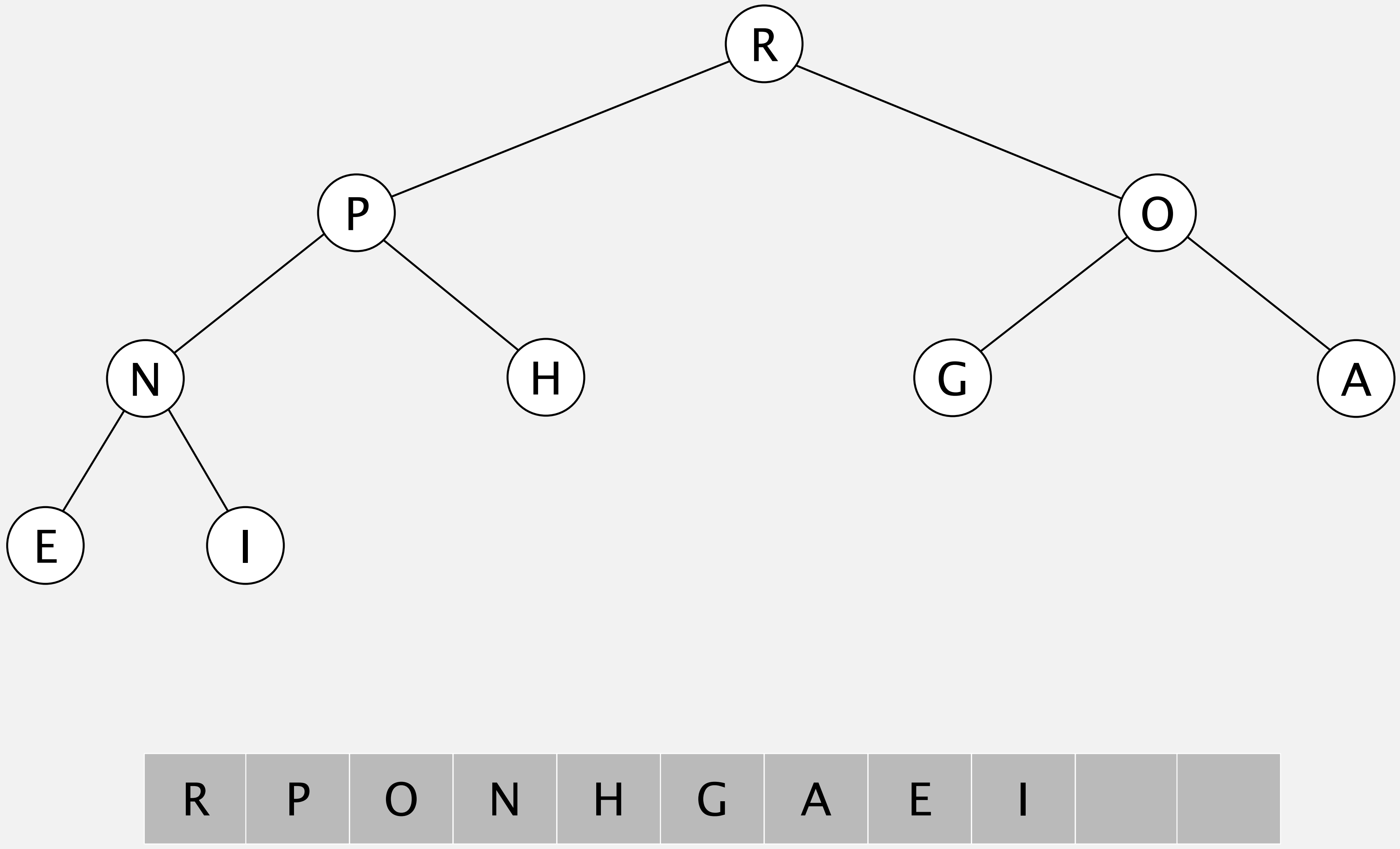
1 3 6

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

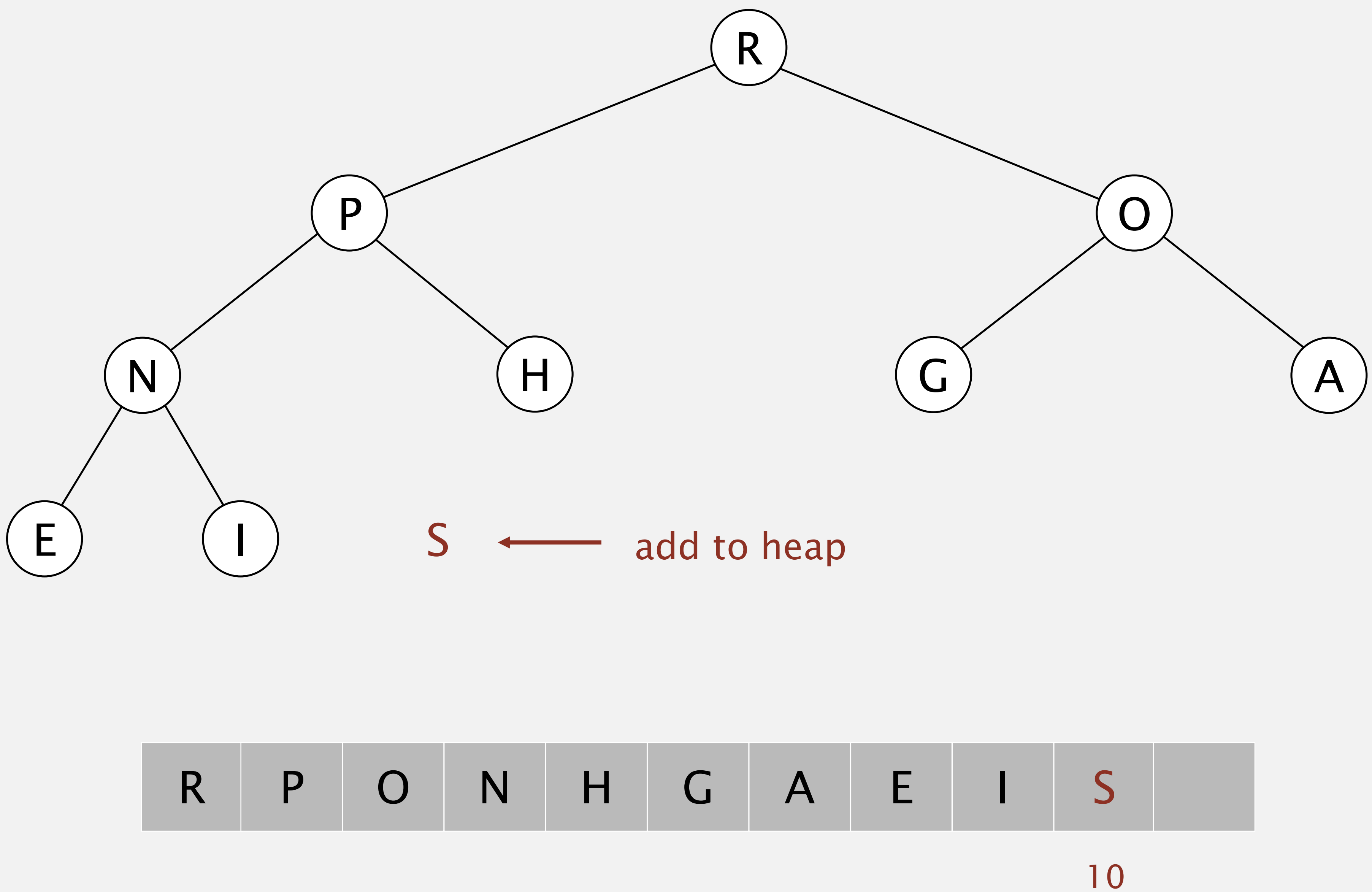


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S

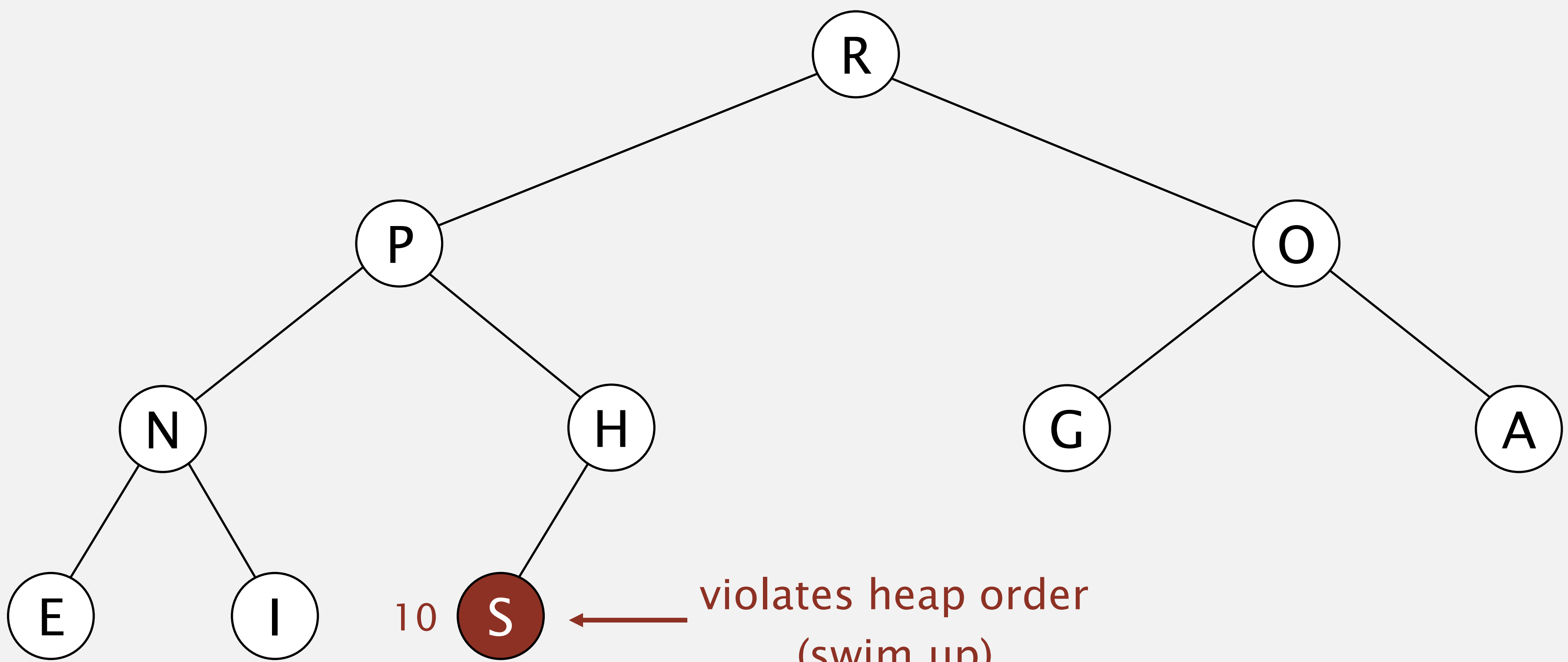


Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



R	P	O	N	H	G	A	E	I	S	

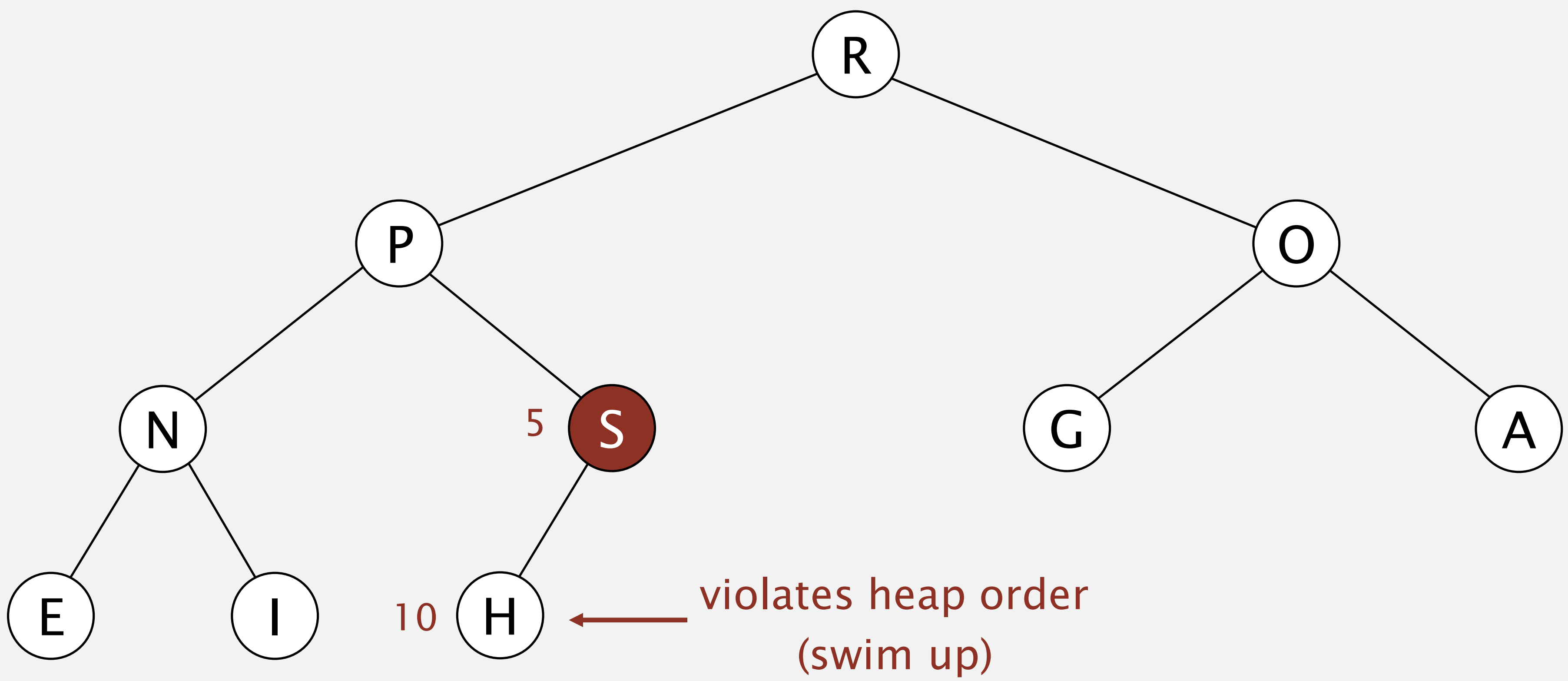
10

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



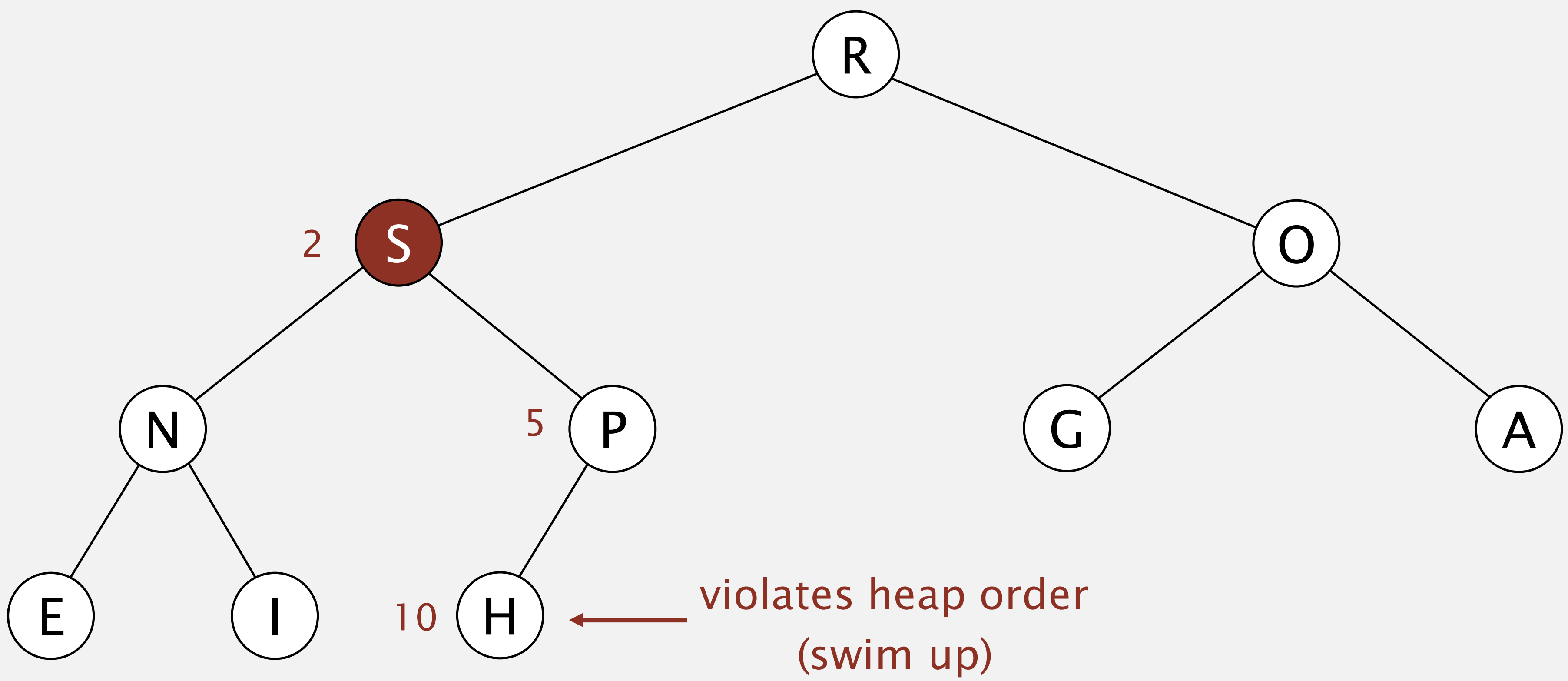
R	P	O	N	S	G	A	E	I	H	
				5					10	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



R	S	O	N	P	G	A	E	I	H	

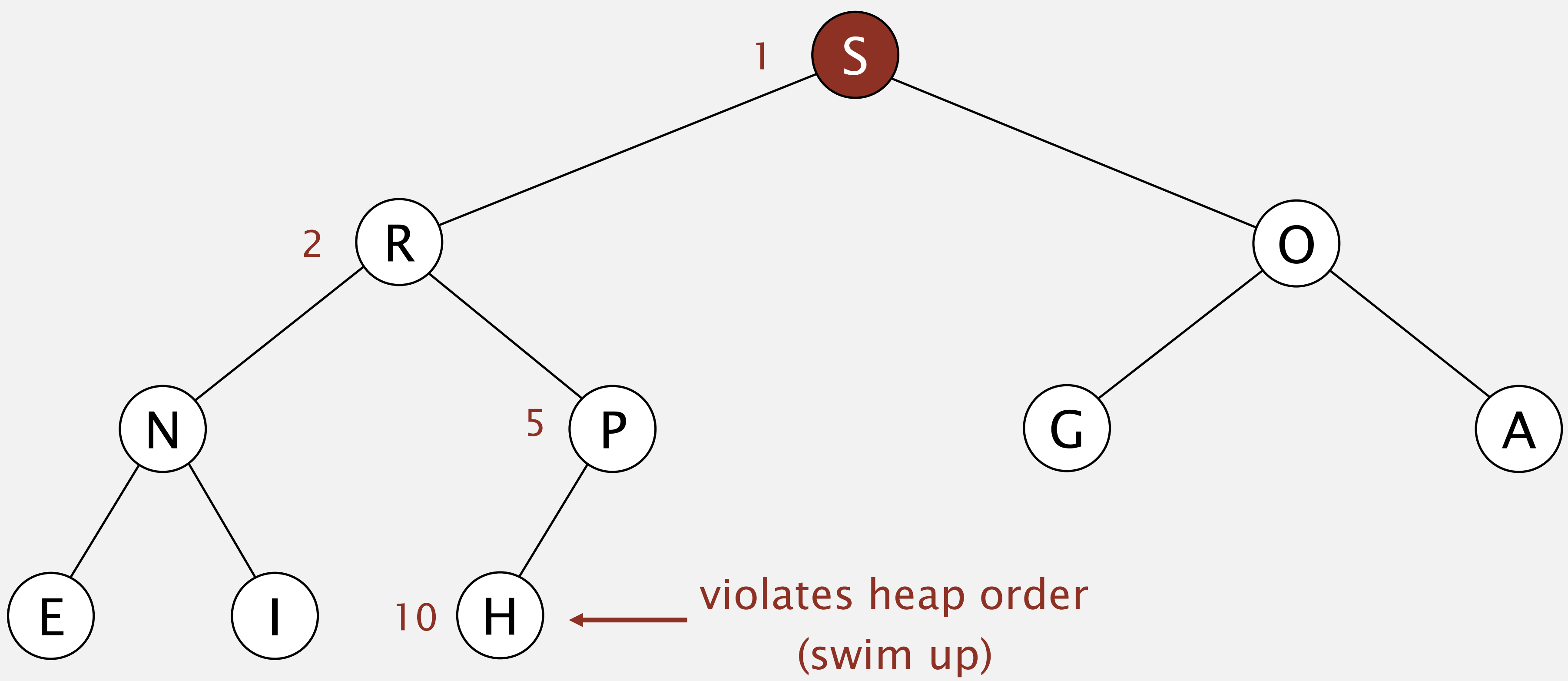
2 5 10

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



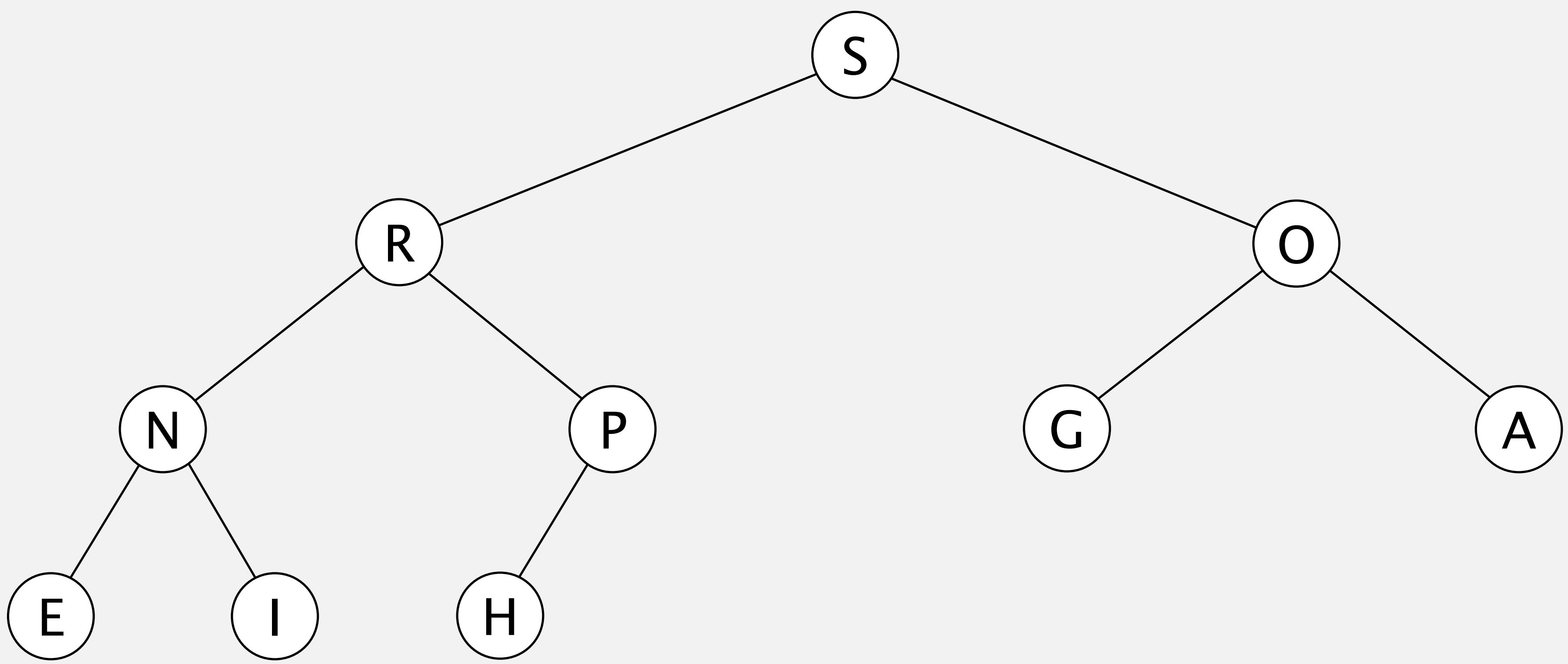
S	R	O	N	P	G	A	E	I	H	
1	2			5					10	

Binary heap demo

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



S	R	O	N	P	G	A	E	I	H	

Promotion in a heap

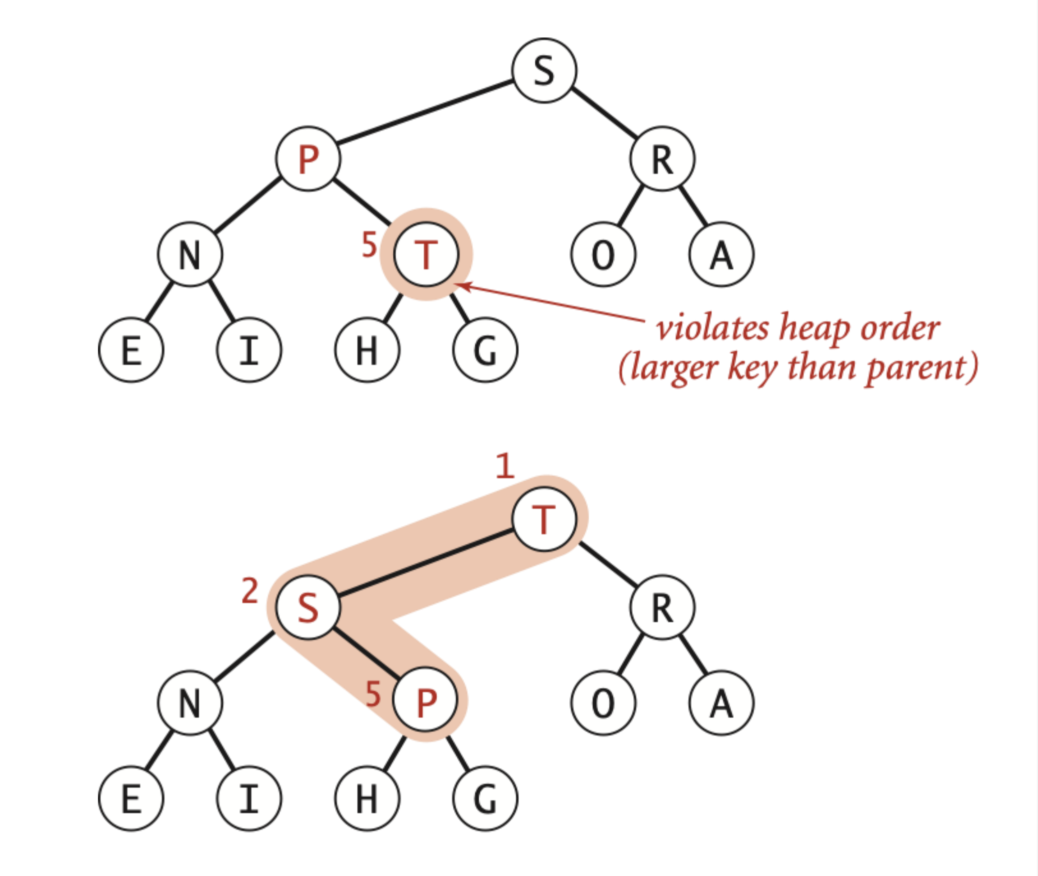
Scenario. Child's key becomes larger key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2



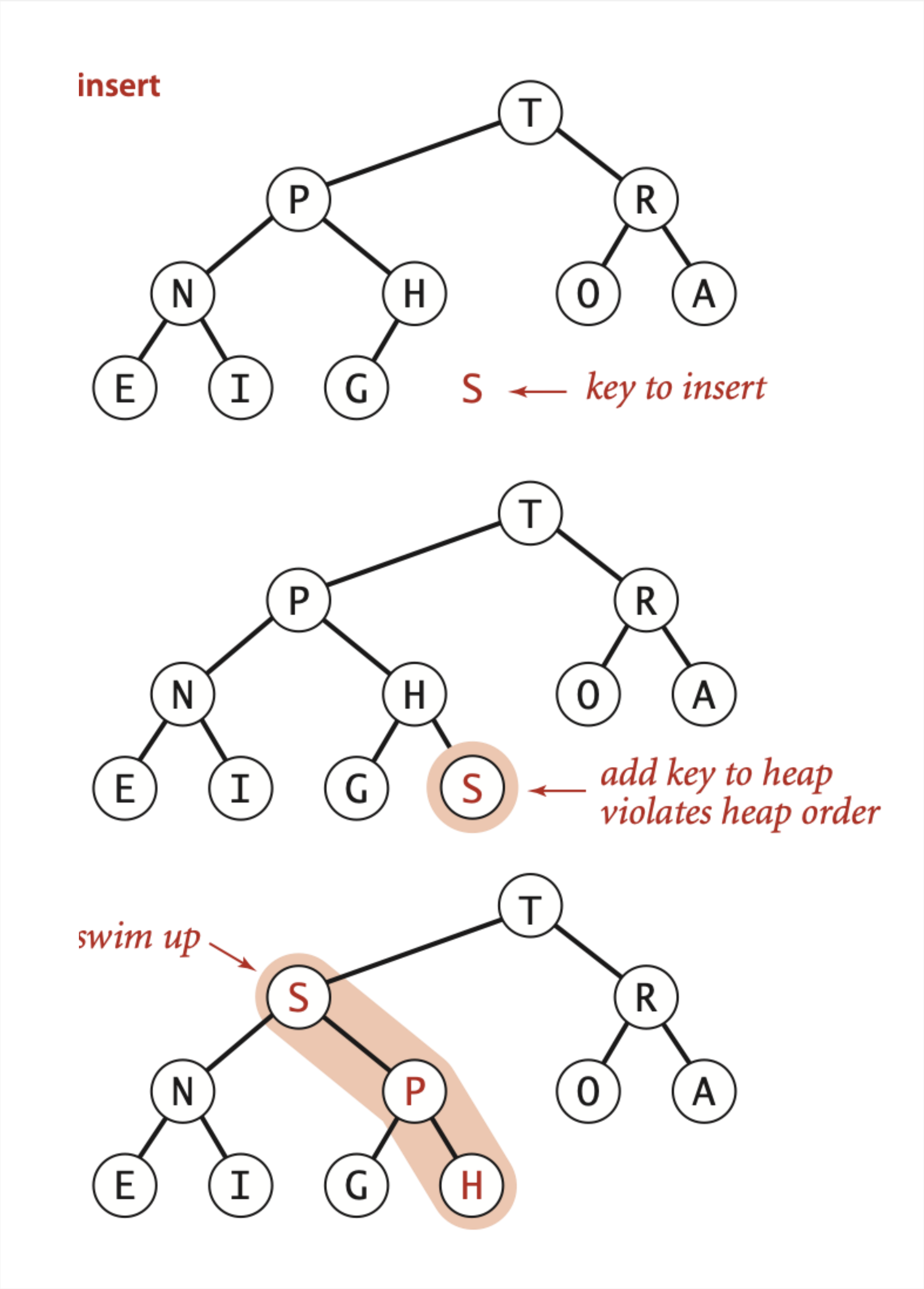
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.

Cost. At most $\lg N$ compares.

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```



Demotion in a heap

Scenario. Parent's key becomes smaller than one (or both) of its children's.

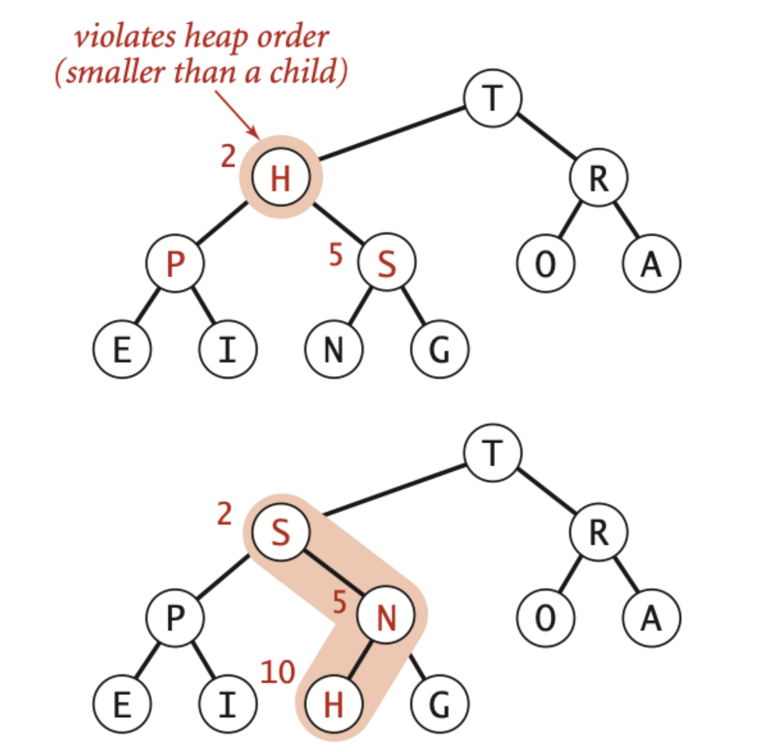
To eliminate the violation:

why not smaller child?

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node at k are 2k and 2k+1



Top-down reheapify (sink)

Power struggle. Better subordinate promoted.

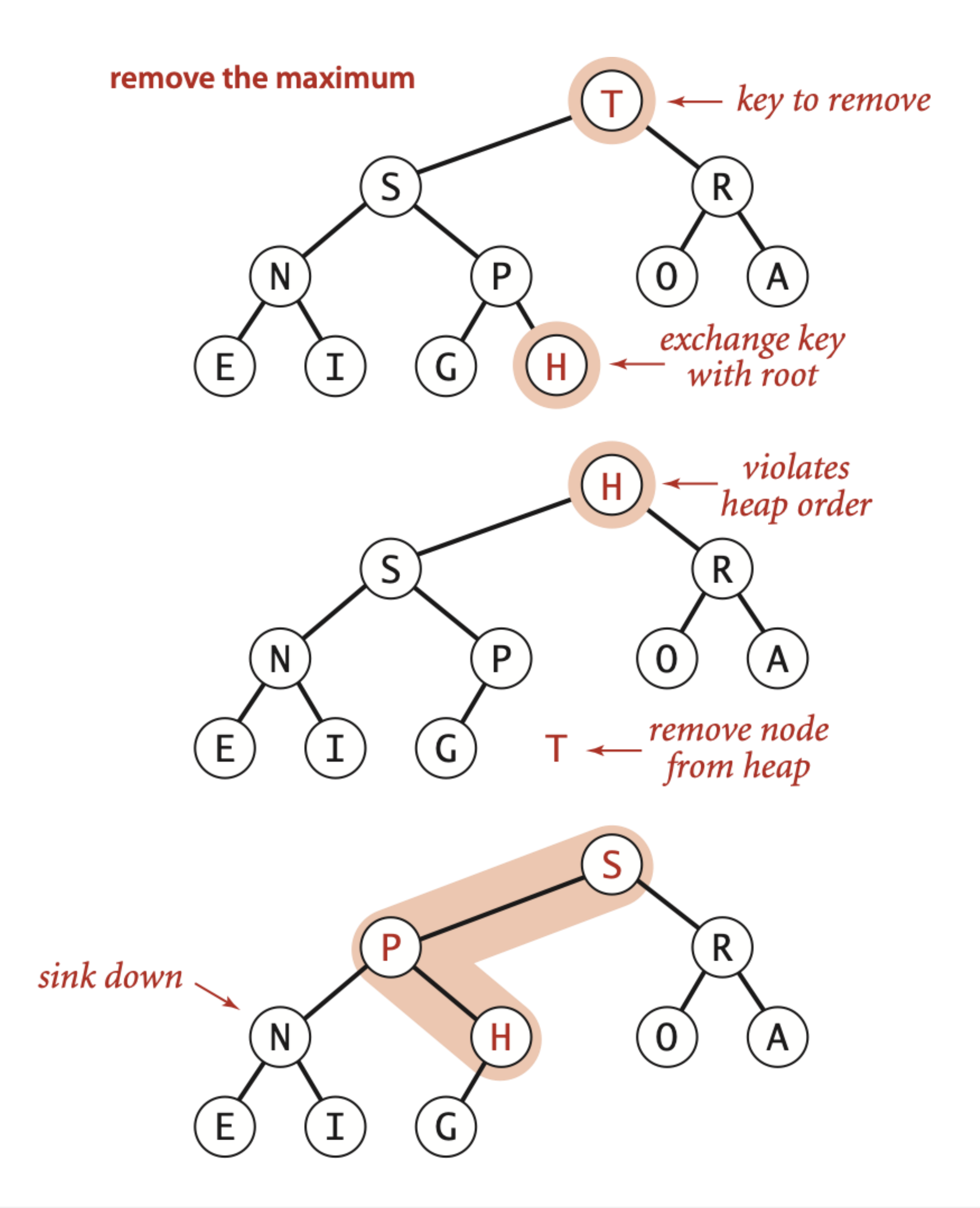
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

prevent loitering



Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;
   private int N;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {   return N == 0;   }
   public void insert(Key key)
   public Key delMax()
   {   /* see previous code */  }

   private void swim(int k)
   private void sink(int k)
   {   /* see previous code */  }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j]) < 0;  }
   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

Priority queues implementation cost summary

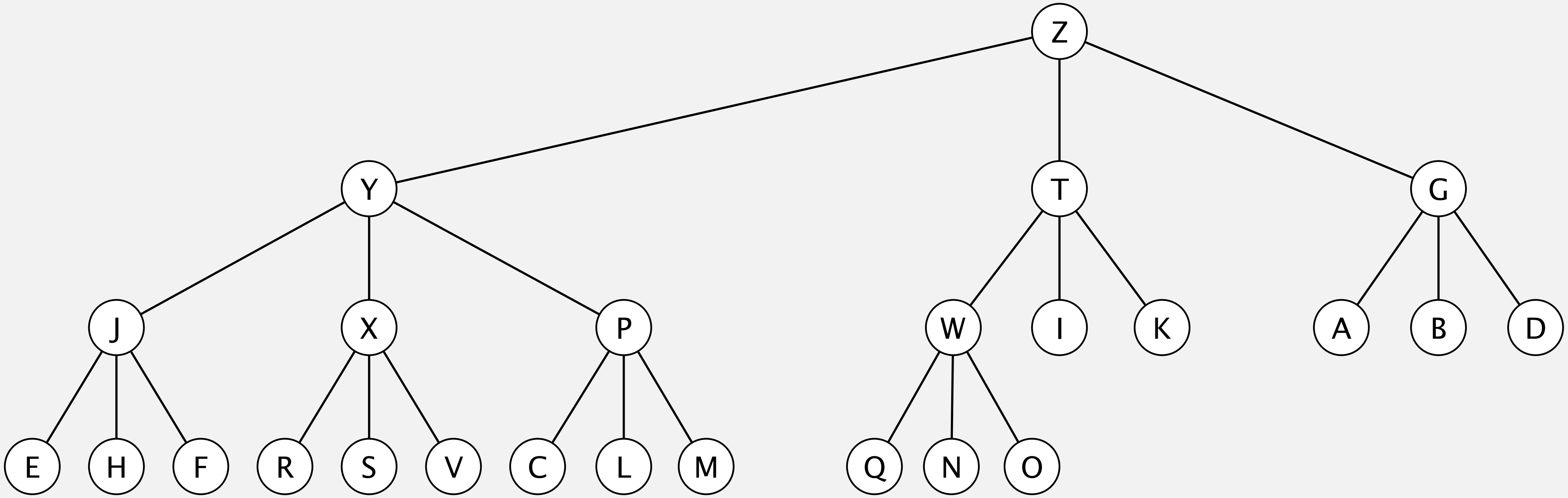
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log N$	$\log N$	1

order-of-growth of running time for priority queue with N items

Binary heap: practical improvements

Multiway heaps.

- Complete d-way tree.
- Parent's key no smaller than its children's keys.
- Swim takes $\log_d N$ compares; sink takes $d \log_d N$ compares.



3-way heap

Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log N$	$\log N$	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	$\log N$ †	1
Brodal queue	1	$\log N$	1
impossible	1	1	1

why impossible?

† amortized

order-of-growth of running time for priority queue with N items

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N
amortized time per op
(how to make worst case?)

Minimum-oriented priority queue.

- Replace `less()` with `greater()`.
- Implement `greater()`.

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.

can implement efficiently with `sink()` and `swim()`

http://algs4.cs.princeton.edu

Priority Queues

- *API and elementary implementations*
- *binary heaps*
- *heapsort*

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Q. What are its properties?

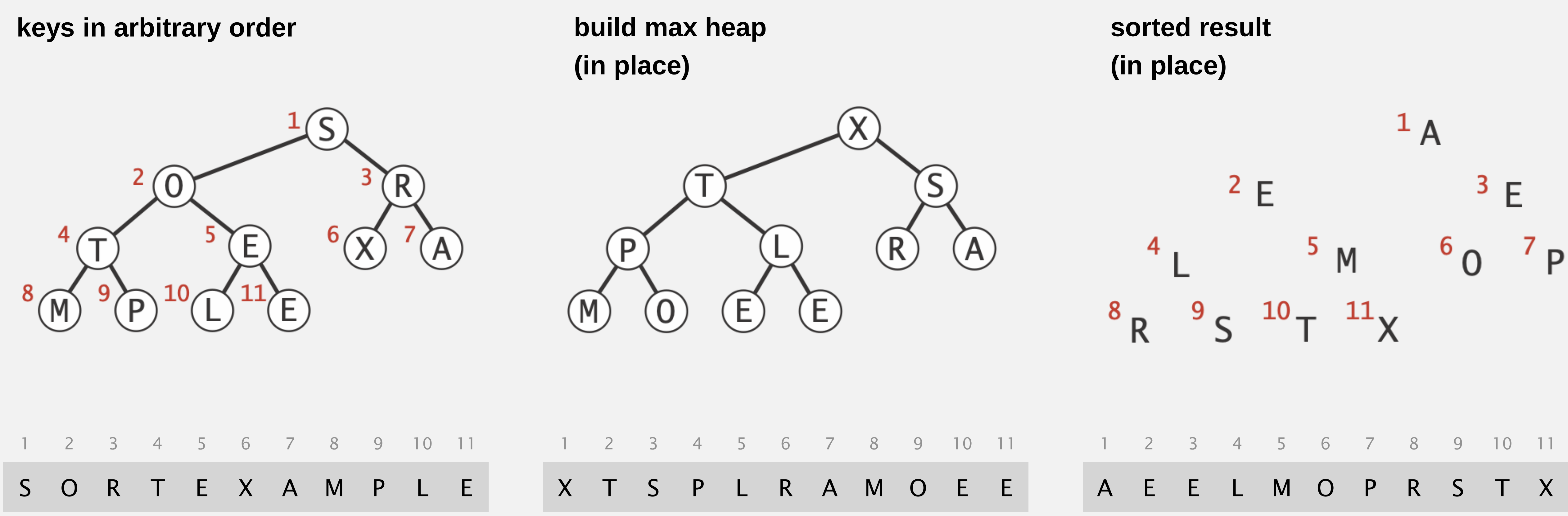
A. $N \log N$, extra array of length N, not stable.

Heapsort intuition. A heap is an array; do sort in place.

Heapsort

Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.
- Sortdown: repeatedly remove the maximum key.



Heapsort: Java implementation

```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int k = N/2; k >= 1; k--)
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   private static void sink(Comparable[] a, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Object[] a, int i, int j)
```

but make static (and pass arguments)

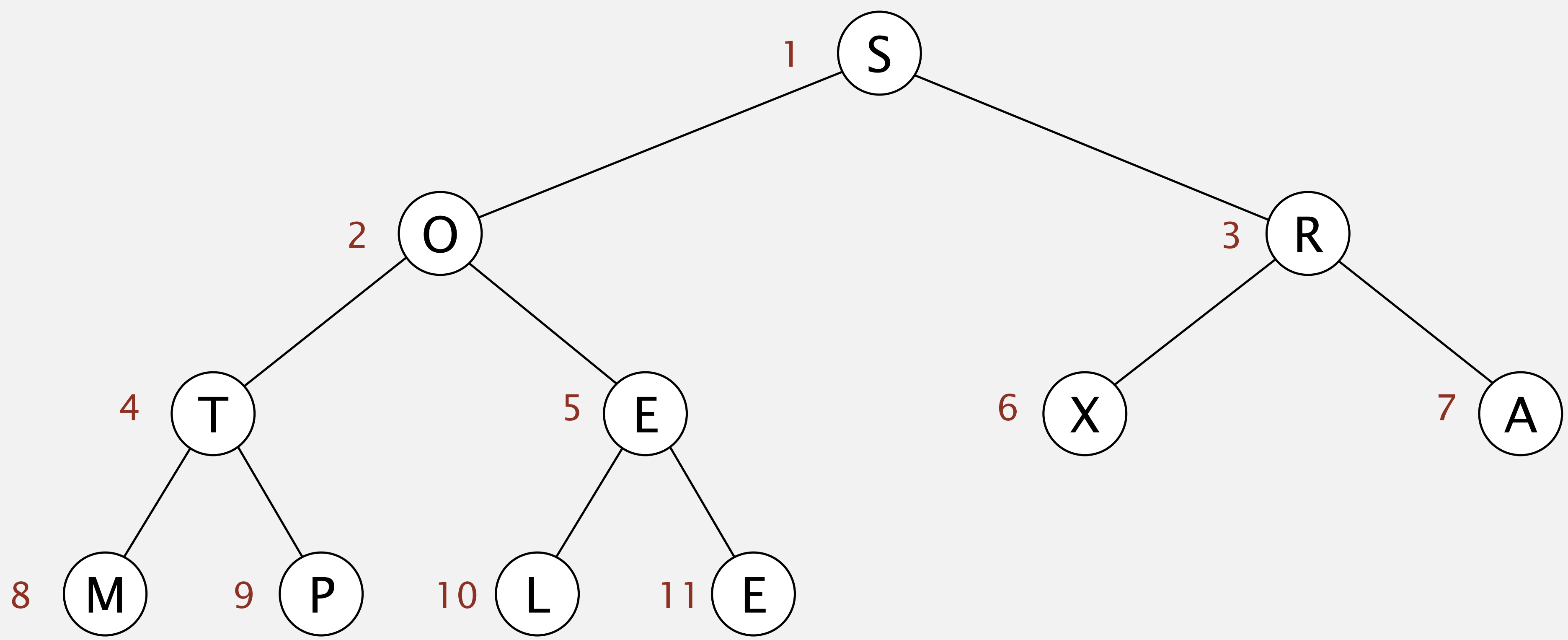
but convert from 1-based indexing to 0-base indexing

Heapsort demo

Heap construction. Build max heap using bottom-up method.

we assume array entries are indexed 1 to N

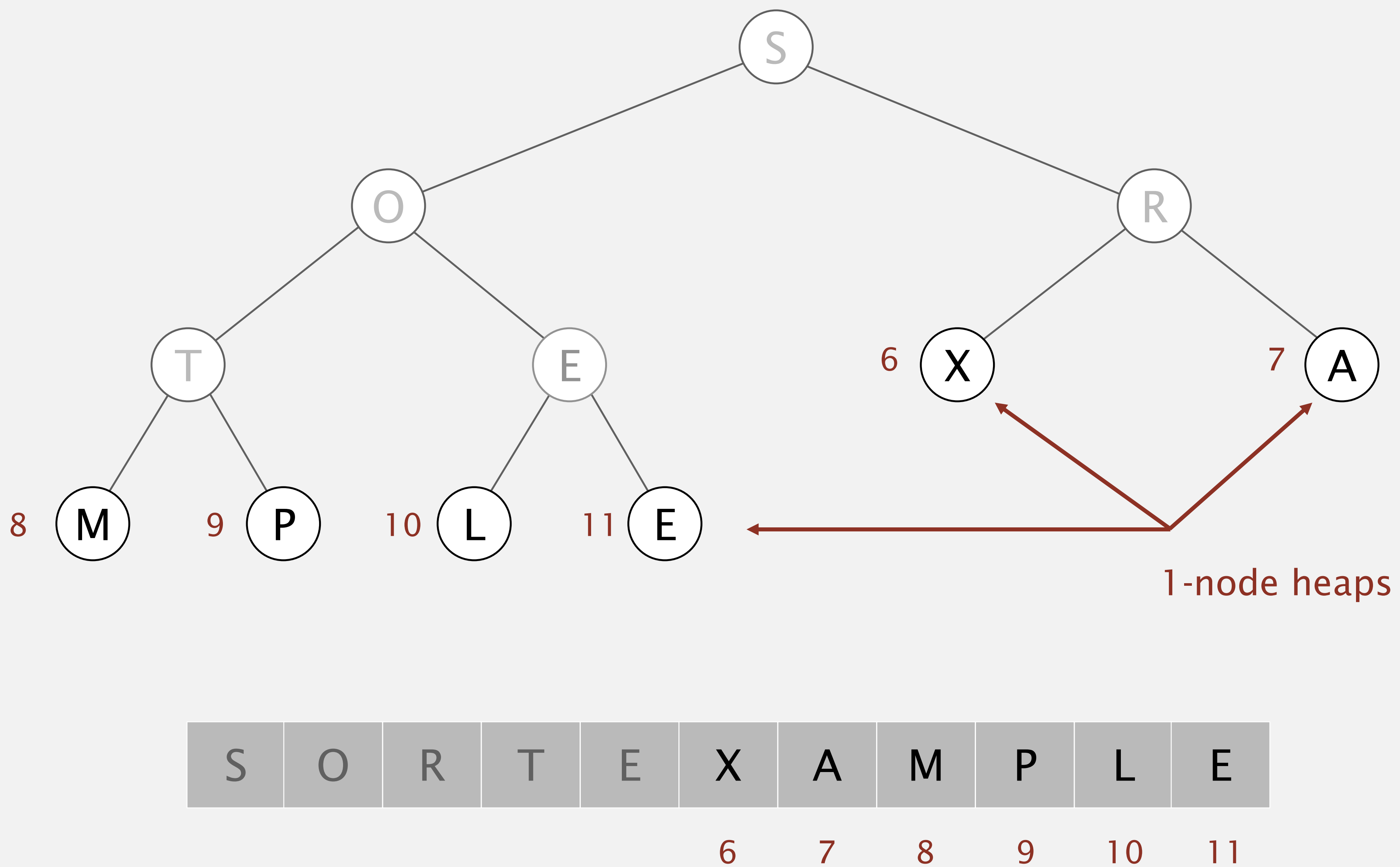
array in arbitrary order



S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort demo

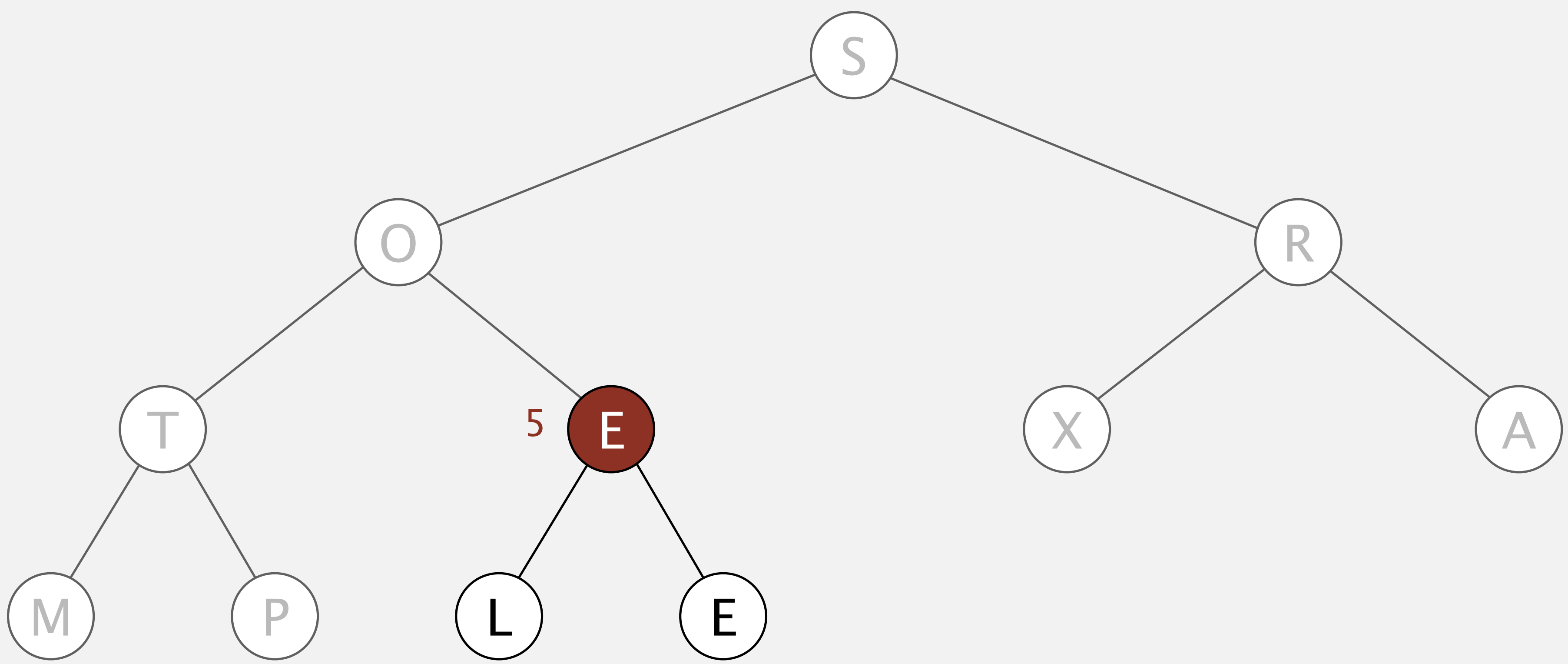
Heap construction. Build max heap using bottom-up method.



Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 5

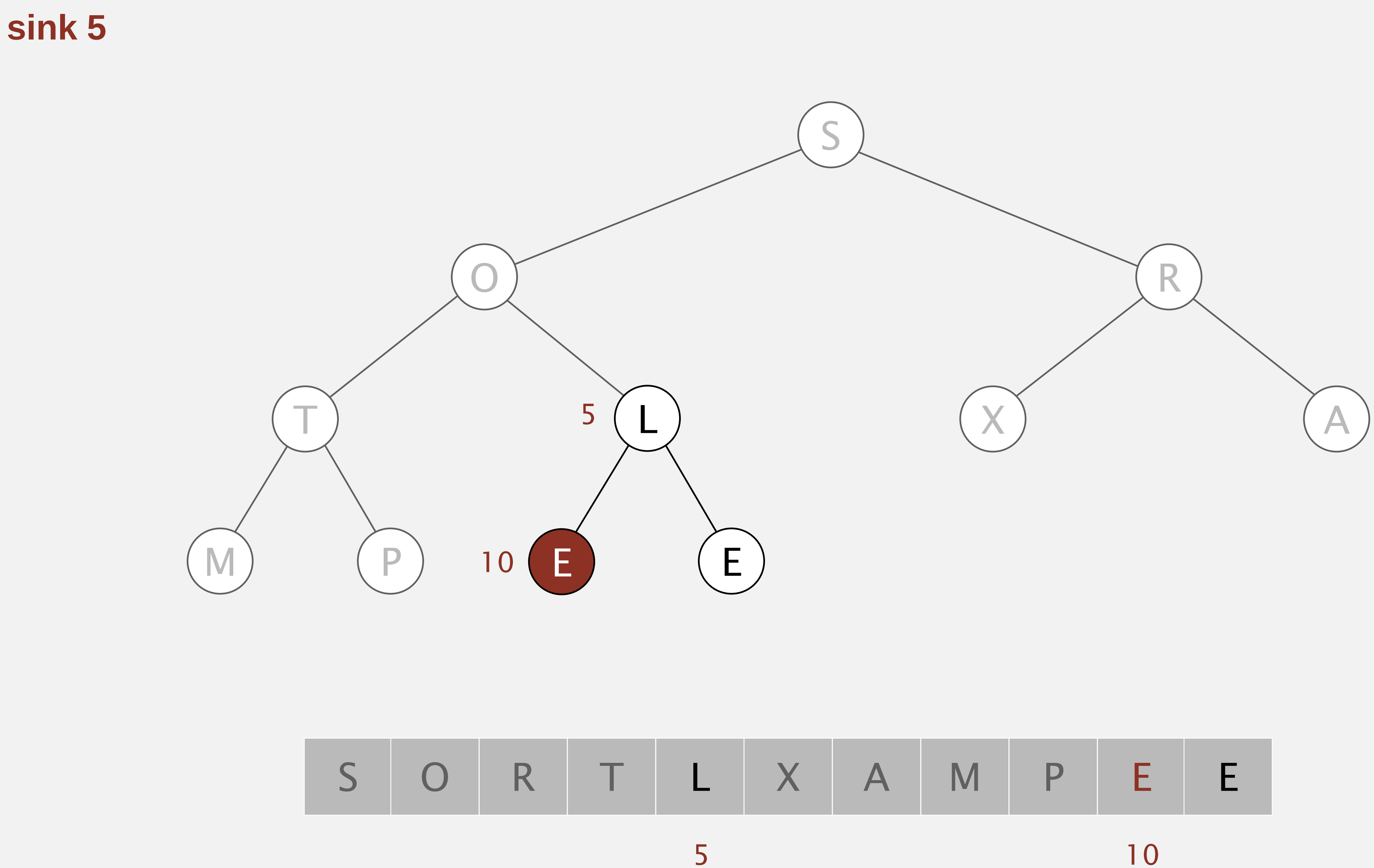


S	O	R	T	E	X	A	M	P	L	E

5

Heapsort demo

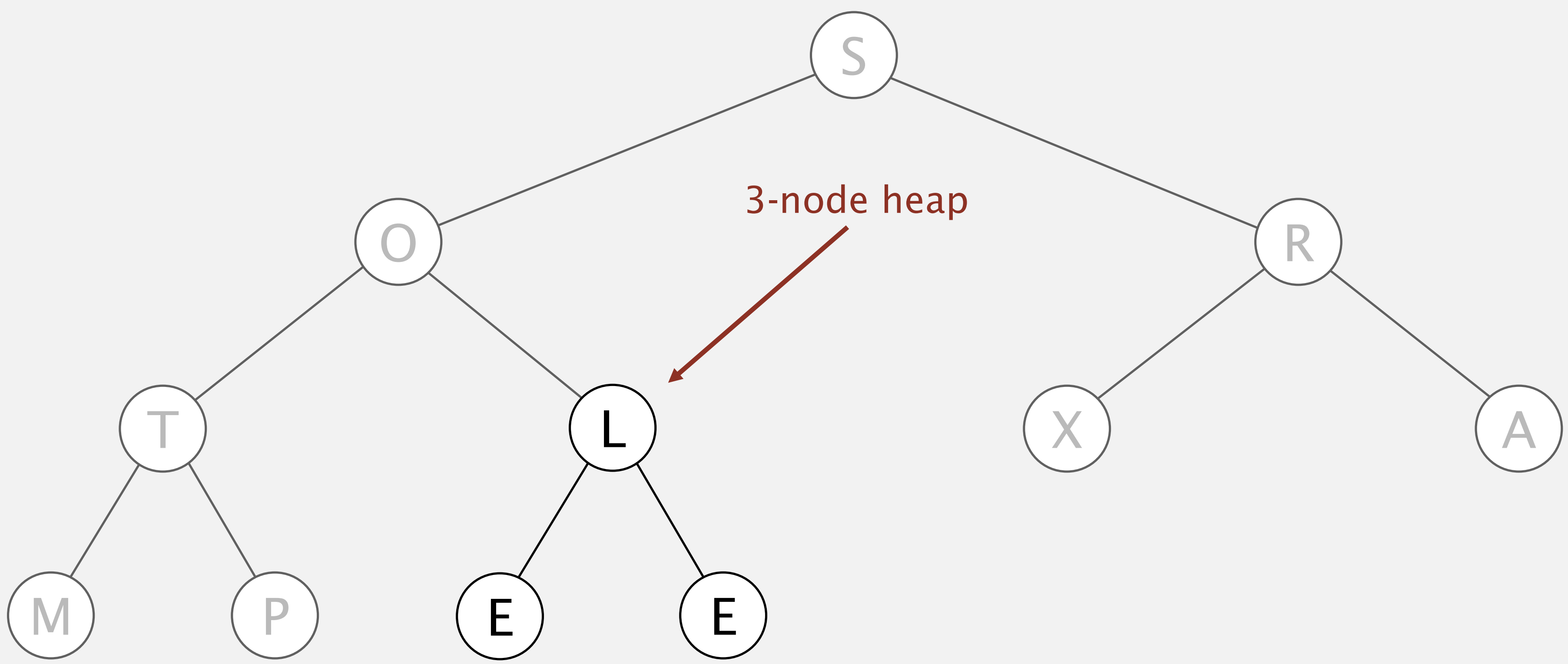
Heap construction. Build max heap using bottom-up method.



Heapsort demo

Heap construction. Build max heap using bottom-up method.

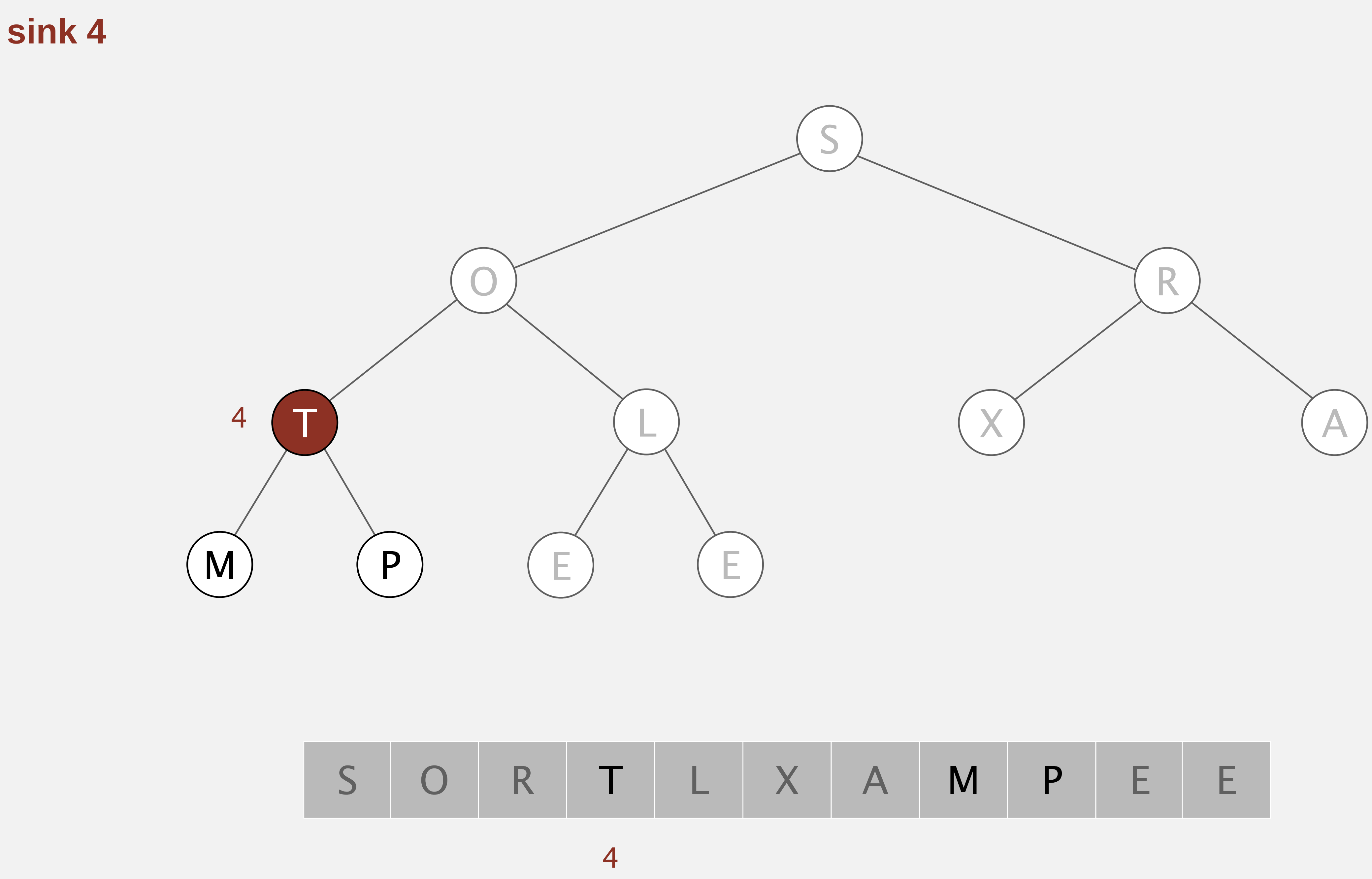
sink 5



S	O	R	T	L	X	A	M	P	E	E

Heapsort demo

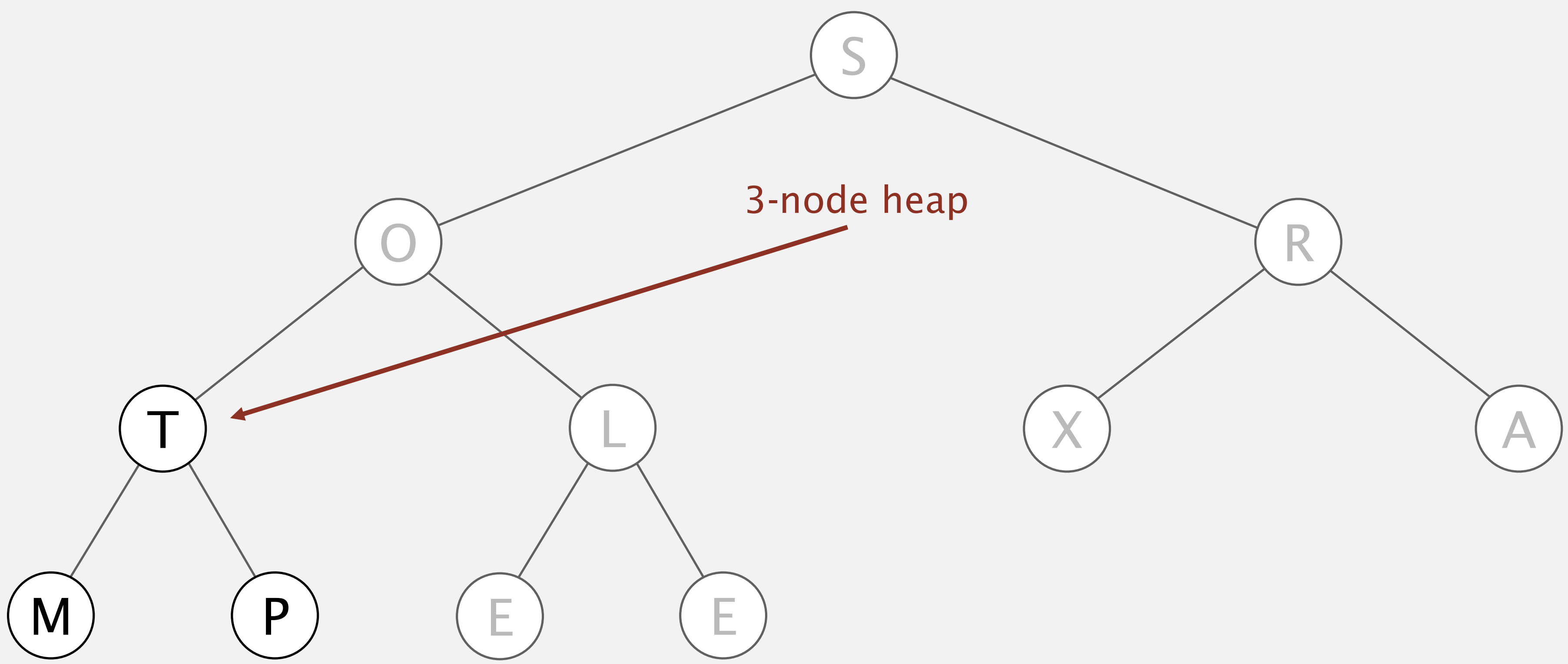
Heap construction. Build max heap using bottom-up method.



Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 4

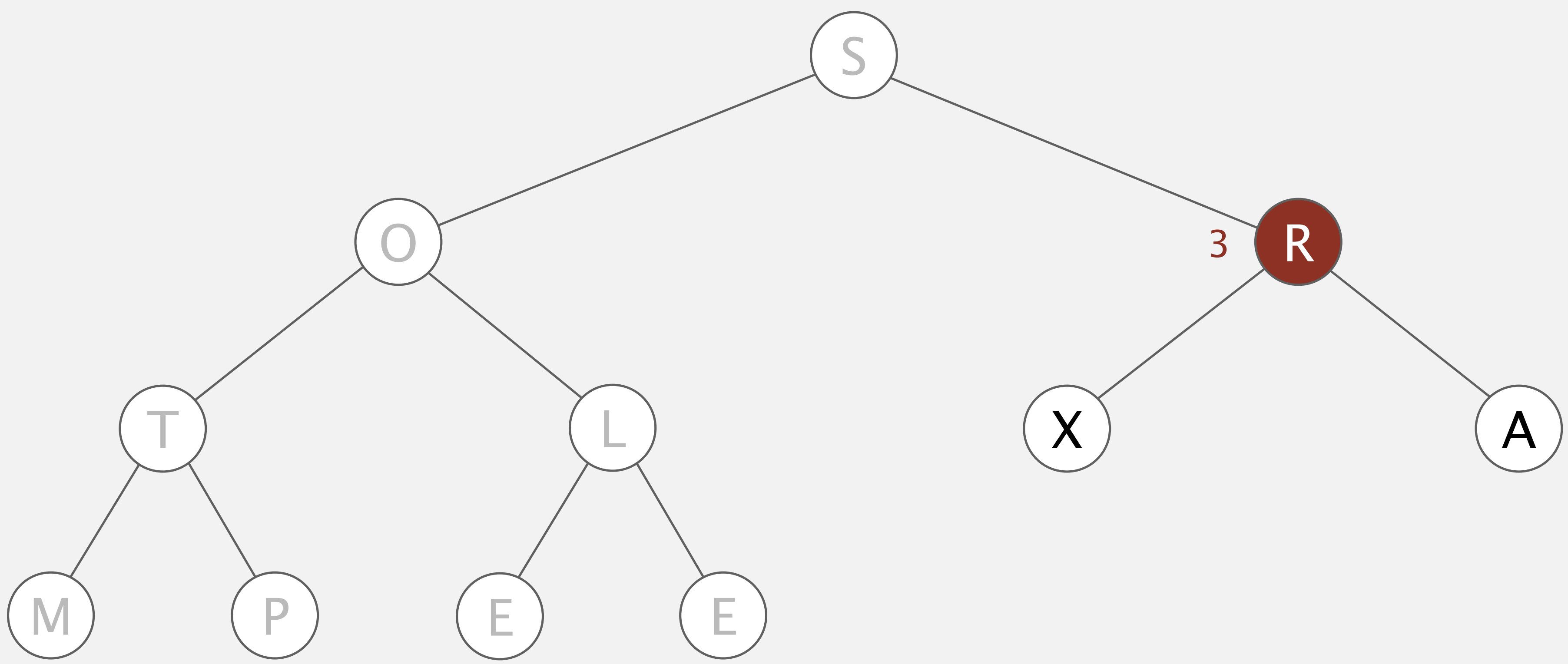


S	O	R	T	L	X	A	M	P	E	E

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

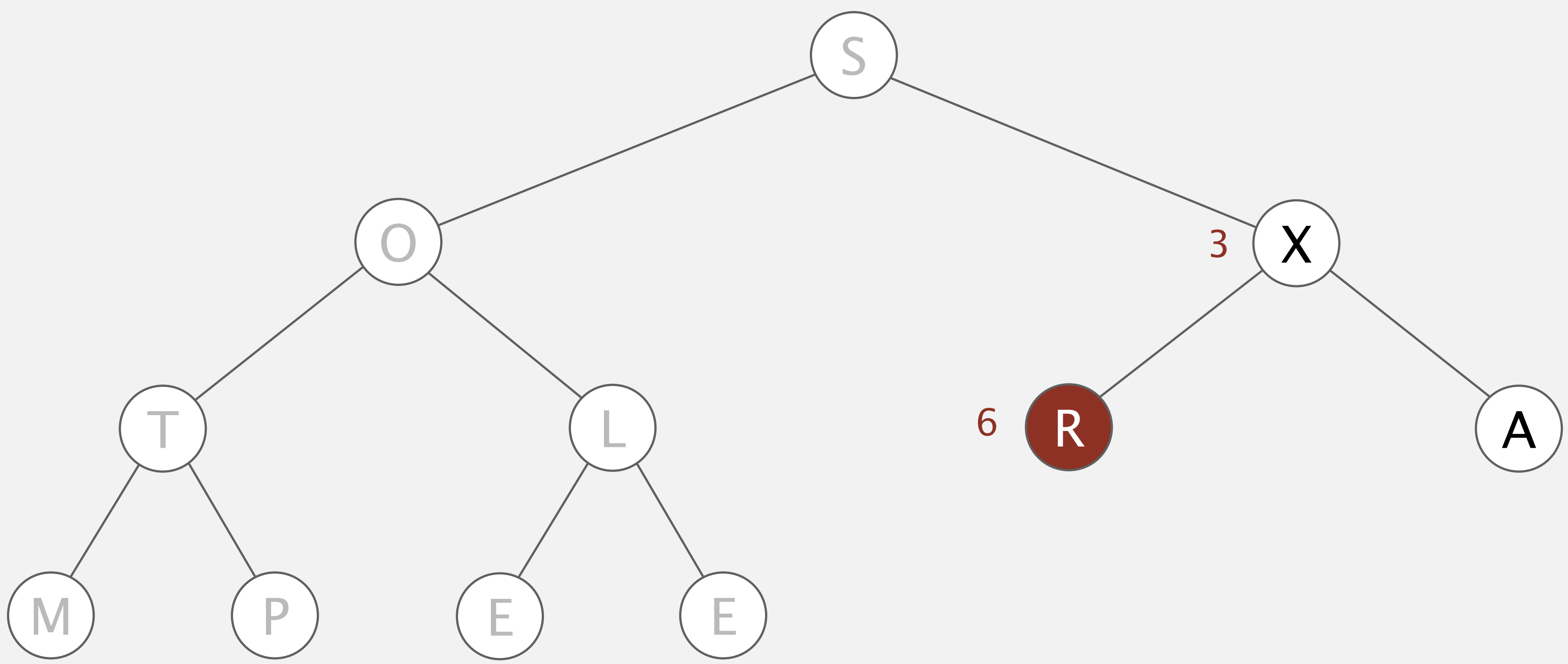


S	O	R	T	L	X	A	M	P	E	E
		3								

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

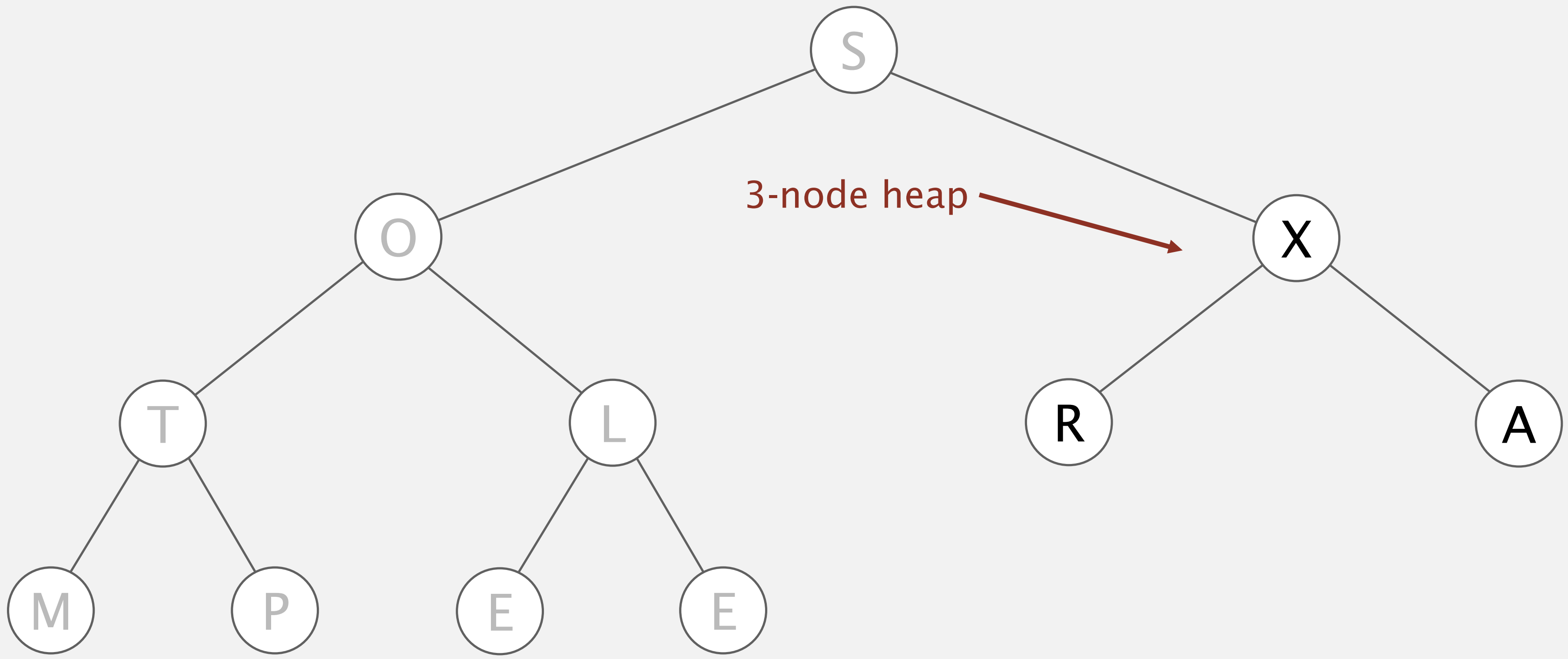


S	O	X	T	L	R	A	M	P	E	E
		3			6					

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 3

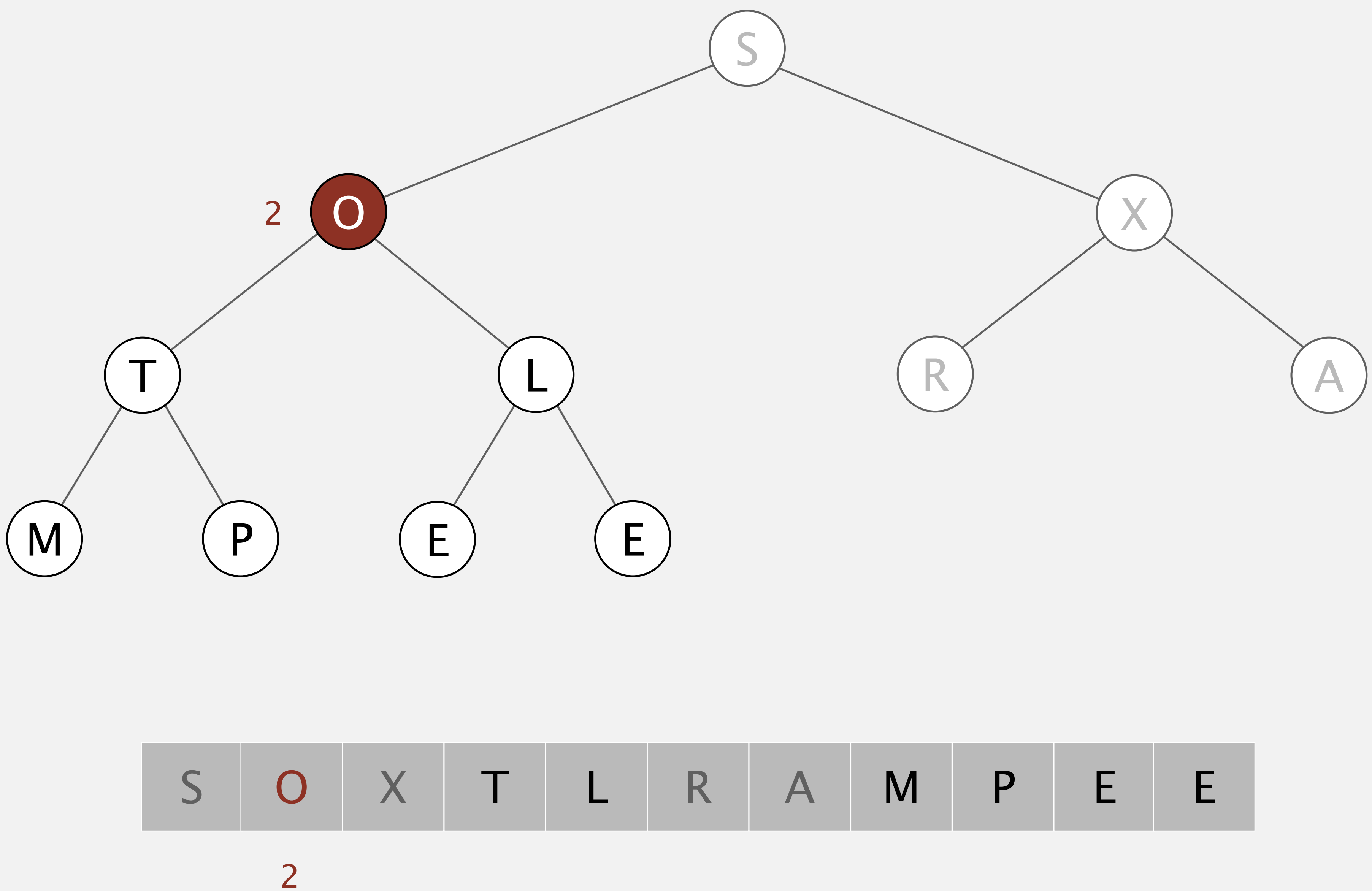


S	O	X	T	L	A	A	M	P	E	E

Heapsort demo

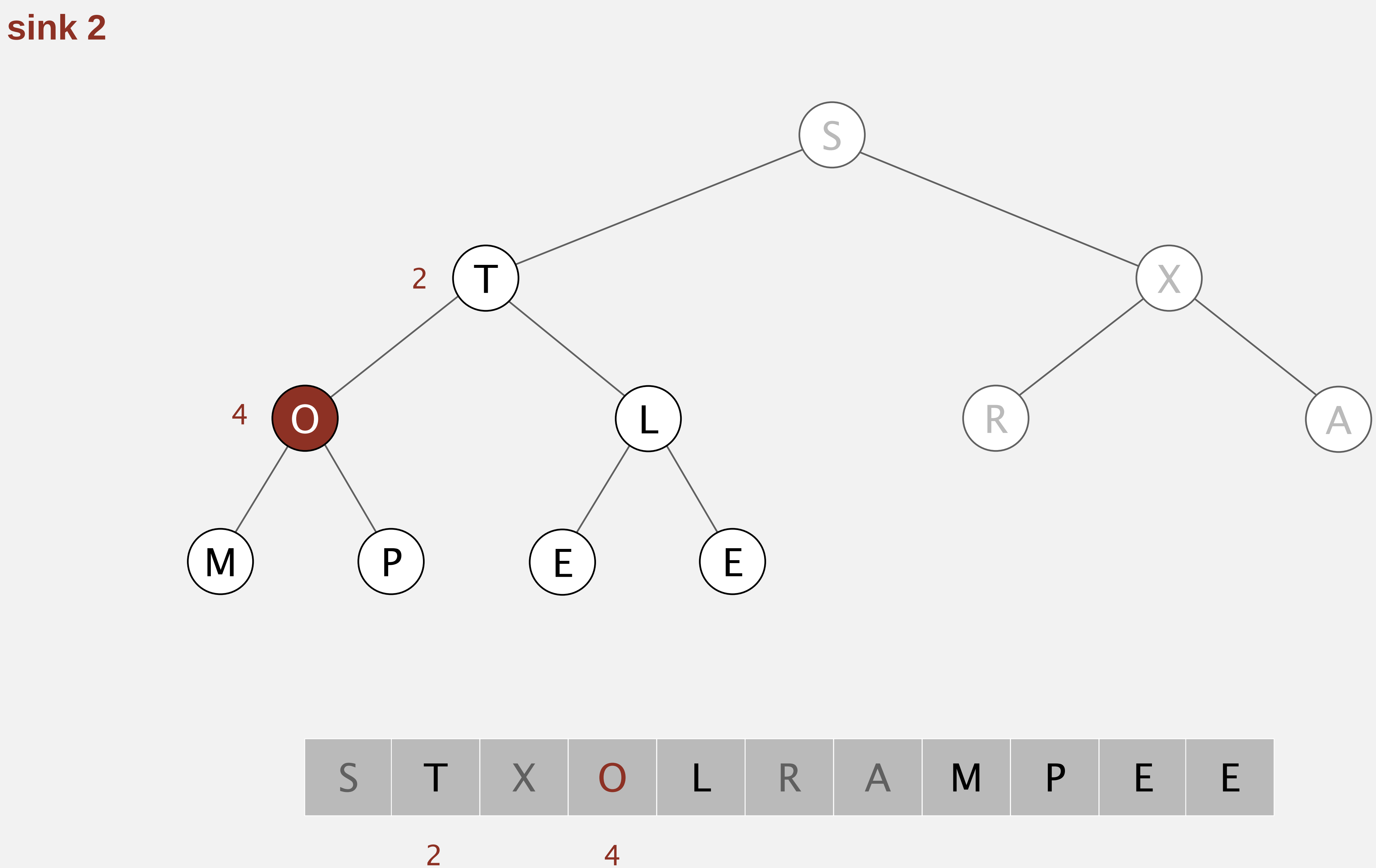
Heap construction. Build max heap using bottom-up method.

sink 2



Heapsort demo

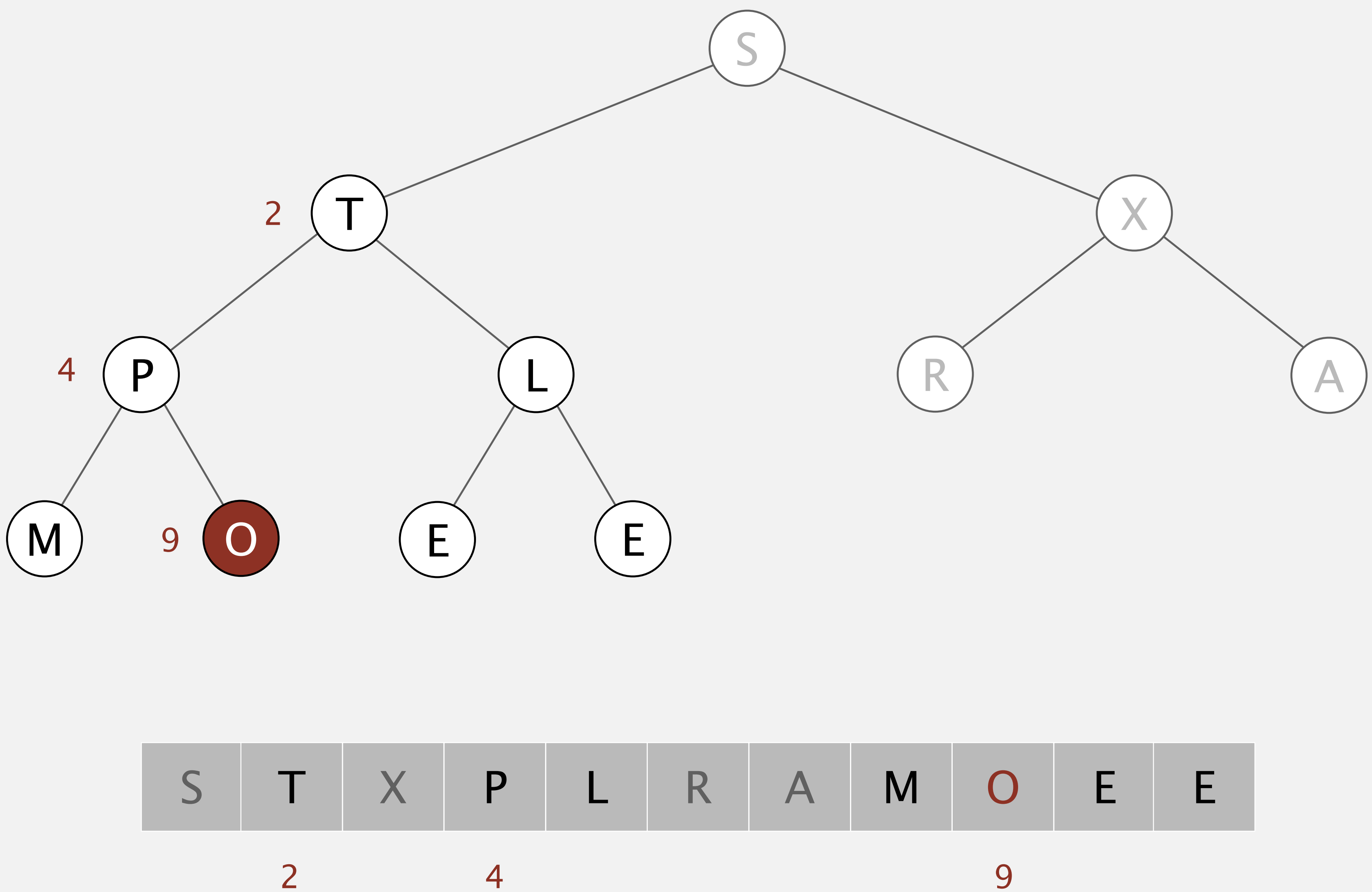
Heap construction. Build max heap using bottom-up method.



Heapsort demo

Heap construction. Build max heap using bottom-up method.

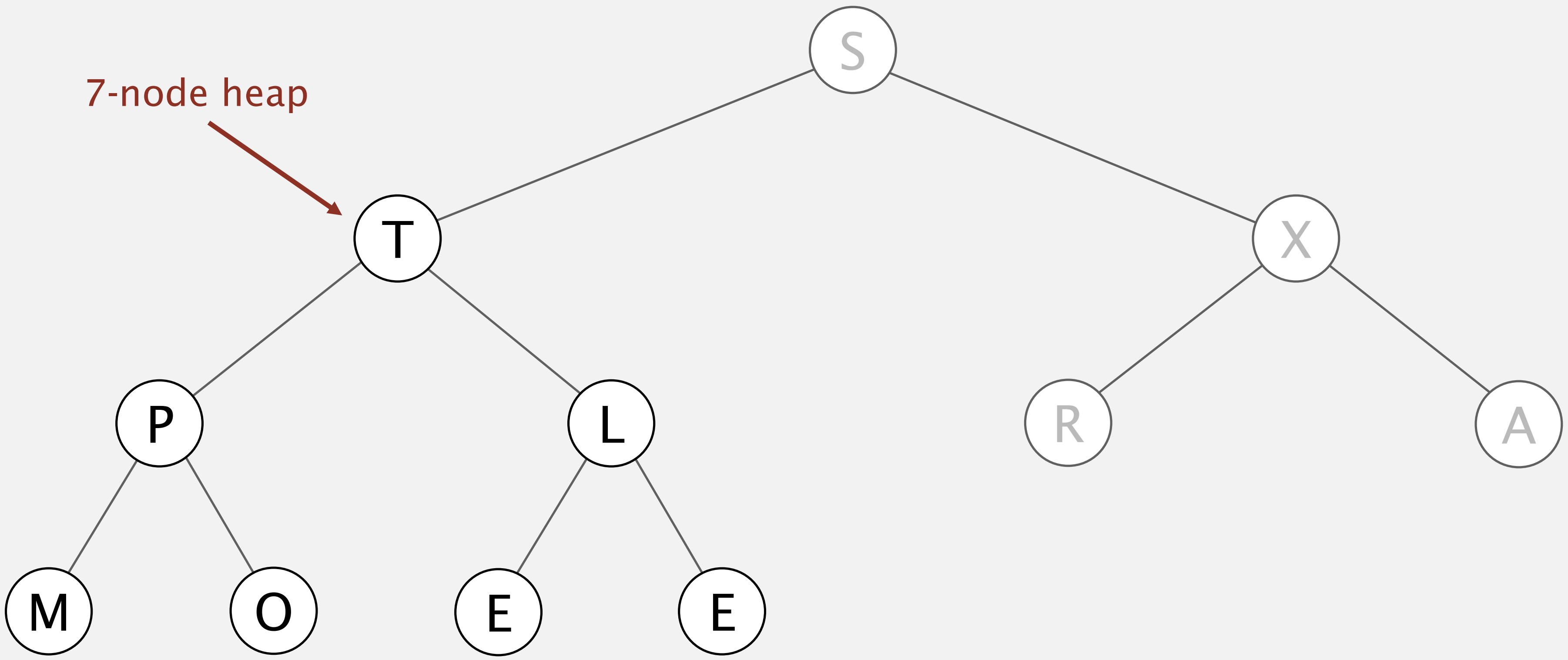
sink 2



Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 2

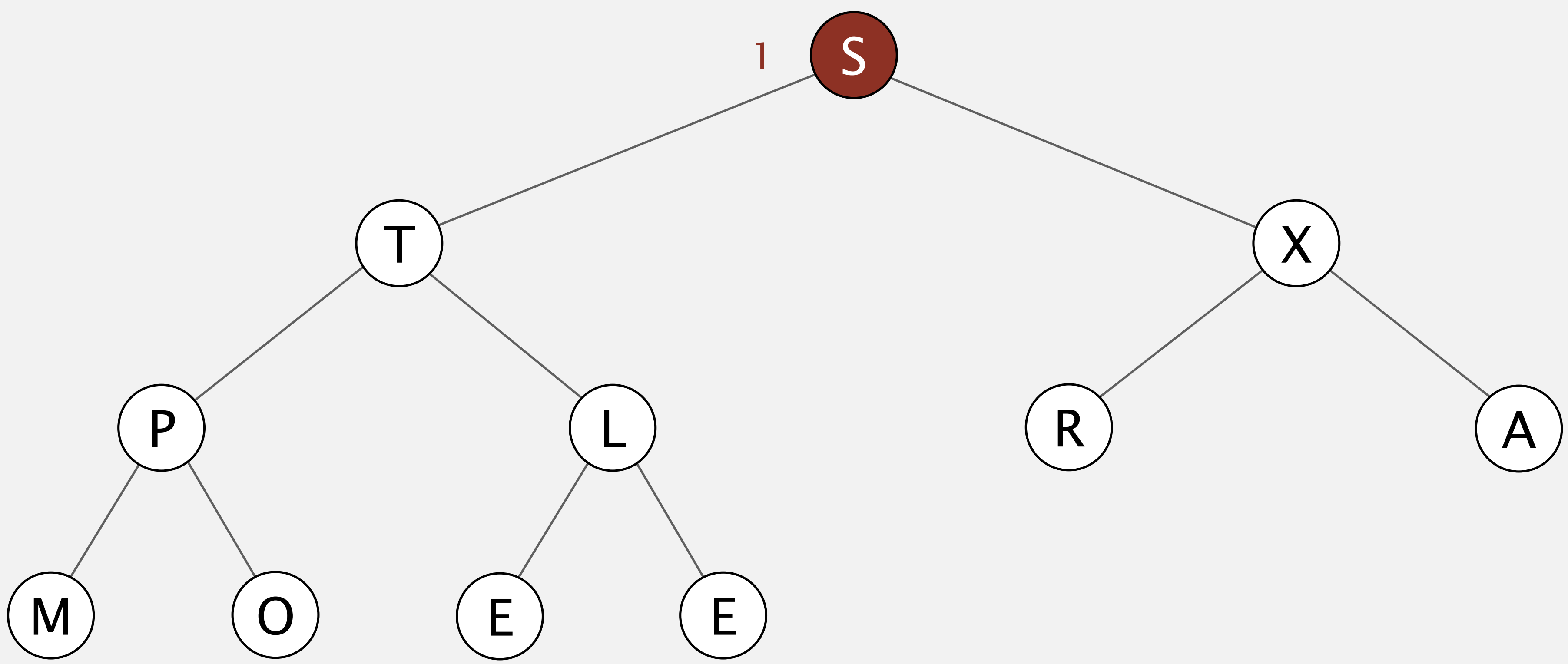


S	T	X	P	L	R	A	M	O	E	E

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 1



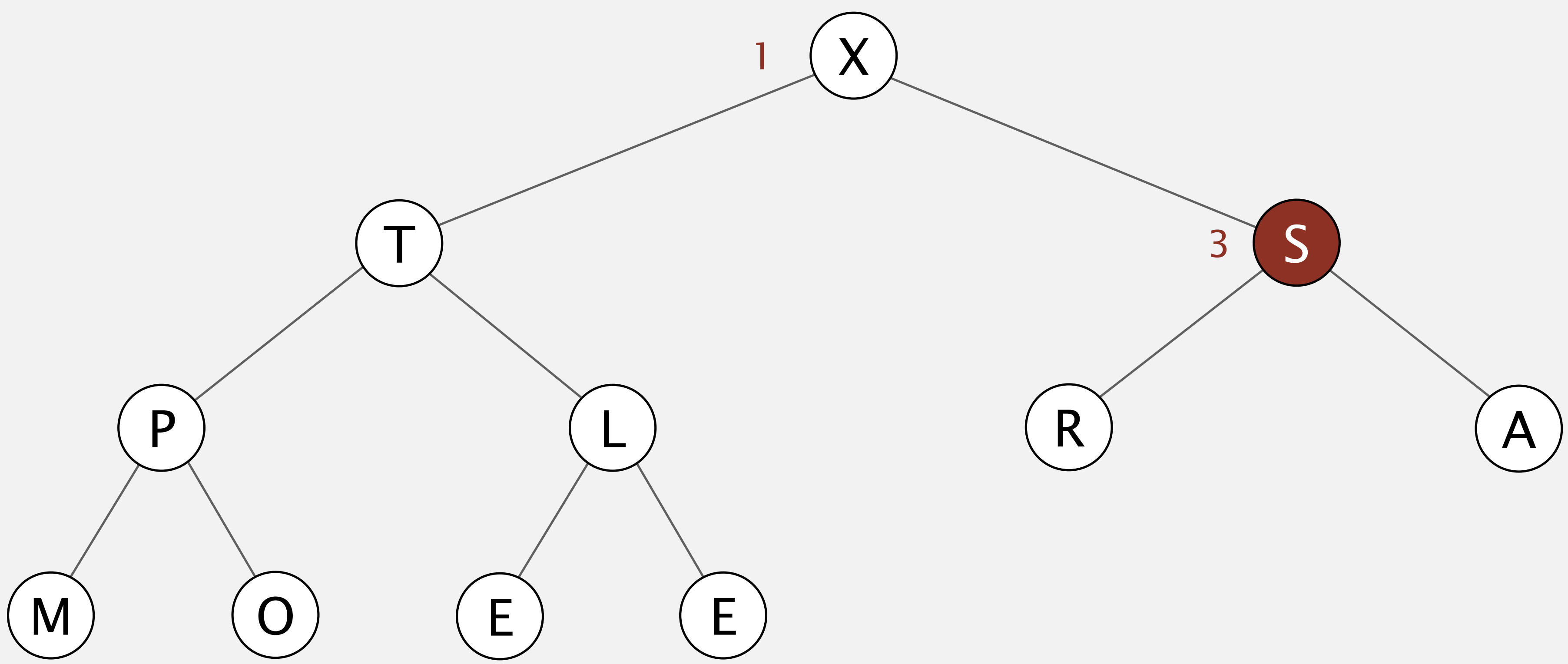
S	T	X	P	L	R	A	M	O	E	E

1

Heapsort demo

Heap construction. Build max heap using bottom-up method.

sink 1

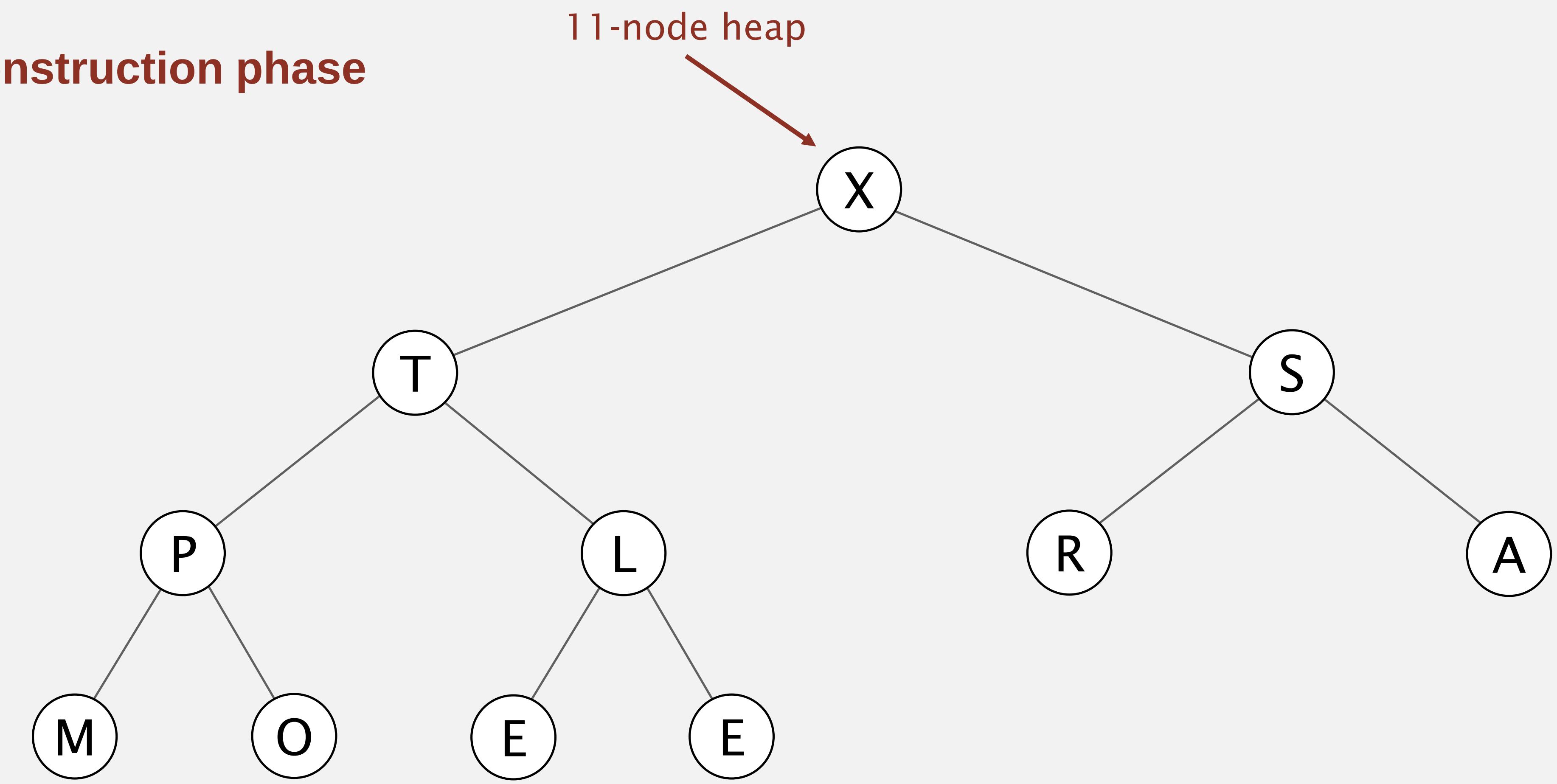


X	T	S	P	L	R	A	M	O	E	E
1		3								

Heapsort demo

Heap construction. Build max heap using bottom-up method.

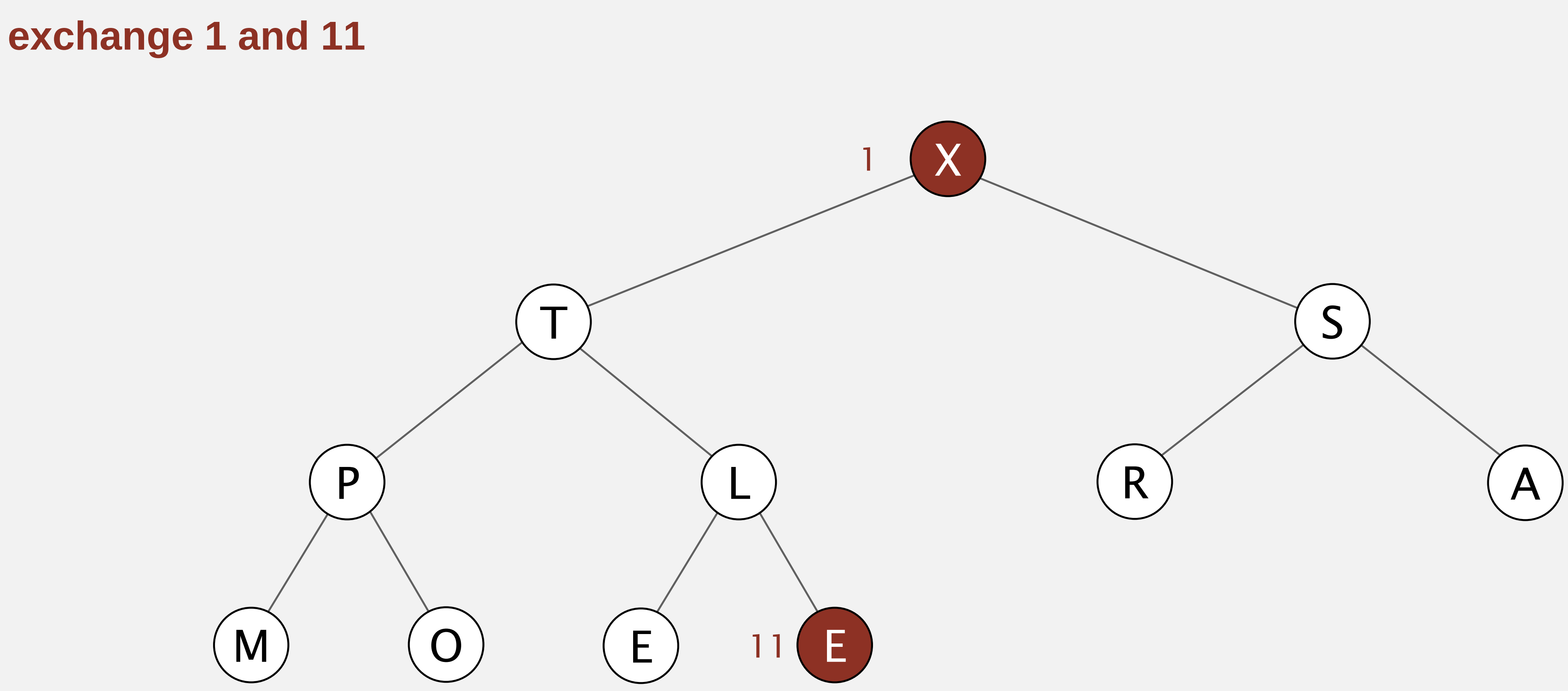
end of construction phase



X	T	S	P	L	R	A	M	O	E	E

Heapsort demo

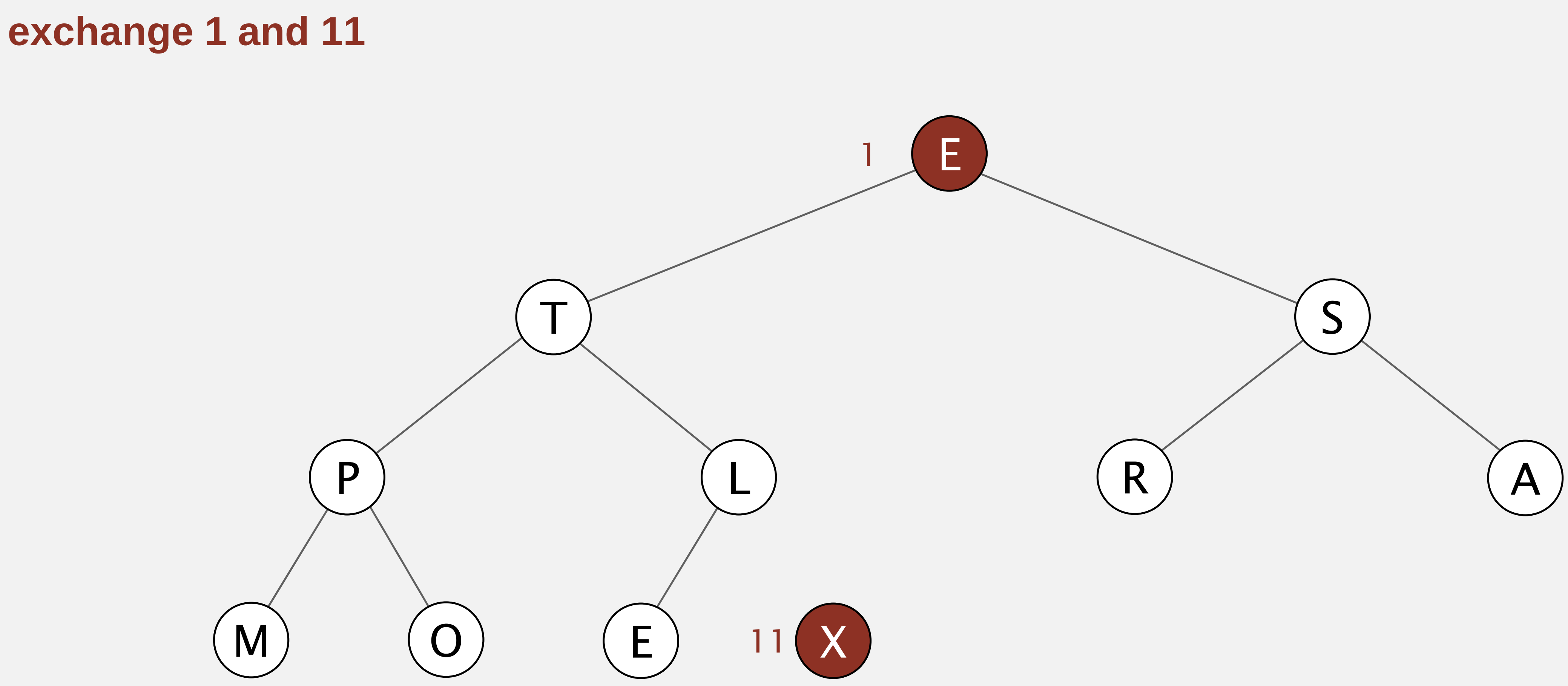
Sortdown. Repeatedly delete the largest remaining item.



X	T	S	P	L	R	A	M	O	E	E
1										11

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

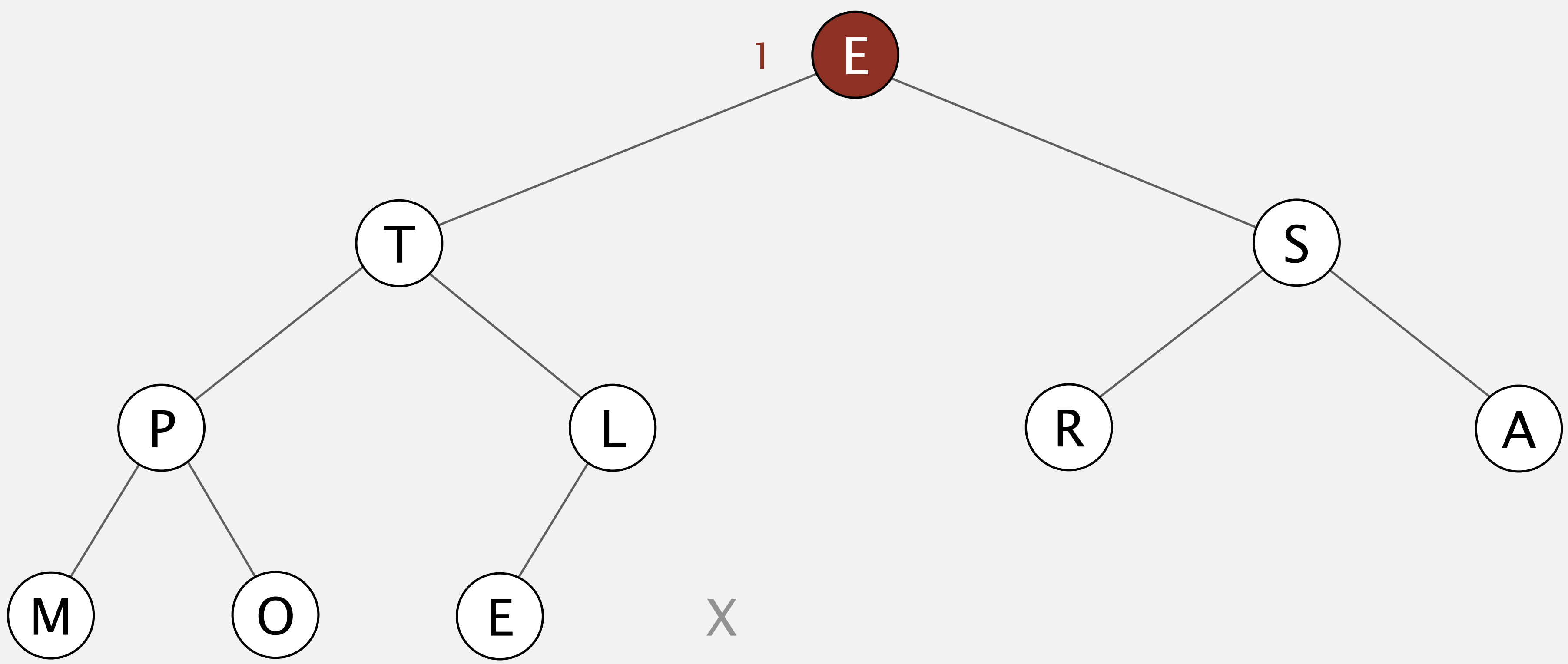


E	T	S	P	L	R	A	M	O	E	X
1										11

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

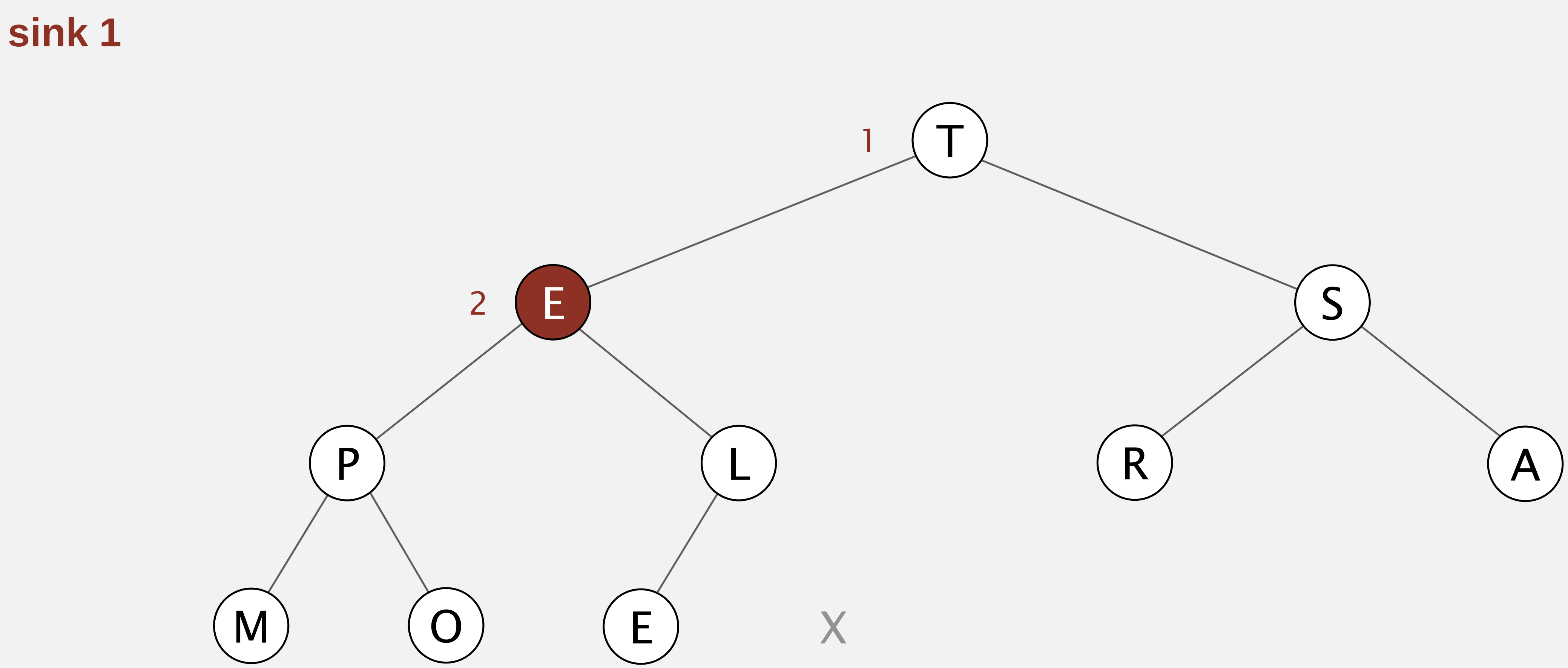
sink 1



E	T	S	P	L	R	A	M	O	E	X
1										

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

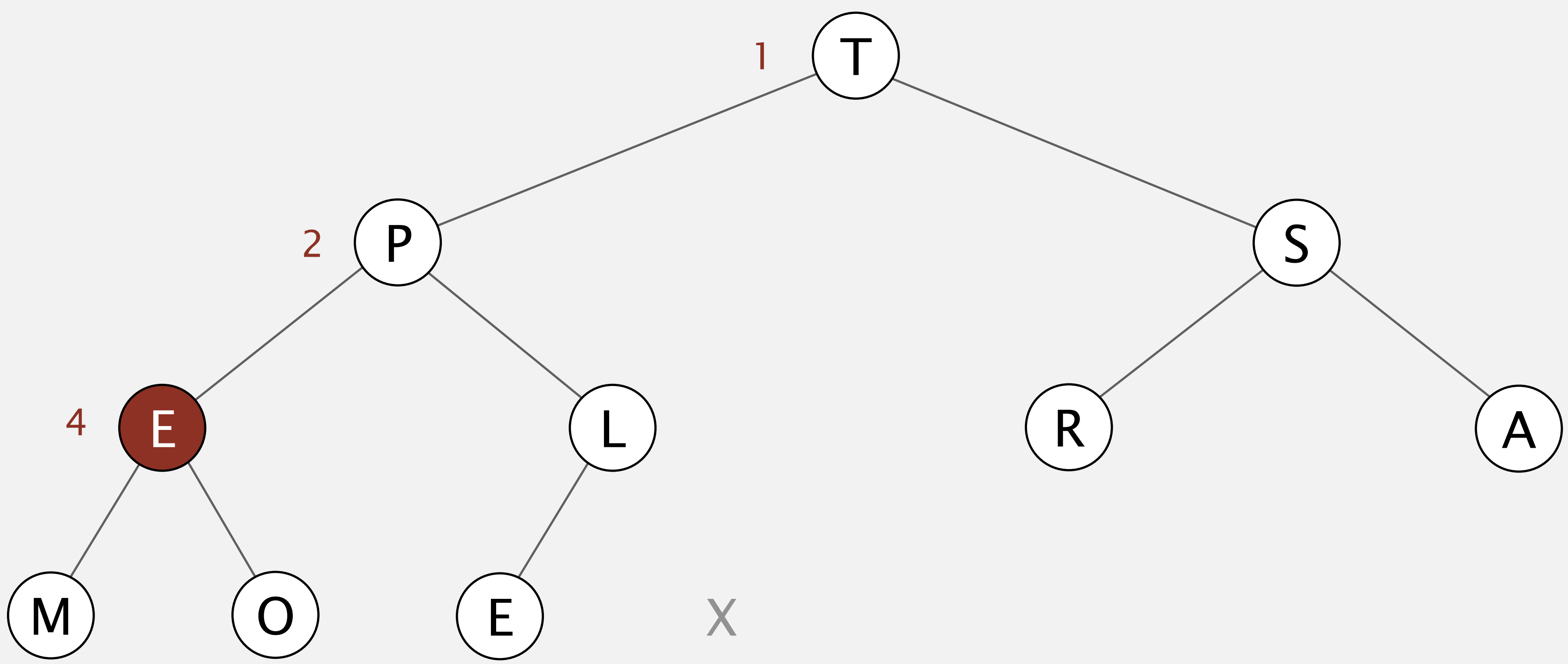


T	E	S	P	L	R	A	M	O	E	X
1	2									

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

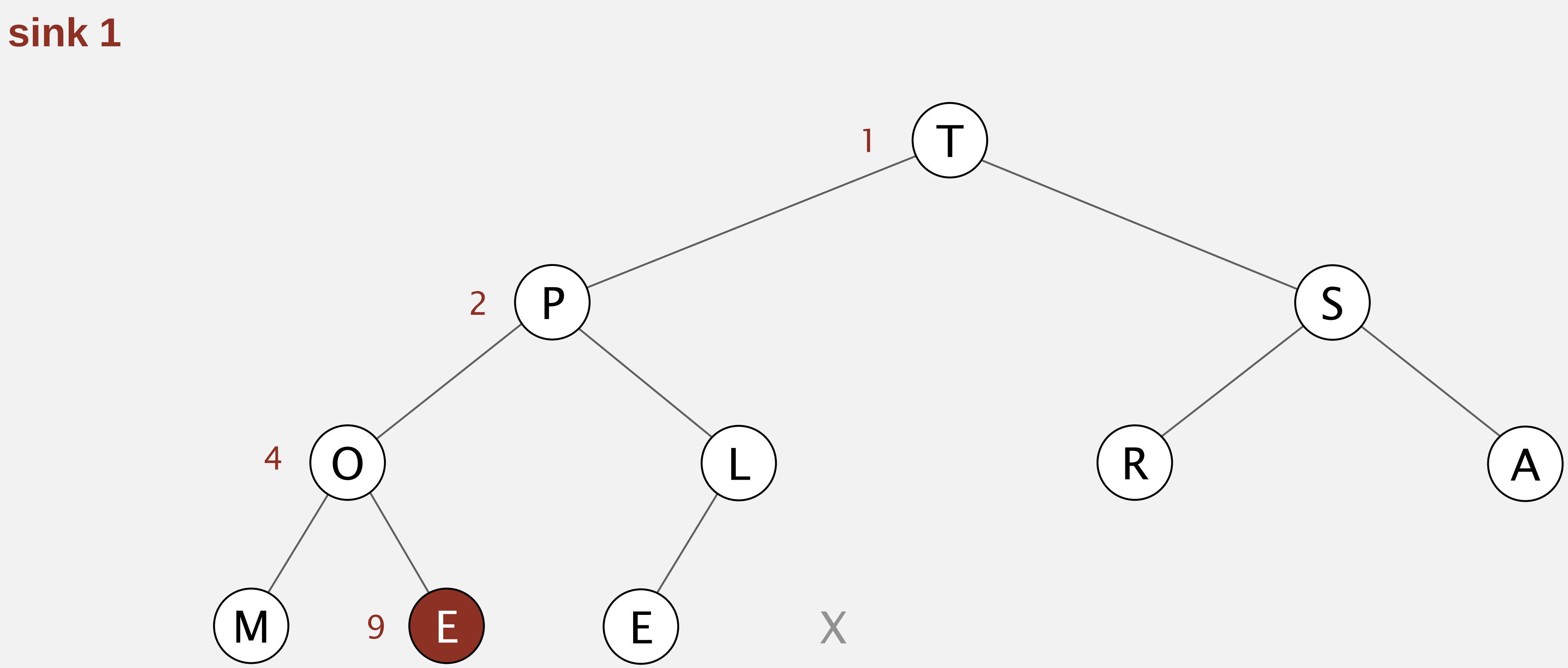
sink 1



T	P	S	E	L	R	A	M	O	E	X
1	2		4							

Heapsort demo

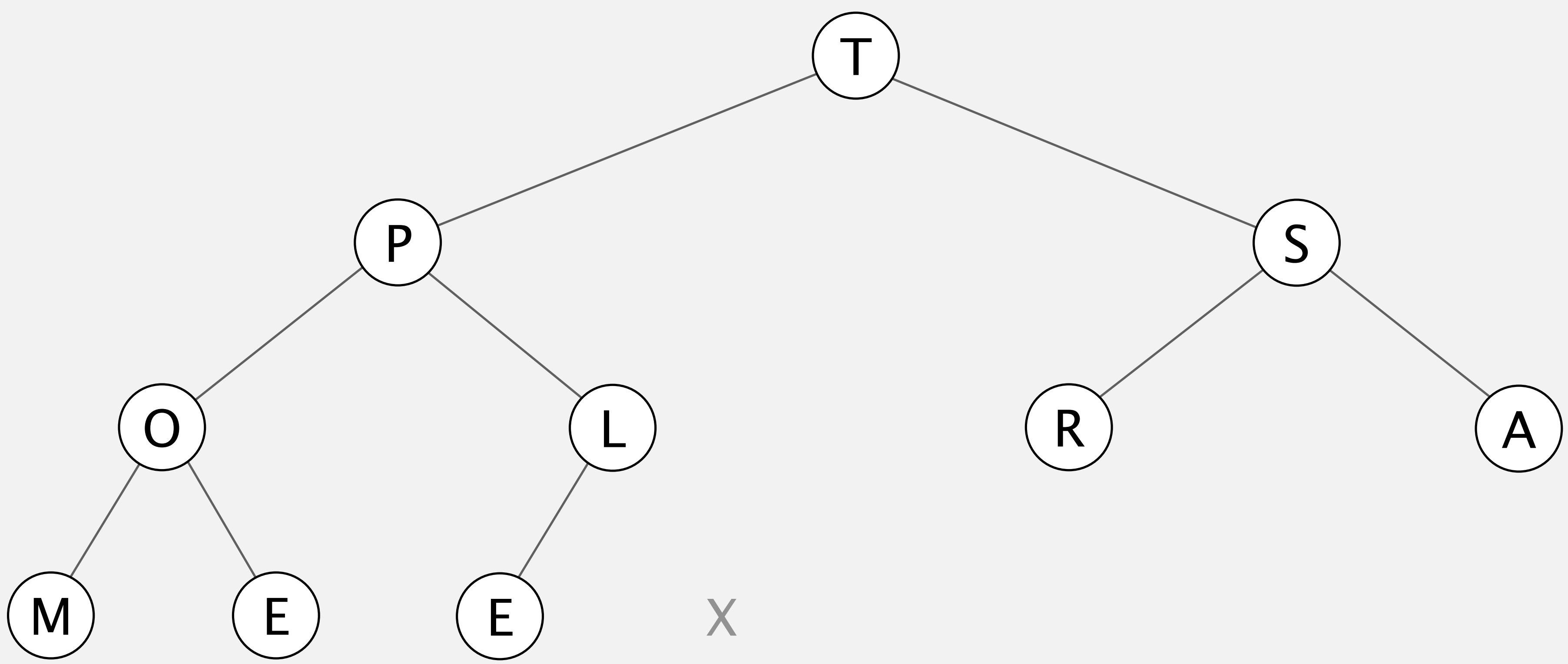
Sortdown. Repeatedly delete the largest remaining item.



T	P	S	O	L	R	A	M	E	E	X
1	2		4					9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

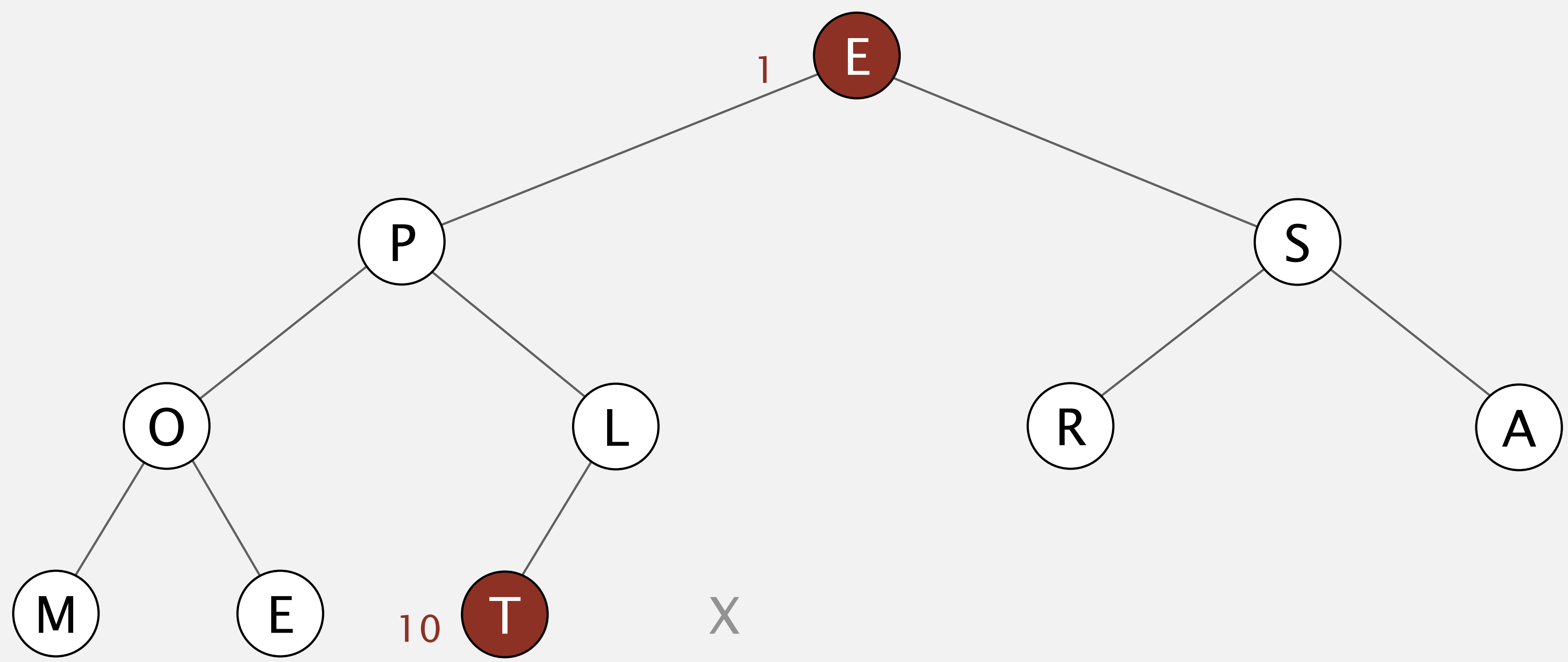


T	P	S	O	L	R	A	M	E	E	X

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

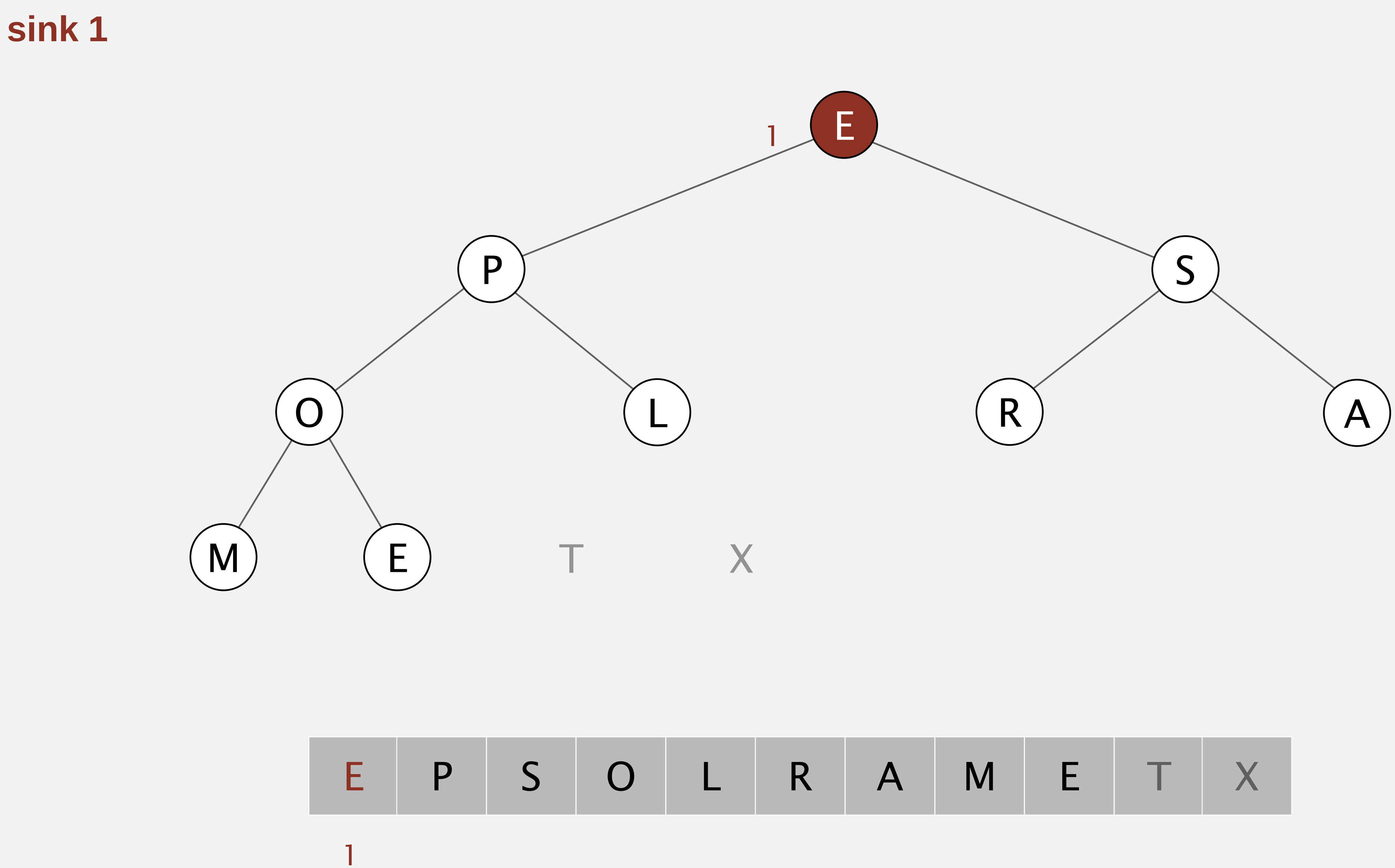
exchange 1 and 10



E	P	S	O	L	R	A	M	E	T	X
1									10	

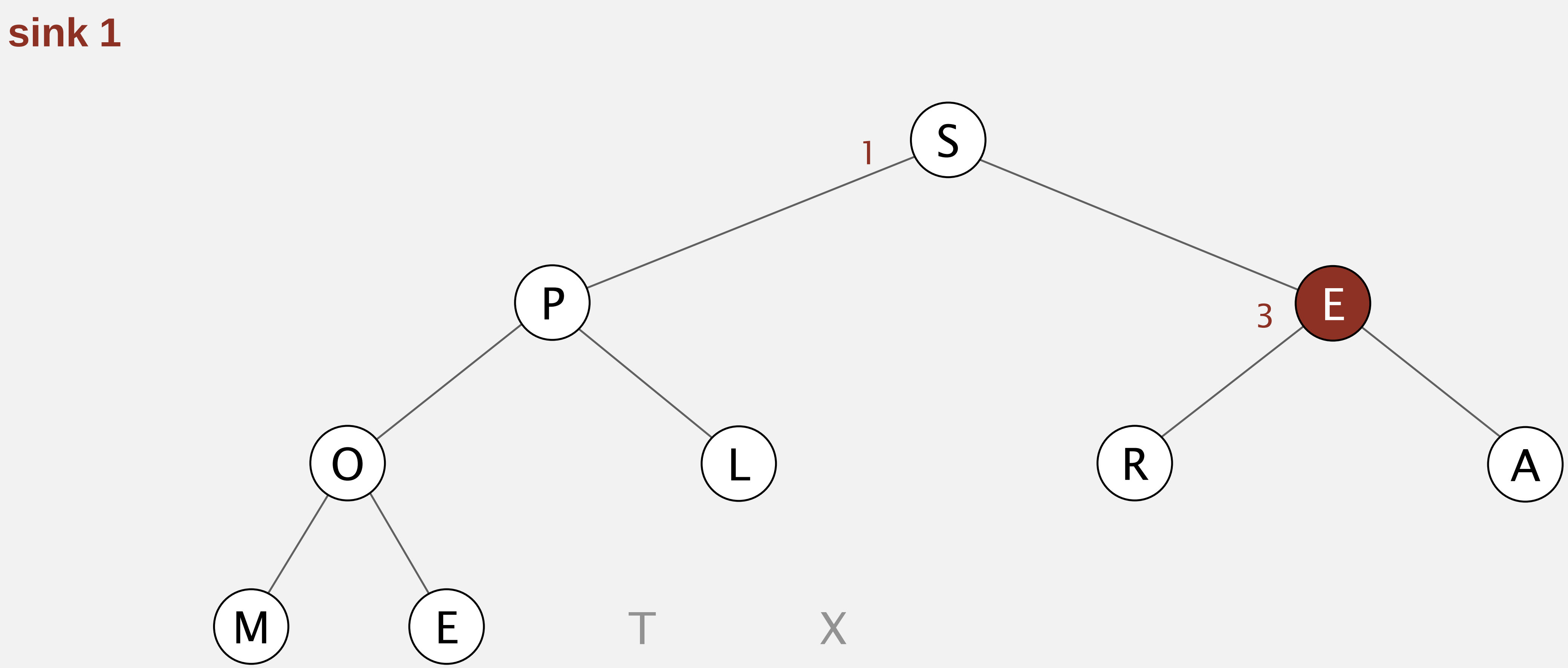
Heapsort demo

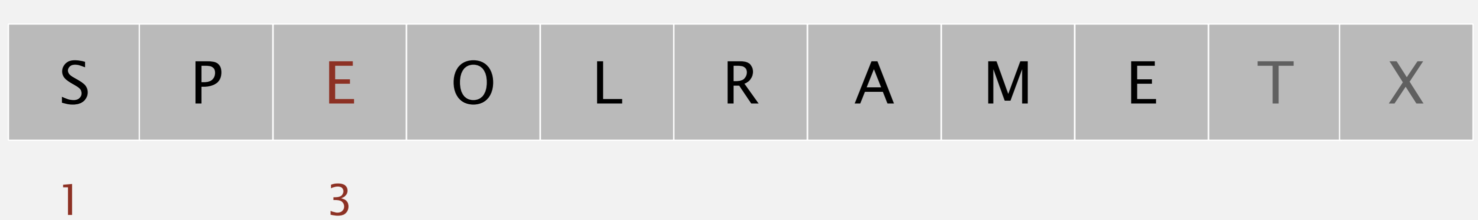
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

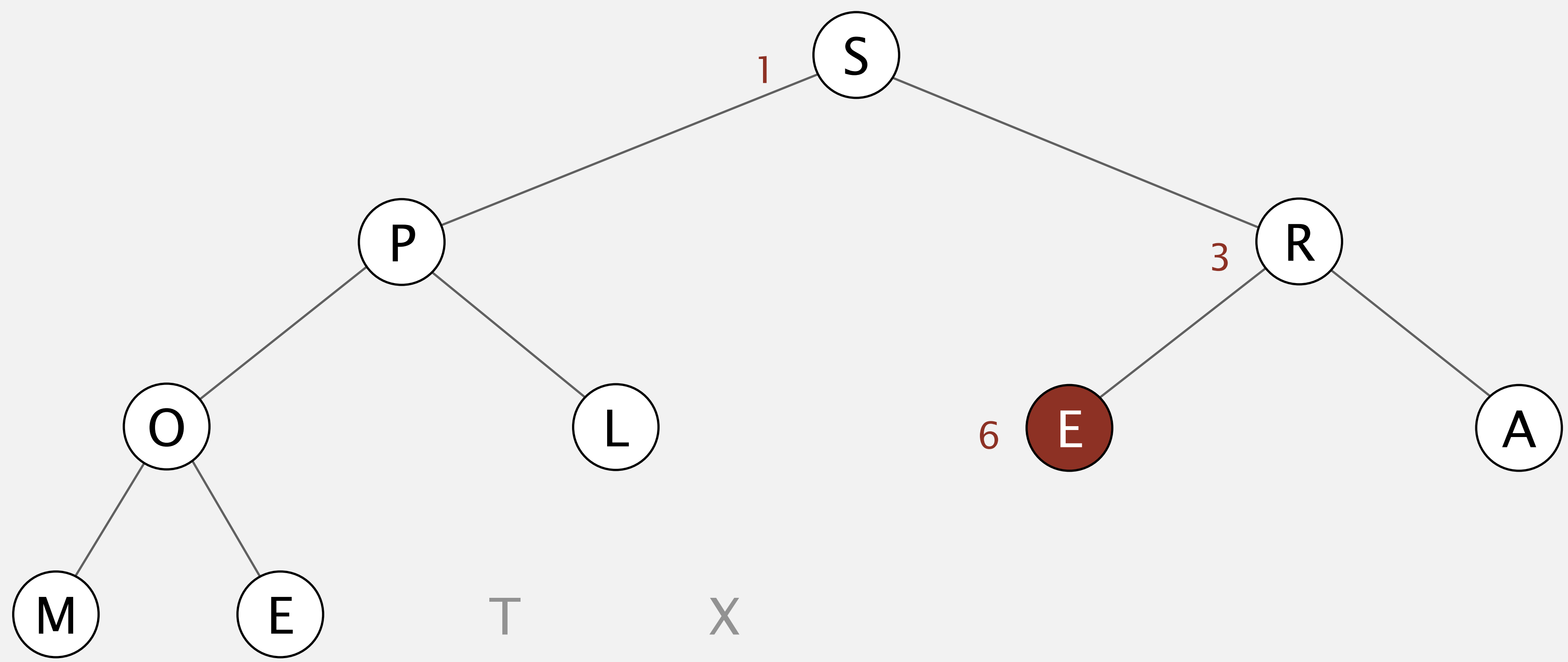




Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

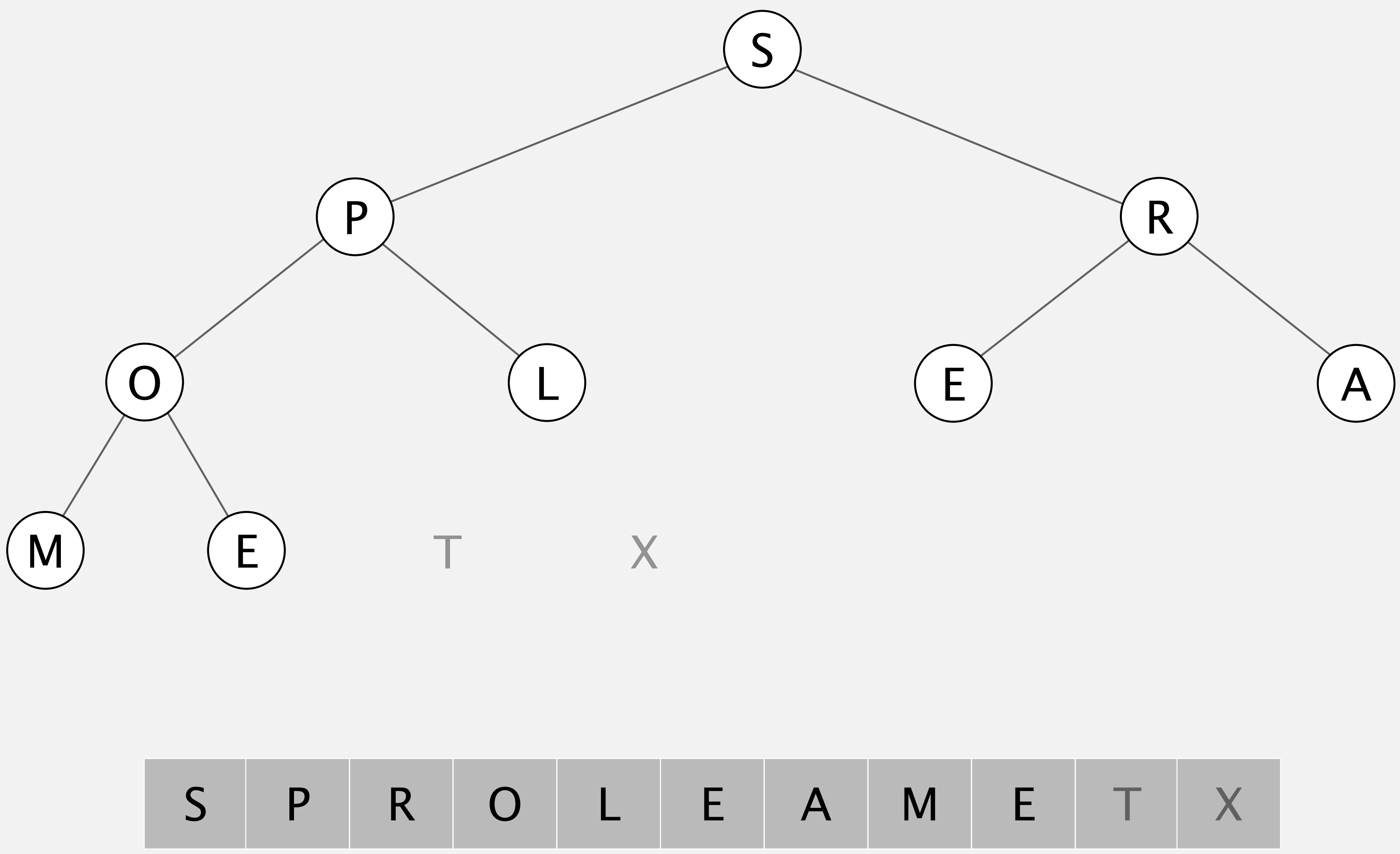
sink 1



S	P	R	O	L	E	A	M	E	T	X
1		3			6					

Heapsort demo

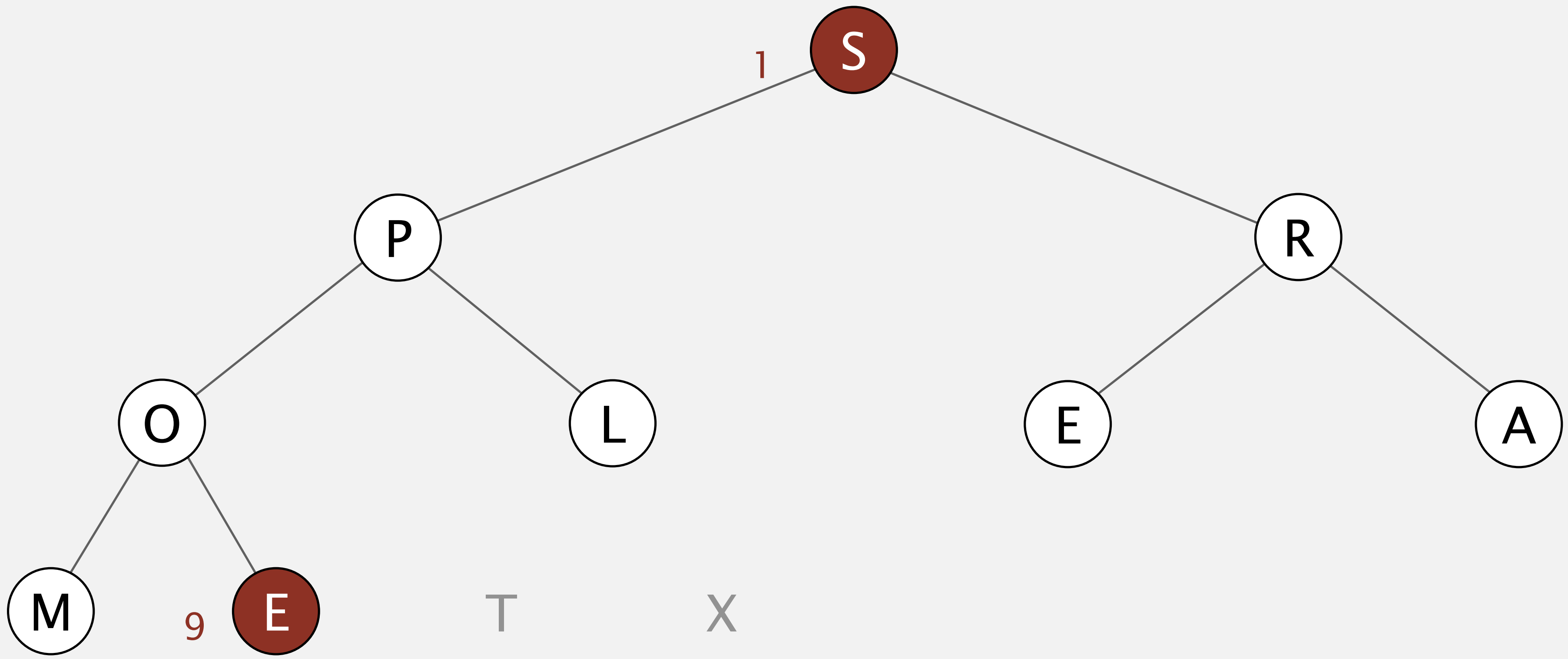
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

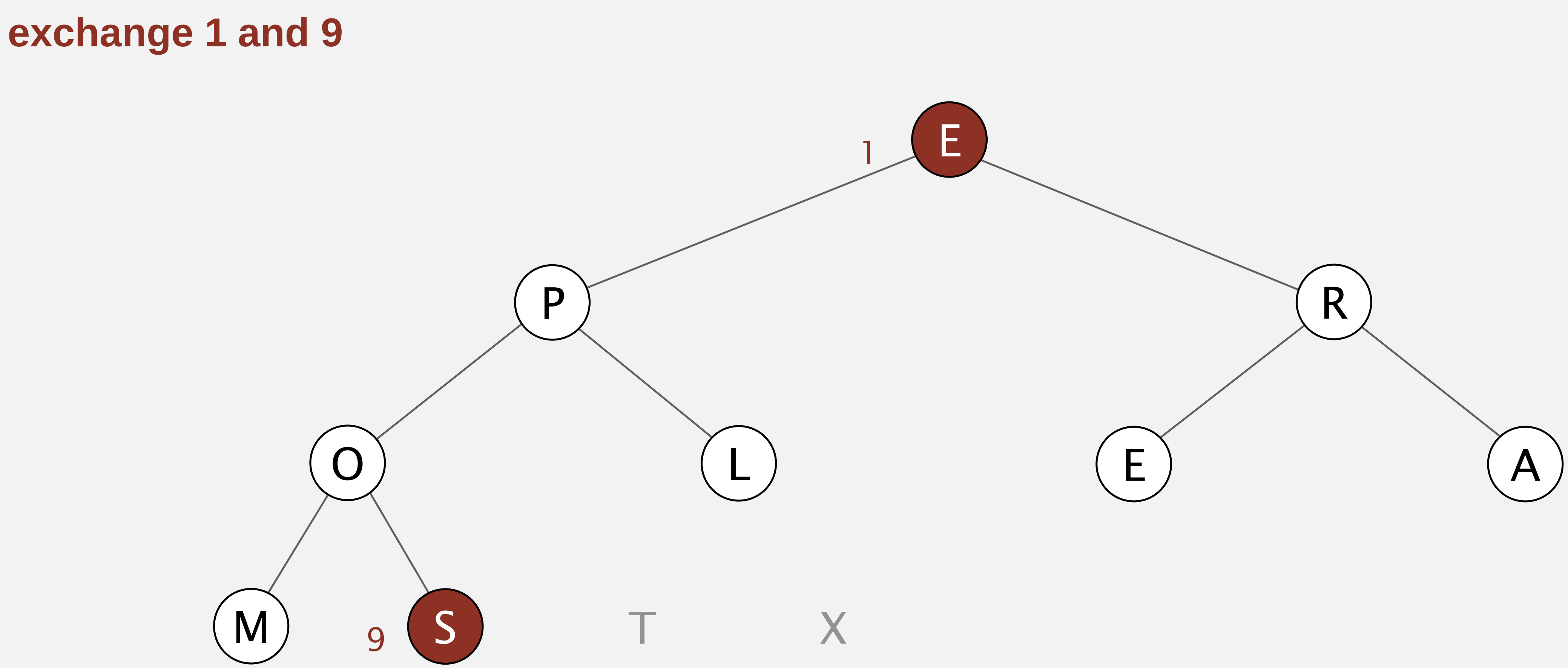
exchange 1 and 9



S	P	R	O	L	E	A	M	E	T	X
1								9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

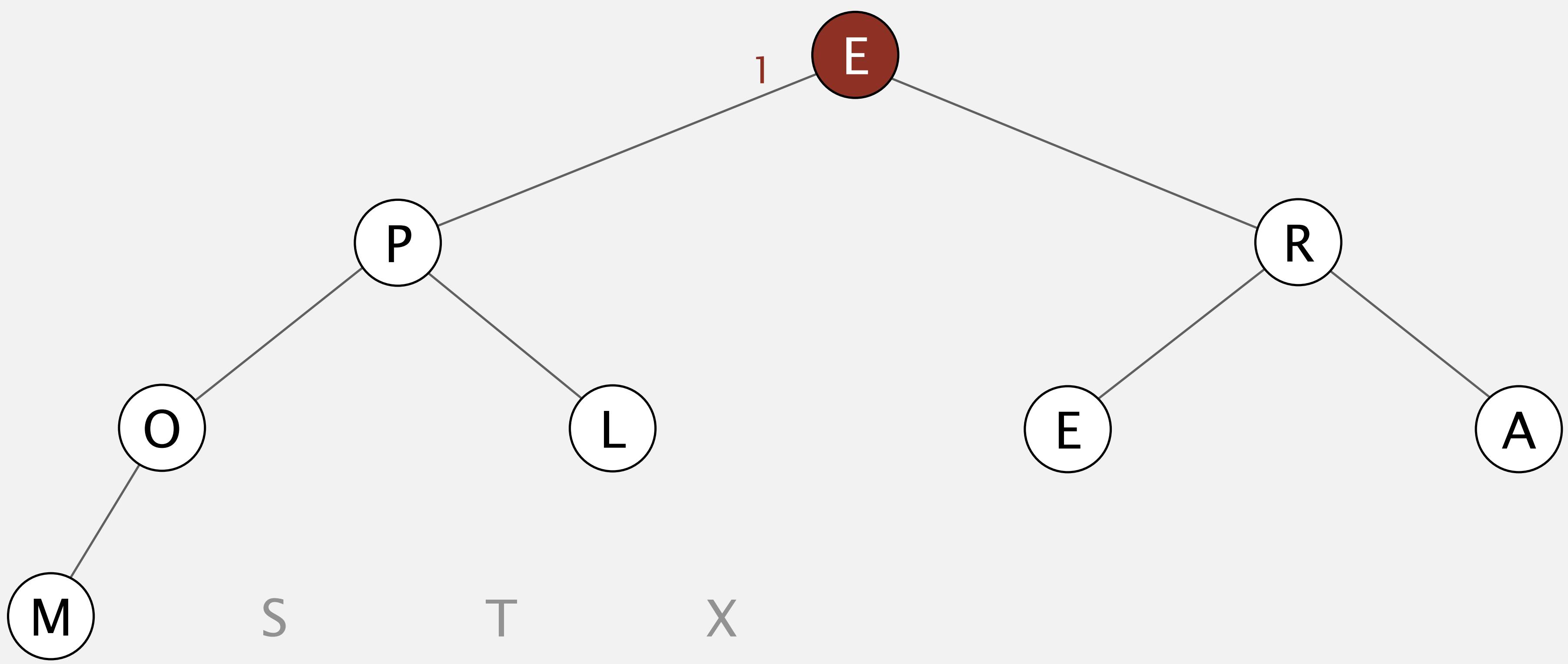


E	P	R	O	L	E	A	M	S	T	X
1								9		

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

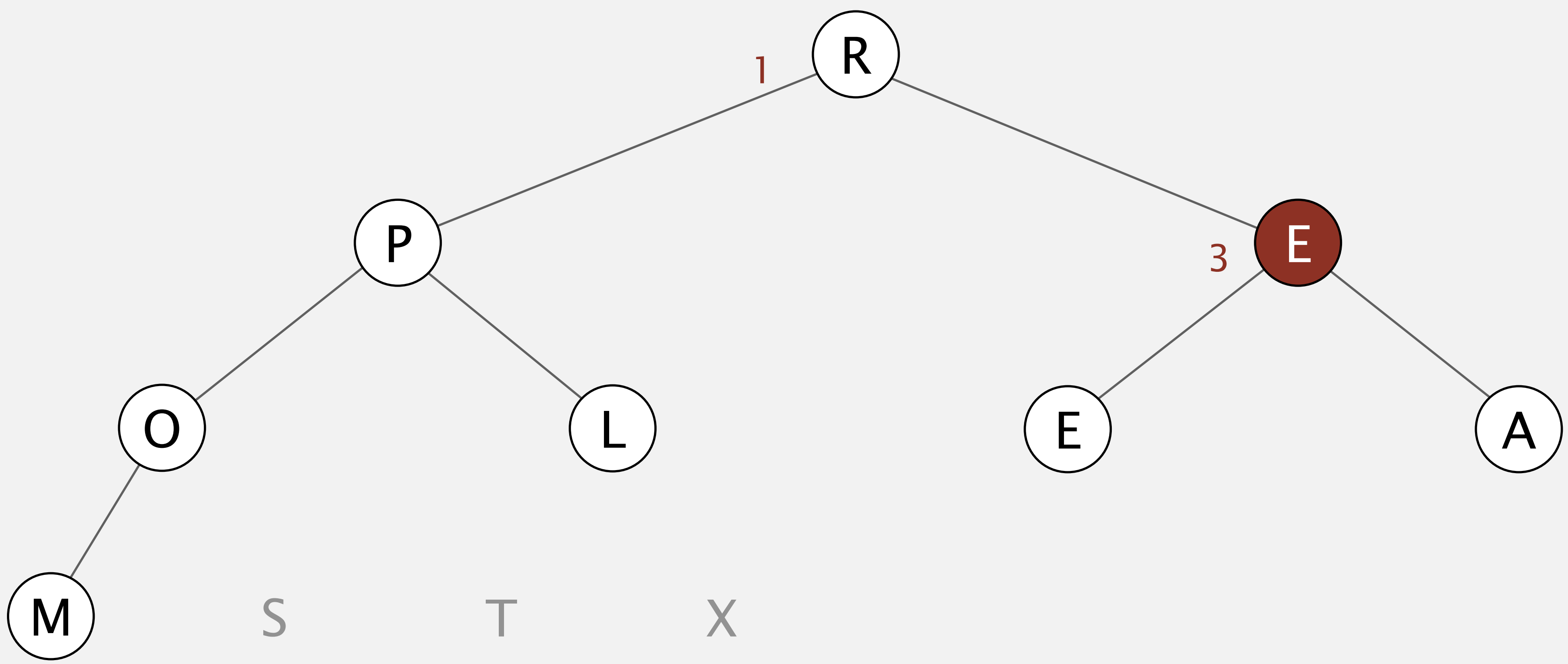


E	P	R	O	L	E	A	M	S	T	X
1										

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

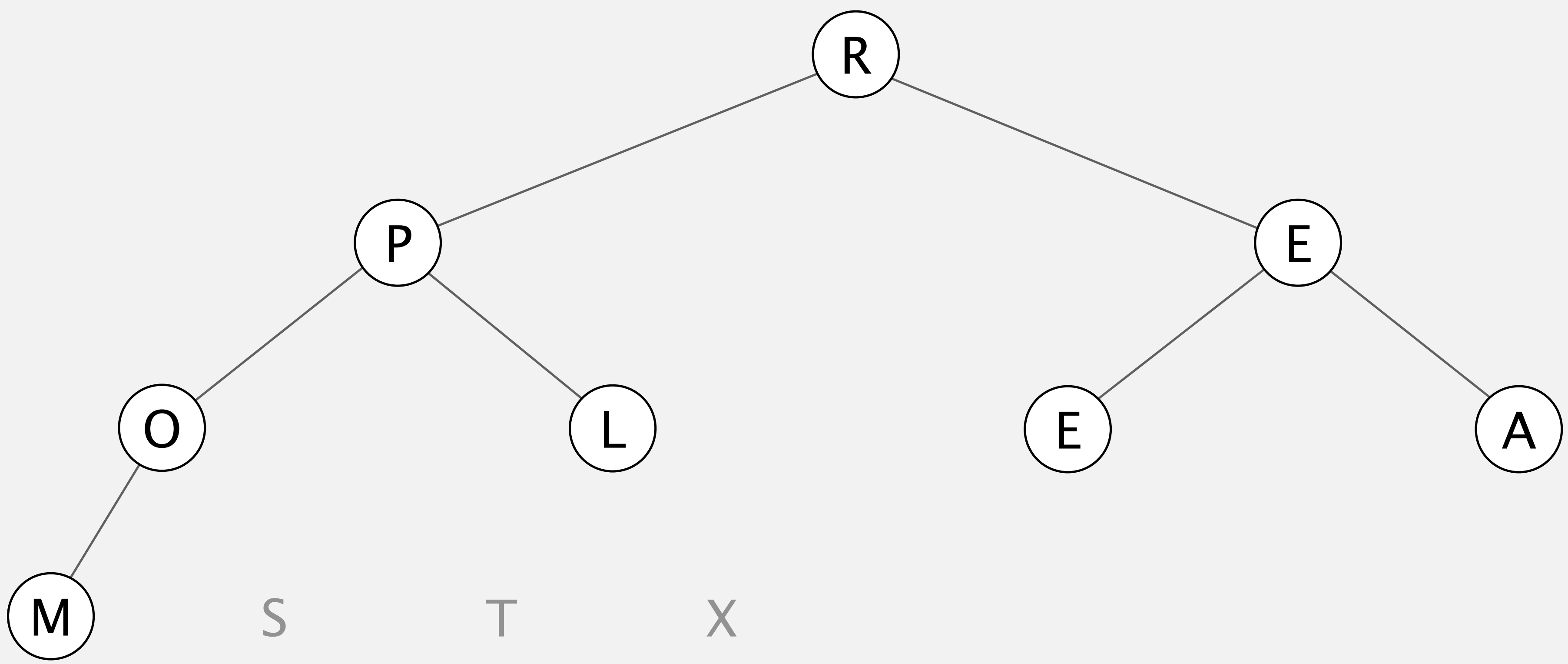
sink 1



R	P	E	O	L	E	A	M	S	T	X
1		3								

Heapsort demo

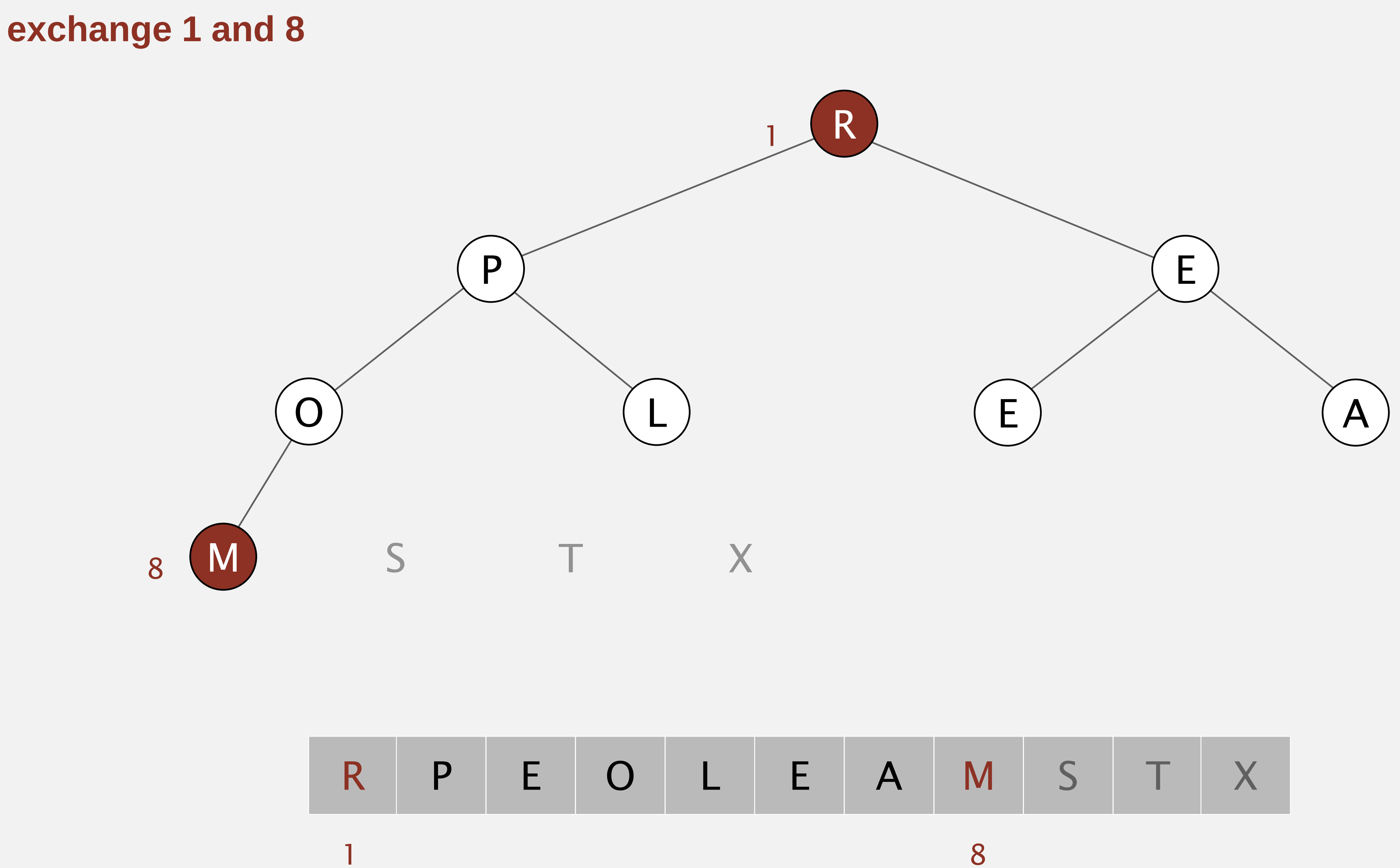
Sortdown. Repeatedly delete the largest remaining item.



R	P	E	O	L	E	A	M	S	T	X

Heapsort demo

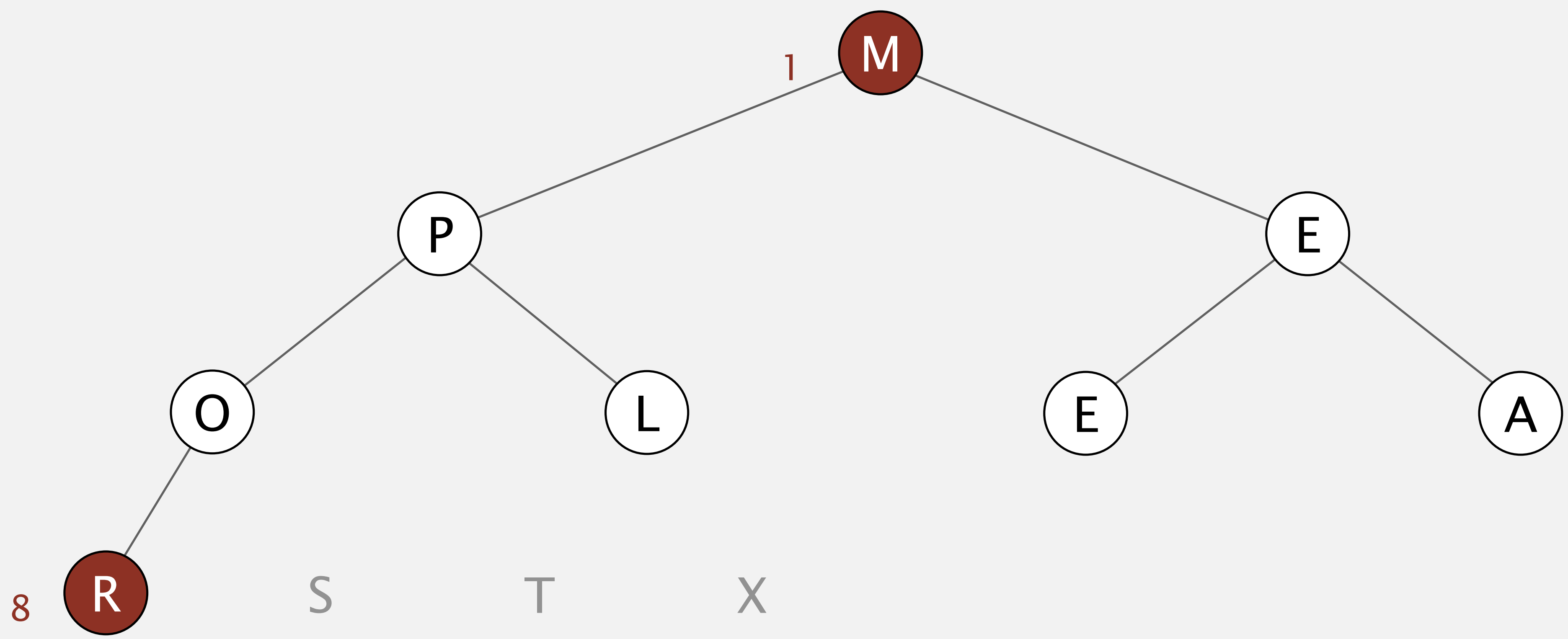
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

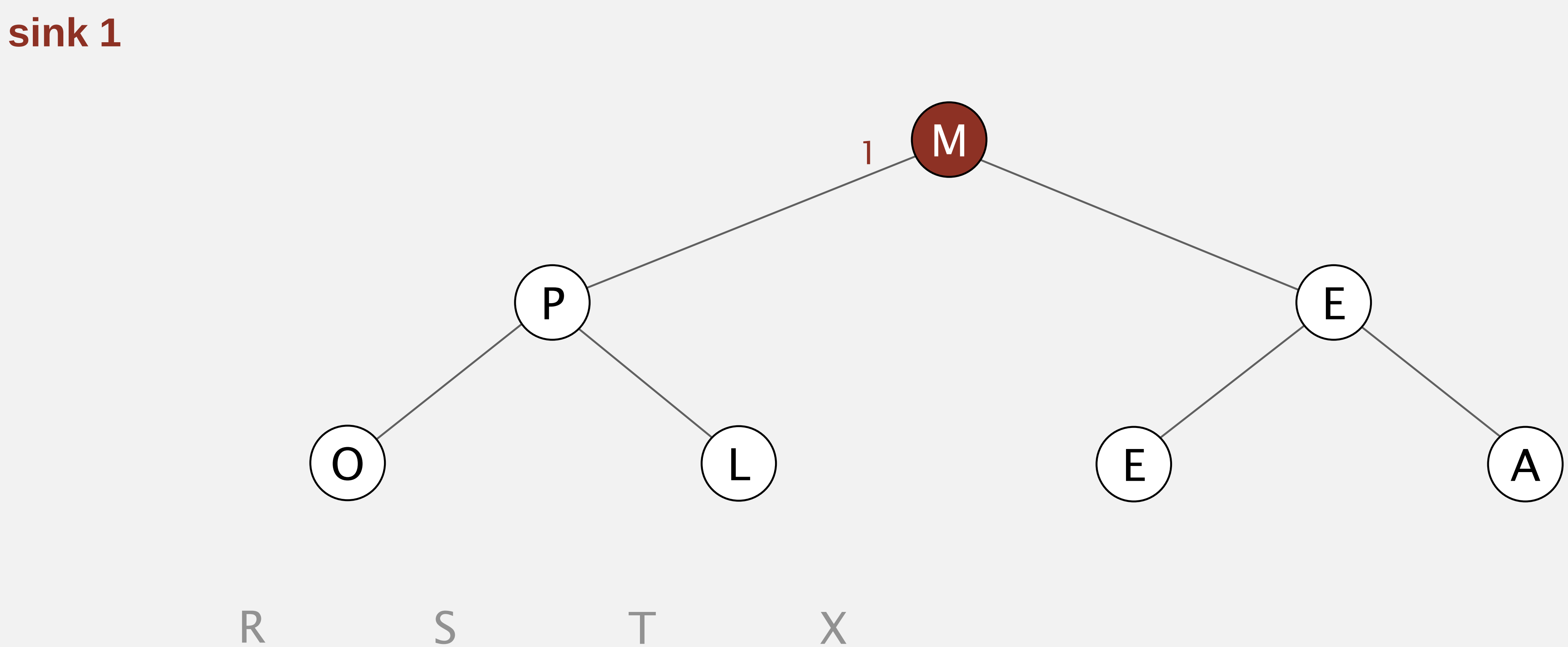
exchange 1 and 8



M	P	E	O	L	E	A	R	S	T	X
1							8			

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



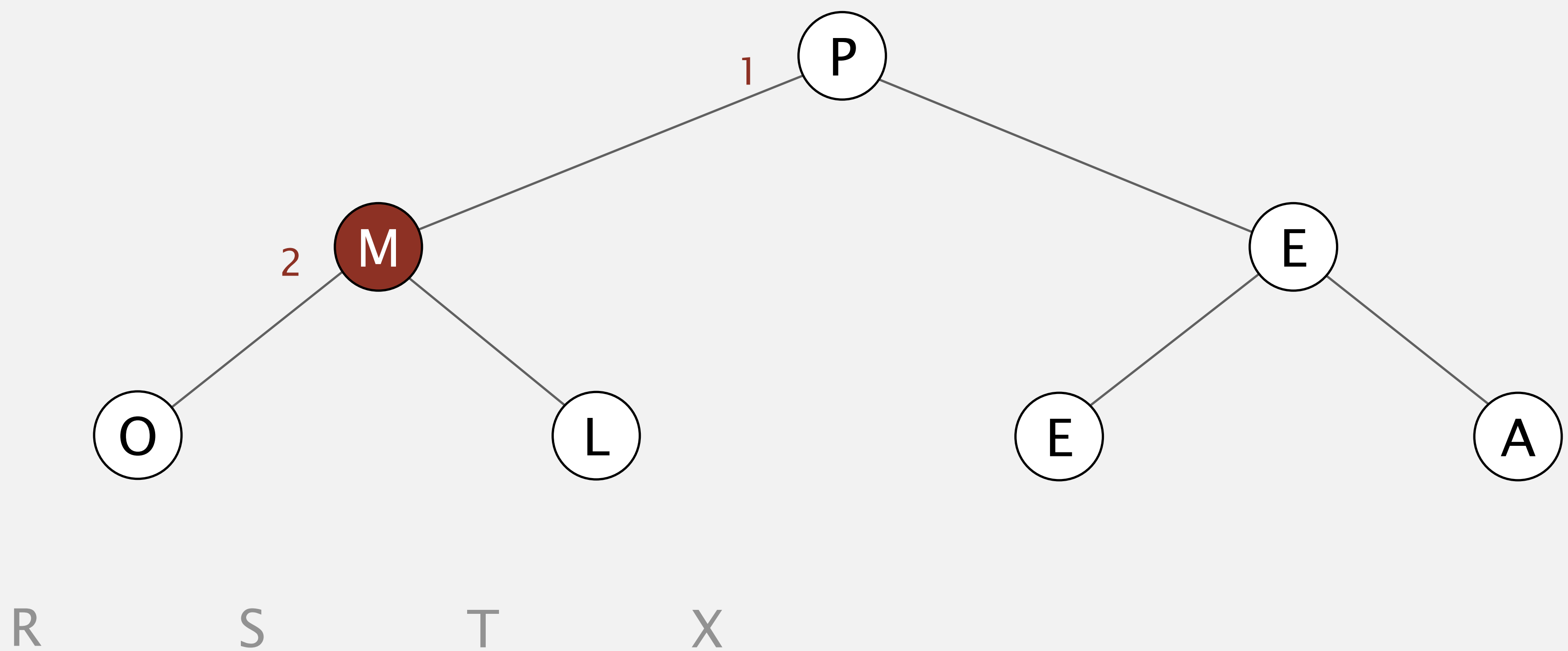
M	P	E	O	L	E	A	R	S	T	X

1

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

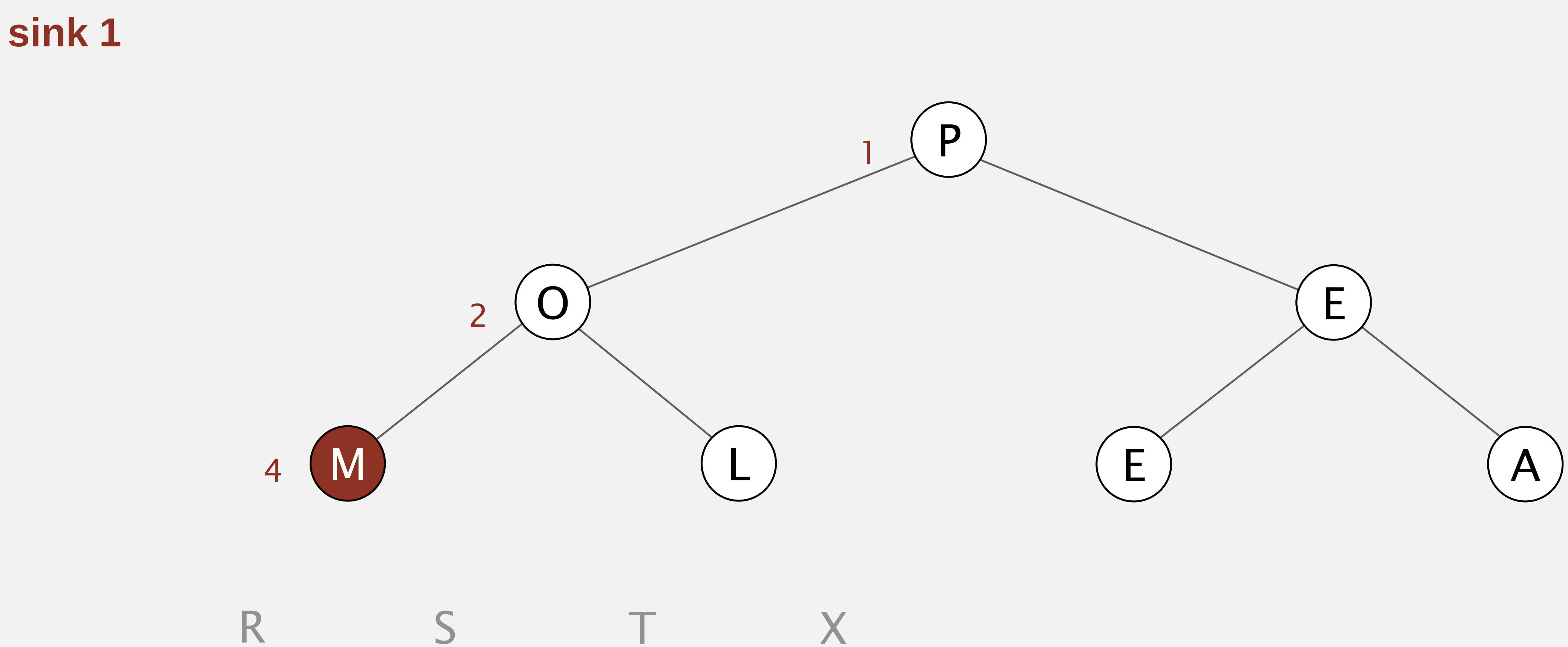
sink 1



P	M	E	O	L	E	A	R	S	T	X
1	2									

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

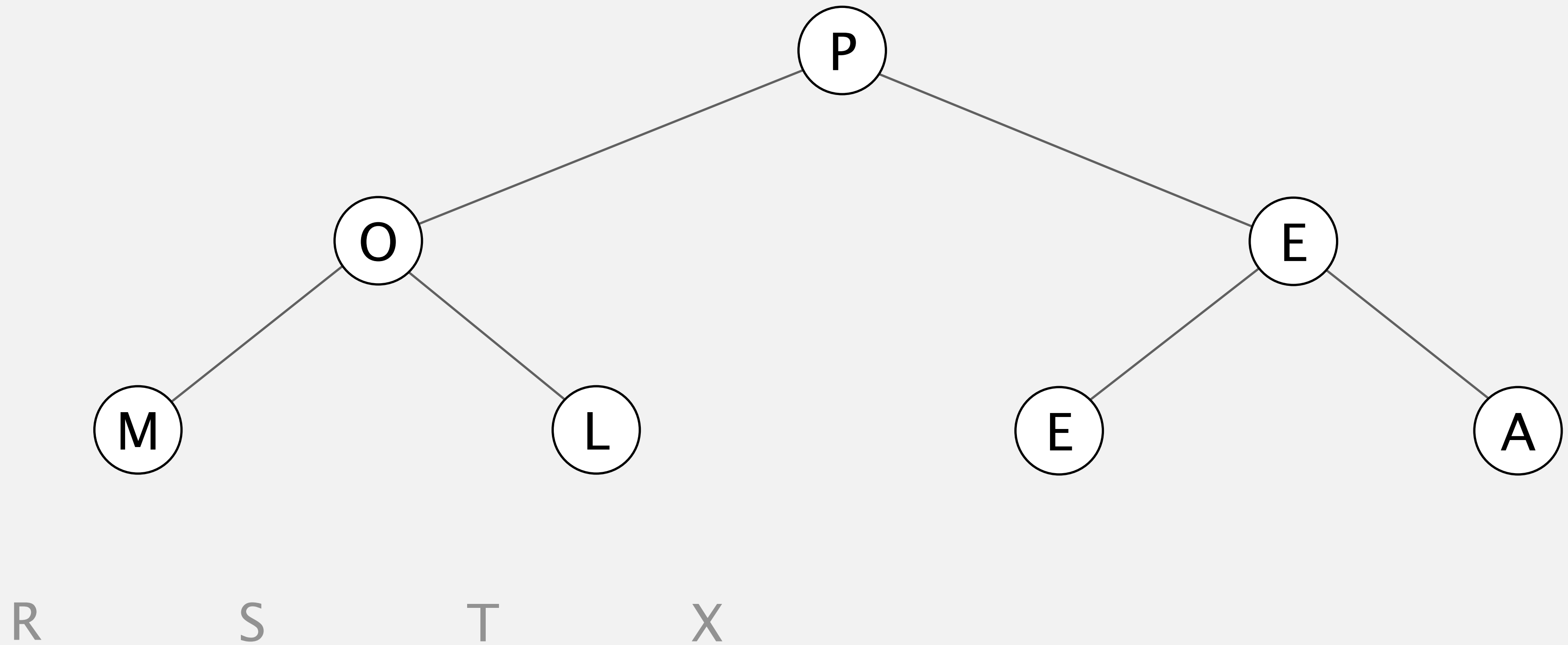


P	O	E	M	L	E	A	R	S	T	X

1 2 4

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

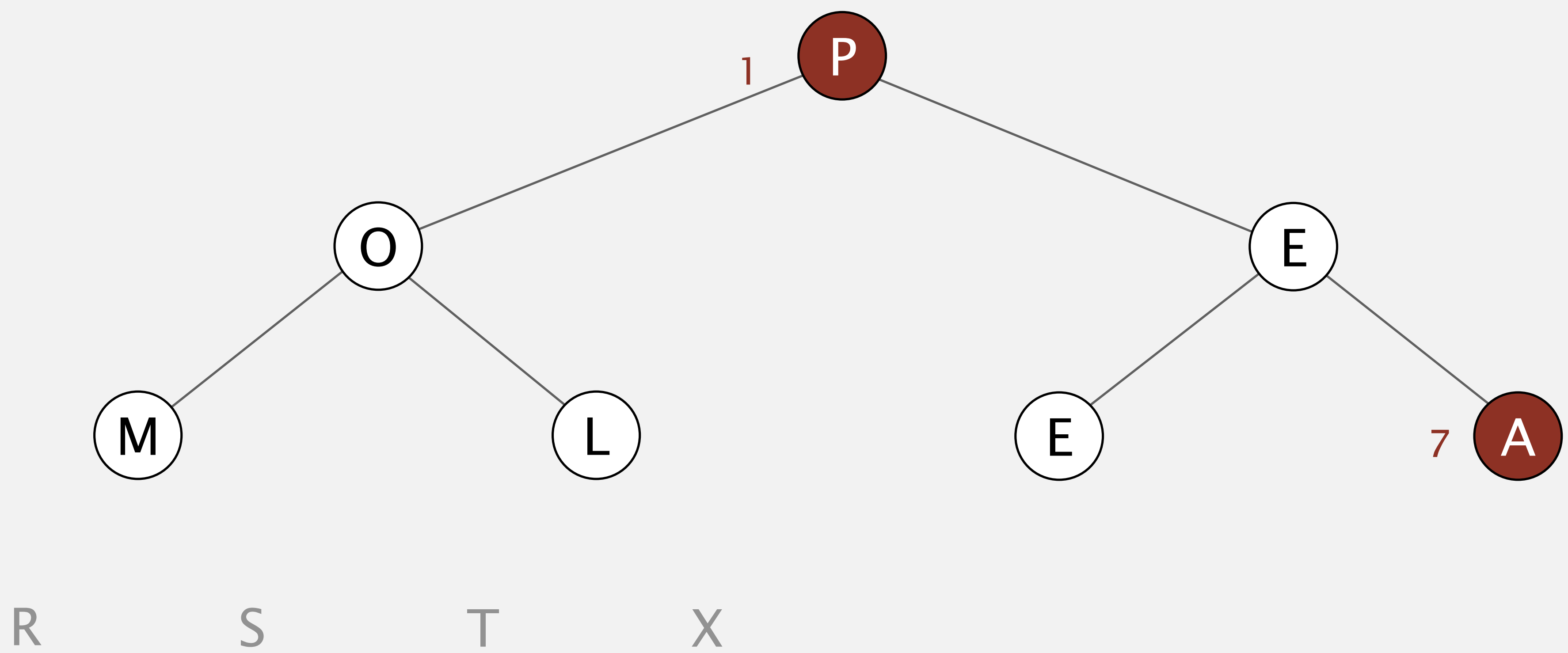


P	O	E	M	L	E	A	R	S	T	X

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

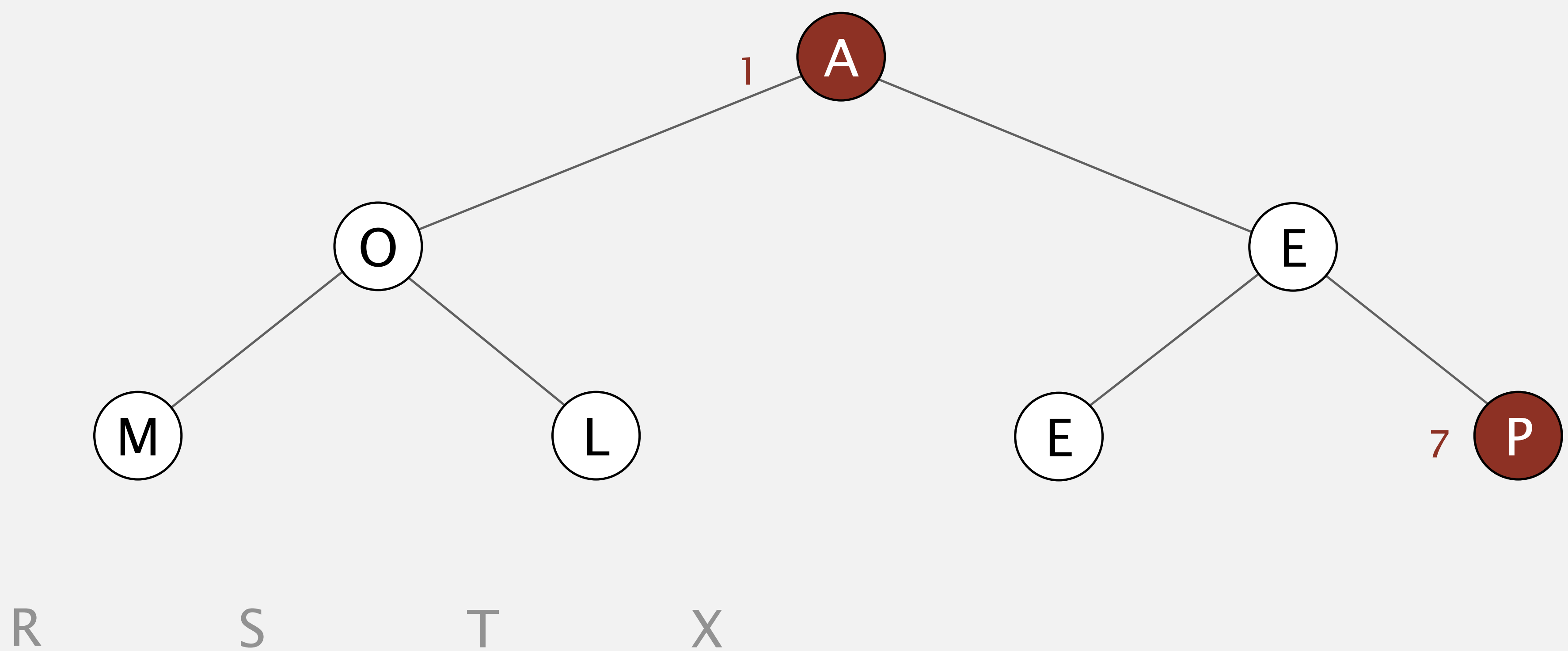


P	O	E	M	L	E	A	R	S	T	X
1						7				

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

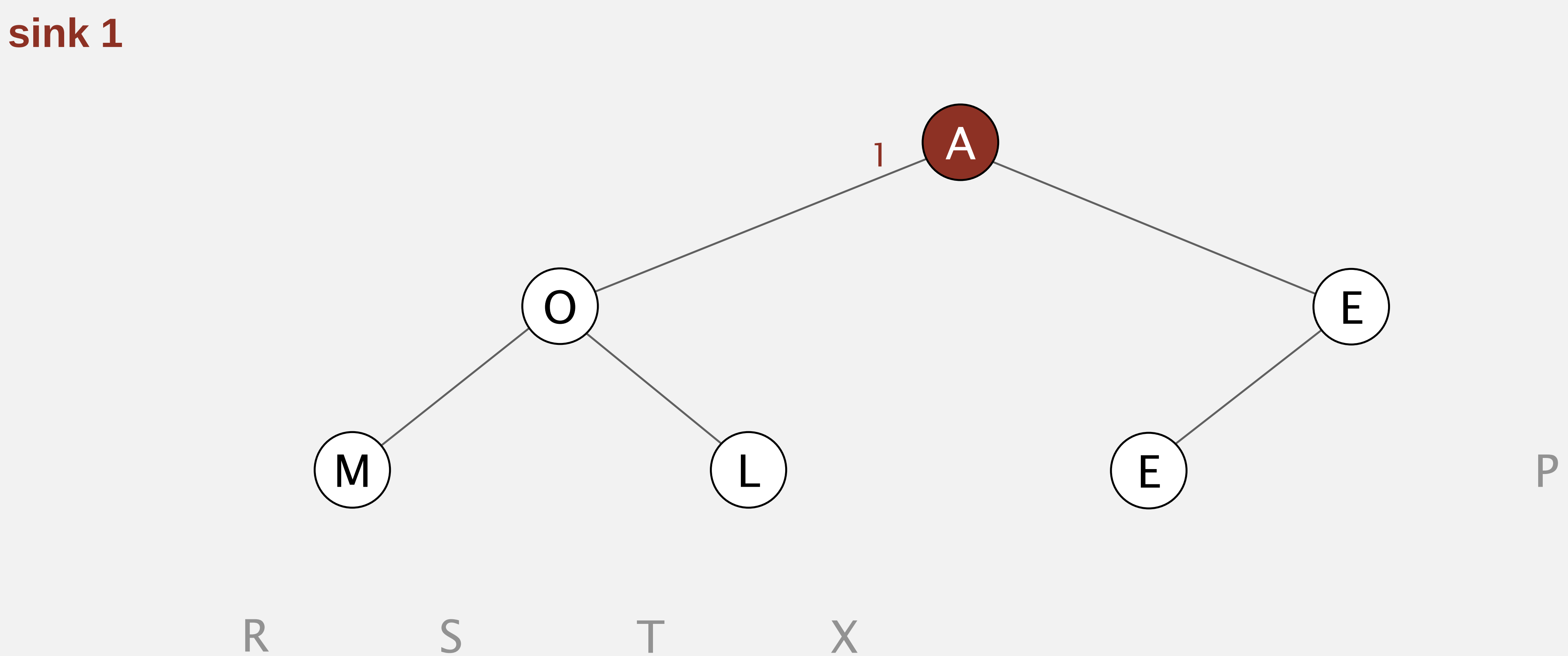
exchange 1 and 7



A	O	E	M	L	E	P	R	S	T	X
1						7				

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

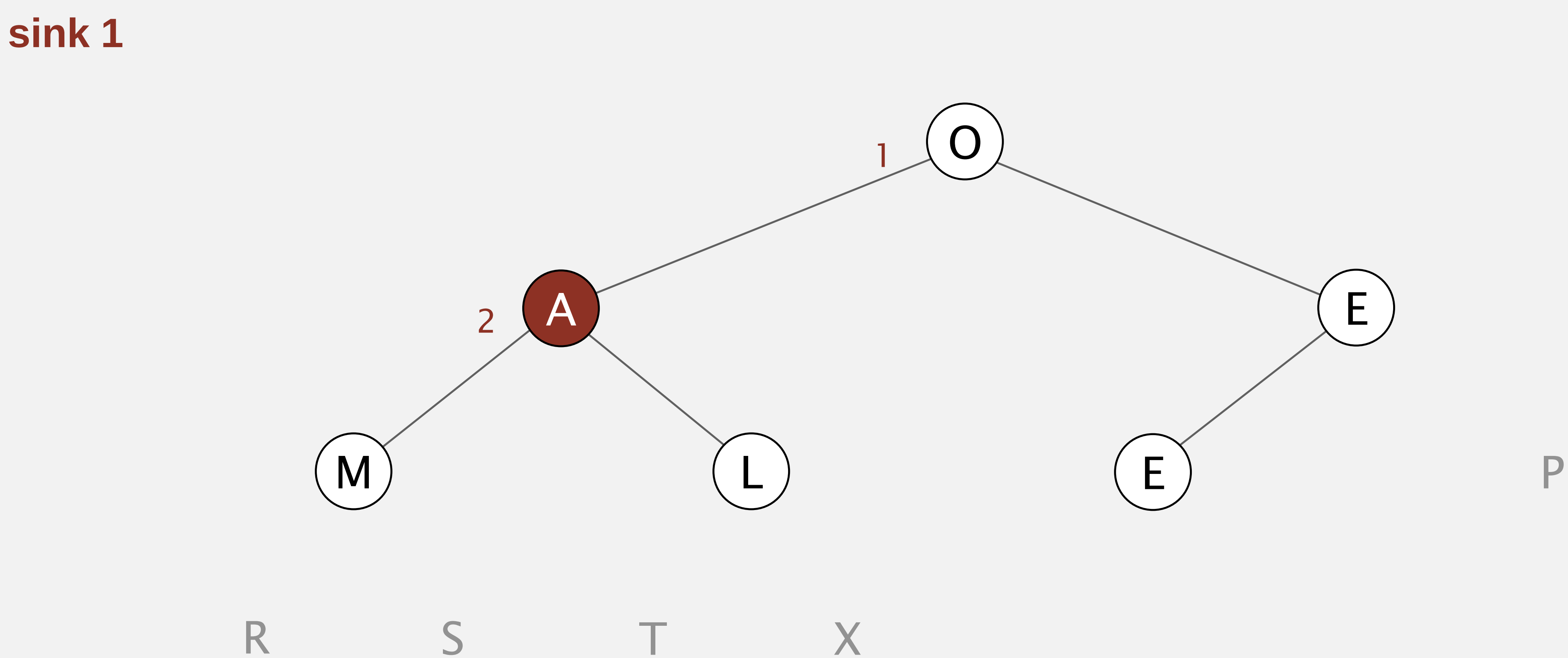


A	O	E	M	L	E	P	R	S	T	X

1

Heapsort demo

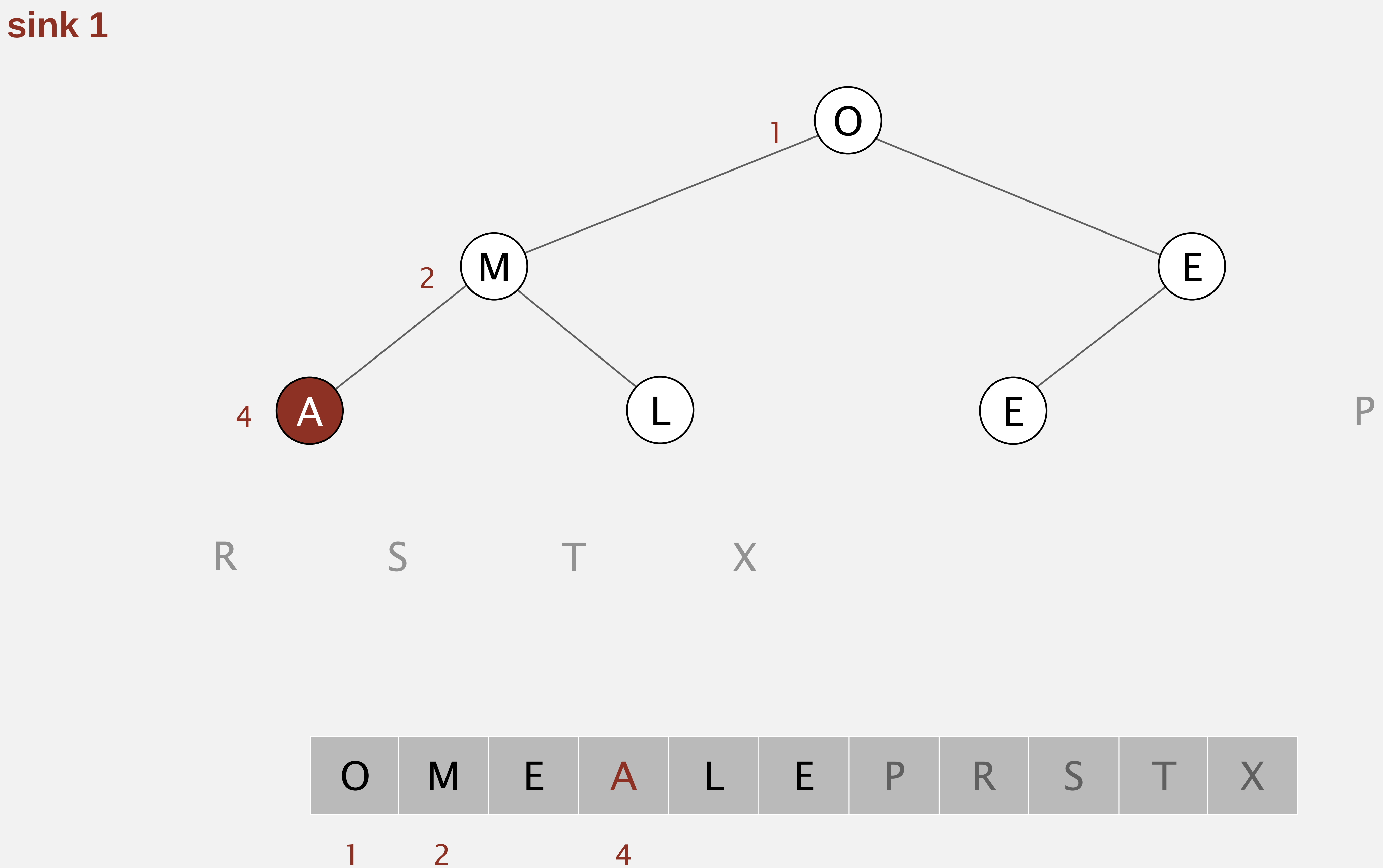
Sortdown. Repeatedly delete the largest remaining item.



O	A	E	M	L	E	P	R	S	T	X
1	2									

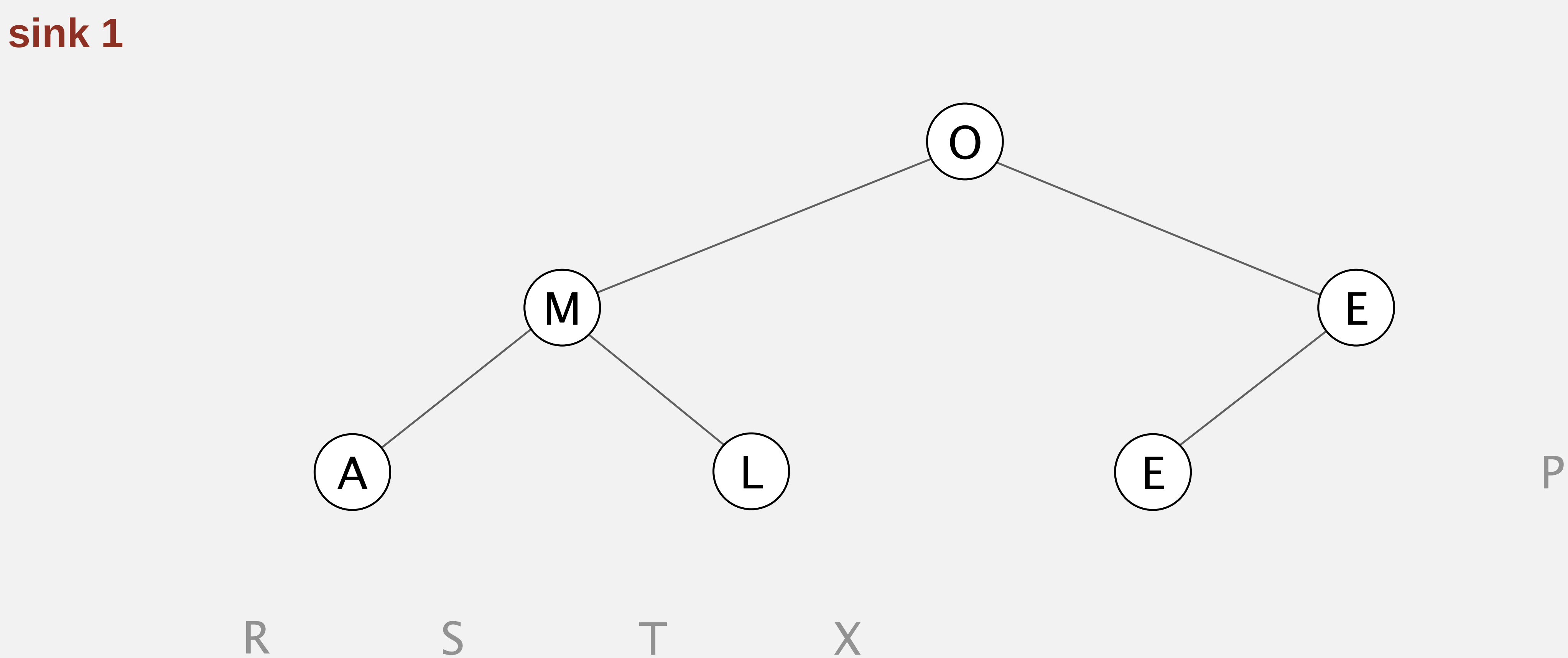
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

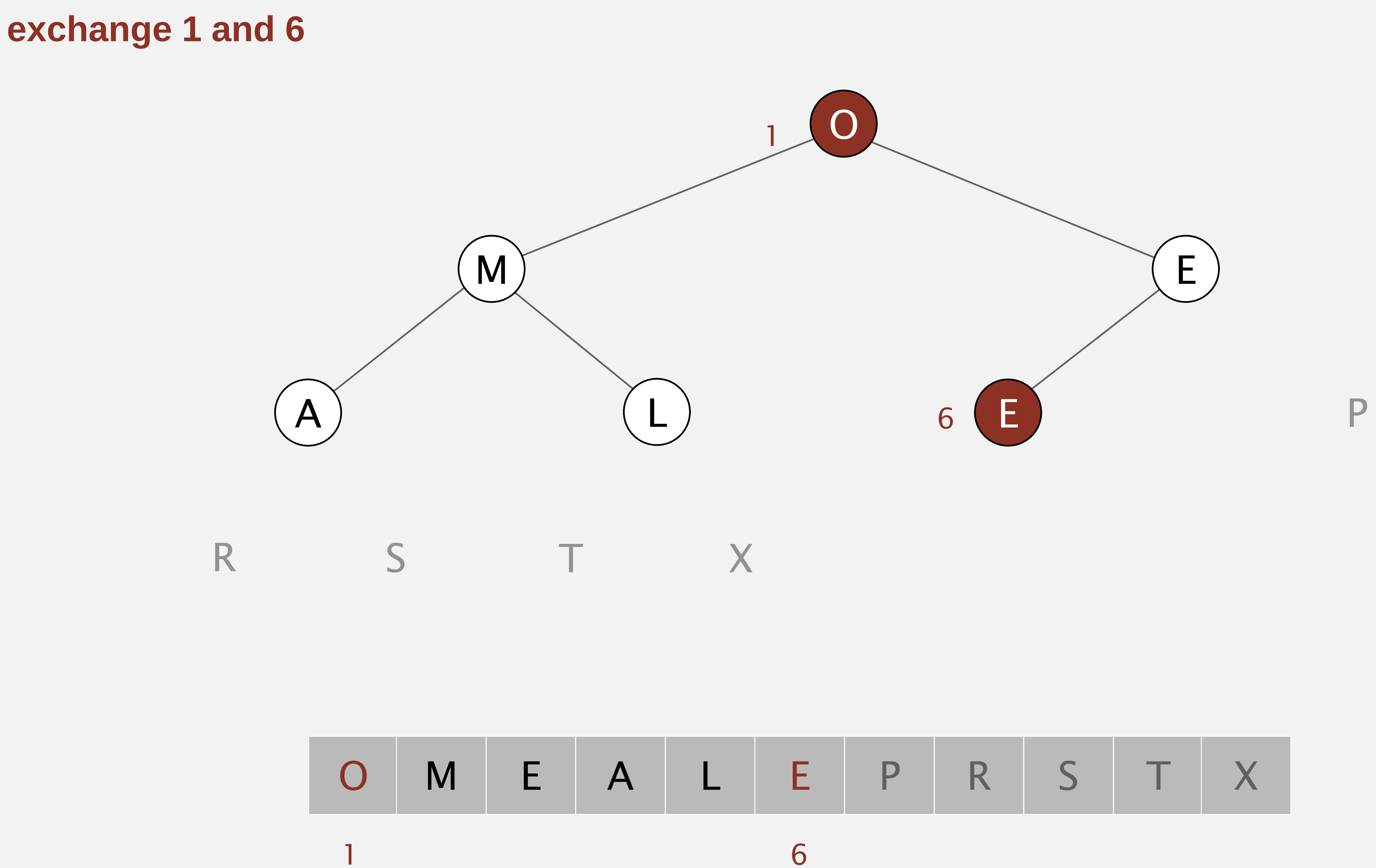
Sortdown. Repeatedly delete the largest remaining item.



O	M	E	A	L	E	P	R	S	T	X

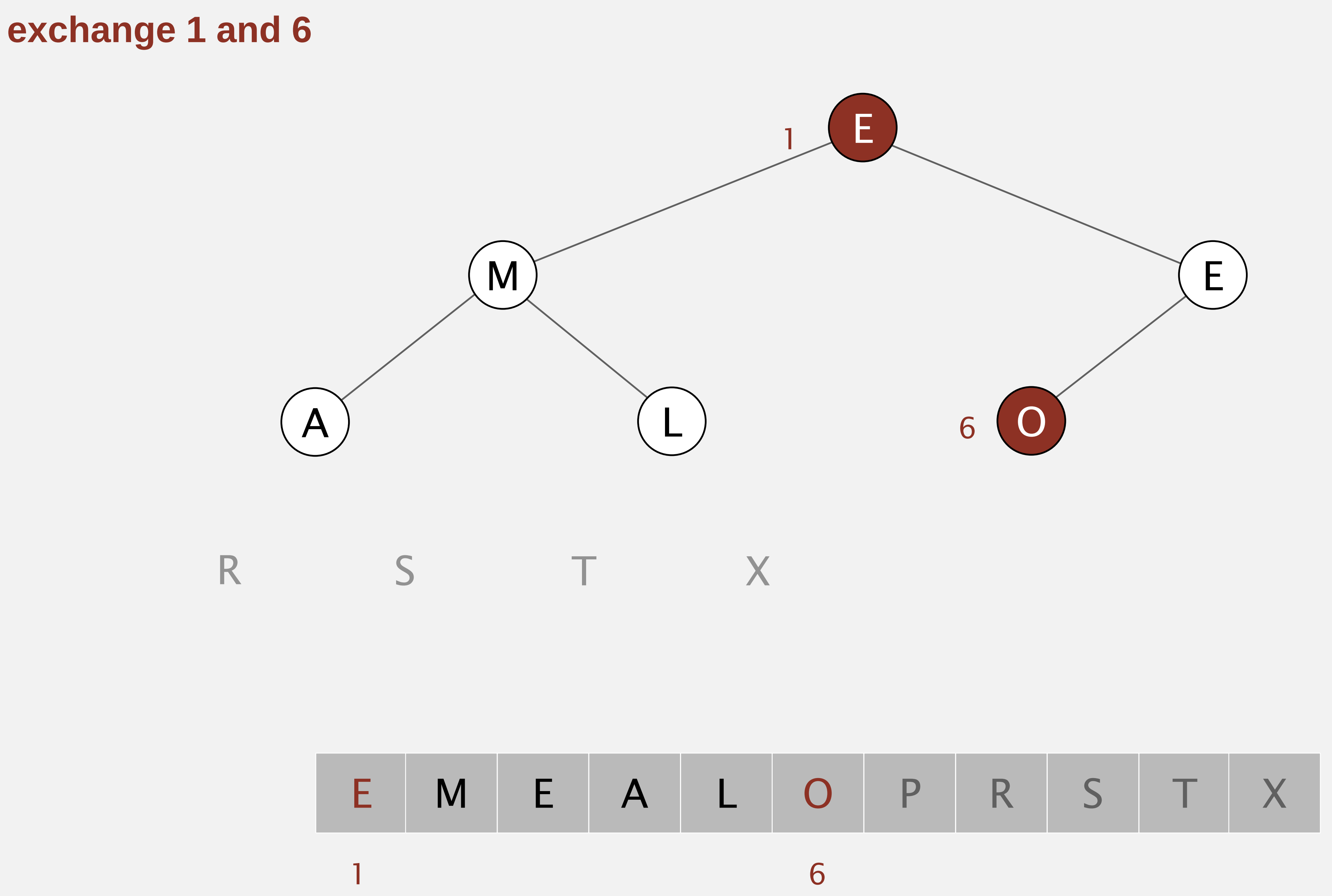
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



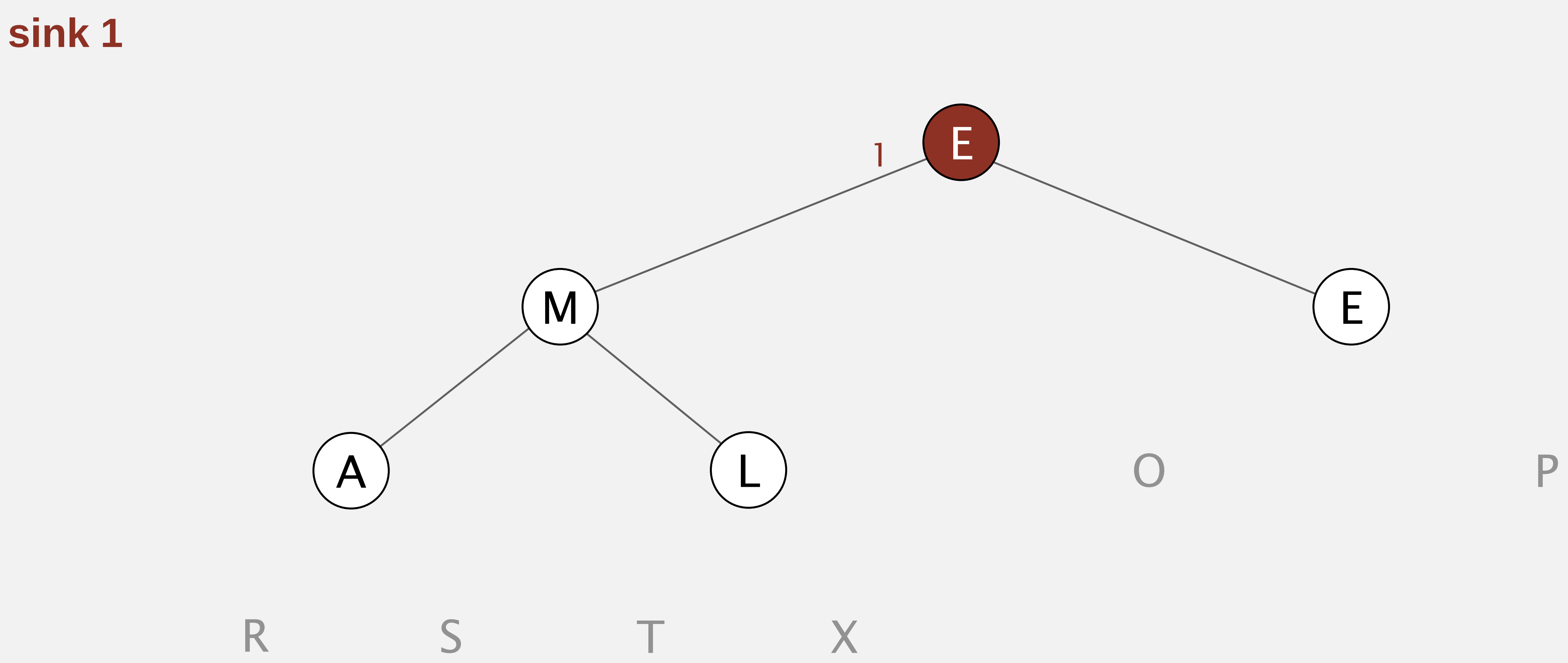
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

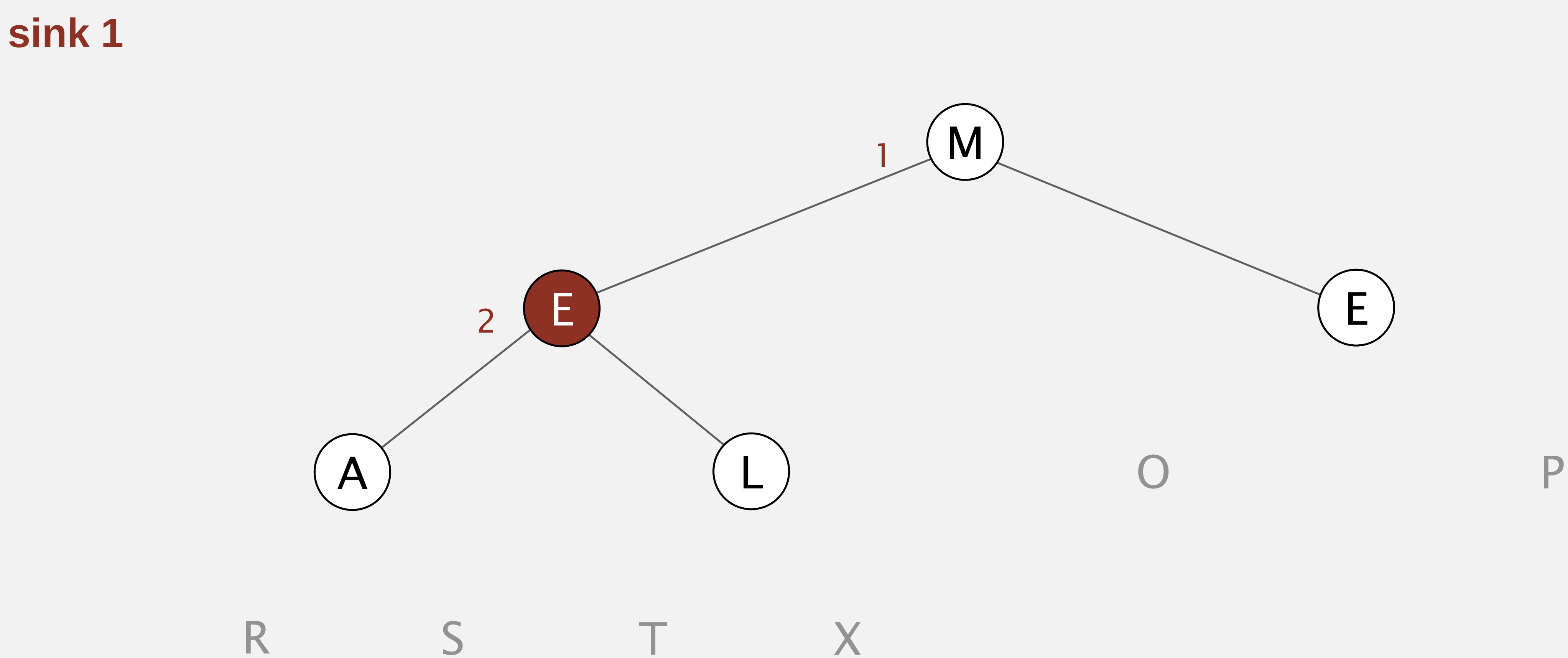


E	M	E	A	L	O	P	R	S	T	X

1

Heapsort demo

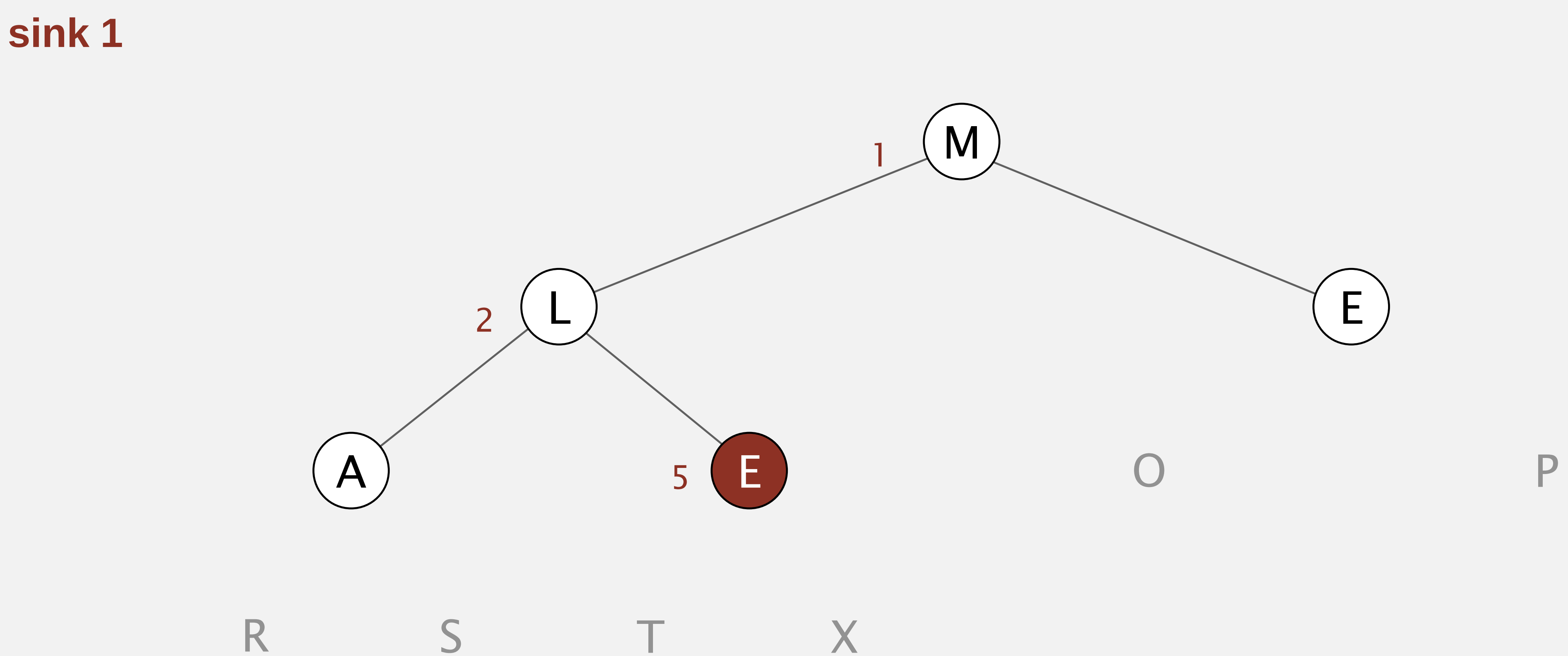
Sortdown. Repeatedly delete the largest remaining item.



M	E	E	A	L	O	P	R	S	T	X
1	2									

Heapsort demo

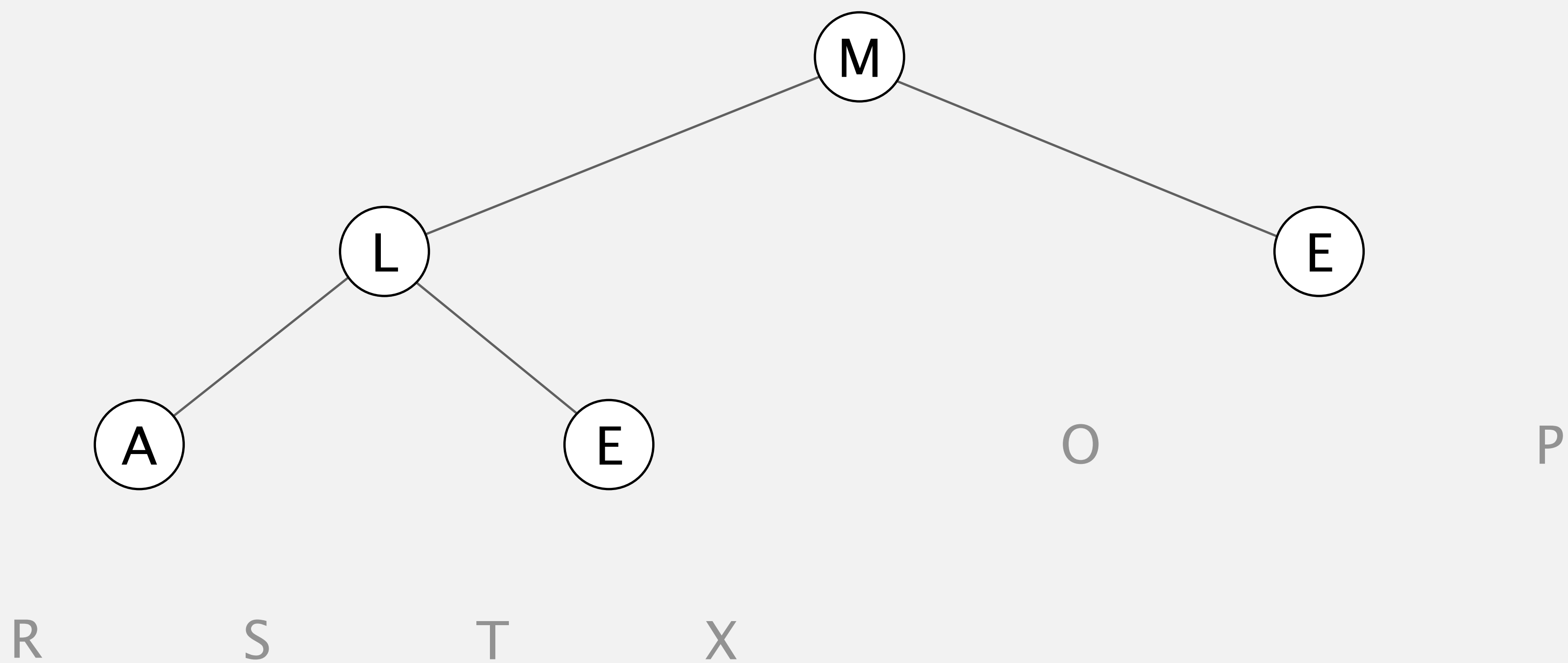
Sortdown. Repeatedly delete the largest remaining item.



M	L	E	A	E	O	P	R	S	T	X
1	2			5						

Heapsort demo

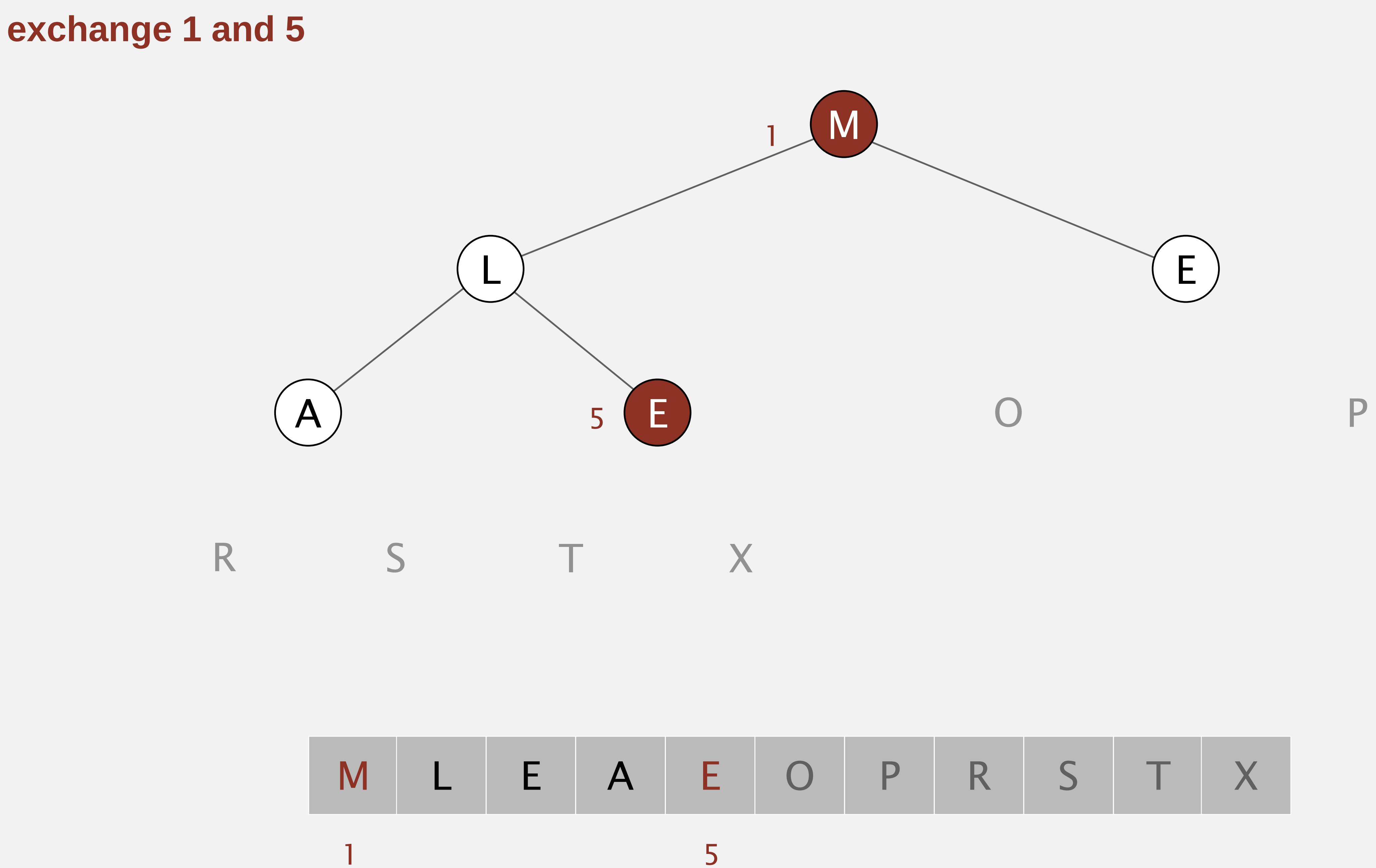
Sortdown. Repeatedly delete the largest remaining item.



M	L	E	A	E	O	P	R	S	T	X

Heapsort demo

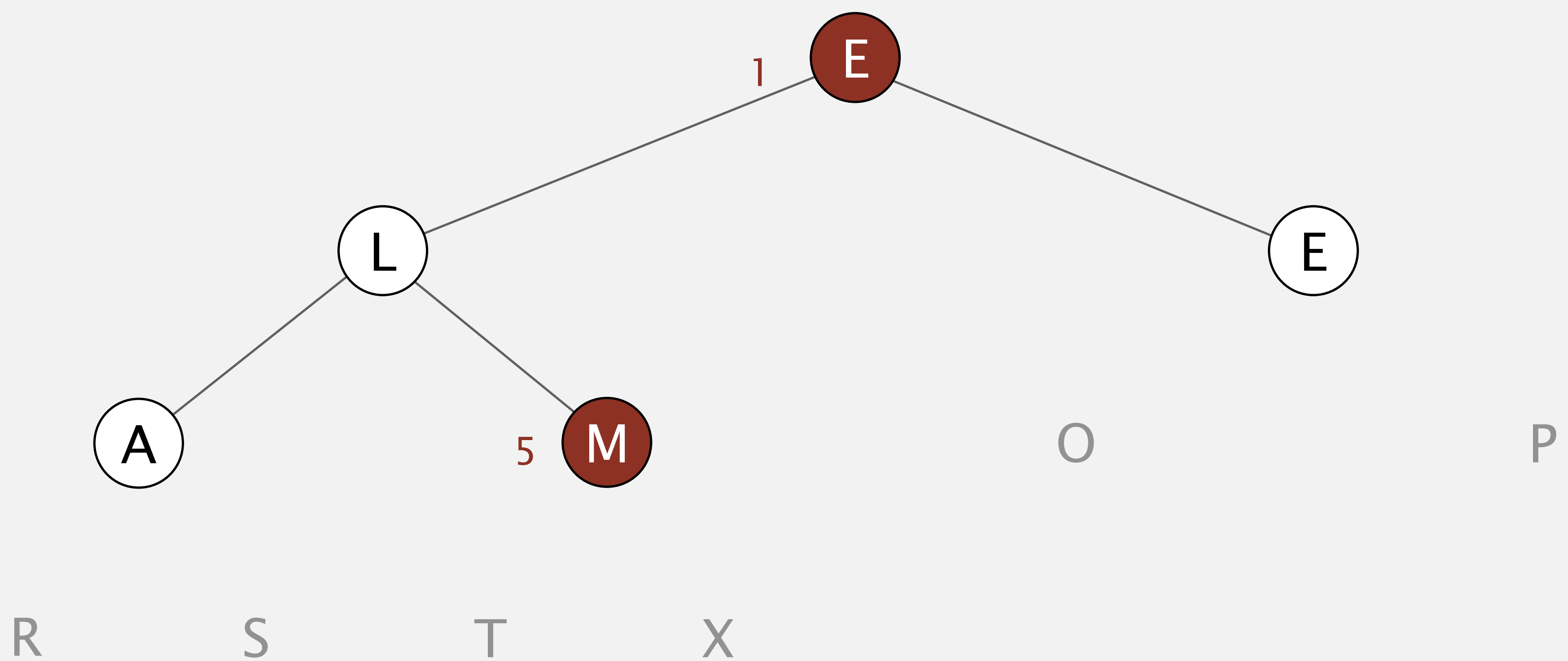
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5

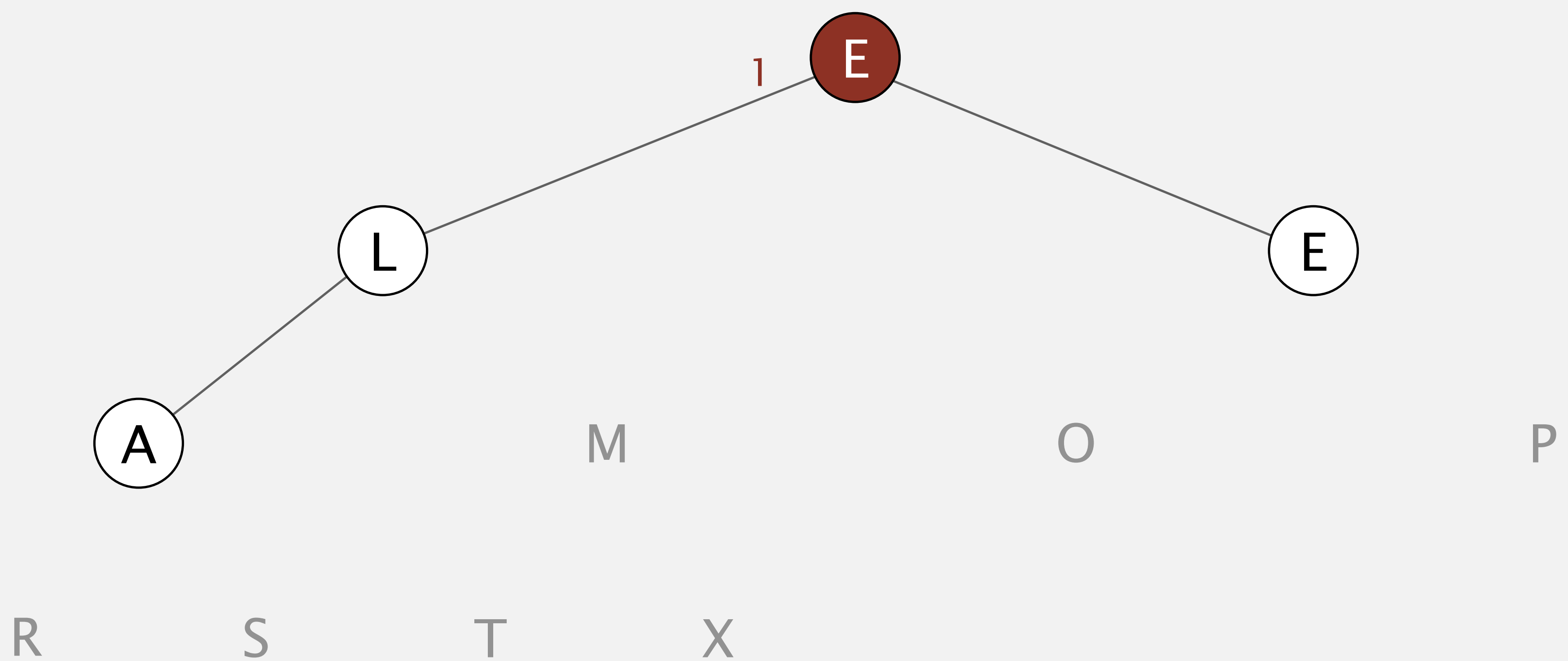


E	L	E	A	M	O	P	R	S	T	X
1				5						

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1



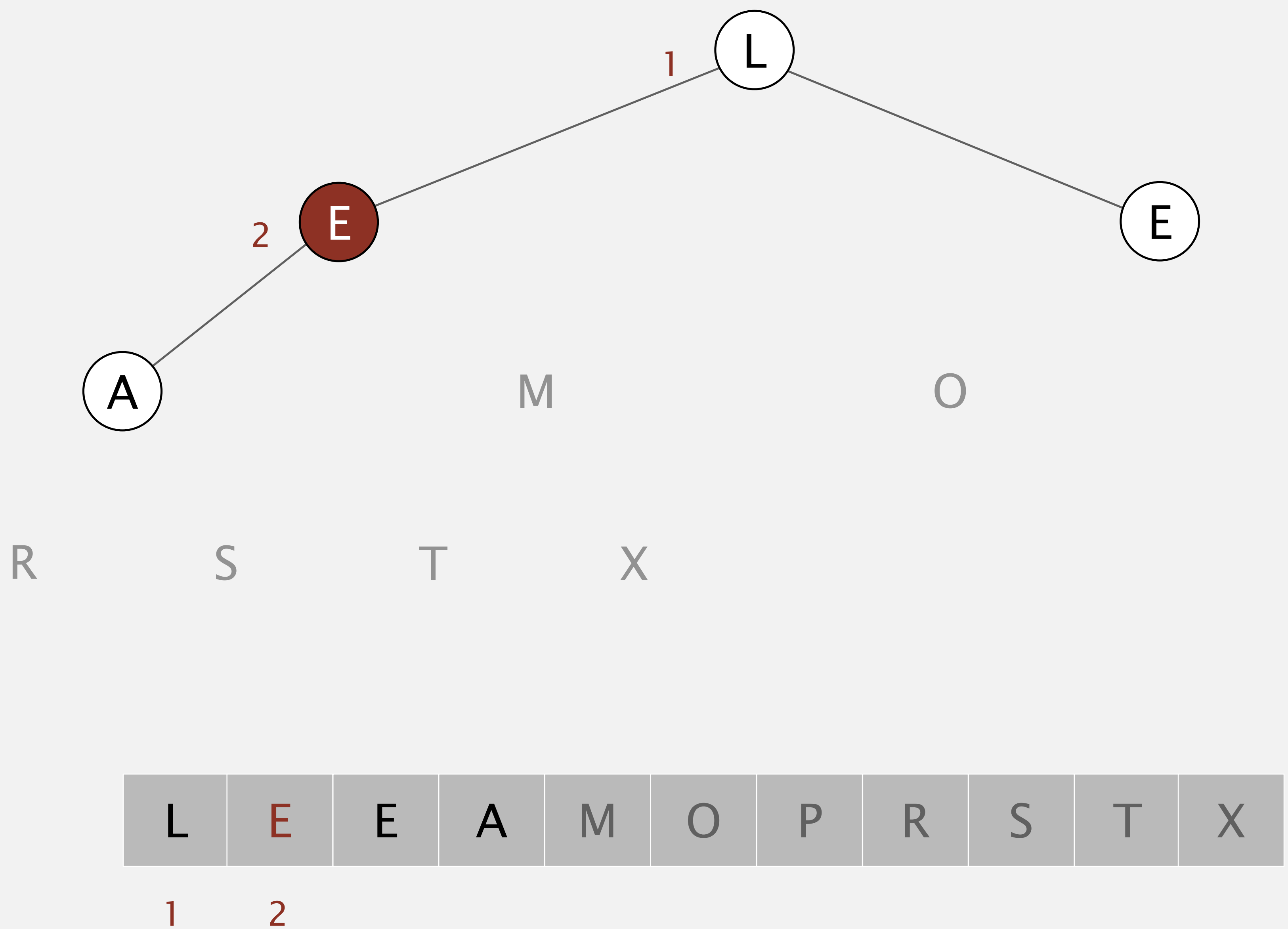
E	L	E	A	M	O	P	R	S	T	X

1

Heapsort demo

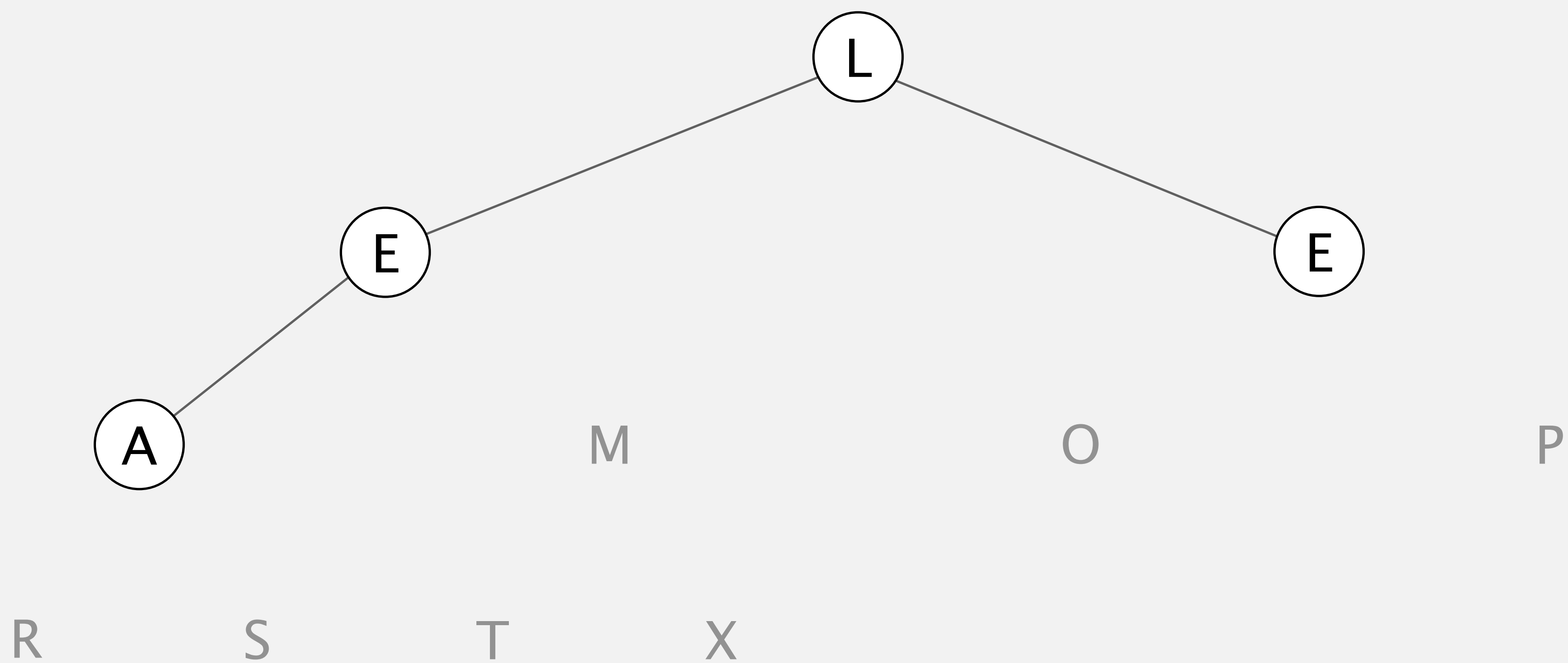
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort demo

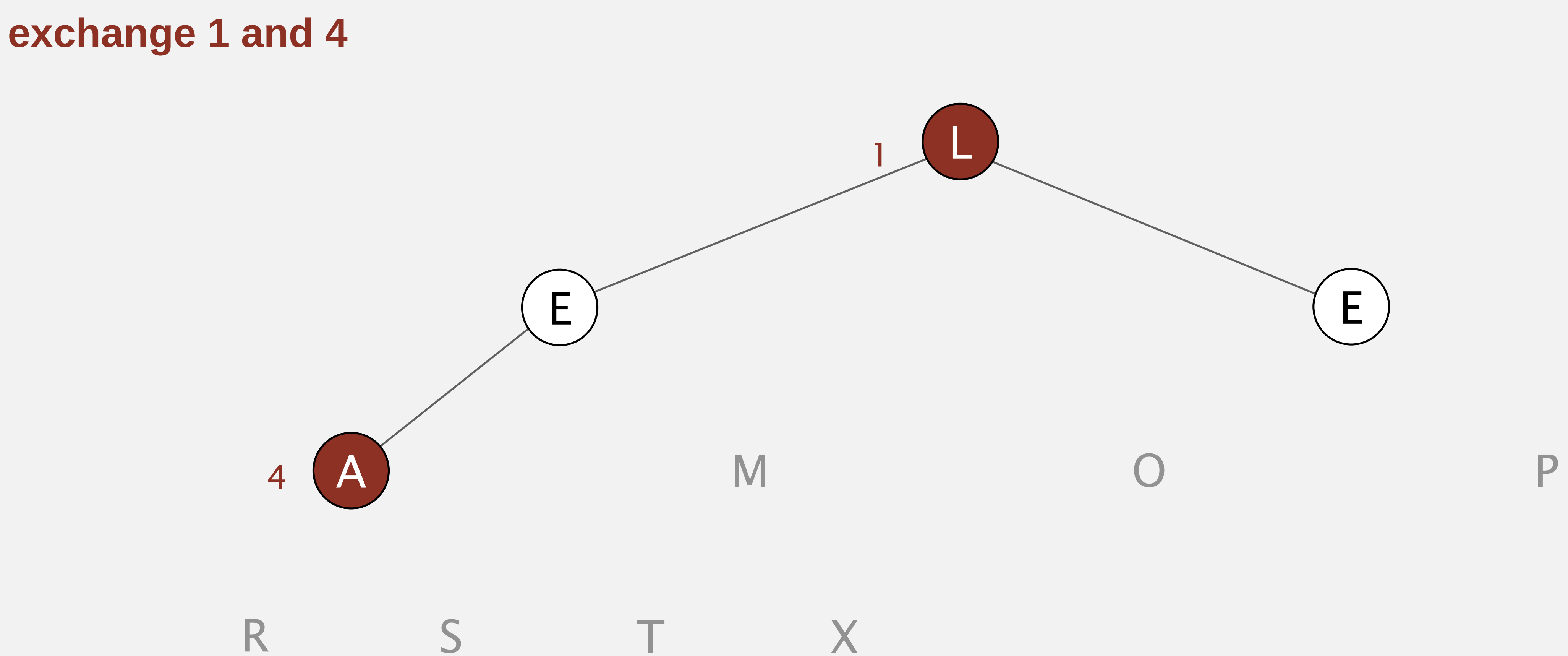
Sortdown. Repeatedly delete the largest remaining item.

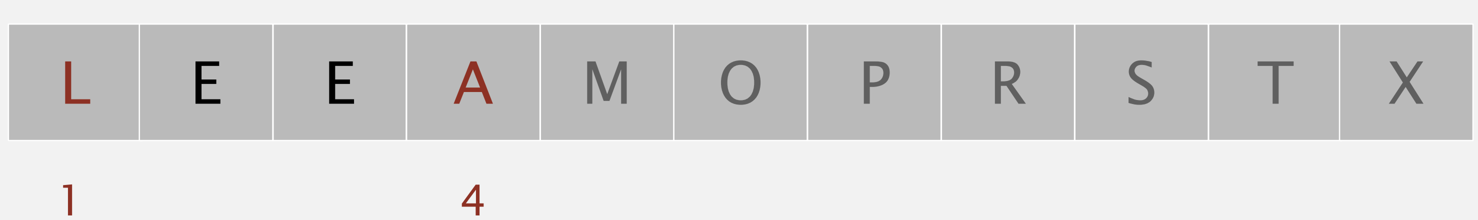


L	E	E	A	M	O	P	R	S	T	X

Heapsort demo

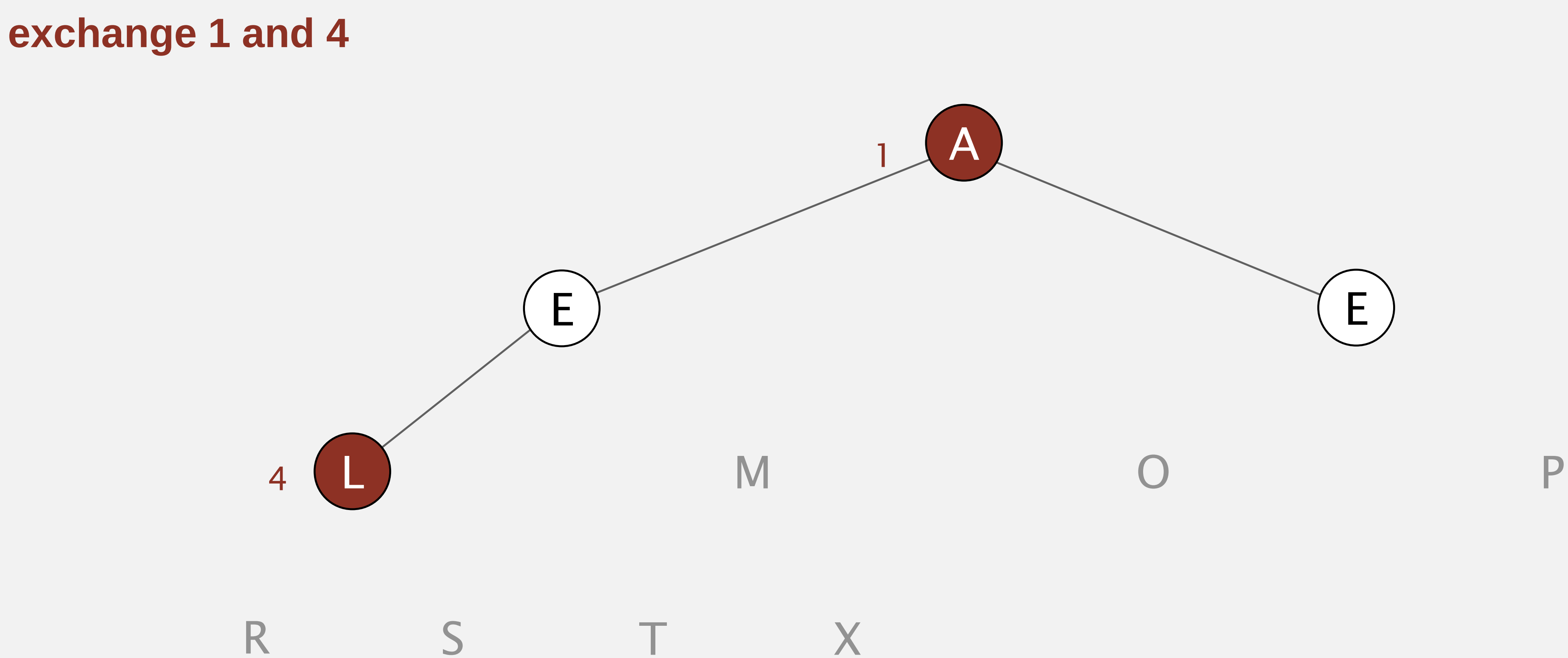
Sortdown. Repeatedly delete the largest remaining item.

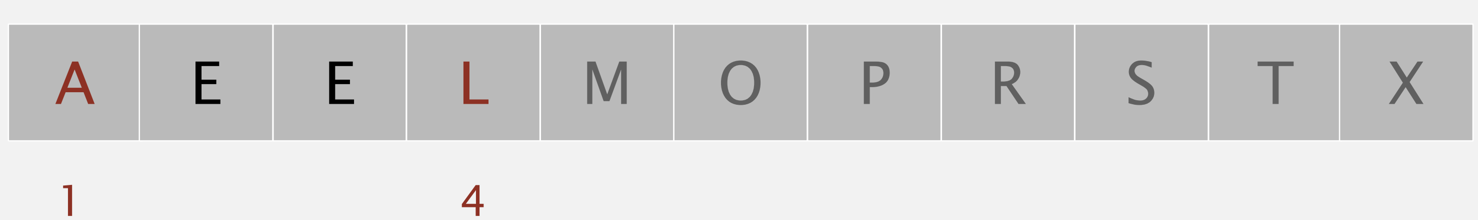




Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

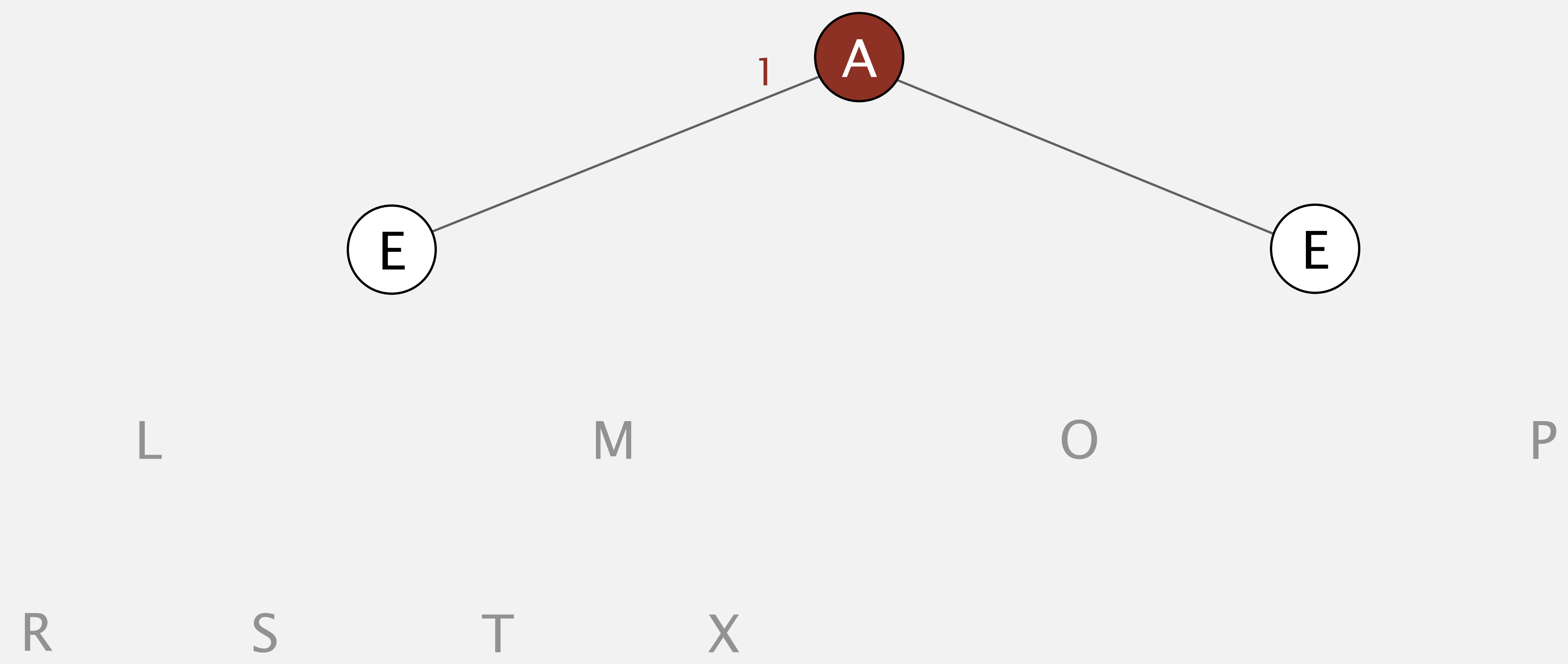




Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

sink 1

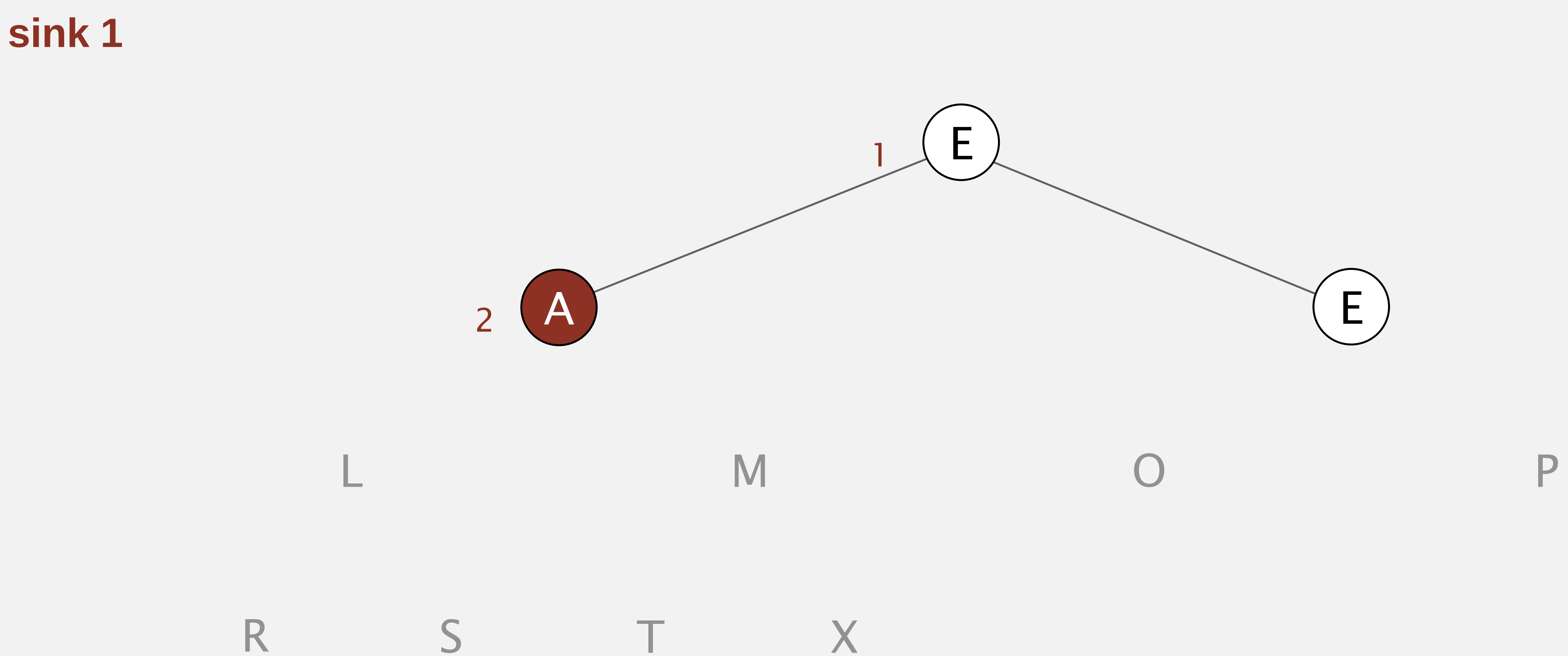


A	E	E	L	M	O	P	R	S	T	X

1

Heapsort demo

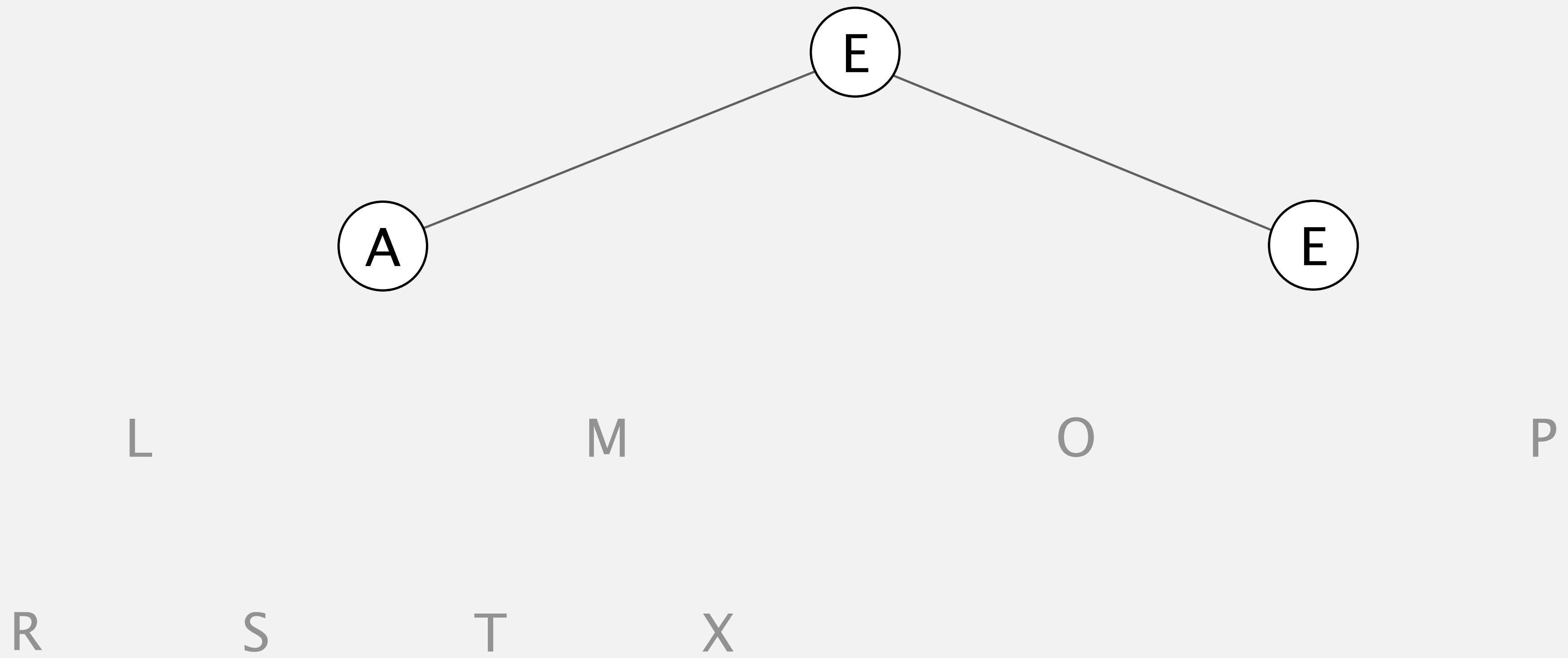
Sortdown. Repeatedly delete the largest remaining item.



E	A	E	L	M	O	P	R	S	T	X
1	2									

Heapsort demo

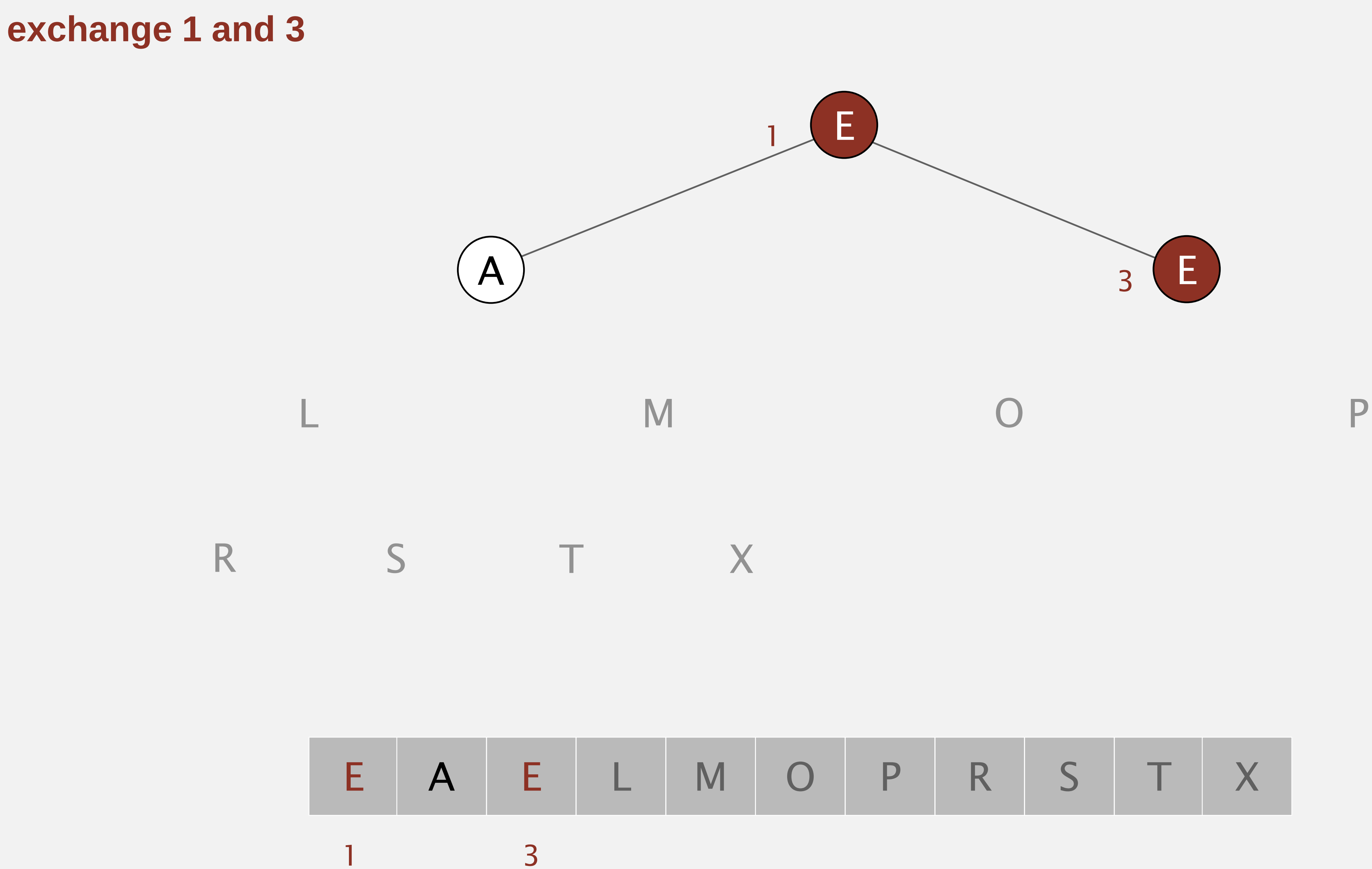
Sortdown. Repeatedly delete the largest remaining item.



E	A	E	L	M	O	P	R	S	T	X

Heapsort demo

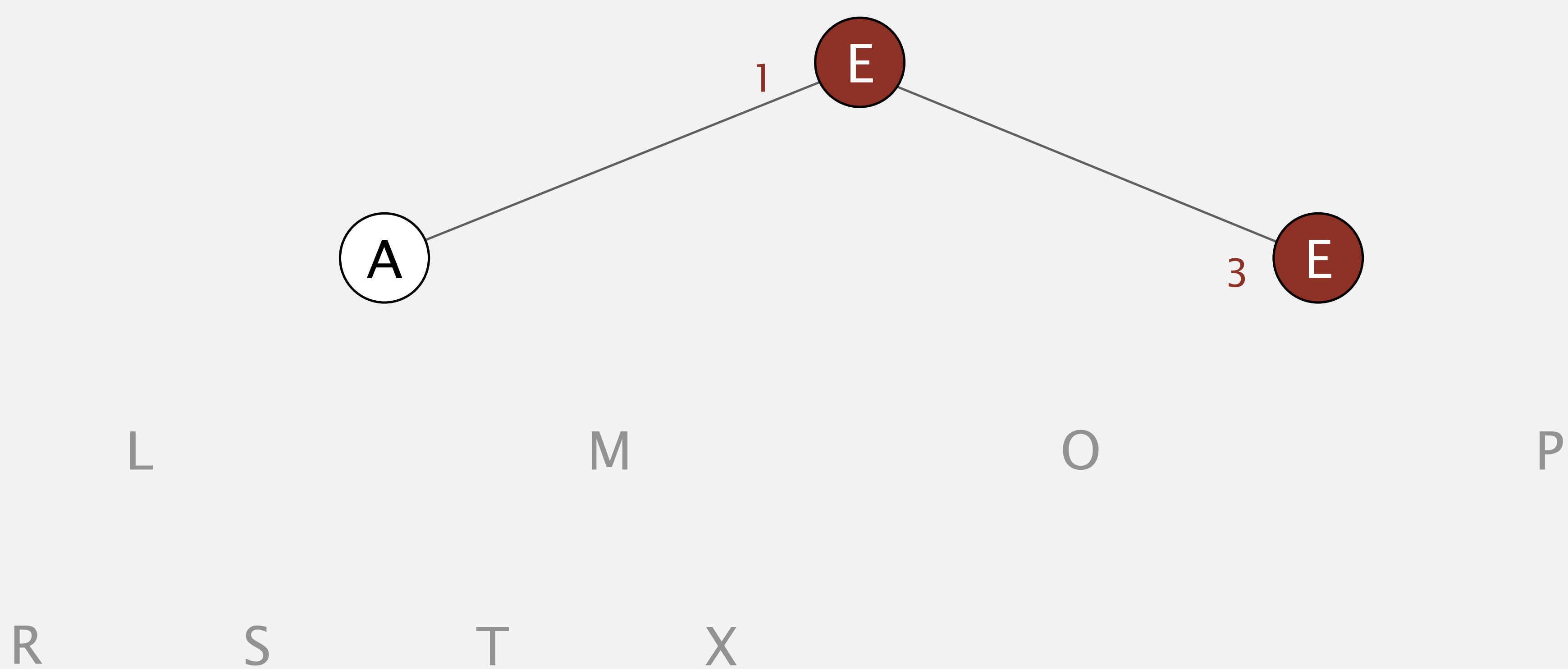
Sortdown. Repeatedly delete the largest remaining item.



Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

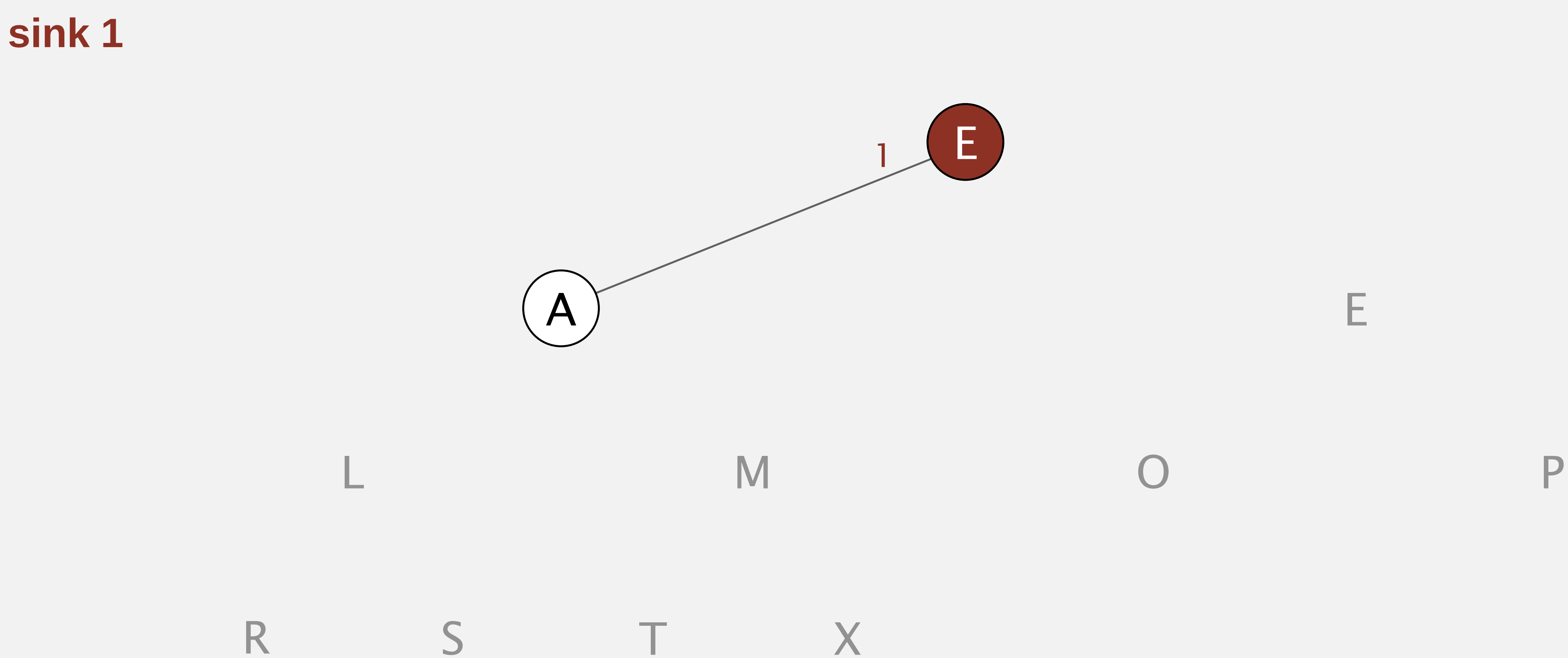
exchange 1 and 3



E	A	E	L	M	O	P	R	S	T	X
1		3								

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

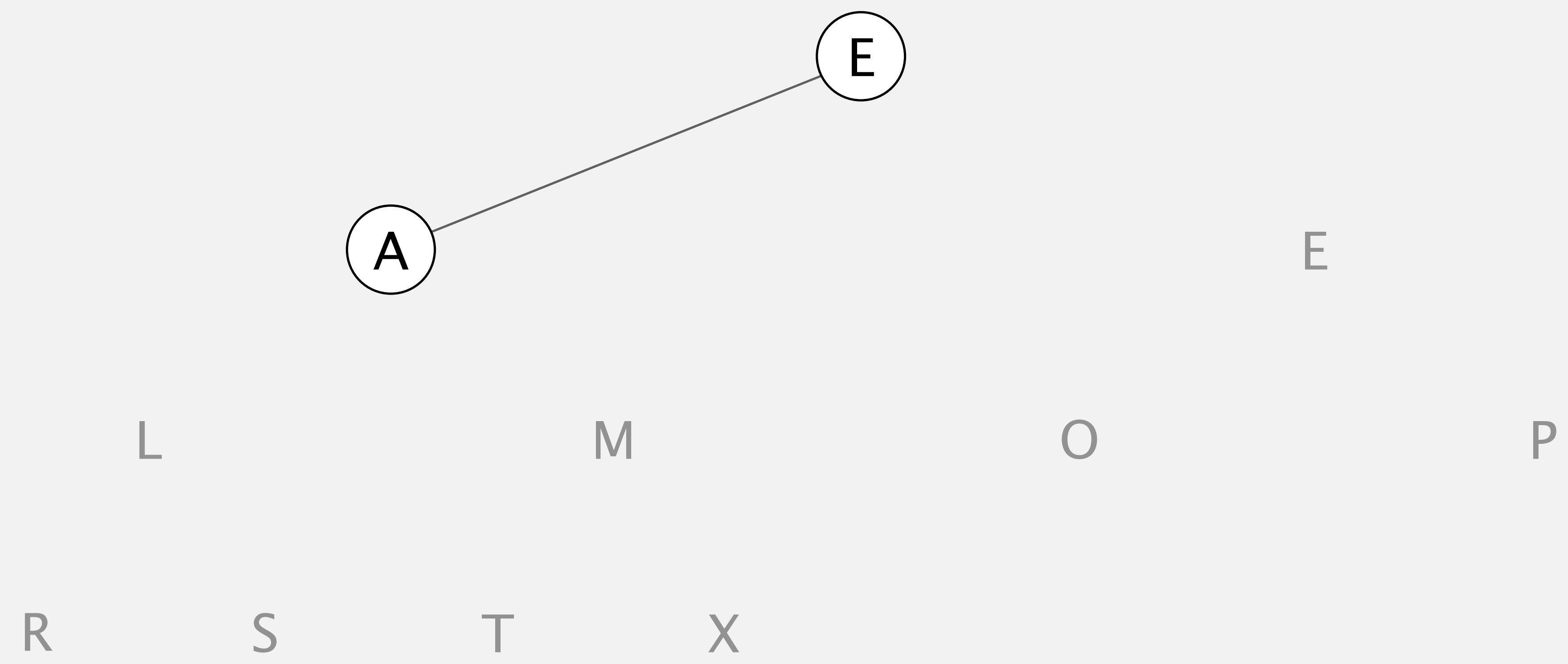


E	A	E	L	M	O	P	R	S	T	X

1

Heapsort demo

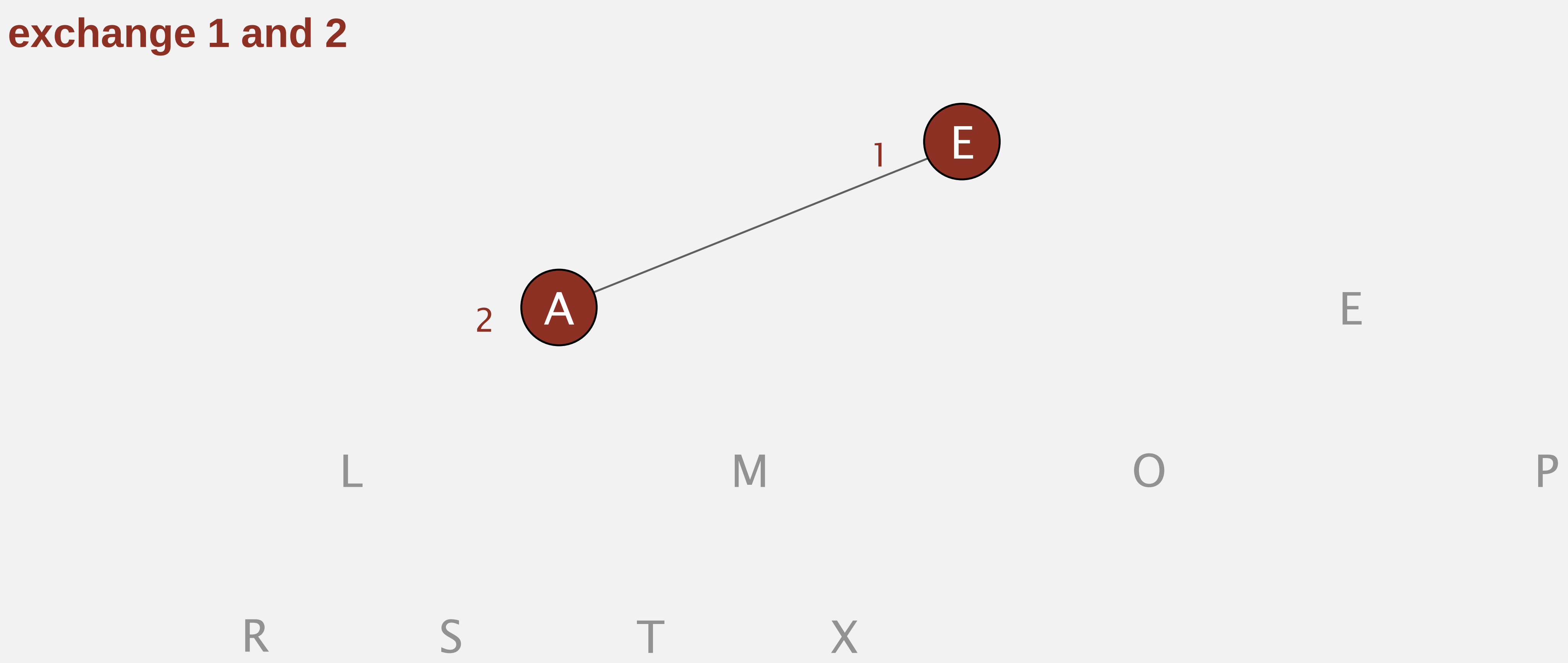
Sortdown. Repeatedly delete the largest remaining item.



E	A	E	L	M	O	P	R	S	T	X

Heapsort demo

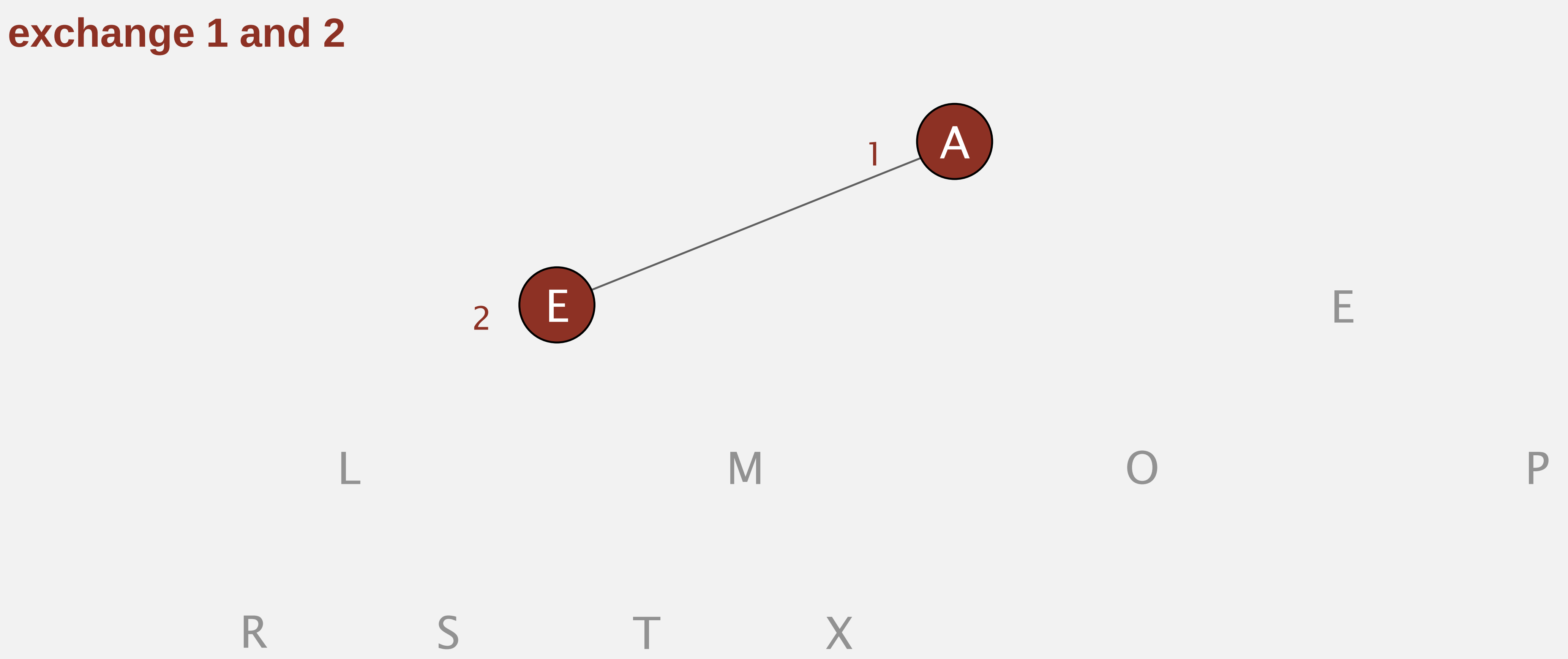
Sortdown. Repeatedly delete the largest remaining item.



E	A	E	L	M	O	P	R	S	T	X
1	2									

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

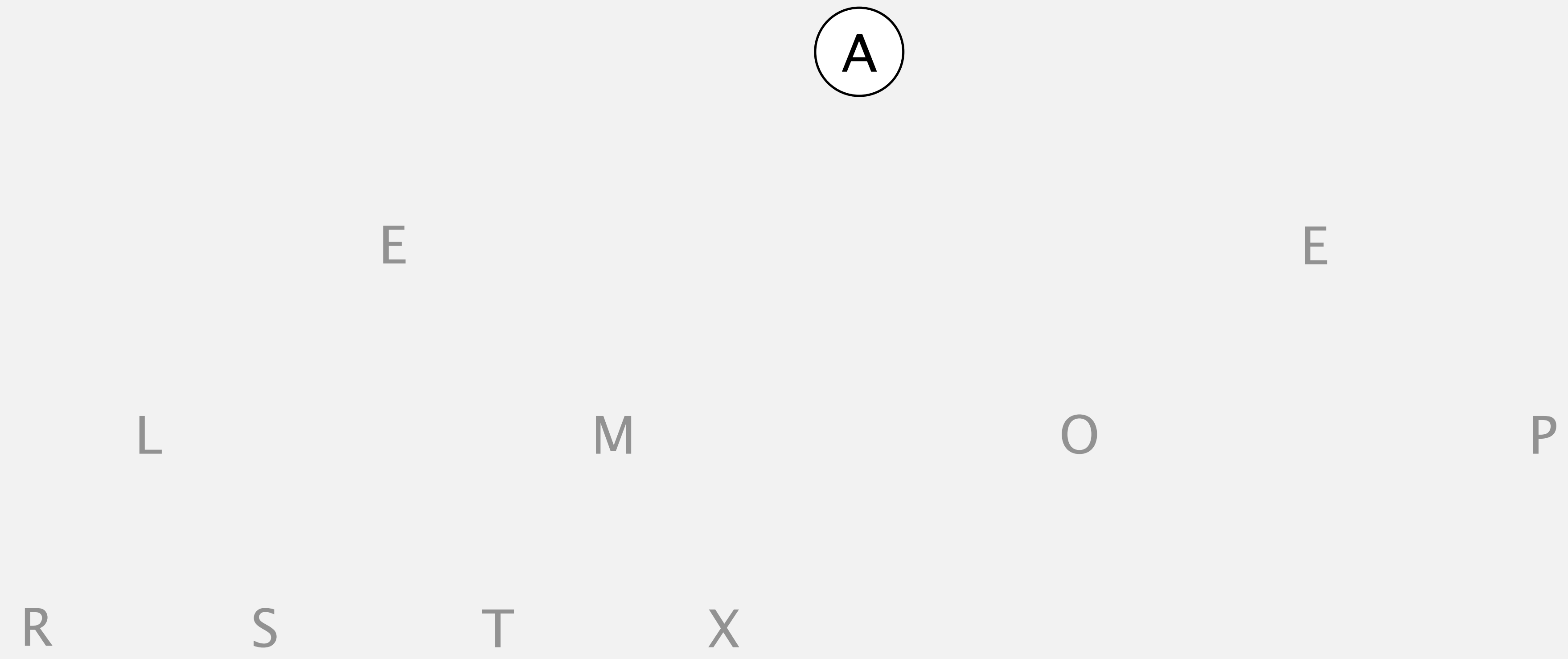


A	E	E	L	M	O	P	R	S	T	X

1 2

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



A	E	E	L	M	O	P	R	S	T	X

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

end of sortdown phase

A

E E

L M O P

R S T X

A	E	E	L	M	O	P	R	S	T	X

Heapsort demo

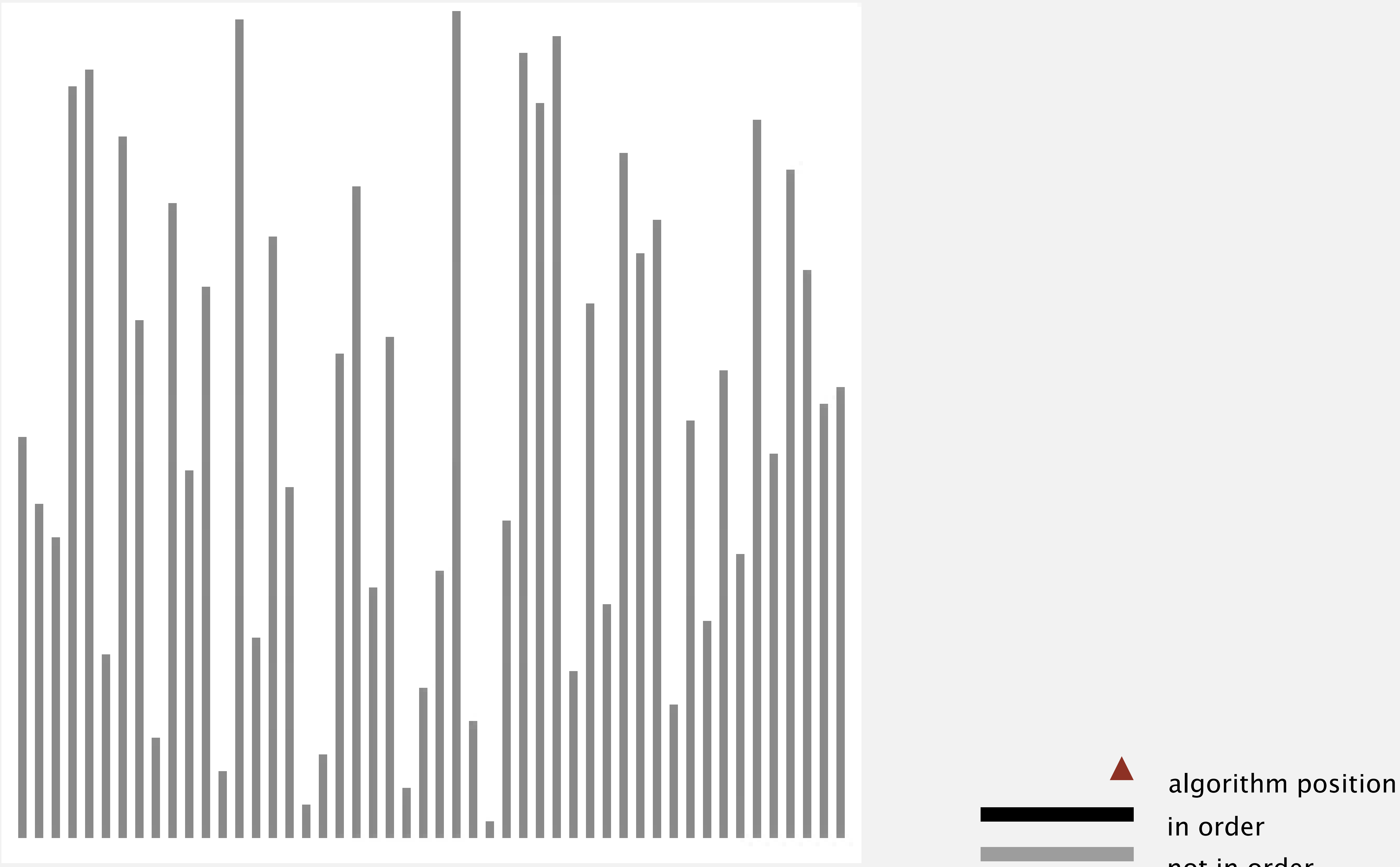
Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

Heapsort animation

50 random items



http://www.sorting-algorithms.com/heap-sort

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2\,N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2\,N \lg N$ compares and exchanges.

algorithm can be improved to ~ 1 N lg N

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← N log N worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:

- Inner loop longer than quicksort's.
- Makes poor use of cache.
- Not stable.

advanced tricks for improving

Sorting algorithms: summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
merge		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
quick	✔		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
heap	✔		N	$2 N \lg N$	$2 N \lg N$	$N \log N$ guarantee; in-place
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail

http://algs4.cs.princeton.edu

Symbol Tables

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

http://algs4.cs.princeton.edu

Symbol Tables

- *API*
- *elementary implementations*
- *ordered operations*

Symbol tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

key　　value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N-1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an associative array

every object is an associative array

table is the only primitive data structure

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```
public class ST<Key, Value>
```

	`ST()`	*create an empty symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table* ← `a[key] = val;`
`Value`	`get(Key key)`	*value paired with key* ← `a[key]`
`boolean`	`contains(Key key)`	*is there a value paired with key?*
`void`	`delete(Key key)`	*remove key (and its value) from table*
`boolean`	`isEmpty()`	*is the table empty?*
`int`	`size()`	*number of key-value pairs in the table*
`Iterable<Key>`	`keys()`	*all the keys in the table*

Conventions

- Values are not `null`. ← Java allows null value
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{  return get(key) != null;  }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{  put(key, null);  }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are `Comparable`, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `Integer`, `Double`, `String`, `java.io.File`, …
- Mutable in Java: `StringBuilder`, `java.net.URL`, arrays, ...

Equality test

All Java classes inherit a method `equals()`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.

equivalence relation

do x and y refer to the same object?

Default implementation. `(x == y)`

Customized implementations. `Integer`, `Double`, `String`, `java.io.File`, …

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

```
public       class Date implements
Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...

   public boolean equals(Date that)
   {

      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
   }
}
```

check that all significant fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

```
public final class Date implements
Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...
   public boolean equals(Object y)
   {
      if (y == this) return true;
      if (y == null) return false;
      if (y.getClass() != this.getClass())
         return false;
      Date that = (Date) y;
      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
   }
}
```

must be Object.
Why?

optimize for true object equality

check for null

objects must be in the same class
(religion: getClass() vs. instanceof)

cast is guaranteed to succeed

check that all significant
fields are the same

Equals design

"Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type and cast.
- Compare each significant field:
  - if field is a primitive type, use == ← but use Double.compare() with double (or otherwise deal with -0.0 and NaN)
  - if field is an object, use `equals()` ← apply rule recursively
  - if field is an array, apply to each entry ← can use `Arrays.deepEquals(a, b)` but not `a.equals(b)`

Best practices.

- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

`x.equals(y)` if and only if `(x.compareTo(y) == 0)`

http://algs4.cs.princeton.edu

Symbol Tables

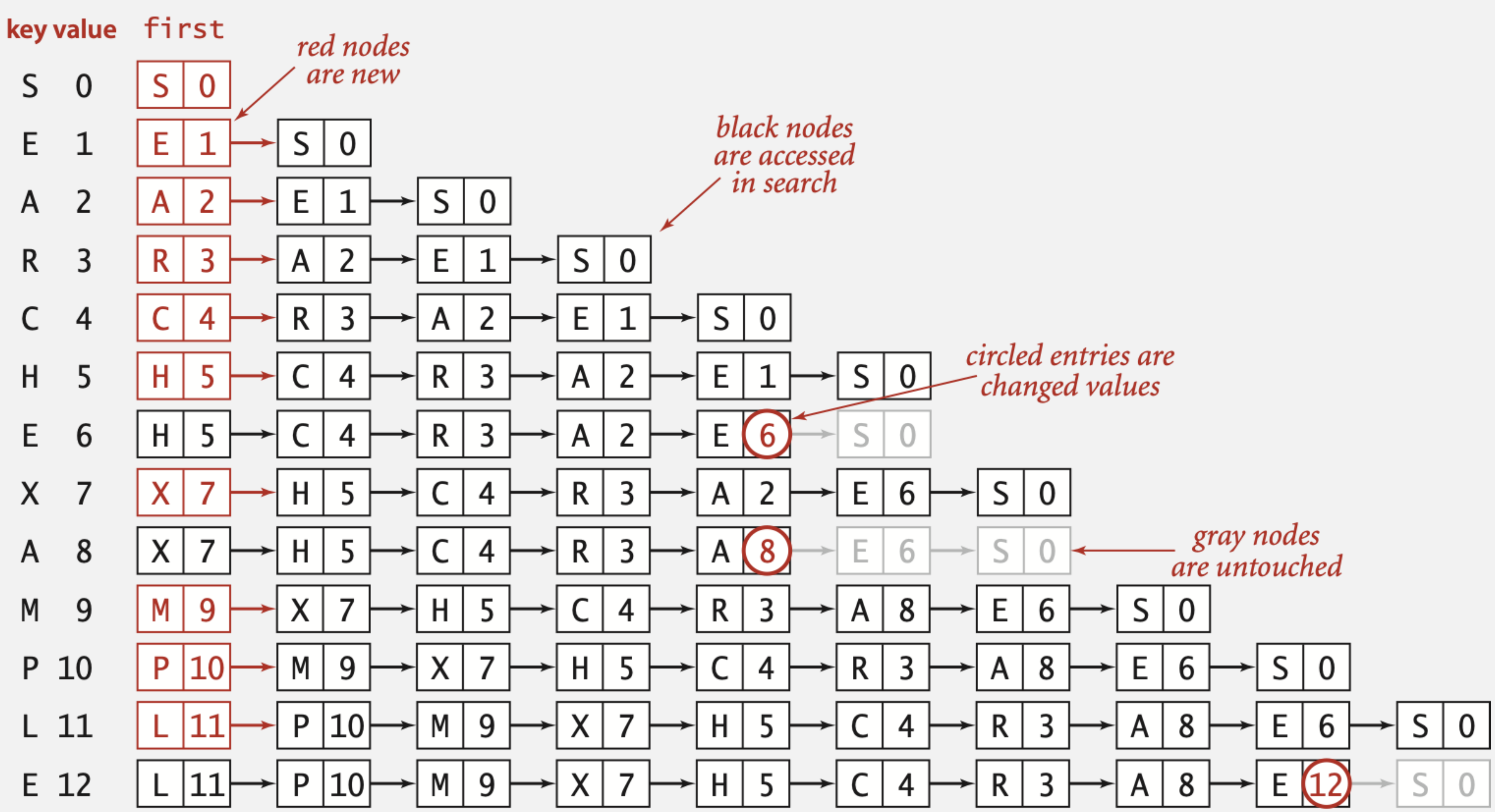
- API
- *elementary implementations*
- ordered operations

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	`equals()`

Challenge. Efficient implementations of both search and insert.

Binary search in an ordered array

Data structure. Maintain an ordered array of key-value pairs.

Rank helper function. How many keys $< k$?

keys[]

0	1	2	3	4	5	6	7	8	9
A	C	E	H	L	M	P	R	S	X

successful search for P

lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X

entries in black are a[lo..hi]

entry in red is a[m]

loop exits with keys[m] = P: *return* 6

unsuccessful search for Q

lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

loop exits with lo > hi: *return* 7

Binary search: Java implementation

```
public Value get(Key key)
{
   if (isEmpty()) return null;
   int i = rank(key);
   if (i < N && keys[i].compareTo(key) == 0) return vals[i];
   else return null;
}
private int rank(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       int cmp = key.compareTo(keys[mid]);
       if      (cmp  < 0) hi = mid - 1;
       else if (cmp  > 0) lo = mid + 1;
       else               return mid;
  }
  return lo;
}
```

number of keys < key

Binary search: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

		keys[]											vals[]									
key	value	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

entries in red were inserted

entries in gray did not move

entries in black moved to the right

circled entries are changed values

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	`equals()`
binary search (ordered array)	$\log N$	$2*N$	$\log N$	N	`compareTo()`

Challenge. Efficient implementations of both search and insert.

http://algs4.cs.princeton.edu

SYMBOL TABLES

- *API*
- *elementary implementations*
- *ordered operations*

Examples of ordered symbol table API

	keys	*values*
min() →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13) →	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00) →	09:03:13	Chicago
	09:10:11	Seattle
select(7) →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00) →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00) →	09:35:21	Chicago
	09:36:14	Seattle
max() →	09:37:44	Phoenix

size(09:15:00, 09:25:00) *is* 5

rank(09:10:25) *is* 7

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value>
```

	`...`	
`Key`	`min()`	*smallest key*
`Key`	`max()`	*largest key*
`Key`	`floor(Key key)`	*largest key less than or equal to key*
`Key`	`ceiling(Key key)`	*smallest key greater than or equal to key*
`int`	`rank(Key key)`	*number of keys less than key*
`Key`	`select(int k)`	*key of rank k*
`void`	`deleteMin()`	*delete smallest key*
`void`	`deleteMax()`	*delete largest key*
`int`	`size(Key lo, Key hi)`	*number of keys between lo and hi*
`Iterable<Key>`	`keys()`	*all keys, in sorted order*
`Iterable<Key>`	`keys(Key lo, Key hi)`	*keys between lo and hi, in sorted order*

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	$\log N$
insert / delete	N	N
min / max	N	1
floor / ceiling	N	$\log N$
rank	N	$\log N$
select	N	1
ordered iteration	$N \log N$	N

order of growth of the running time for ordered symbol table operations

http://algs4.cs.princeton.edu

BINARY SEARCH TREES

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

http://algs4.cs.princeton.edu

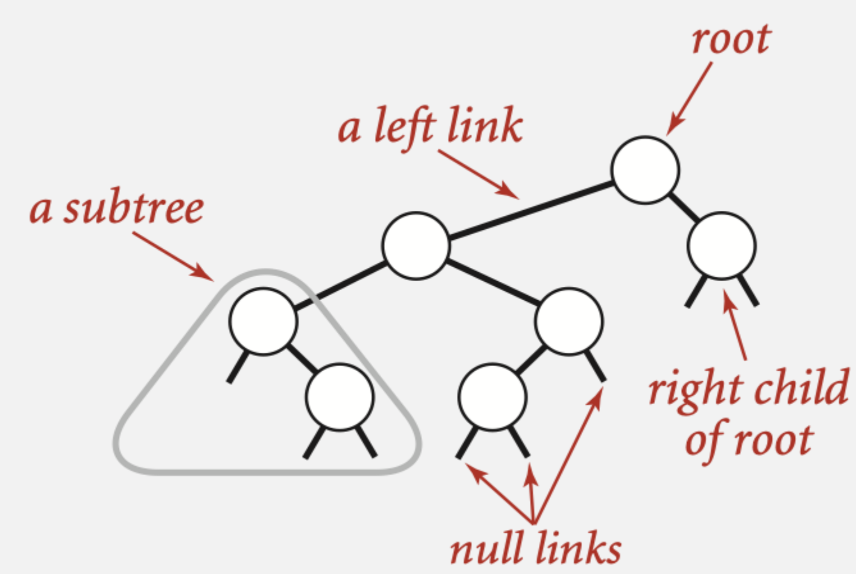
Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

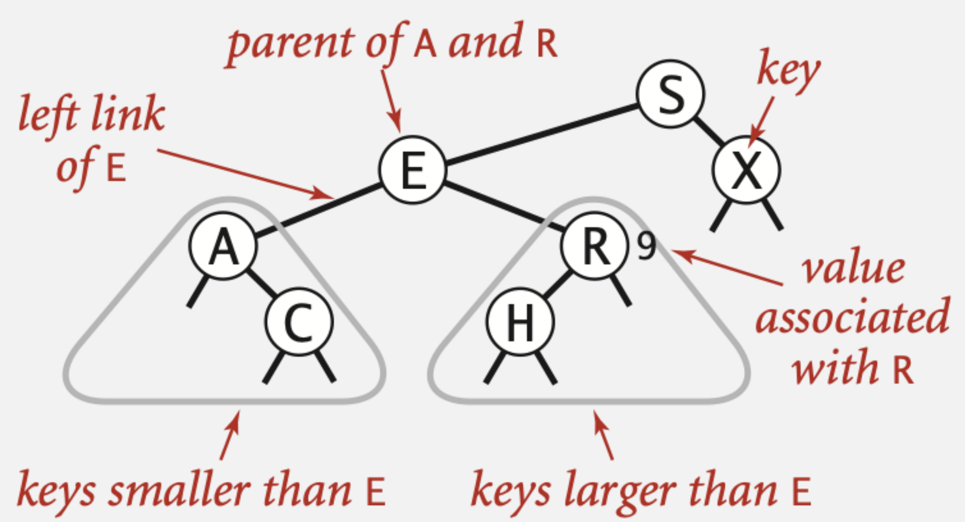
Binary search trees

Definition. A BST is a binary tree in symmetric order.

A binary tree is either:
- Empty.
- Two disjoint binary trees (left and right).



Symmetric order. Each node has a key, and every node's key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



BST representation in Java

Java definition. A BST is a reference to a root Node.

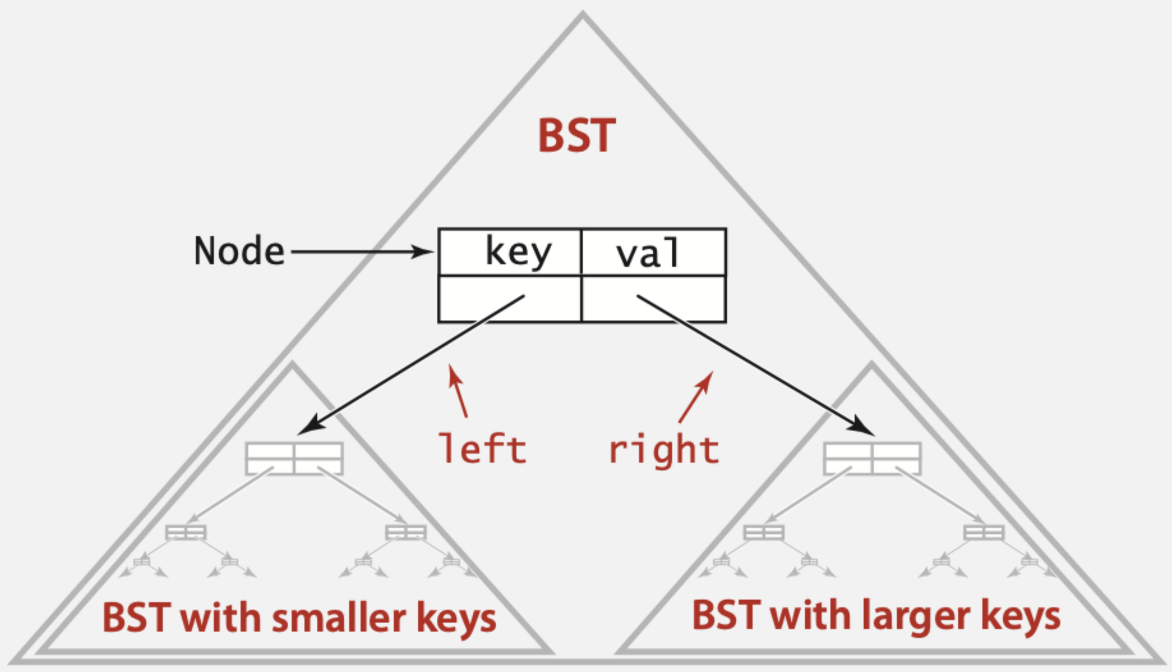
A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

smaller keys   larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

Key and Value are generic types; Key is Comparable



Binary search tree

BST implementation (skeleton)

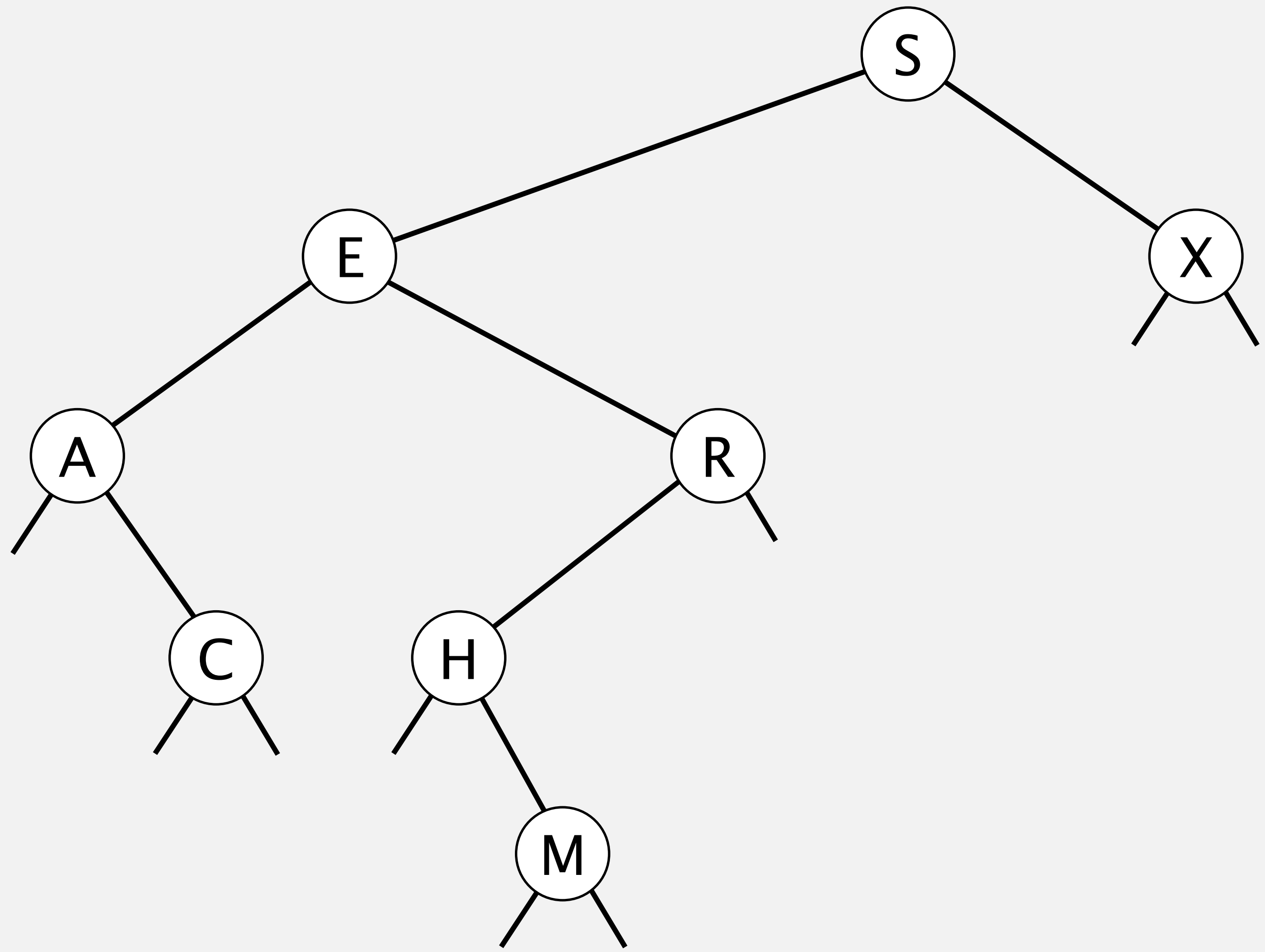
```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;
   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

root of BST

Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

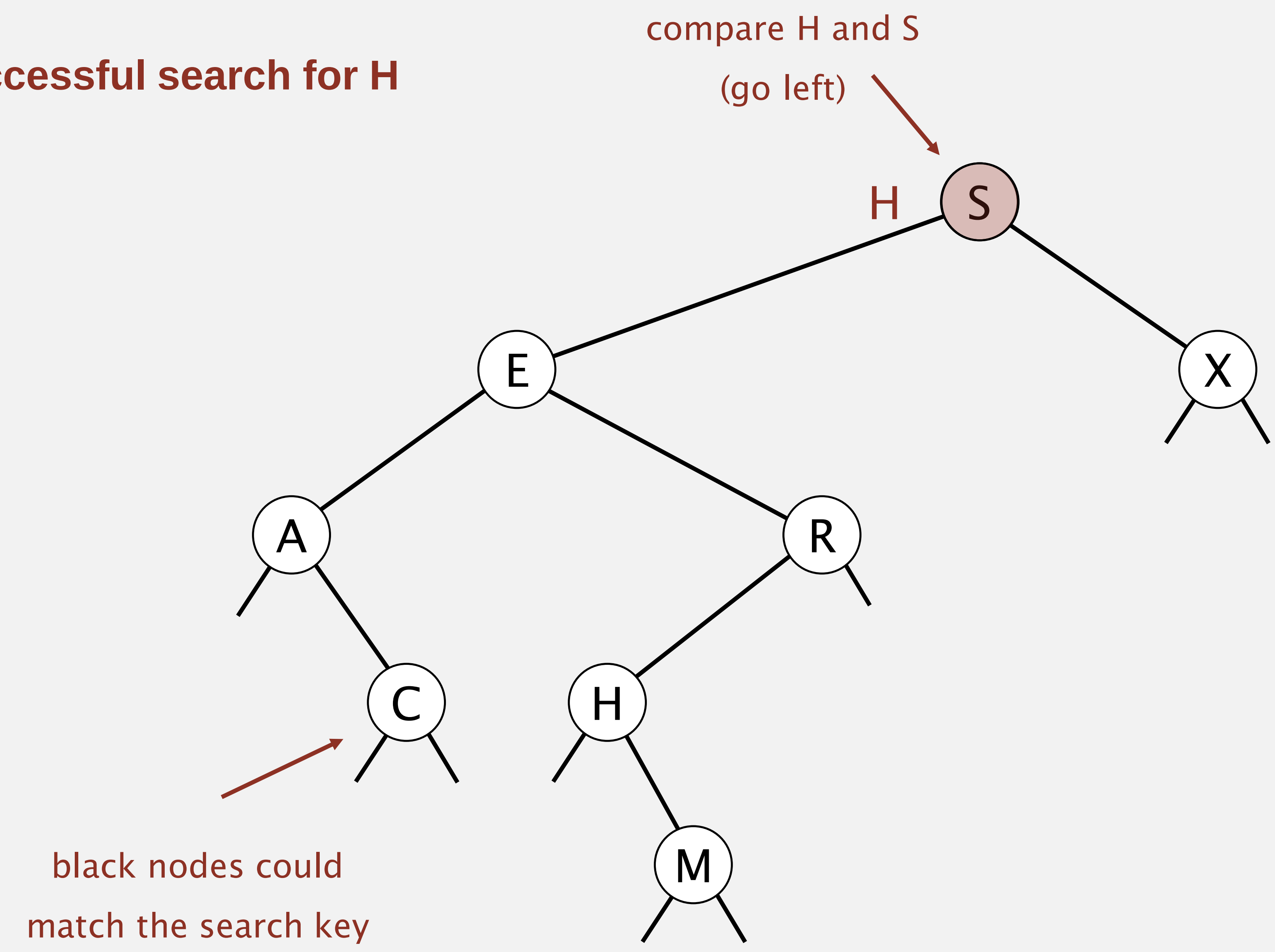
successful search for H



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

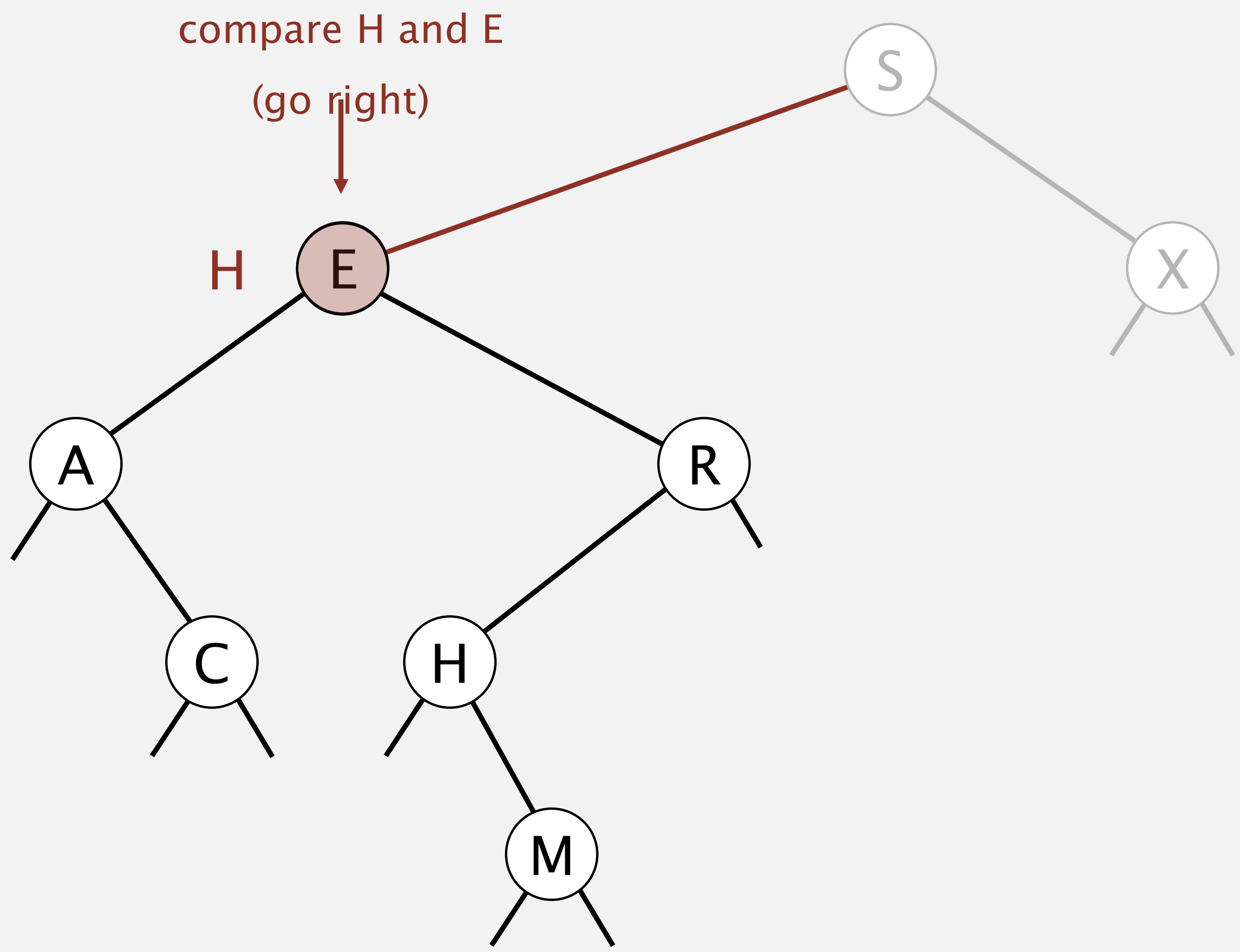
successful search for H



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

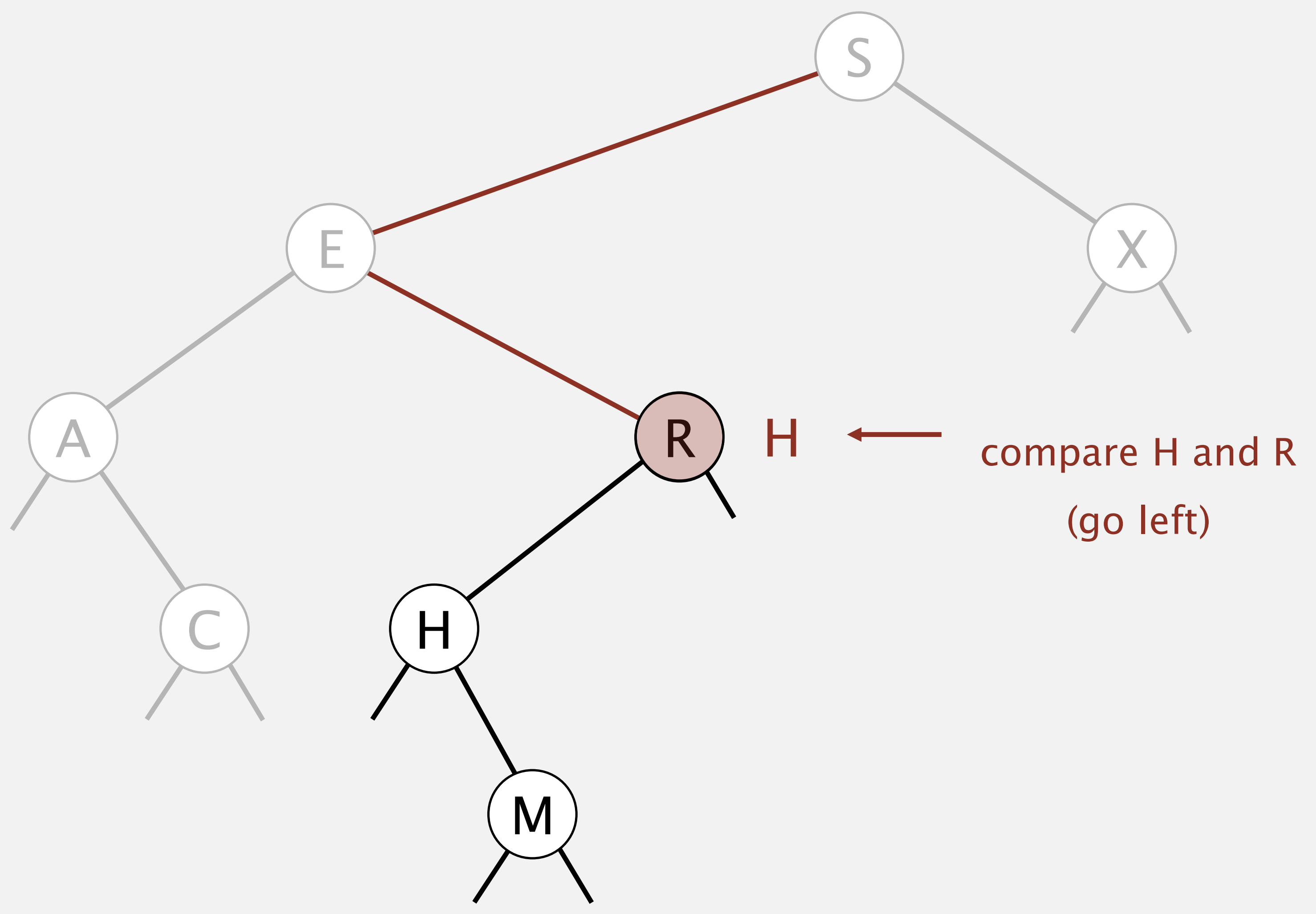
successful search for H



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

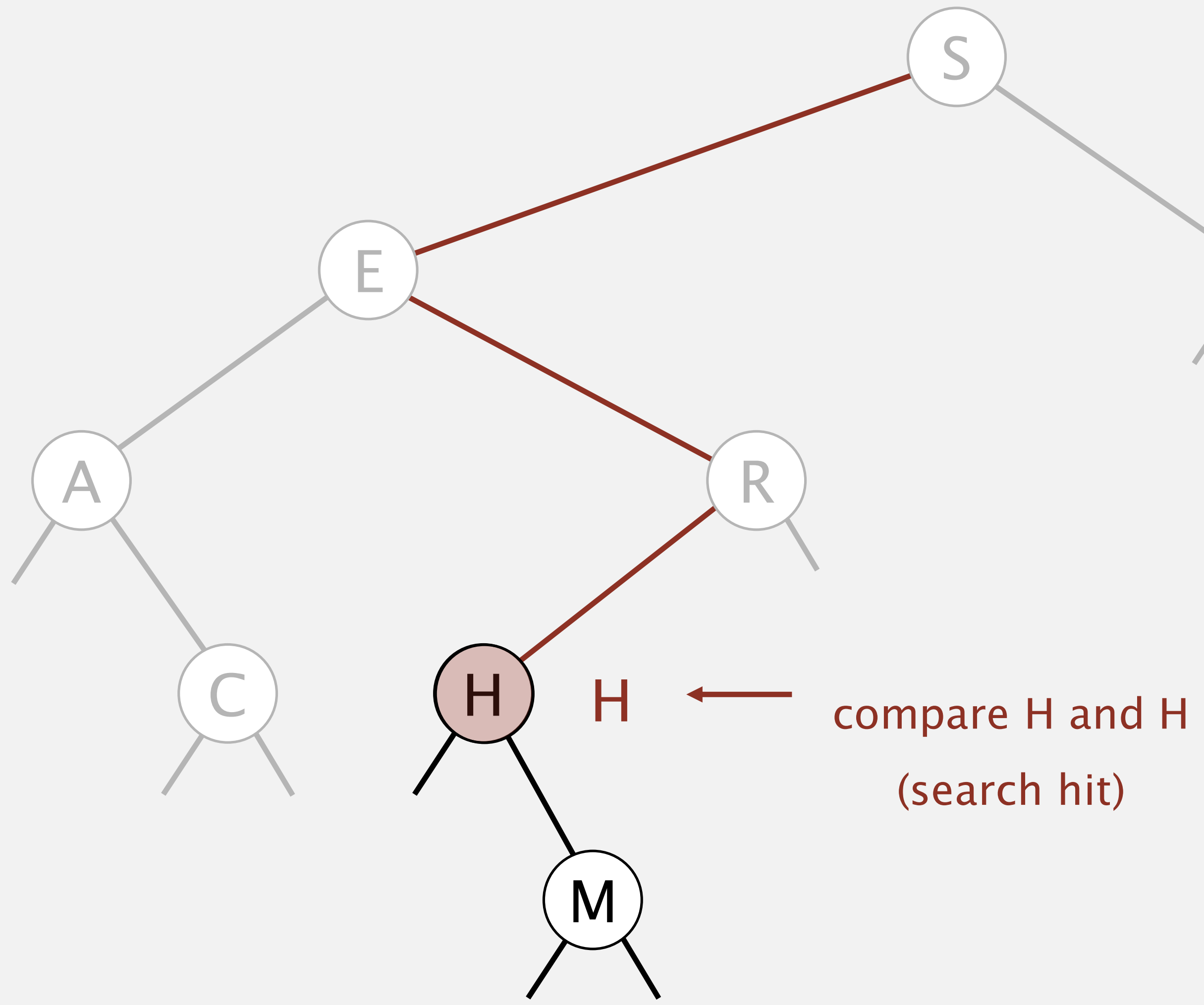
successful search for H



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

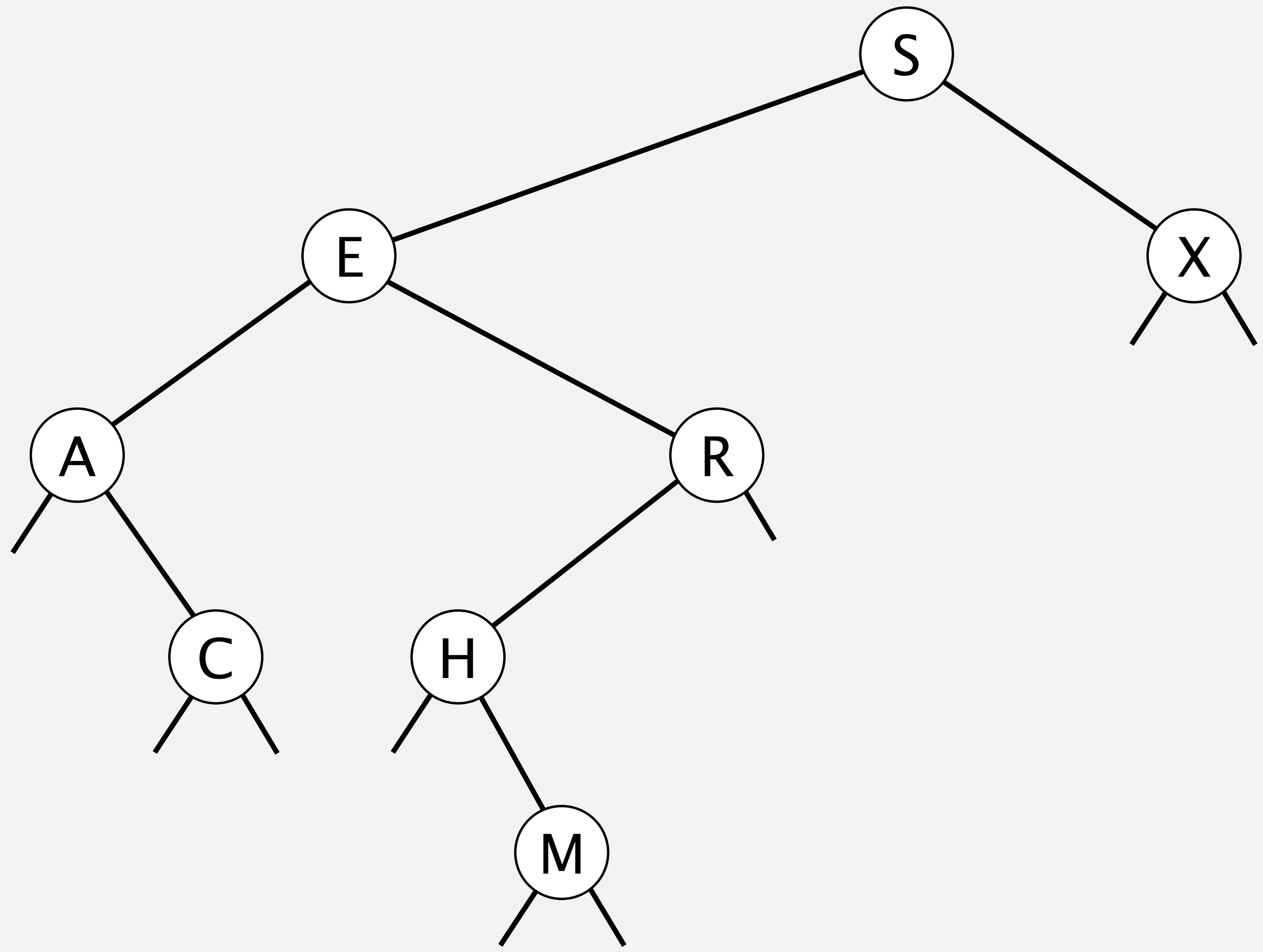
successful search for H



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

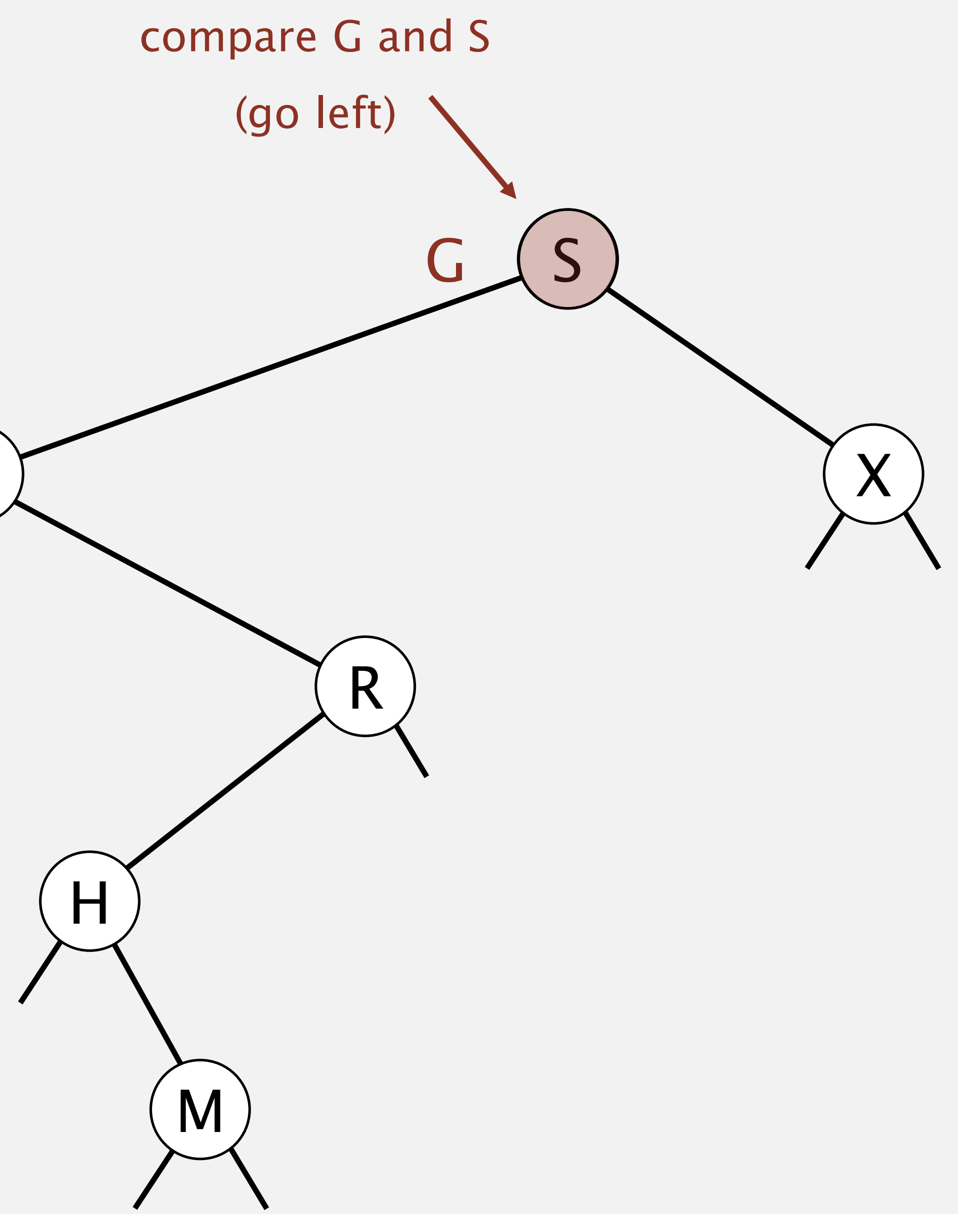
unsuccessful search for G



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

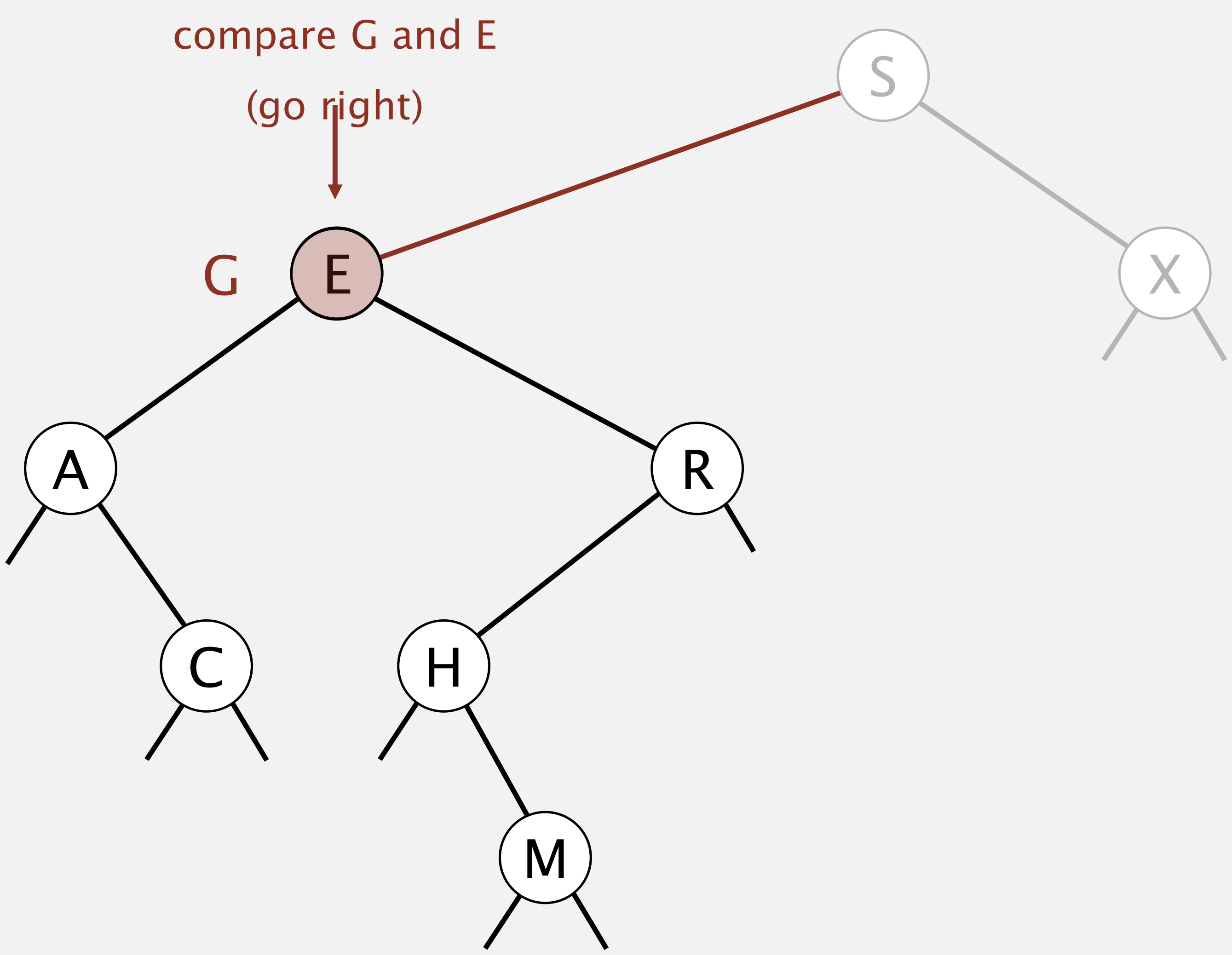
unsuccessful search for G



Binary search tree demo

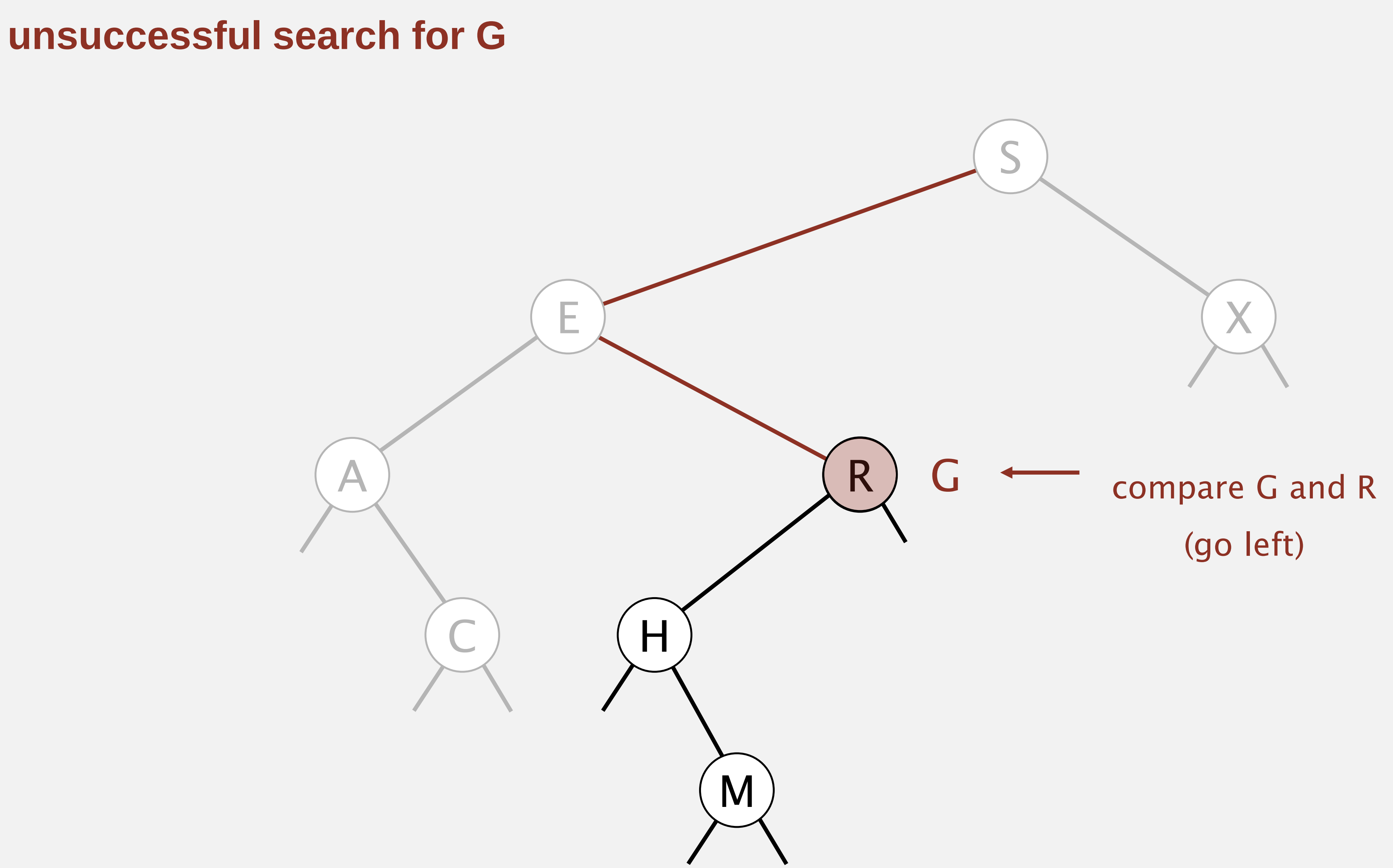
Search. If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G



Binary search tree demo

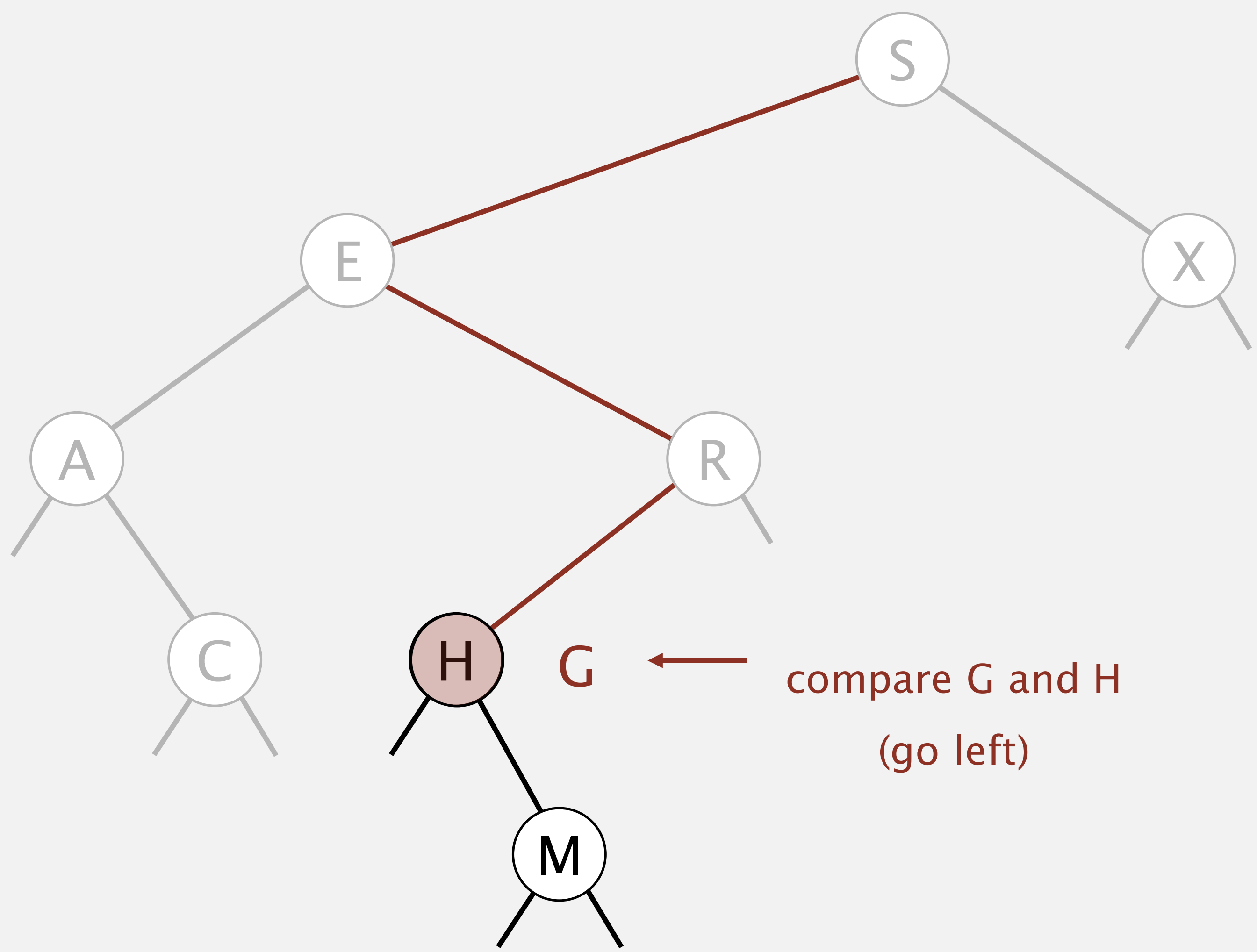
Search. If less, go left; if greater, go right; if equal, search hit.



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

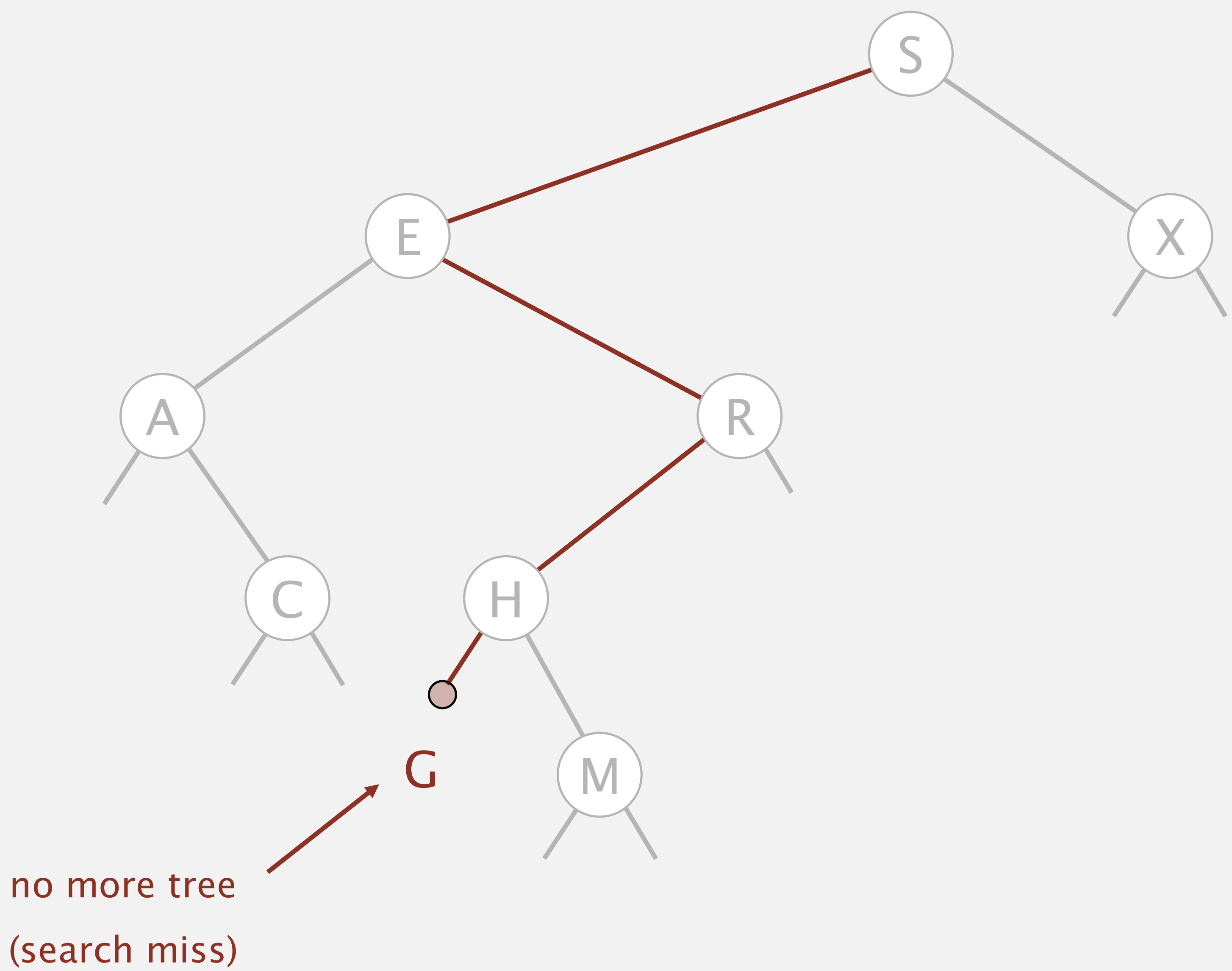
unsuccessful search for G



Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

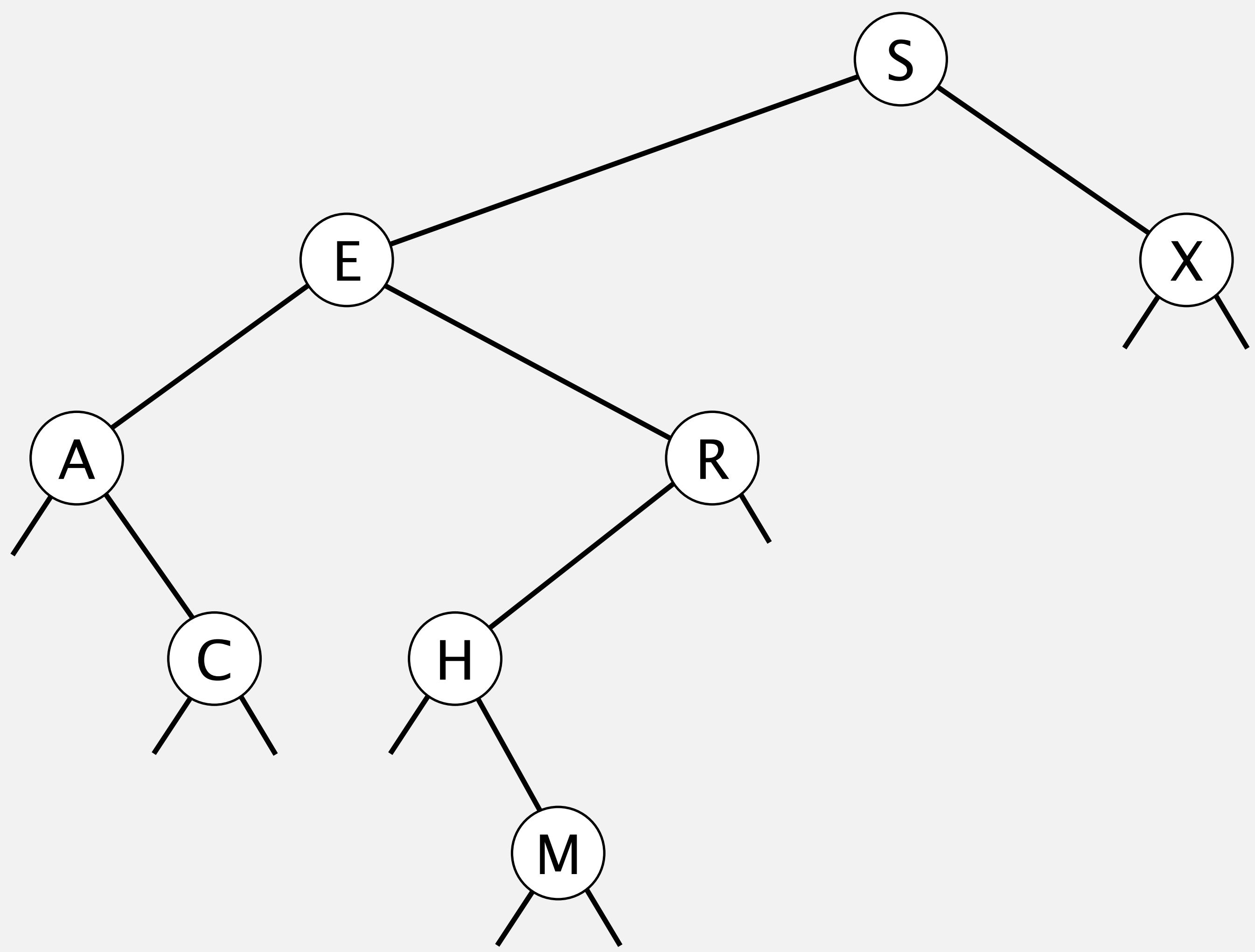
unsuccessful search for G



Binary search tree demo

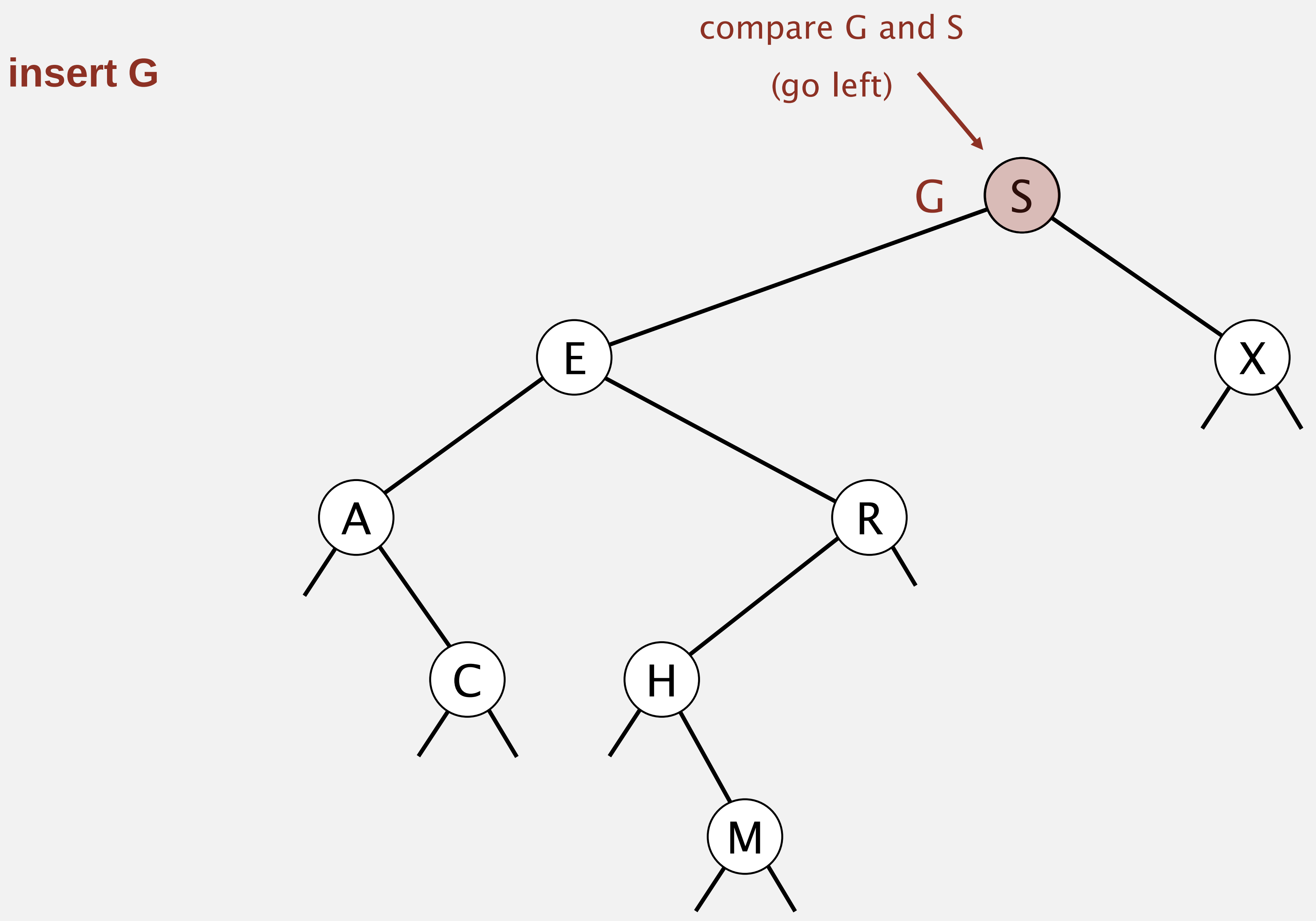
Insert. If less, go left; if greater, go right; if null, insert.

insert G



Binary search tree demo

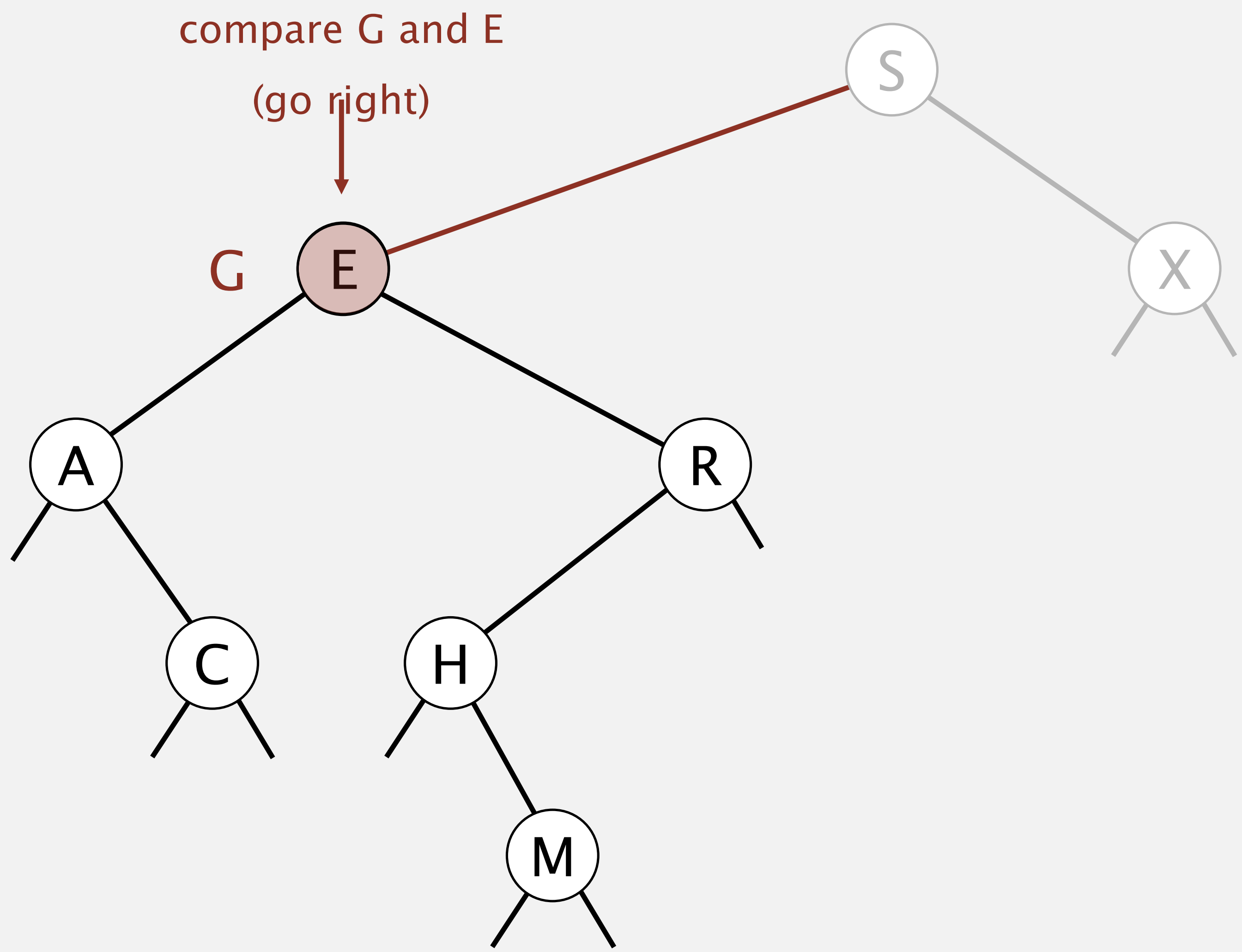
Insert. If less, go left; if greater, go right; if null, insert.



Binary search tree demo

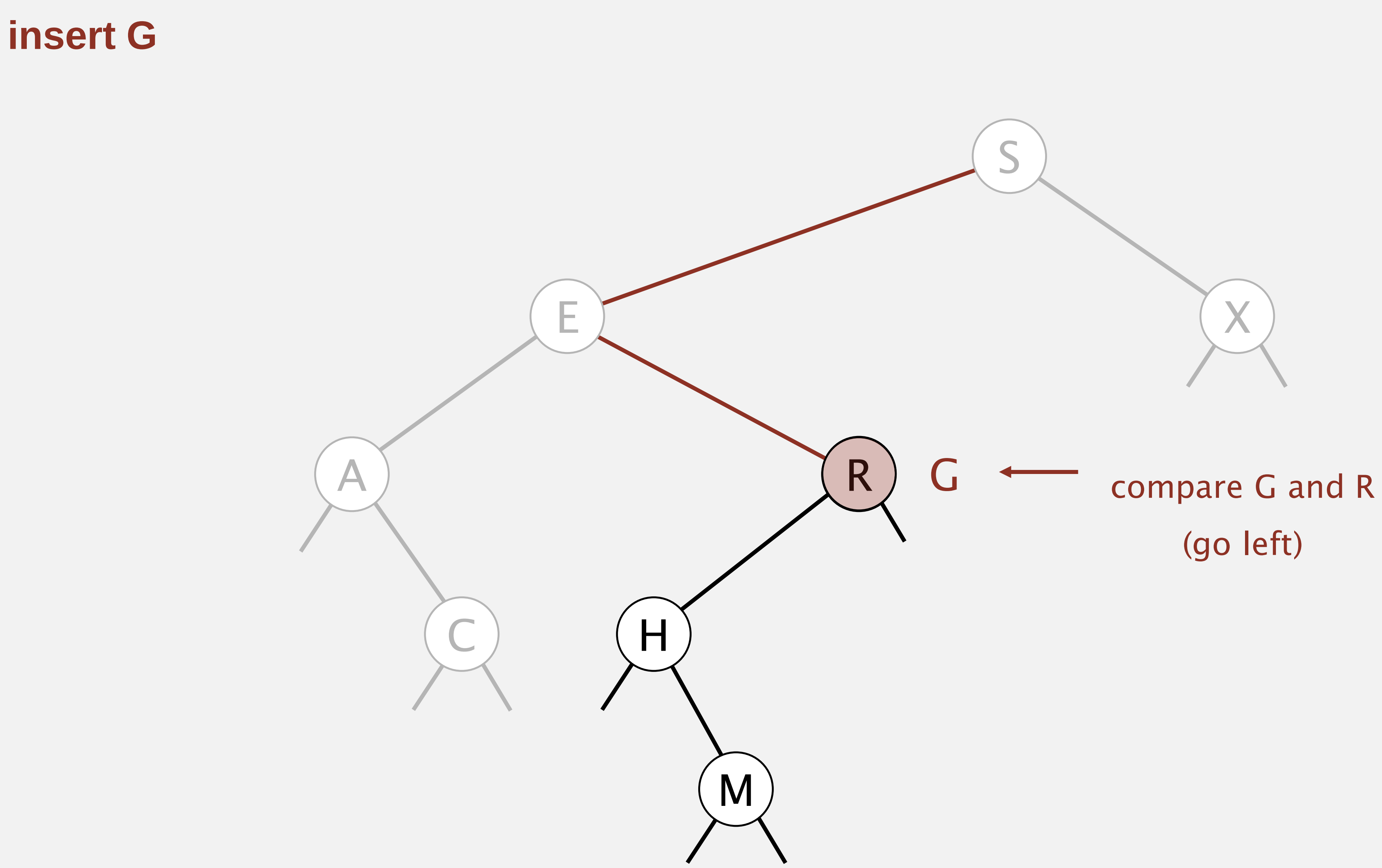
Insert. If less, go left; if greater, go right; if null, insert.

insert G



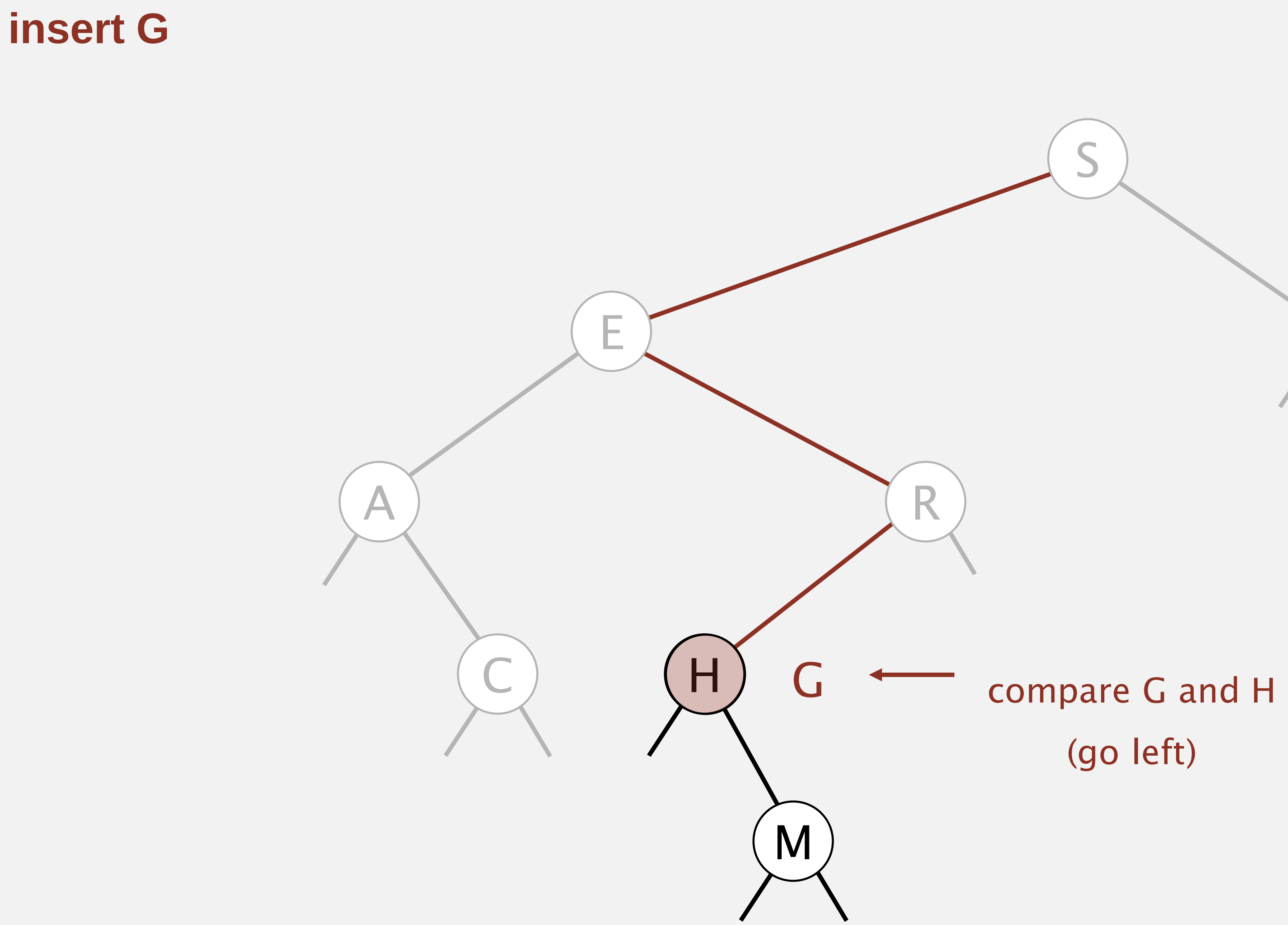
Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.



Binary search tree demo

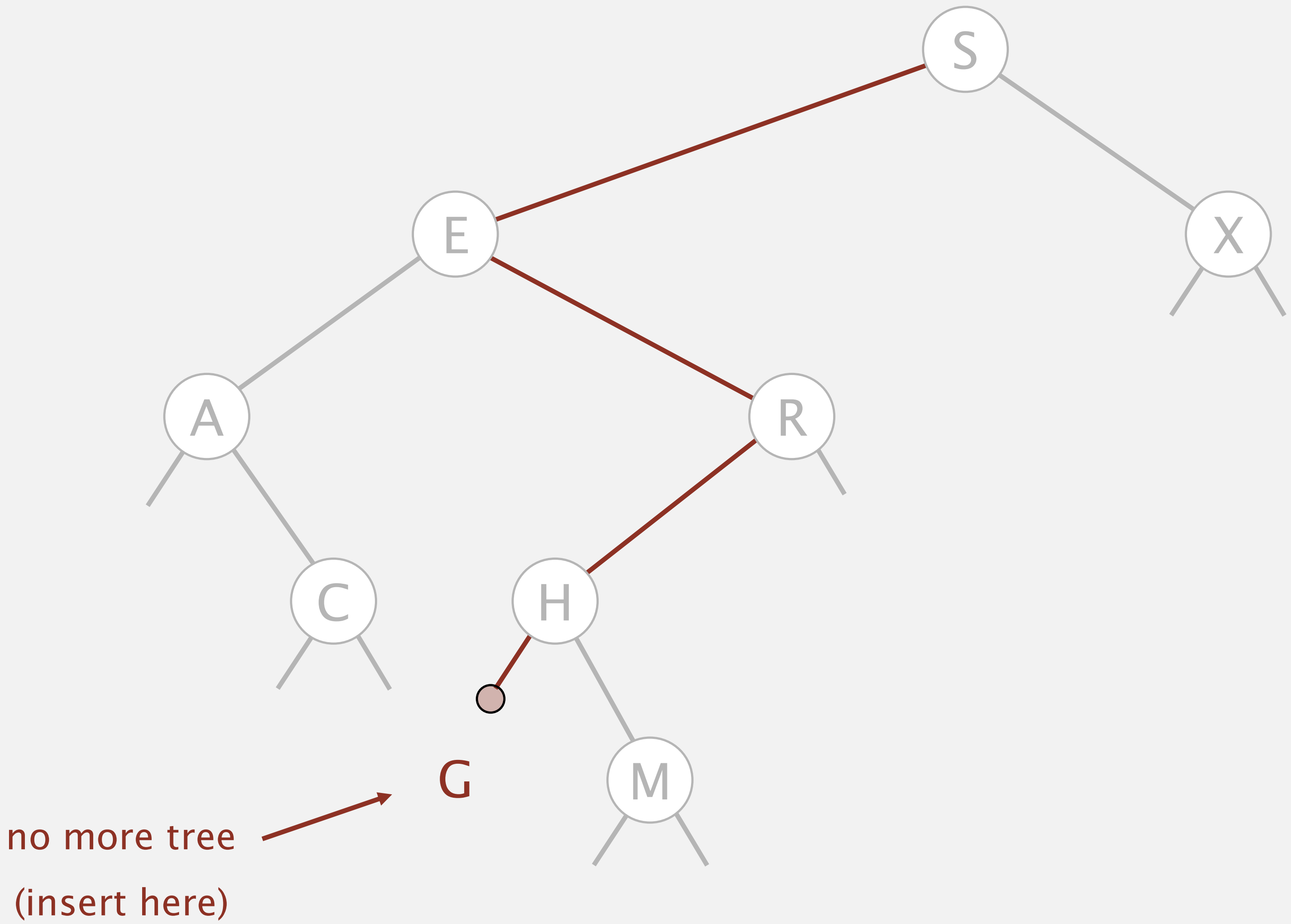
Insert. If less, go left; if greater, go right; if null, insert.



Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

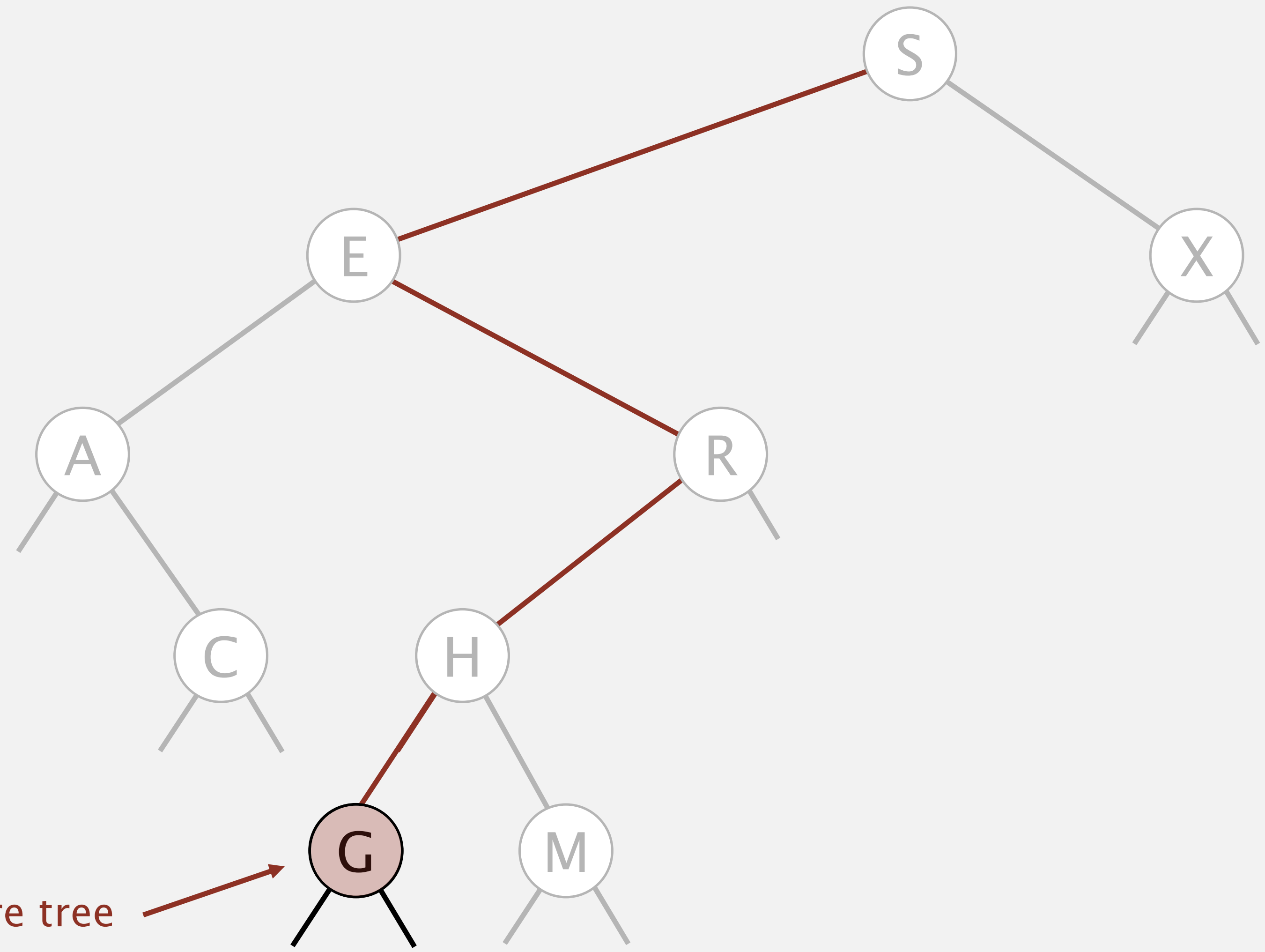
insert G



Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

insert G

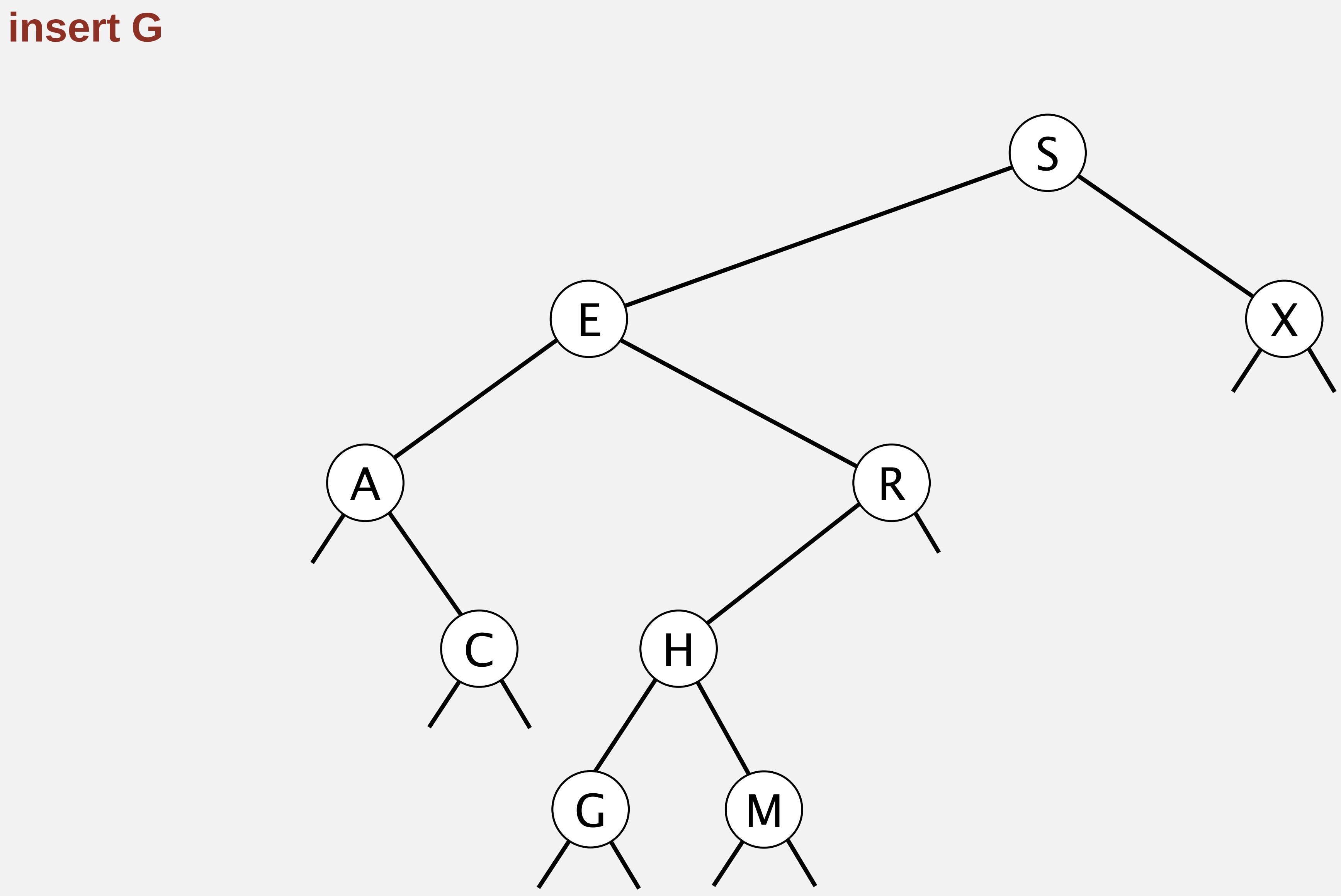


no more tree

(insert here)

Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.



BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

Cost. Number of compares is equal to 1 + depth of node.

BST insert: Java implementation

Put. Associate value with key.

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0)
      x.left  = put(x.left,  key, val);
   else if (cmp  > 0)
      x.right = put(x.right, key, val);
   else if (cmp == 0)
      x.val = val;
   return x;
}
```

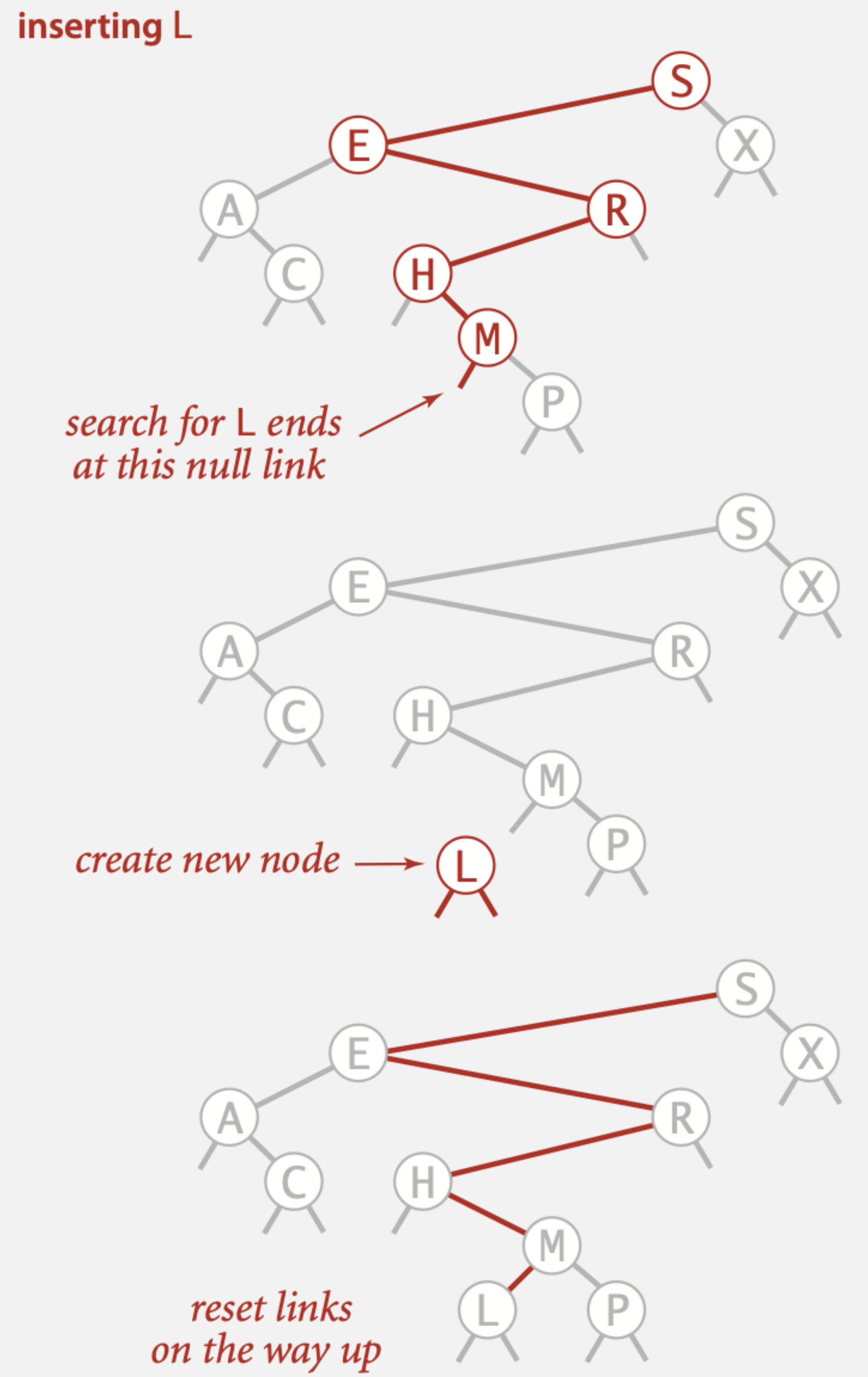
Cost. Number of compares is equal to 1 + depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

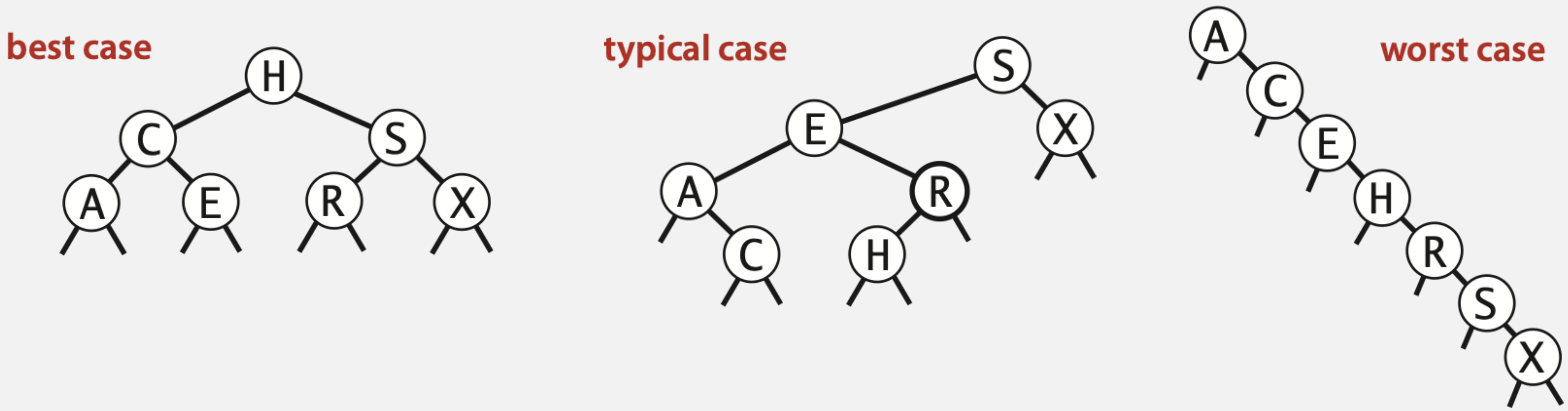
- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.



Insertion into a BST

Tree shape

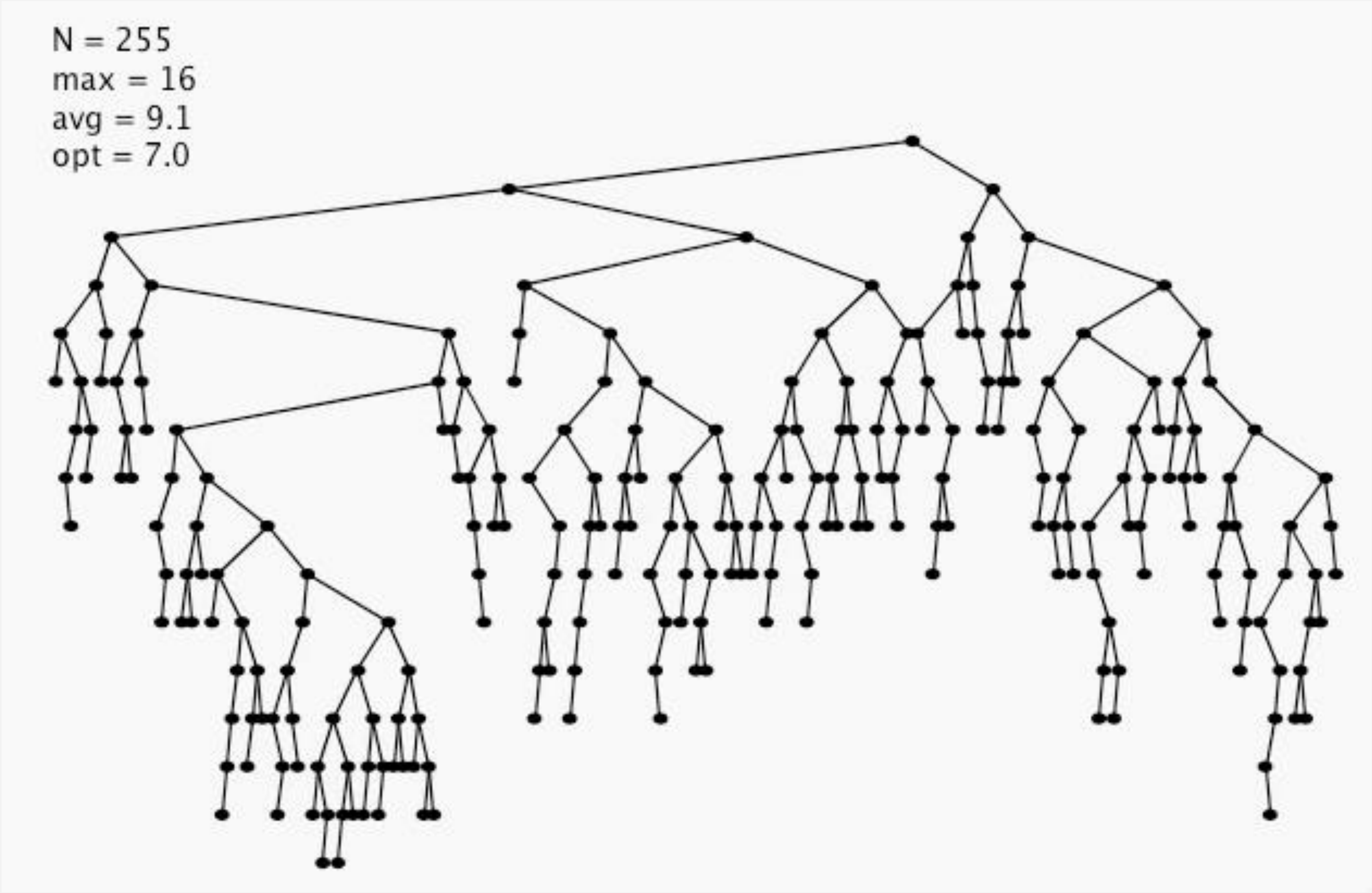
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.



ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	½ N	N	`equals()`
binary search (ordered array)	$\lg N$	N	$\lg N$	½ N	`compareTo()`
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	`compareTo()`

http://algs4.cs.princeton.edu

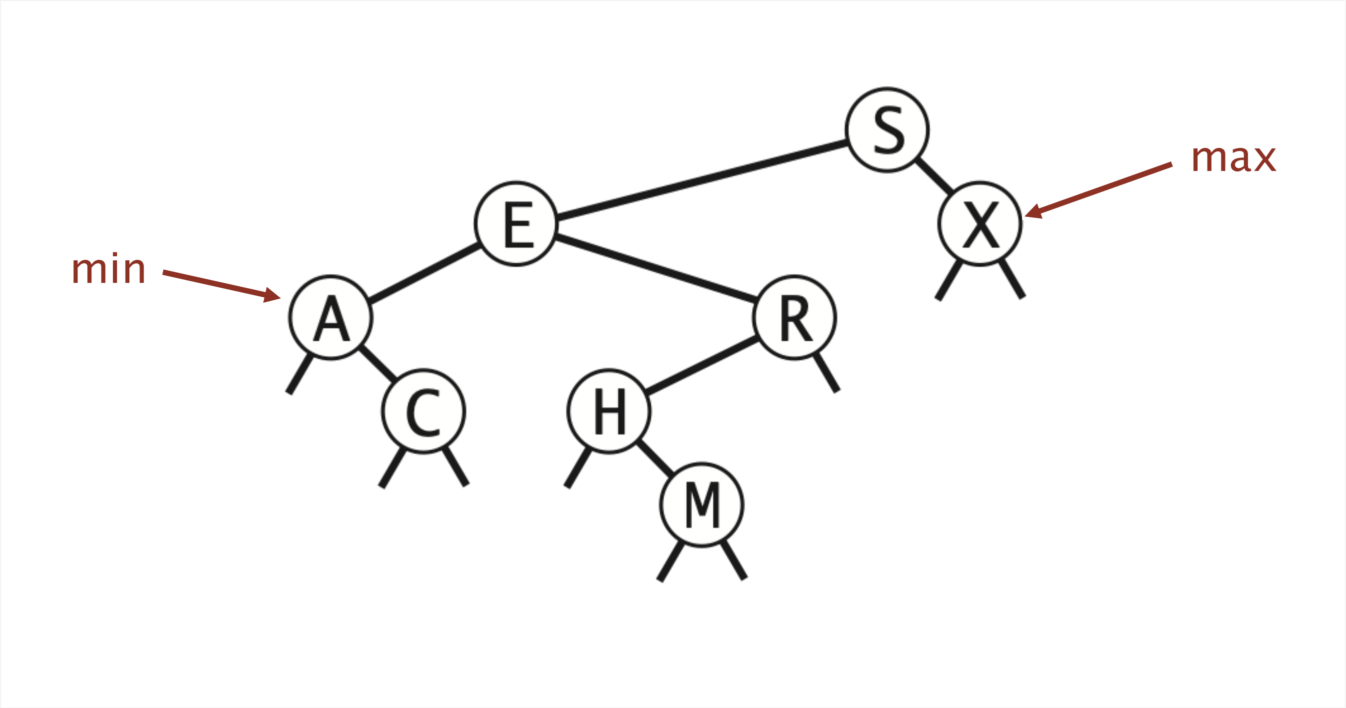
BINARY SEARCH TREES

- *BSTs*
- *ordered operations*
- *deletion*

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

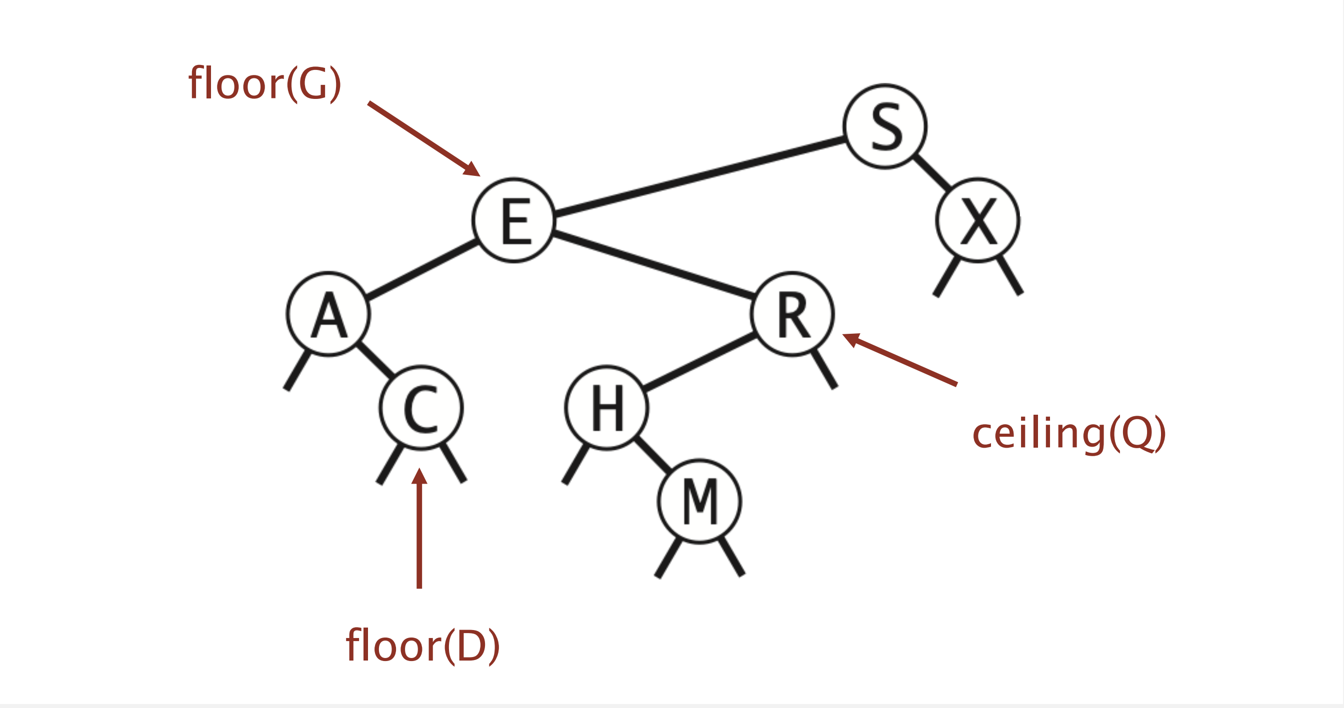


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key ≤ a given key.

Ceiling. Smallest key ≥ a given key.



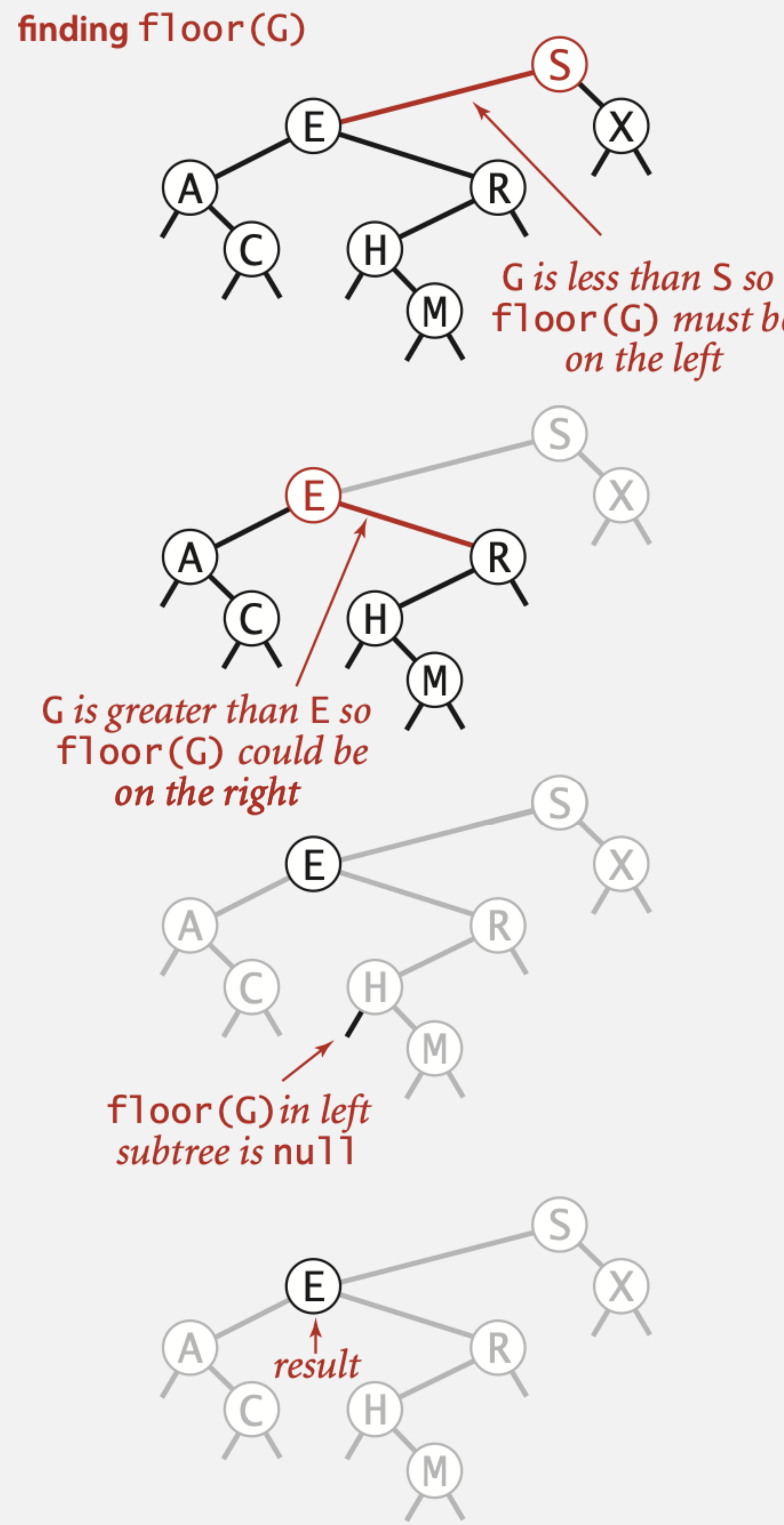
Q. How to find the floor / ceiling?

Computing the floor

Case 1. [k equals the key in the node]
The floor of k is k.

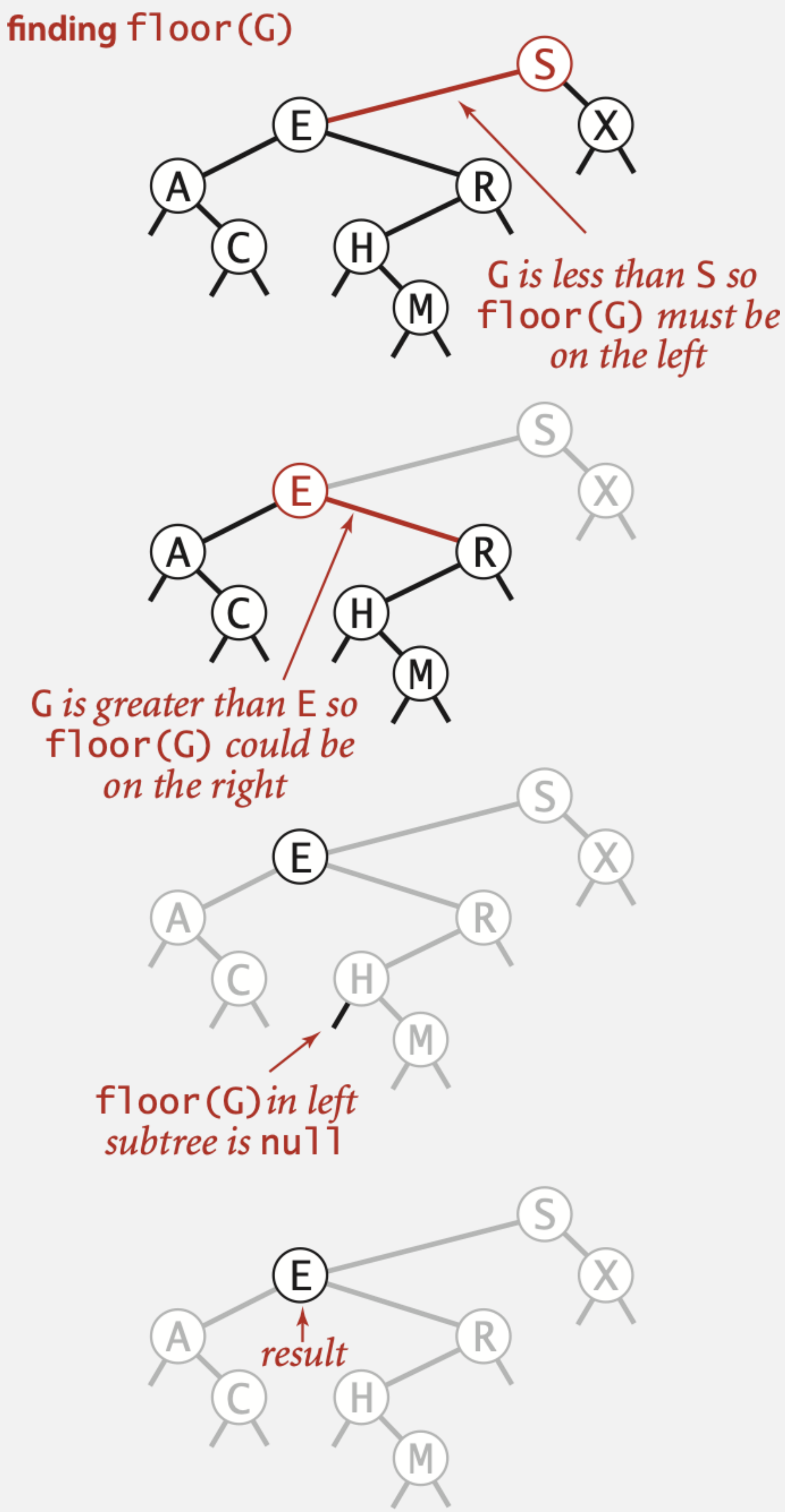
Case 2. [k is less than the key in the node]
The floor of k is in the left subtree.

Case 3. [k is greater than the key in the node]
The floor of k is in the right subtree
(if there is any key $\leq k$ in right subtree);
otherwise it is the key in the node.



Computing the floor

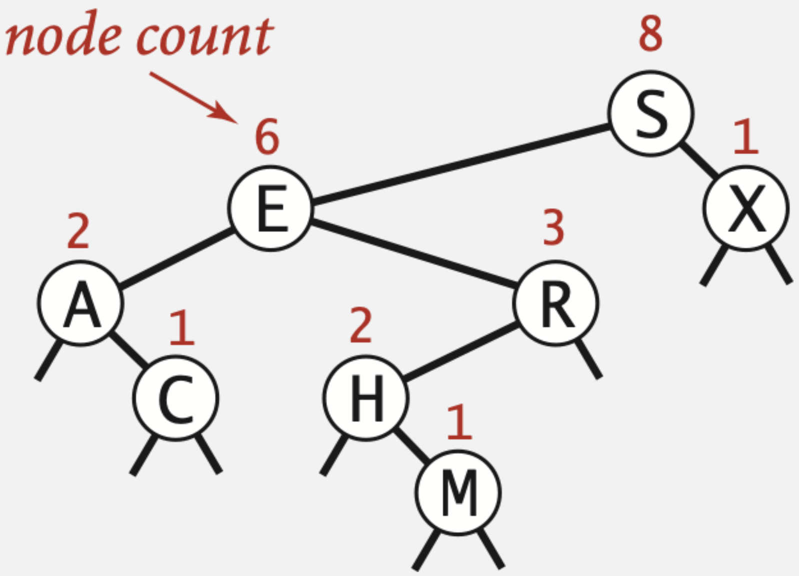
```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```



Rank and select

Q. How to implement `size()`, `rank()` and `select()` efficiently?

A. In each node, we store the number of nodes in the subtree rooted at that node; to implement `size()`, return the count at the root.



BST implementation: subtree counts

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int count;
}
```

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

number of nodes in subtree

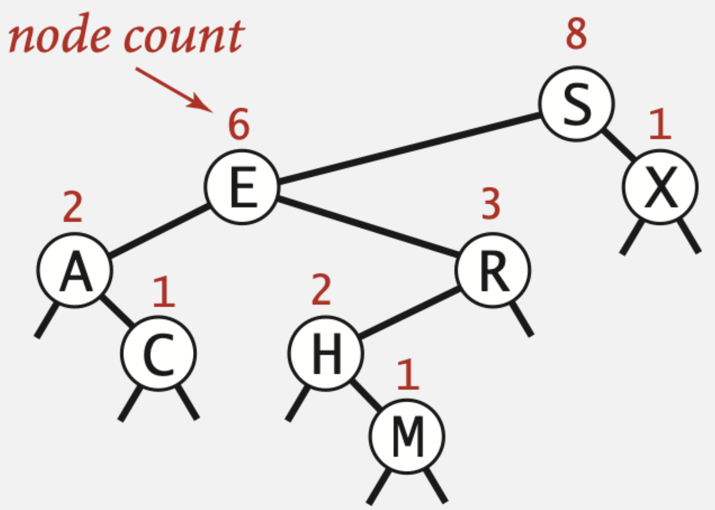
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

initialize subtree count to 1

Rank

Rank. How many keys $< k$?

Easy recursive algorithm (3 cases!)

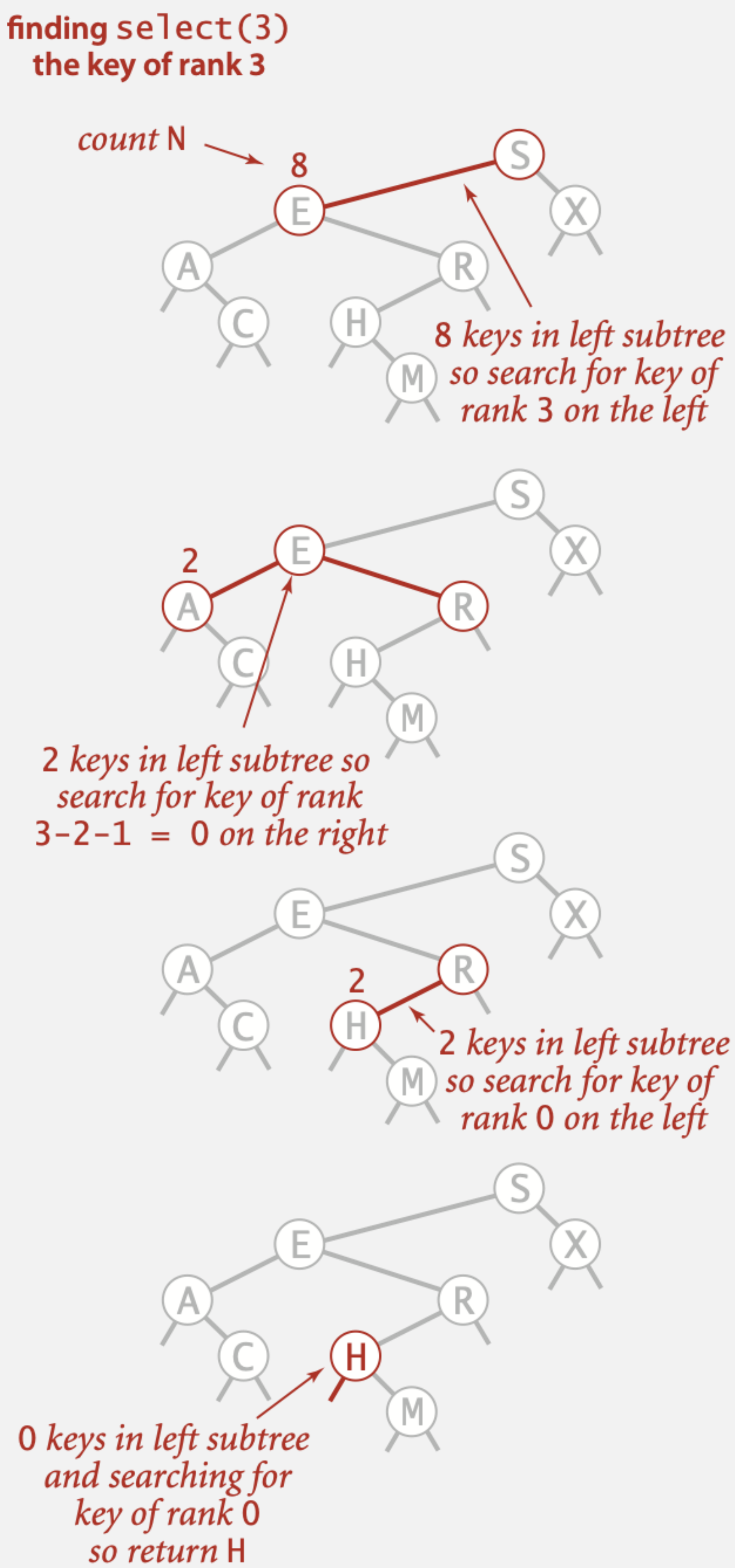



```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

Selection

Select. Key of given rank.

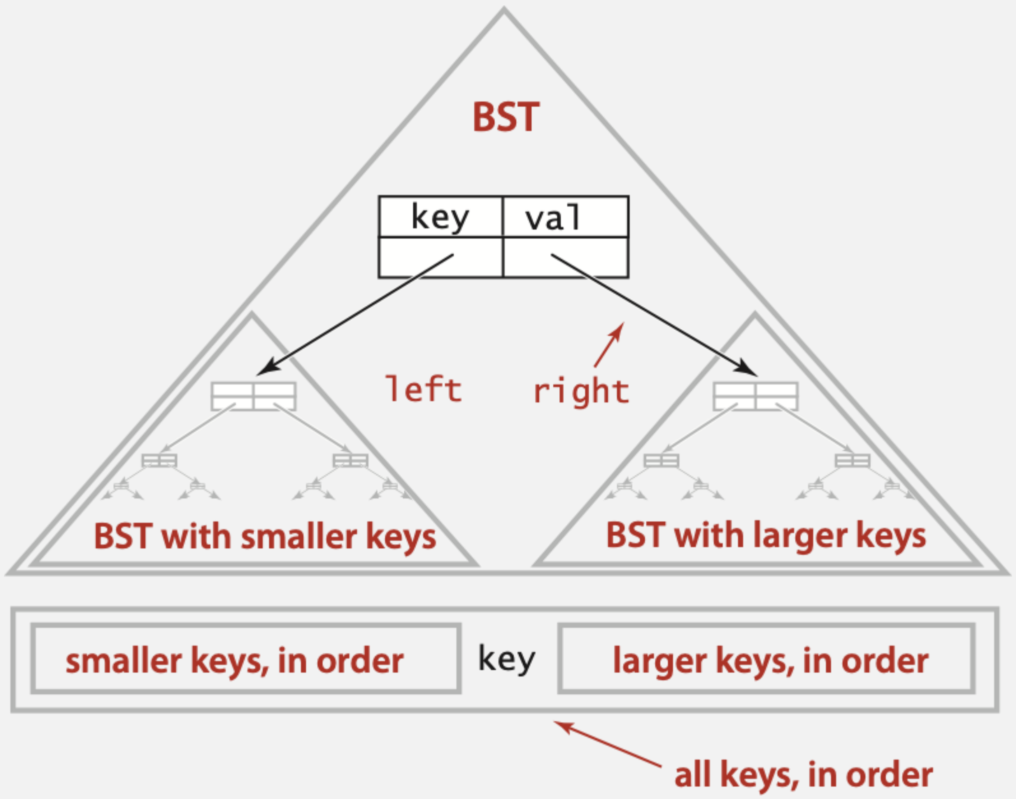
```
public Key select(int k)
{
    if (k < 0) return null;
    if (k >= size()) return null;
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k)
{
   if (x == null) return null;
   int t = size(x.left);
   if       (t  > k)
      return select(x.left,  k);
   else if (t  < k)
      return select(x.right, k-t-1);
   else if (t == k)
      return x;
}
```



Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

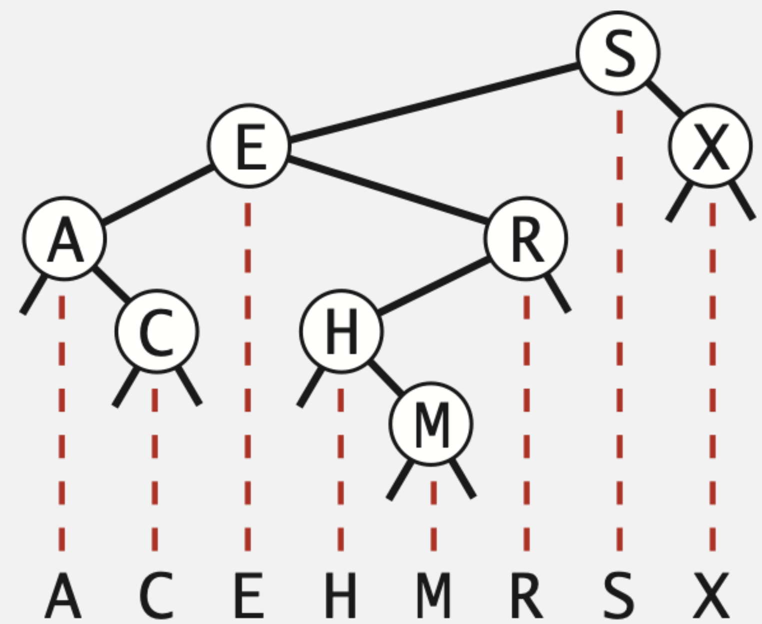


Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

recursive calls	queue	function call stack
inorder(S)		S
inorder(E)		S E
inorder(A)		S E A
enqueue A	A	
inorder(C)		S E A C
enqueue C	C	
enqueue E	E	
inorder(R)		S E R
inorder(H)		S E R H
enqueue H	H	
inorder(M)		S E R H M
enqueue M	M	
enqueue R	R	
enqueue S	S	
inorder(X)		S X
enqueue X	X	



BST: ordered symbol table operations summary

	Sequential search	Binary search	BST
search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to log N
if keys inserted in random order)

order of growth of running time of ordered symbol table operations

http://algs4.cs.princeton.edu

Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

ST implementations: summary

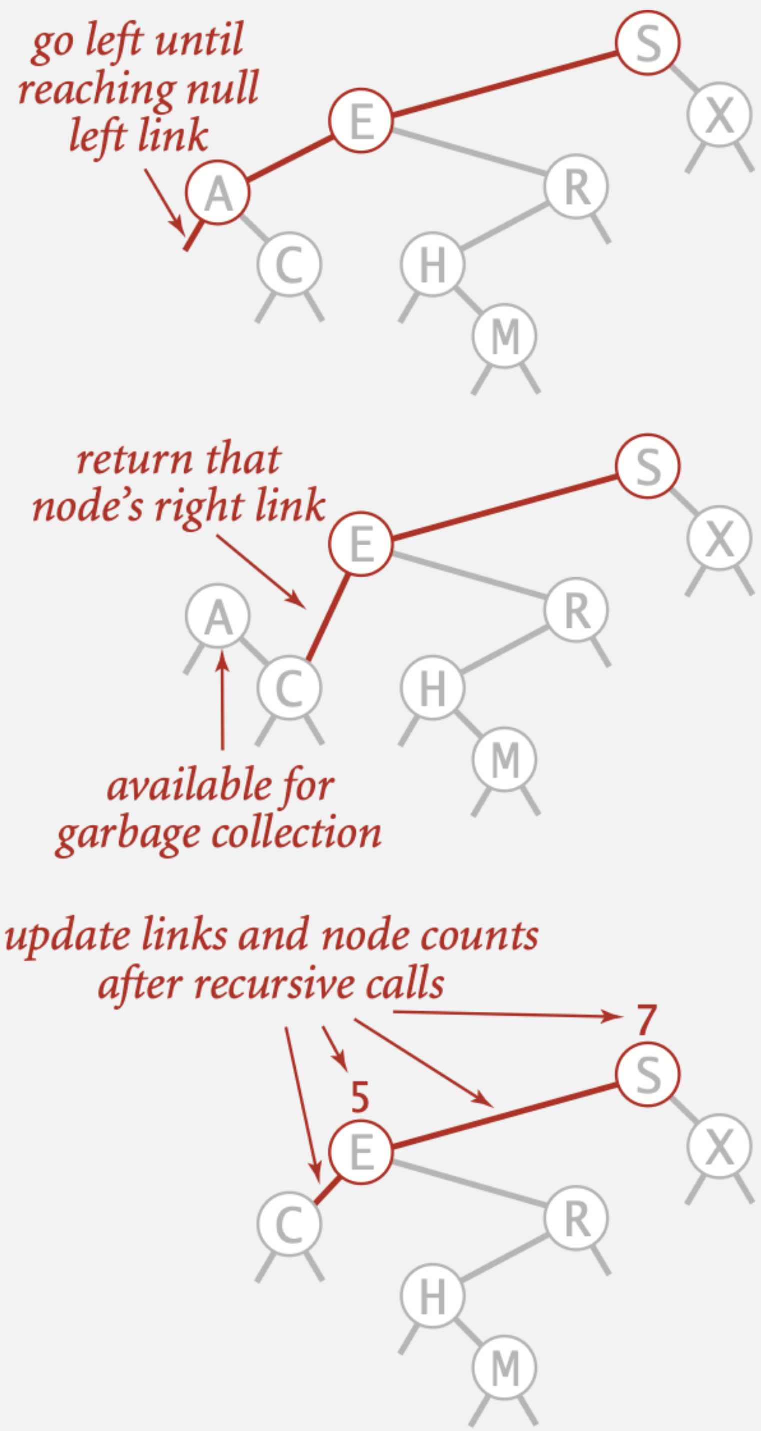
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✓	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	? ? ?	✓	`compareTo()`

Next. Deletion in BSTs.

Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

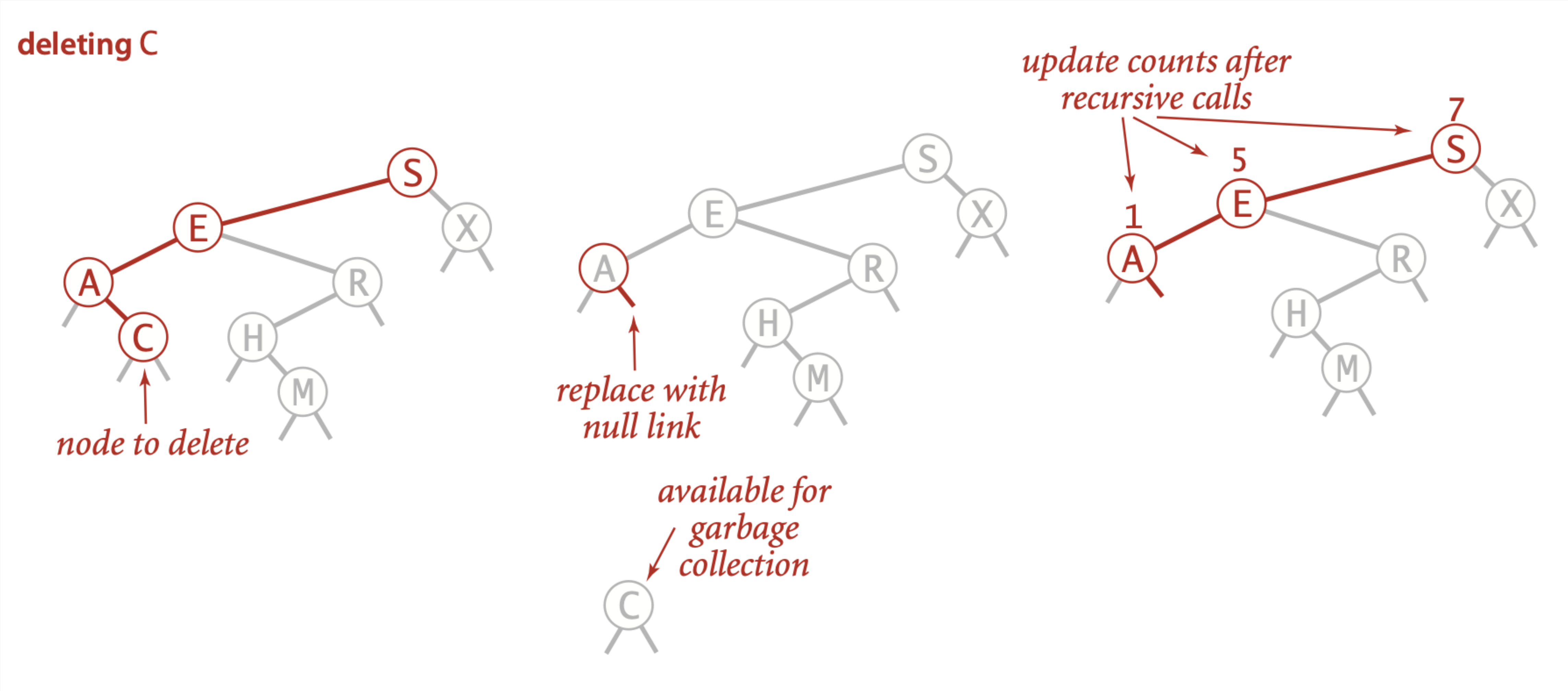



```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

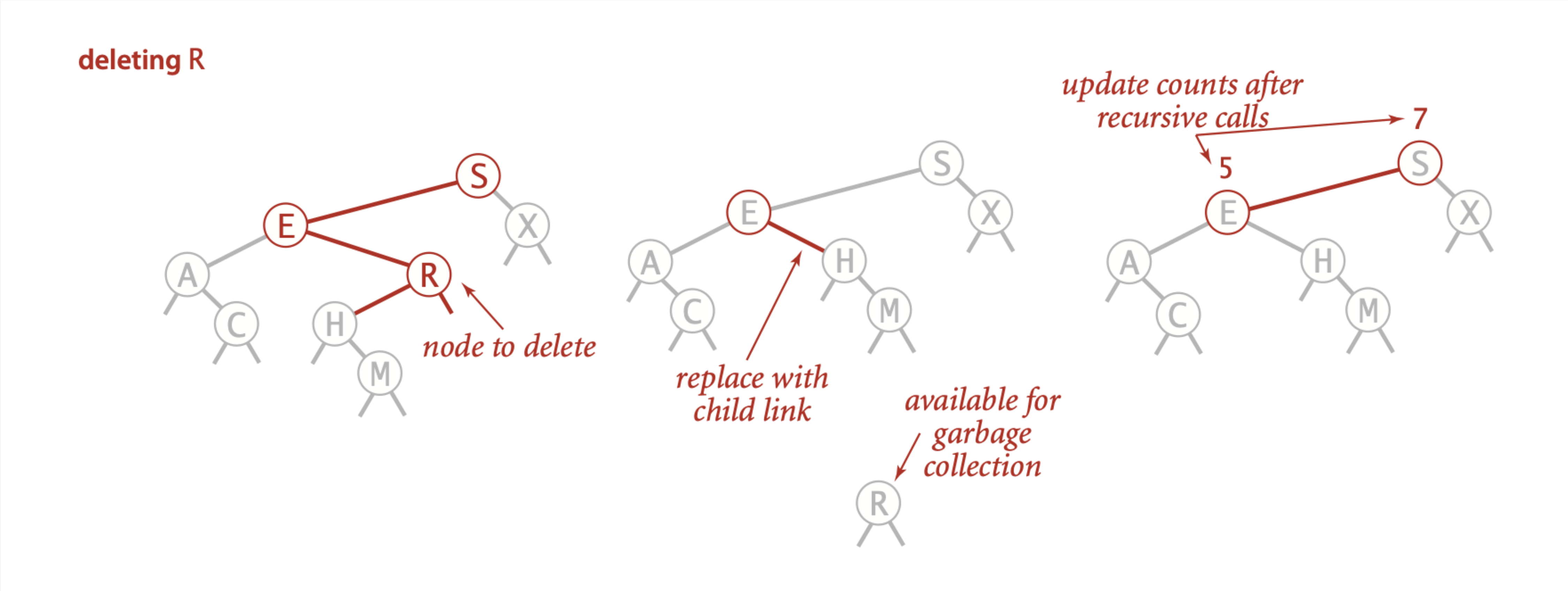
Case 0. [0 children] Delete `t` by setting parent link to null.



Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 1. [1 child] Delete `t` by replacing parent link.

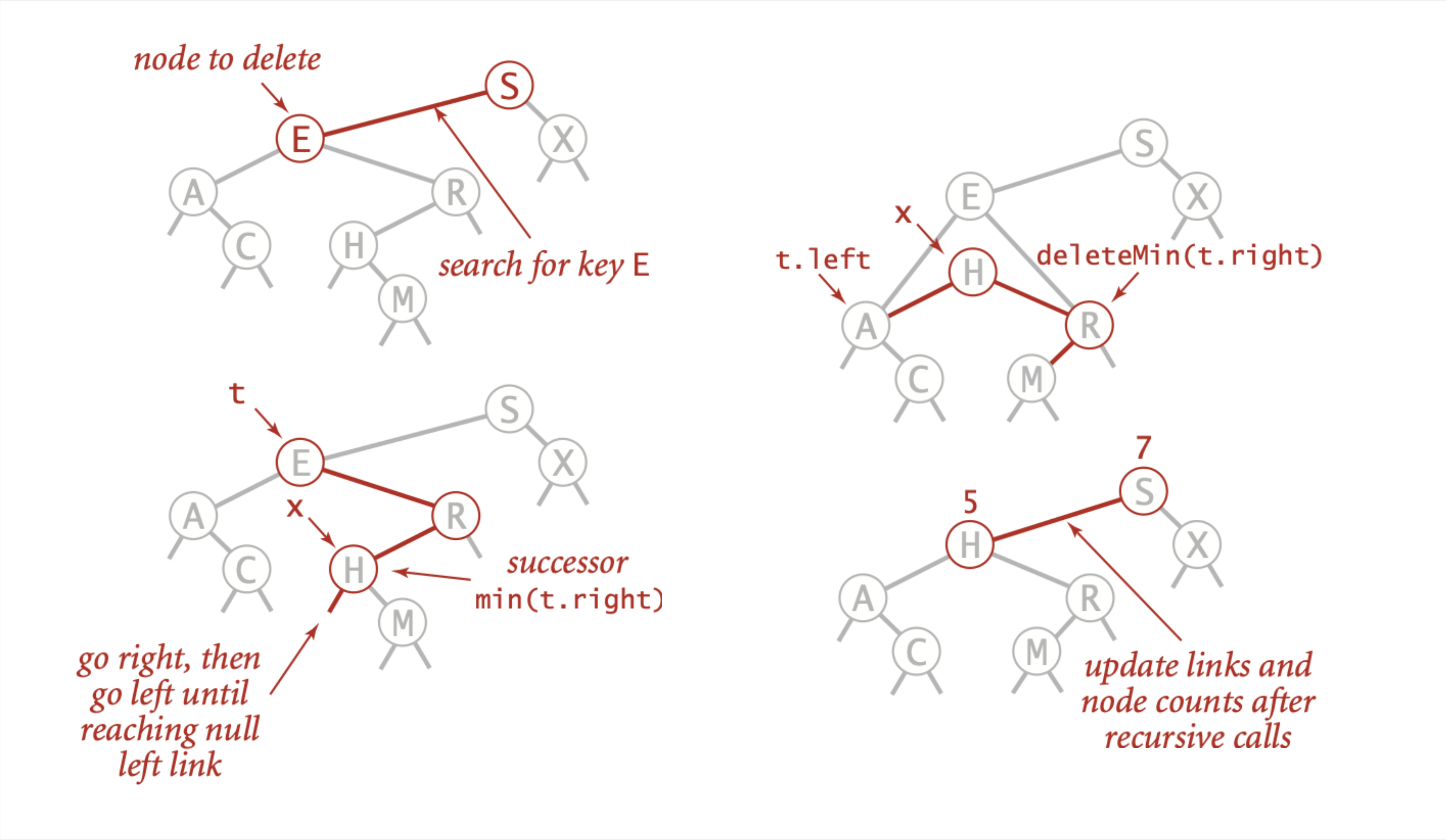


Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children]

- Find successor `x` of `t`. ← x has no left child
- Delete the minimum in `t`'s right subtree. ← but don't garbage collect x
- Put `x` in `t`'s spot. ← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

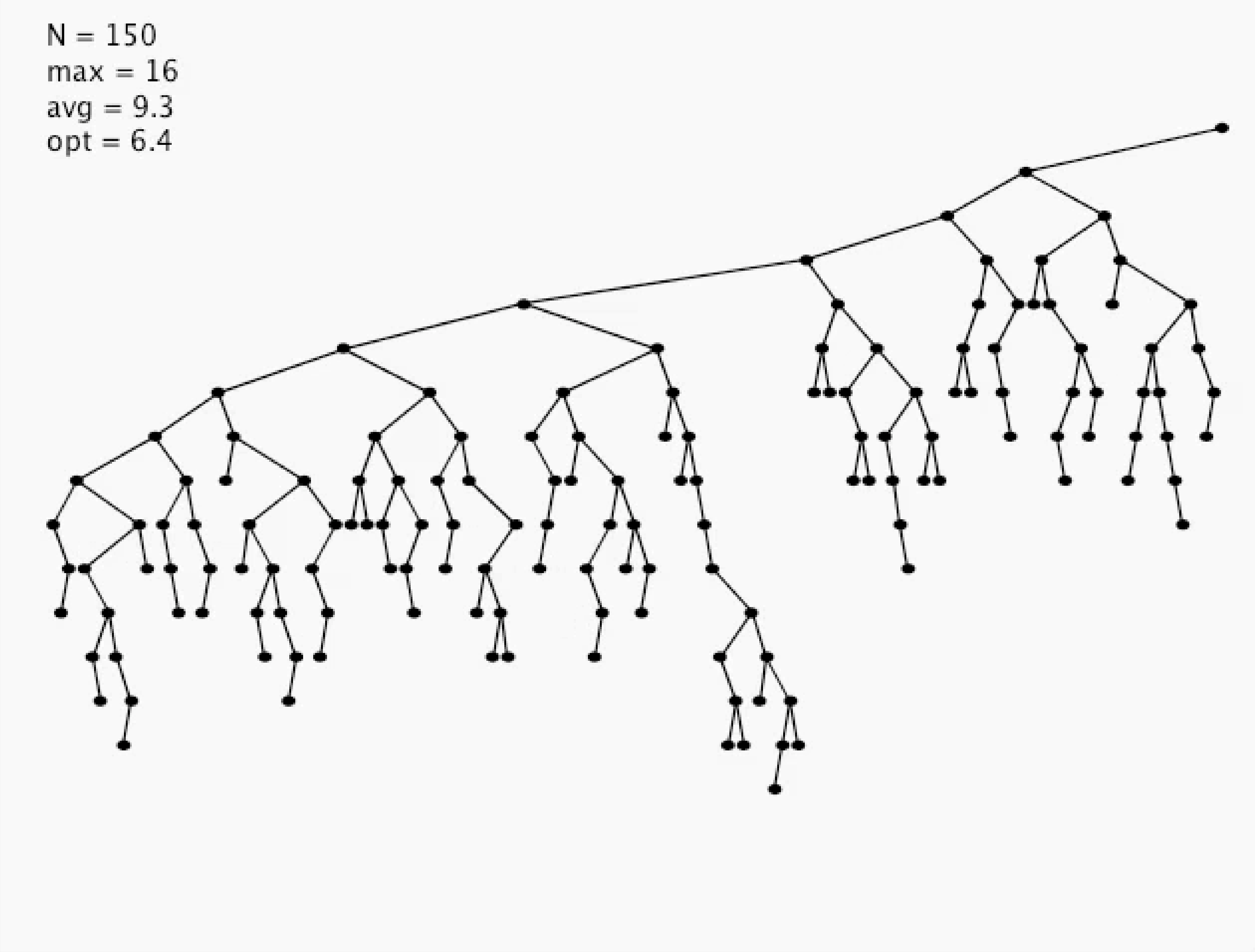
no left child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow$ $\sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✓	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	`compareTo()`

other operations also become √N
if deletions allowed

http://algs4.cs.princeton.edu

Hash Tables

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`

Optional Read: red-black BST, 3.5 in textbook

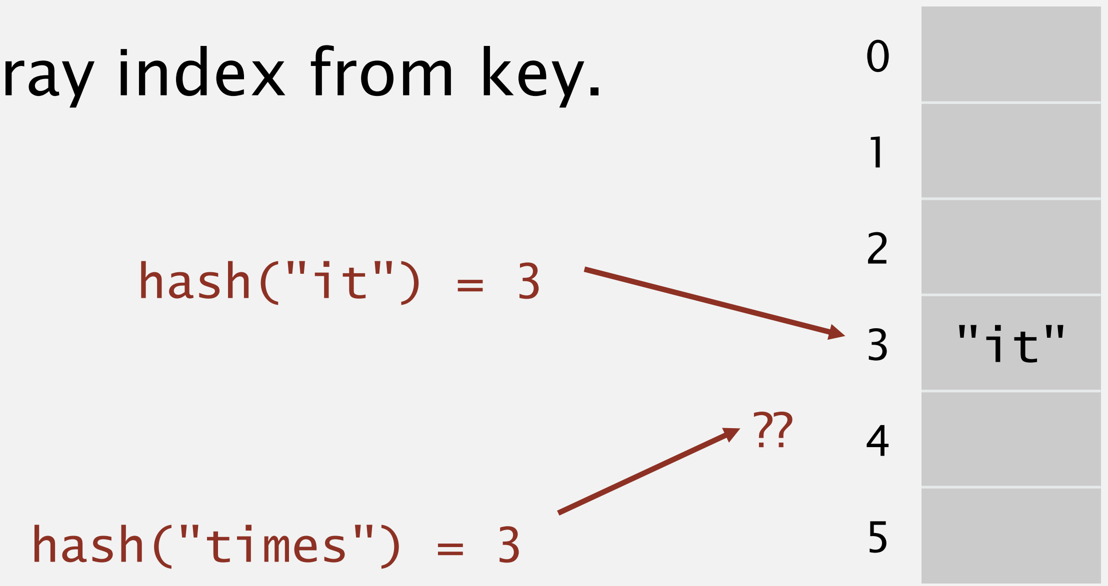
Q. Can we do better?

A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

http://algs4.cs.princeton.edu

Hash Tables

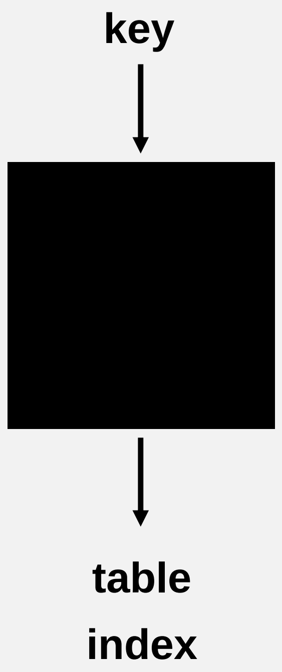
- *hash functions*
- *separate chaining*
- *linear probing*

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

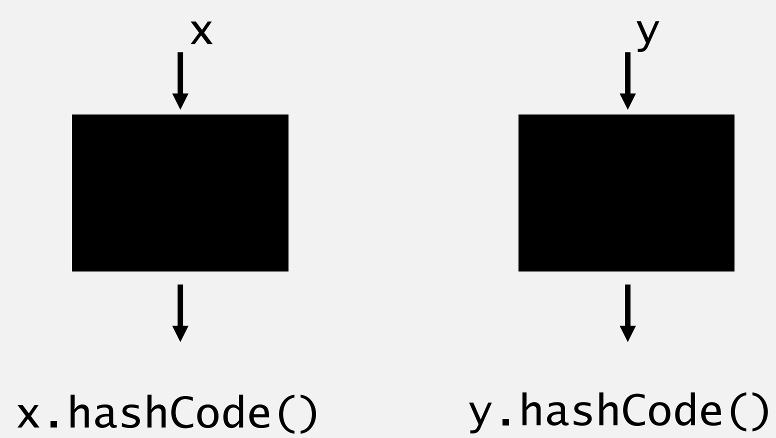
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return `17`.

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, …

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

Java library implementation

```
public final class String
{
   private final char[] s;
   ...
   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	...

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

(Horner's method)

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
   private int hash = 0;
   private final char[] s;
   ...

   public int hashCode()
   {
      int h = hash;
      if (h != 0) return h;
      for (int i = 0; i < length(); i++)
         h = s[i] + (31 * h);
      hash = h;
      return h;
   }
}
```

cache of hash code

return cached value

store cache of hash code

Q. What if `hashCode()` of string is 0?

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  amount;

   public Transaction(String who, Date when, double amount)
   {  /* as before */  }
   ...

   public boolean equals(Object y)
   {  /* as before */  }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

typically a small prime

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, return 0.
- If field is a reference type, use `hashCode()`. ← applies rule recursively
- If field is an array, apply to each entry. ← or use `Arrays.deepHashCode()`

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" hash functions exist.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31} - 1$.

Hash function. An `int` between `0` and `M - 1` (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

bug

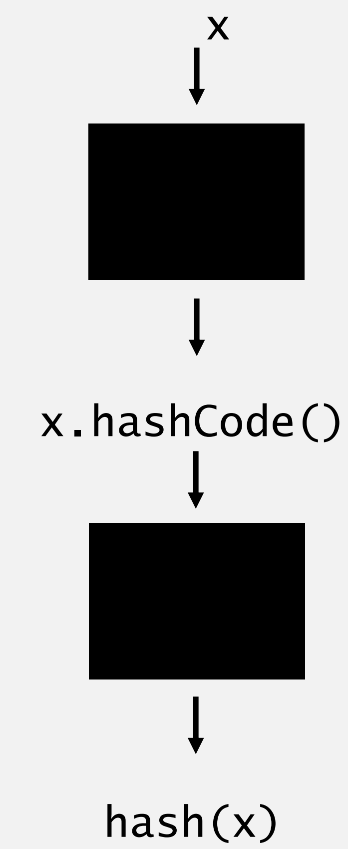
```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

`hashCode()` of `"polygenelubricants"` is -2^{31}

```
private int hash(Key key)
{  return (key.hashCode() & 0x7fff ffff) % M;  }
```

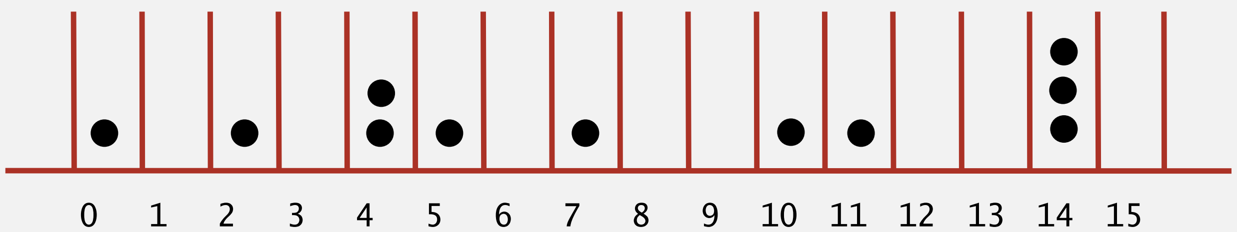
correct



Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M-1$.

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

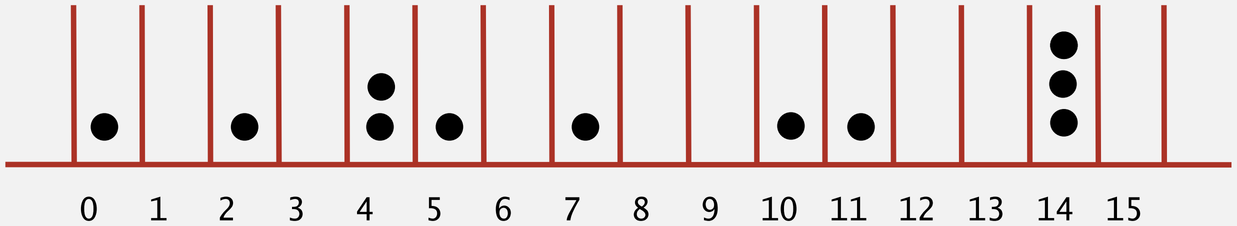
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

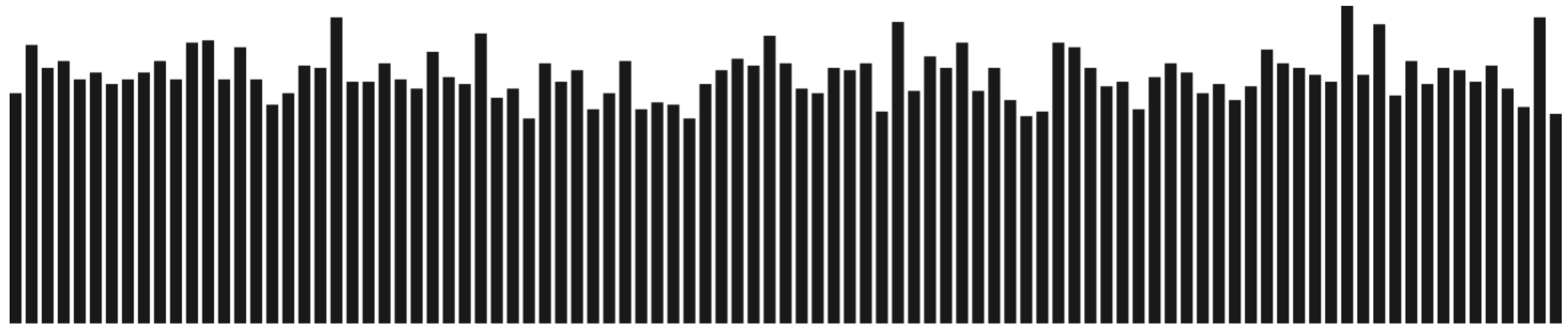
Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M-1$.

Bins and balls. Throw balls uniformly at random into M bins.





Hash value frequencies for words in Tale of Two Cities (M = 97)

Java's `String` data uniformly distribute the keys of Tale of Two Cities

http://algs4.cs.princeton.edu

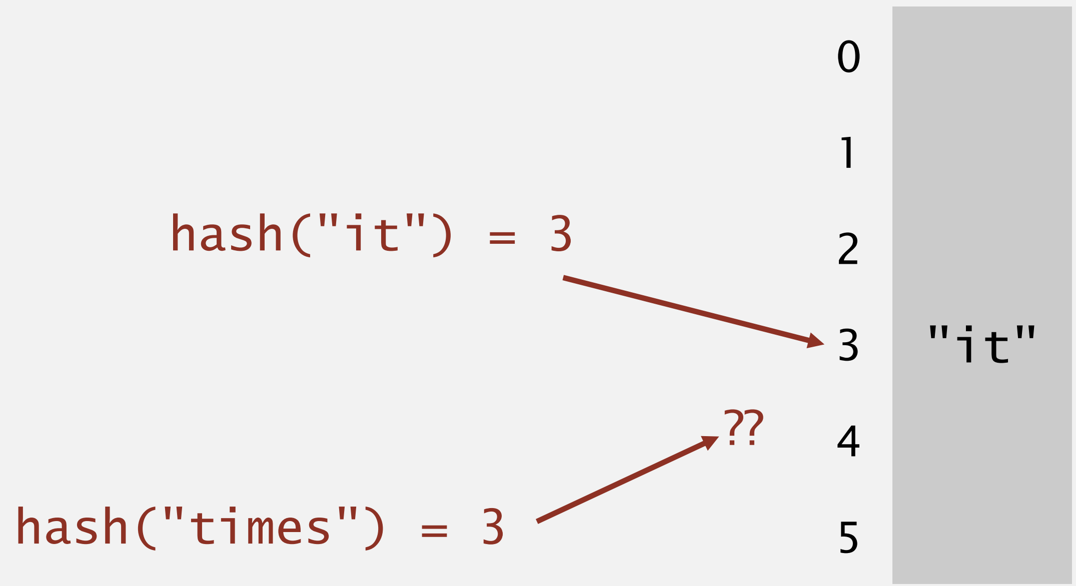
Hash Tables

- *hash functions*
- *separate chaining*
- *linear probing*

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem $\Rightarrow$ can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing $\Rightarrow$ collisions are evenly distributed.



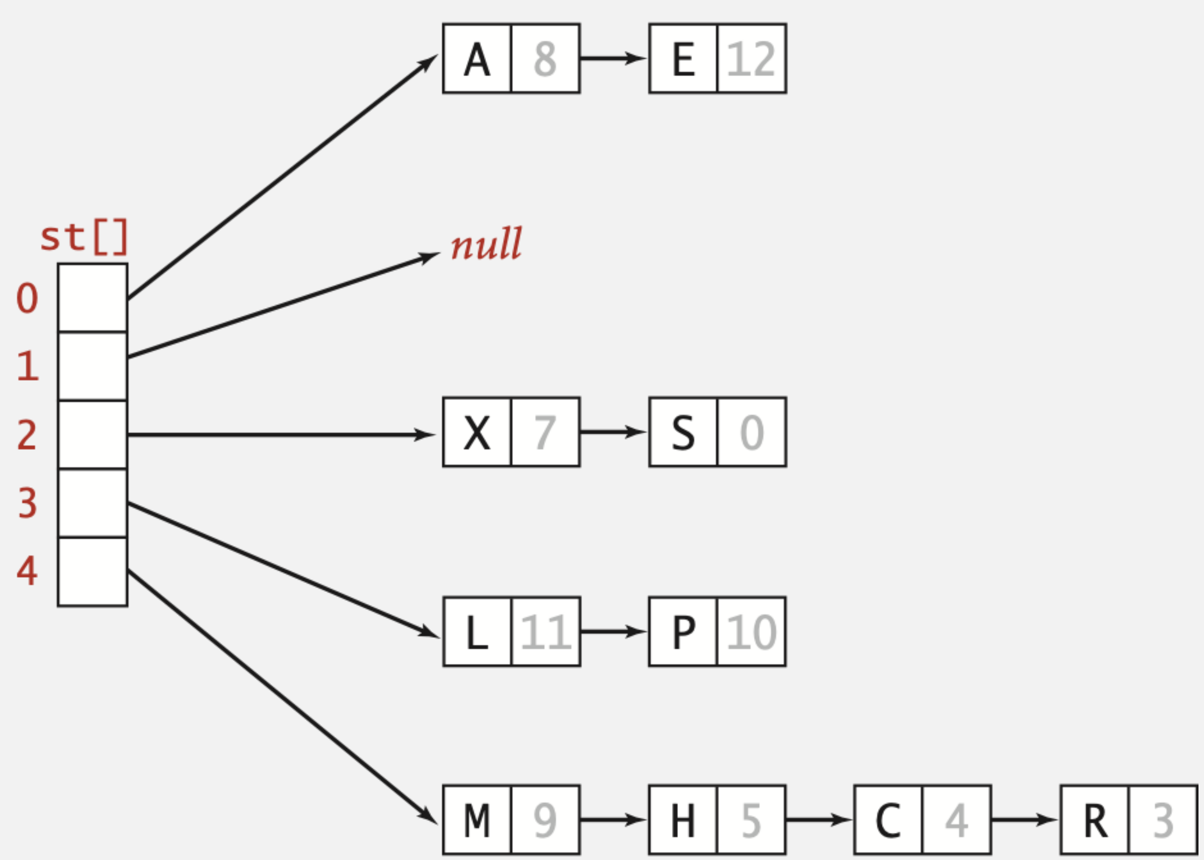
Challenge. Deal with collisions efficiently.

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];   // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }
}
```

array doubling and halving code omitted

no generic array creation (declare key and value of type Object)

Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;               // number of chains
   private Node[] st = new Node[M];  // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public void put(Key key, Value val) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) { x.val = val; return; }
      st[i] = new Node(key, val, st[i]);
   }
}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Consequence. Number of probes for search/insert is proportional to N / M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 4 \Rightarrow$ constant-time ops.

Resizing in a separate-chaining hash table

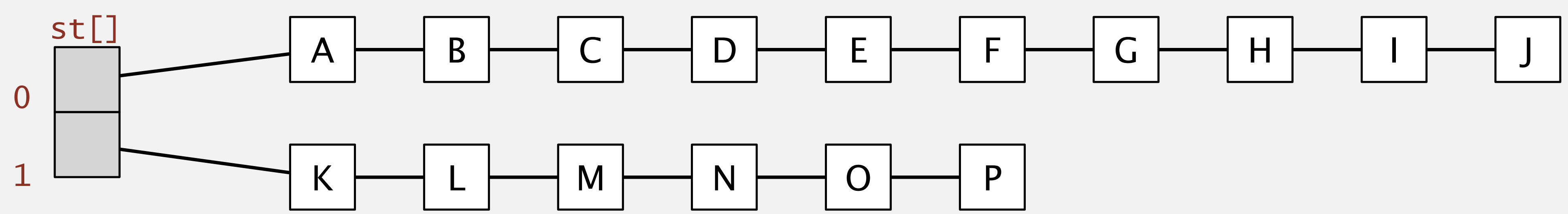
Goal. Average length of list N / M = constant.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing.

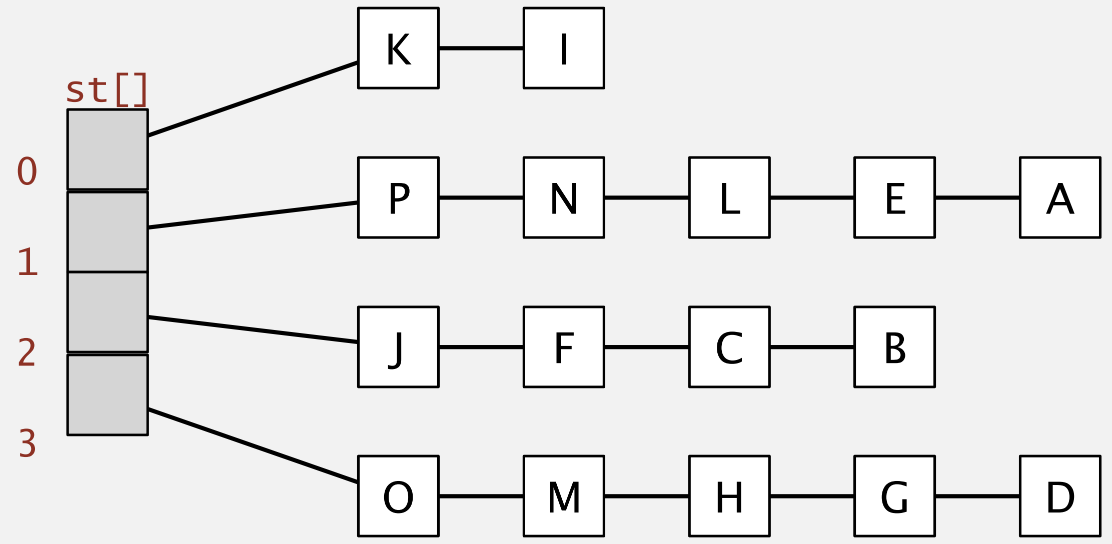
`x.hashCode()` does not change

but `hash(x)` can change

before resizing



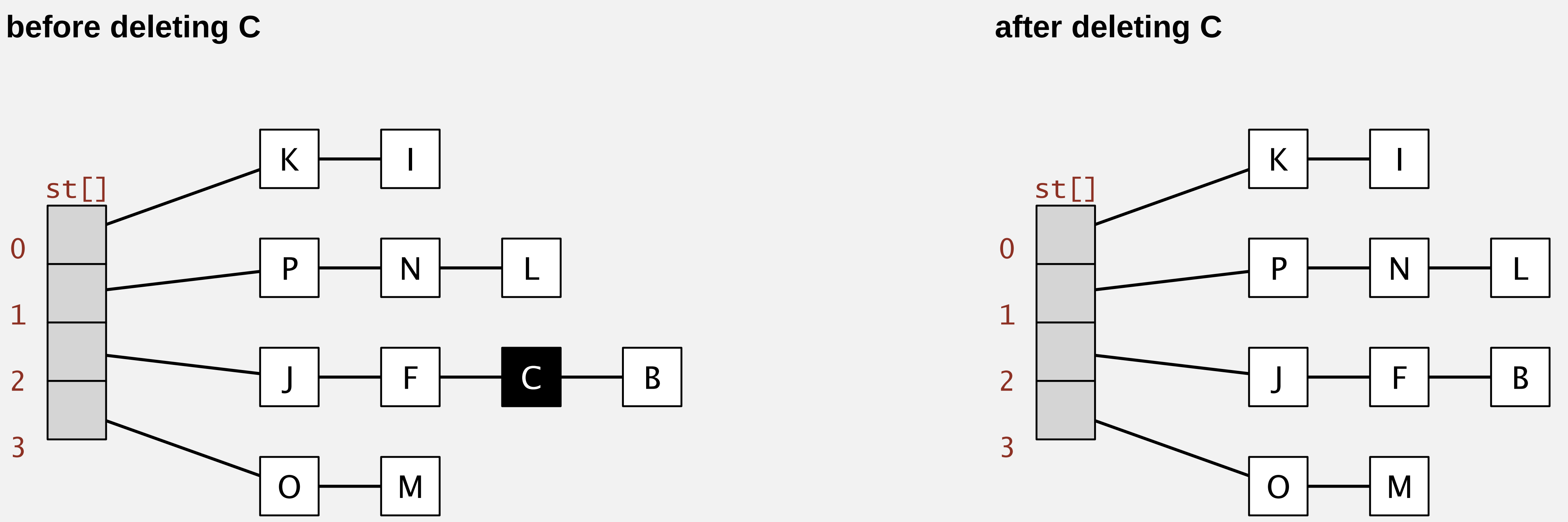
after resizing



Deletion in a separate-chaining hash table

Q. How to delete a key (and its associated value)?

A. Easy: need only consider chain containing key.



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	`compareTo()`
red-black BST	2 lg N	2 lg N	2 lg N	1.0 lg N	1.0 lg N	1.0 lg N	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

* under uniform hashing assumption

http://algs4.cs.princeton.edu

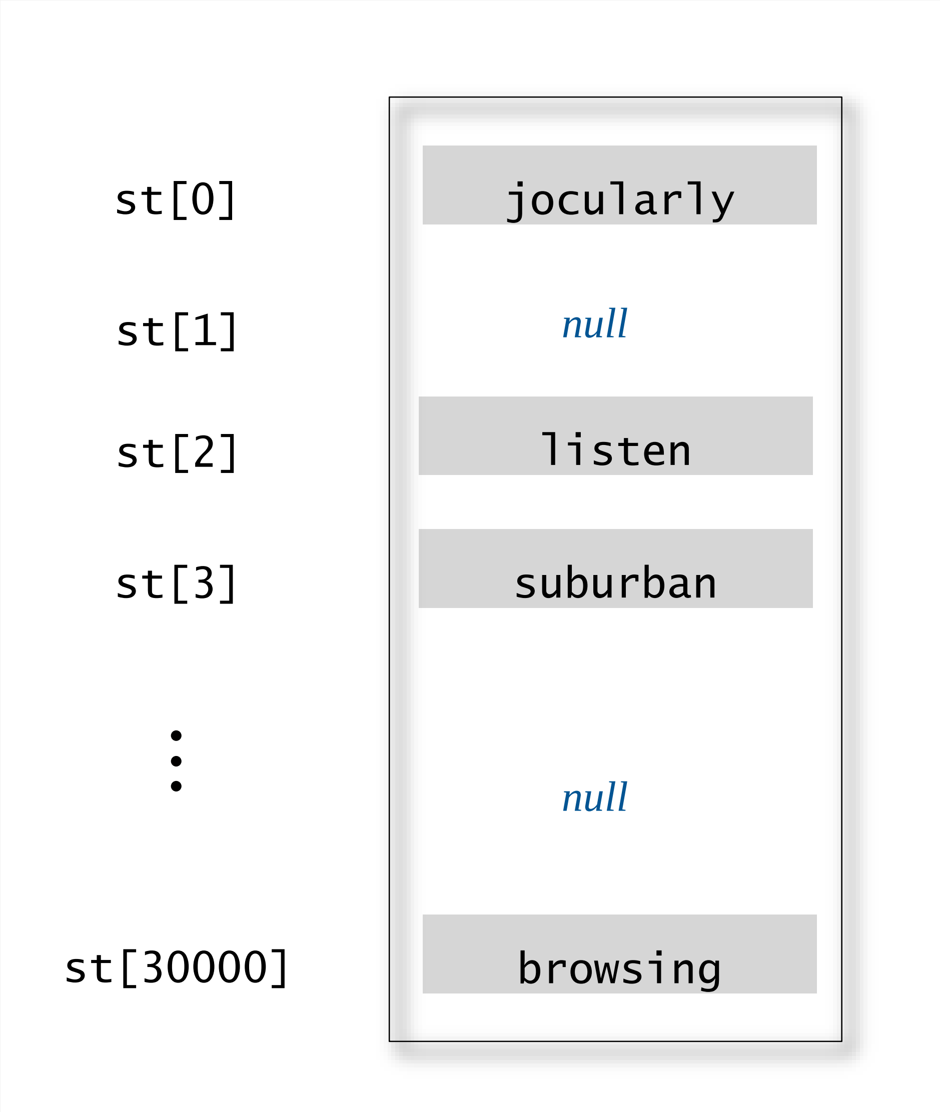
Hash Tables

- hash functions
- separate chaining
- *linear probing*

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert S

hash(S) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

M = 16

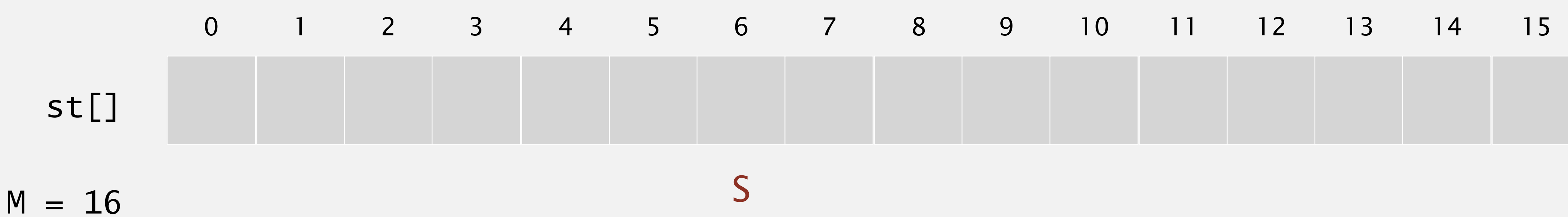
Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert S

hash(S) = 6



Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert S

hash(S) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`							S									

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`							S									

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									
M = 16											E					

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`							S				E					

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert A

hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert A

hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					
					A											

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert A

hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A		S				E					

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert R

hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert R

hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					
															R	

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert R

hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A		S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A		S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert C

hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A		S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert C

hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A		S				E				R	
						C										

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert C

hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S				E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
					H											

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
						H										

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
							H									

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S				E				R	
								H								

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert X

hash(X) = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert X

hash(X) = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	
																X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert X

hash(X) = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert M

hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X

M = 16

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert M

hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`					A	C	S	H			E				R	X
`M = 16`		M														

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert M

hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`		M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`		M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert P

hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`		M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert P

hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`		M			A	C	S	H			E				R	X
															P	

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert P

hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`		M			A	C	S	H			E				R	X
`M = 16`	P															P

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert P

hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X
							L									

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X
								L								

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H			E				R	X
									L							

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: insert

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
											E					

`M = 16`

search hit

(return corresponding value)

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
							L									

M = 16

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
								L								

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

L

search hit

(return corresponding value)

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
						K										

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
							K									

M = 16

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
								K								

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
									K							

`M = 16`

Linear-probing hash table demo: search

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
										K						

`M = 16`

search miss

(return null)

Linear-probing hash table summary

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

Note. Array size `M` must be greater than number of key-value pairs `N`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)             {  /* as before */   }

   private void put(Key key, Value val) {  /* next slide */  }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }
}
```

array doubling and halving code omitted

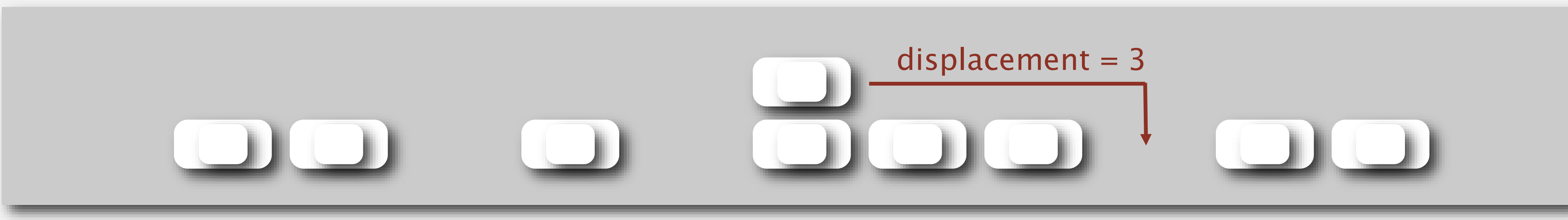
Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)   { /* as before      */  }

   private Value get(Key key)  { /* previous slide */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

}
```

Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.
Each desires a random space i : if space i is taken, try $i + 1$, $i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.

Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

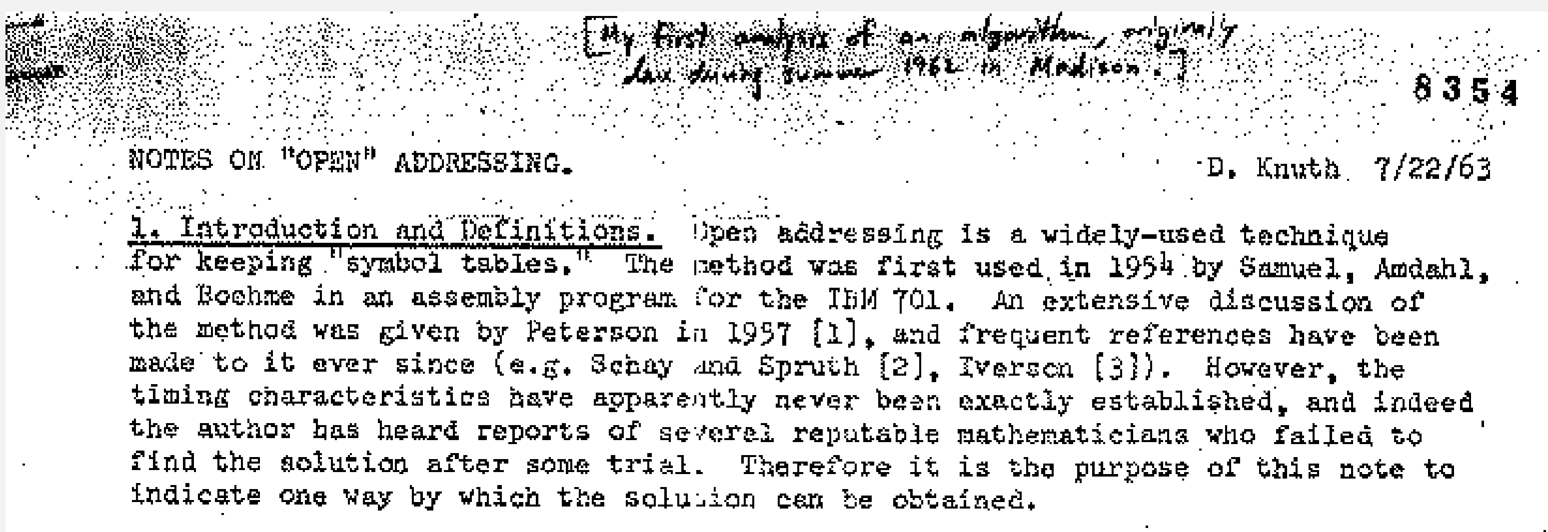
Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit **search miss / insert**

Pf.

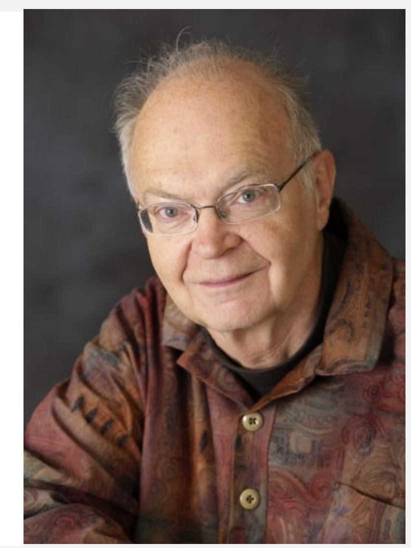


[My first analysis of an algorithm, originally done during summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ⟵ # probes for search hit is about 3/2

probes for search miss is about 5/2

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq ½$.

- Double size of array M when $N / M \geq ½$.
- Halve size of array M when $N / M \leq ⅛$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can't just delete array entries.

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if hash(H) = 4

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

* under uniform hashing assumption

http://algs4.cs.princeton.edu

Undirected Graphs

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

http://algs4.cs.princeton.edu

UNDIRECTED GRAPHS

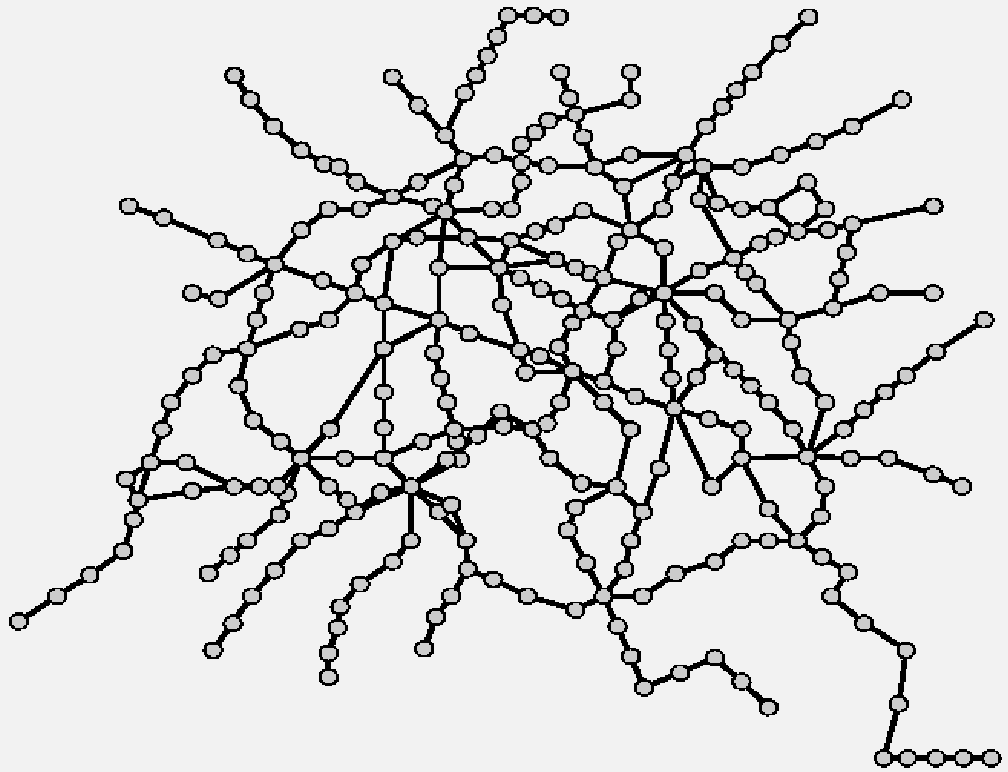
- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *connected components*
- *challenges*

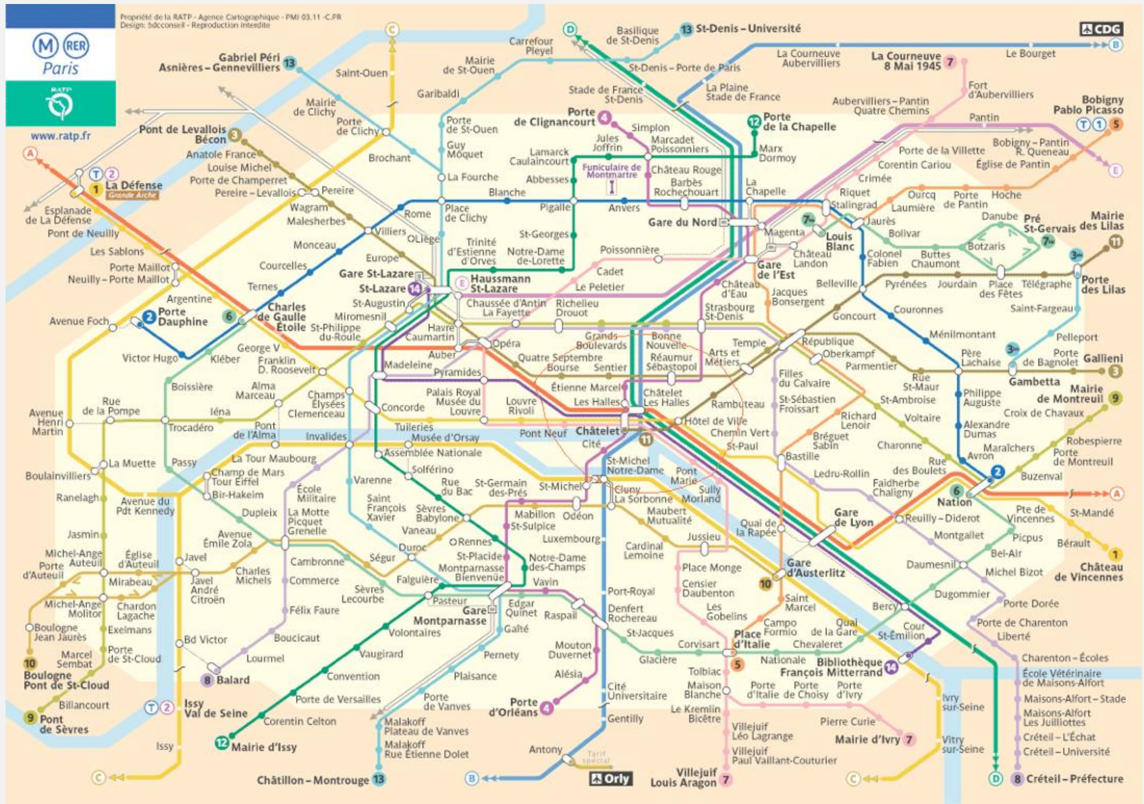
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.





Graph applications

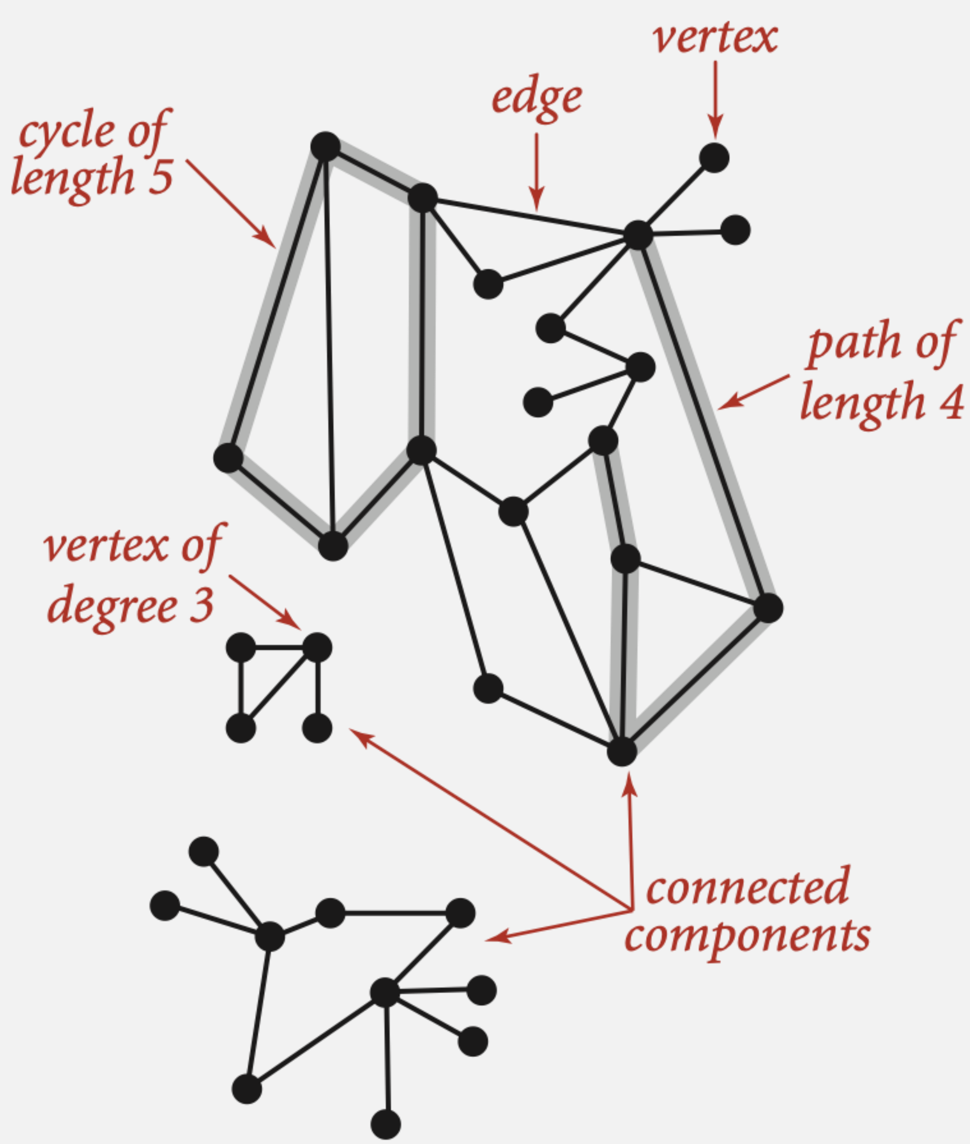
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

Graph terminology

Path. Sequence of vertices connected by edges.

Cycle. Path whose first and last vertices are the same.

Two vertices are connected if there is a path between them.



Some graph-processing problems

problem	description
s-t path	*Is there a path between s and t ?*
shortest s-t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once ?*
connectivity	*Is there a way to connect all of the vertices ?*
biconnectivity	*Is there a vertex whose removal disconnects the graph ?*
planarity	*Can the graph be drawn in the plane with no crossing edges ?*
graph isomorphism	*Do two adjacency lists represent the same graph ?*

Challenge. Which graph problems are easy? difficult? intractable?

http://algs4.cs.princeton.edu

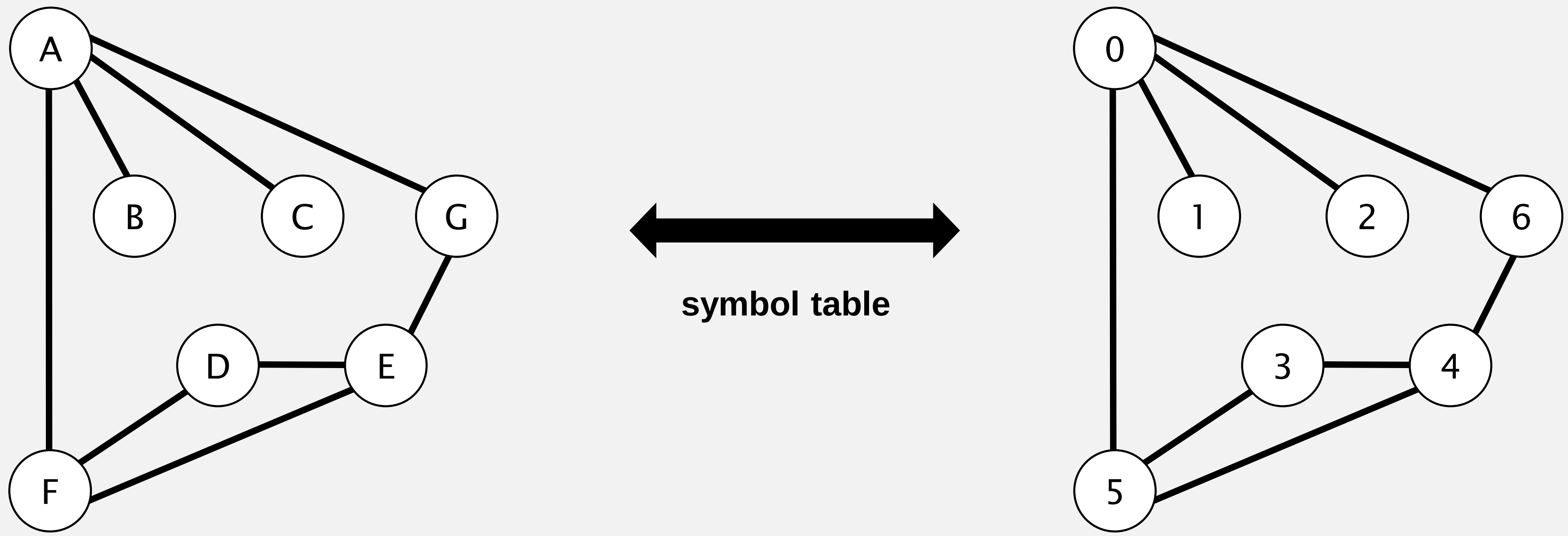
Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

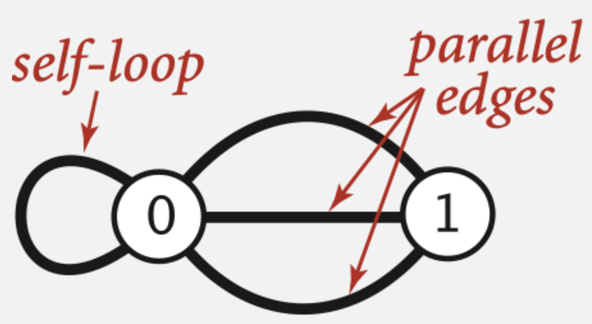
Graph representation

Vertex representation.

- This lecture: use integers between 0 and $V-1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

public class **Graph**		
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*

```
// degree of vertex v in graph G
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v))
       degree++;
    return degree;
}
```

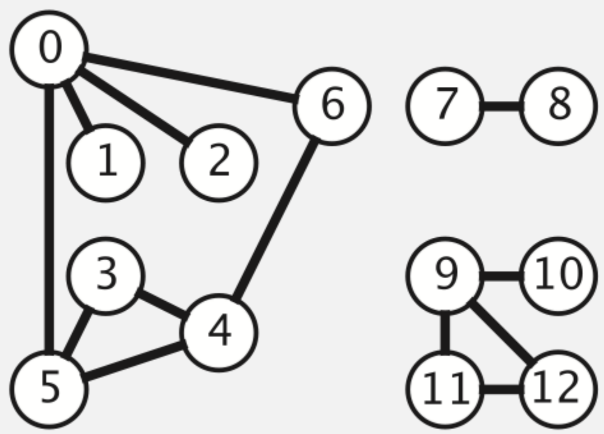
Graph API: sample client

Graph input format.

tinyG.txt

V → 13
13 ← E

```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```




```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
⋮
12-11
12-9
```

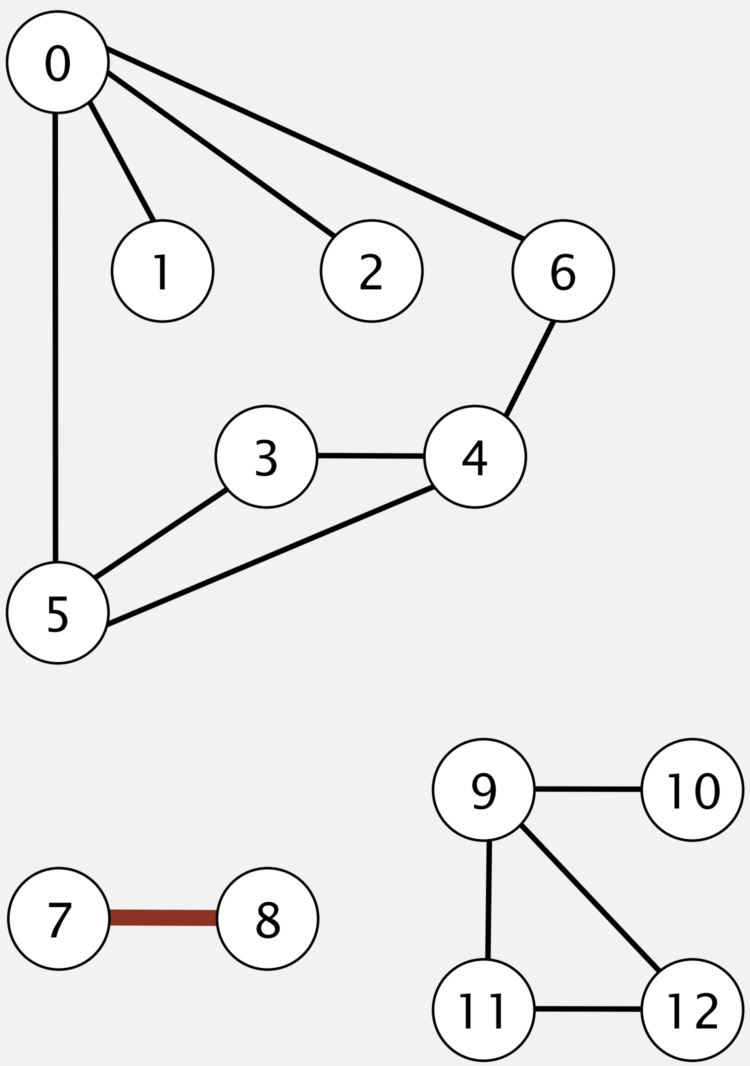
```
In in = new In(args[0]);
Graph G = new Graph(in);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
```

read graph from input stream

print out each edge (twice)

Graph representation: set of edges

Maintain a list of the edges (linked list or array).



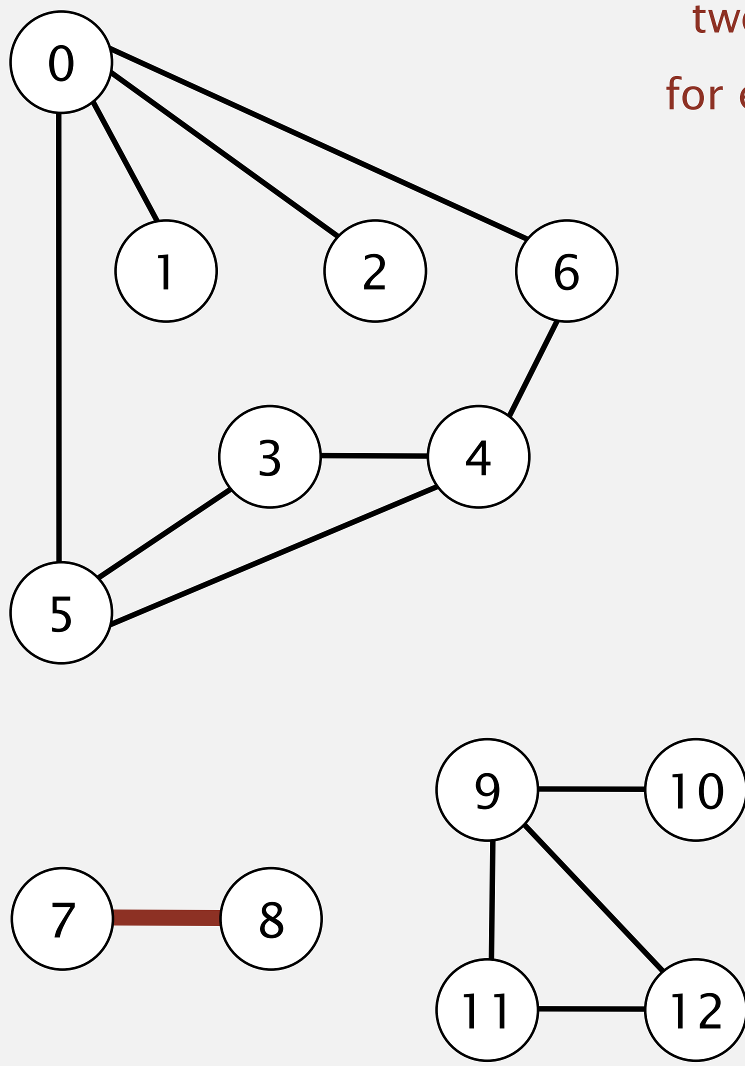
0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Q. How long to iterate over vertices adjacent to v ?

Graph representation: adjacency matrix

Maintain a two-dimensional V-by-V boolean array;

for each edge $v–w$ in graph: `adj[v][w] = adj[w][v] = true`.



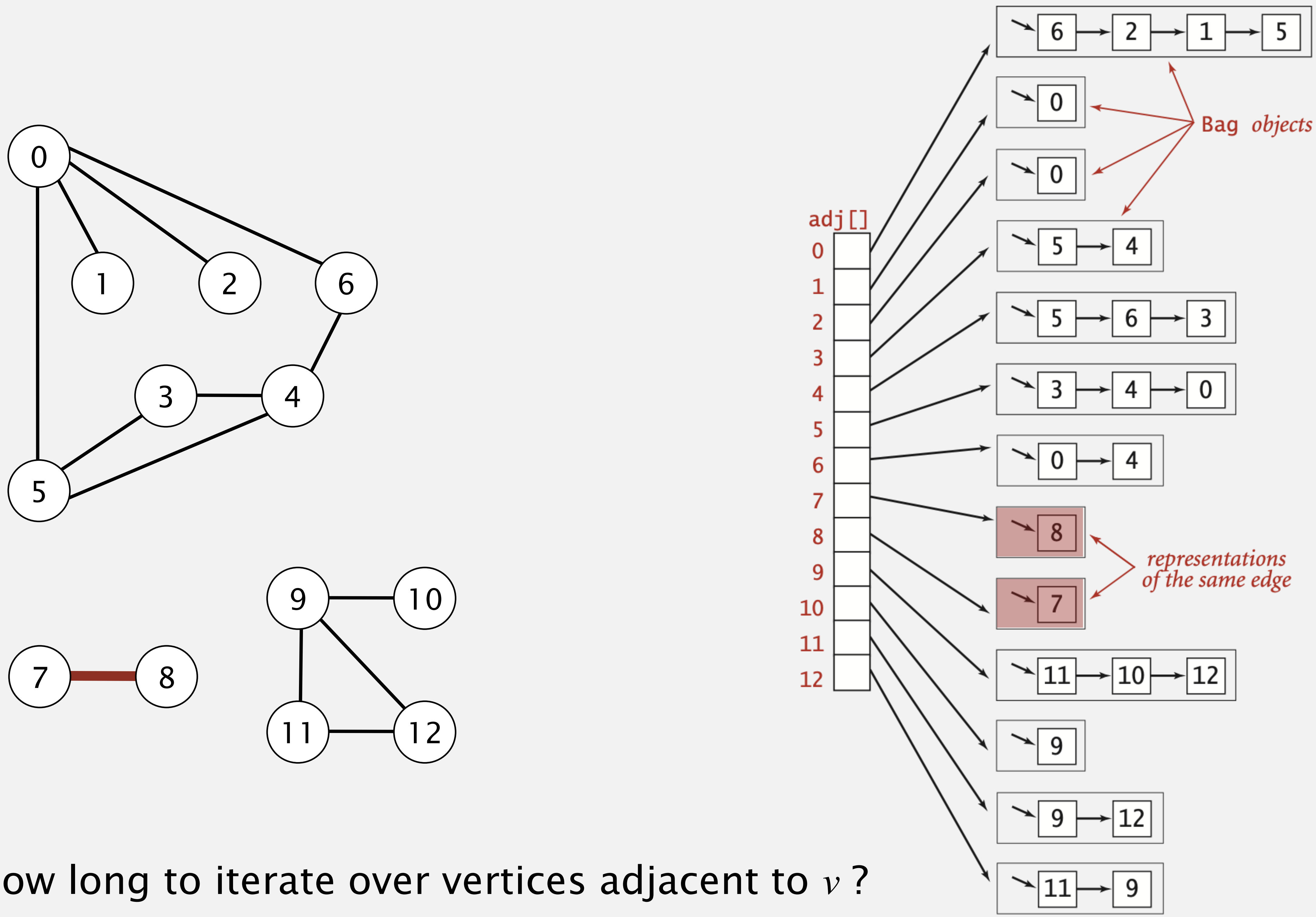
two entries for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1

Q. How long to iterate over vertices adjacent to v ?

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.



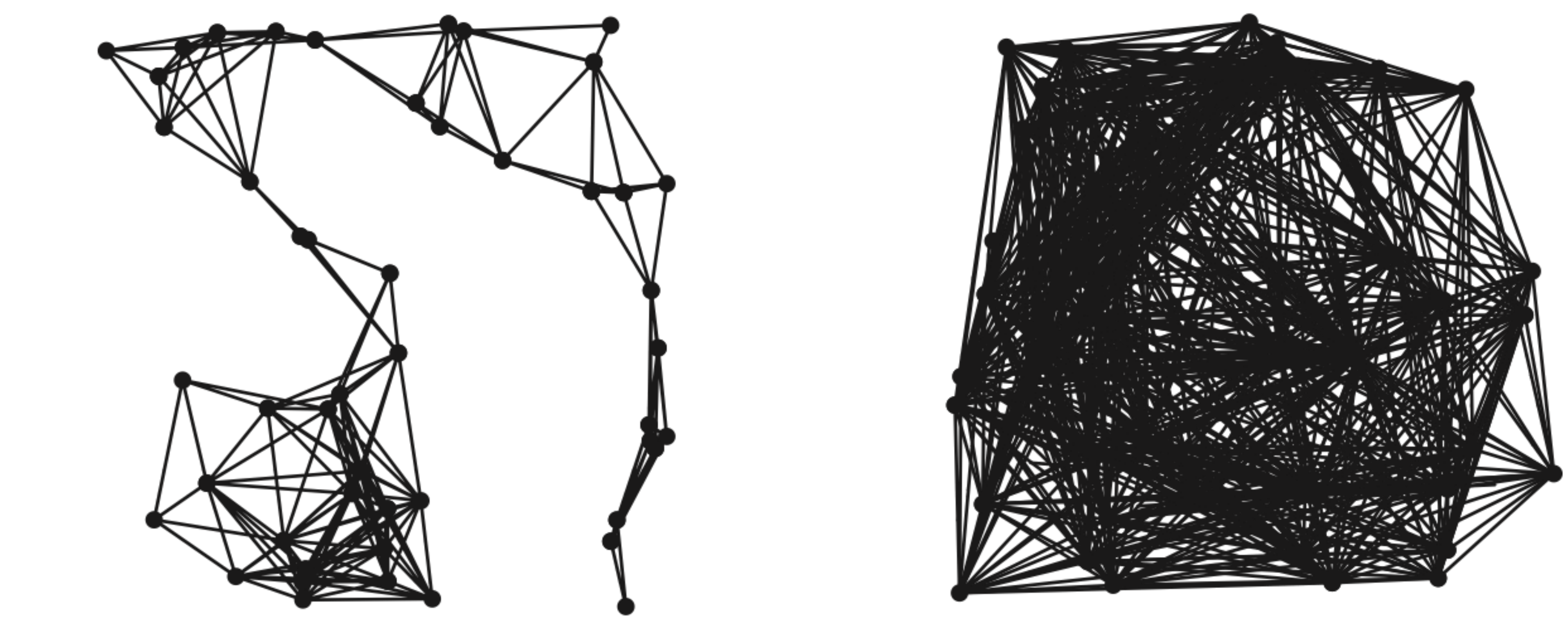
Q. How long to iterate over vertices adjacent to v ?

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse.

huge number of vertices,
small average vertex degree



Two graphs (V = 50)

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

* disallows parallel edges

Adjacency-list graph representation: Java implementation

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;
   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }
   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }
   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists
(using Bag data type)

create empty graph
with V vertices, Initialize all lists to
empty

add edge v-w
(parallel edges and
self-loops allowed)

iterator for vertices adjacent to v

http://algs4.cs.princeton.edu

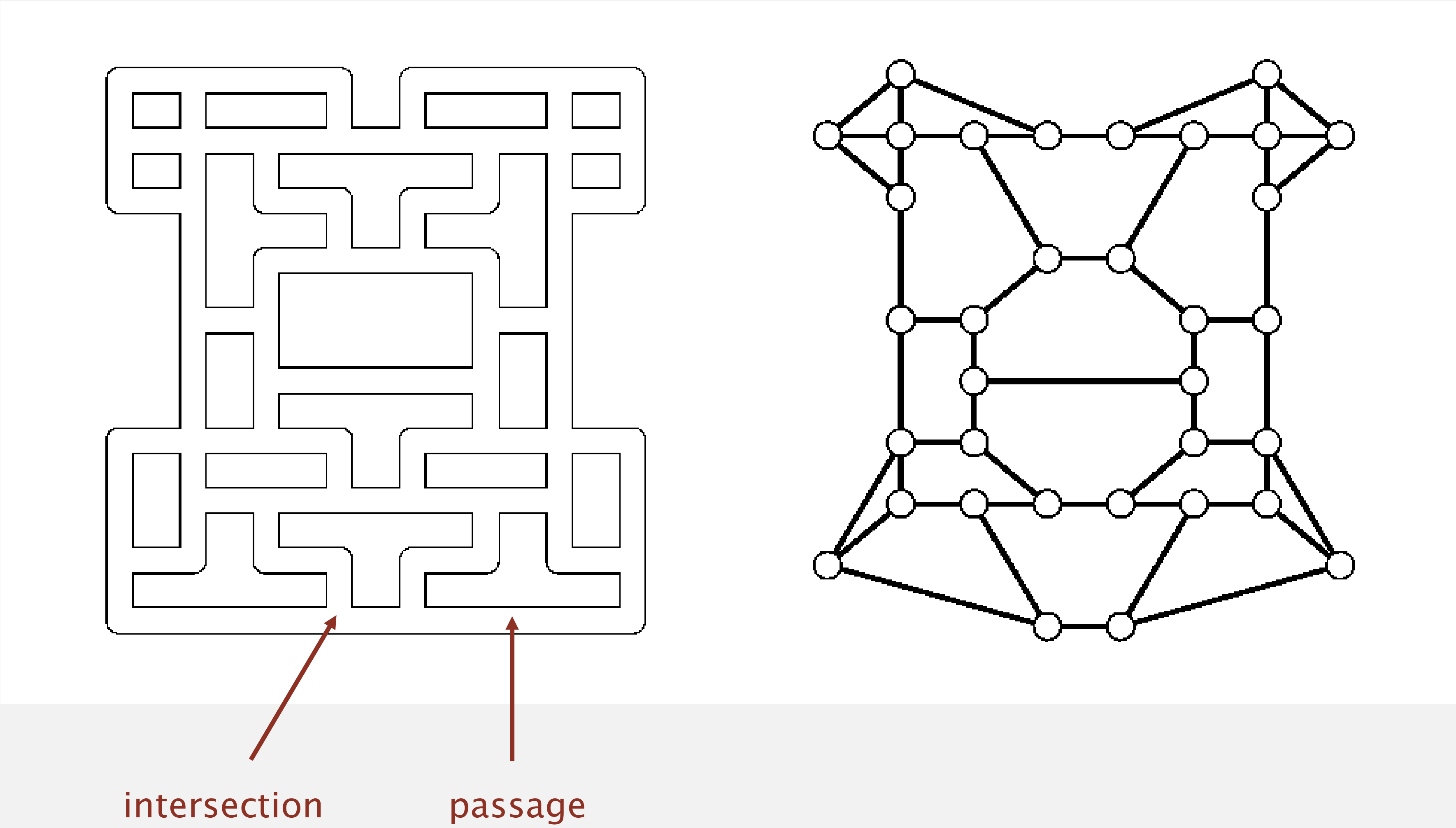
Undirected Graphs

- introduction
- graph API
- **depth-first search**
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

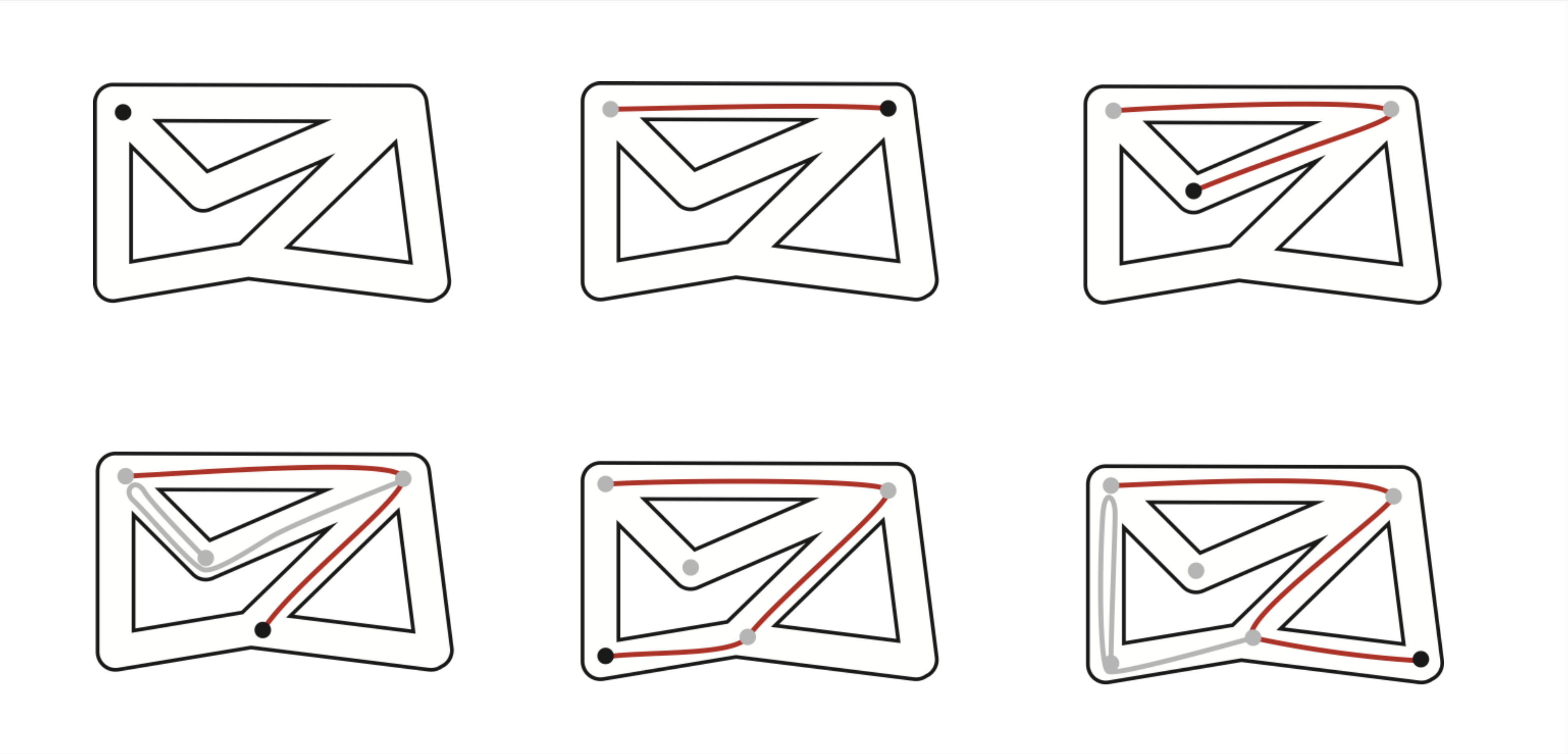


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



Depth-first search

Goal. Systematically traverse a graph.

Idea. Mimic maze exploration. ← function-call stack acts as ball of string

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

Depth-first search: data structures

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

Data structures.

- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.
  `(edgeTo[w] == v)` means that edge `v-w` taken to visit `w` for first time
- Function-call stack for recursion.

Depth-first search: Java implementation

```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int s;
   public DepthFirstPaths(Graph G, int s)
   {
      ...
      dfs(G, s);
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            dfs(G, w);
            edgeTo[w] = v;
         }
   }
}
```

marked[v] = true
if v connected to s

edgeTo[v] = previous vertex
on path from s to v

initialize data structures

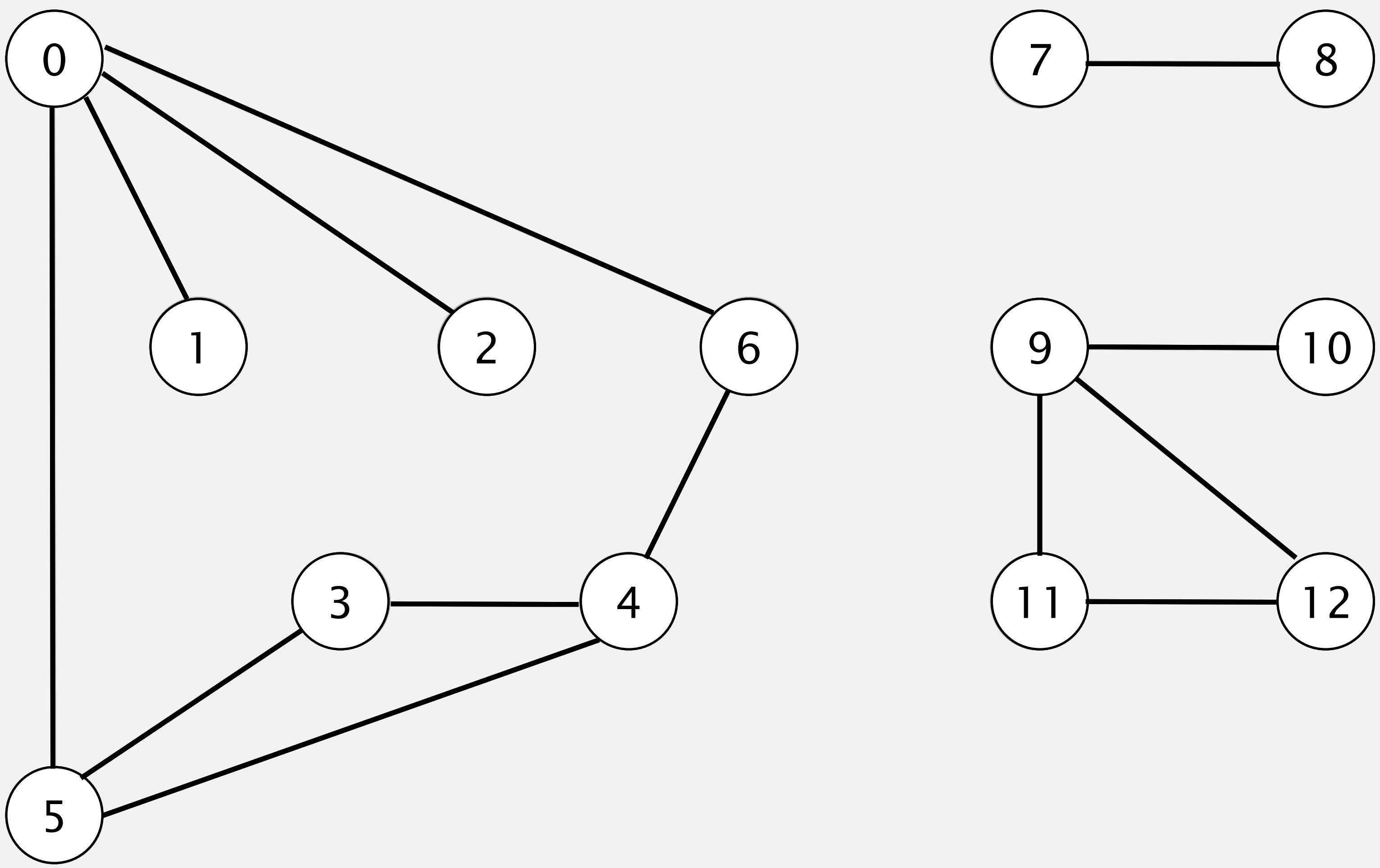
find vertices connected to s

recursive DFS does the work

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



tinyG.txt

V → 13

13 ← E

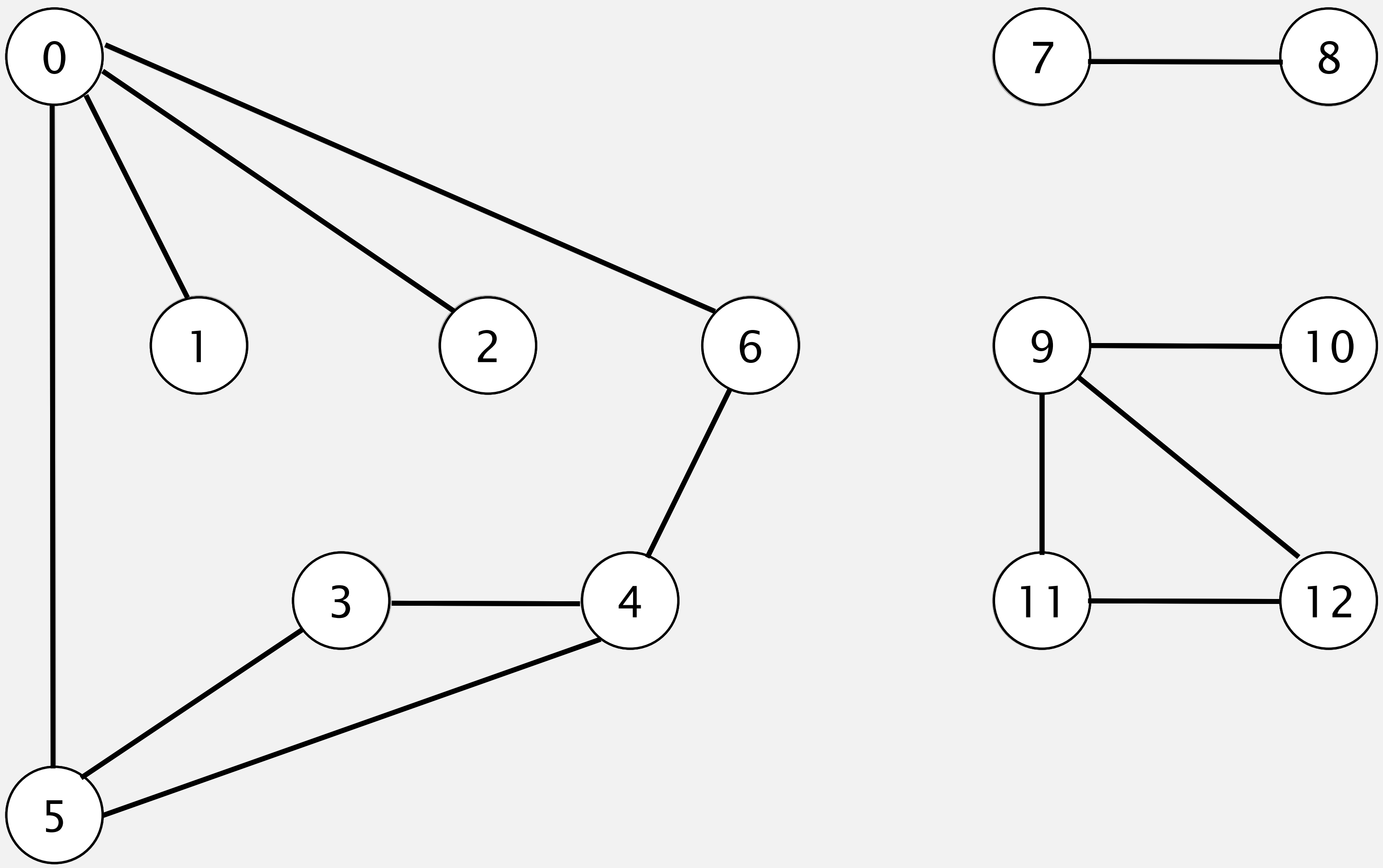
```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

graph G

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



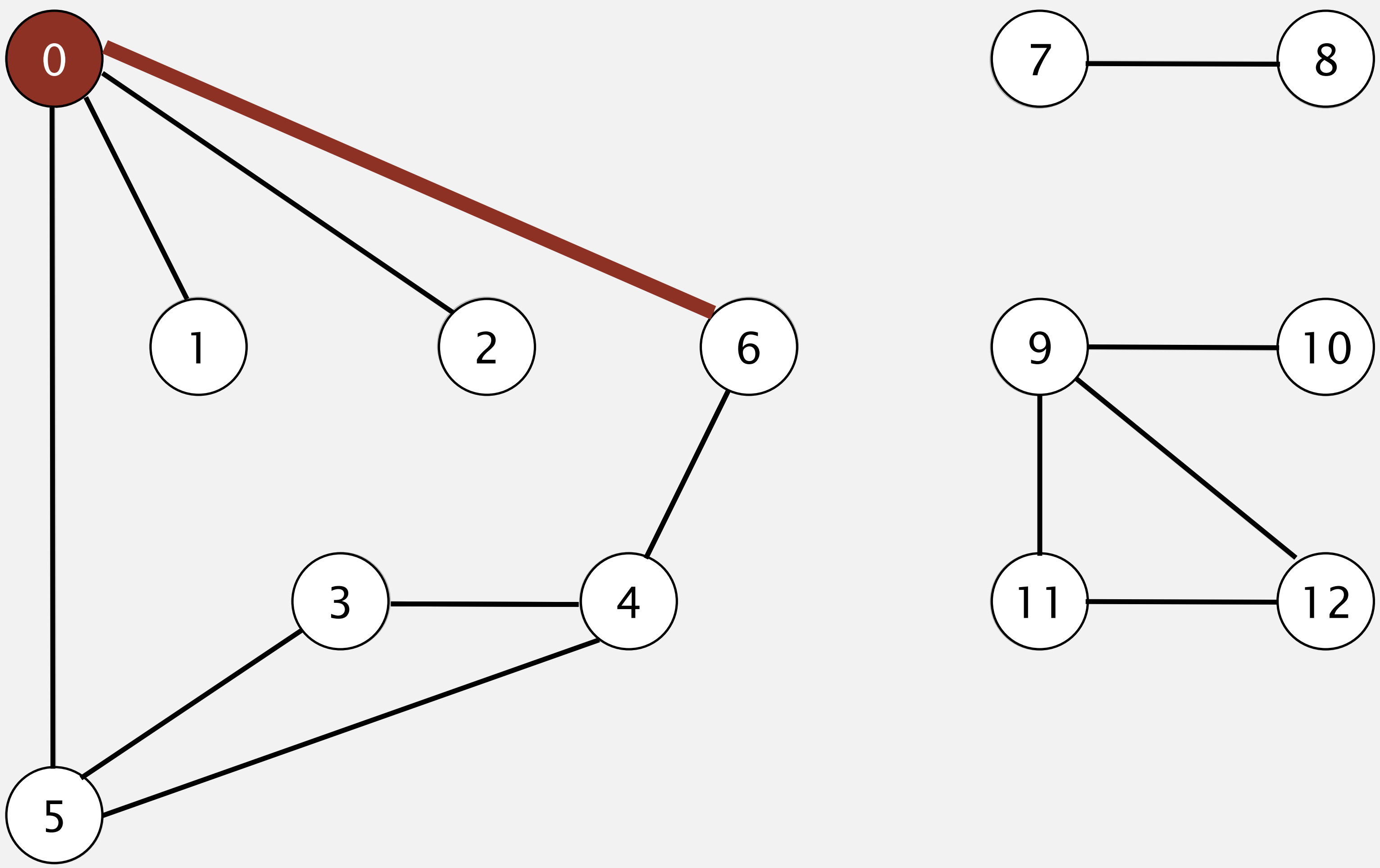
graph G

v	marked[]	edgeTo[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



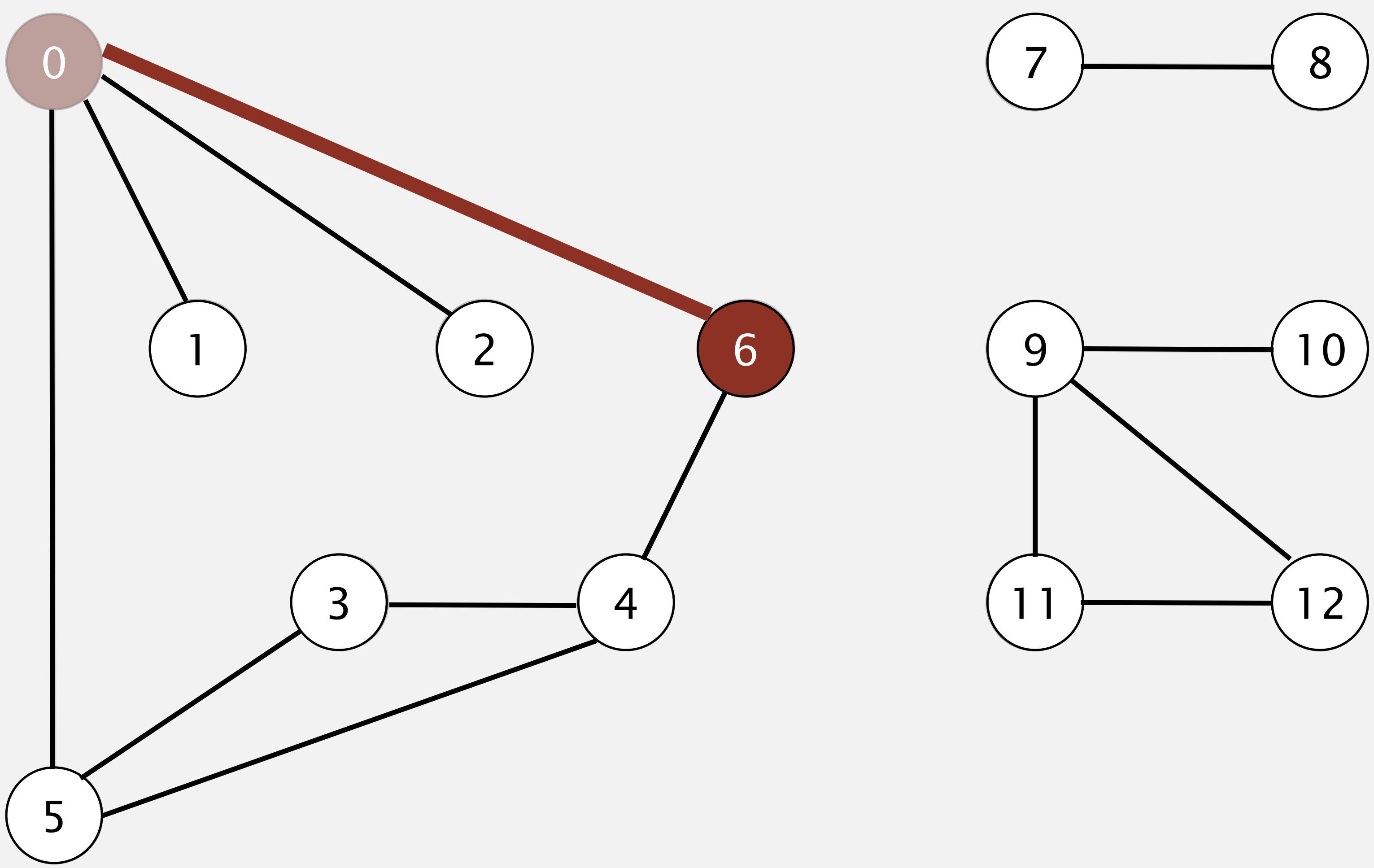
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



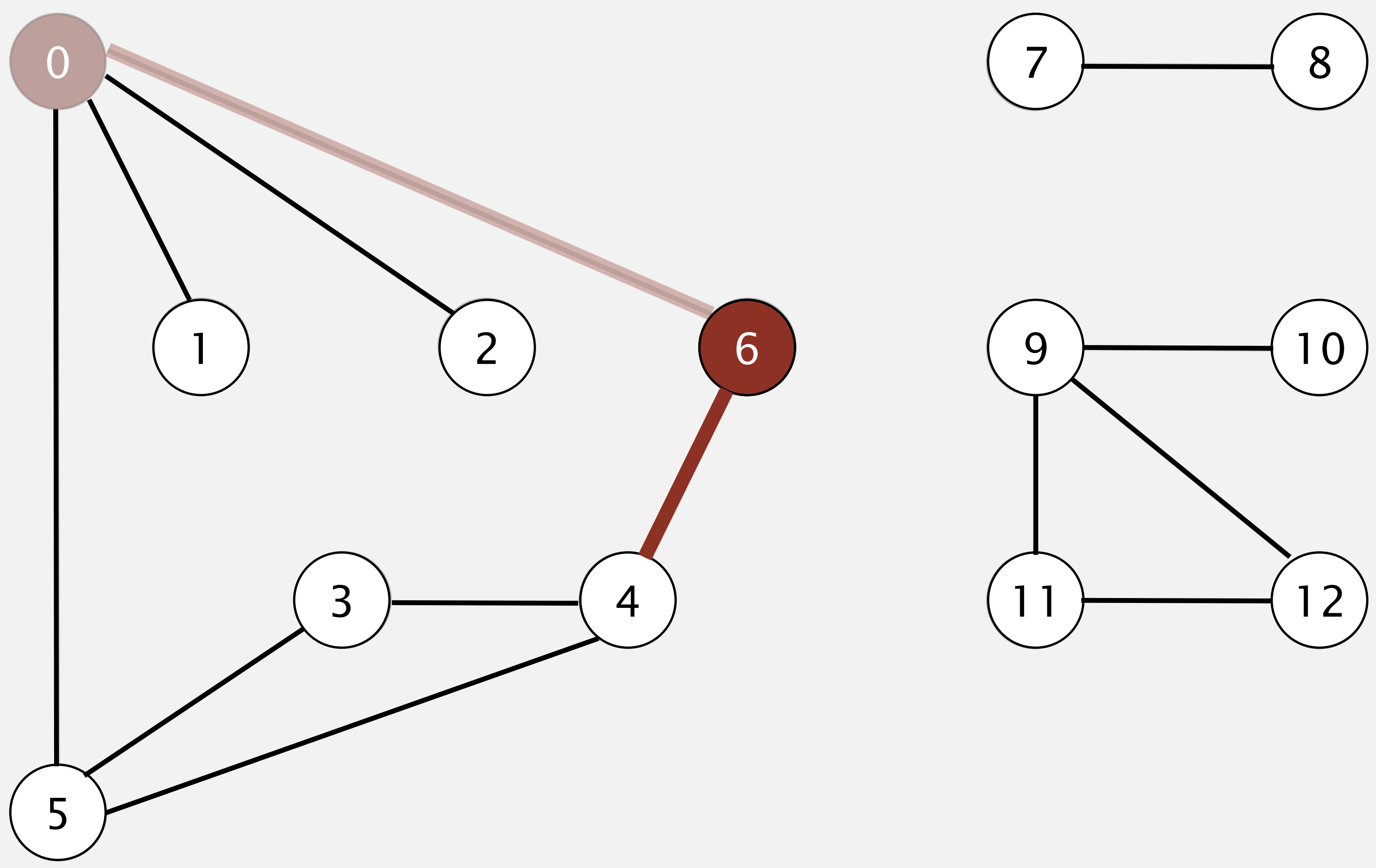
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



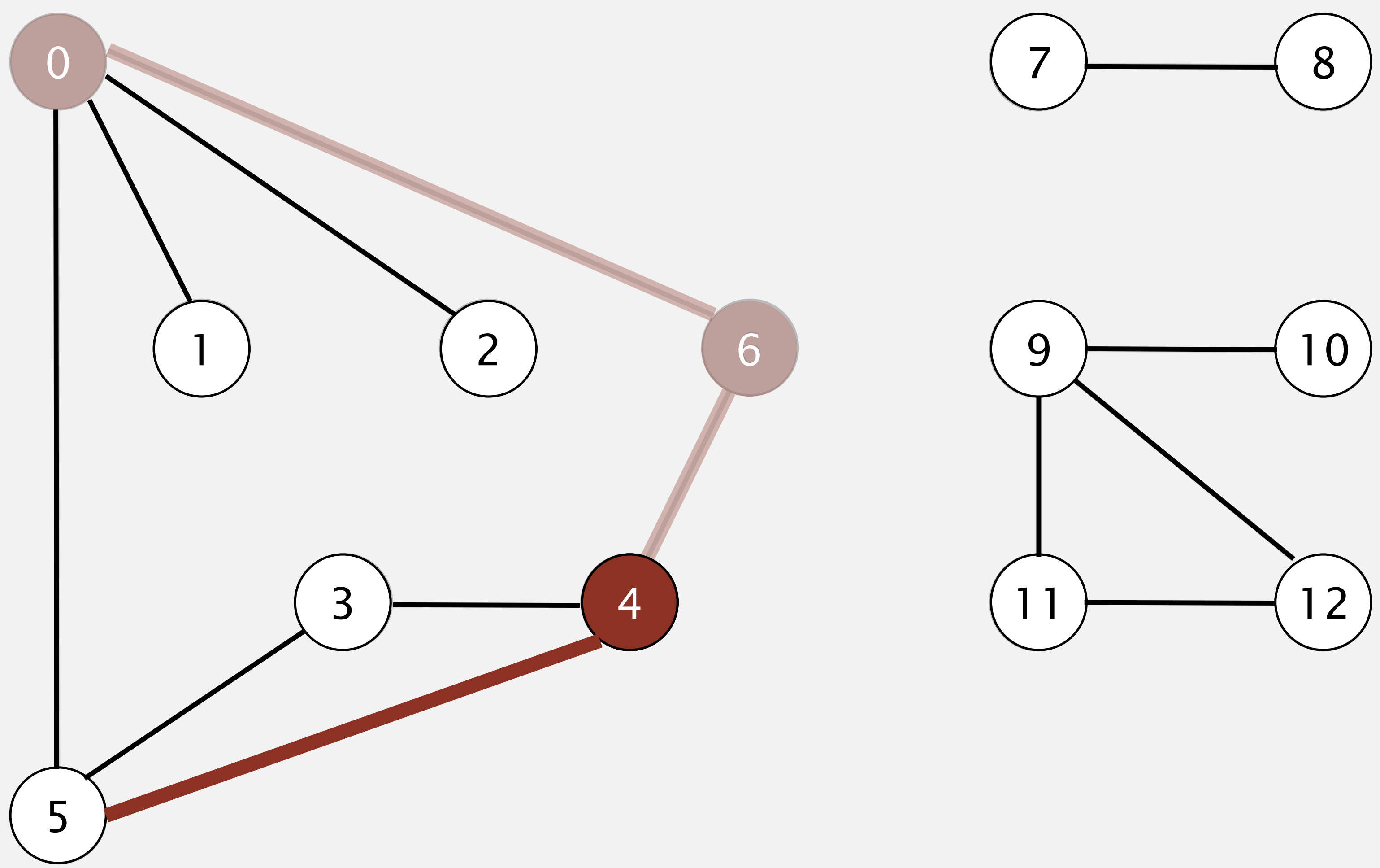
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



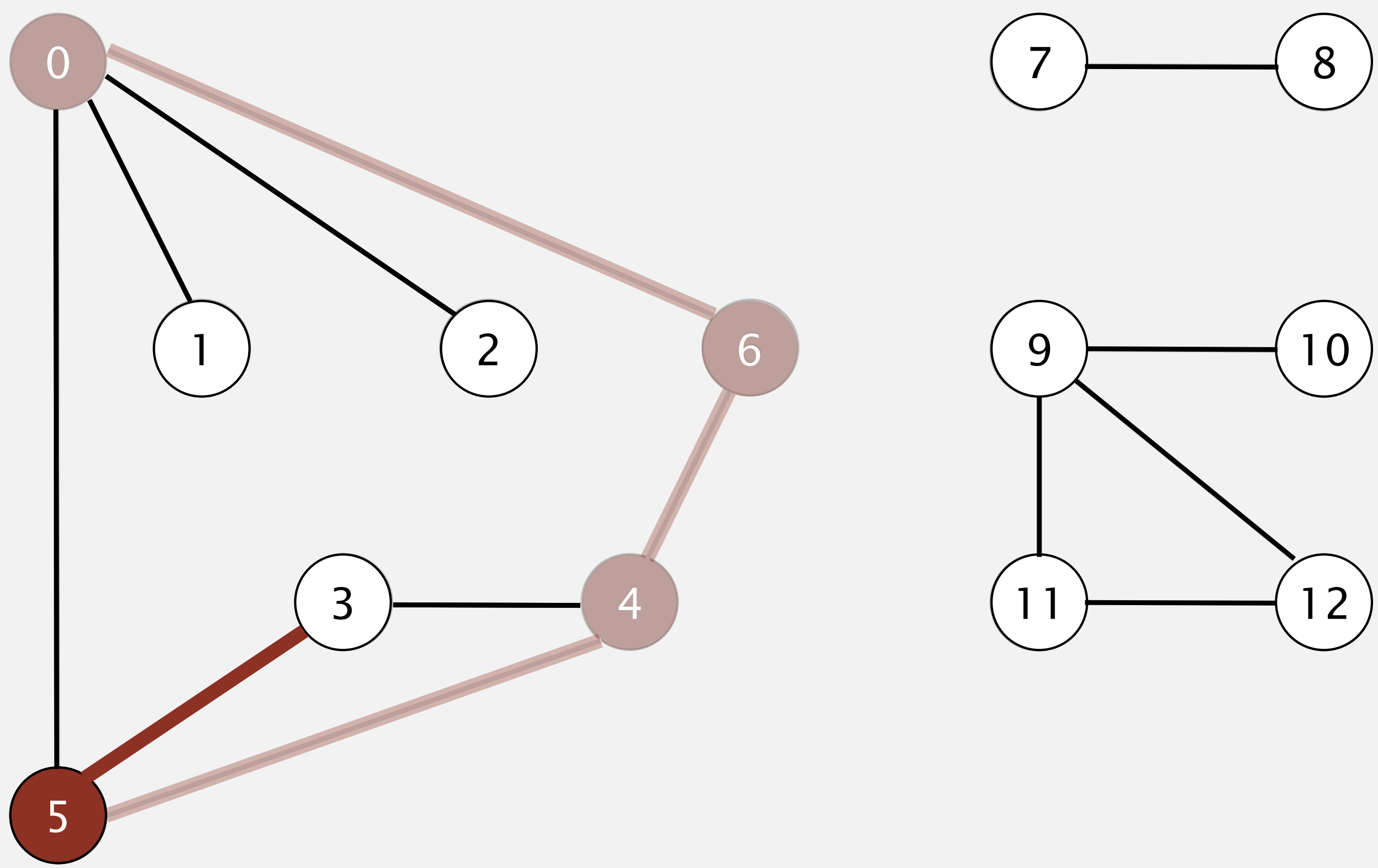
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



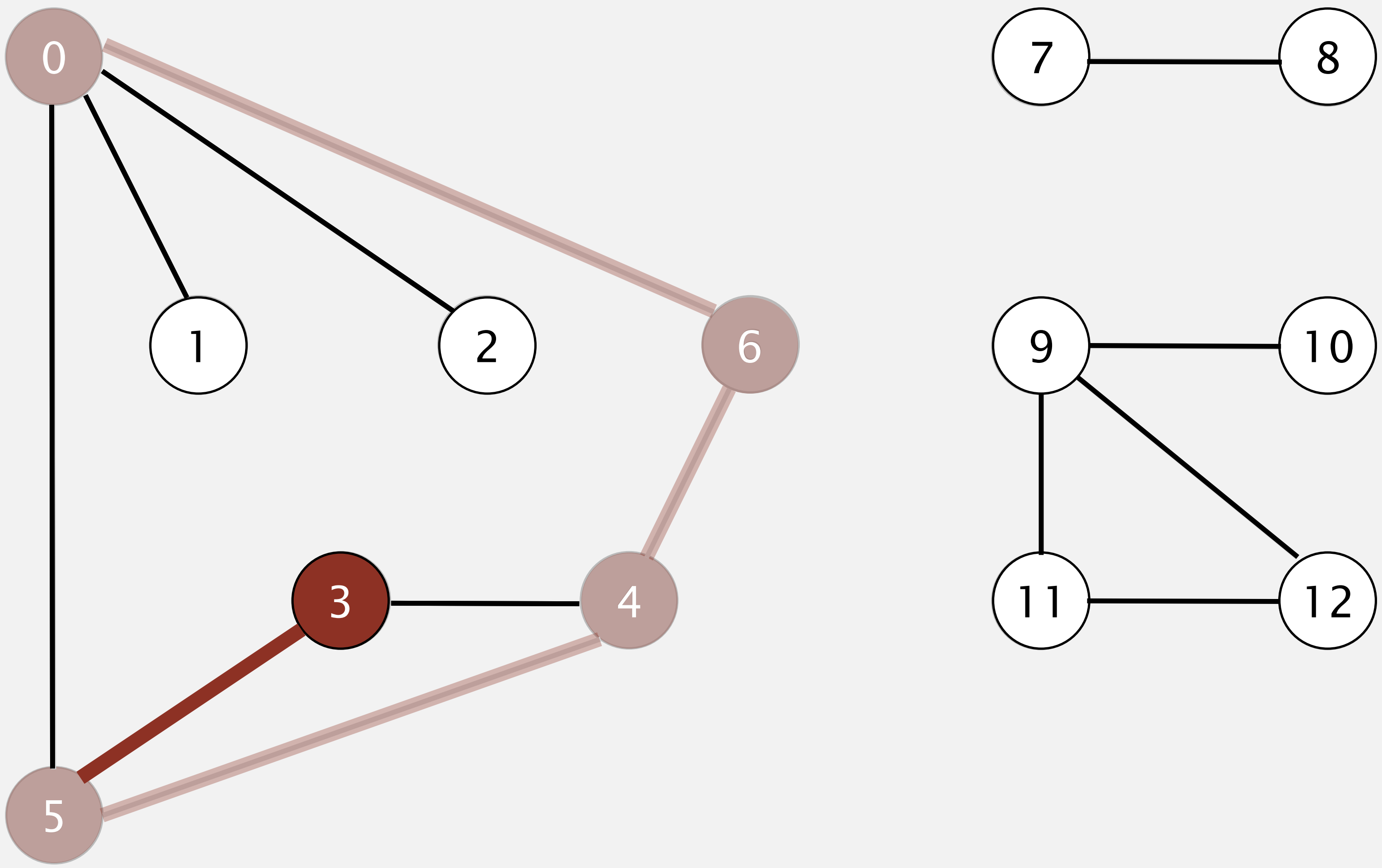
visit 5

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



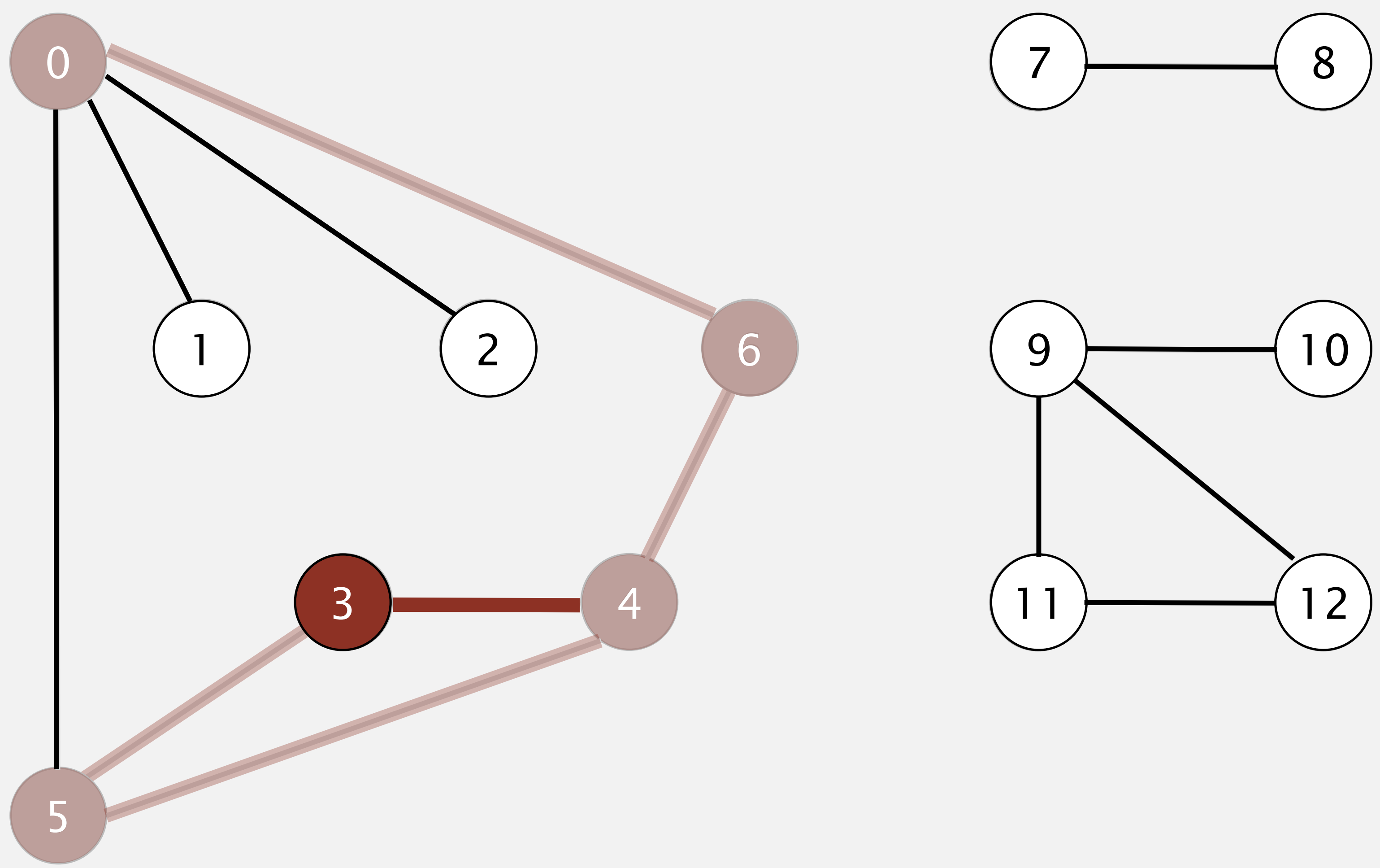
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



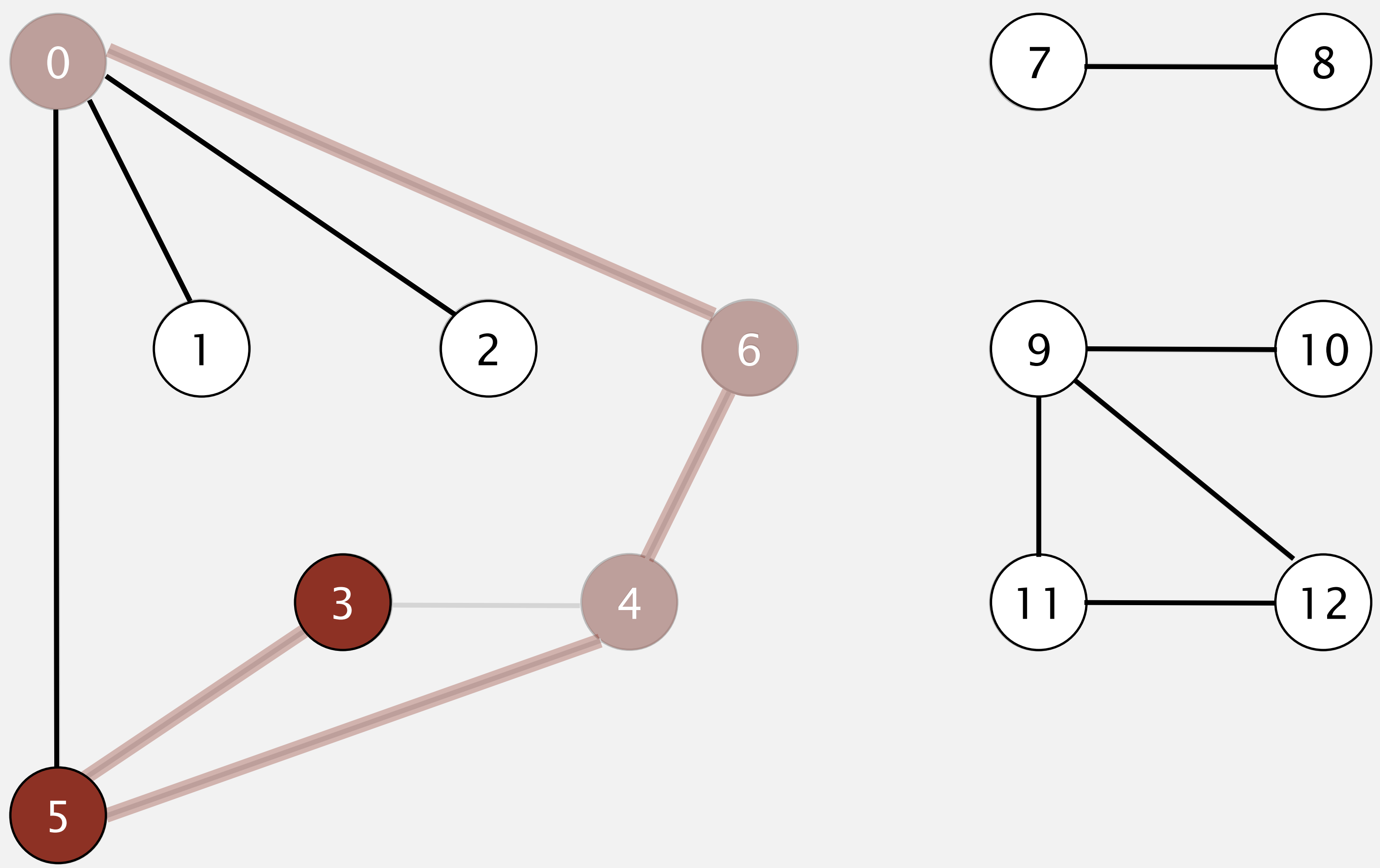
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



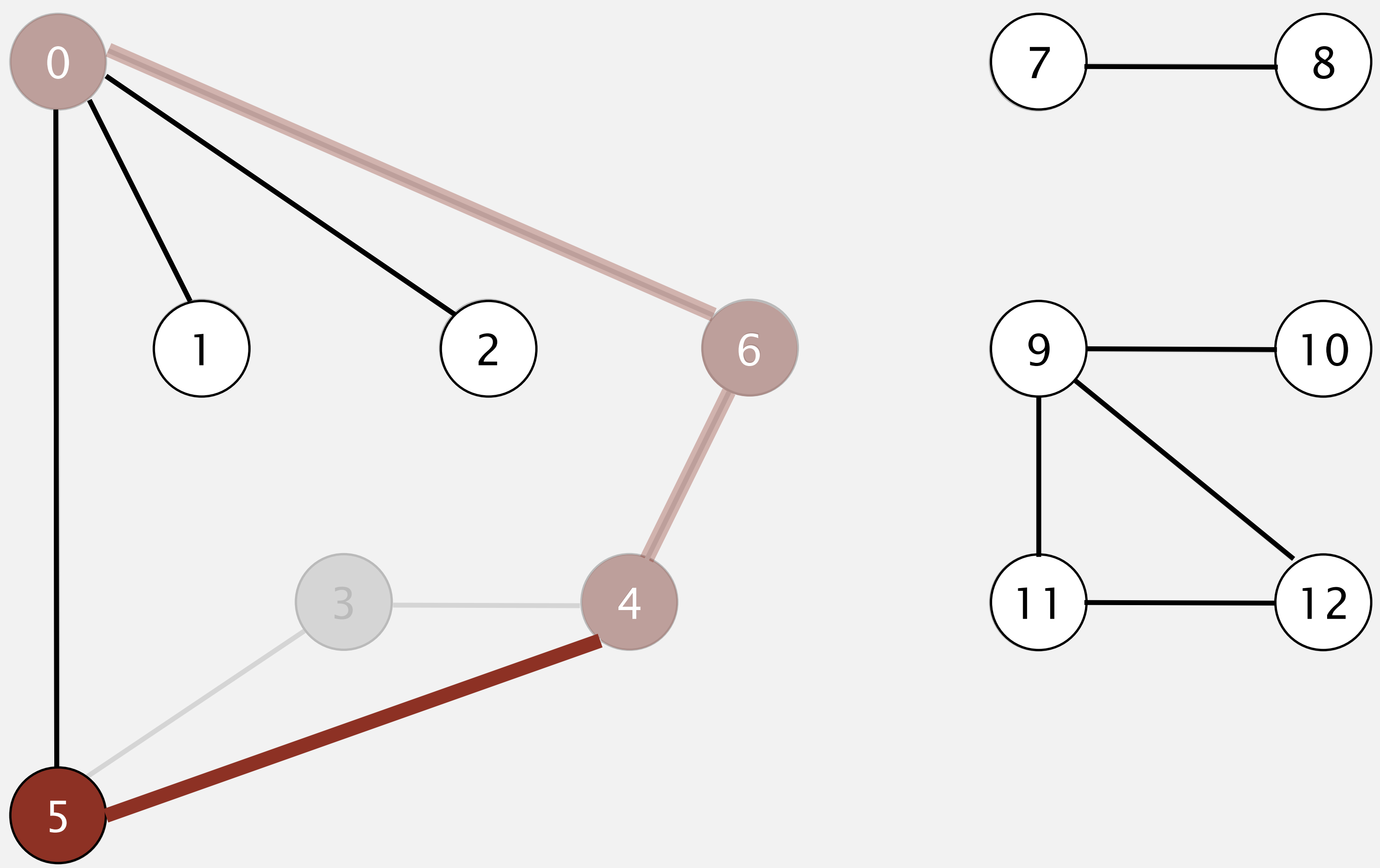
3 done

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



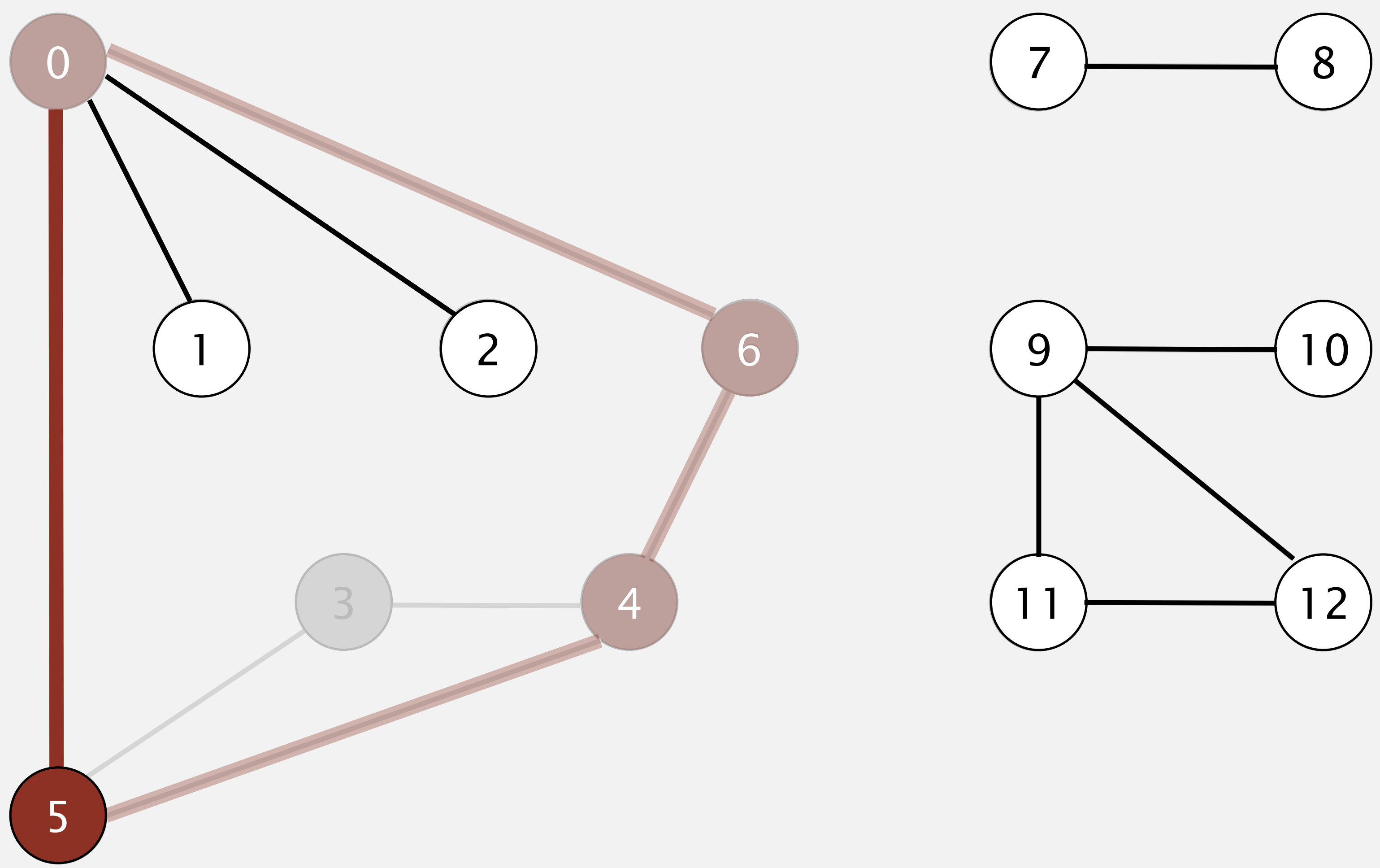
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5

Depth-first search demo

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



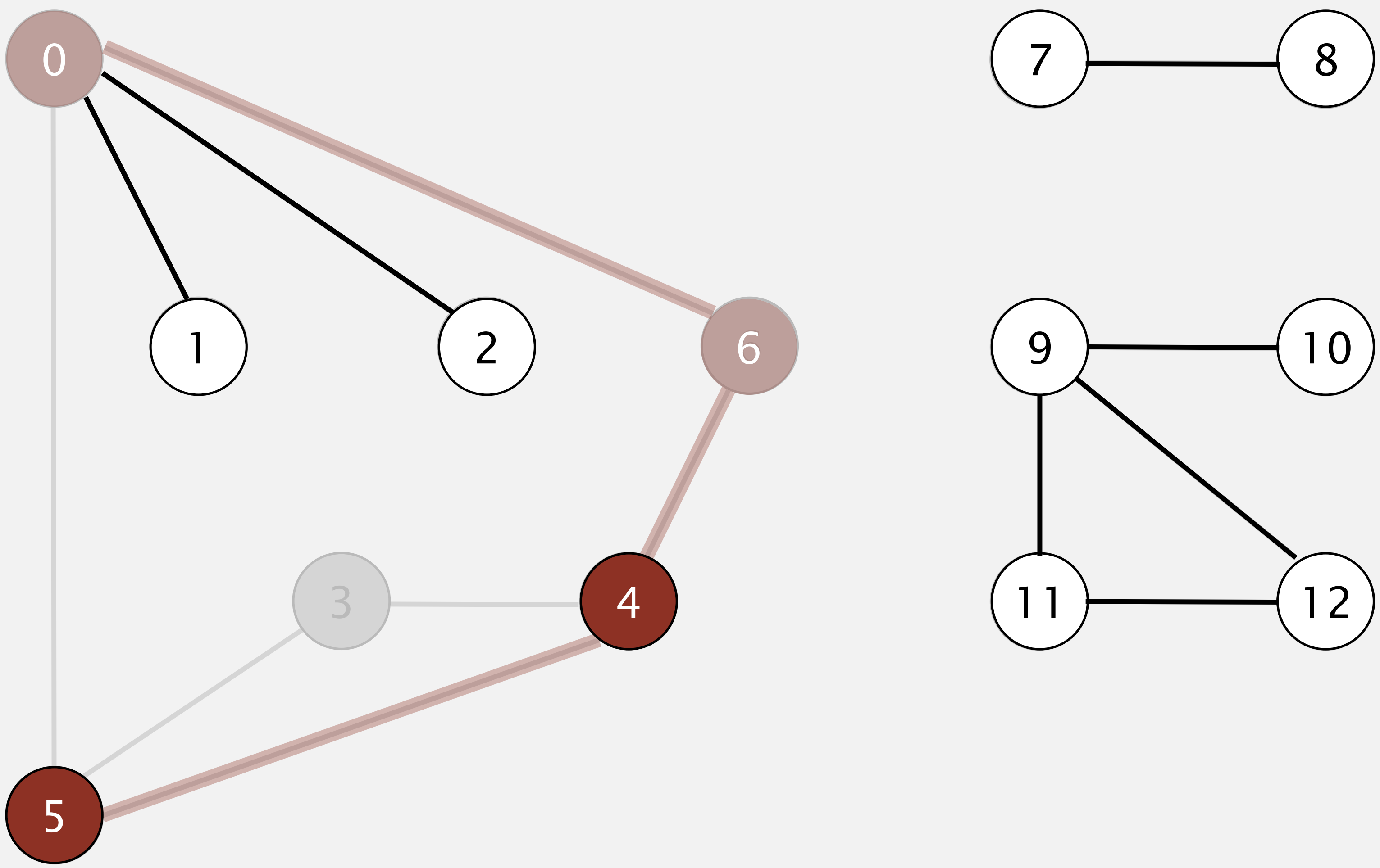
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



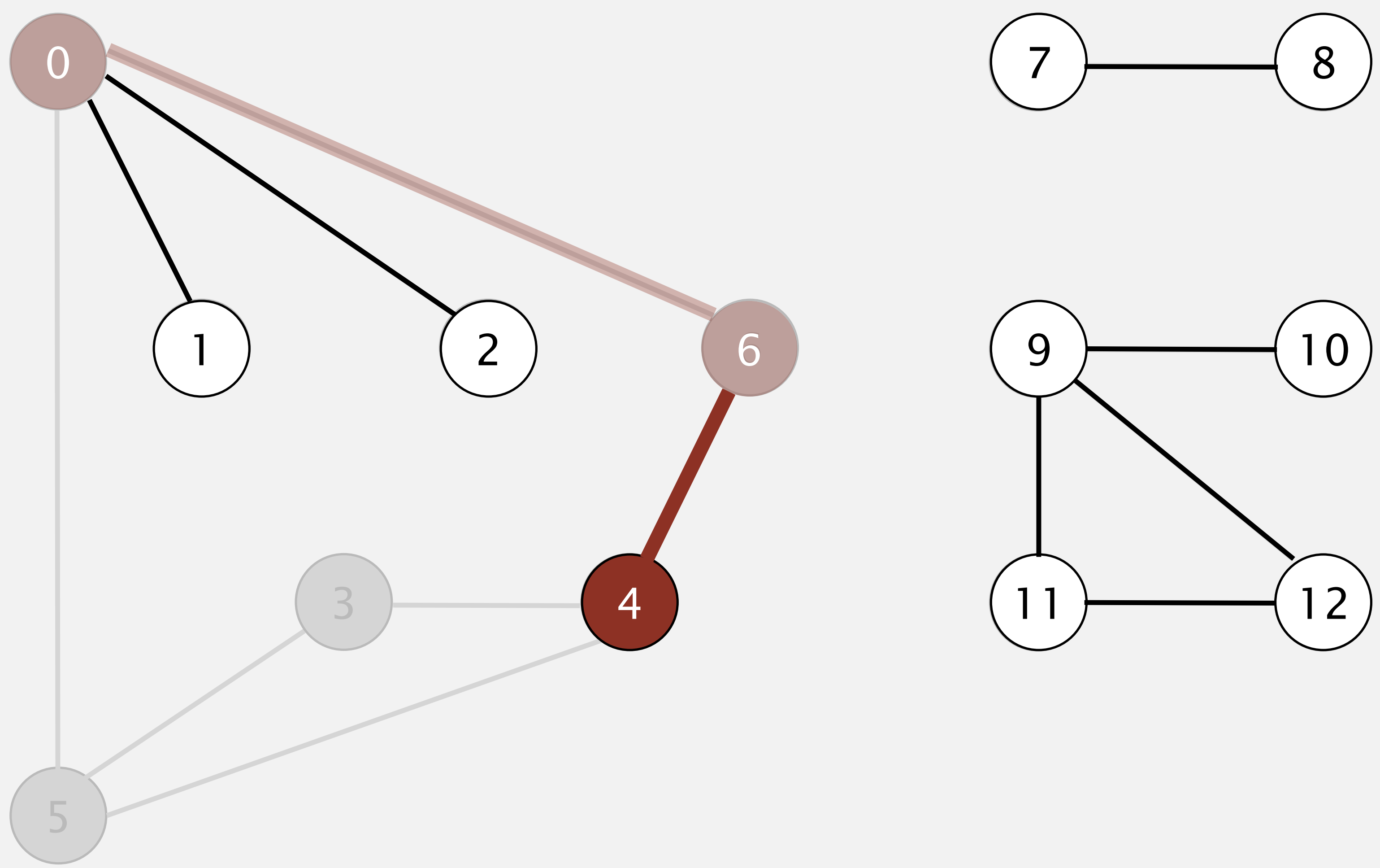
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

5 done

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



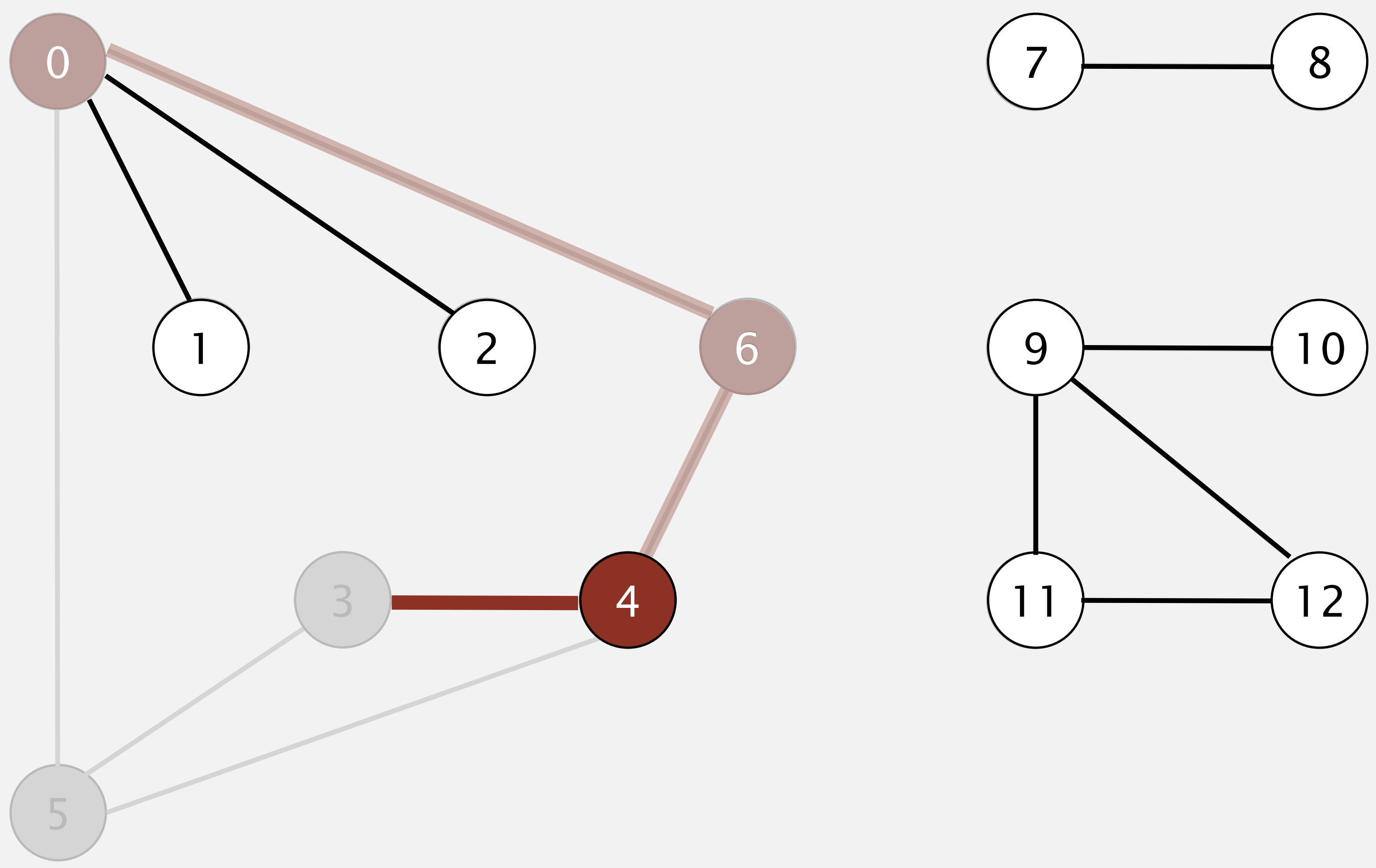
visit 4

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



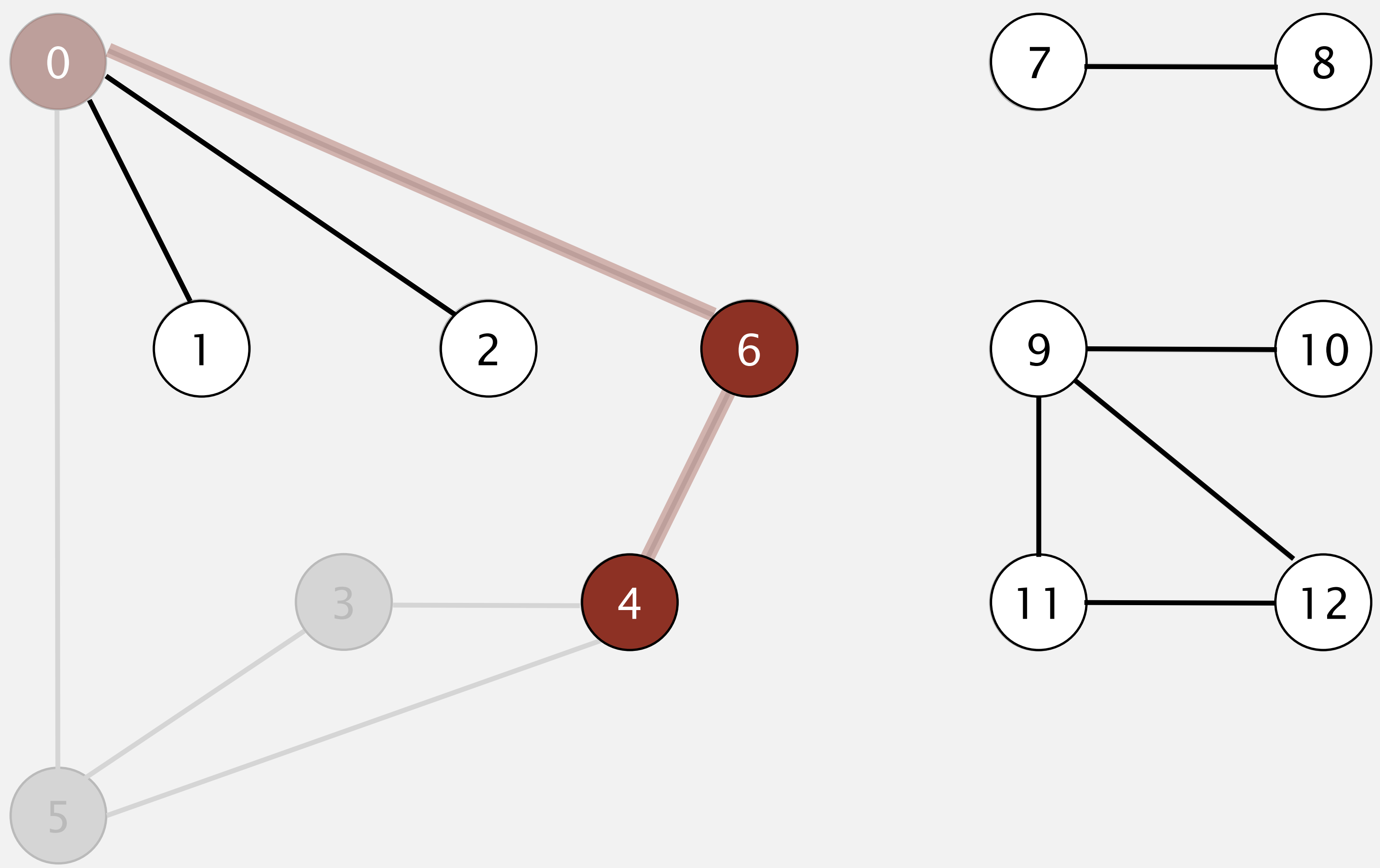
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



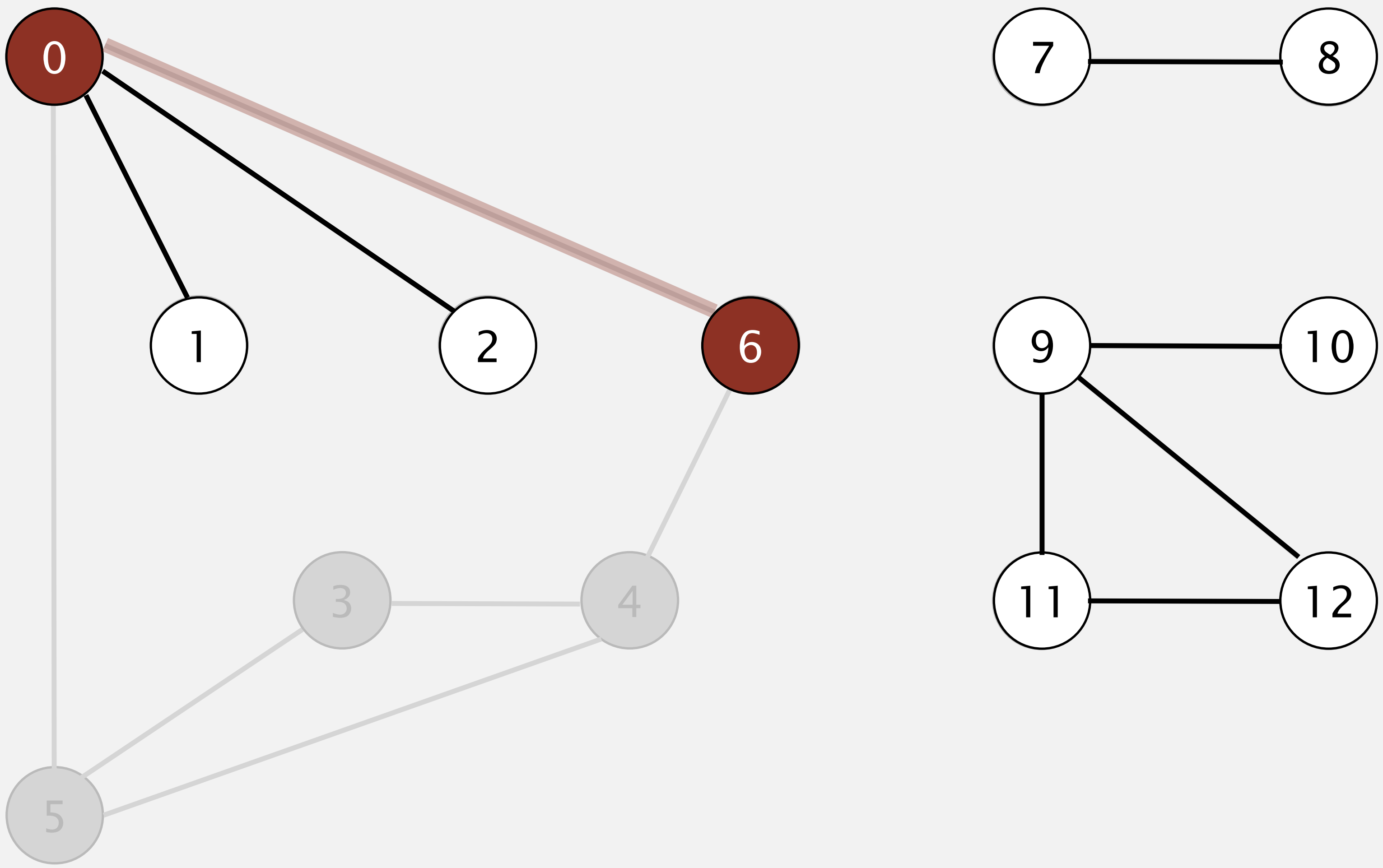
4 done

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



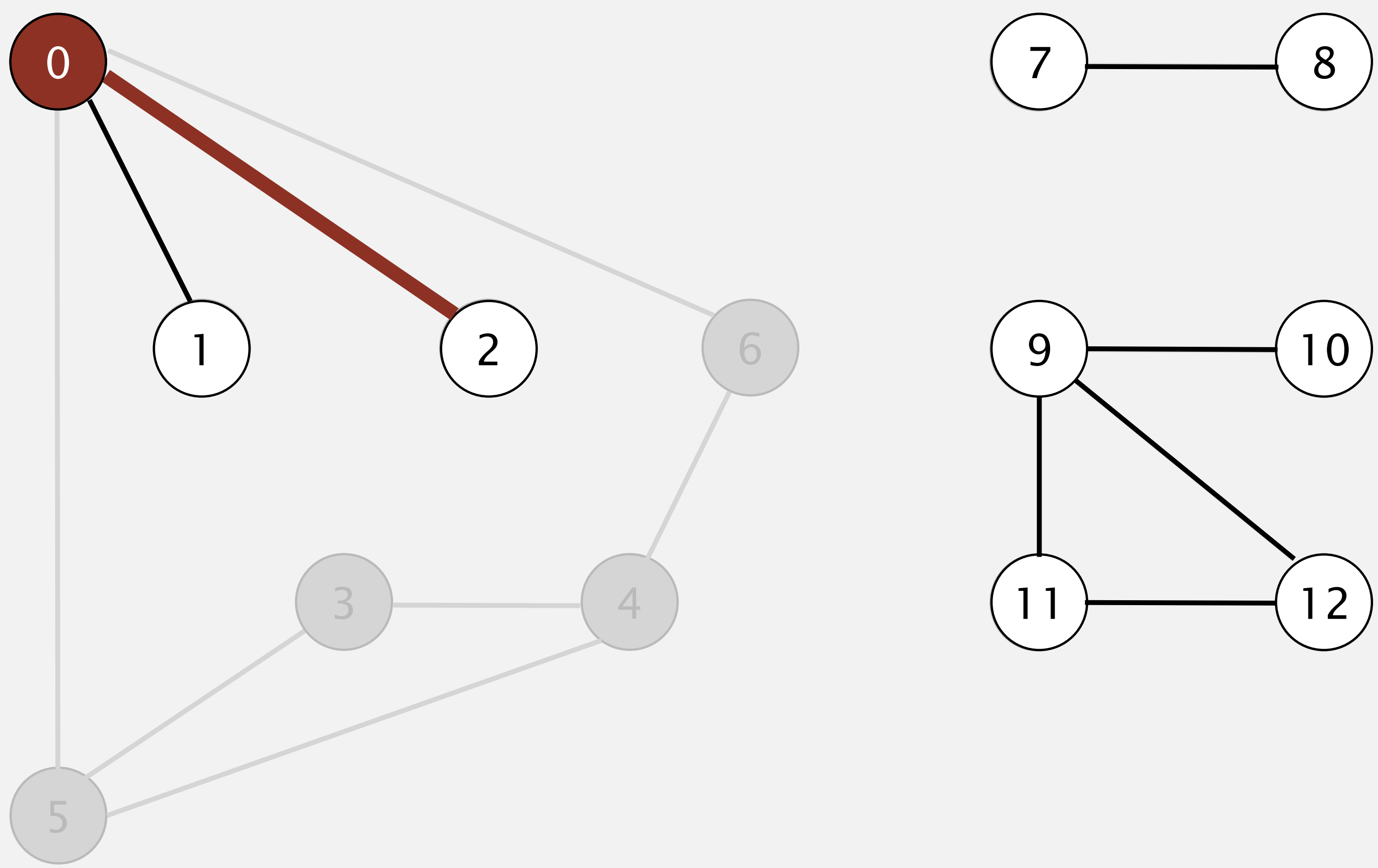
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

6 done

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



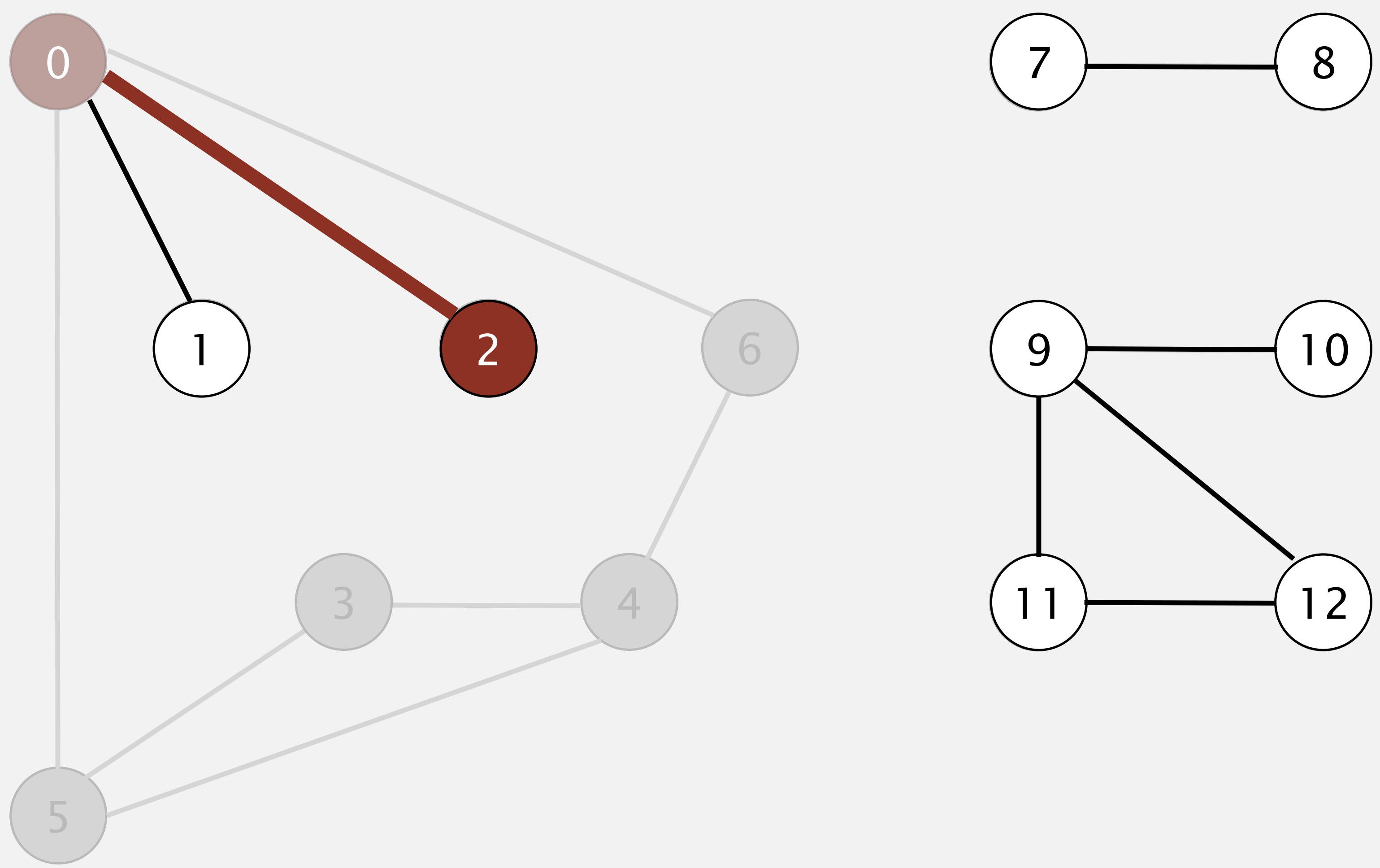
visit 0

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



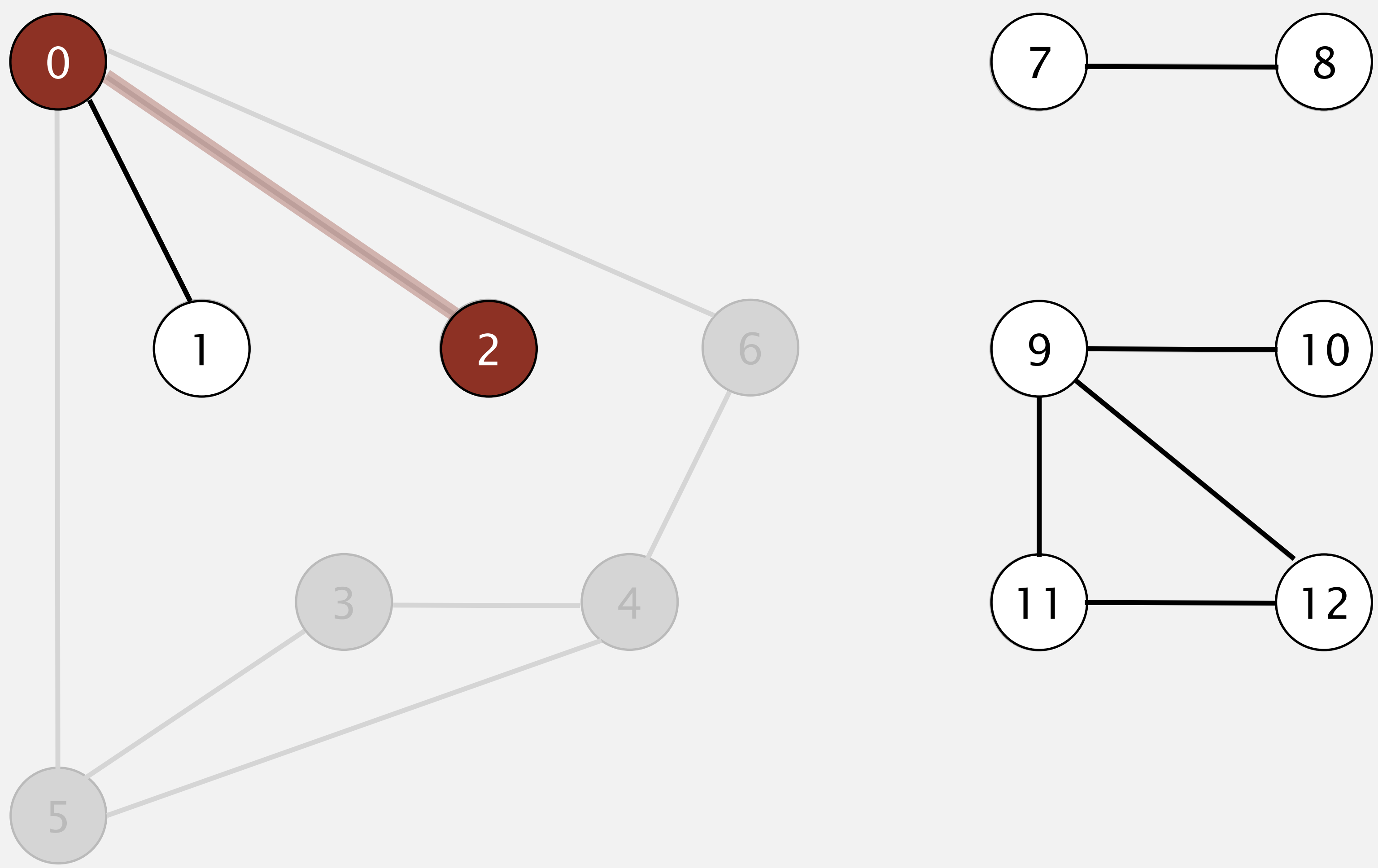
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



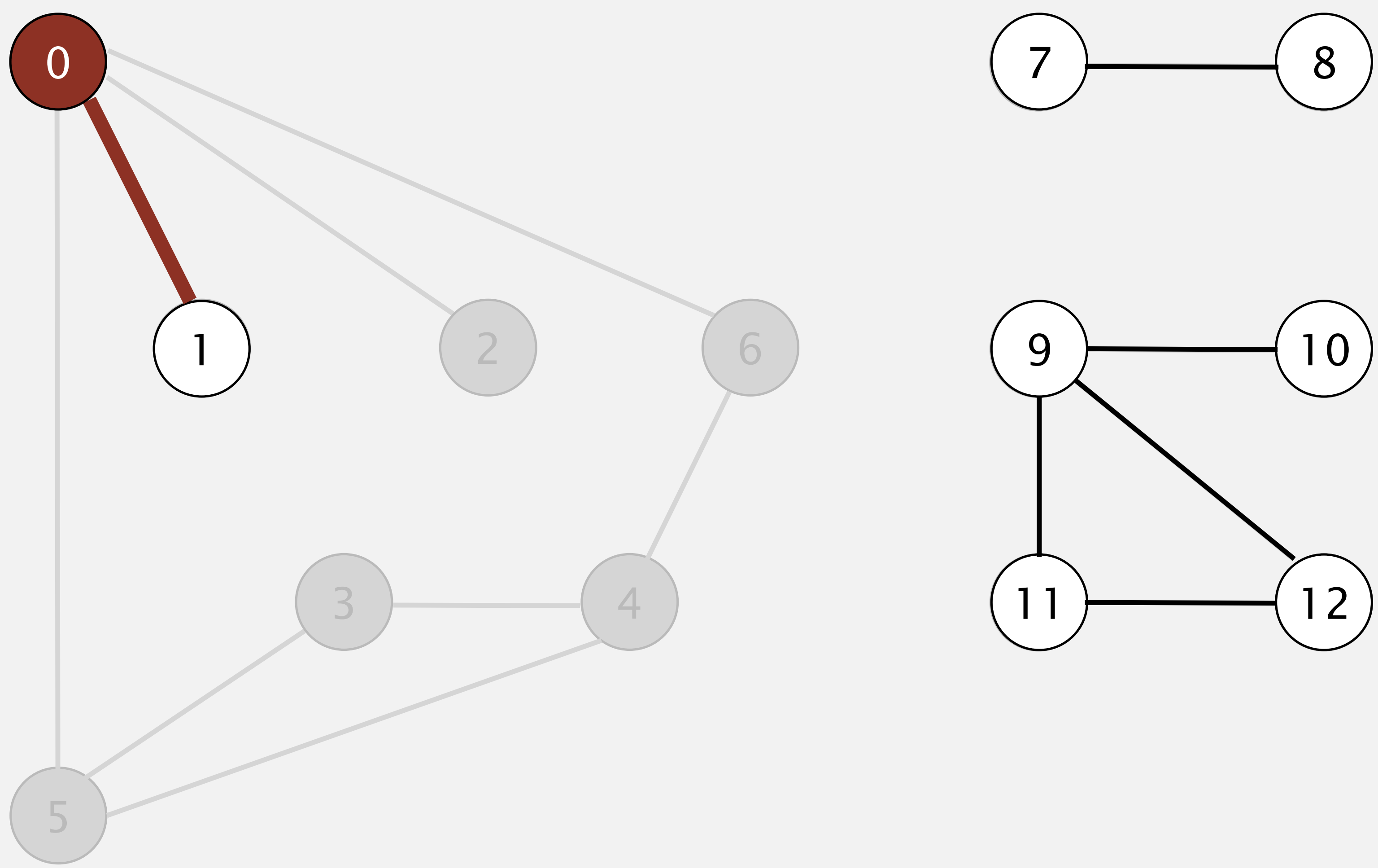
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

2 done

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



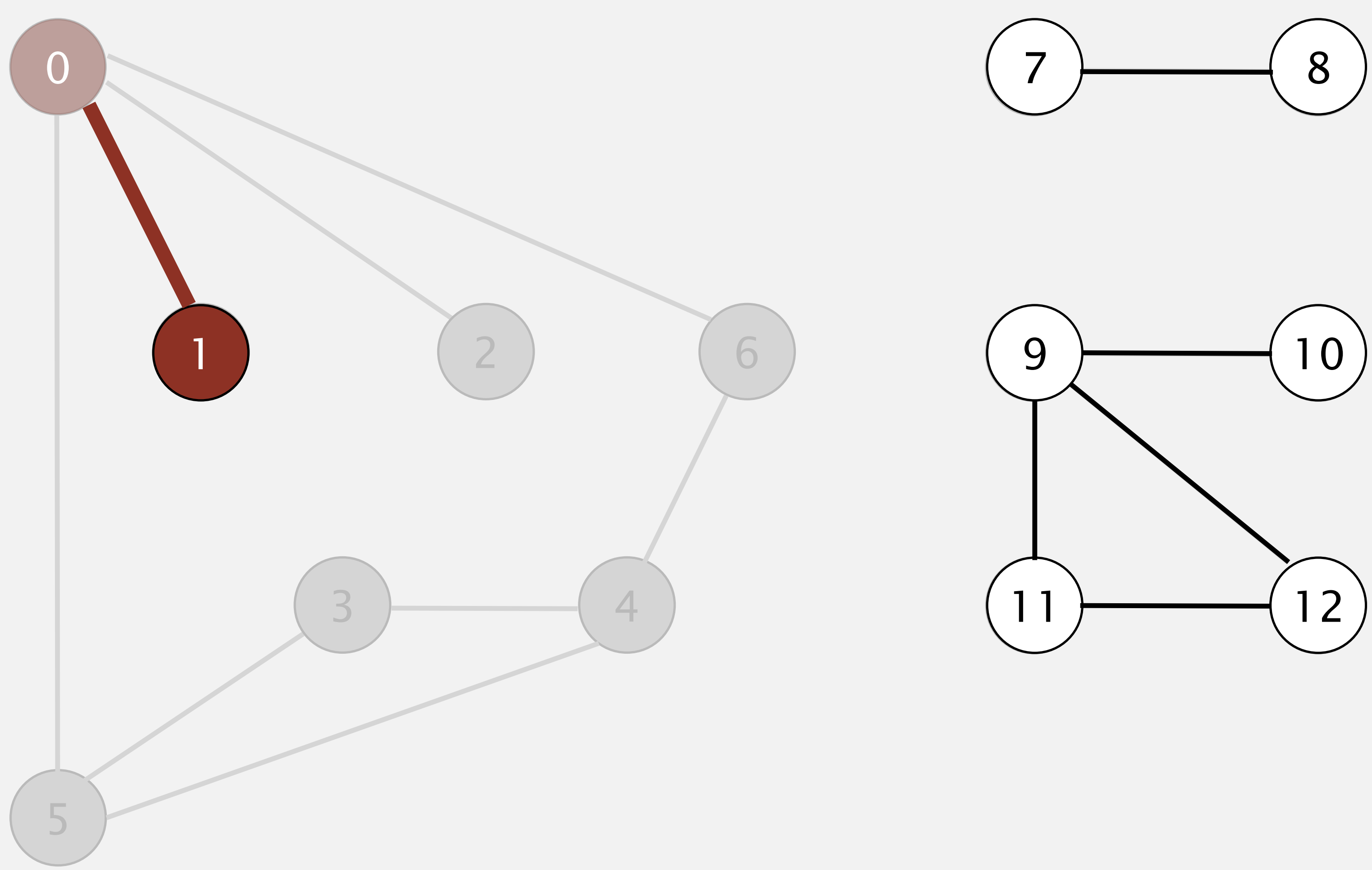
visit 0

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



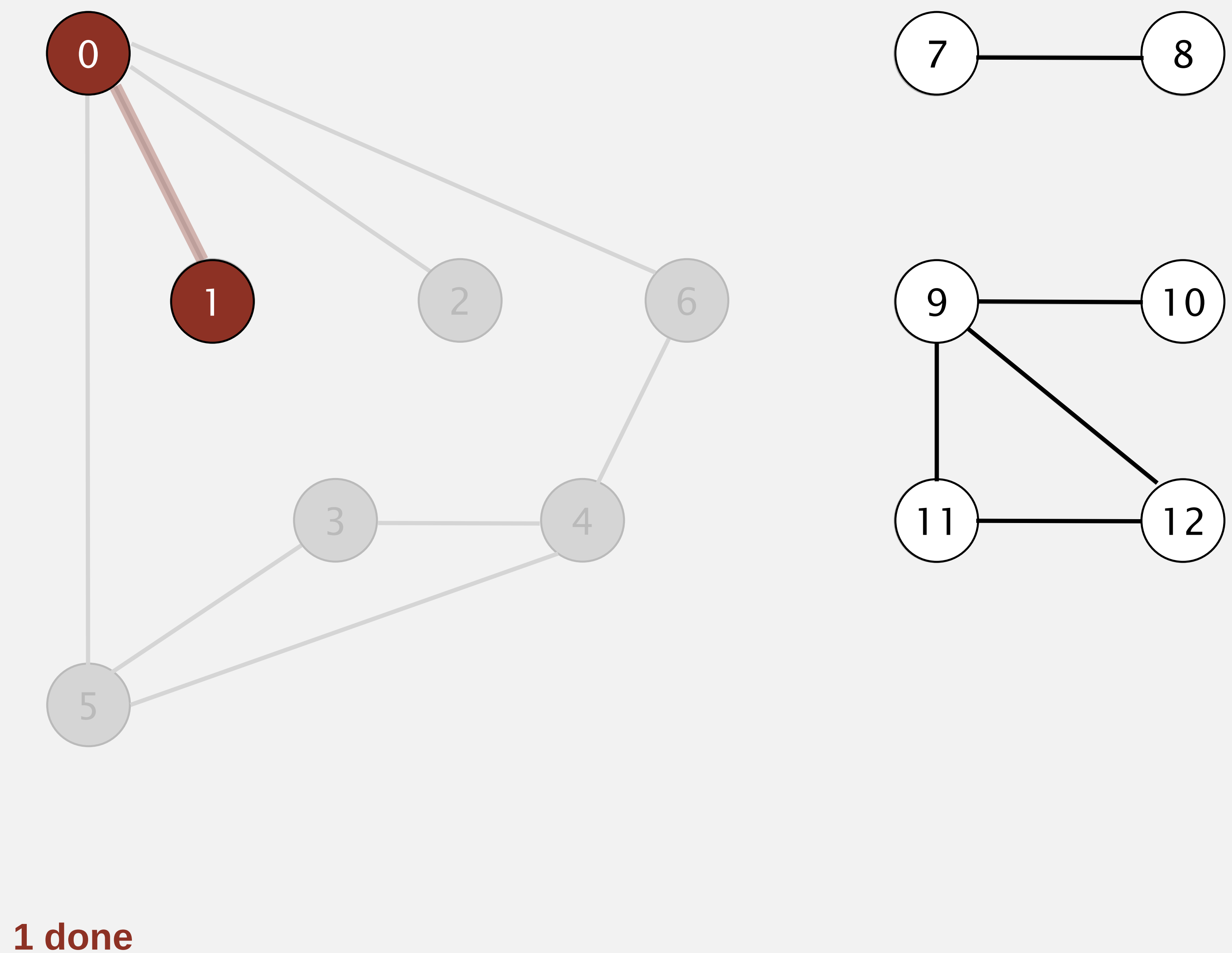
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 1

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

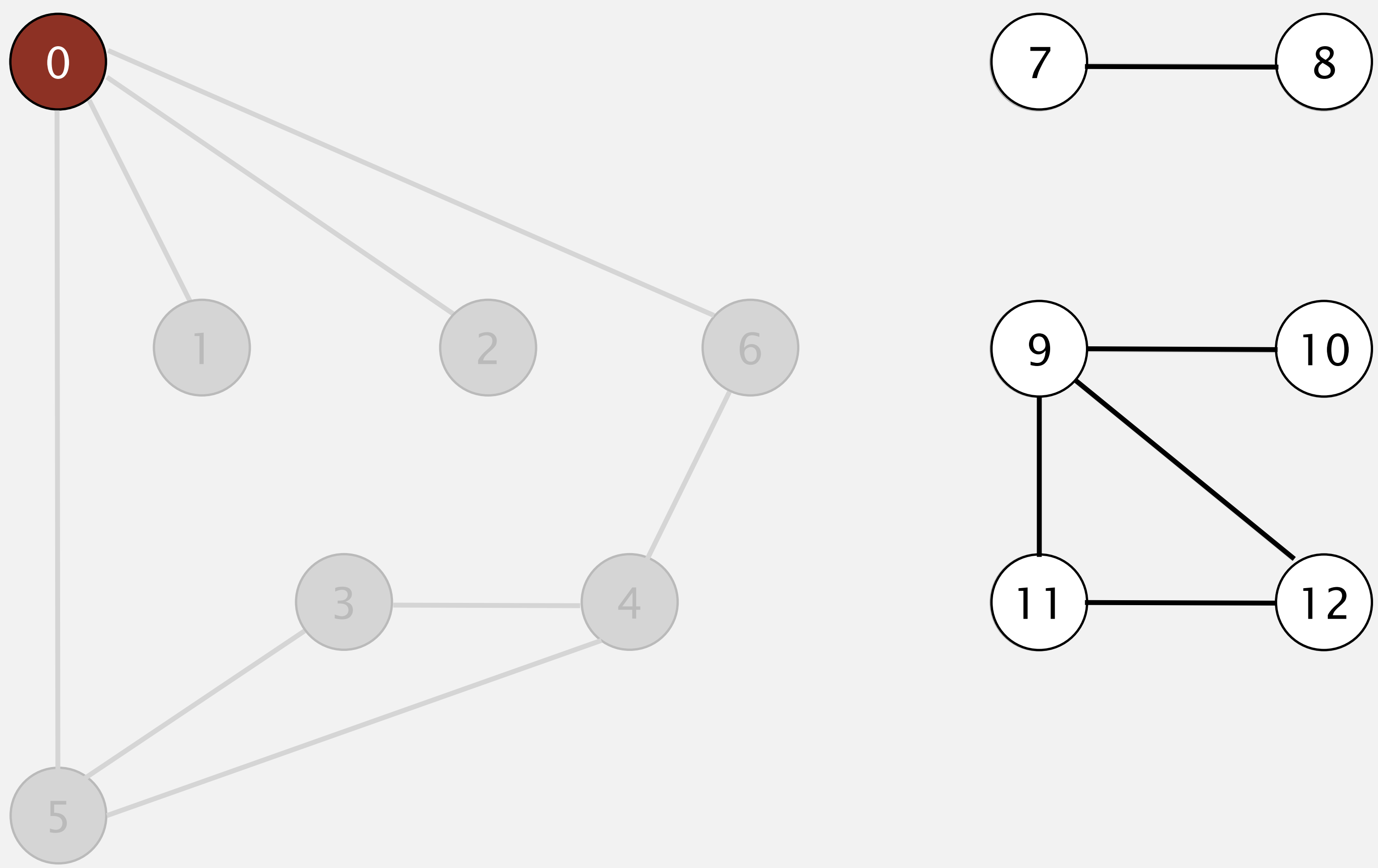


v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



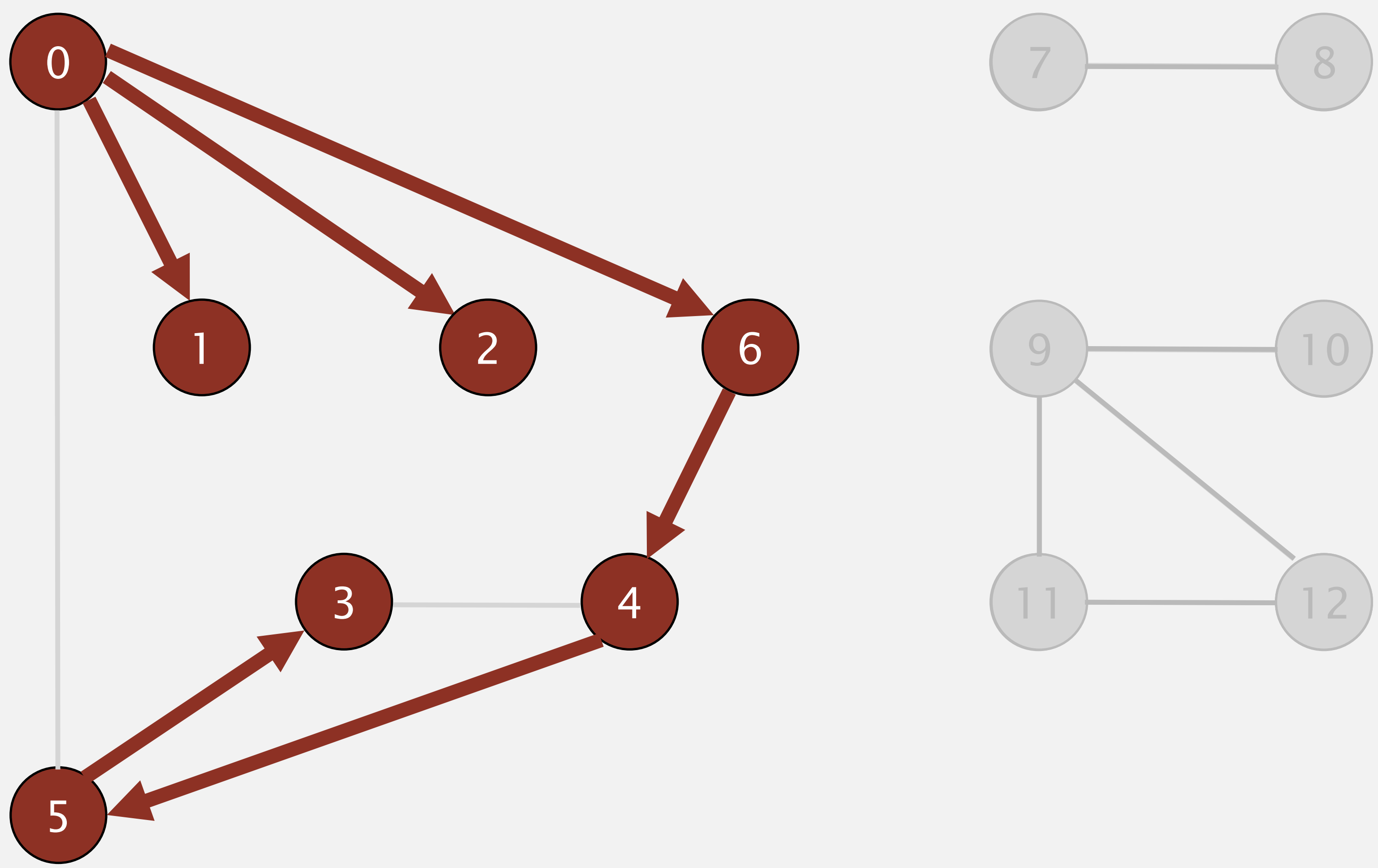
0 done

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



vertices reachable from 0

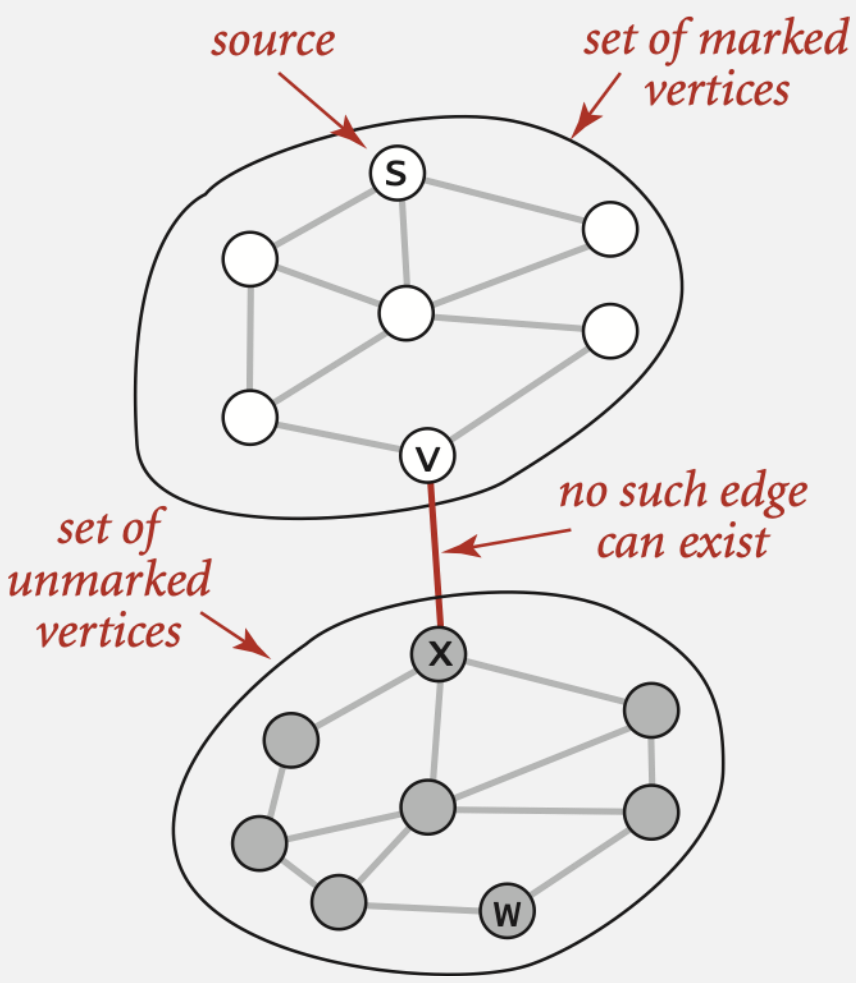
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search: properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the `marked[]` array).

Pf. [correctness]

- If w marked, then w connected to s (why?)
- If w connected to s, then w marked.
  (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one).



Pf. [running time]

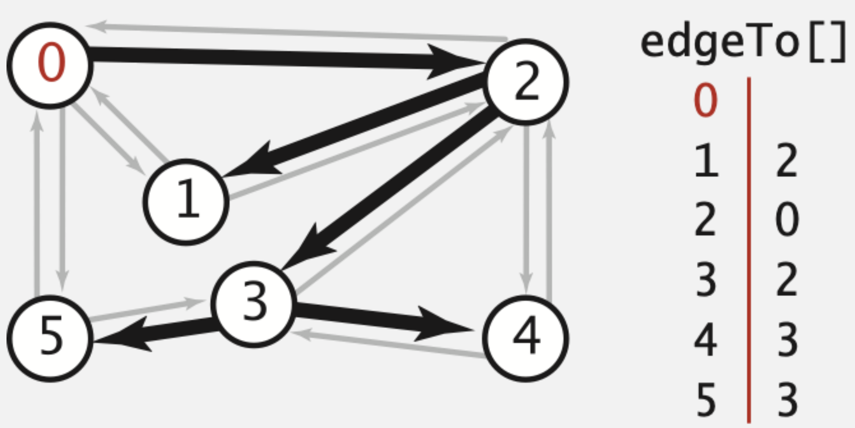
Each vertex connected to s is visited once.

Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find v–s path (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is parent-link representation of a tree rooted at vertex `s`.

```
public boolean hasPathTo(int v)
{  return marked[v];  }

public Iterable<Integer> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new tack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push(x);
   path.push(s);
   return path;
}
```



http://algs4.cs.princeton.edu

Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *connected components*
- *challenges*

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.

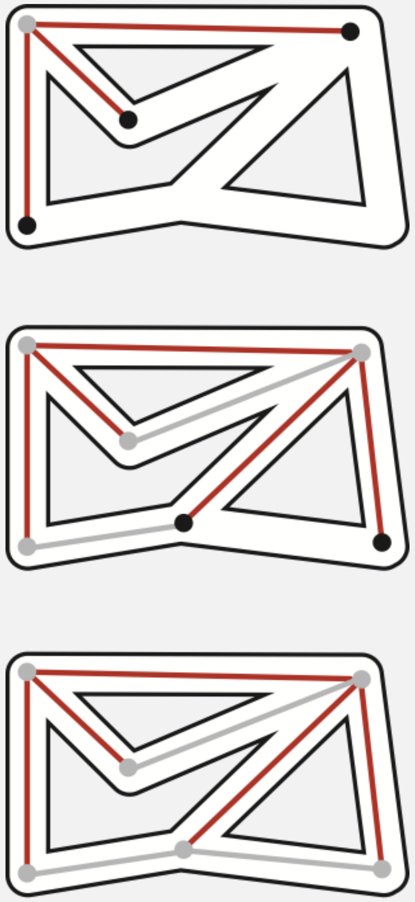
BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v

- add each of v's unvisited neighbors to the queue, and mark them as visited.



Breadth-first search: Java implementation

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;

   …
   private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      distTo[s] = 0;

      while (!q.isEmpty()) {
         int v = q.dequeue();
         for (int w : G.adj(v)) {
            if (!marked[w]) {
               q.enqueue(w);
               marked[w] = true;
               edgeTo[w] = v;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
}
```

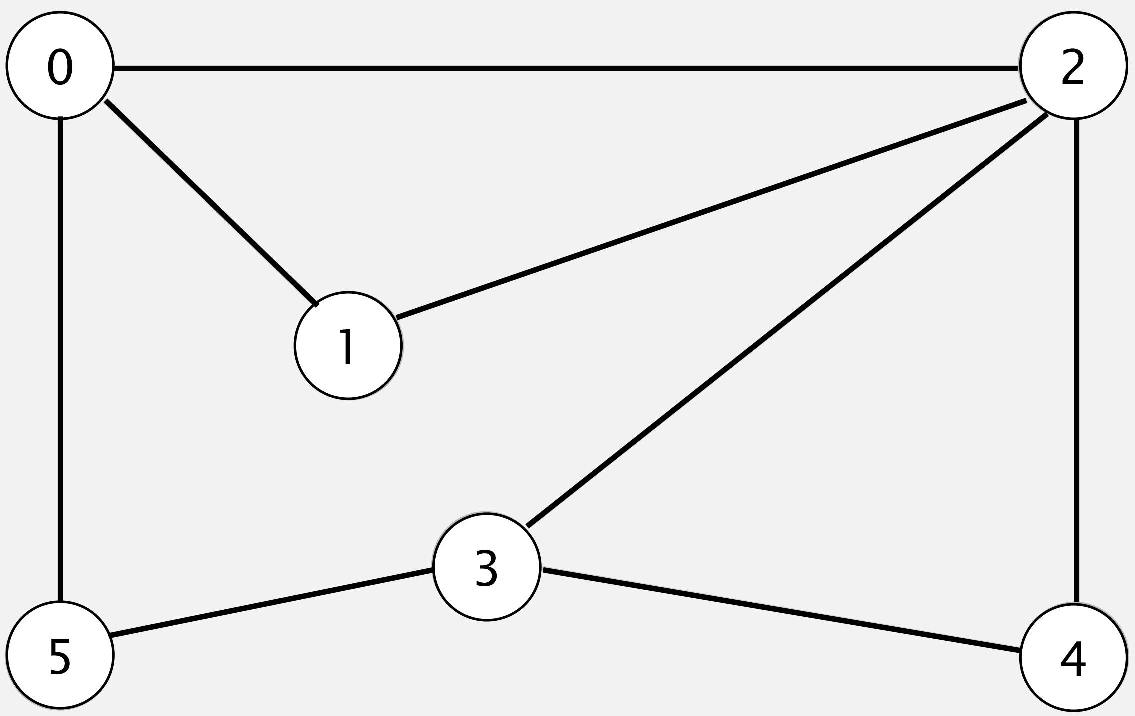
initialize FIFO queue of vertices to explore

found new vertex w via edge v-w

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



graph G

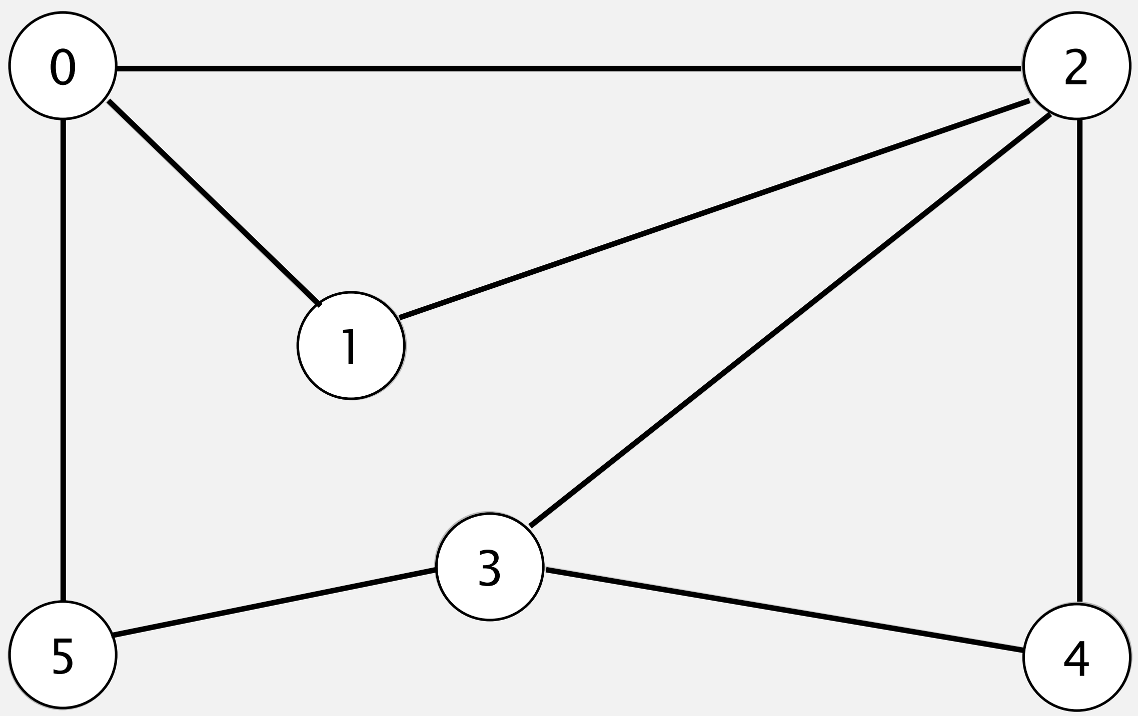
tinyCG.txt

V 6
8 E
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

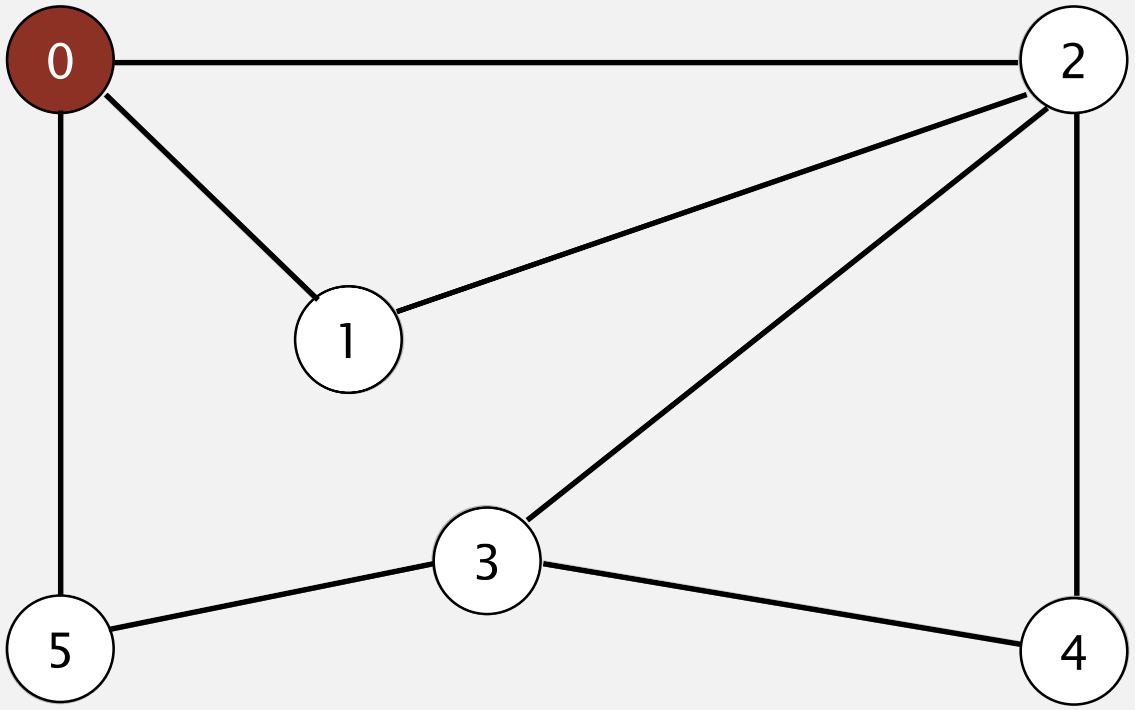
v	edgeTo[]	distTo[]
0	–	0
1	–	–
2	–	–
3	–	–
4	–	–
5	–	–

add 0 to queue

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
0

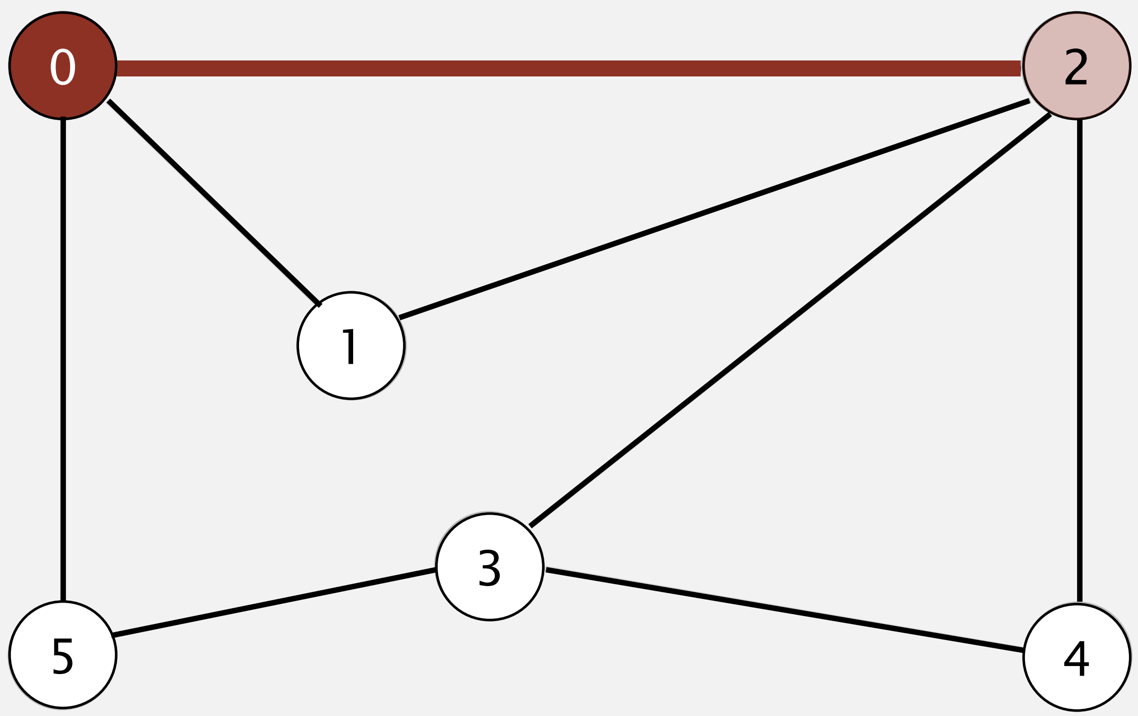
v	edgeTo[]	distTo[]
0	–	0
1	–	–
2	–	–
3	–	–
4	–	–
5	–	–

dequeue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

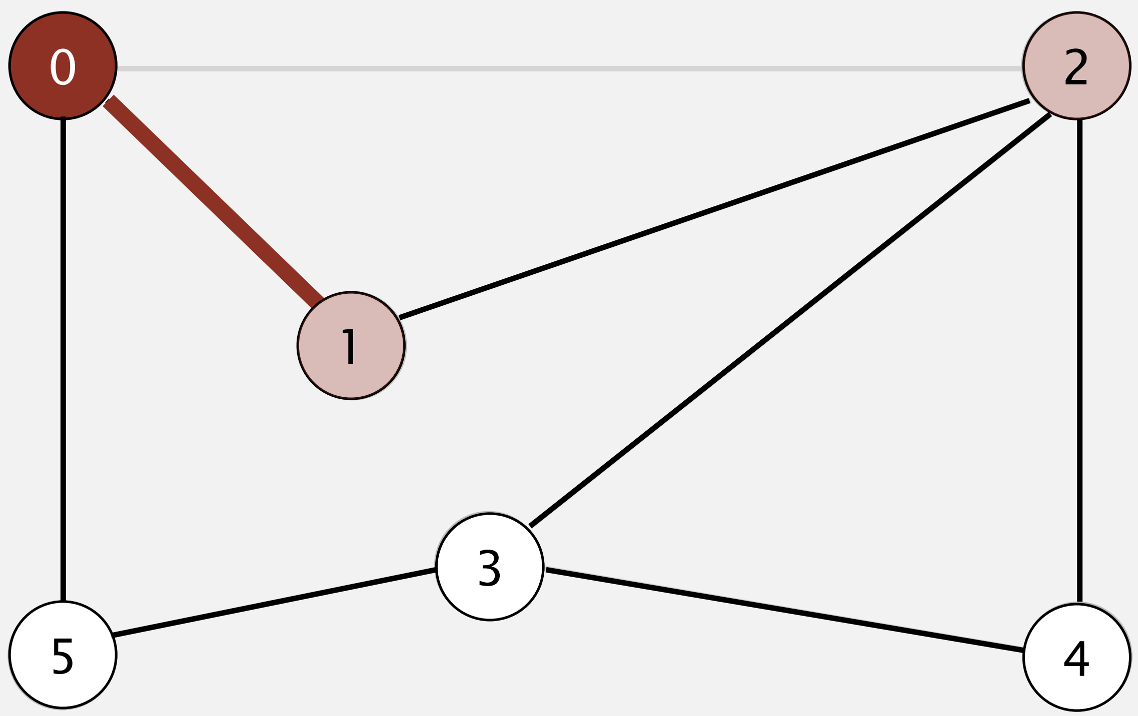
v	edgeTo[]	distTo[]
0	–	0
1		
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

2

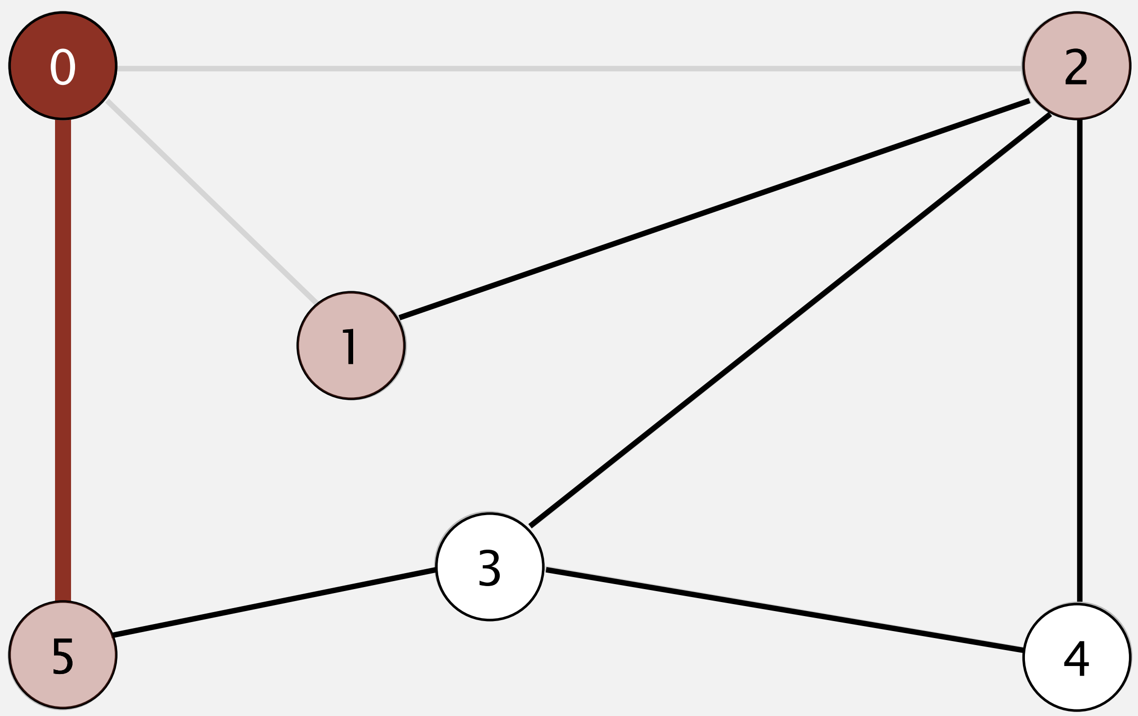
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

1
2

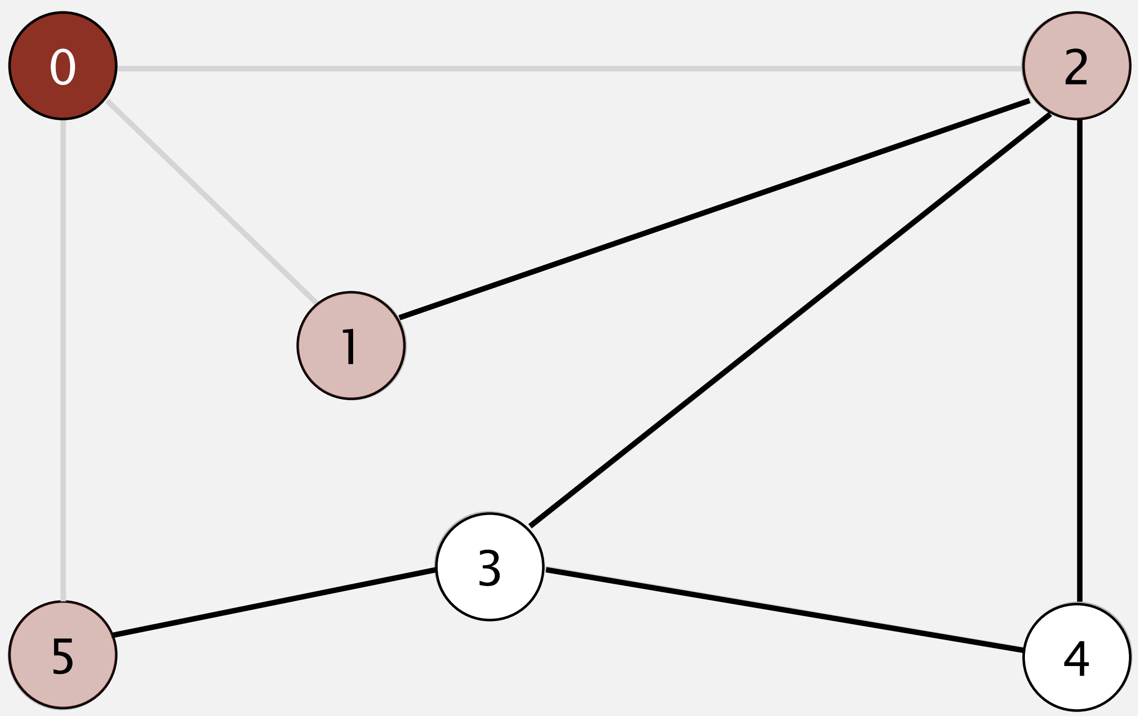
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	0	1
5	–	–

dequeue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

5
1
2

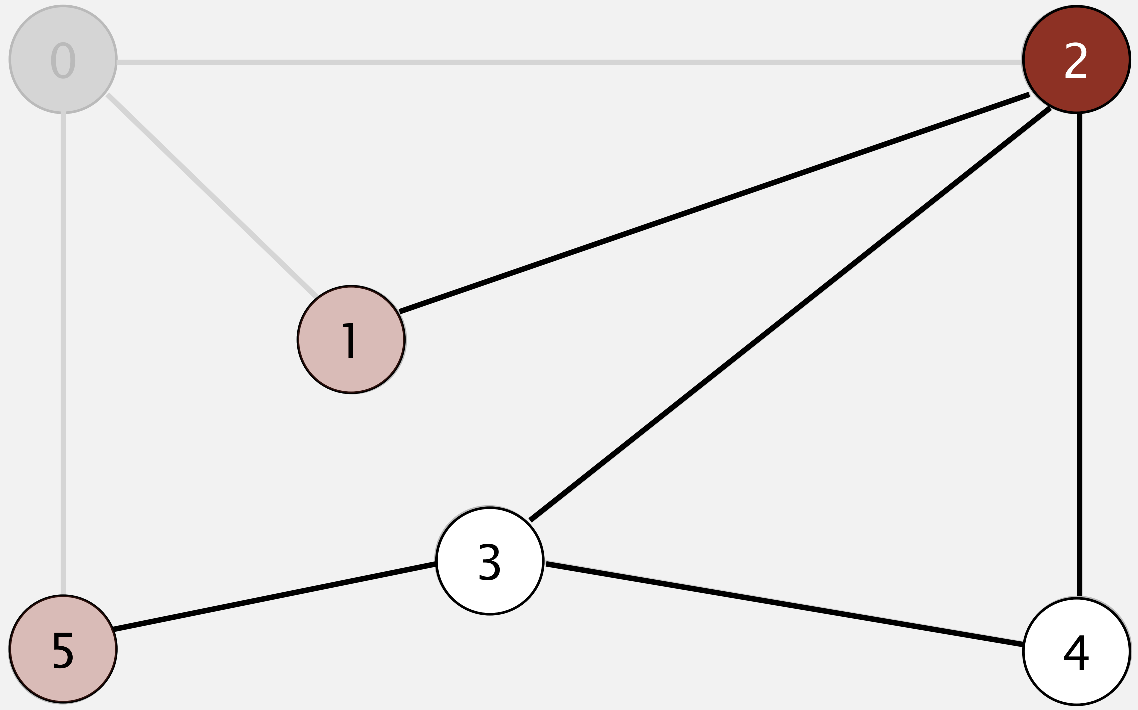
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	0	1

0 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
5
1
2

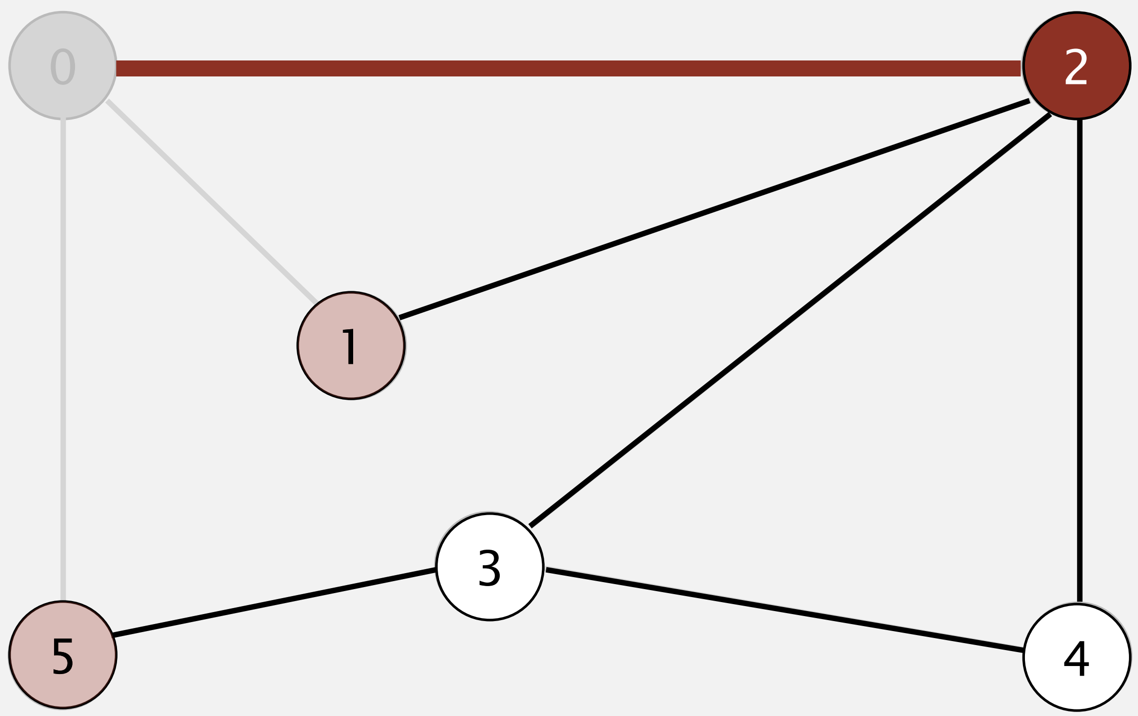
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	0	1

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

5
1

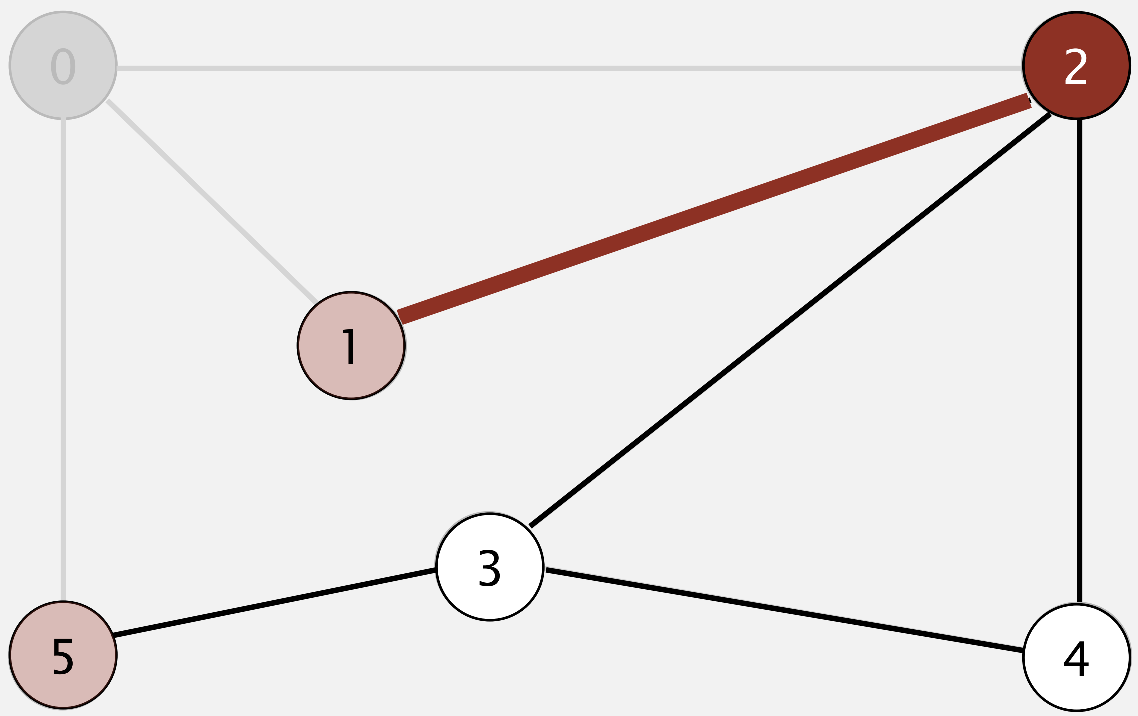
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	0	1

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue
5
1

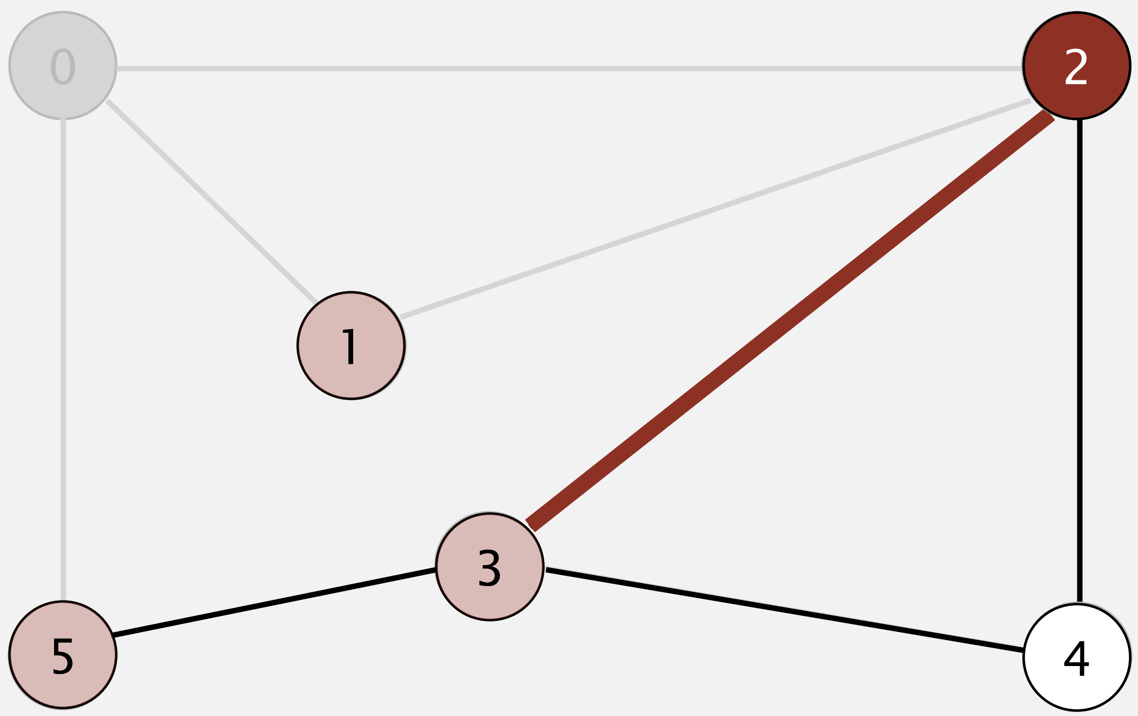
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	0	1

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
5
1

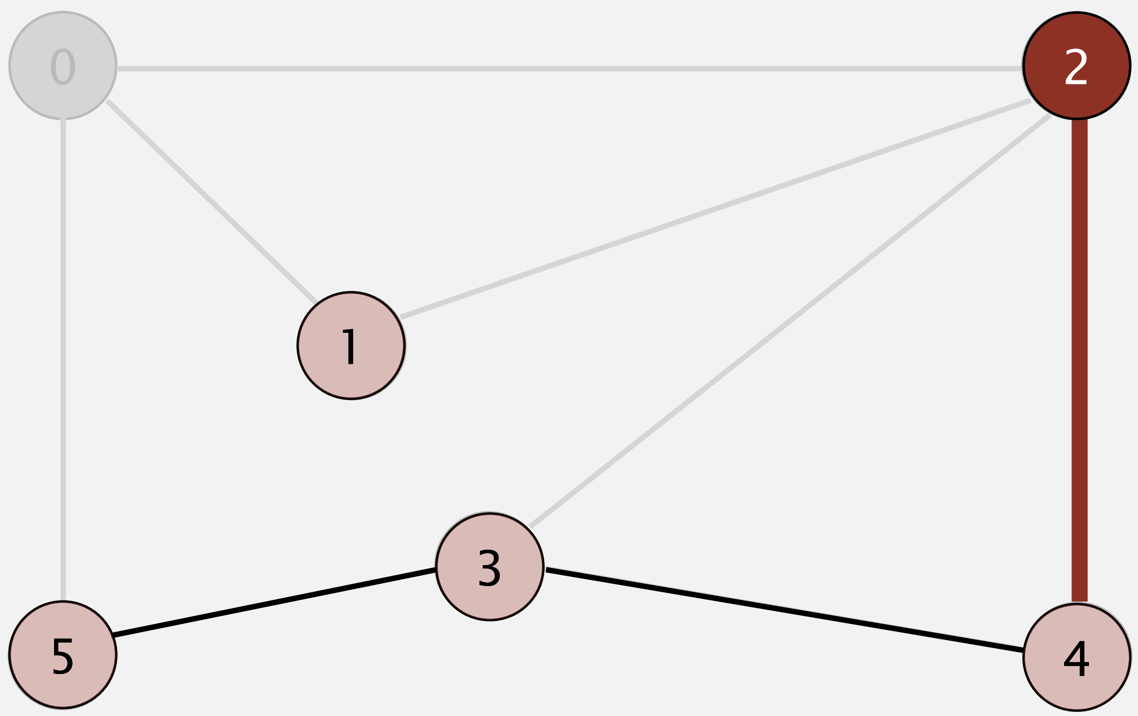
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	2	2
3	–	–
4	–	–
5	0	1

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
3
5
1

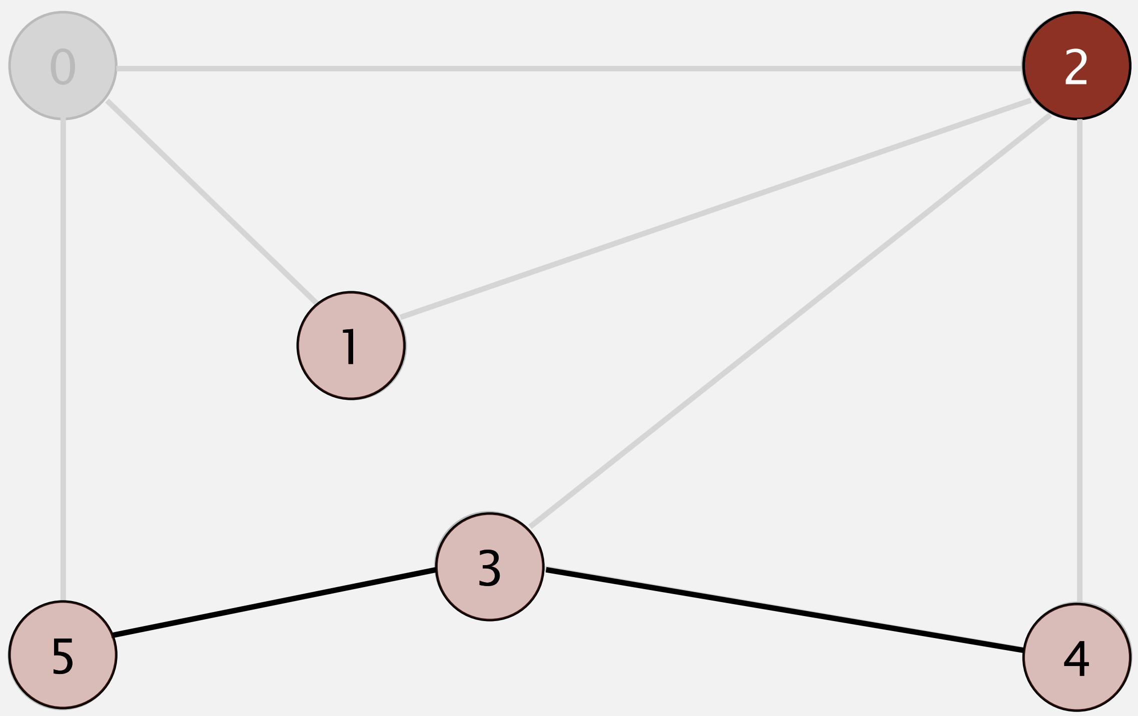
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	–	–
5	0	1

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
4
3
5
1

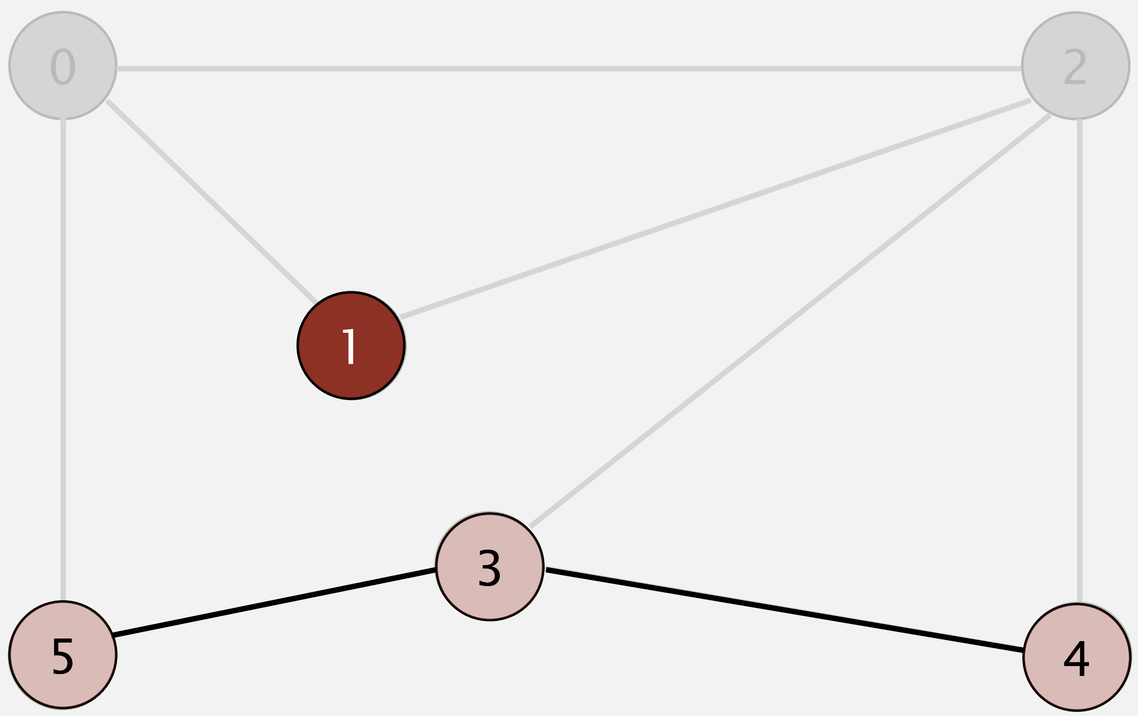
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

2 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
4
3
5
1

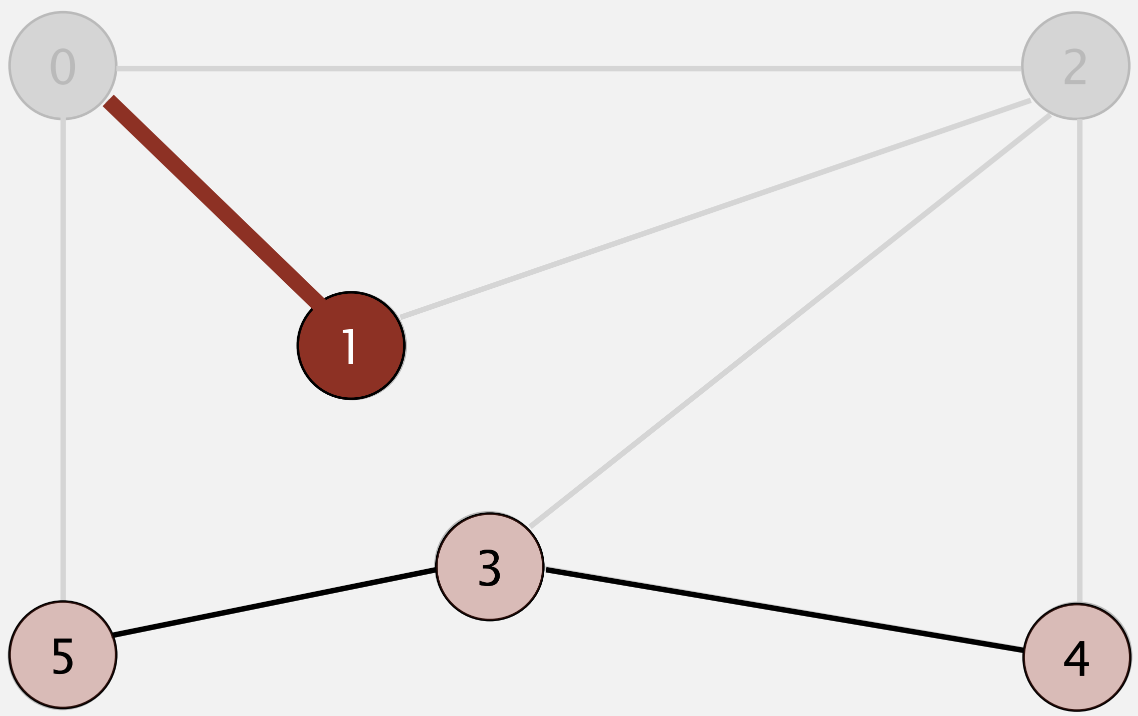
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 1

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5

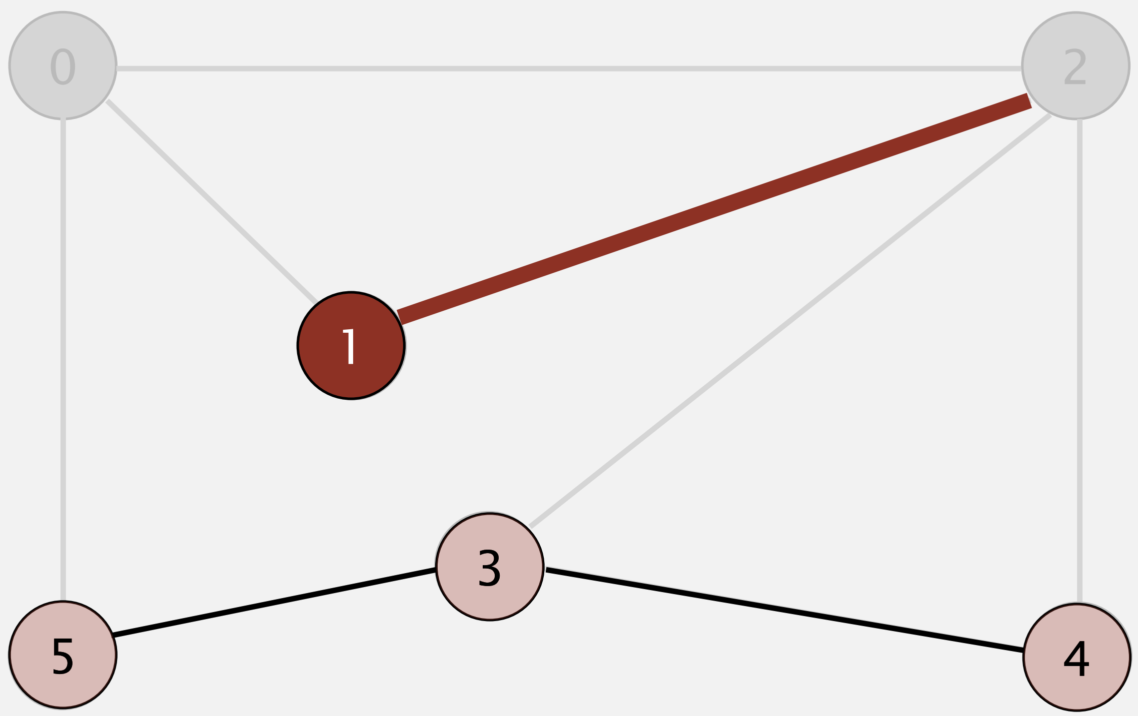
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 1

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

4
3
5

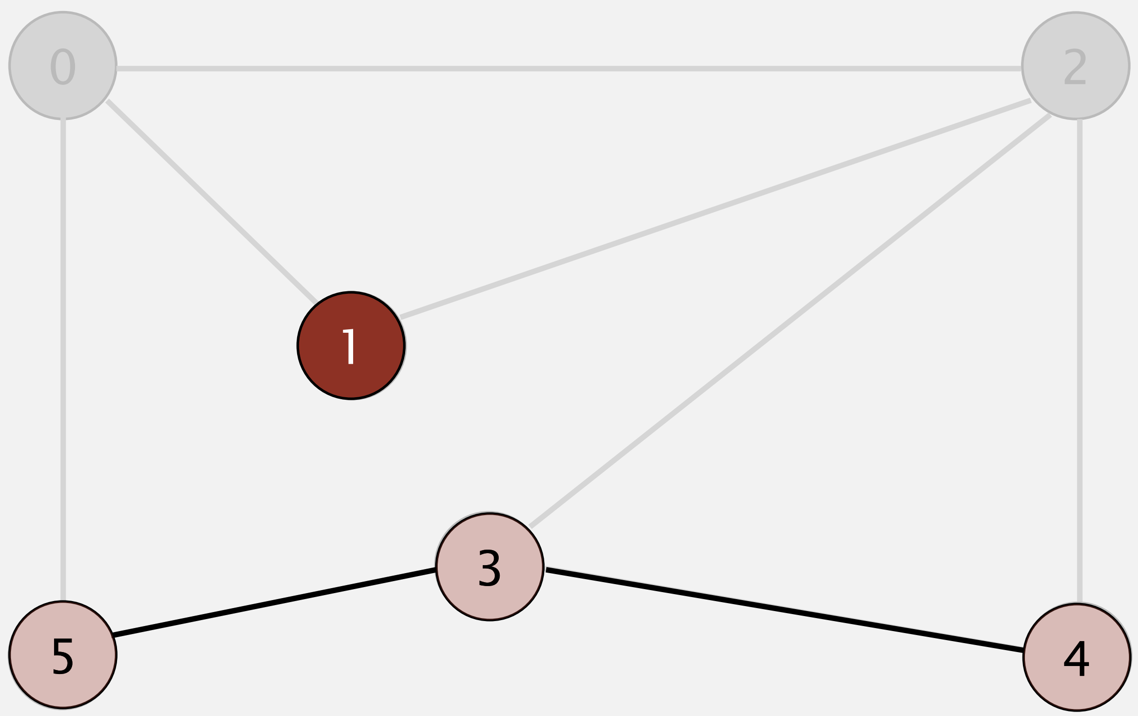
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 1

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5

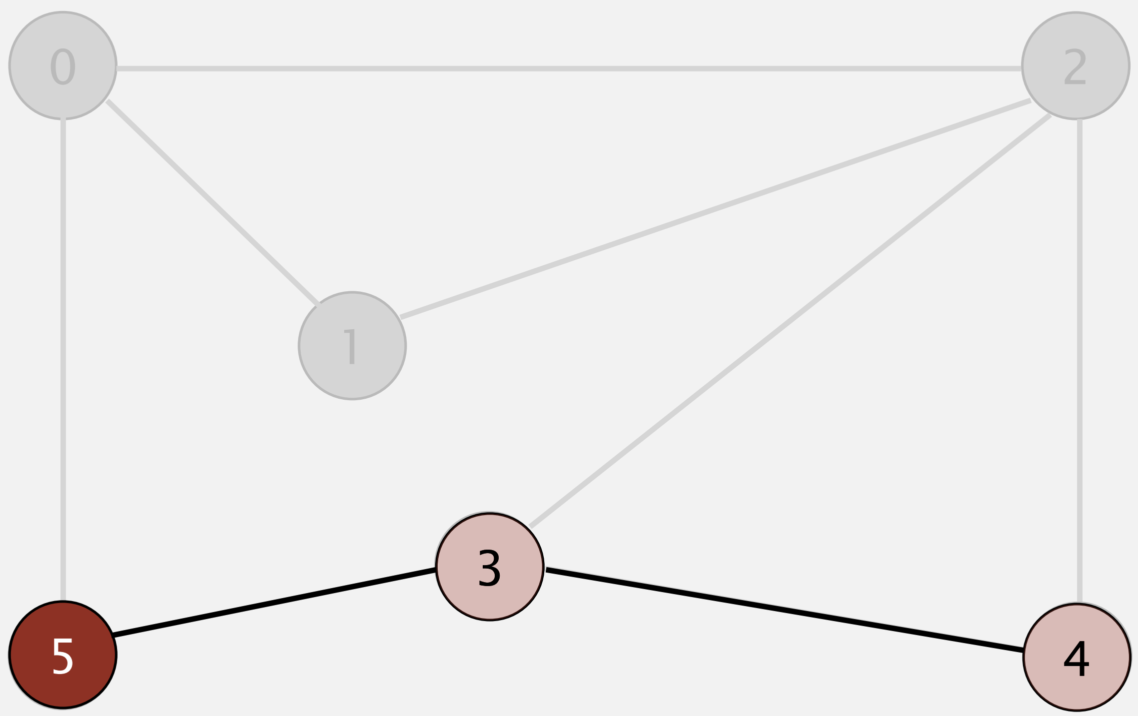
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

1 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

4
3
5

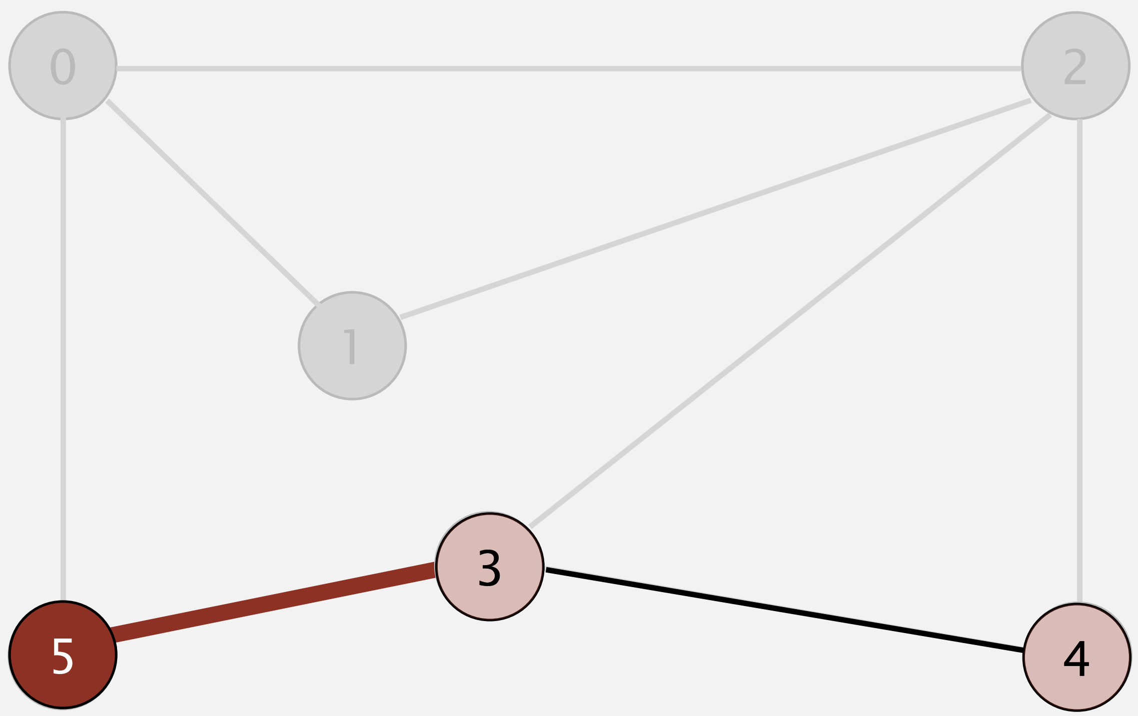
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue
4
3

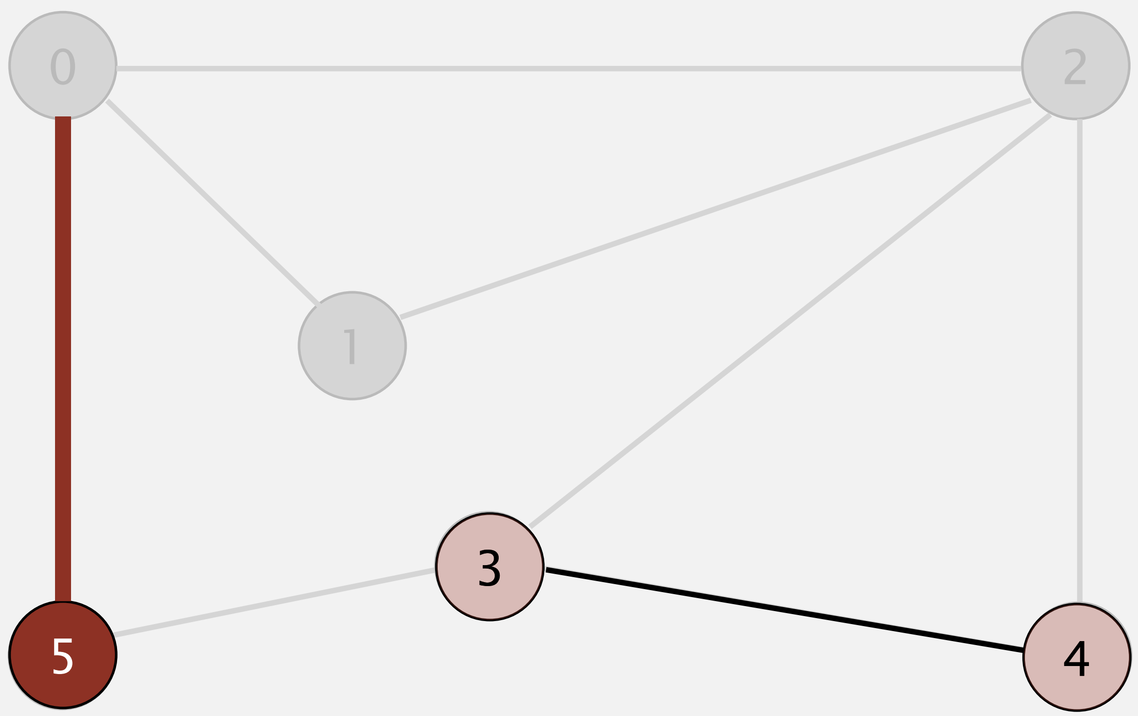
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue
4
3

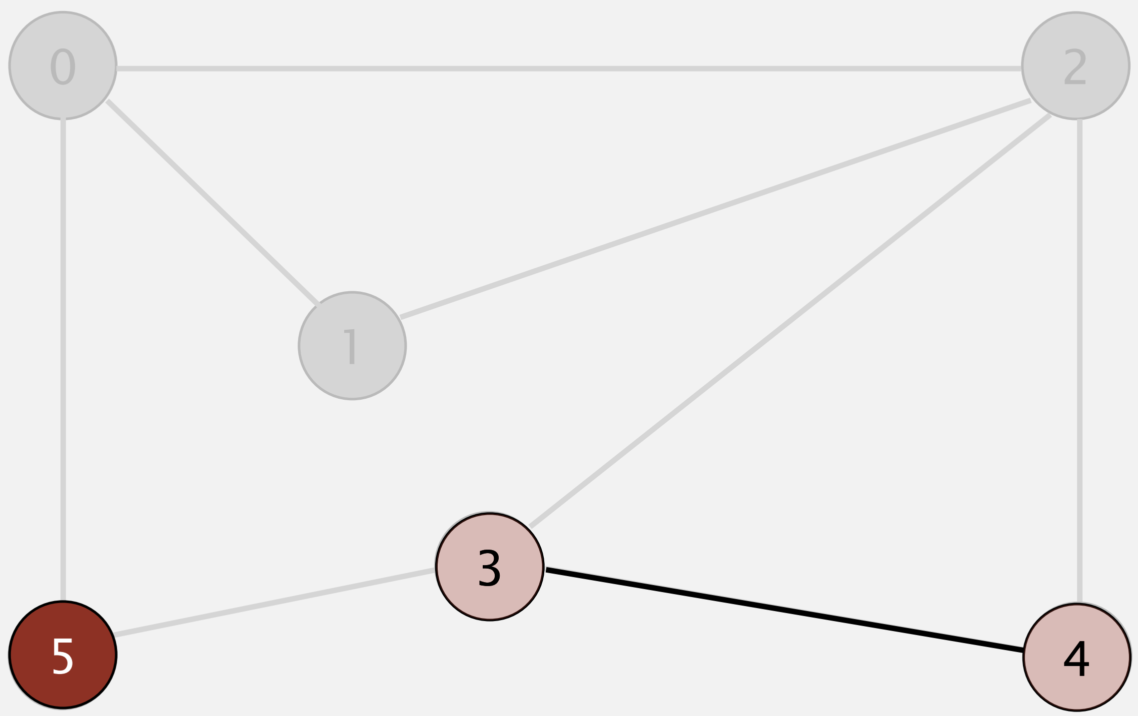
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

4
3

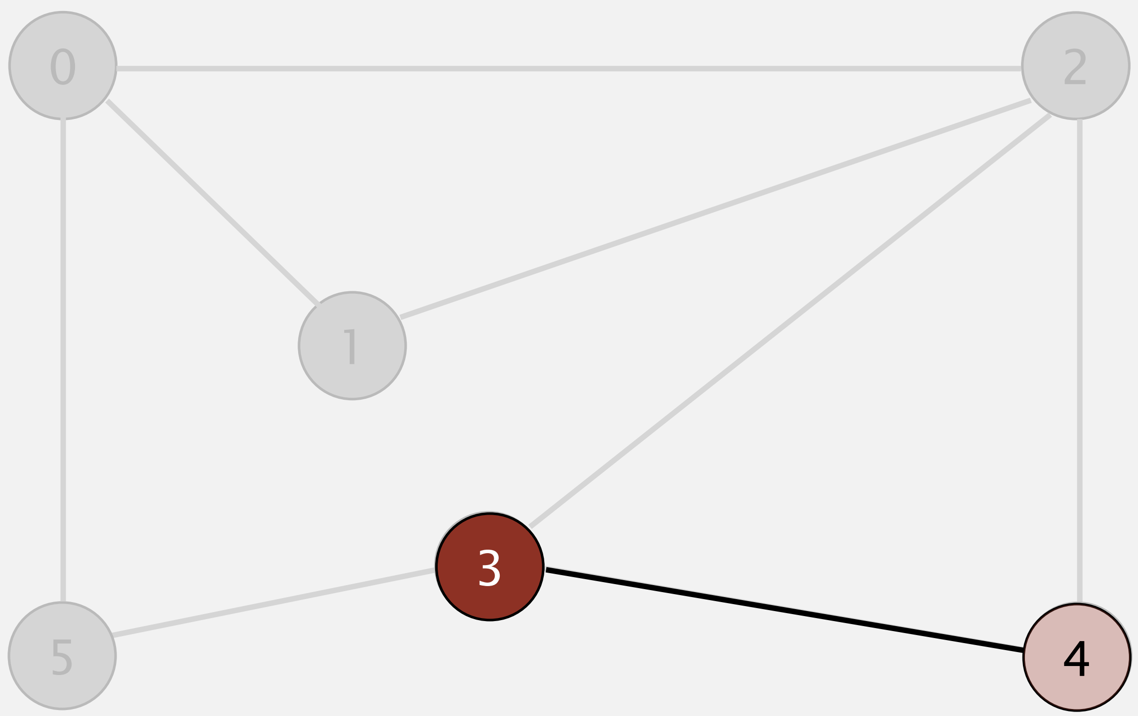
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

5 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
4
3

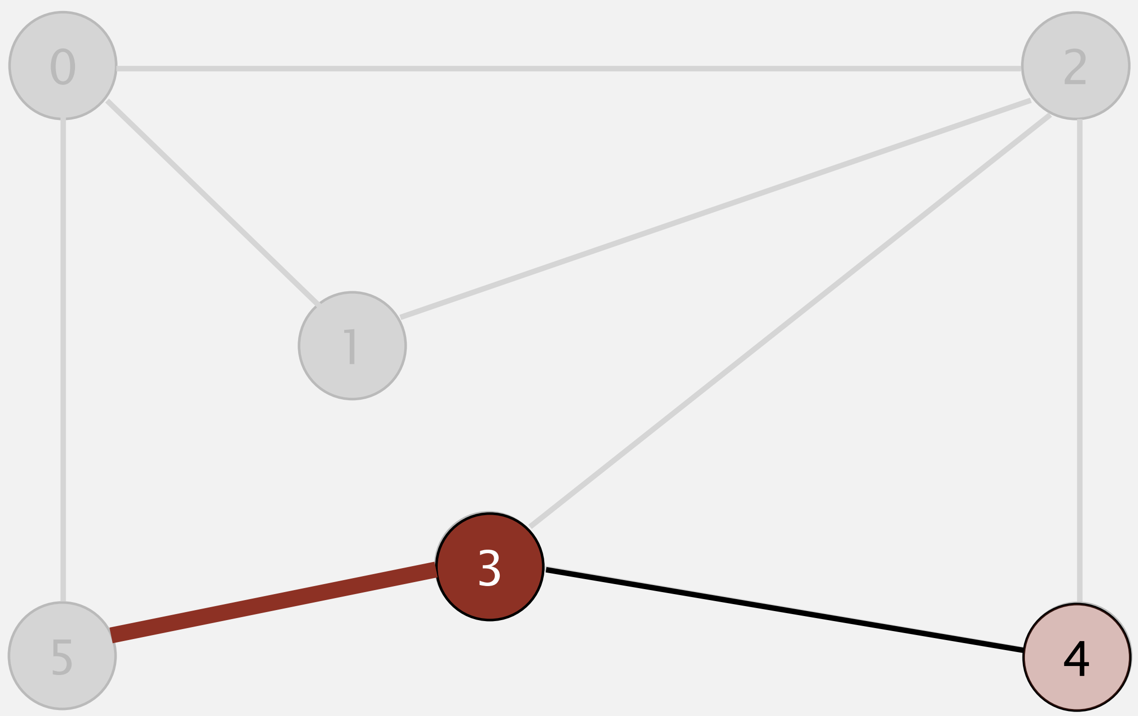
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 3

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue
4

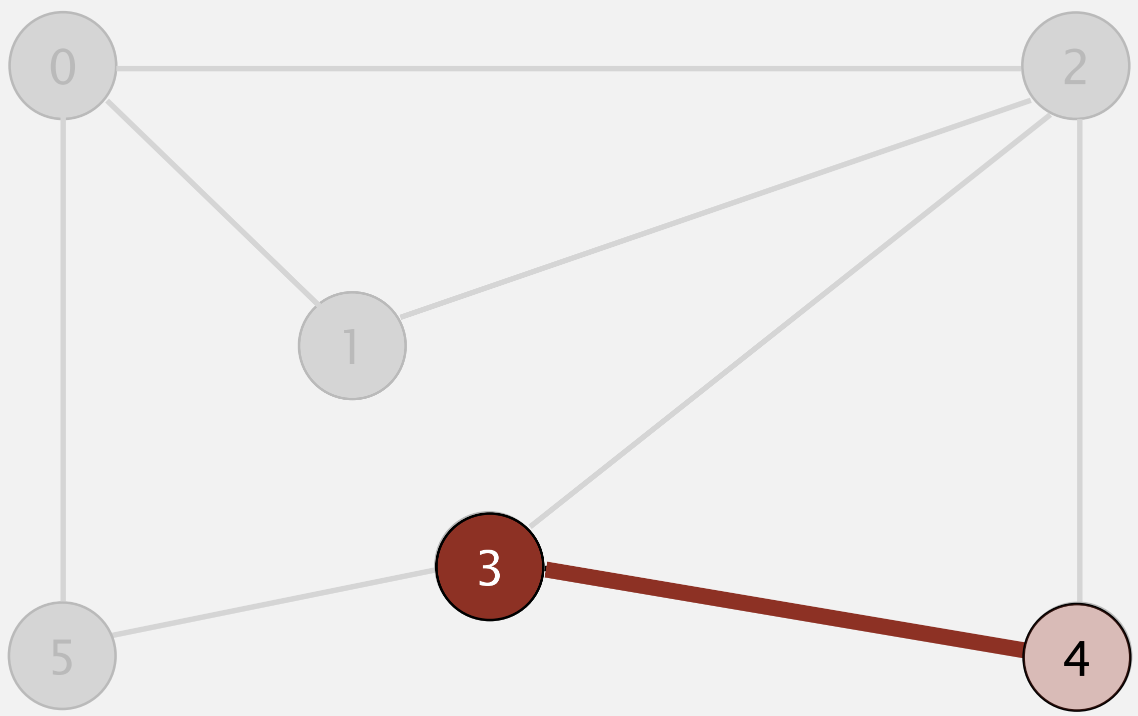
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 3

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

4

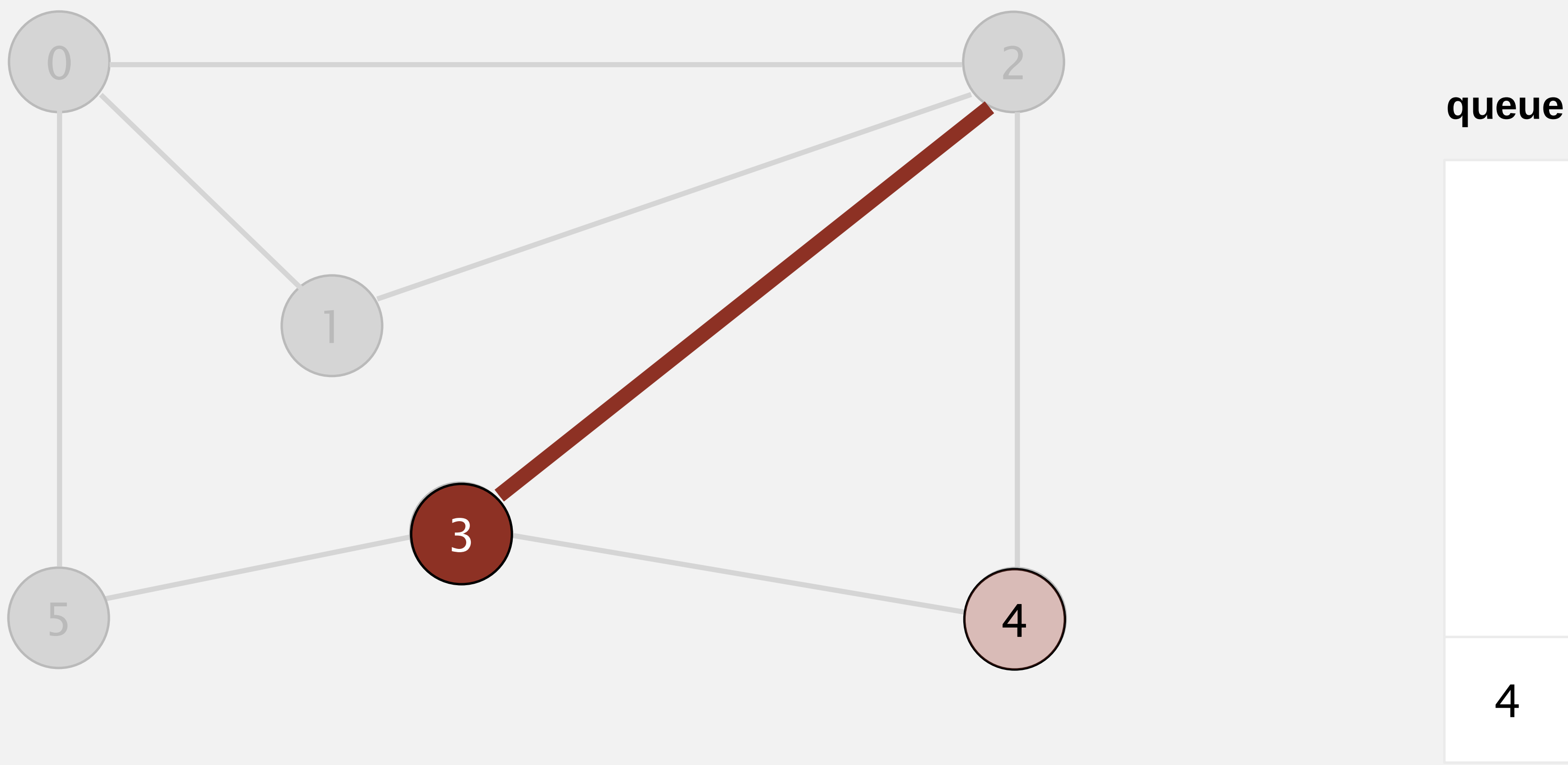
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 3

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



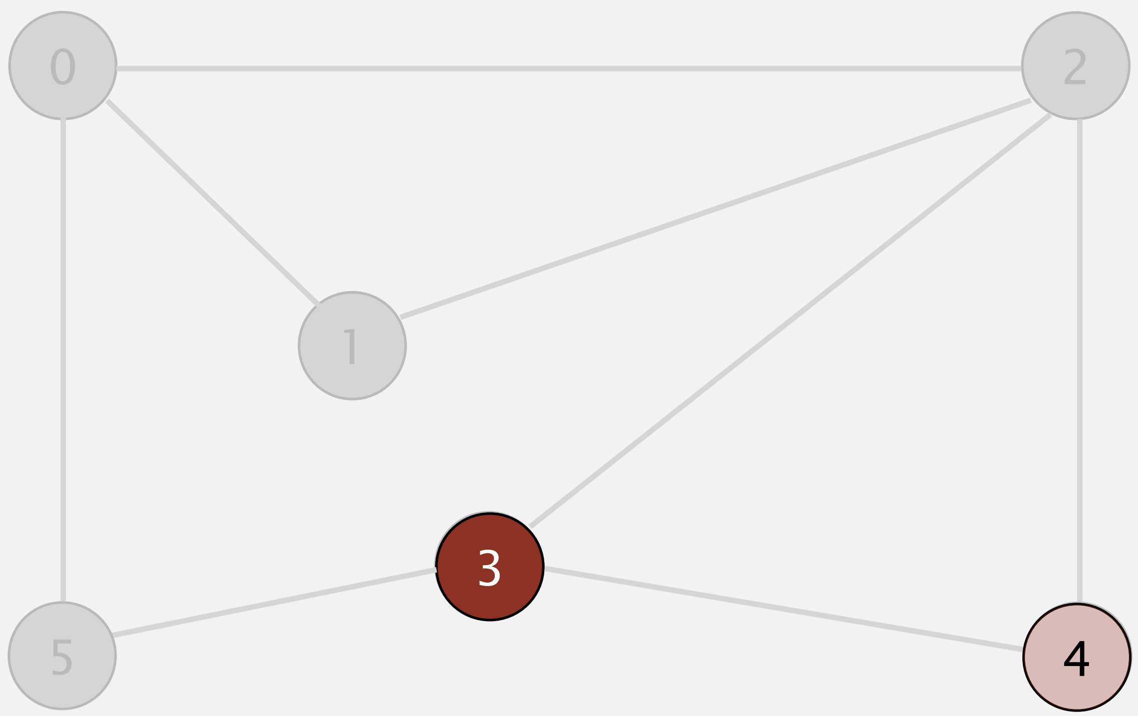
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 3

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
4

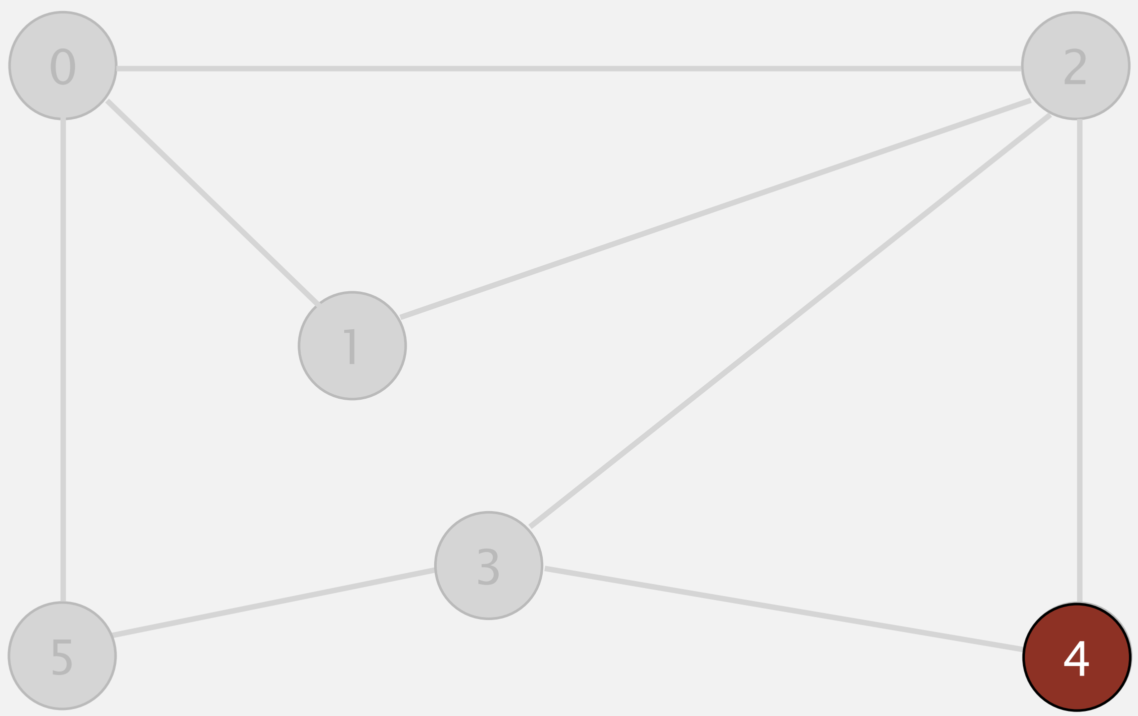
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

3 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue
4

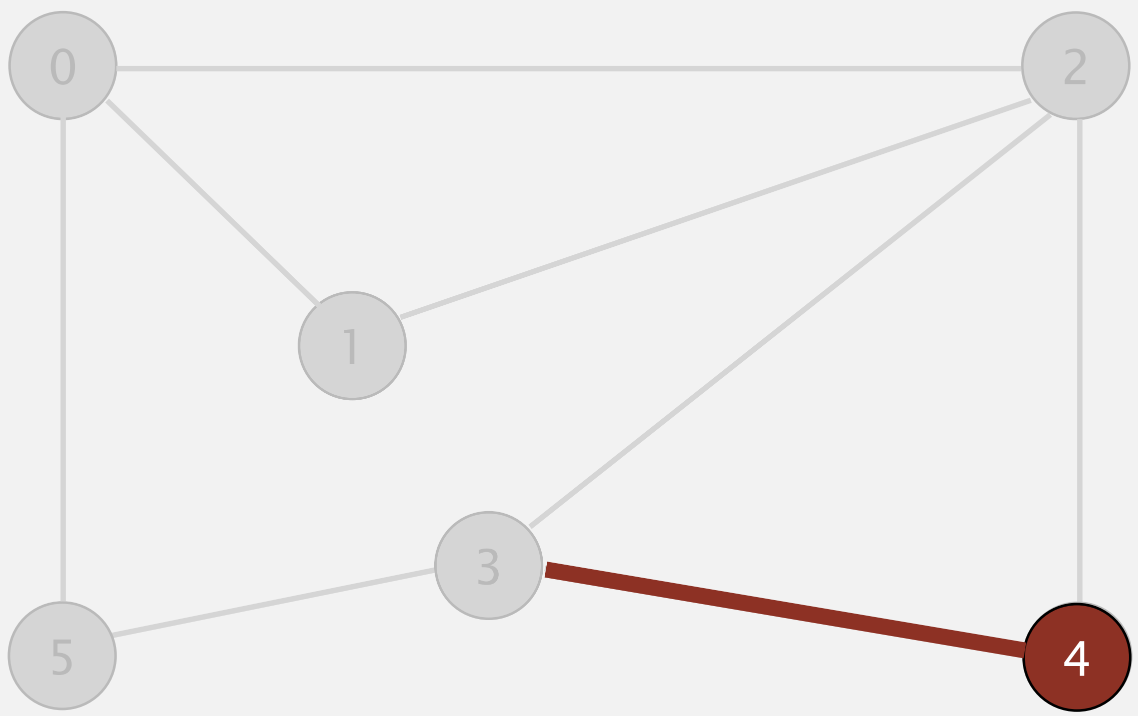
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

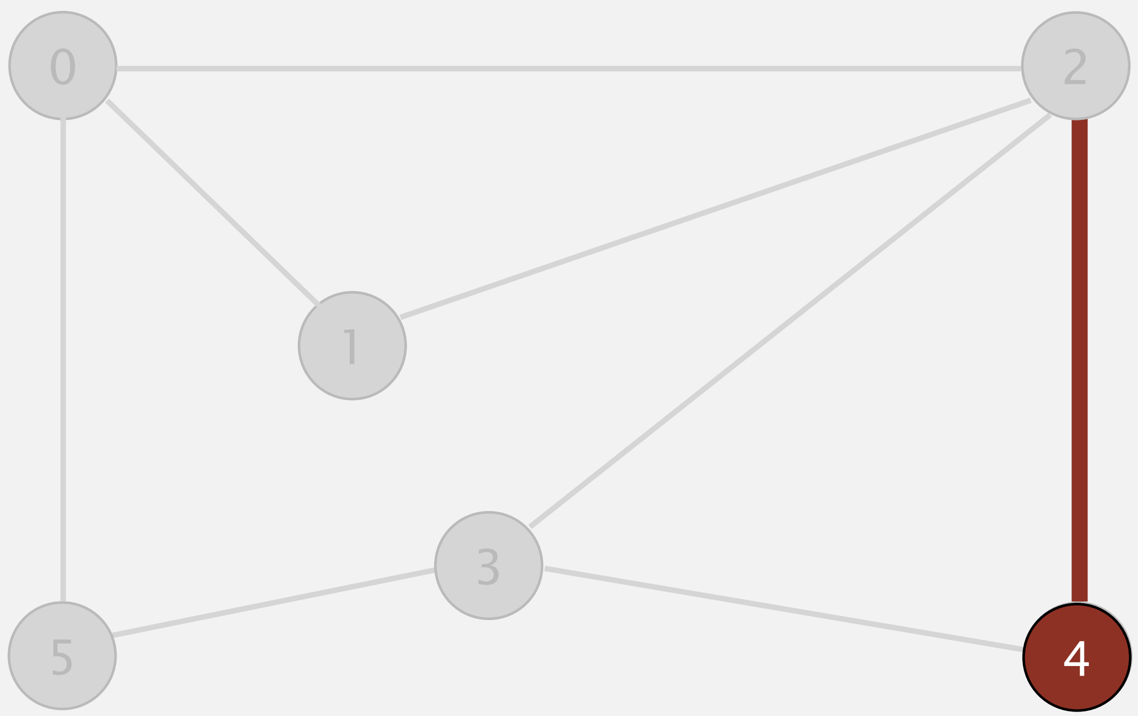
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

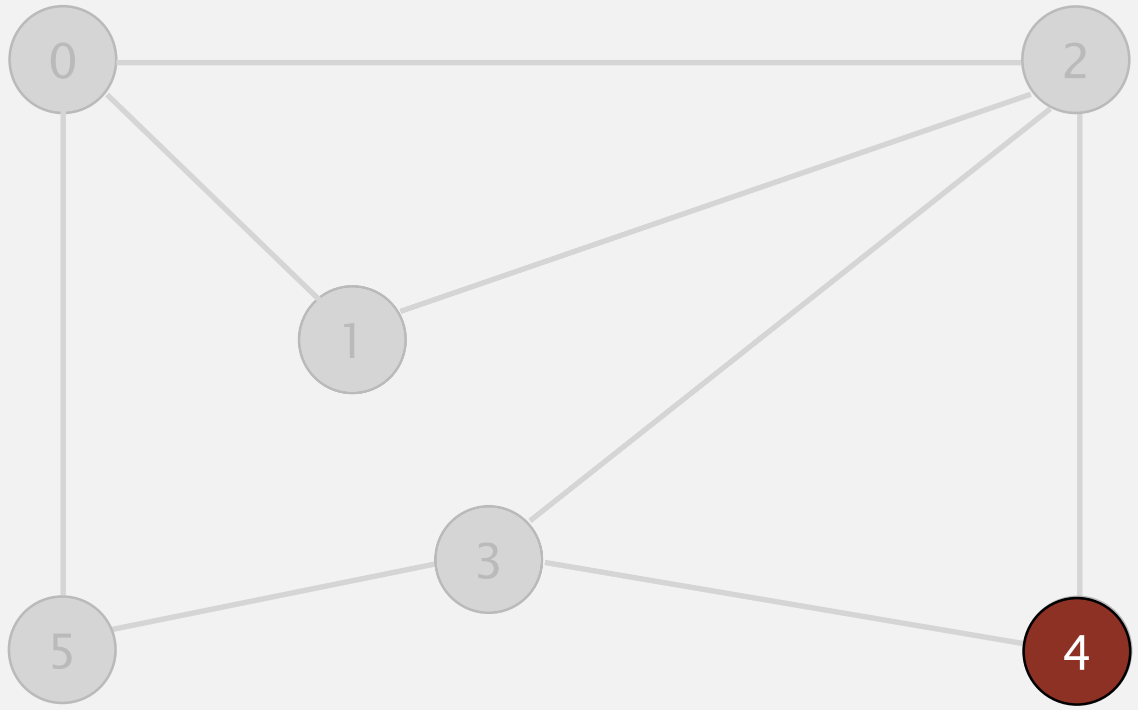
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

dequeue 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

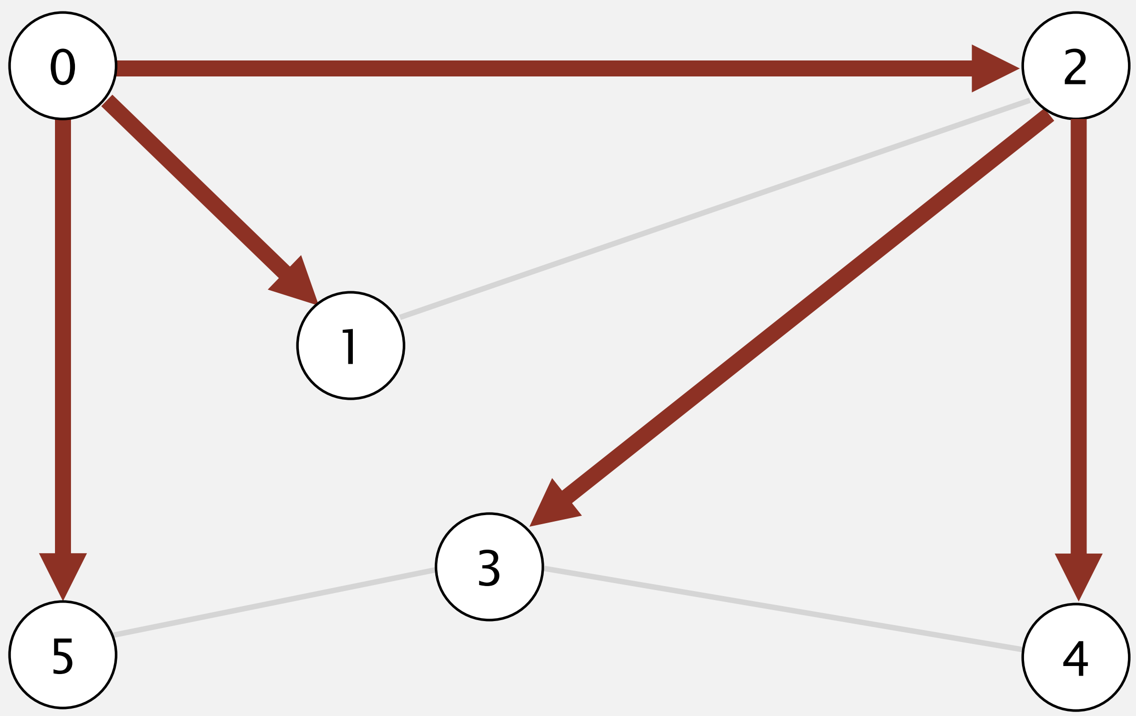
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

4 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

done

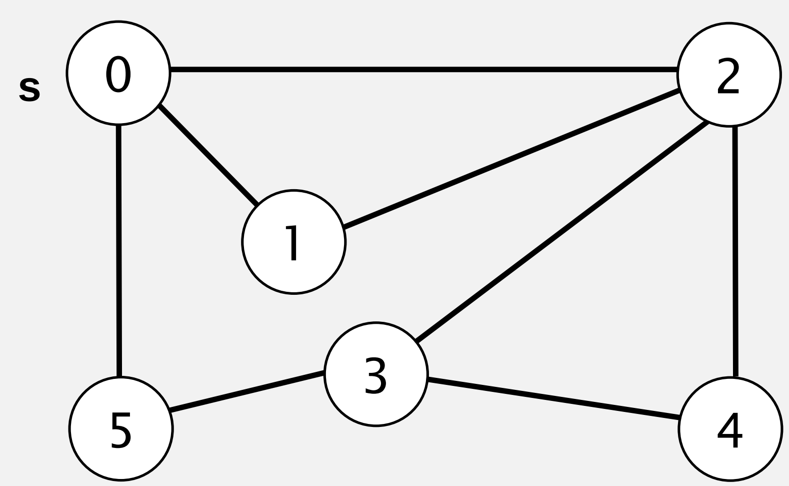
Breadth-first search properties

Q. In which order does BFS examine vertices?

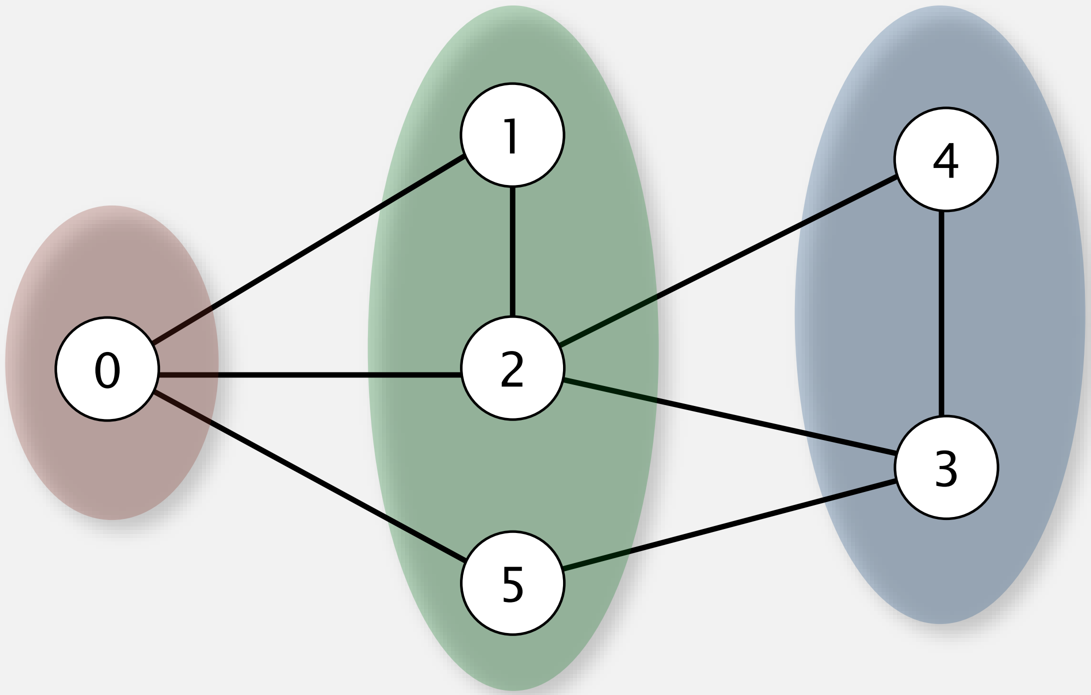
A. Increasing distance (number of edges) from s.

queue always consists of ≥ 0 vertices of distance k from s,
followed by ≥ 0 vertices of distance $k+1$

Proposition. In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



dist = 0 dist = 1 dist = 2

http://algs4.cs.princeton.edu

Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- *connected components*
- challenges

Connectivity queries

Def. Vertices *v* and *w* are connected if there is a path between them.

Goal. Preprocess graph to answer queries of the form *is v connected to w?* in constant time.

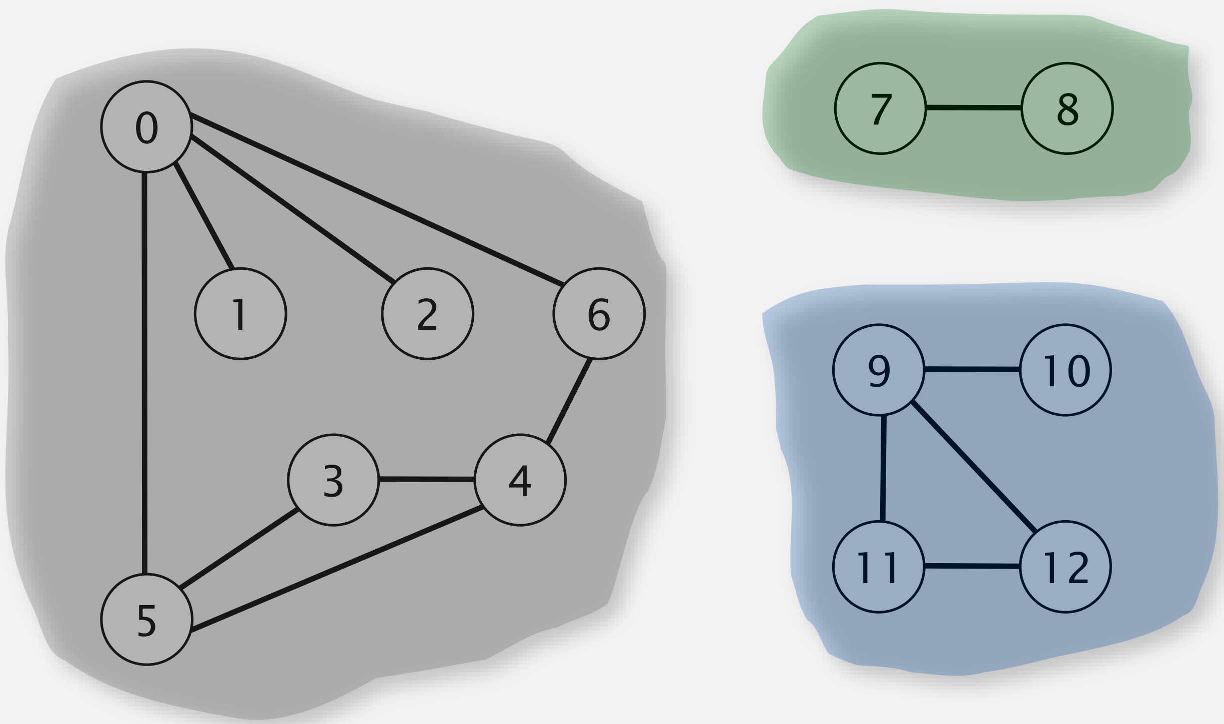
public class CC		
	`CC(Graph G)`	*find connected components in G*
`boolean`	`connected(int v, int w)`	*are v and w connected?*
`int`	`count()`	*number of connected components*
`int`	`id(int v)`	*component identifier for v (between* `0` *and* `count() - 1`*)*

Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v connected to w and w connected to x, then v connected to x.

Def. A connected component is a maximal set of connected vertices.



3 connected components

v	id[]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Remark. Given connected components, can answer queries in constant time.

Finding connected components with DFS

```
public class CC
{
   private boolean[] marked;
   private int[] id;
   private int count;
   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }
   public int count()
   public int id(int v)
   public boolean connected(int v, int w)
   private void dfs(Graph G, int v)
}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding connected components with DFS (continued)

```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

public boolean connected(int v, int w)
{ return id[v] == id[w];  }

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}
```

number of components

id of component containing v

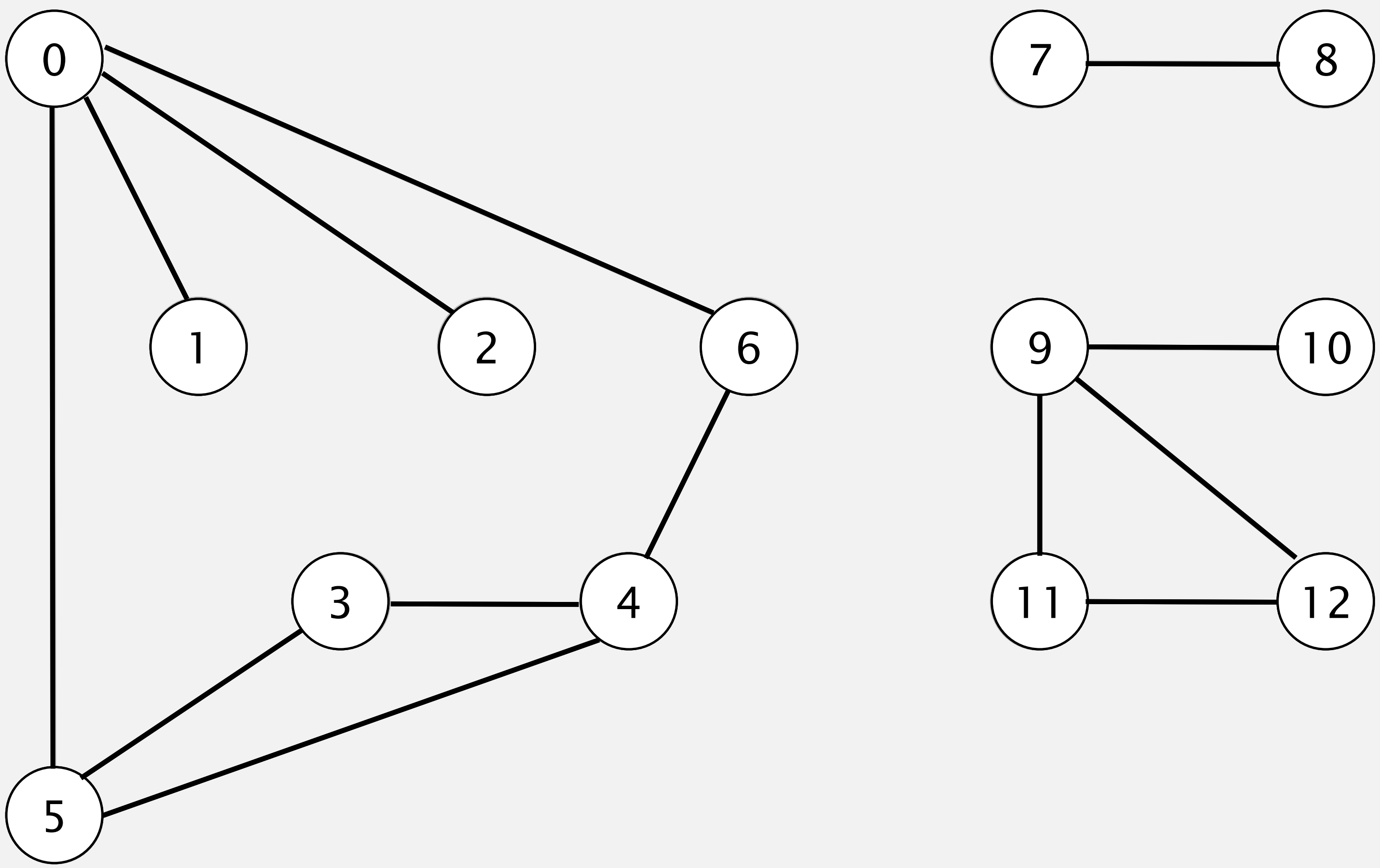
v and w connected iff same id

all vertices discovered in same call of dfs have same id

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



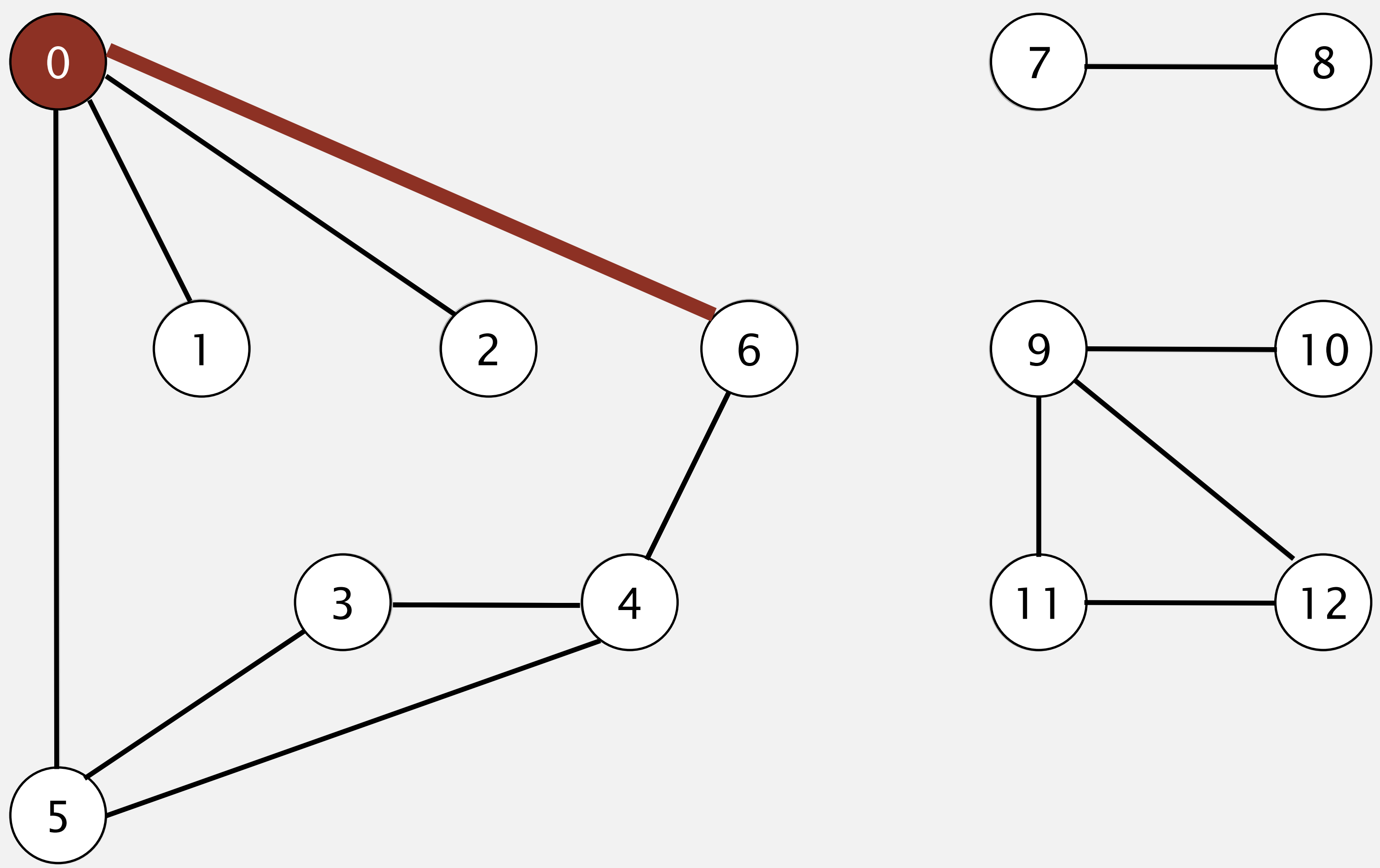
graph G

v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



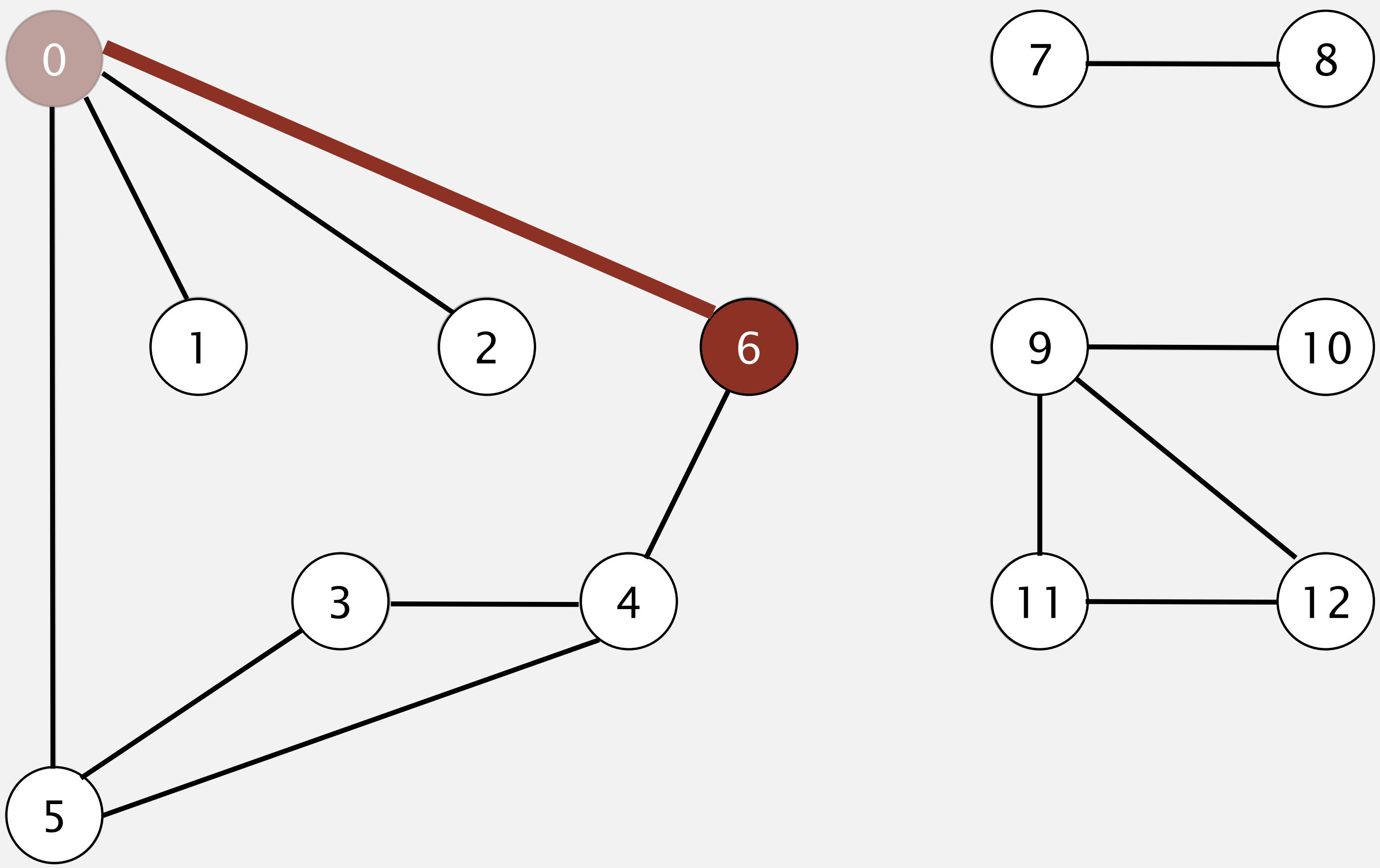
visit 0

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



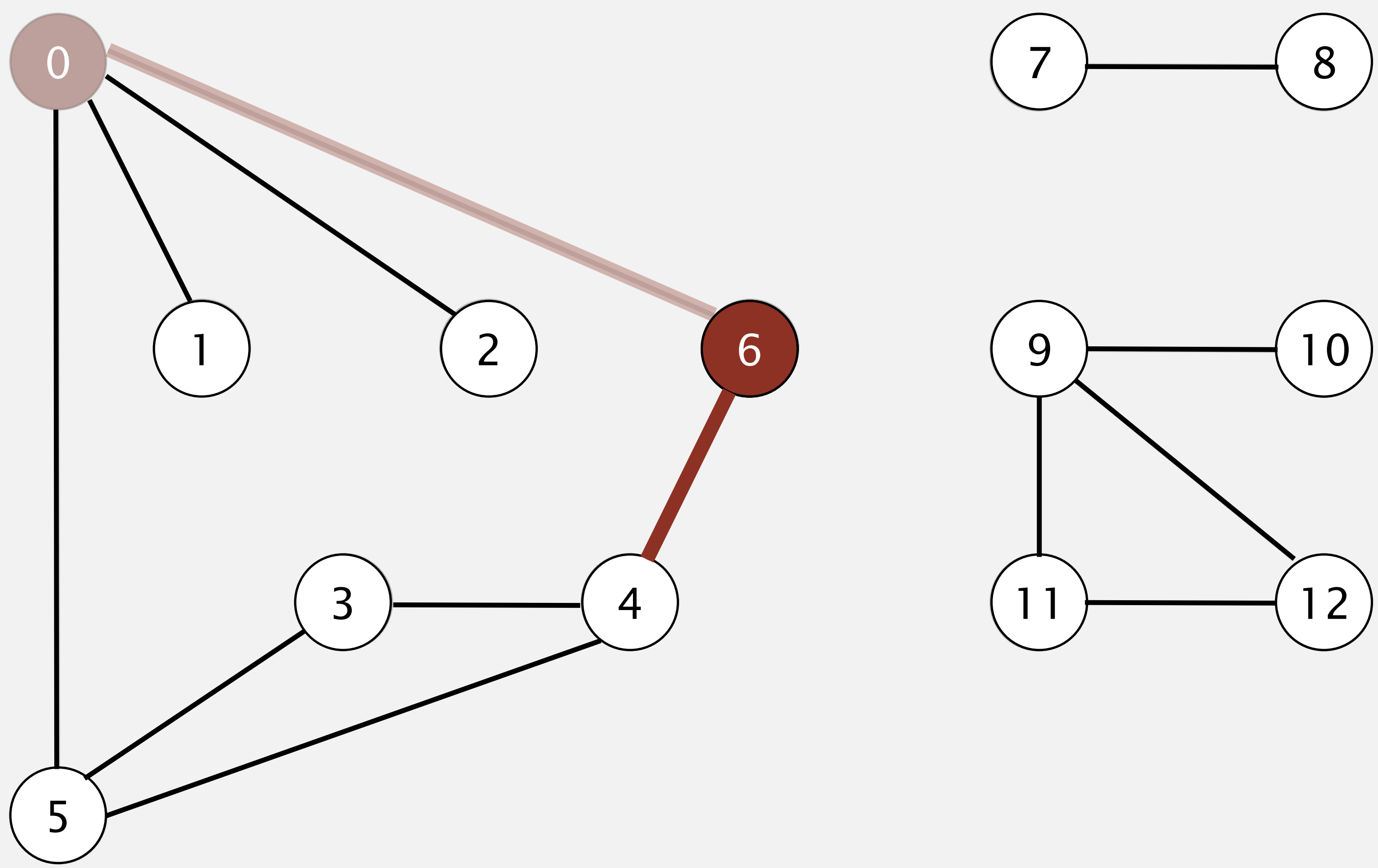
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

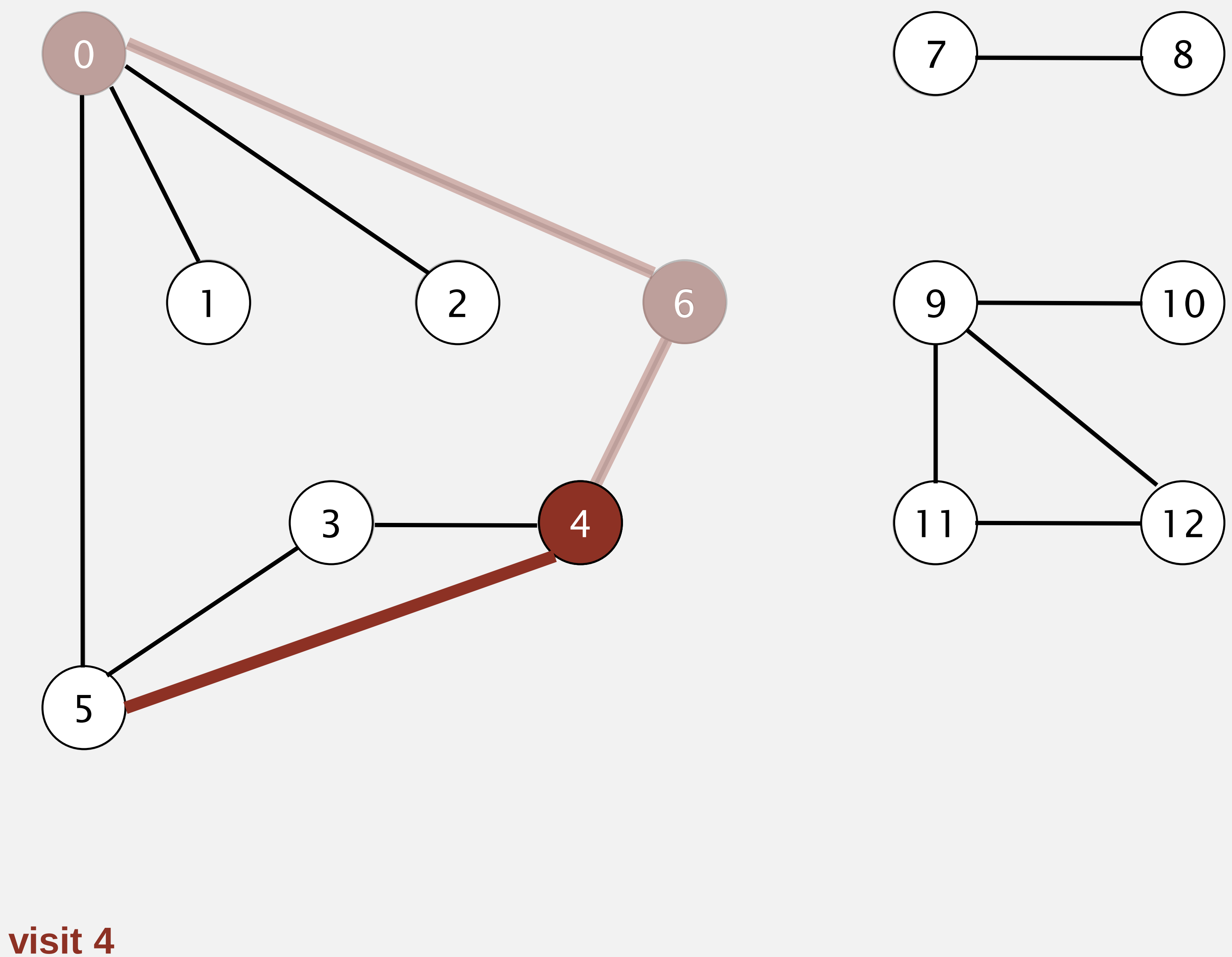


v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

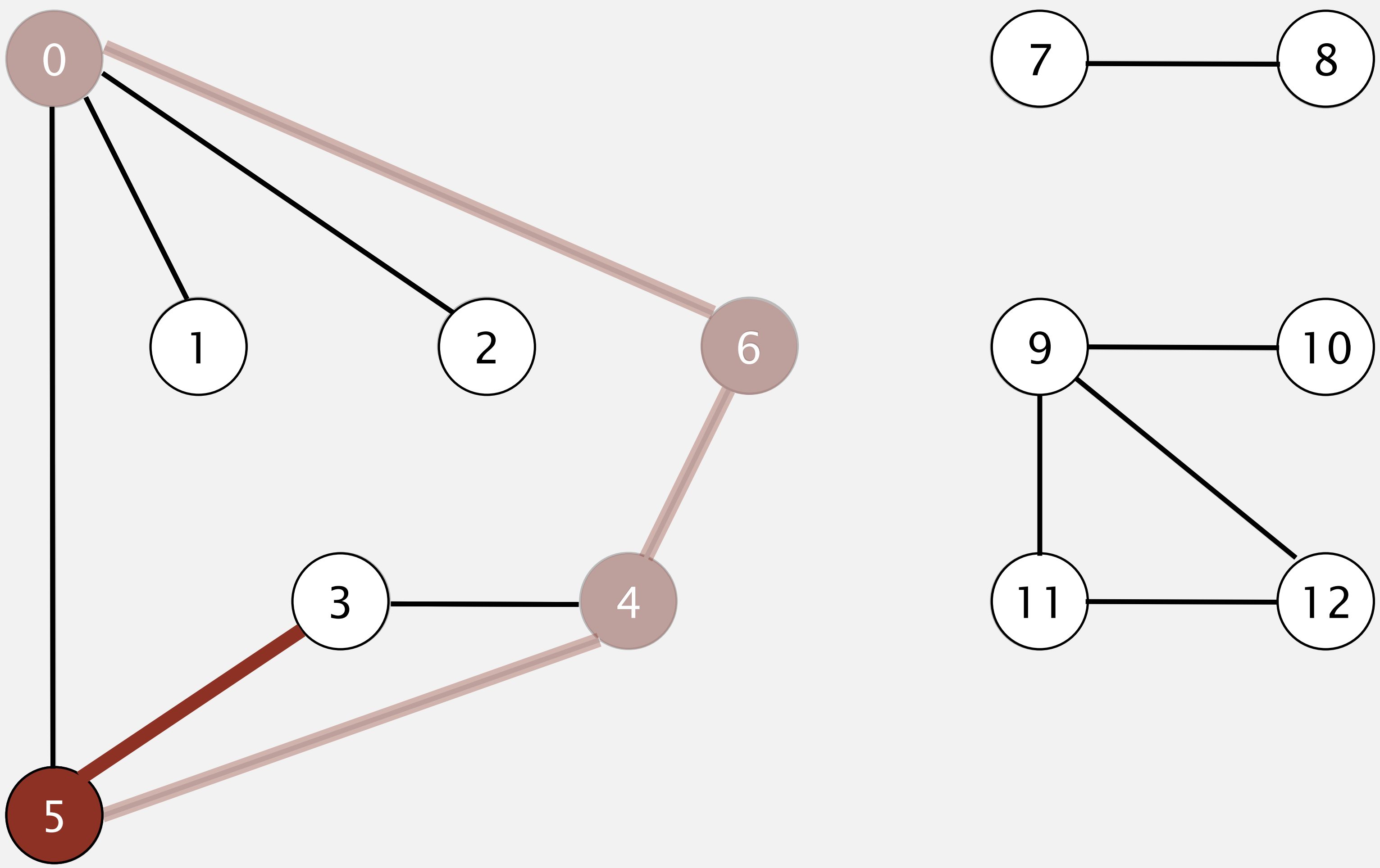


v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



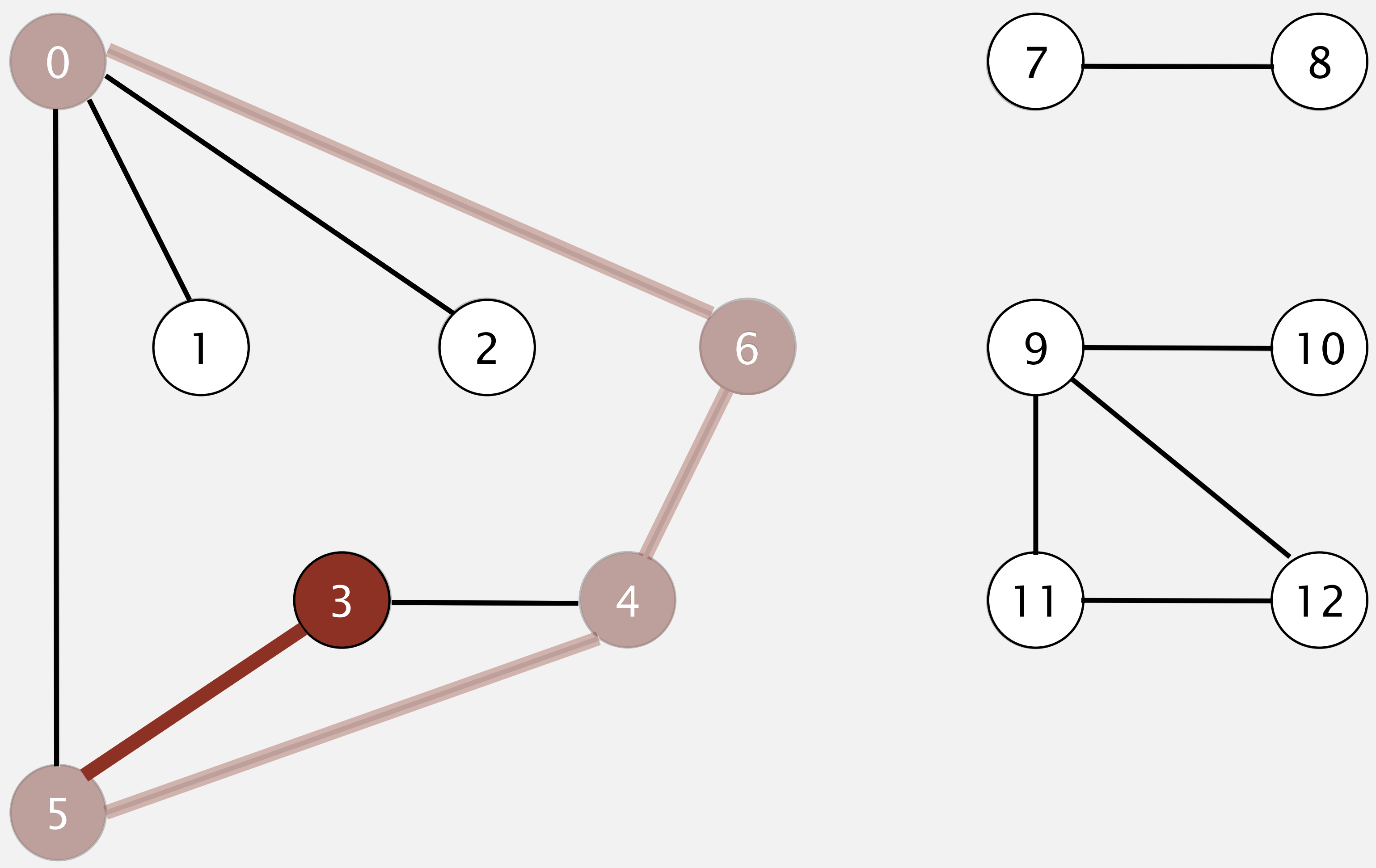
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



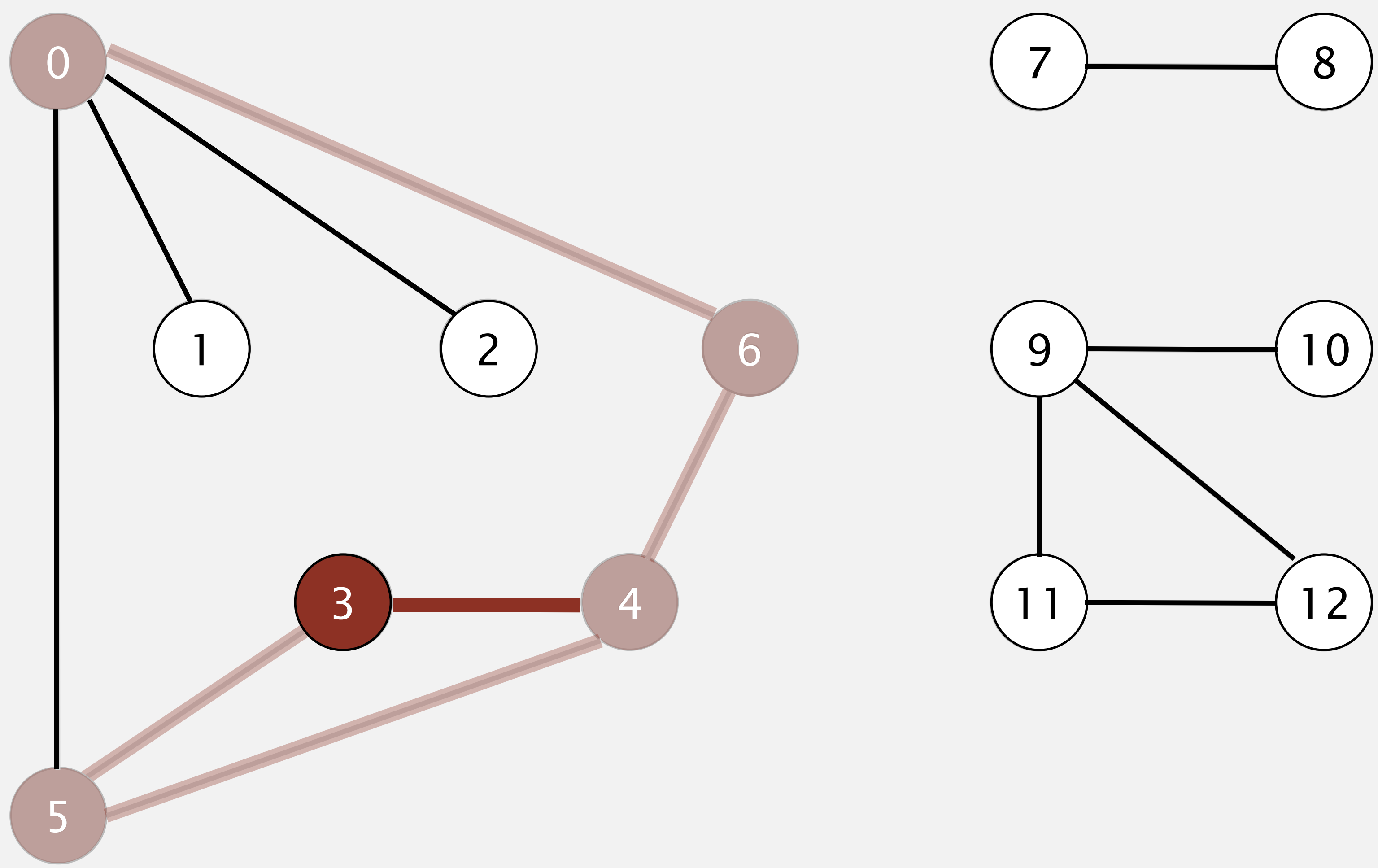
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



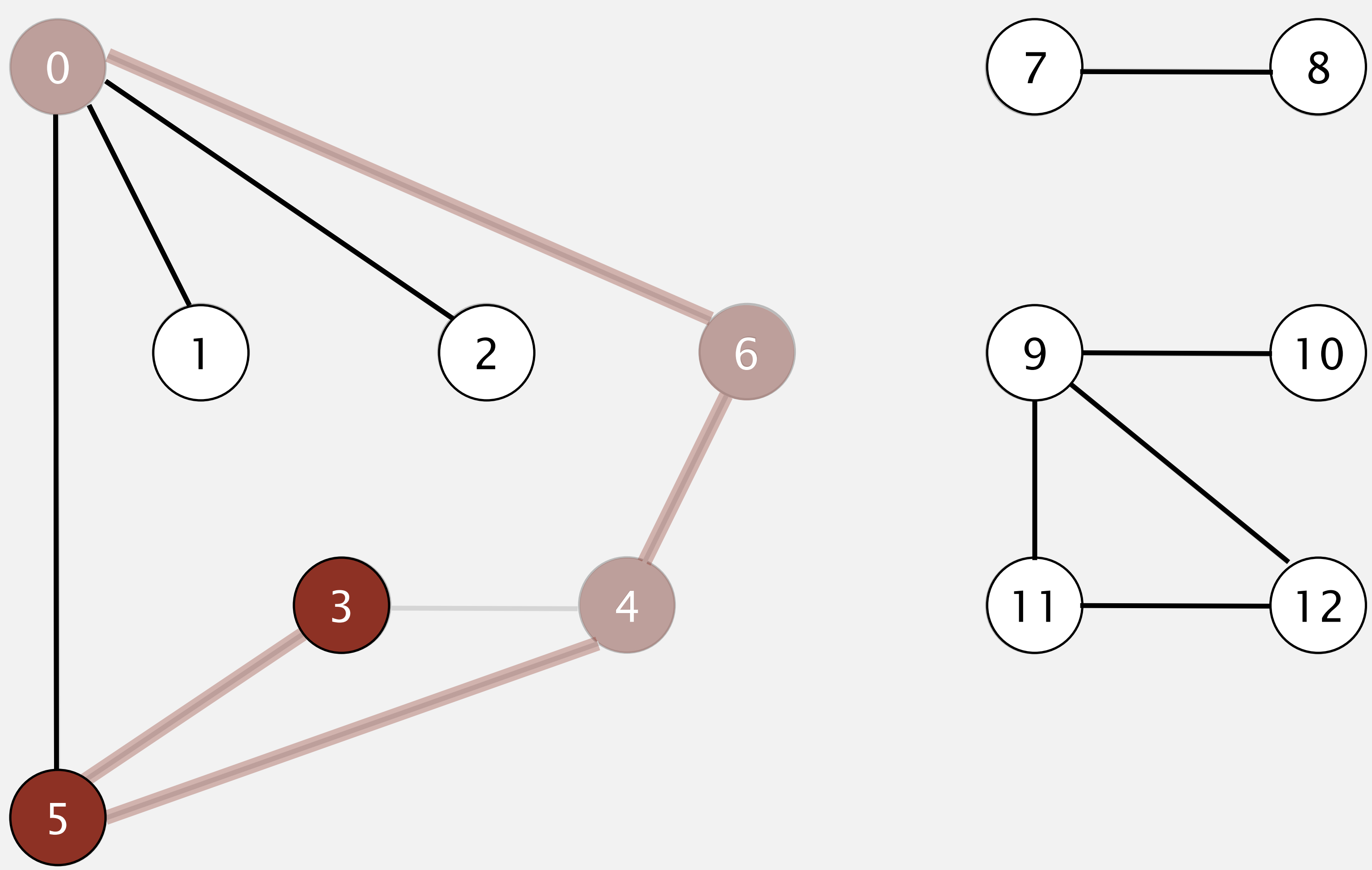
visit 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



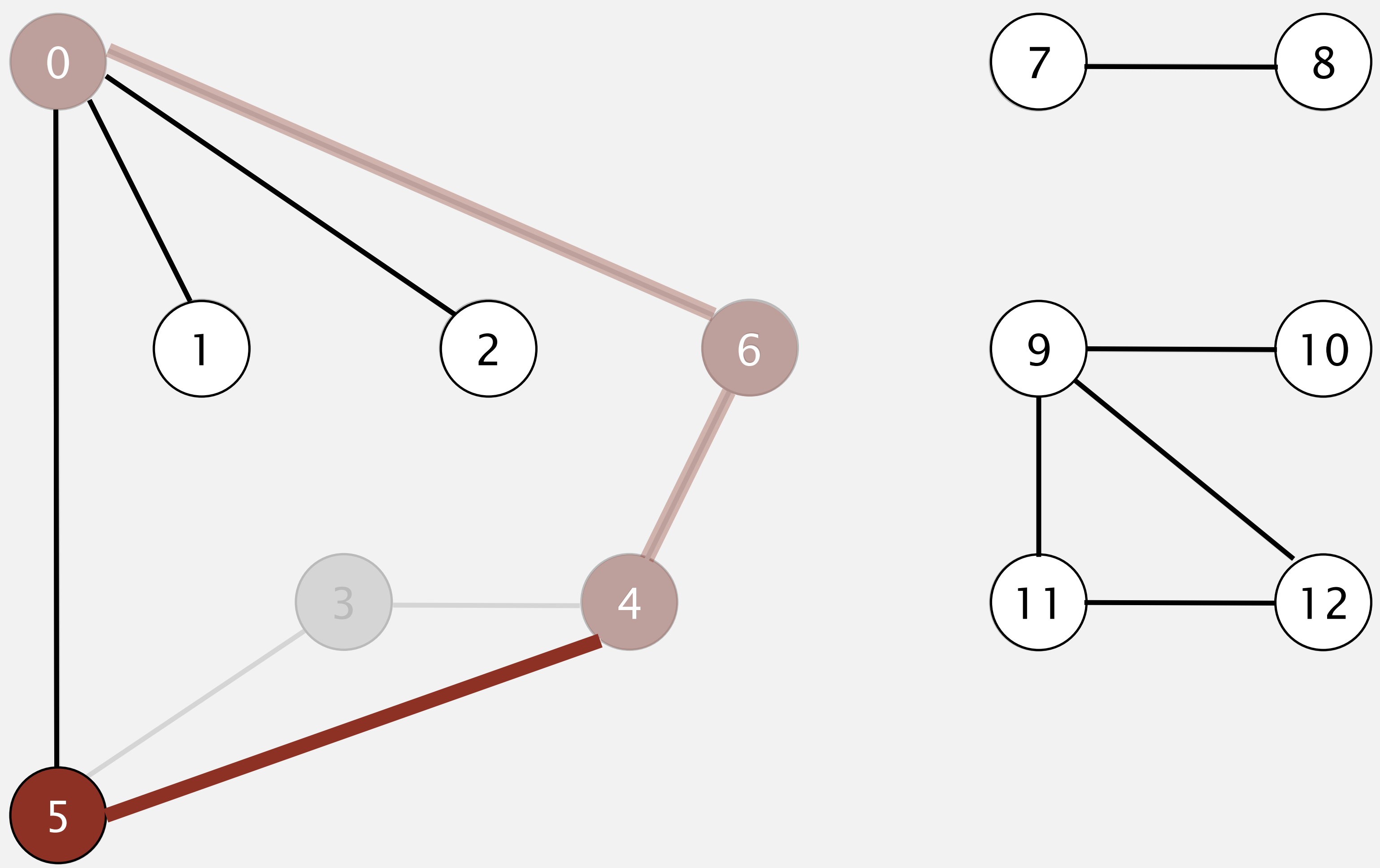
3 done

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



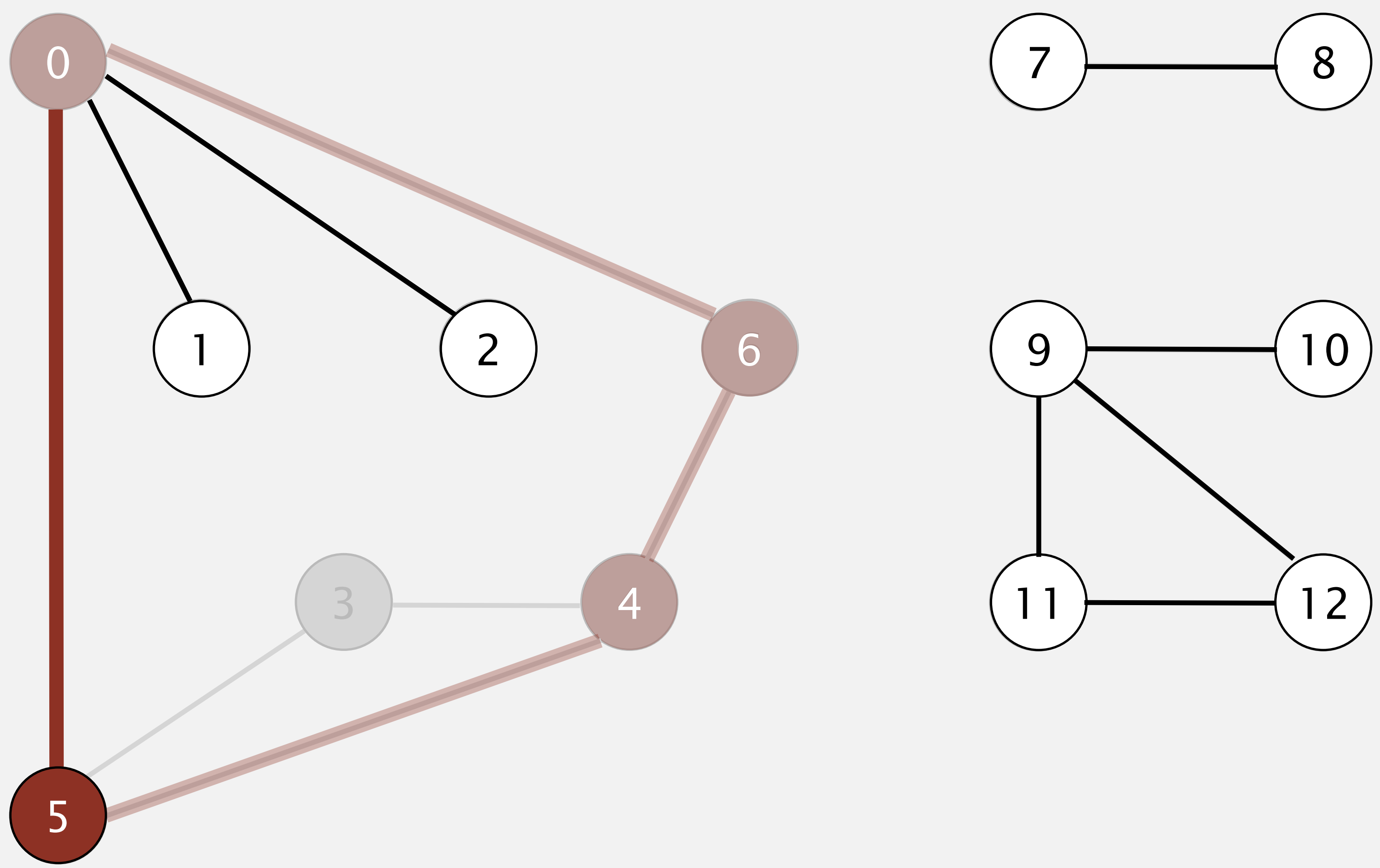
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



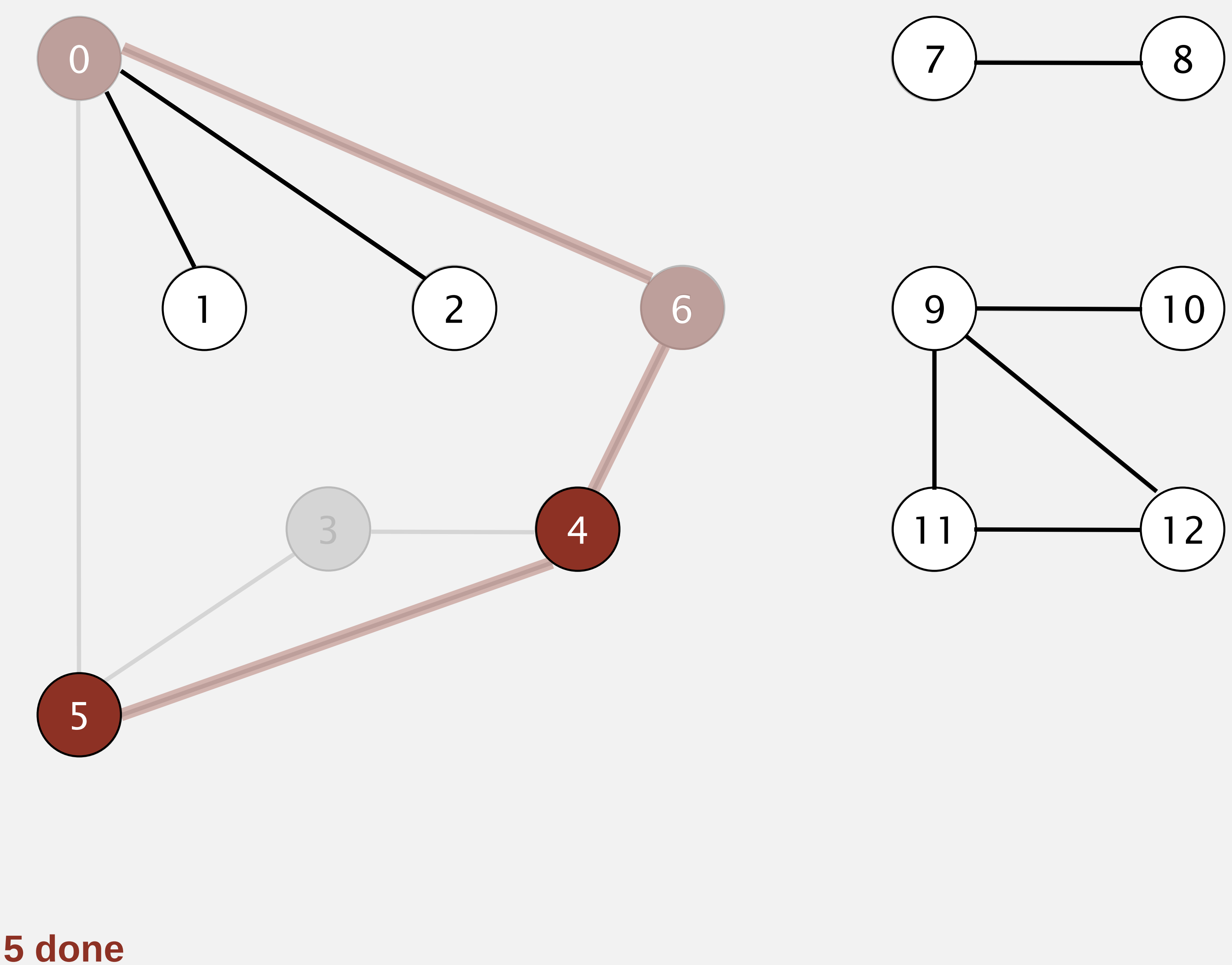
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

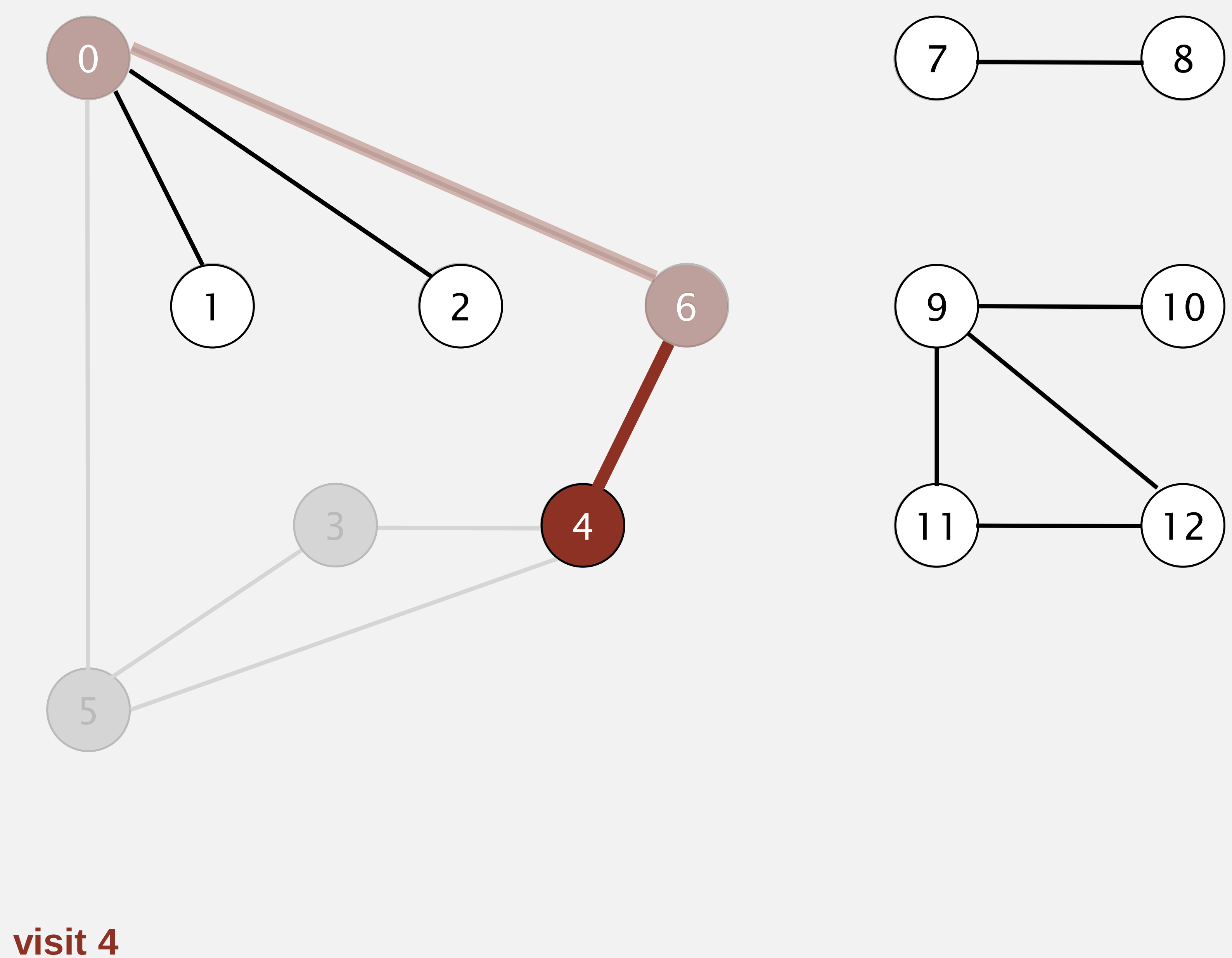


v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

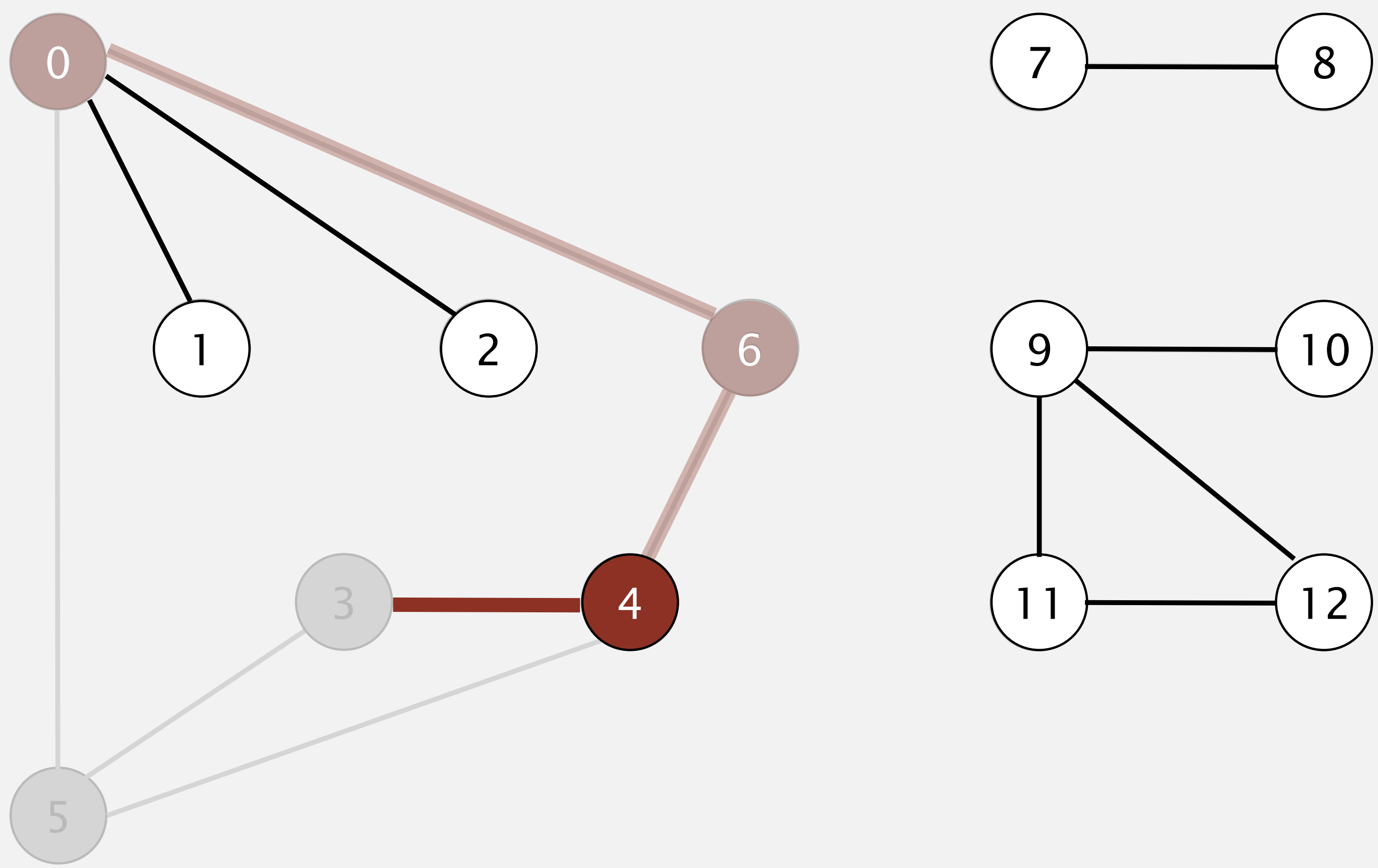


v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



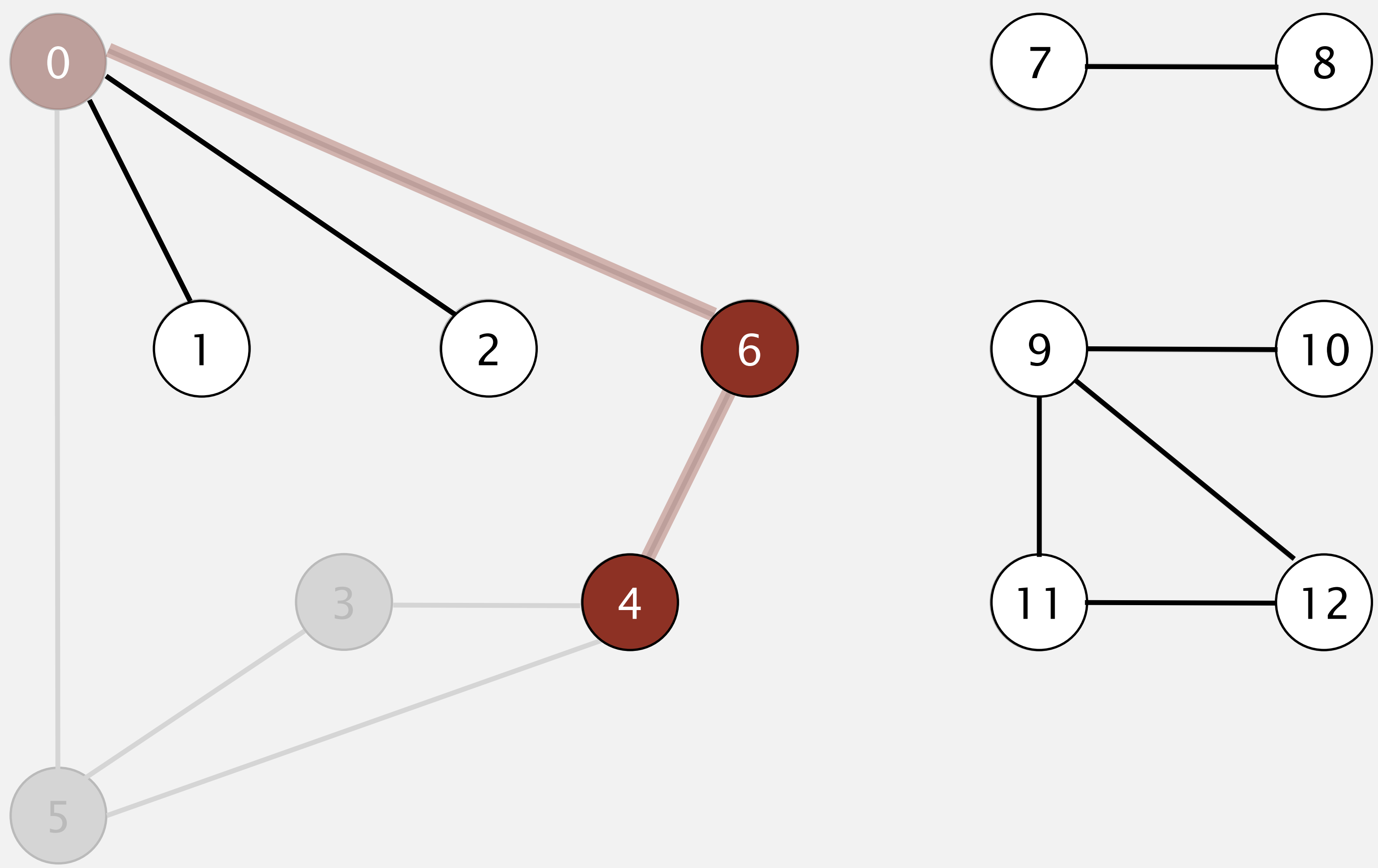
visit 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



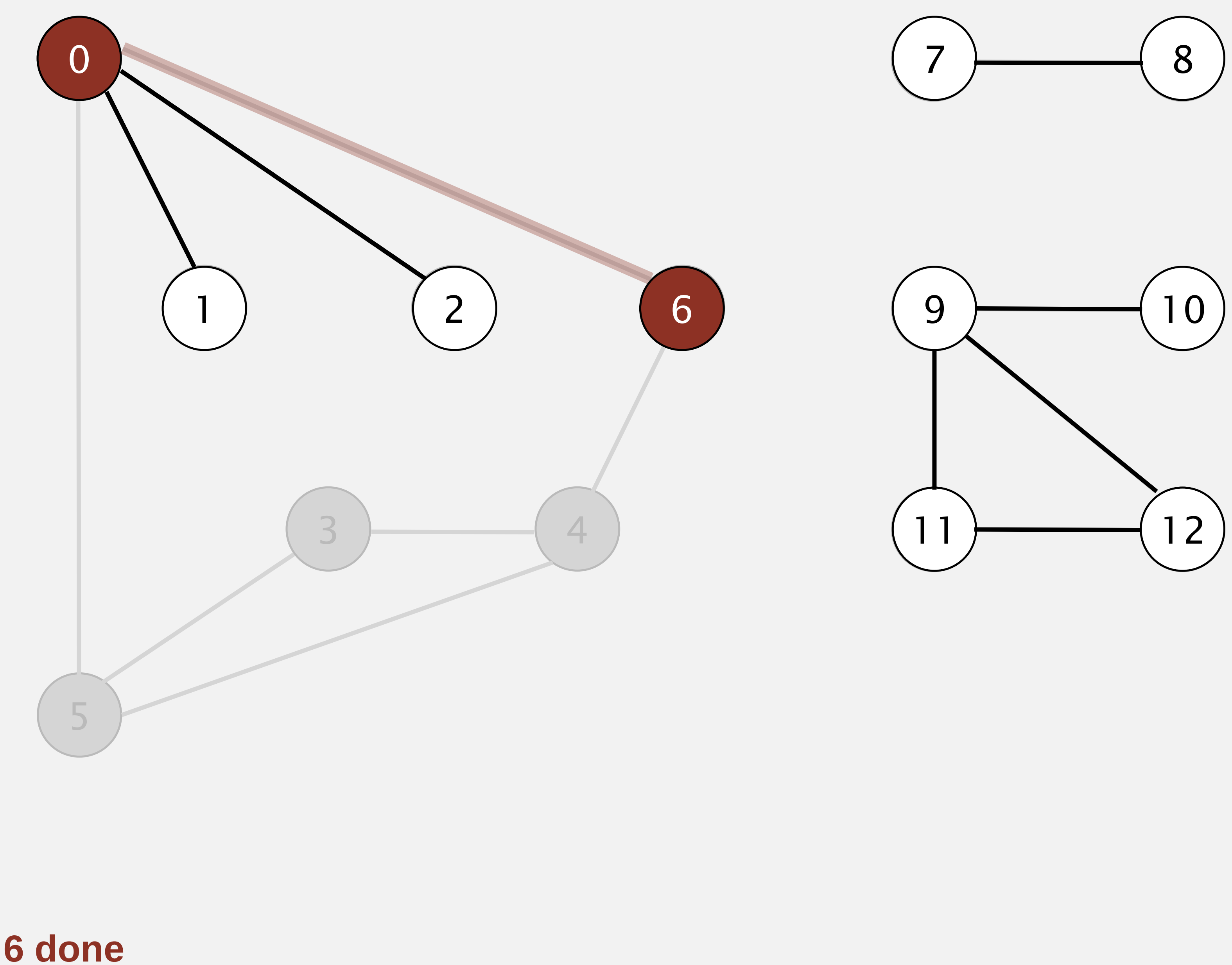
4 done

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

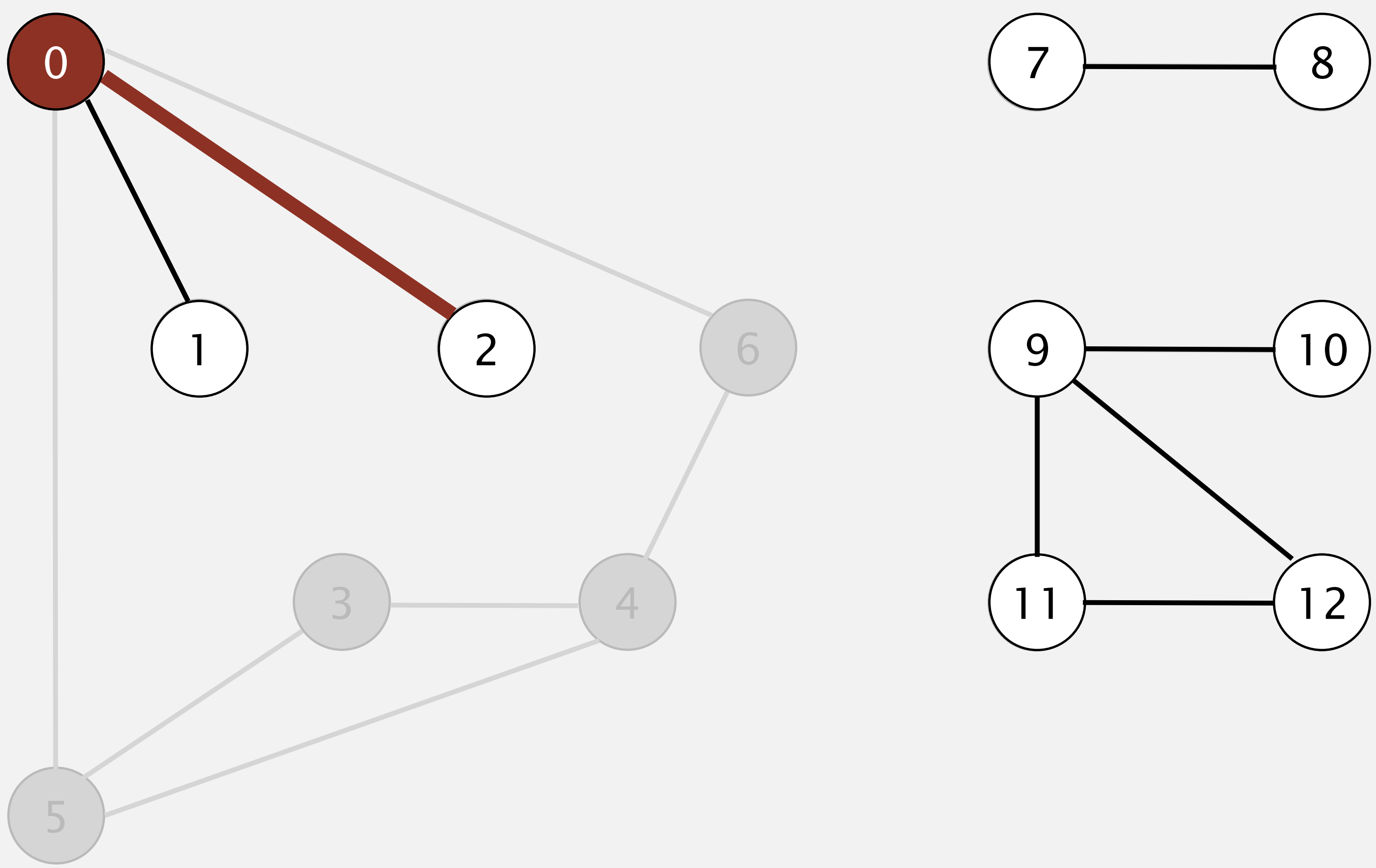


v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



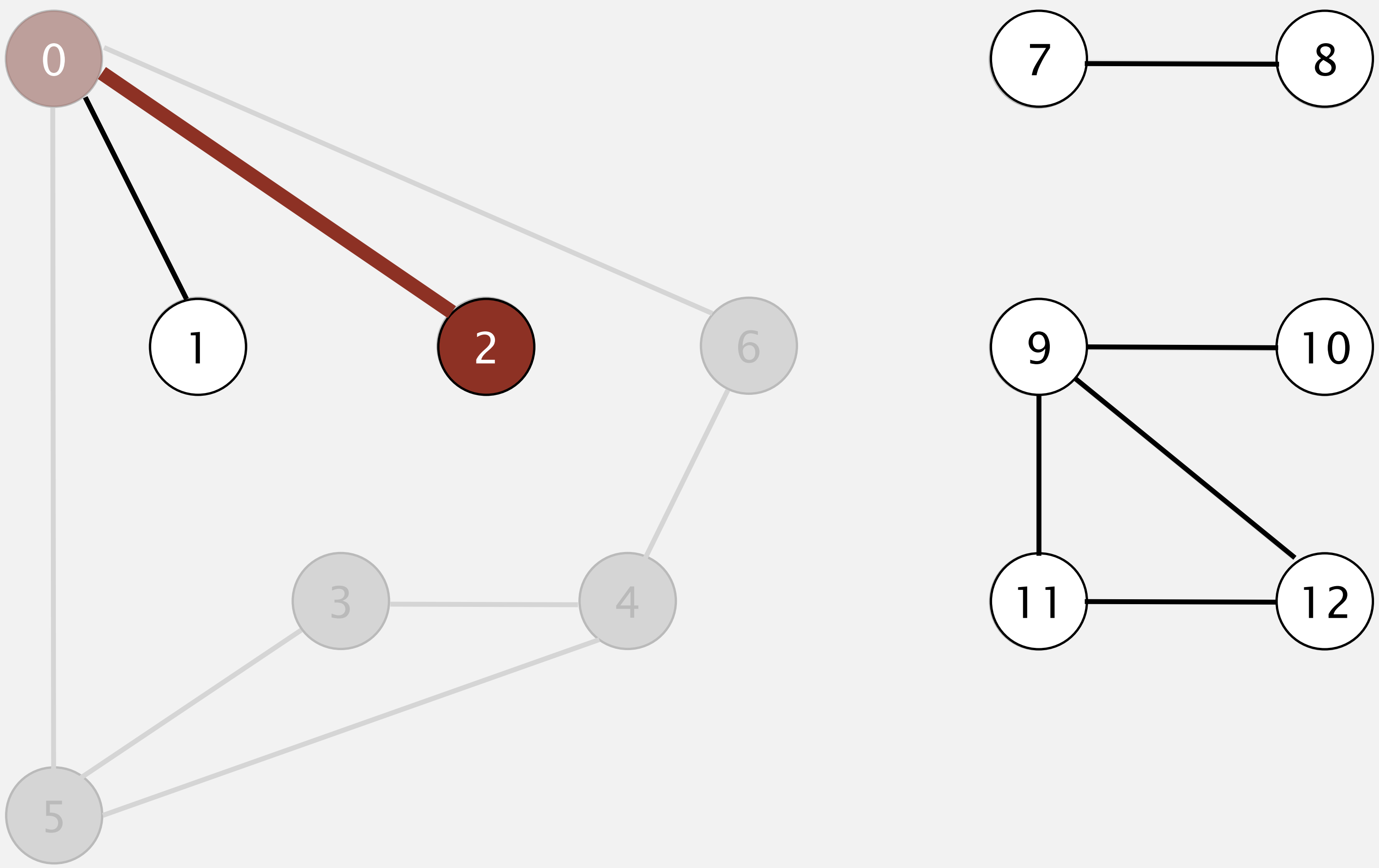
v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



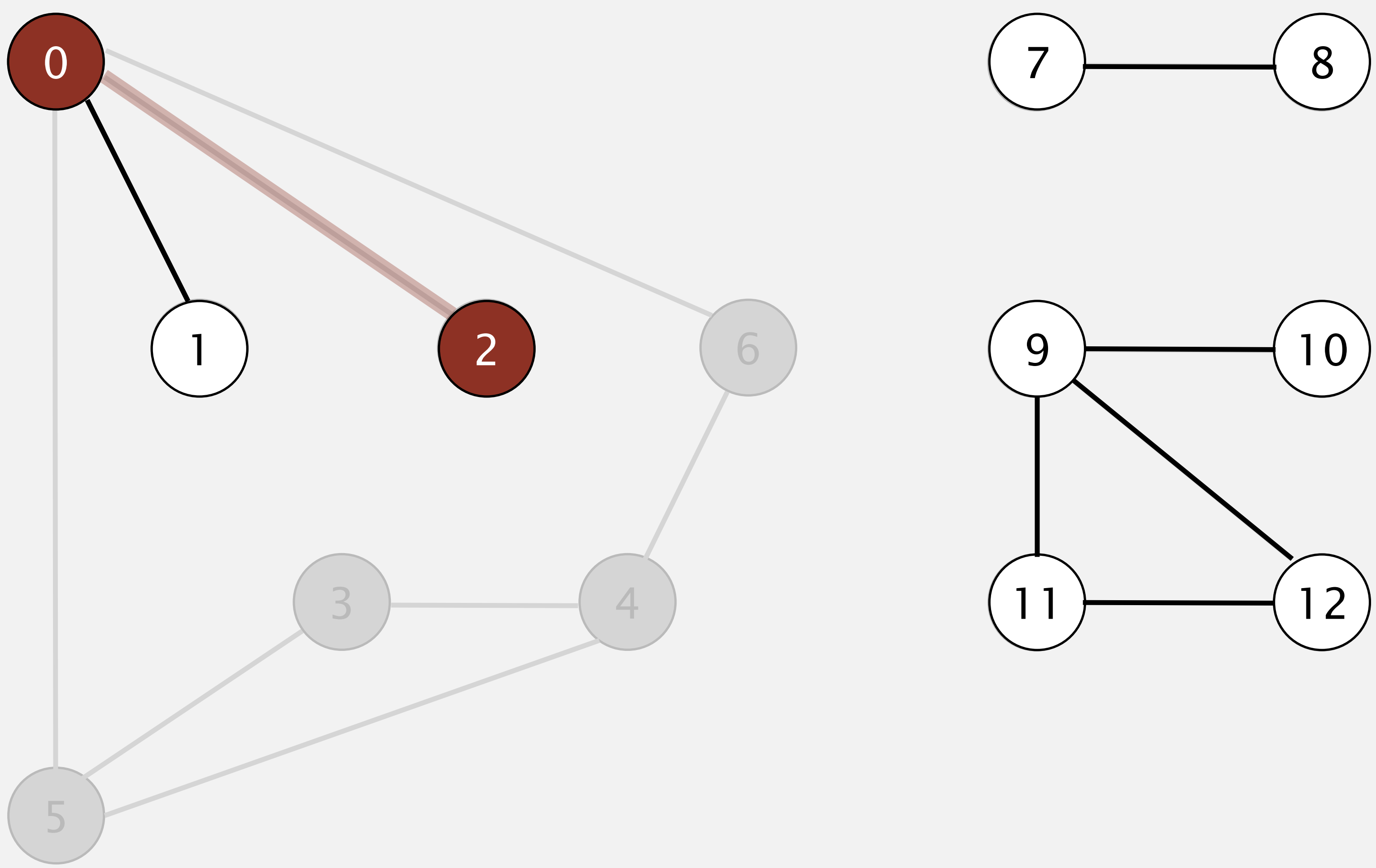
v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



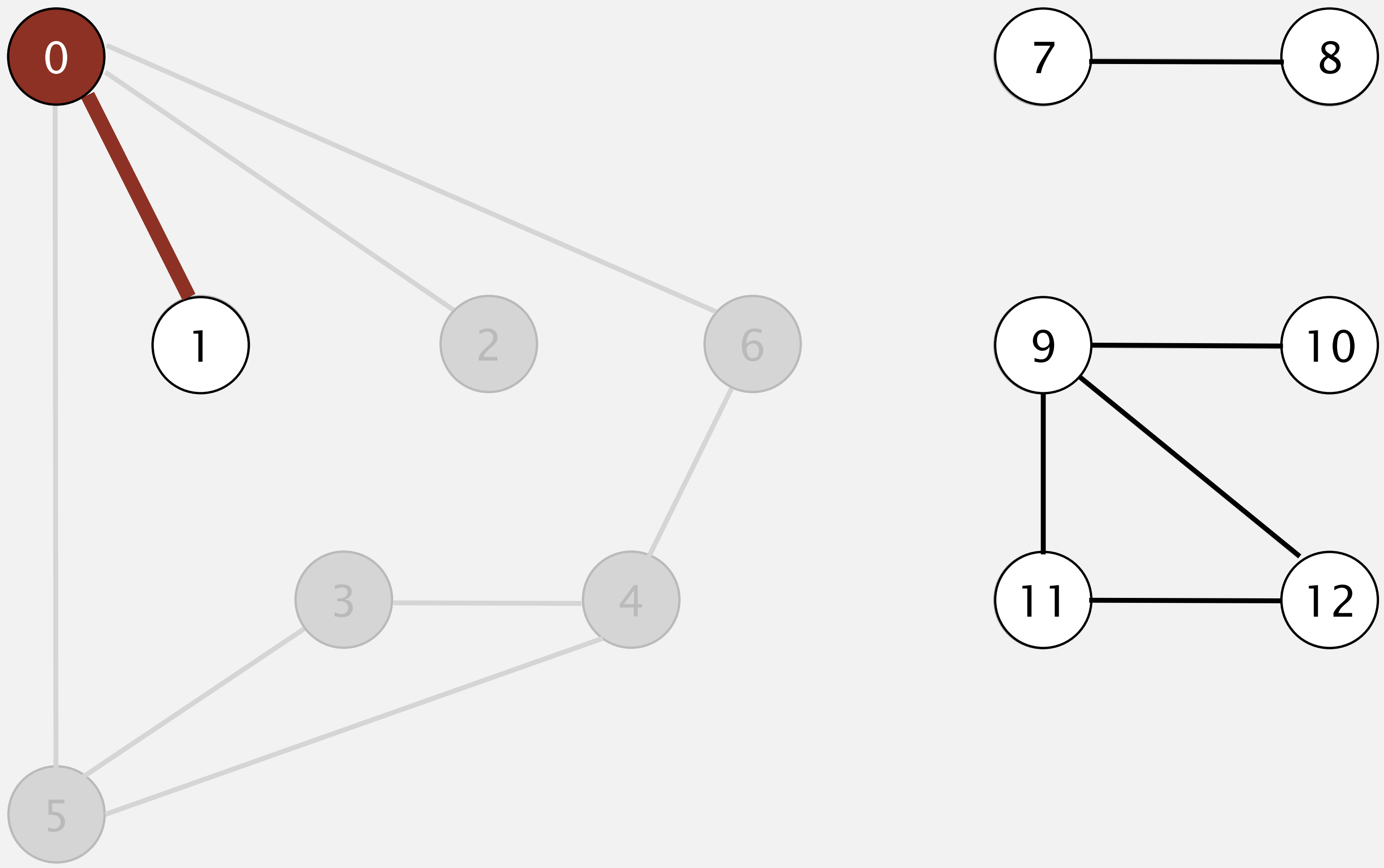
2 done

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



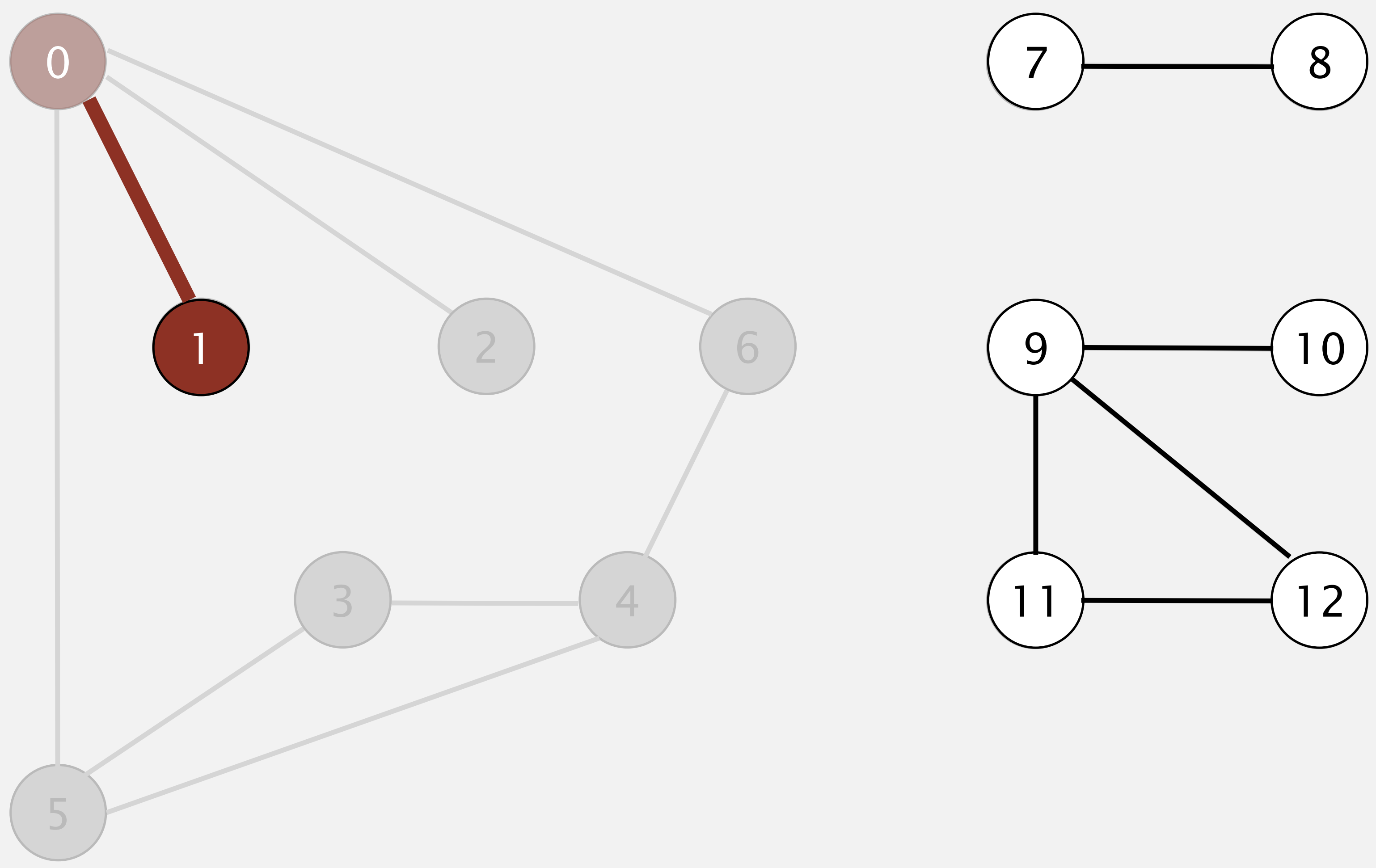
v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



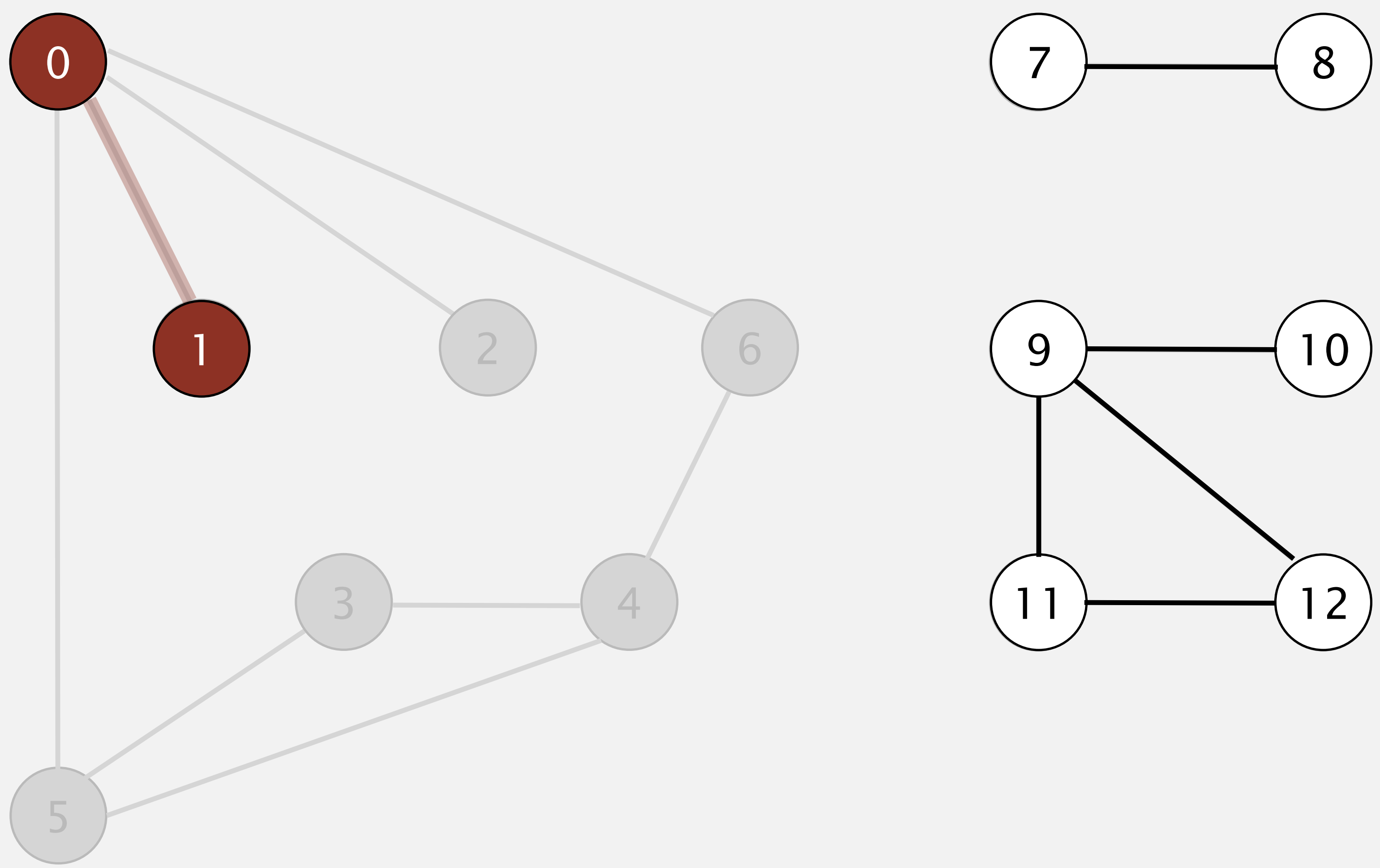
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 1

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



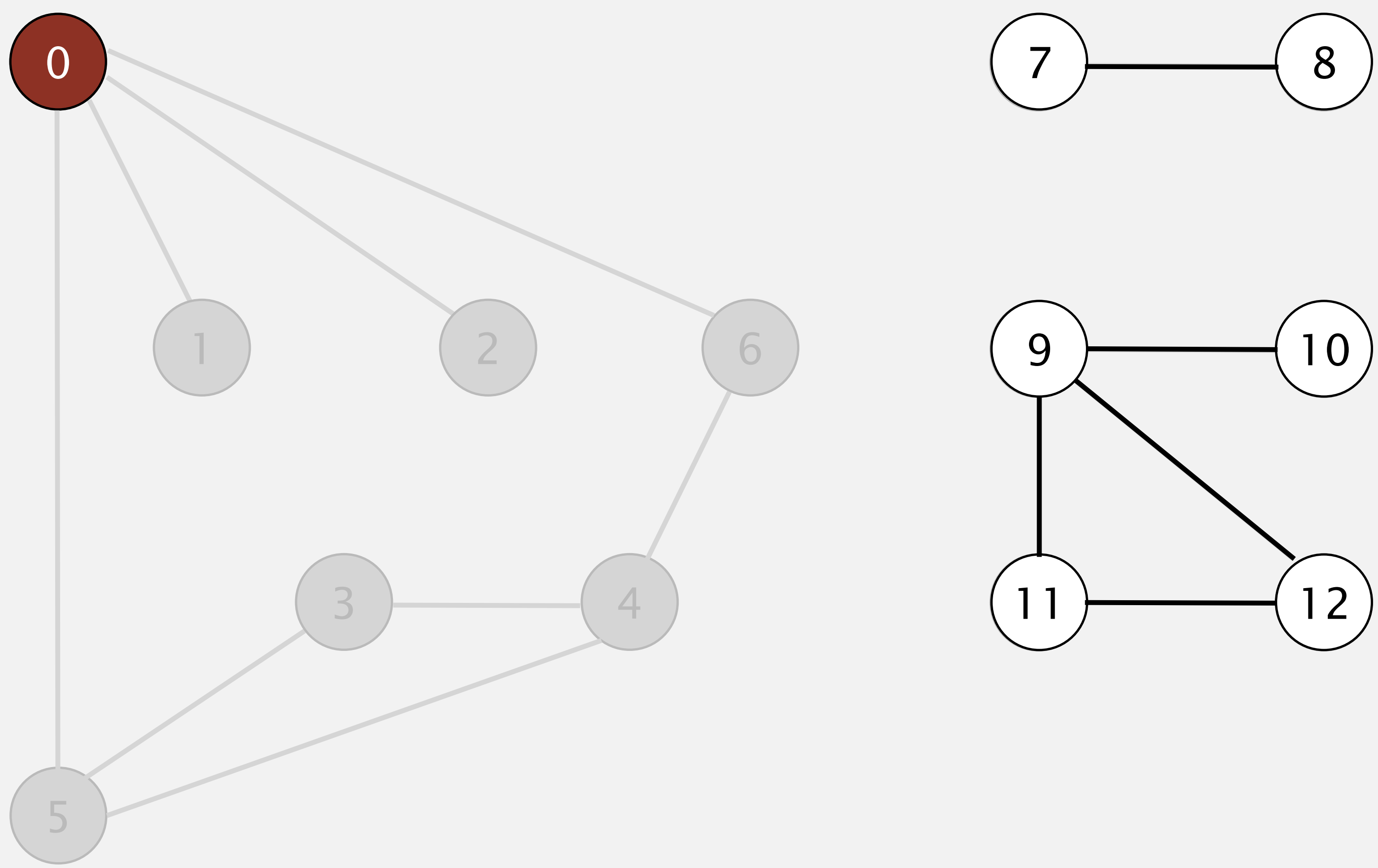
1 done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



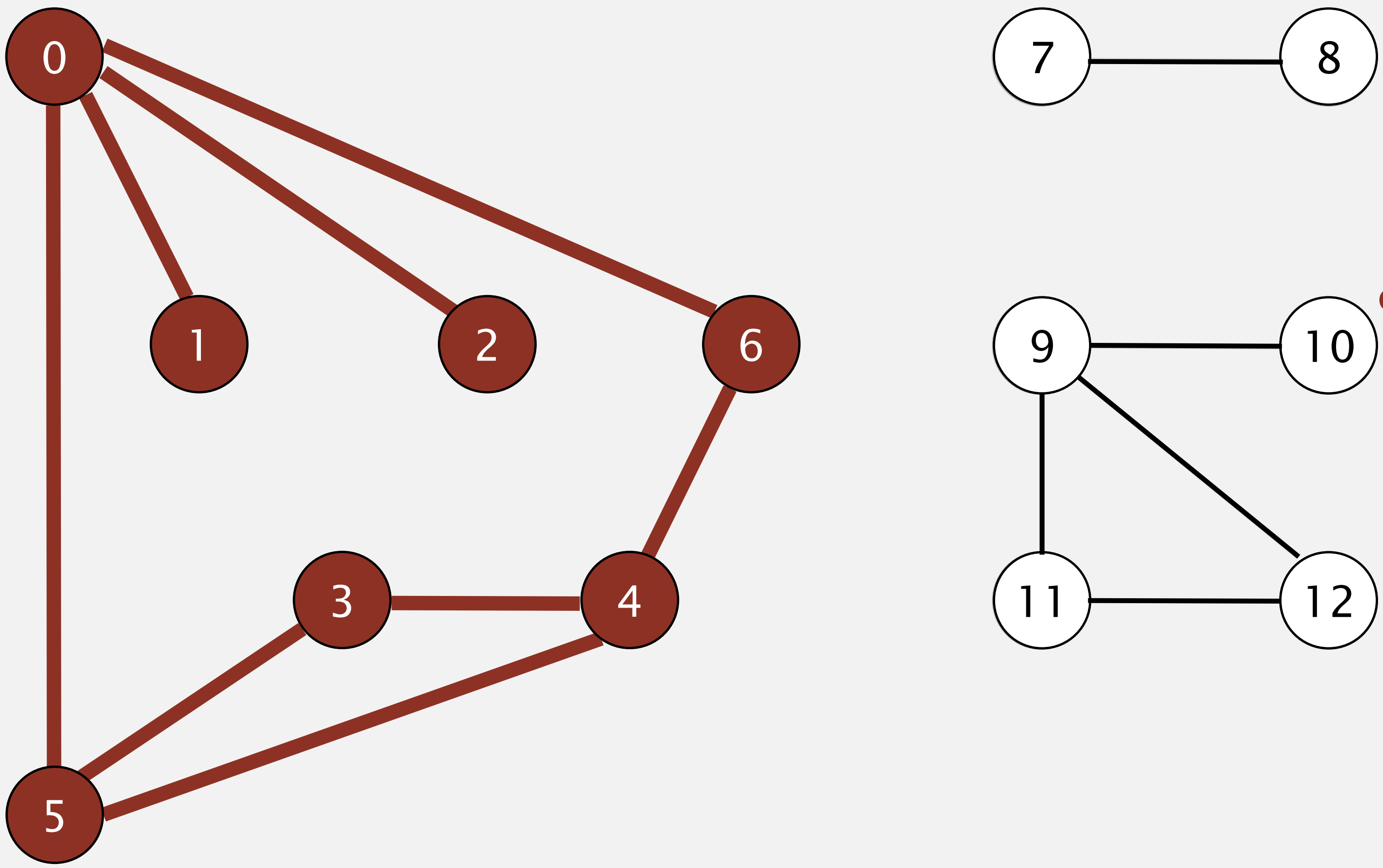
0 done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



connected component: 0 1 2 3 4 5 6

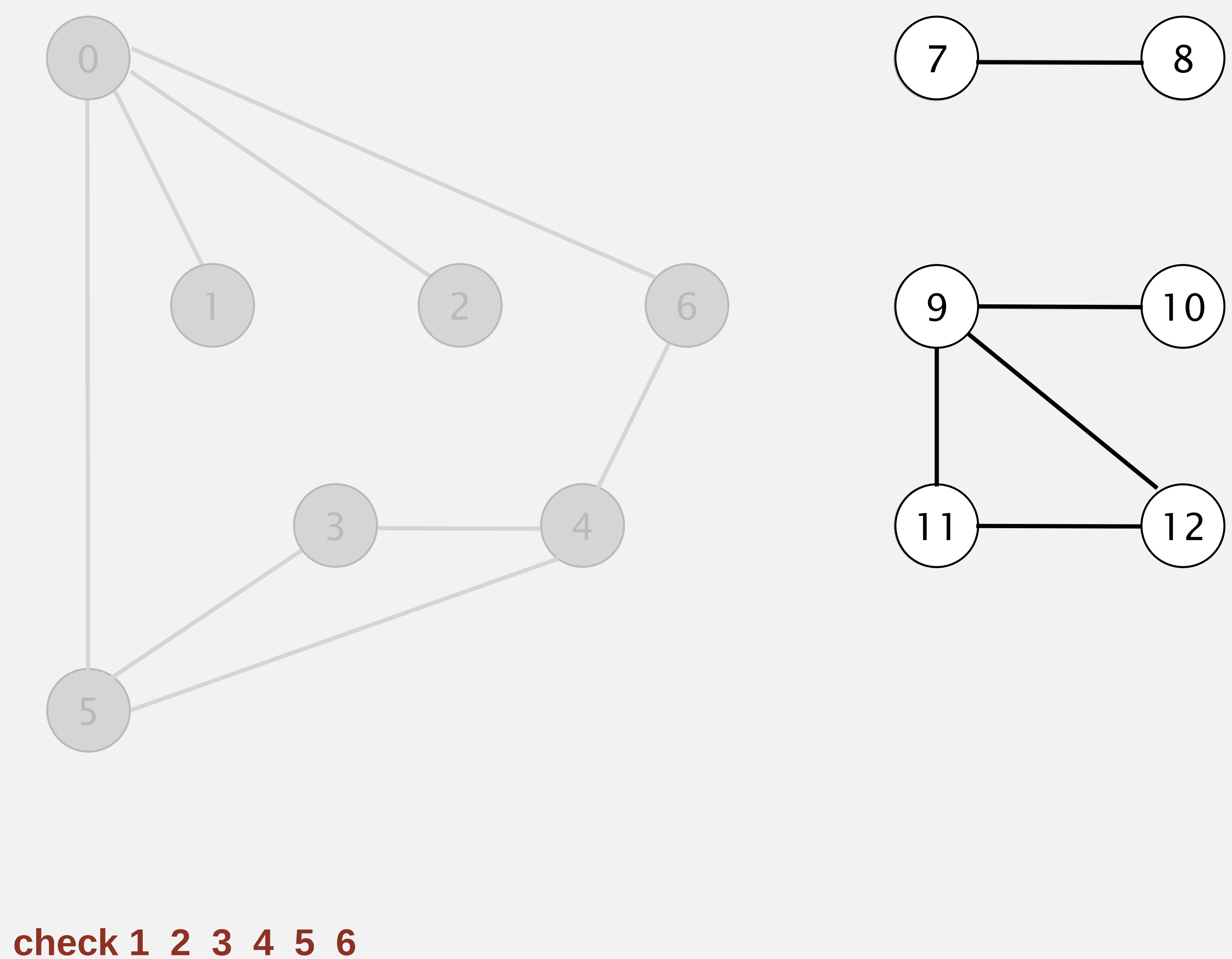
connected component

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

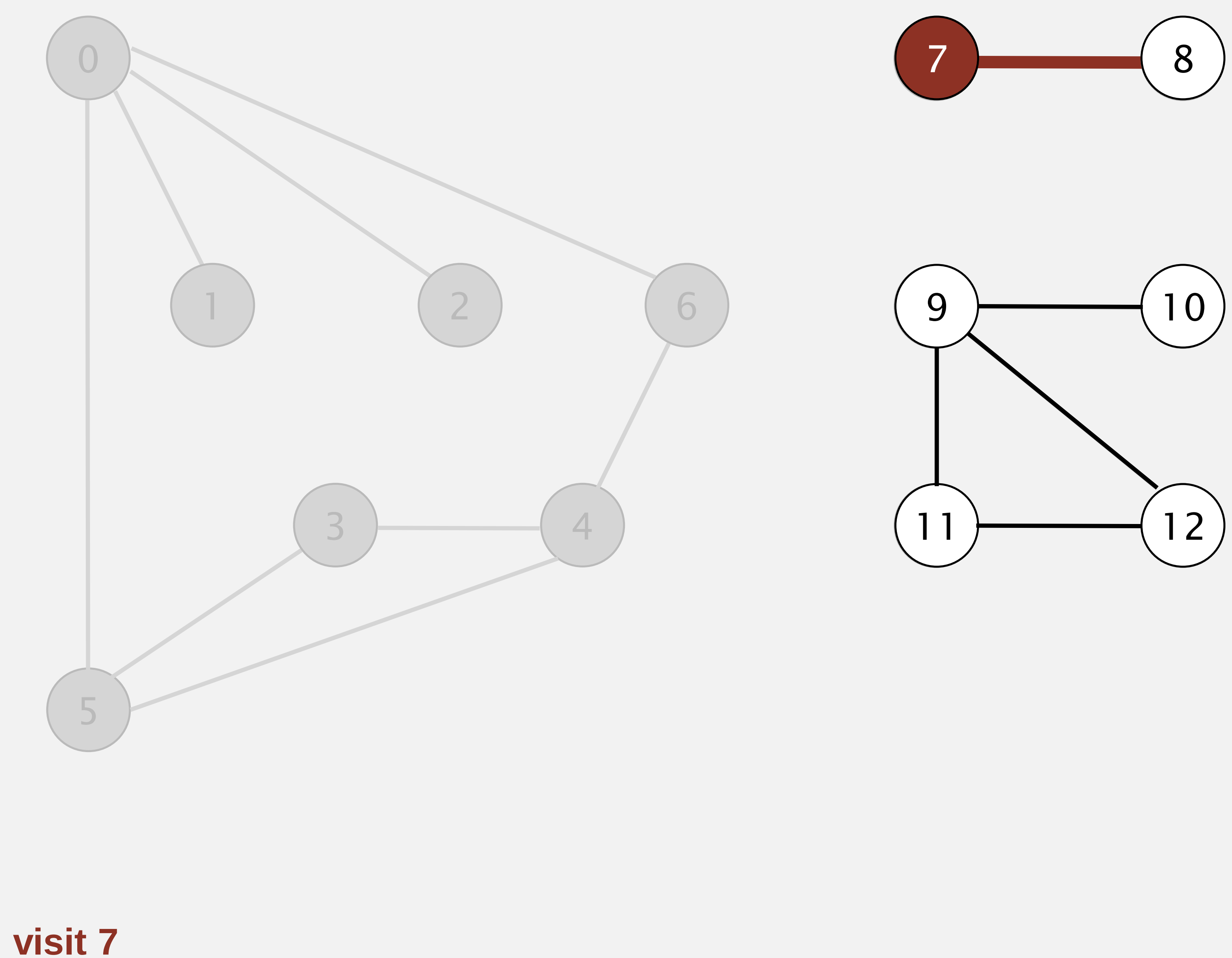


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

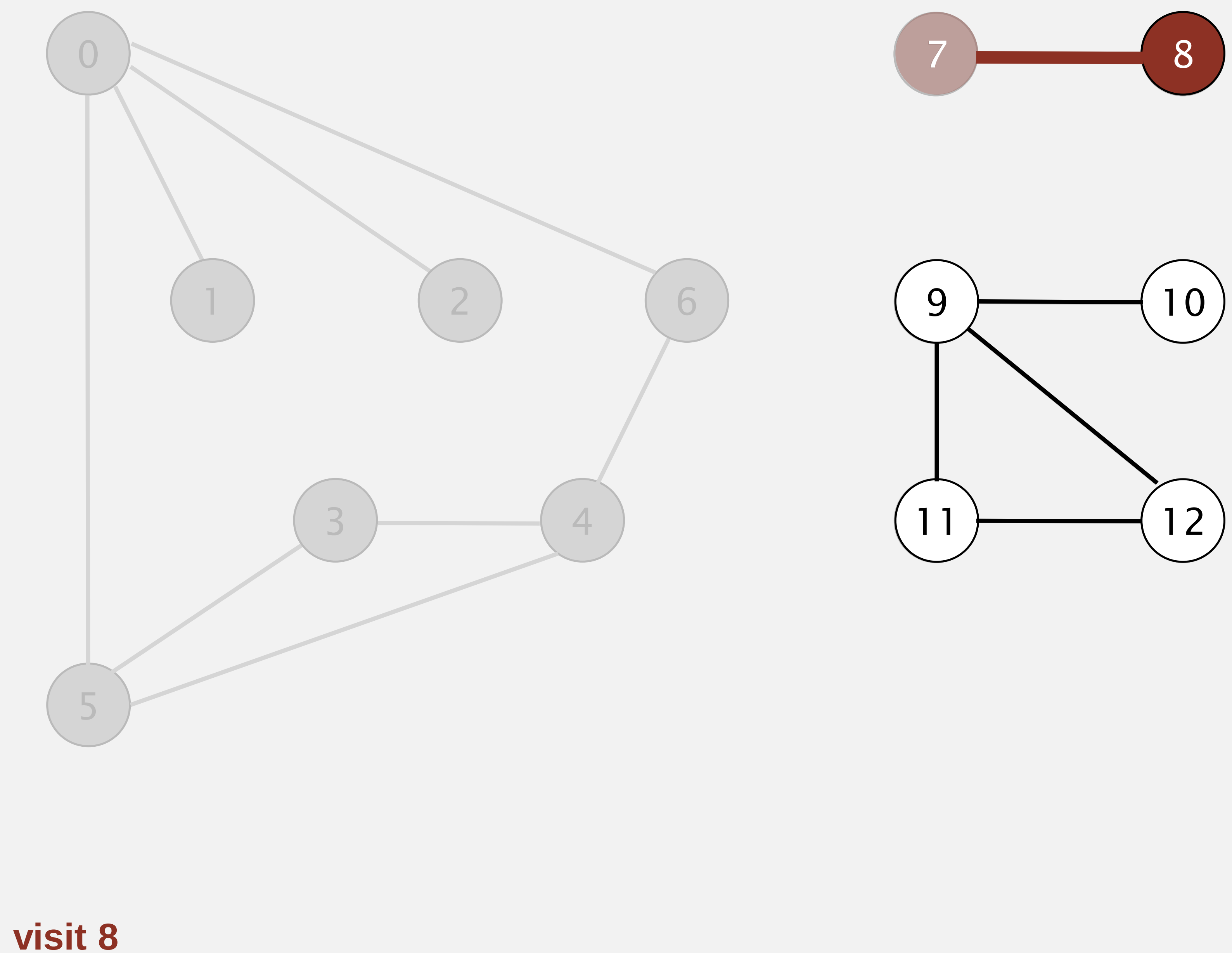


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

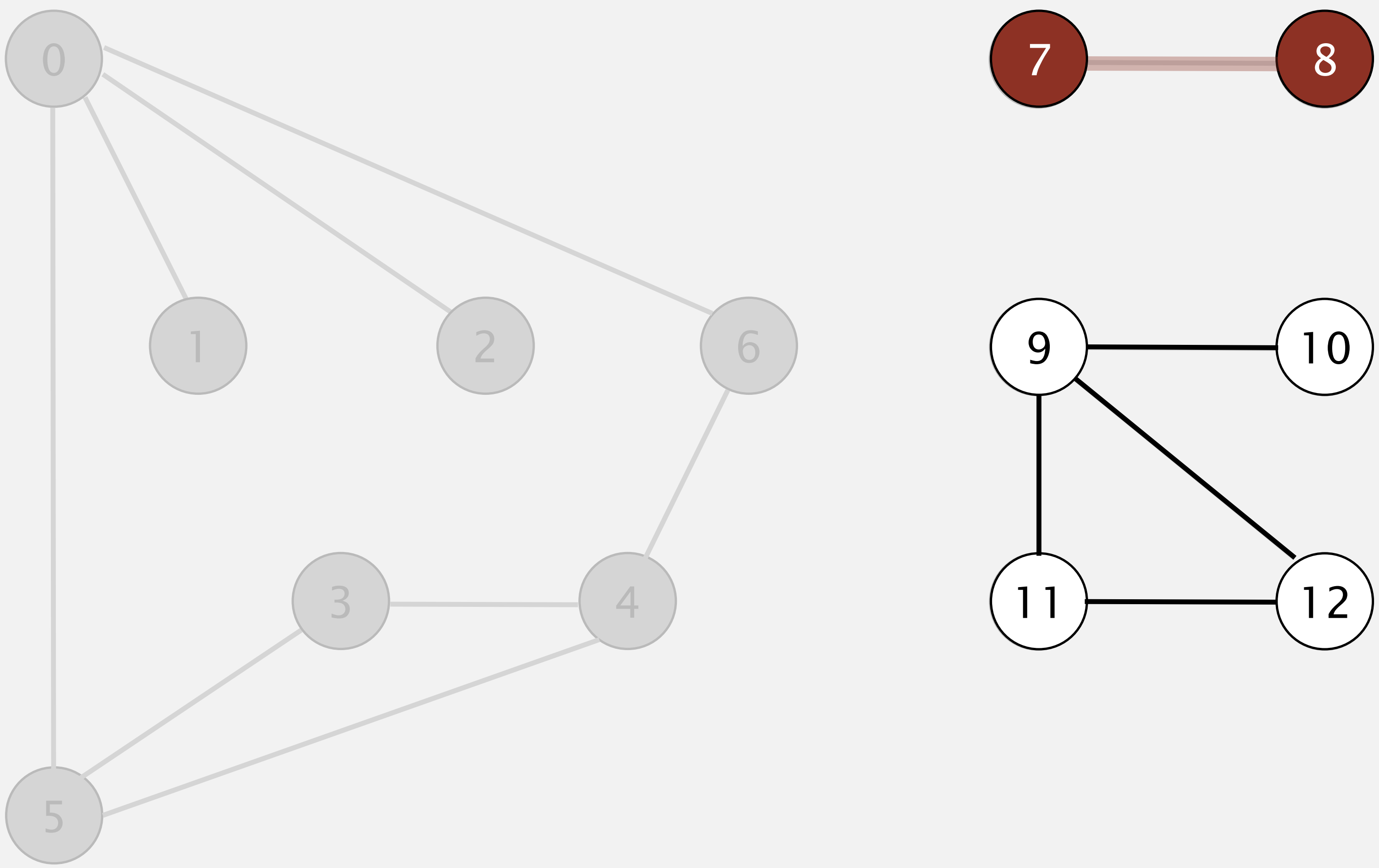


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



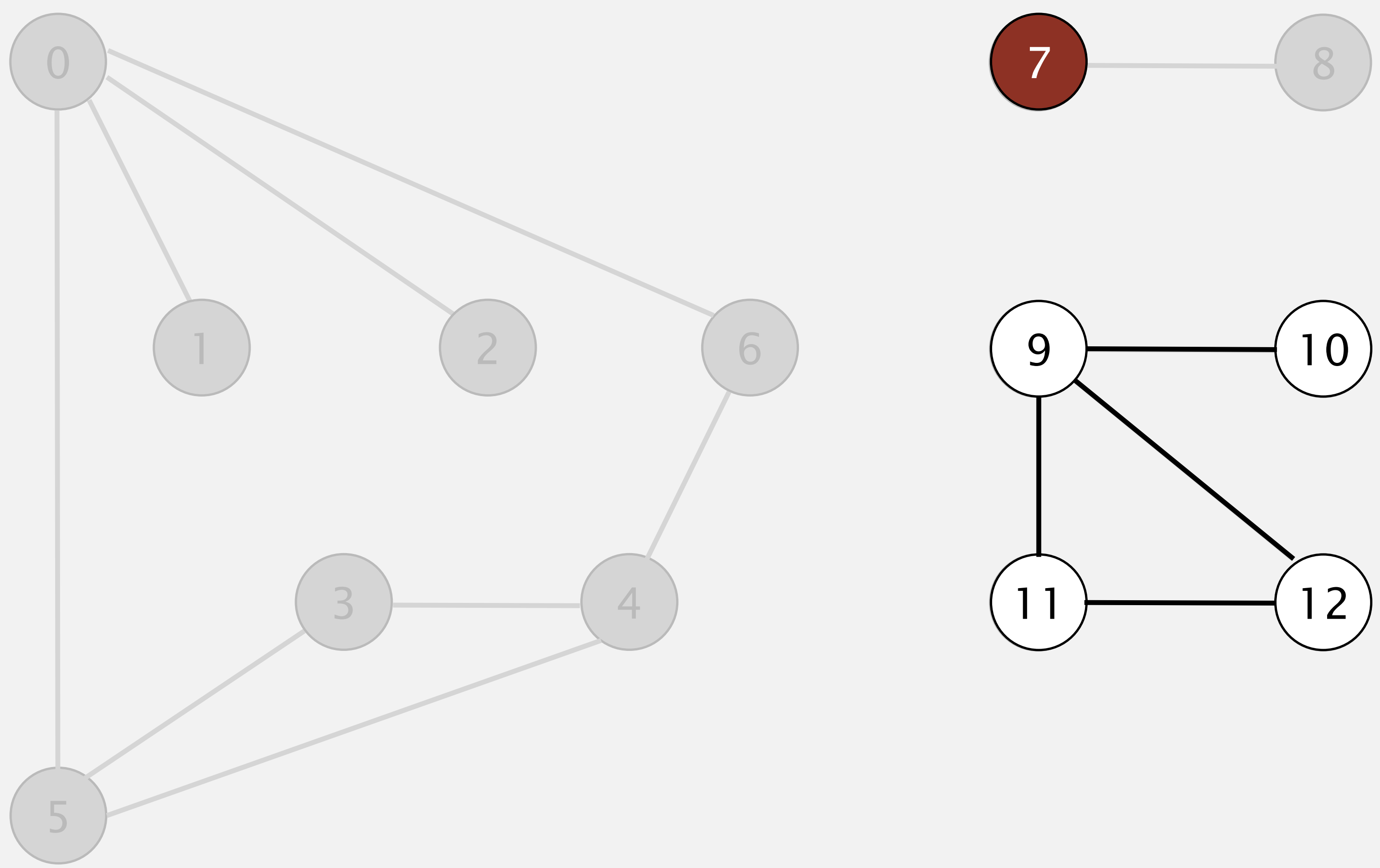
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

8 done

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



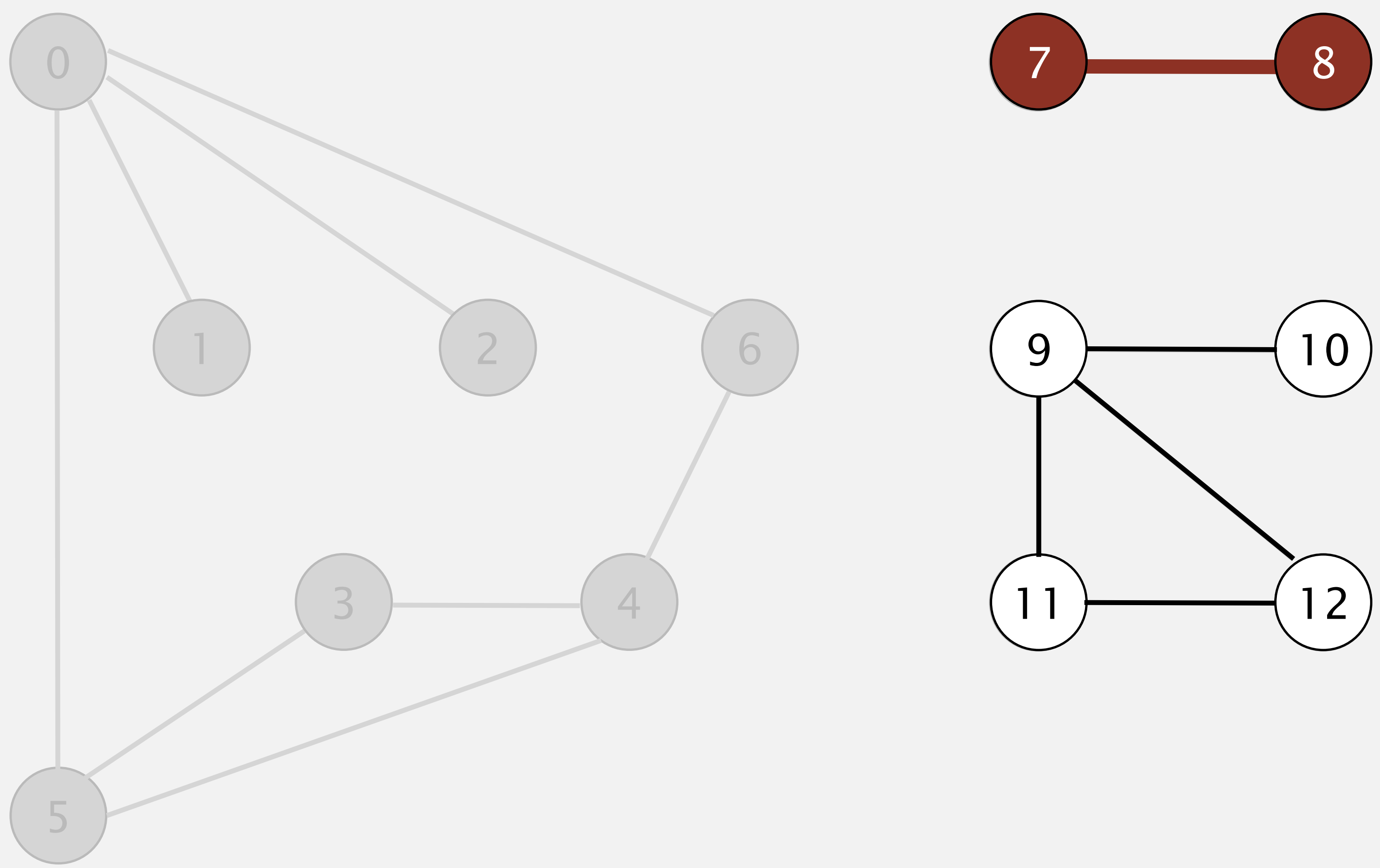
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

7 done

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



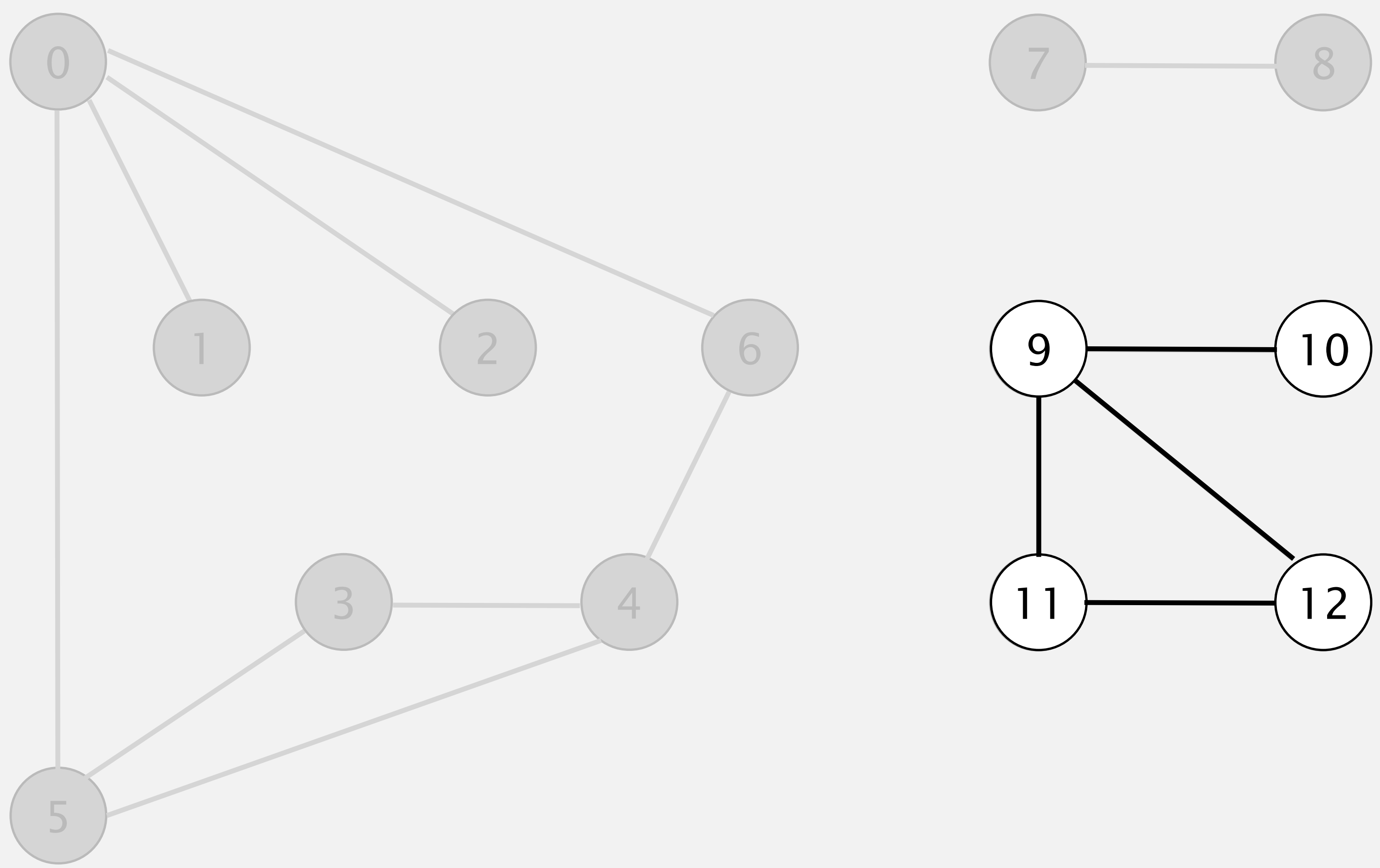
connected component: 7 8

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



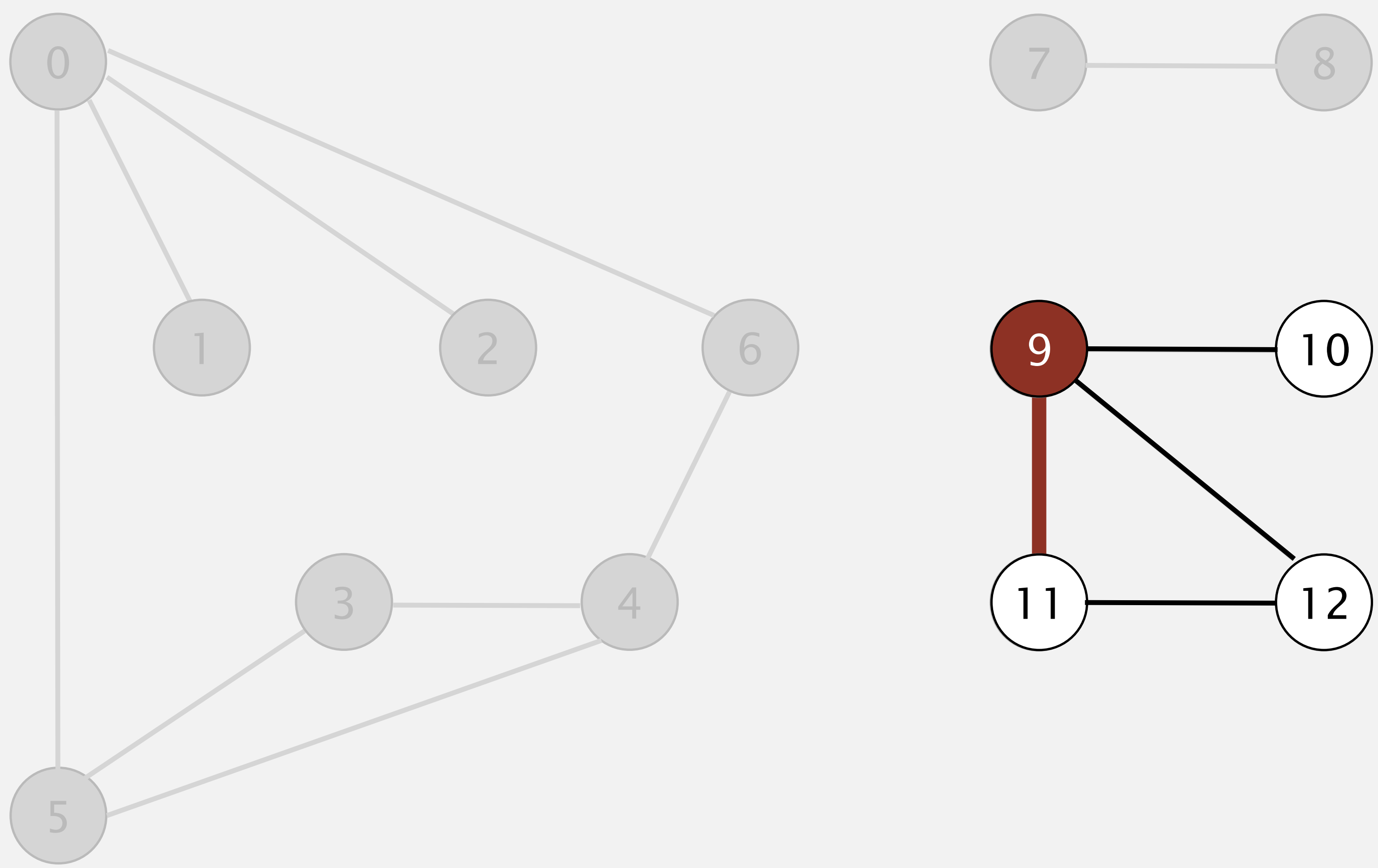
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

check 8

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



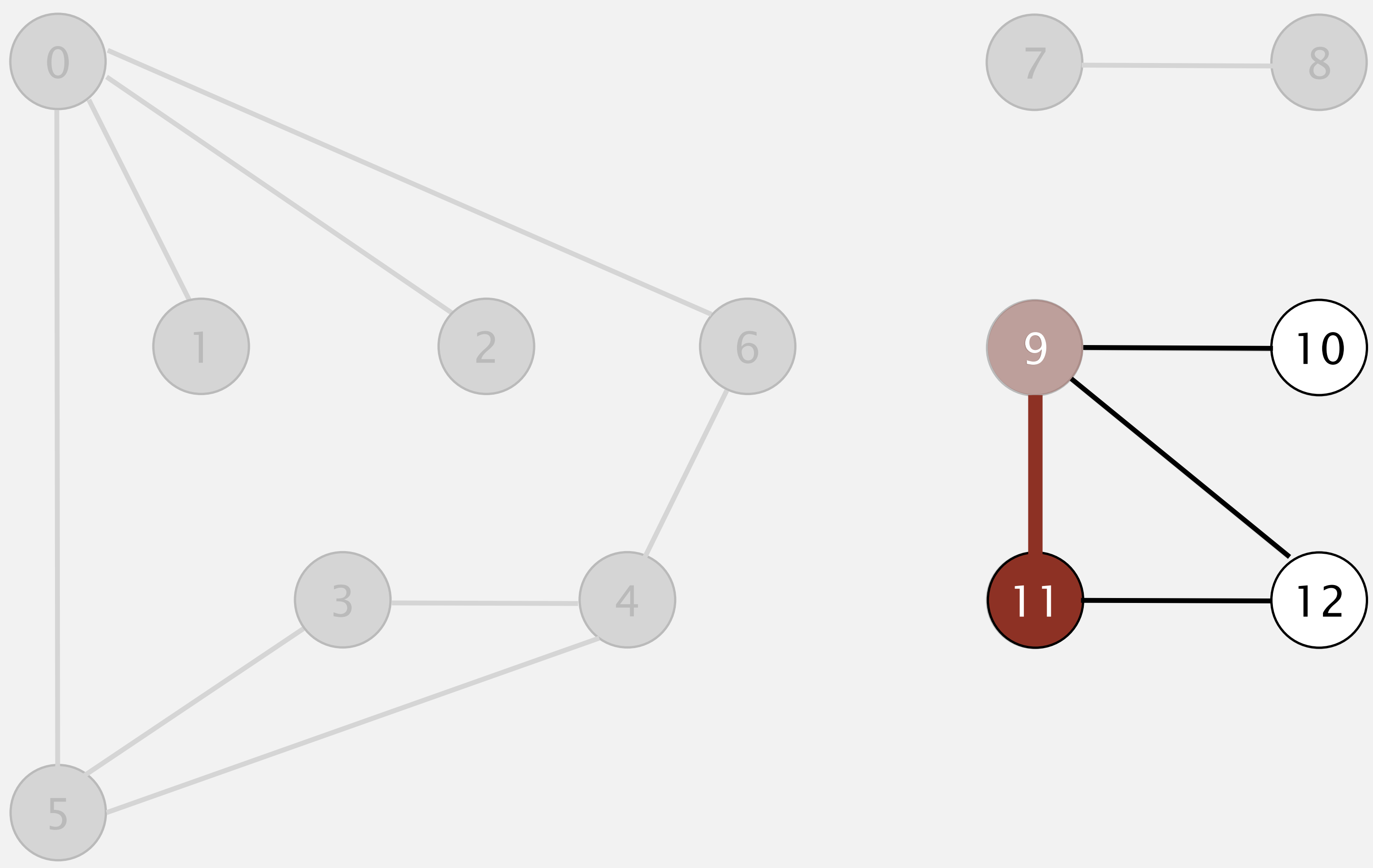
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	F	–
12	F	–

visit 9

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



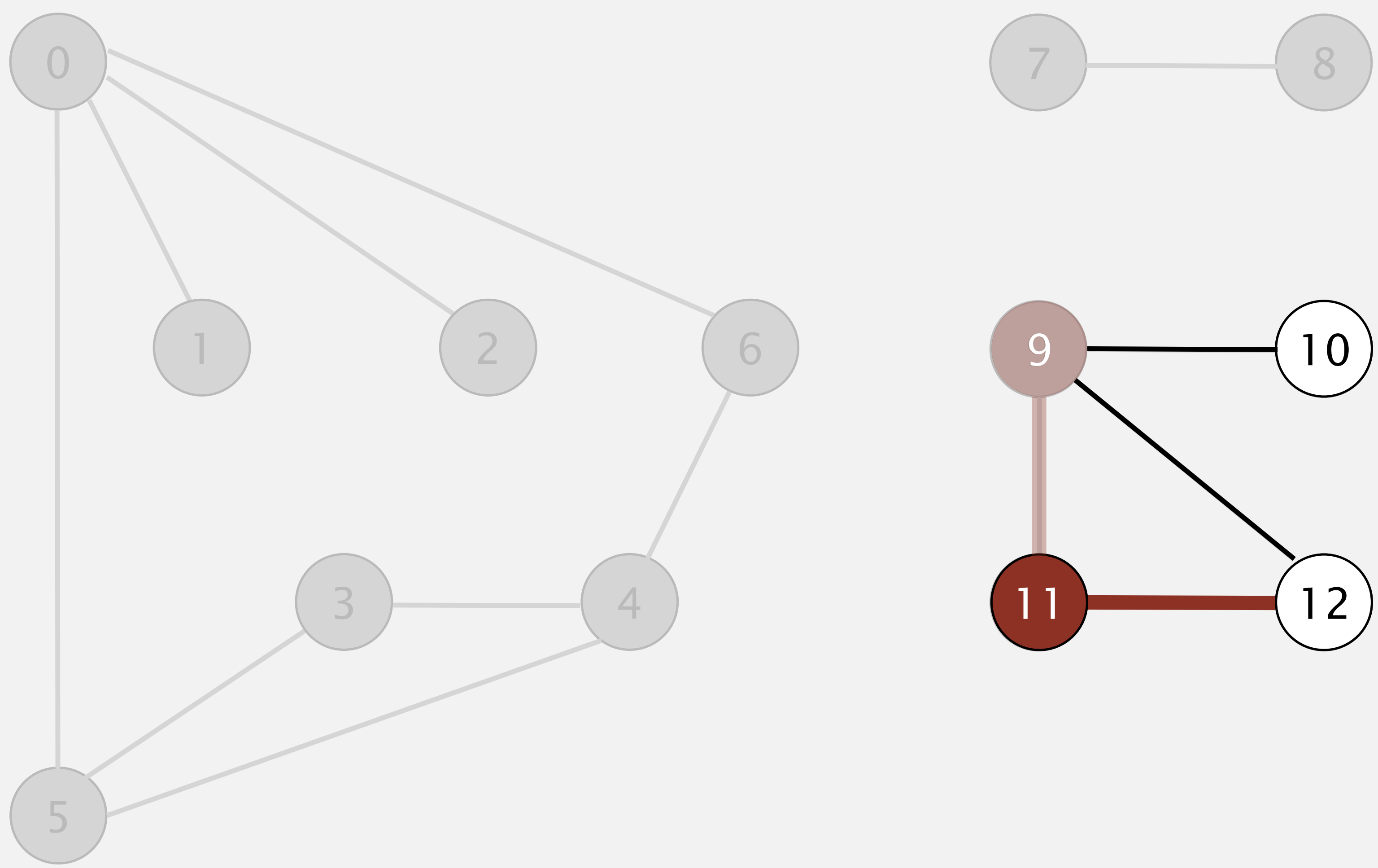
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	F	–

visit 11

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



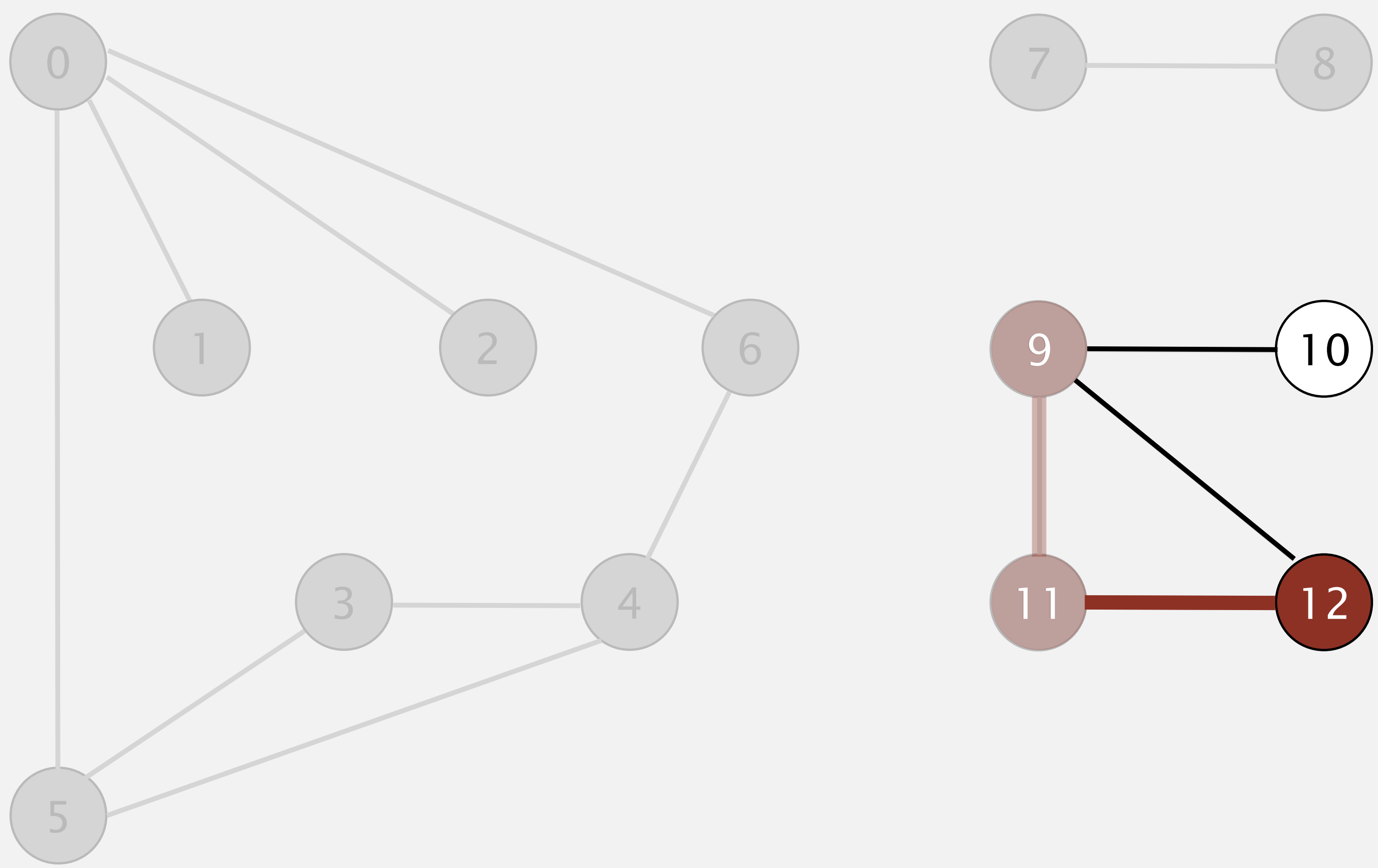
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	F	–

visit 11

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



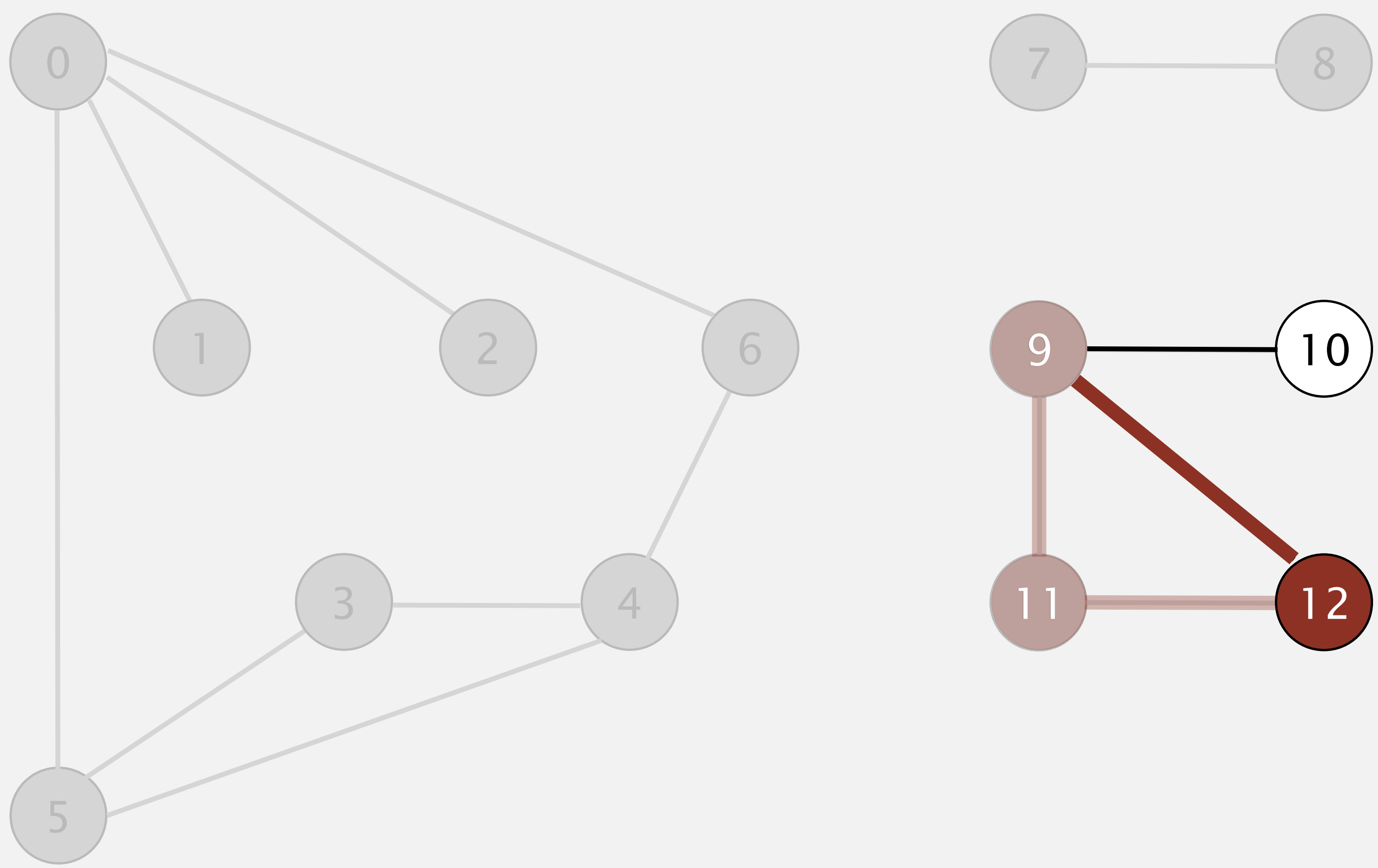
visit 12

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



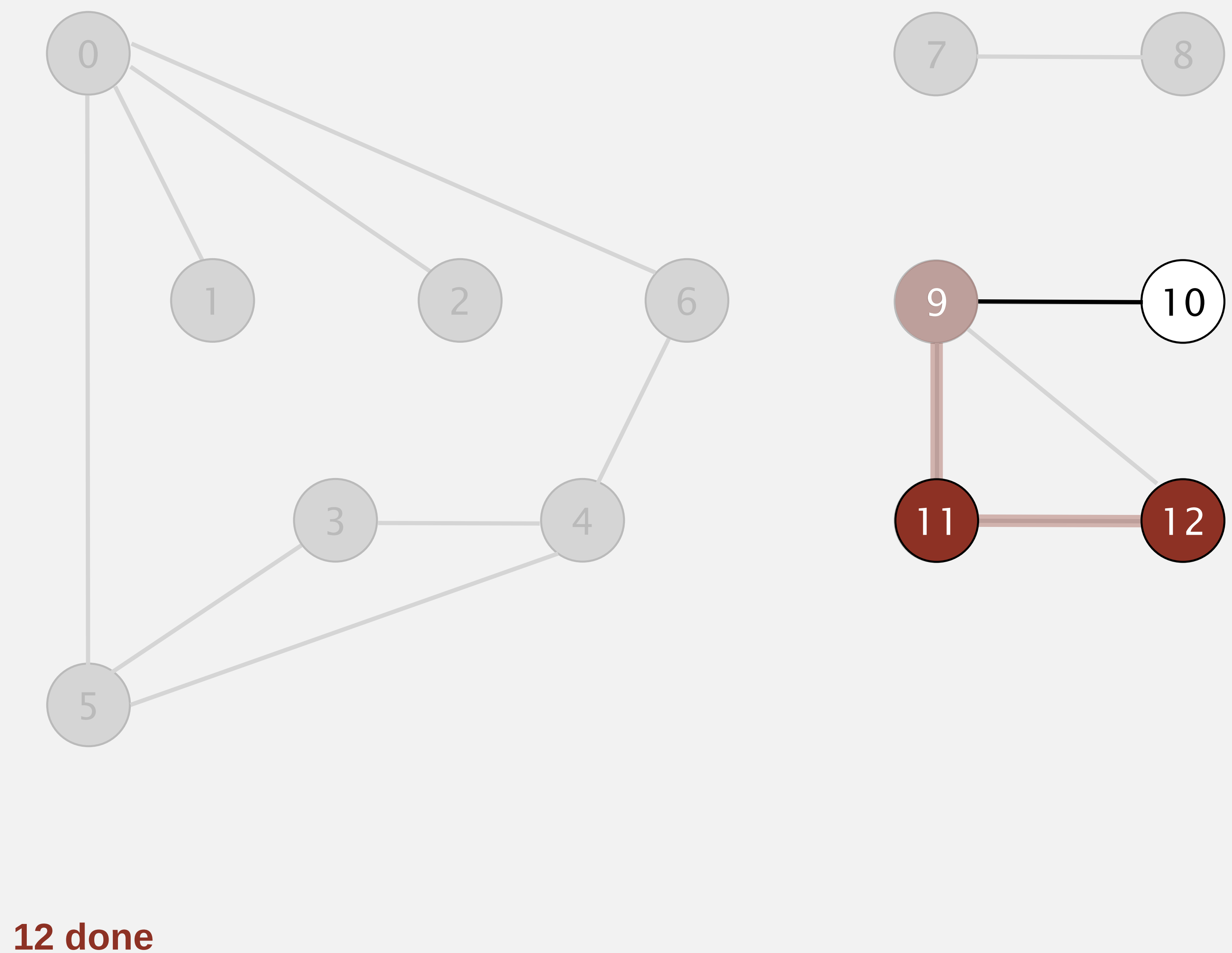
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	T	2

visit 12

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

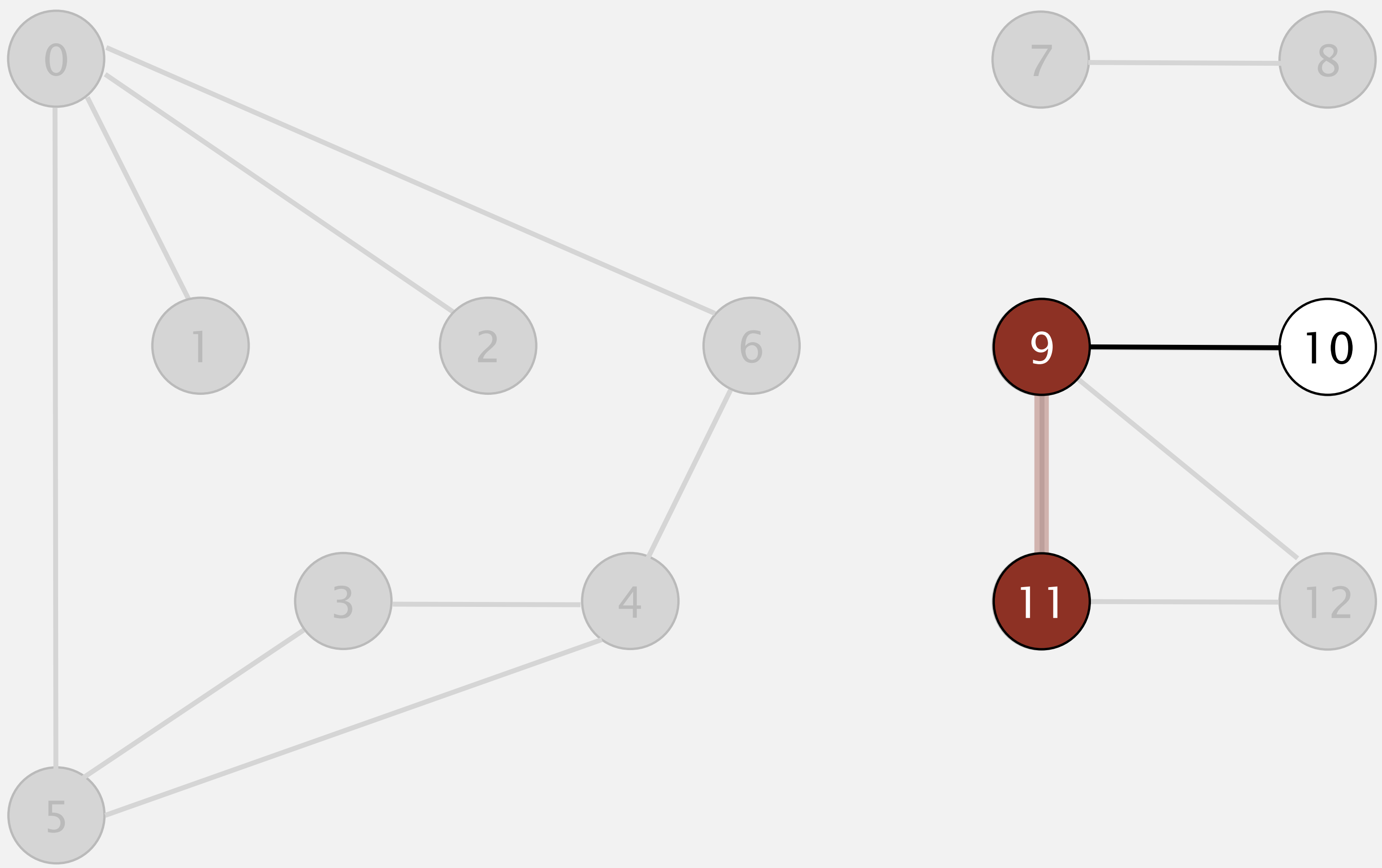


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



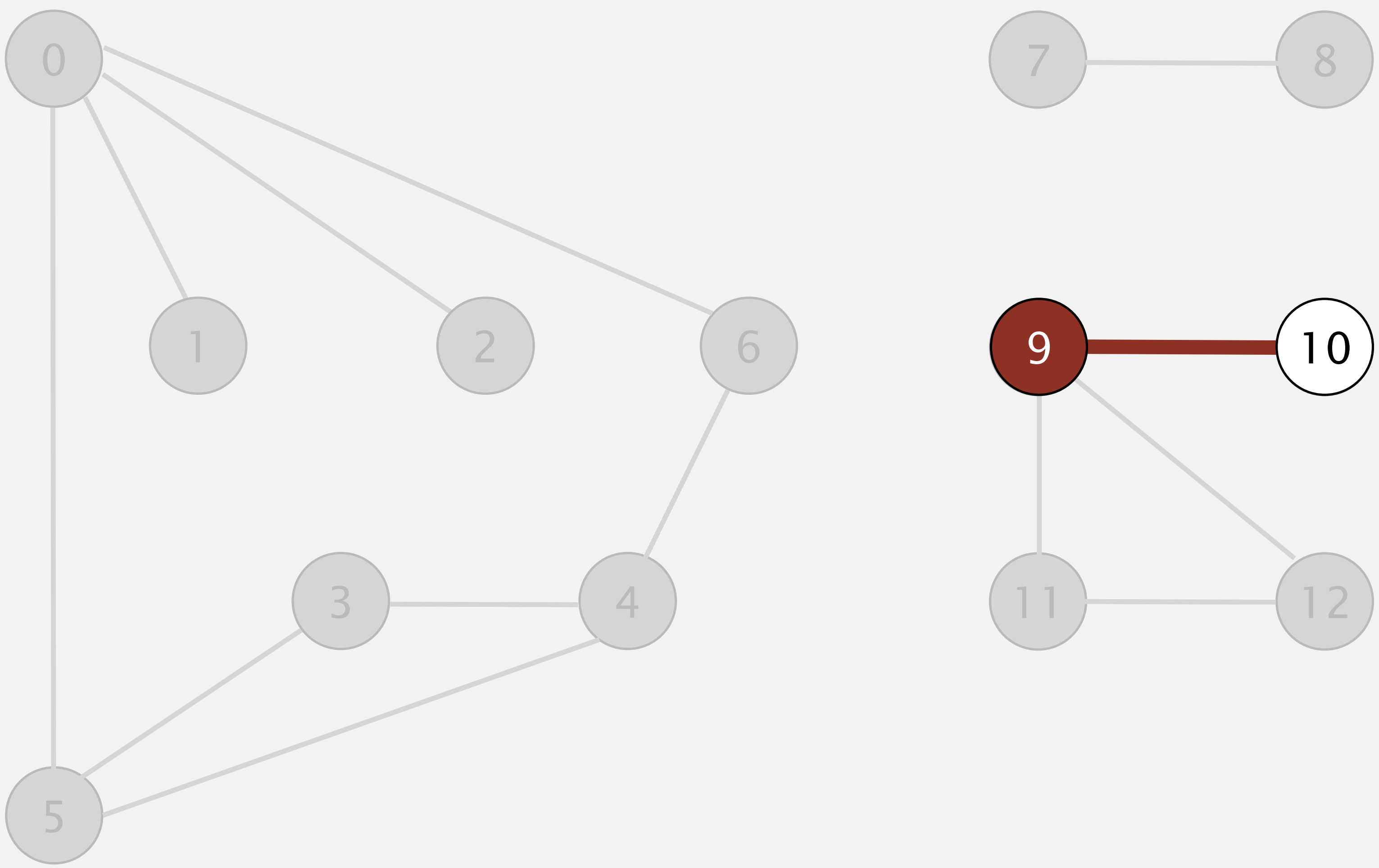
11 done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



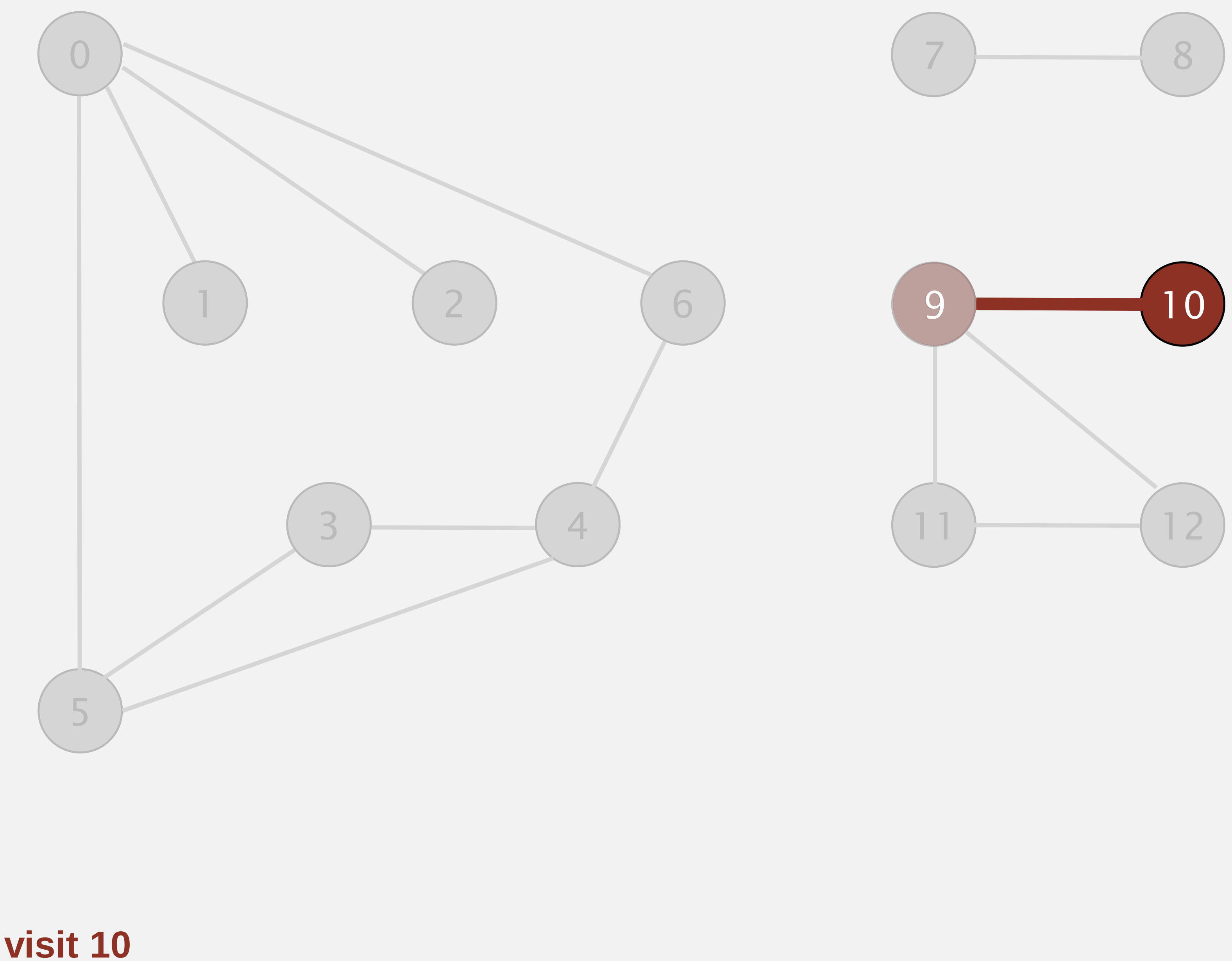
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	–
11	T	2
12	T	2

visit 9

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

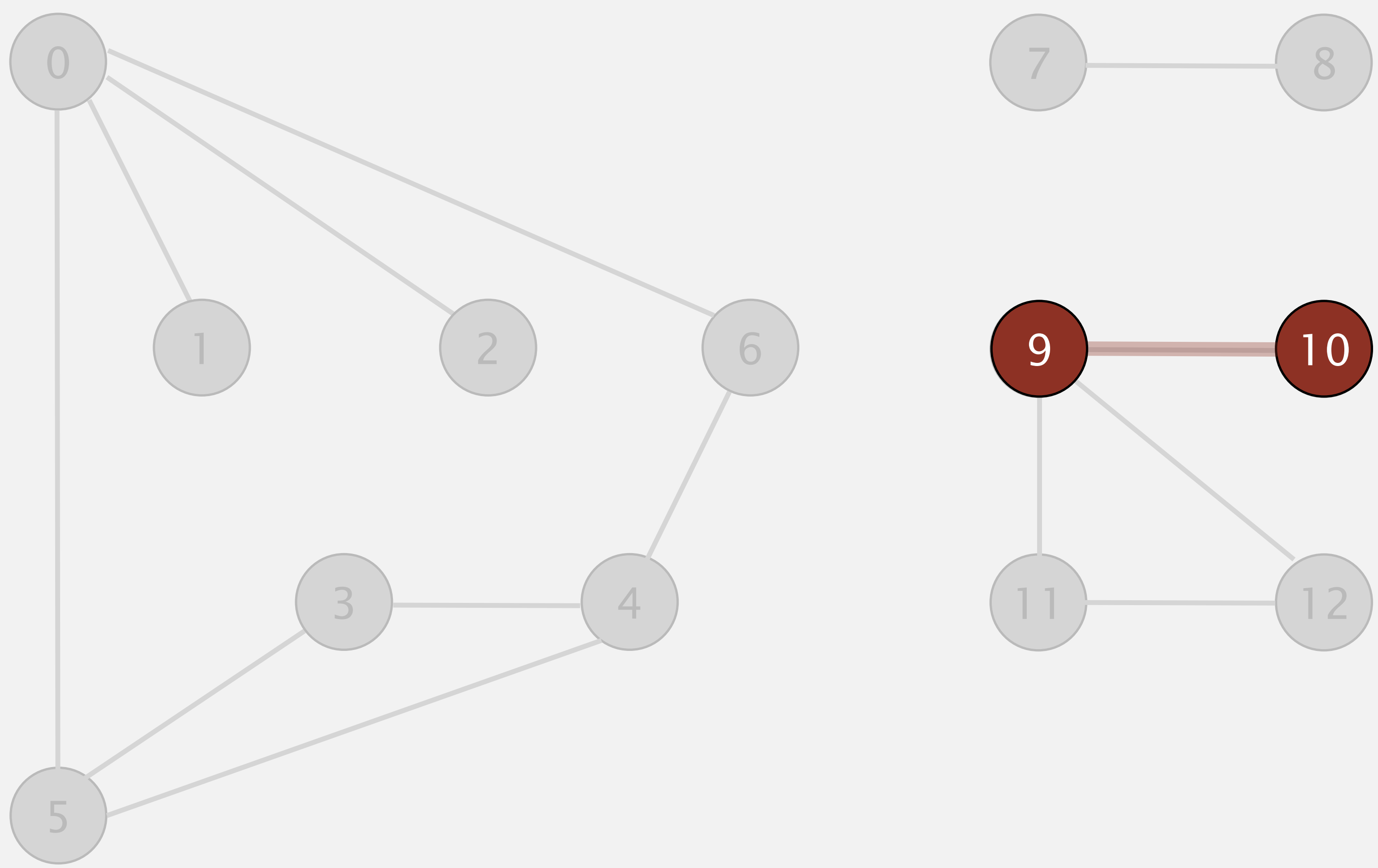


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



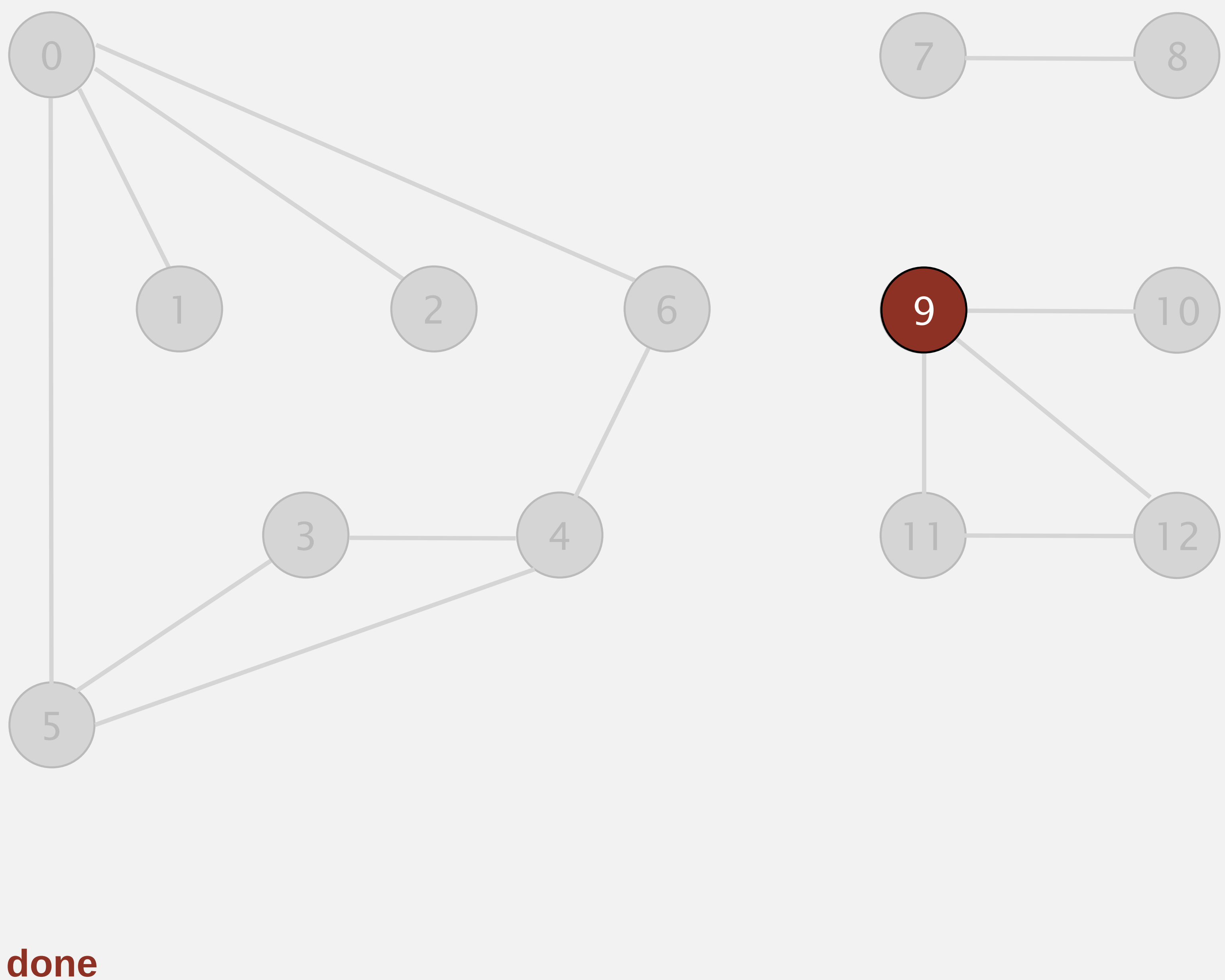
v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

10 done

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

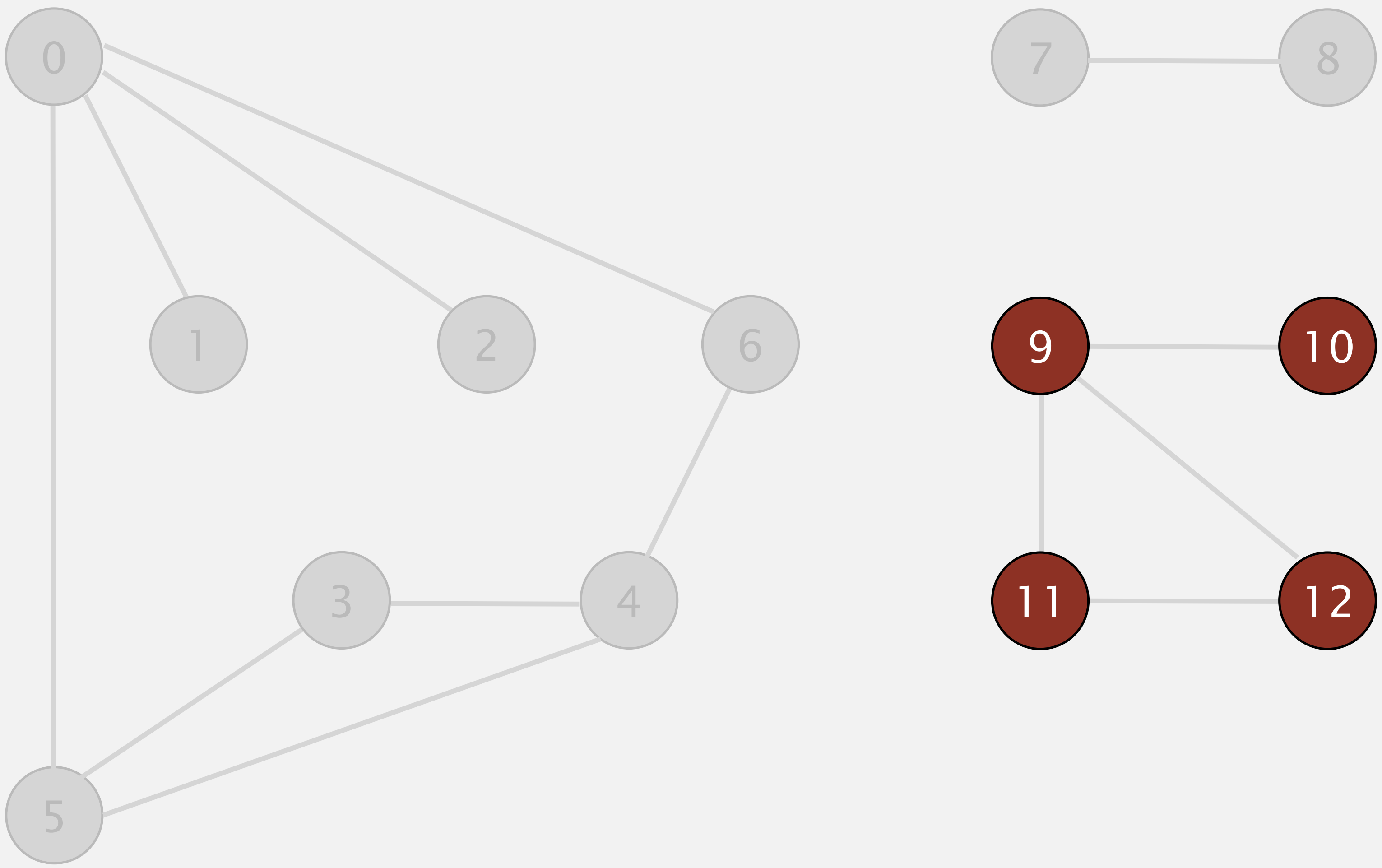


v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



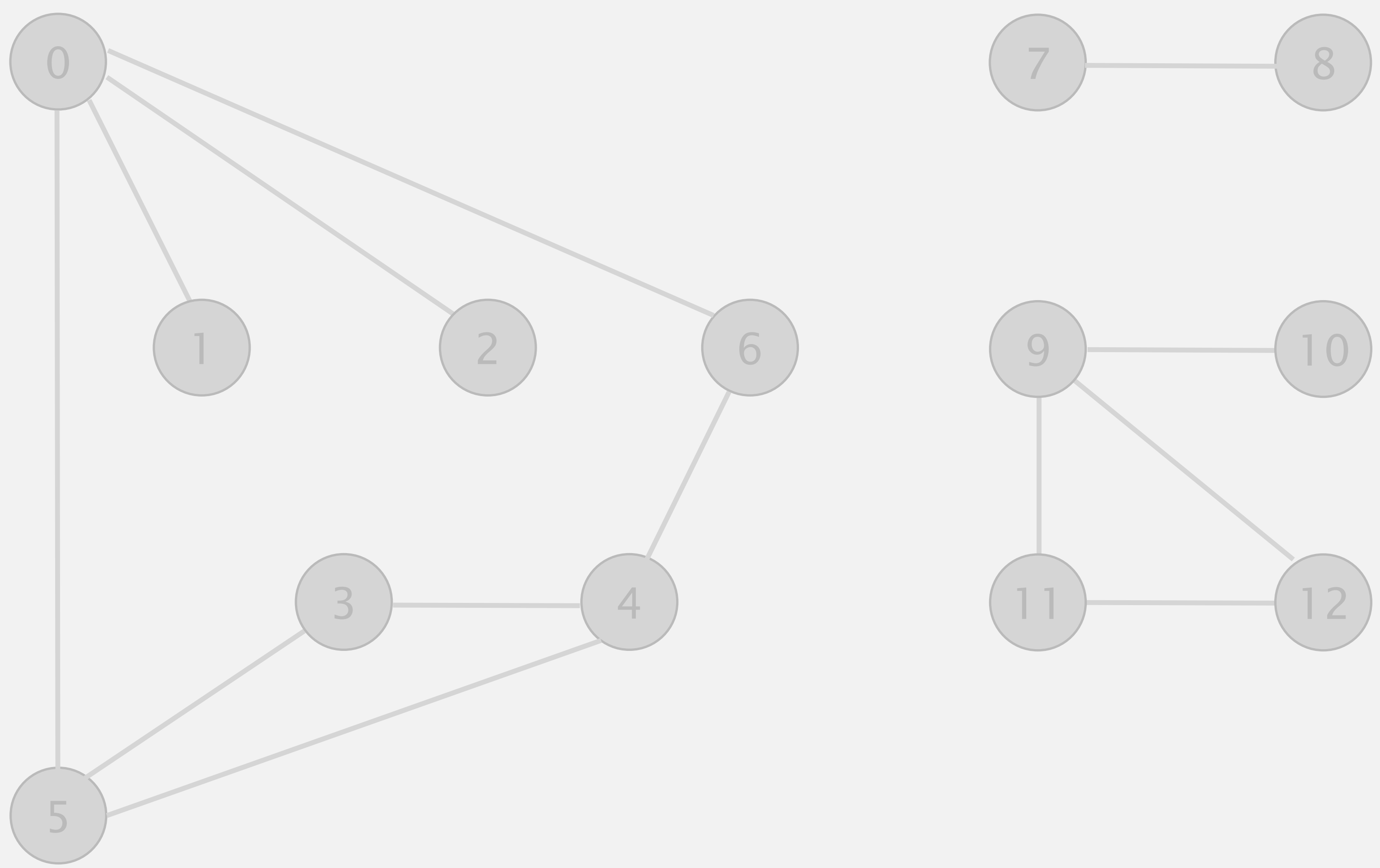
connected component: 9 10 11 12

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



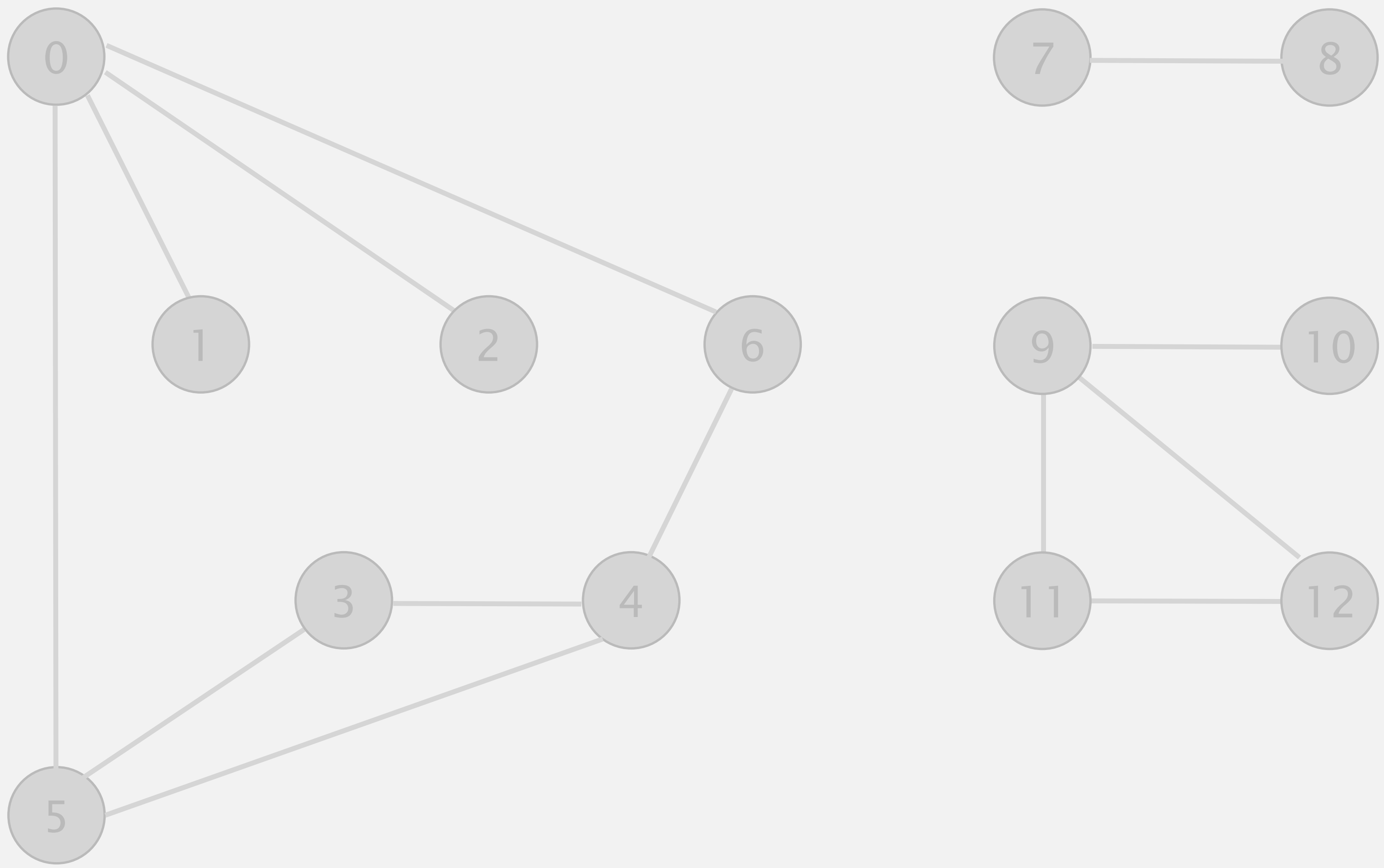
check 10 11 12

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

http://algs4.cs.princeton.edu

DIRECTED GRAPHS

Modified by: Dr. Fahed Jubair and Dr. Ramzi Saifan

Computer Engineering Department

University of Jordan

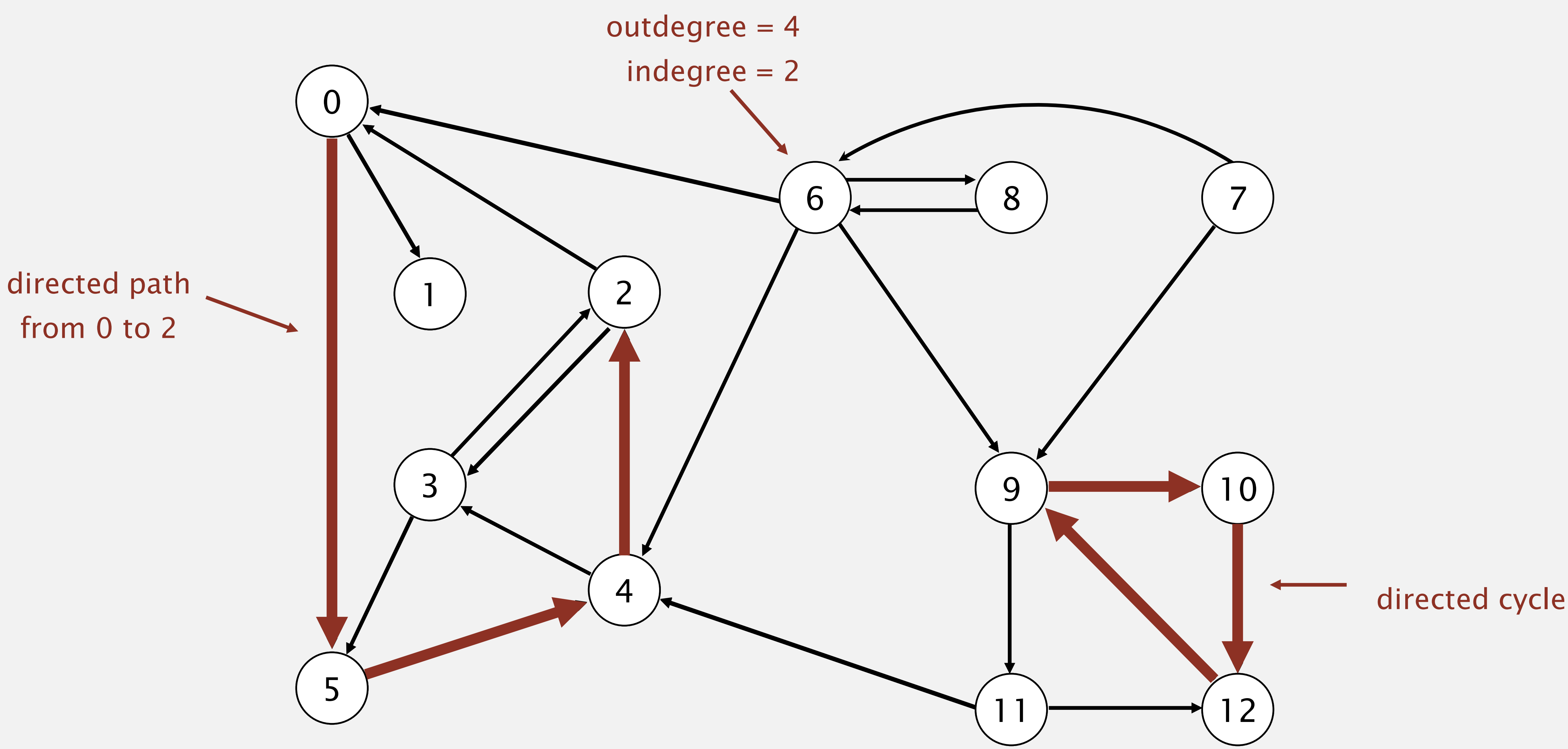
http://algs4.cs.princeton.edu

Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components (Bonus)*

Directed graphs

Digraph. Set of vertices connected pairwise by **directed** edges.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

problem	description
s→t path	*Is there a path from s to t ?*
shortest s→t path	*What is the shortest path from s to t ?*
directed cycle	*Is there a directed cycle in the graph ?*
topological sort	*Can the digraph be drawn so that all edges point upwards?*
strong connectivity	*Is there a directed path between all pairs of vertices ?*
transitive closure	*For which vertices v and w is there a directed path from v to w ?*
PageRank	*What is the importance of a web page ?*

http://algs4.cs.princeton.edu

Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components (Bonus)*

Digraph API

Almost identical to `Graph` API.

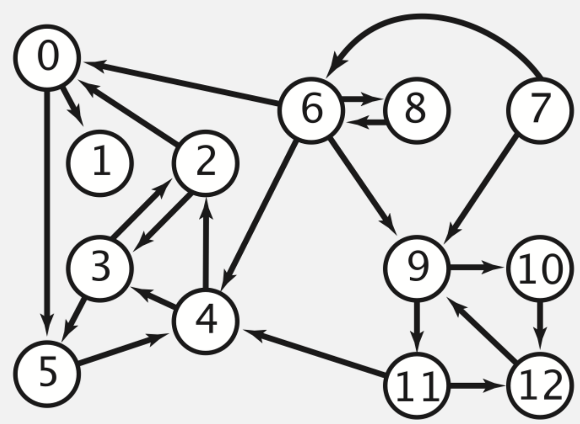
`public class Digraph`		
	`Digraph(int V)`	*create an empty digraph with V vertices*
	`Digraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(int v, int w)`	*add a directed edge v→w*
`Iterable<Integer>`	`adj(int v)`	*vertices pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Digraph`	`reverse()`	*reverse of this digraph*
`String`	`toString()`	*string representation*

Digraph API

tinyDG.txt

V → 13 (number of vertices), *E* → 22 (number of edges)

```
13
22
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 ⋮
```




```
% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
⋮
11->4
11->12
12->9
```

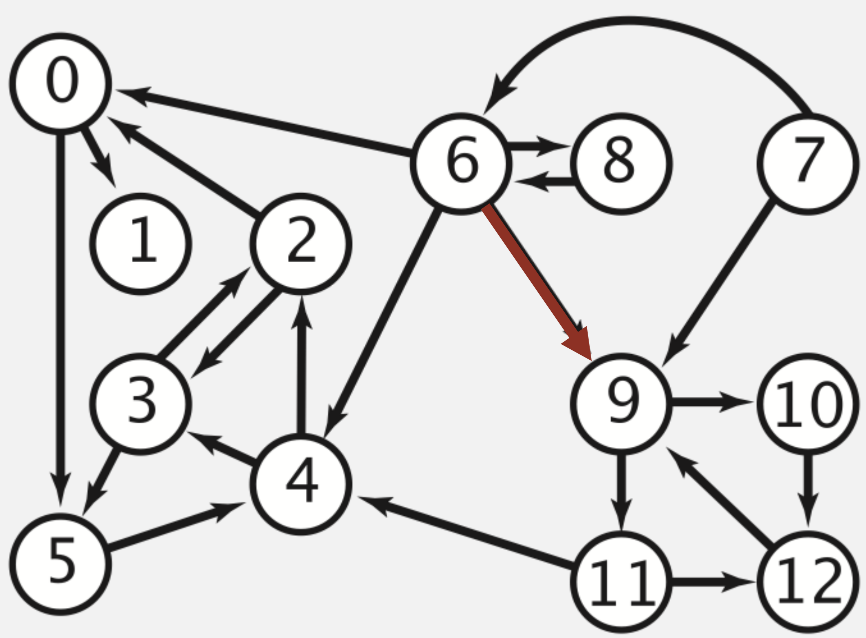
```
In in = new In(args[0]);
Digraph G = new Digraph(in);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "->" + w);
```

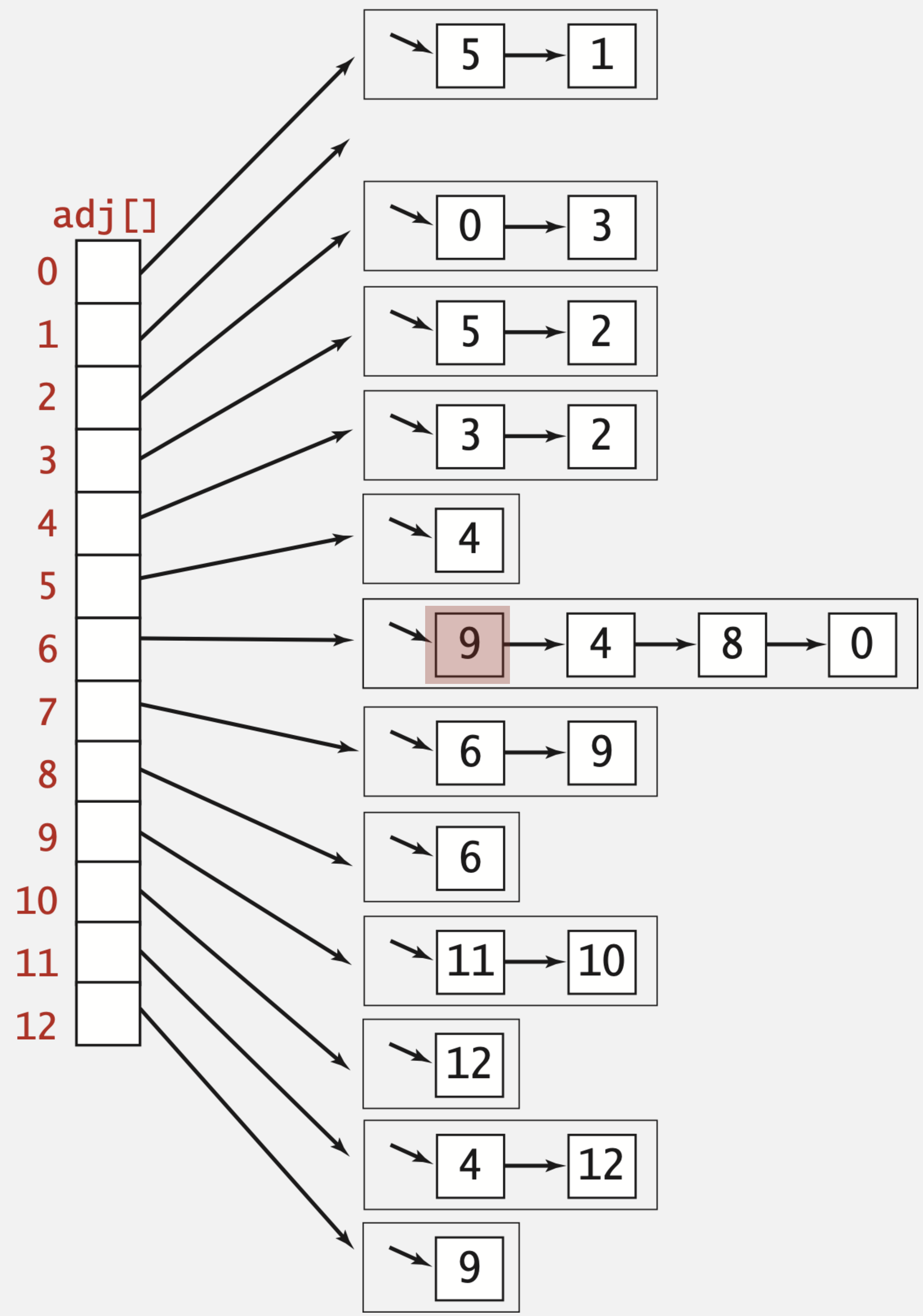
read digraph from input stream

print out each edge (once)

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.





Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V^2	$1^{\dagger}$	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

† disallows parallel edges

Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
   private final int V;
   private final Bag<Integer>[] adj;
   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }
   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }
   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists

create empty graph with V vertices

add edge v–w

iterator for vertices adjacent to v

Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
   private final int V;
   private final Bag<Integer>[] adj;
   public Digraph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }
   public void addEdge(int v, int w)
   {
      adj[v].add(w);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists

create empty digraph with V vertices

add edge v→w

iterator for vertices pointing from v

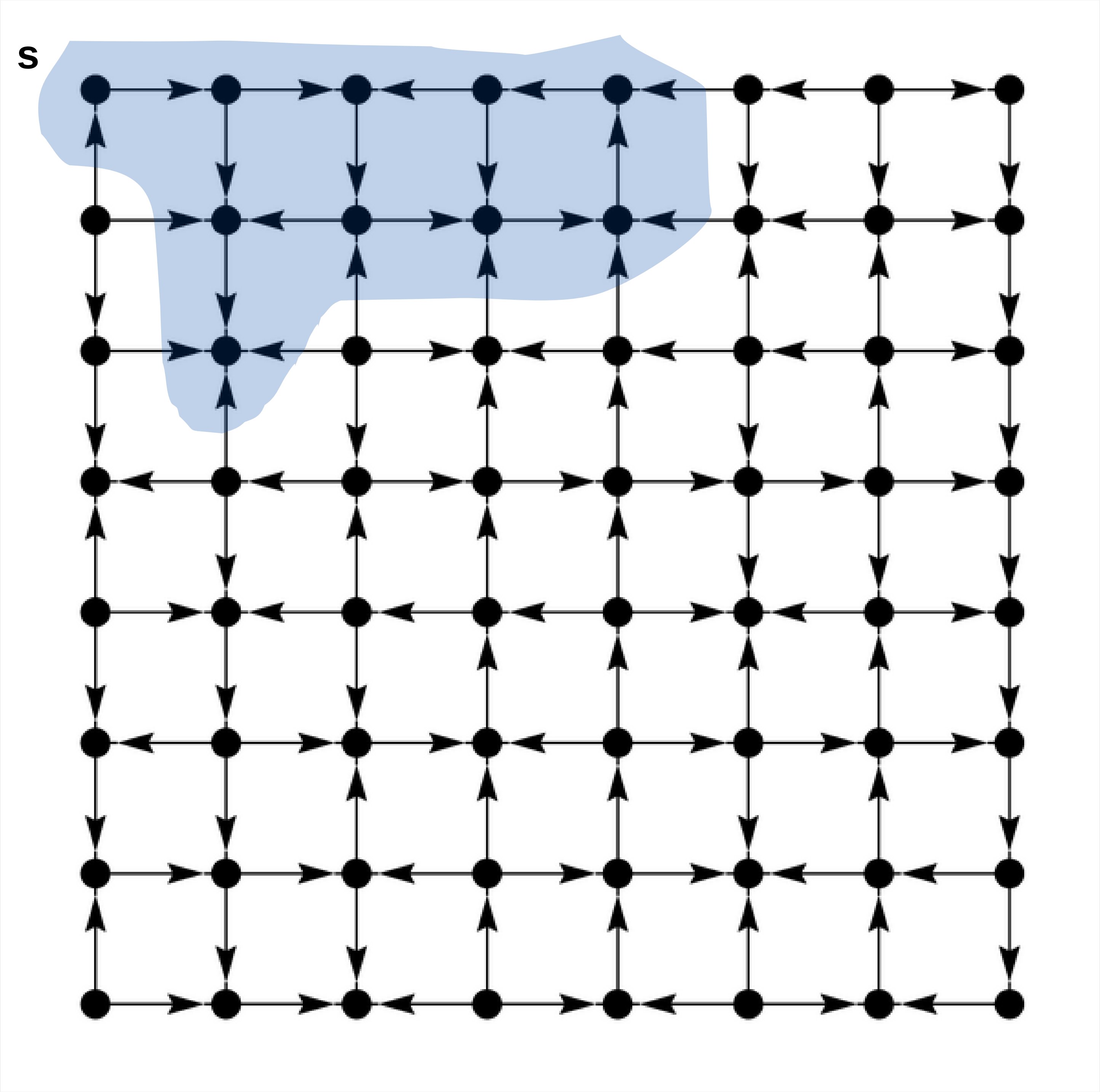
http://algs4.cs.princeton.edu

Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components (Bonus)

Reachability

Problem. Find all vertices reachable from s along a directed path.



Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

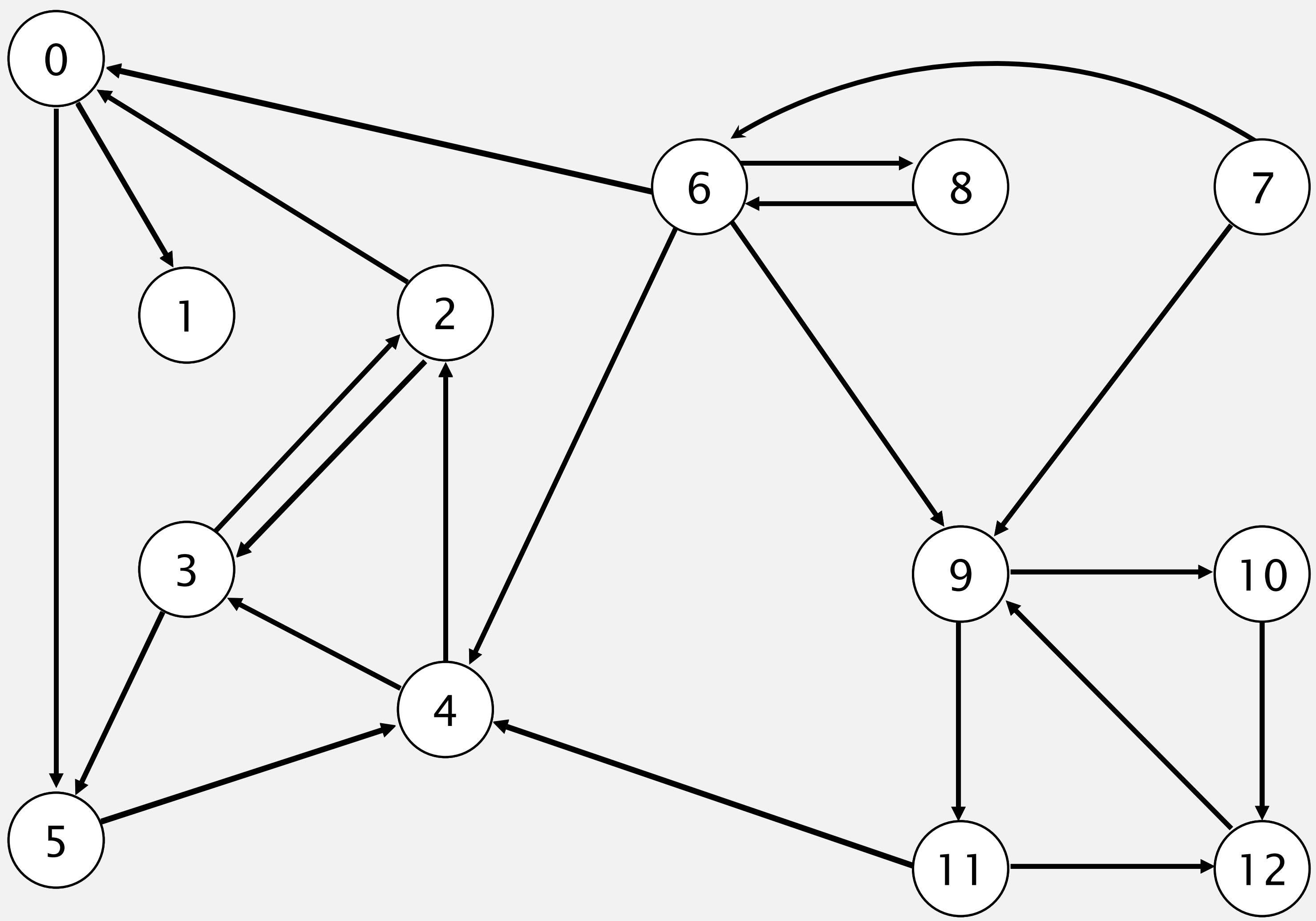
Mark v as visited.

Recursively visit all unmarked vertices w pointing from v.

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



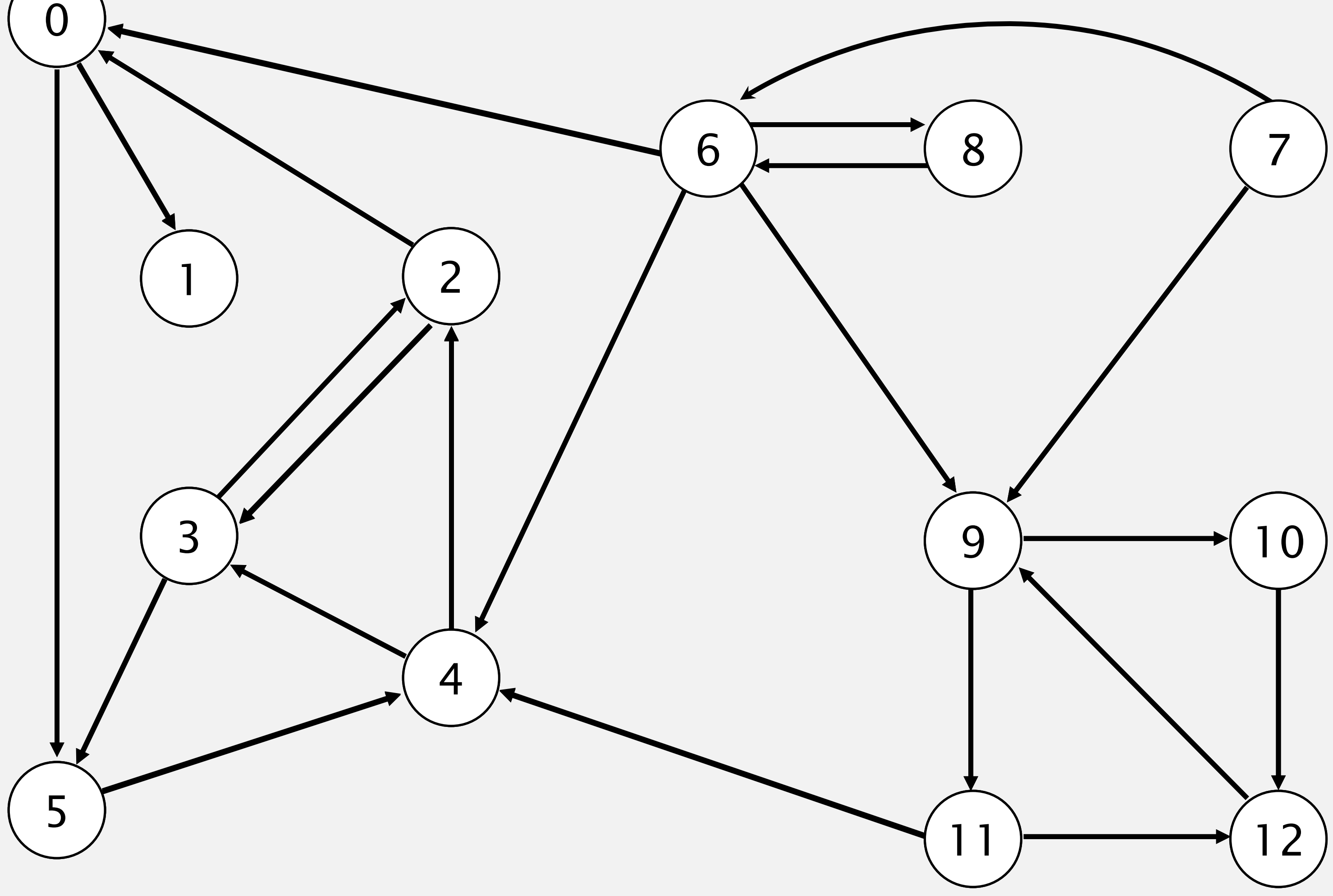
a directed graph

4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



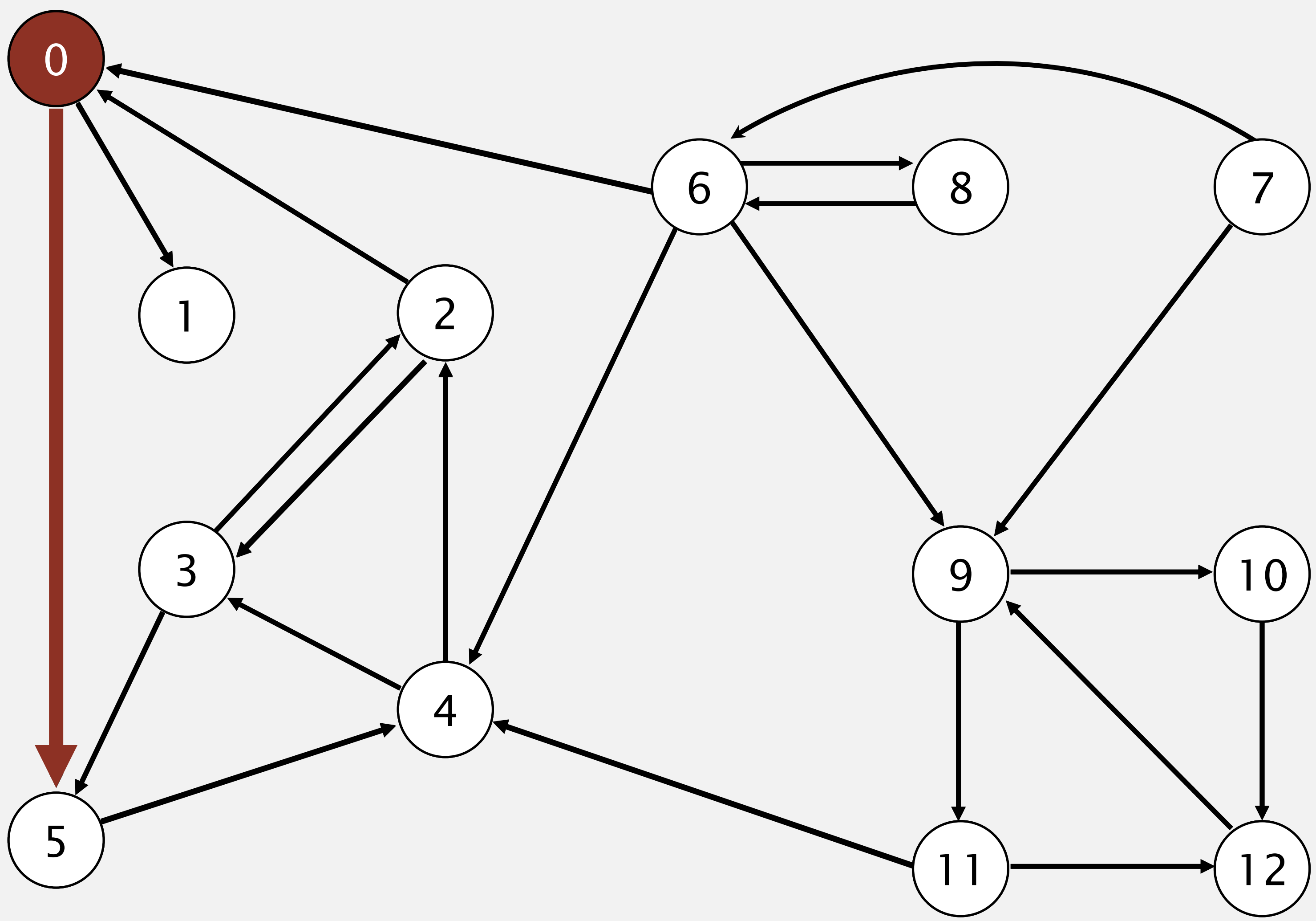
a directed graph

v	marked[]	edgeTo[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



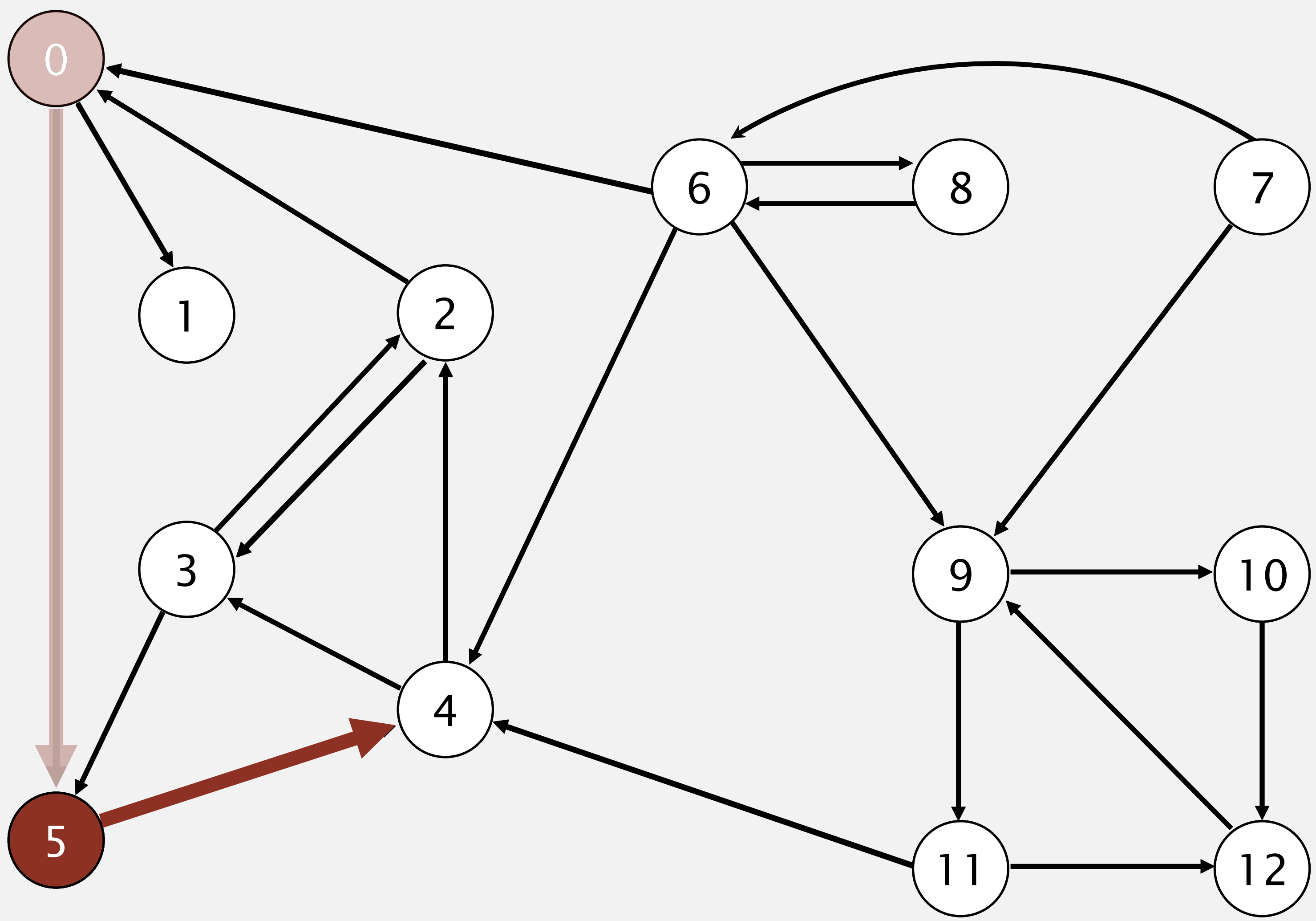
visit 0: check 5 and check 1

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



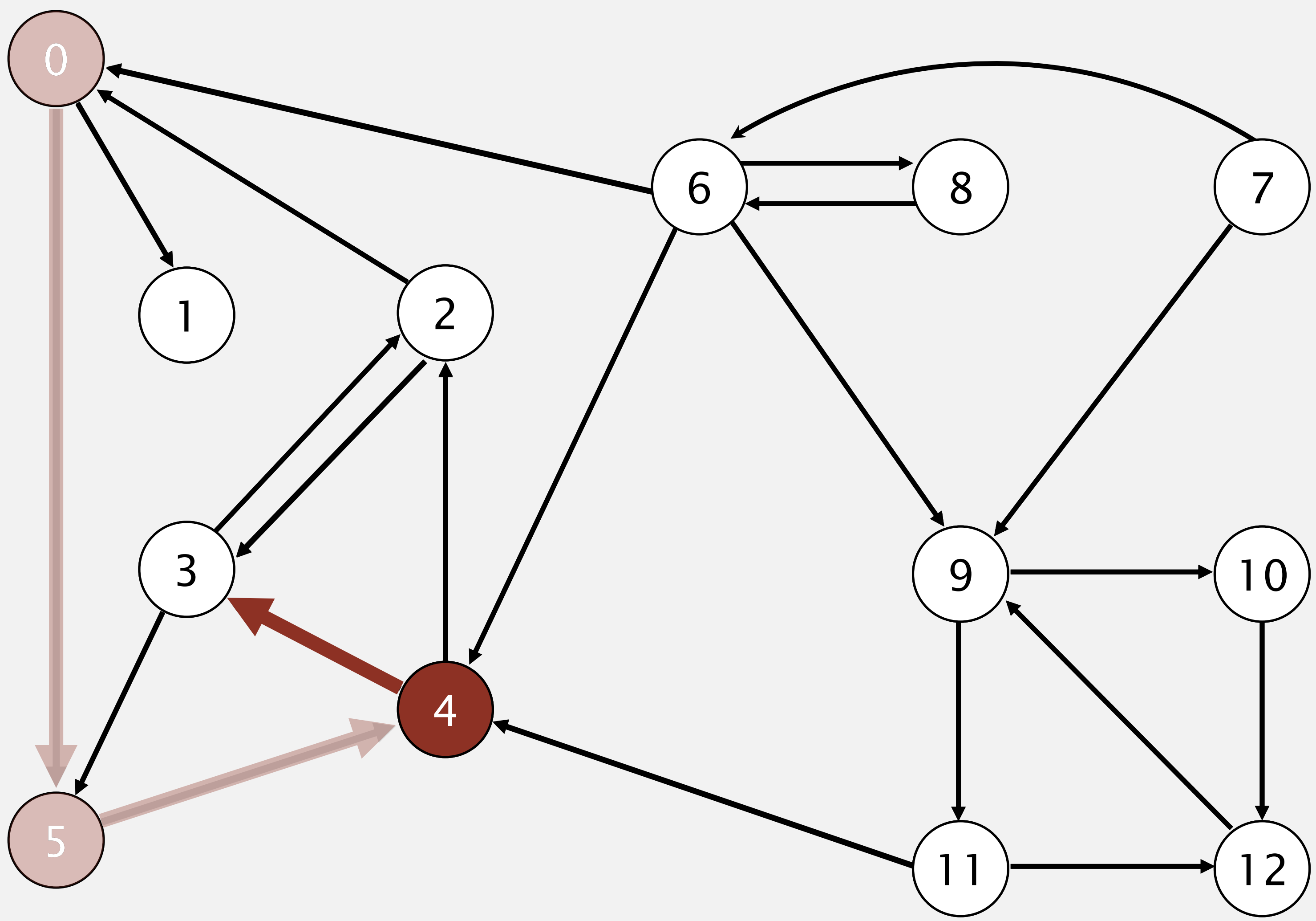
visit 5: check 4

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



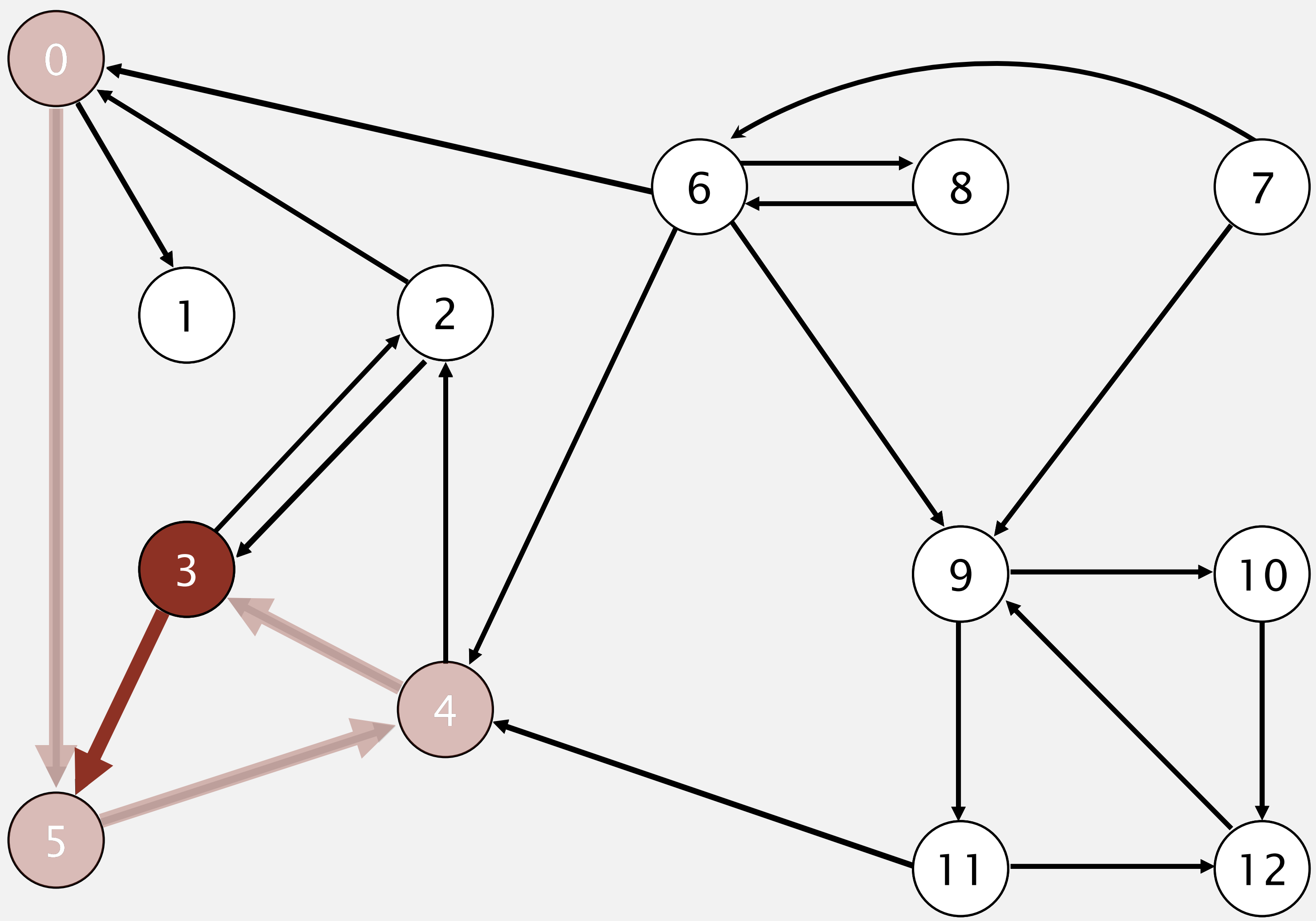
visit 4: check 3 and check 2

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



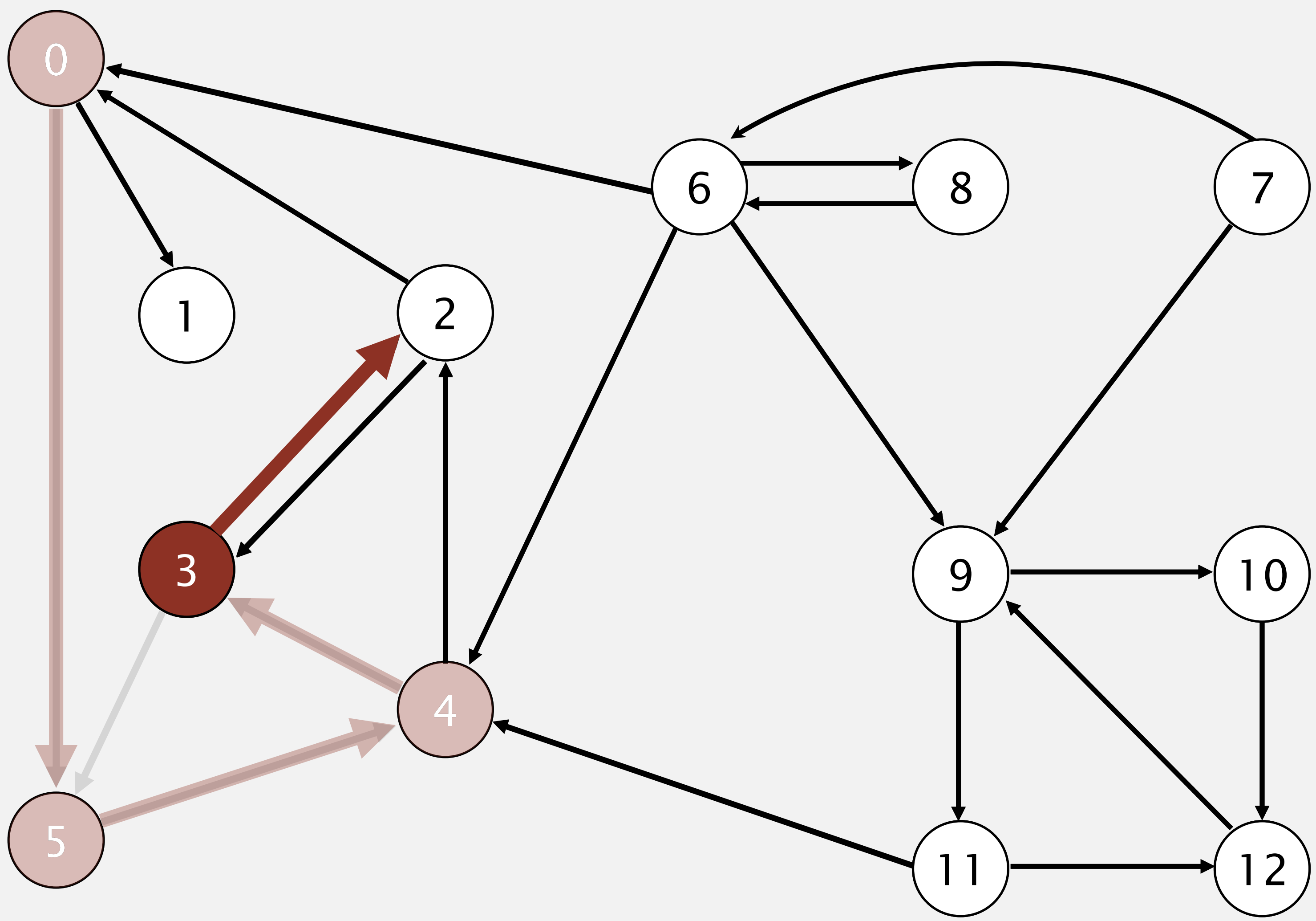
visit 3: check 5 **and check 2**

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



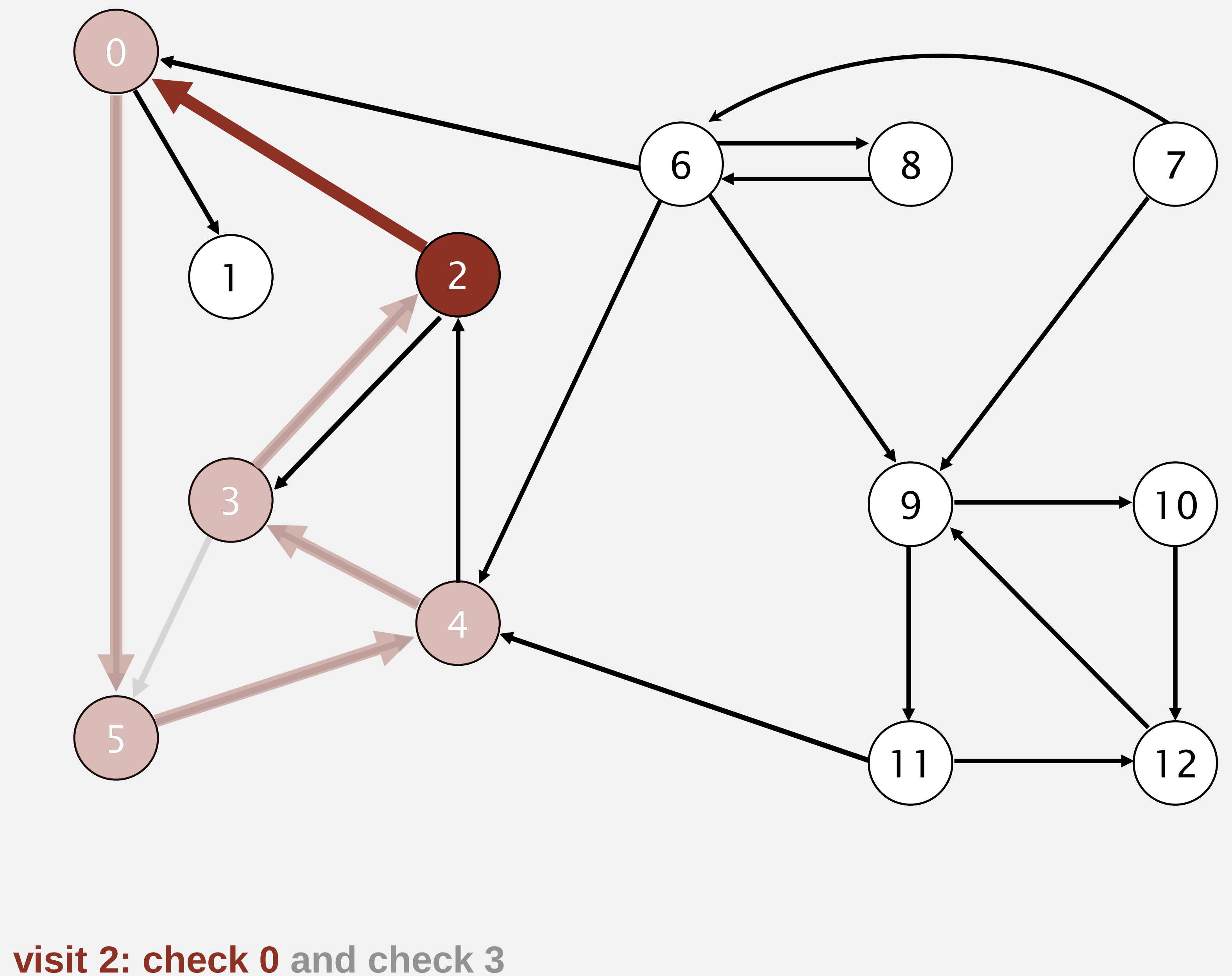
visit 3: check 5 and check 2

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.

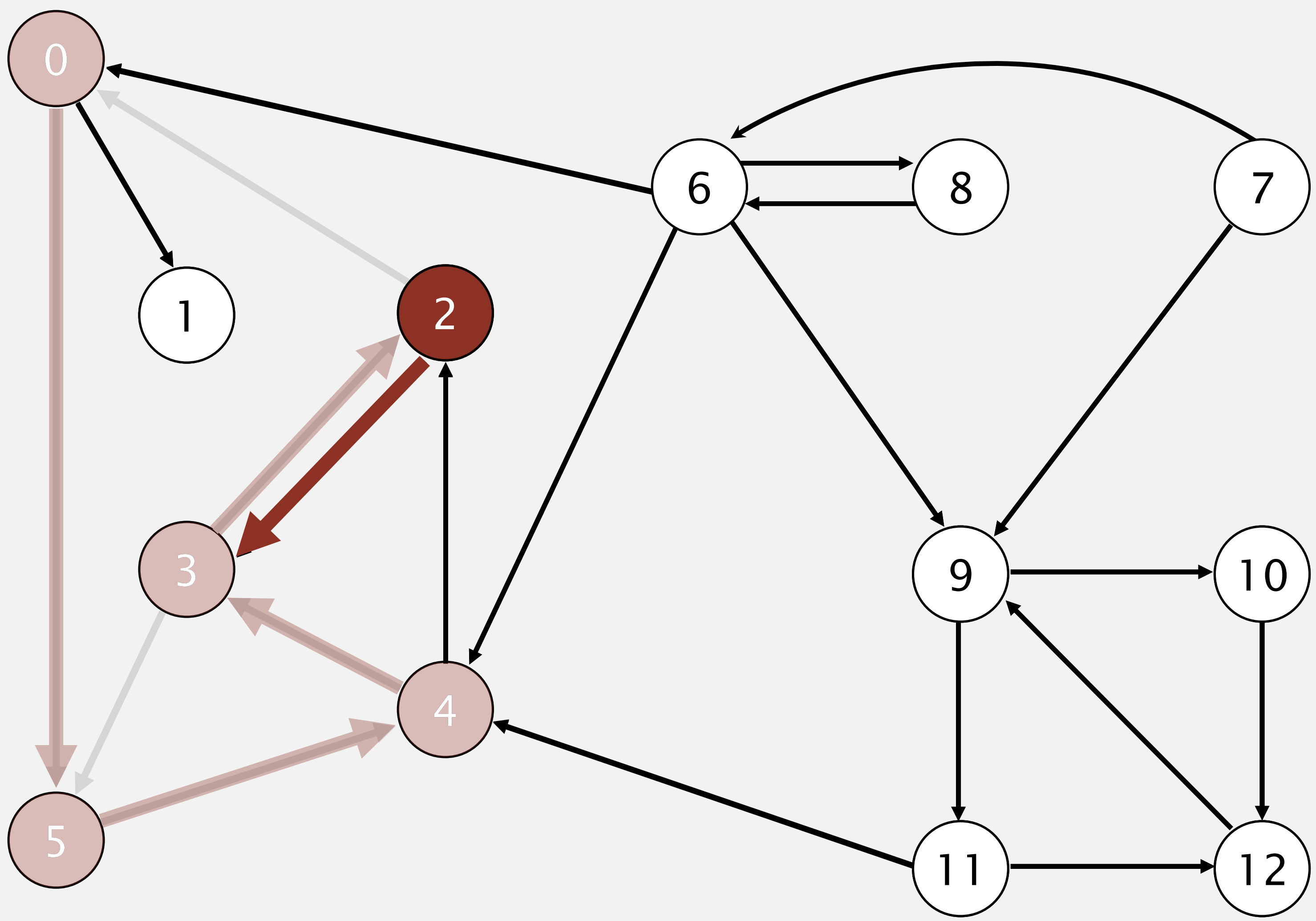


v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



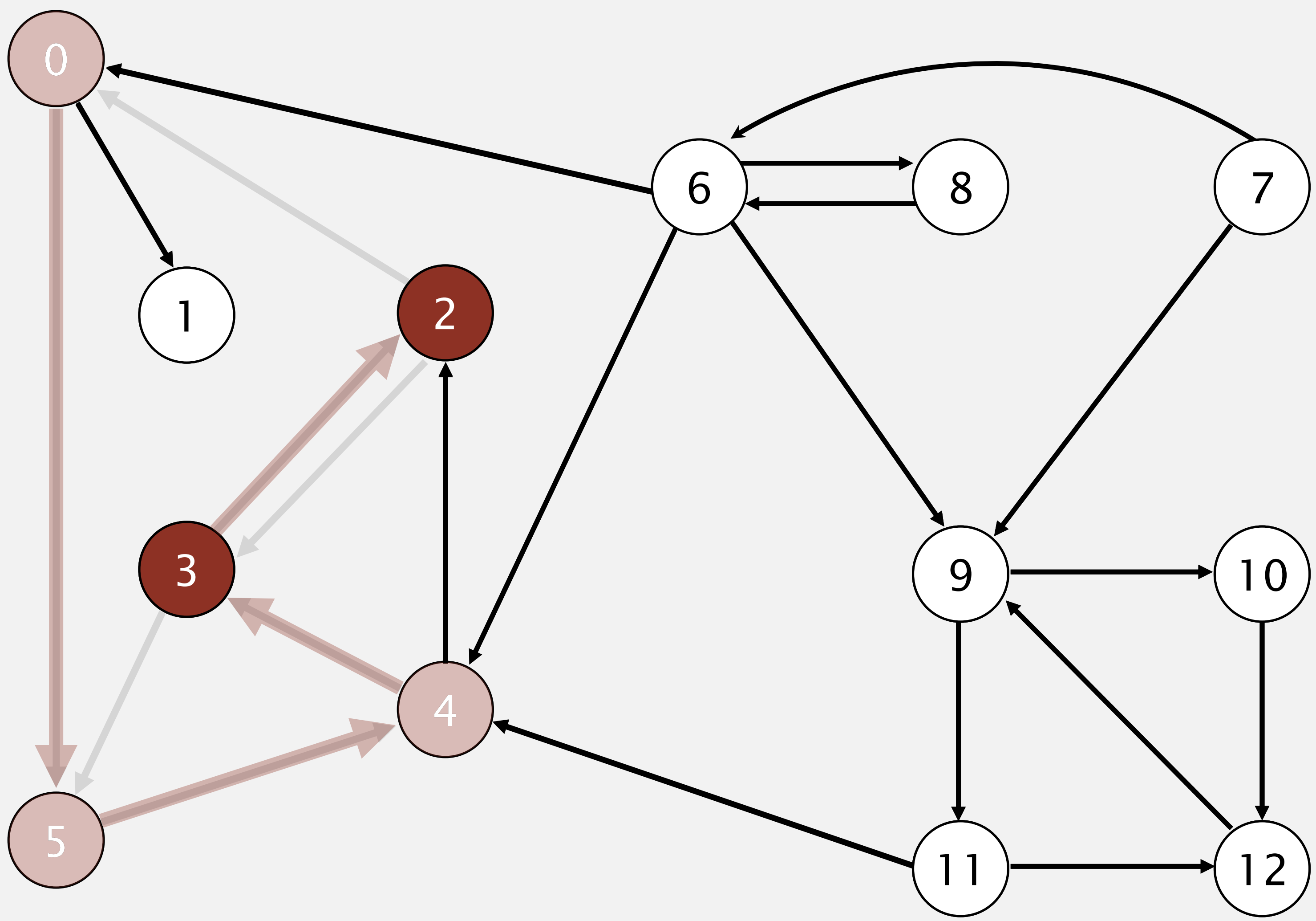
visit 2: check 0 and **check 3**

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



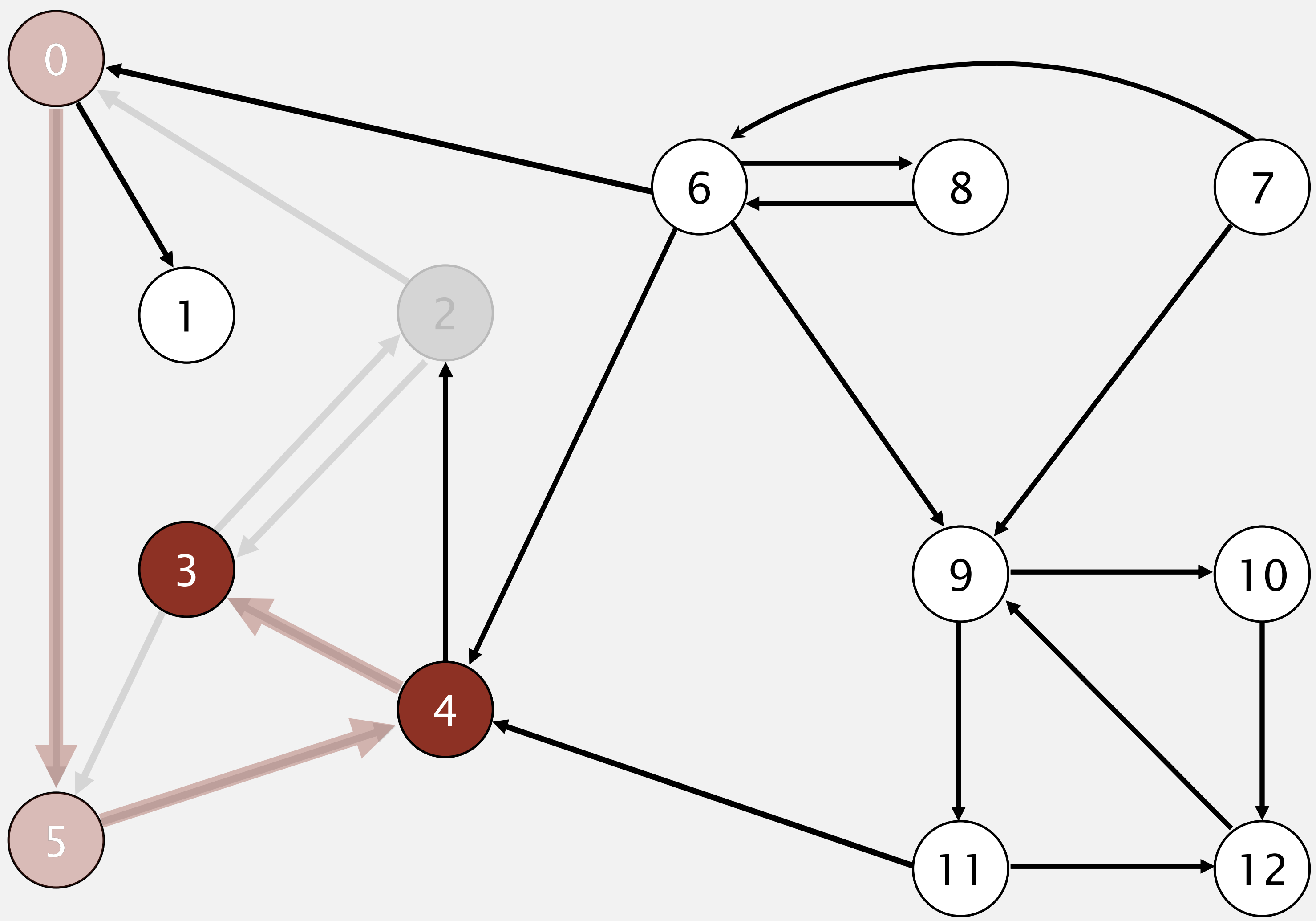
done 2

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



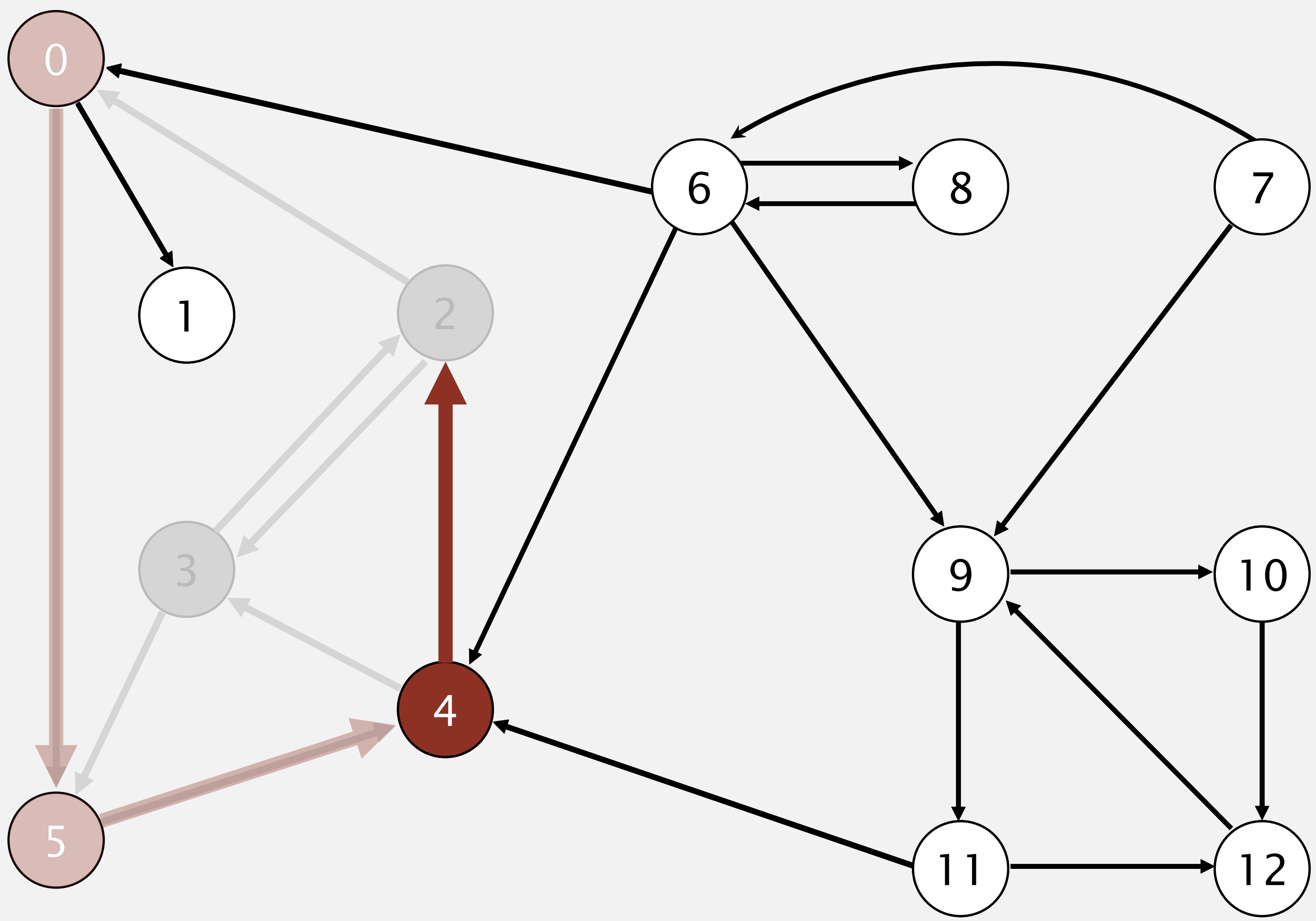
done 3

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



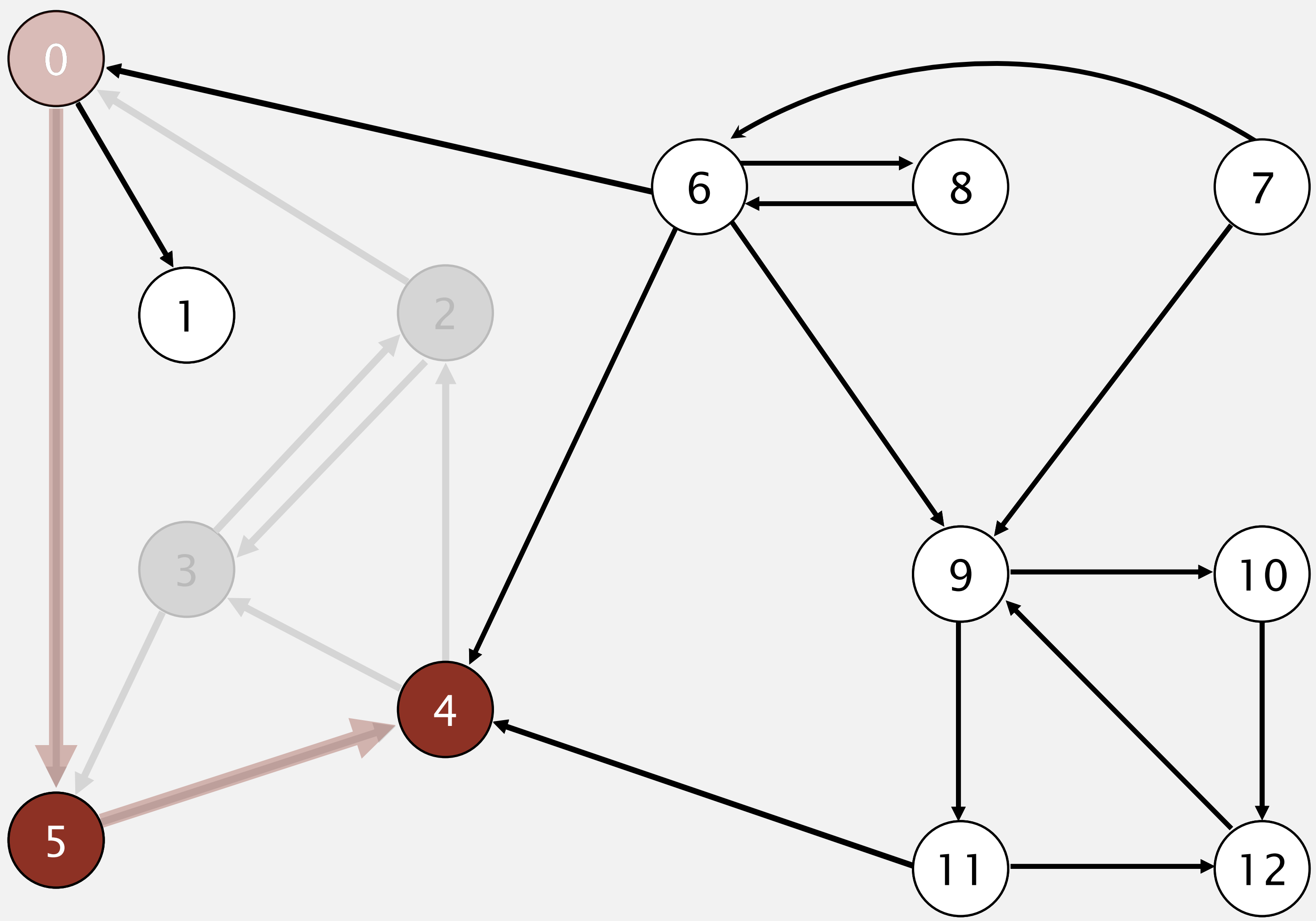
visit 4: check 3 and check 2

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



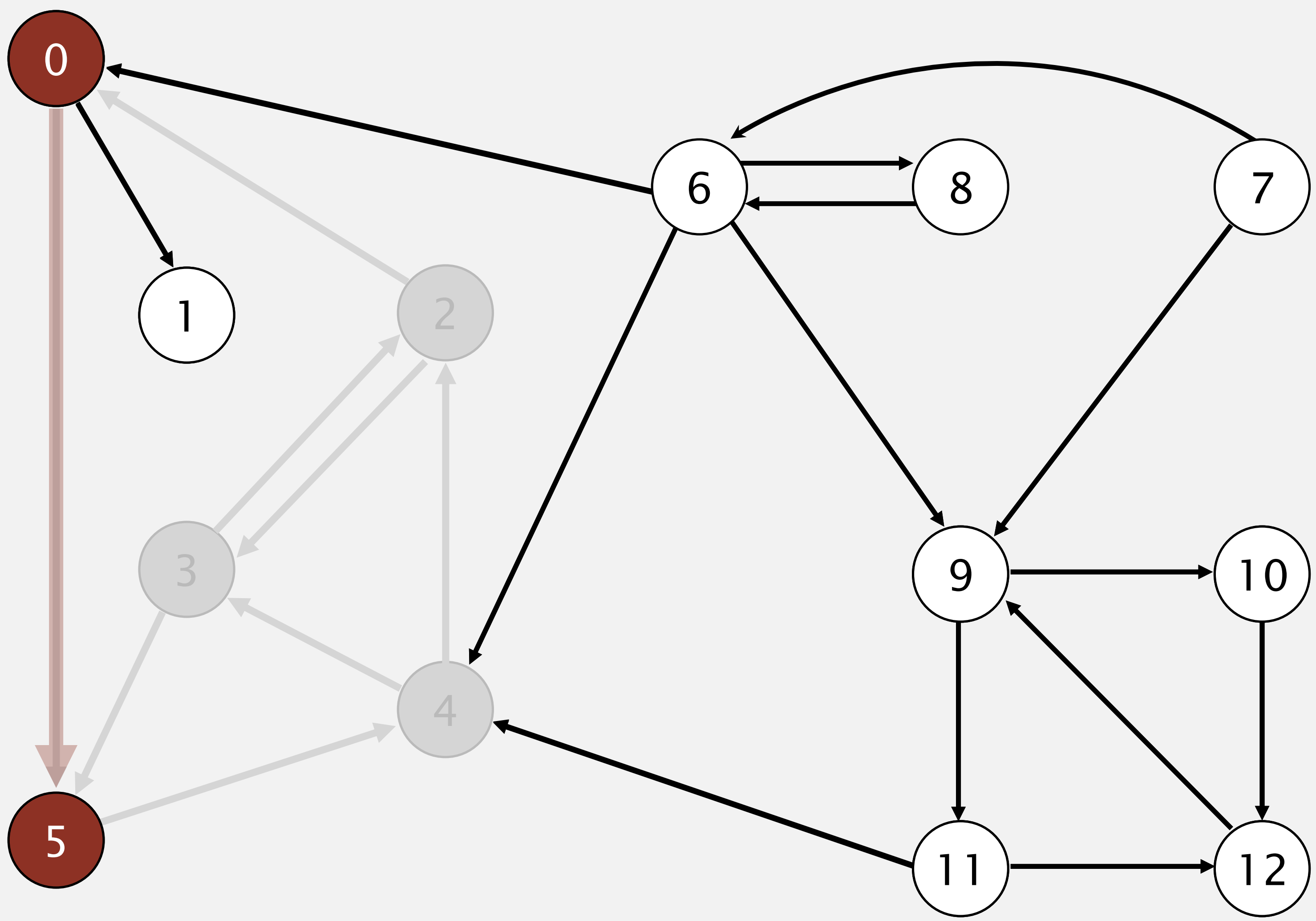
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 4

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



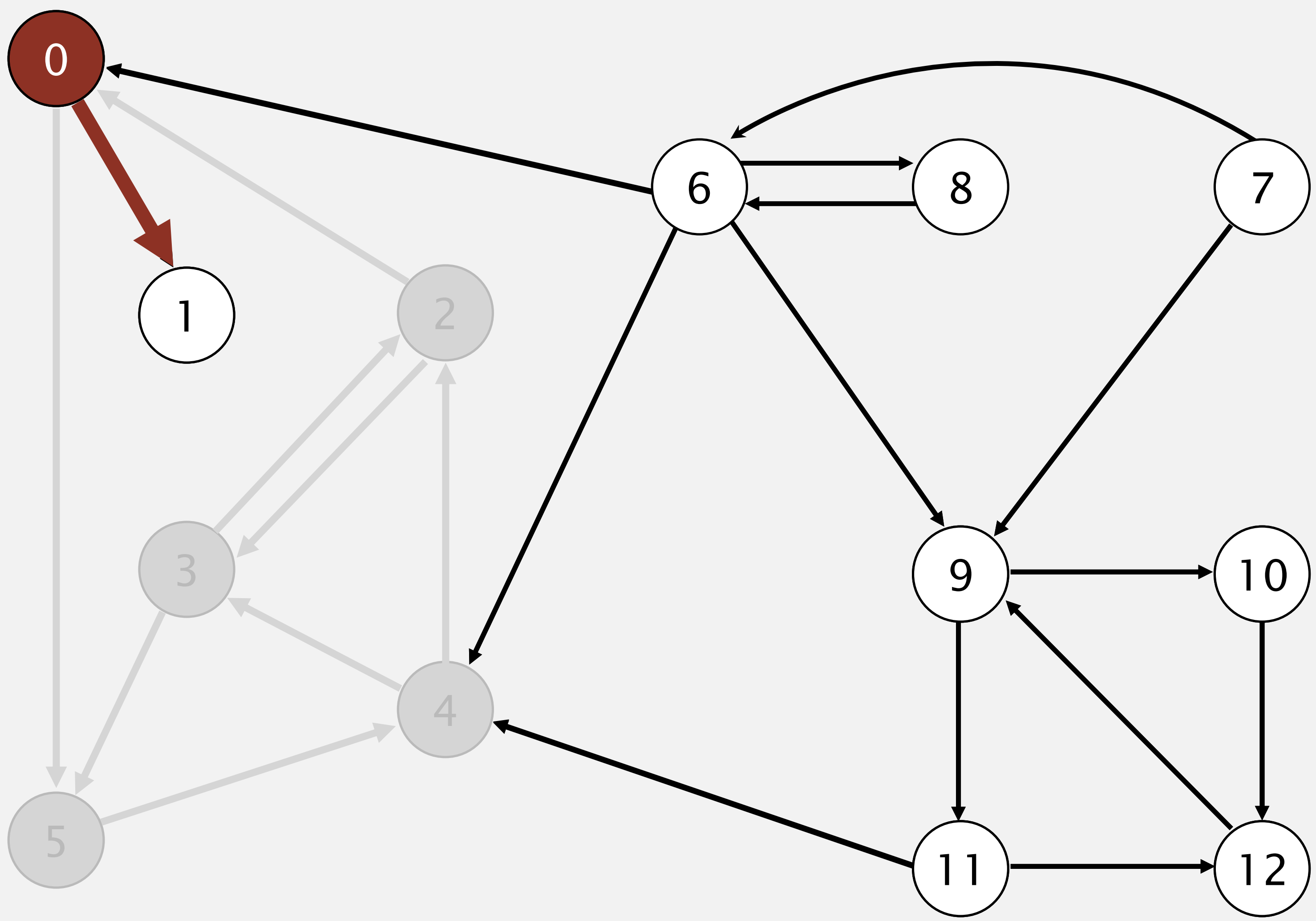
done 5

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



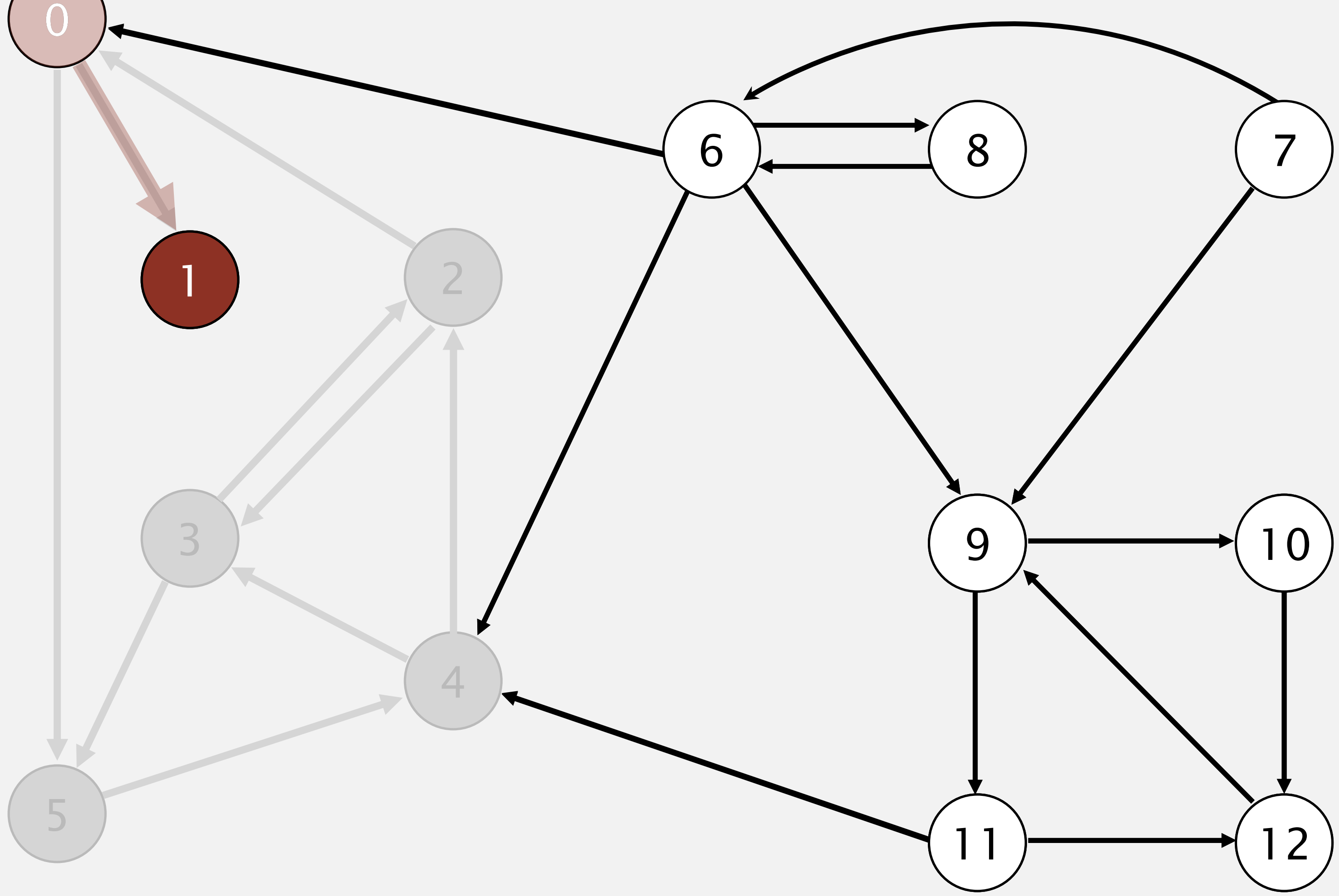
visit 0: check 5 and check 1

v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



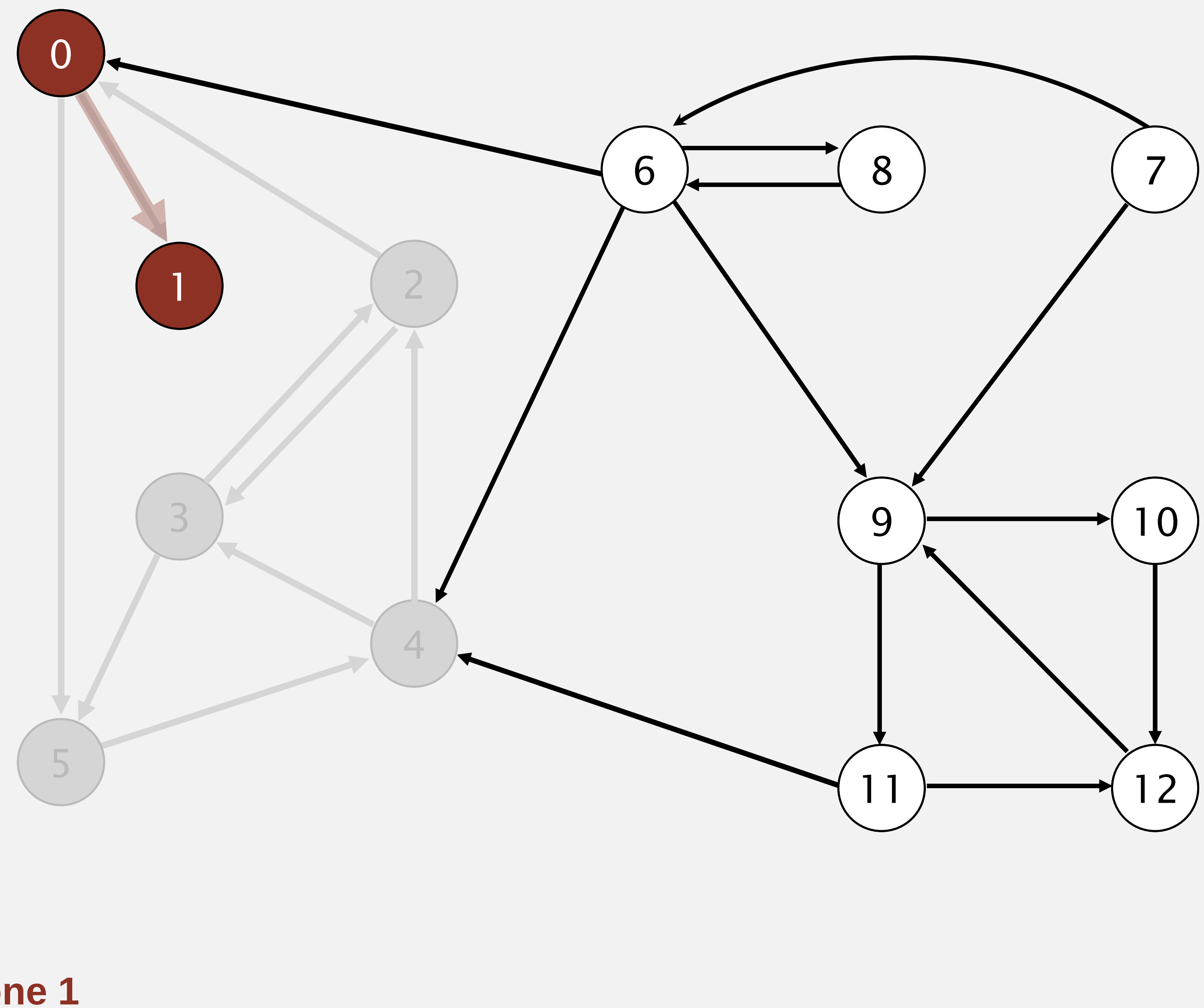
visit 1

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.

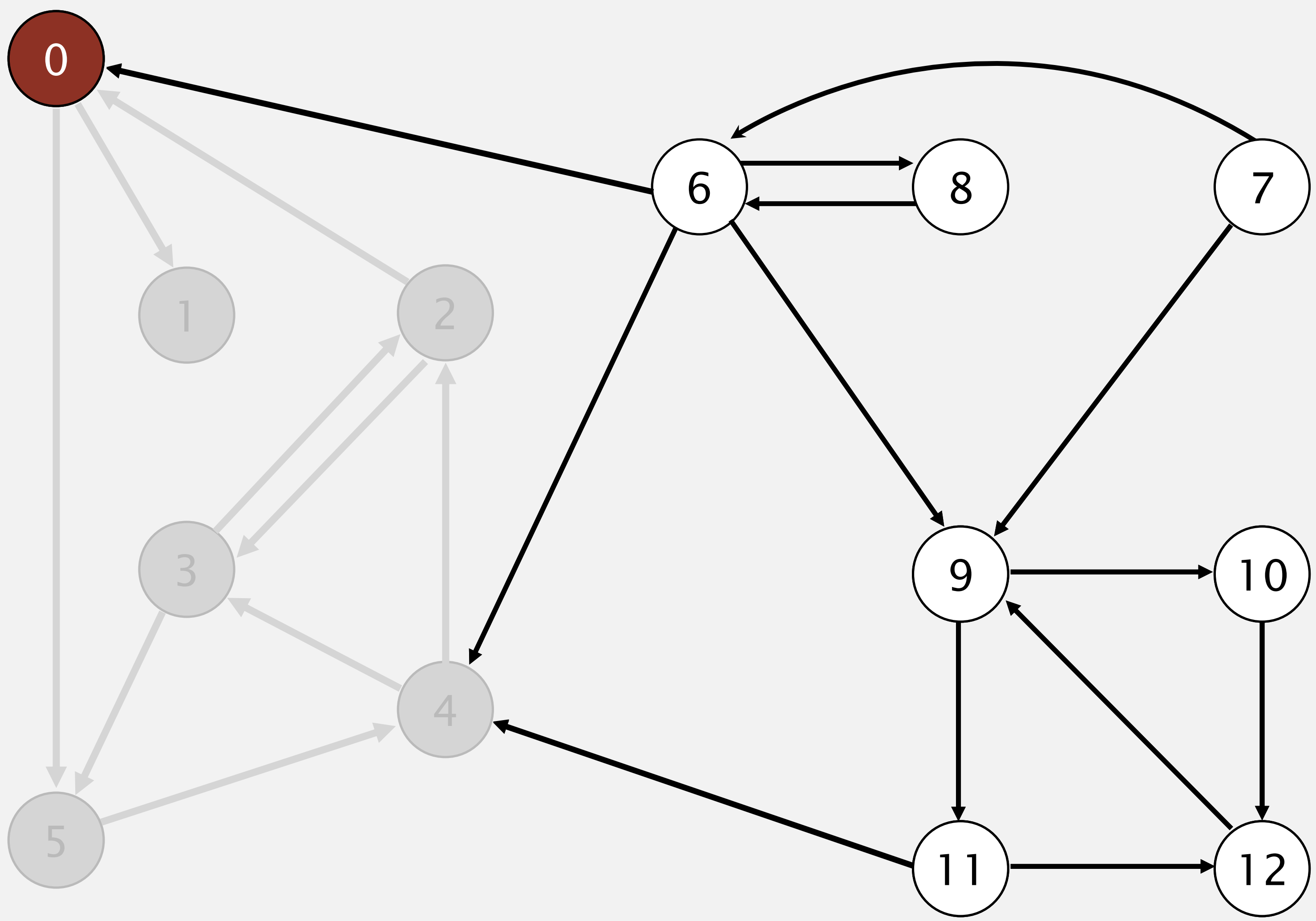


v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



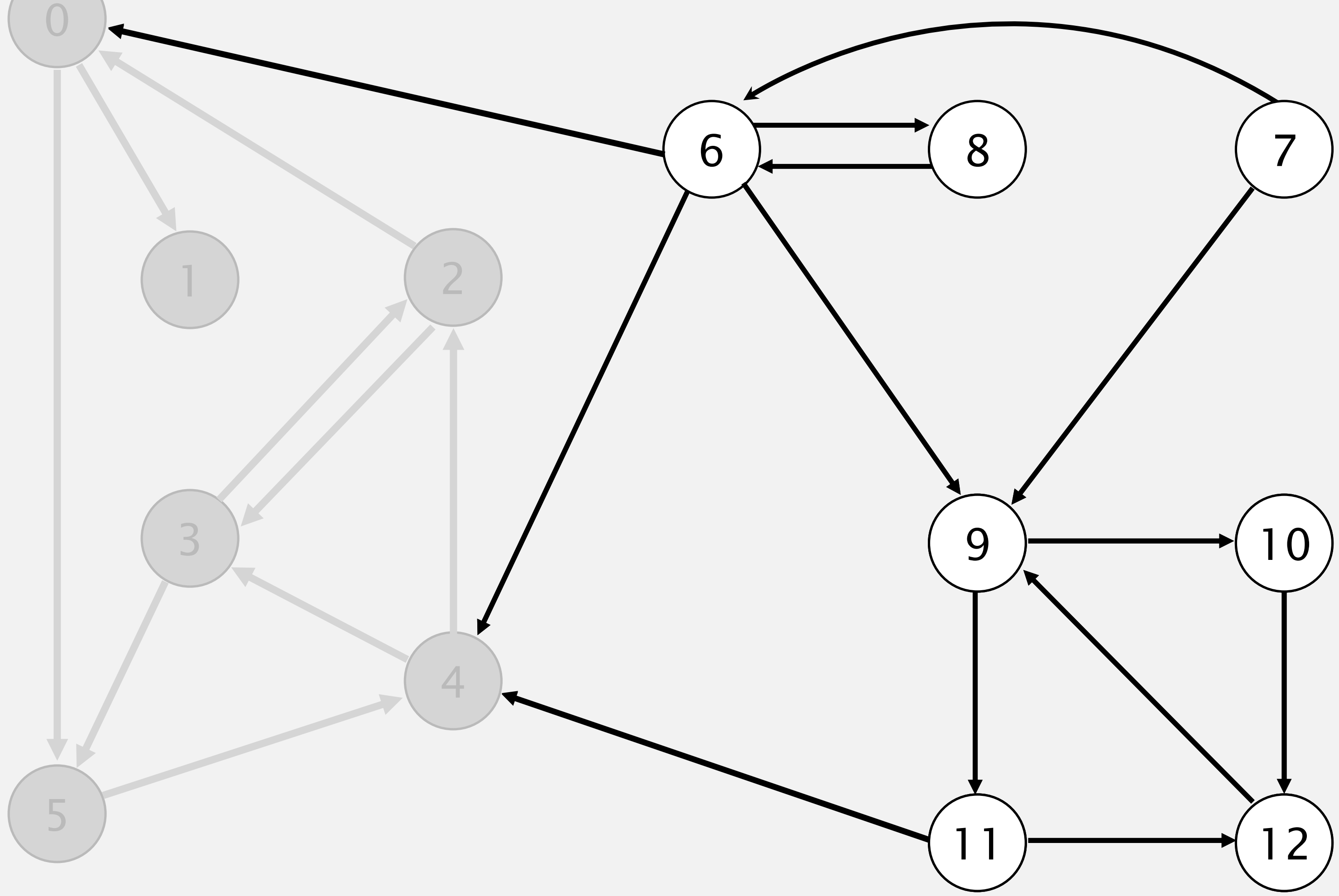
done 0

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



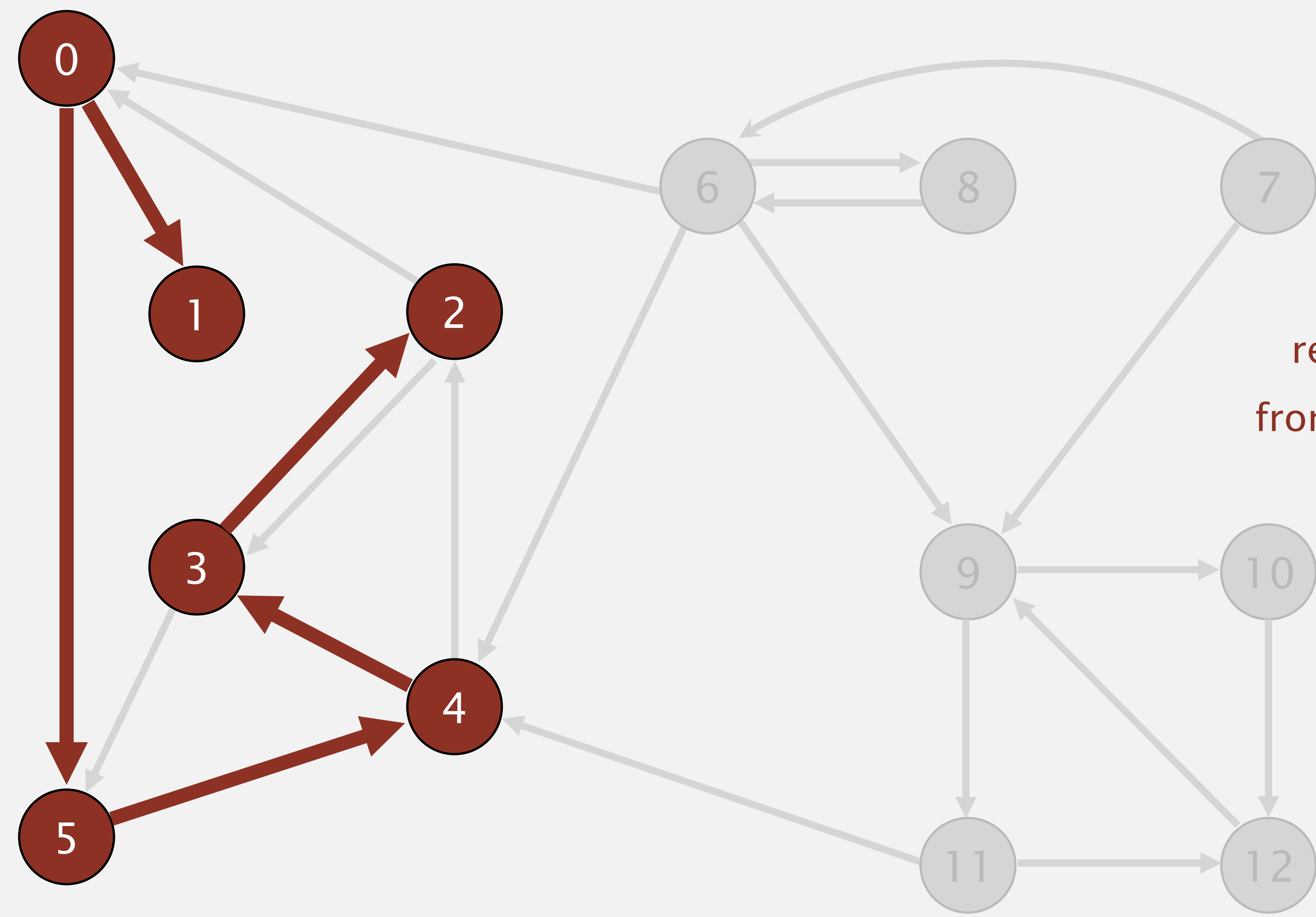
done

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.



reachable from 0

reachable from vertex 0

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search (in undirected graphs)

Recall code for undirected graphs.

```
public class DepthFirstSearch
{
   private boolean[] marked;

   public DepthFirstSearch(Graph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean visited(int v)
   {  return marked[v];  }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.

[substitute `Digraph` for `Graph`]

```
public class DirectedDFS
{
   private boolean[] marked;

   public DirectedDFS(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }
   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean visited(int v)
   {  return marked[v];  }
}
```

true if path from s

constructor marks vertices reachable from s

recursive DFS does the work

client can ask whether any vertex is reachable from s

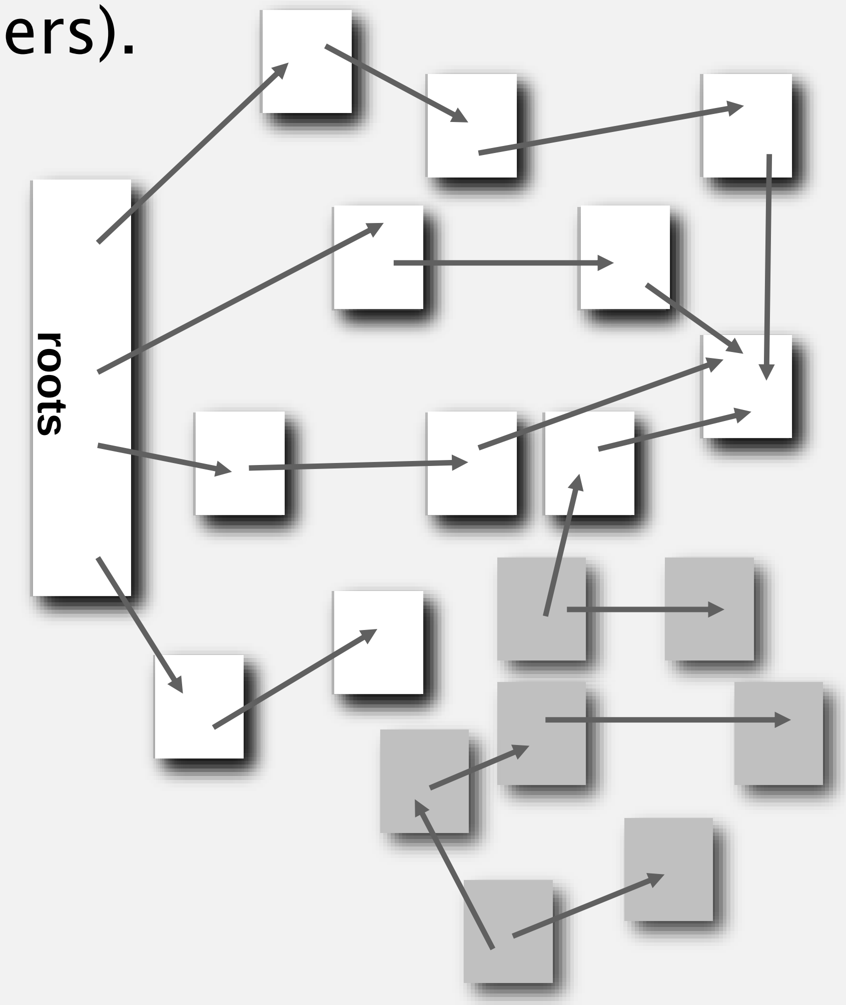
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

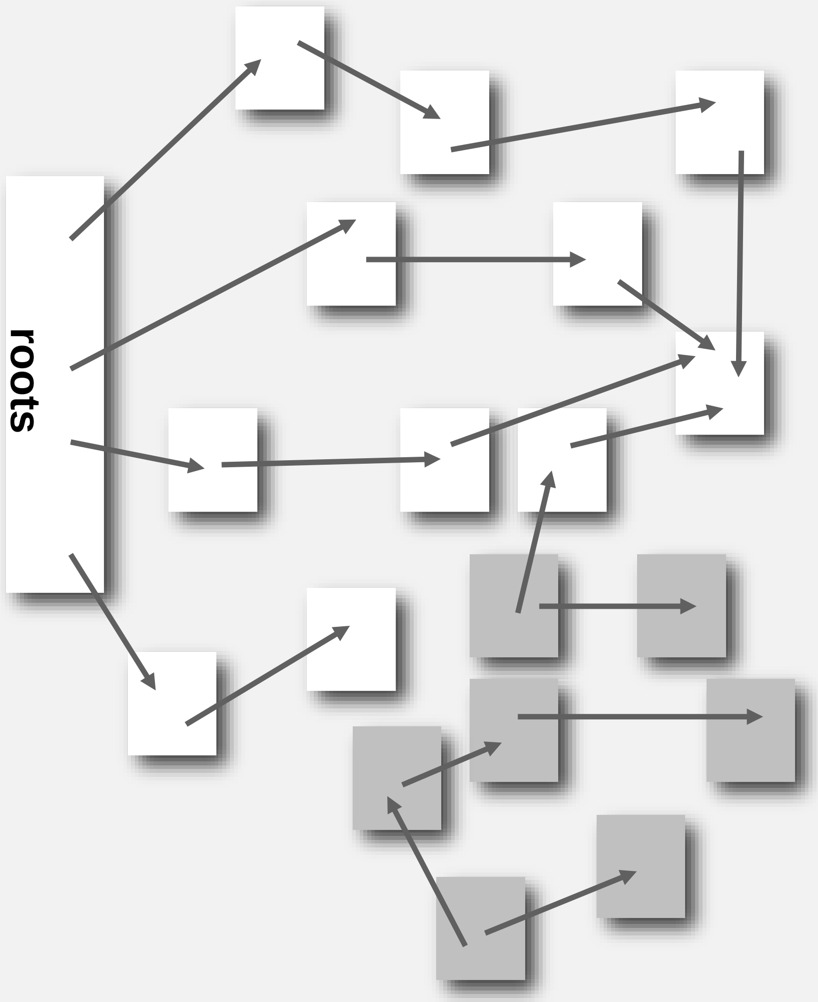


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

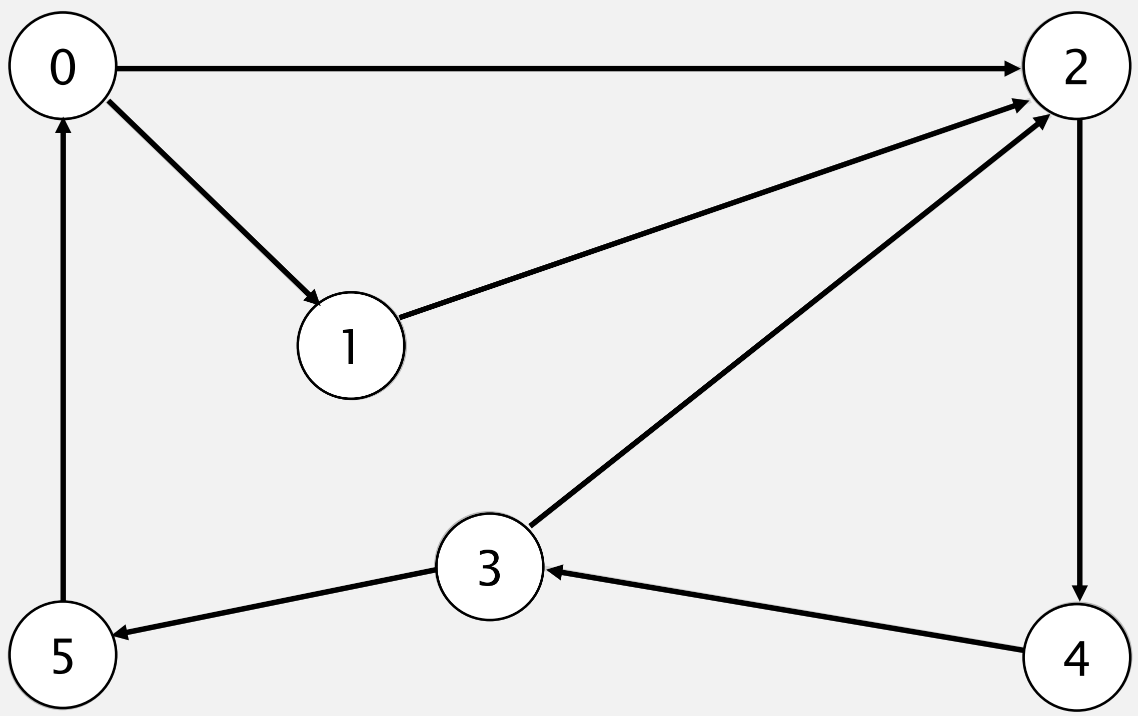
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



graph G

tinyDG2.txt

V

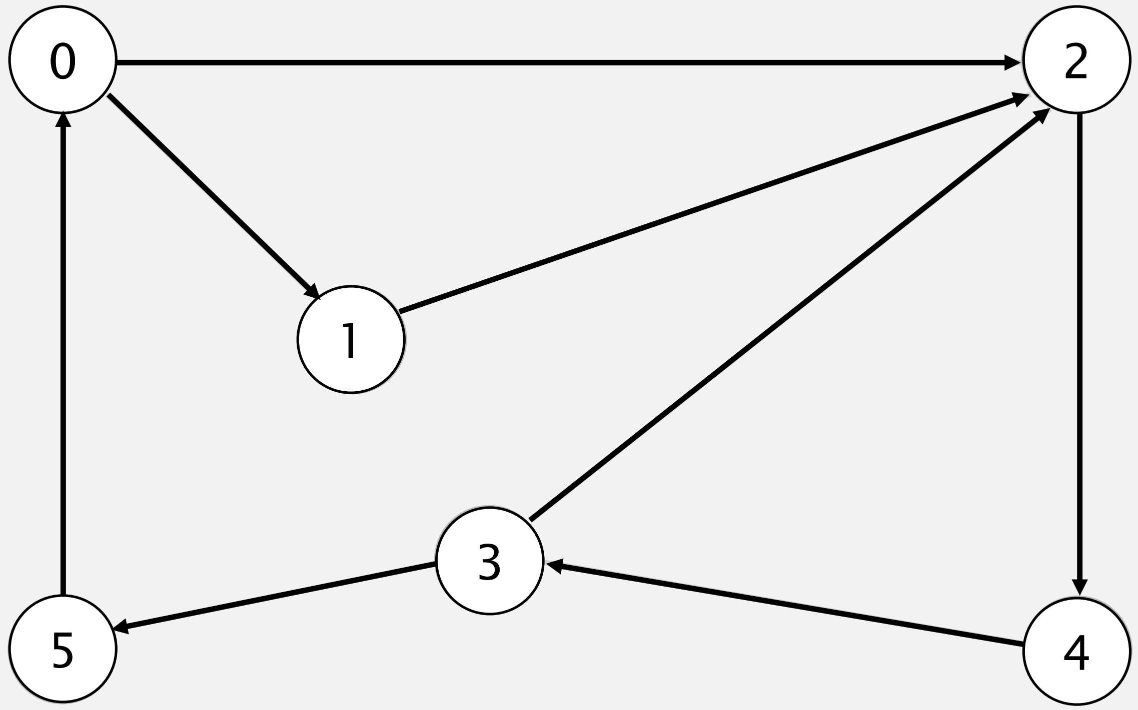
E

```
6
8
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2
```

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



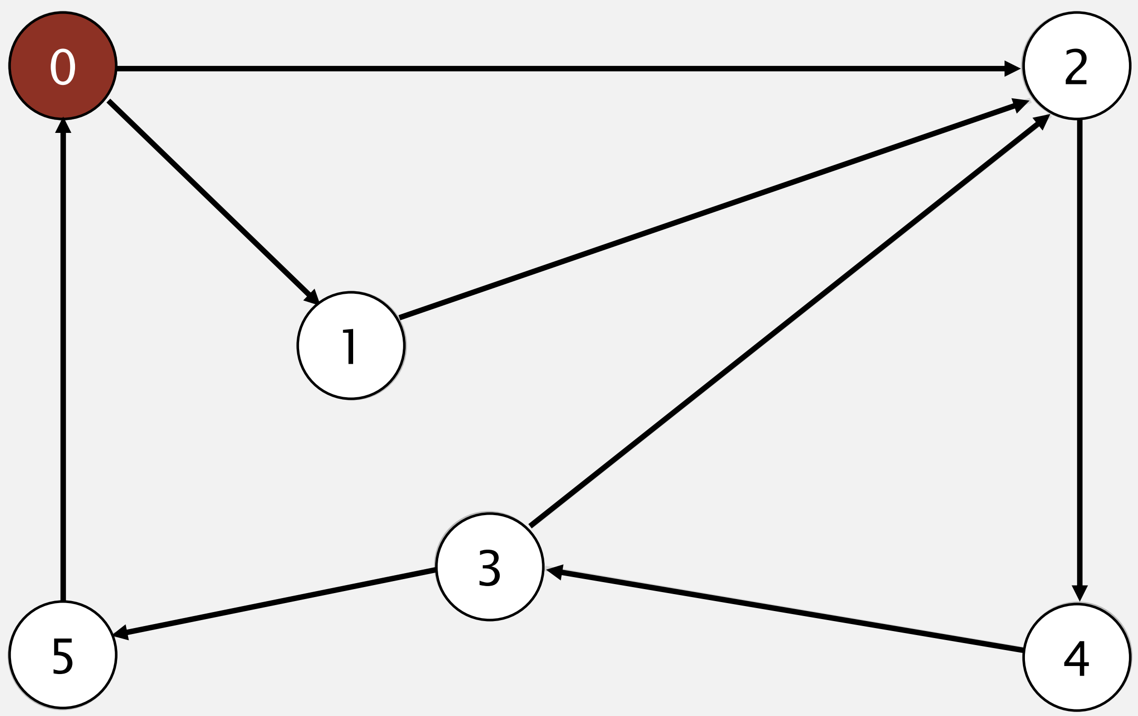
queue	v	edgeTo[]	distTo[]
	0	–	0
	1	–	–
	2	–	–
	3	–	–
	4	–	–
	5	–	–

add 0 to queue

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue
0

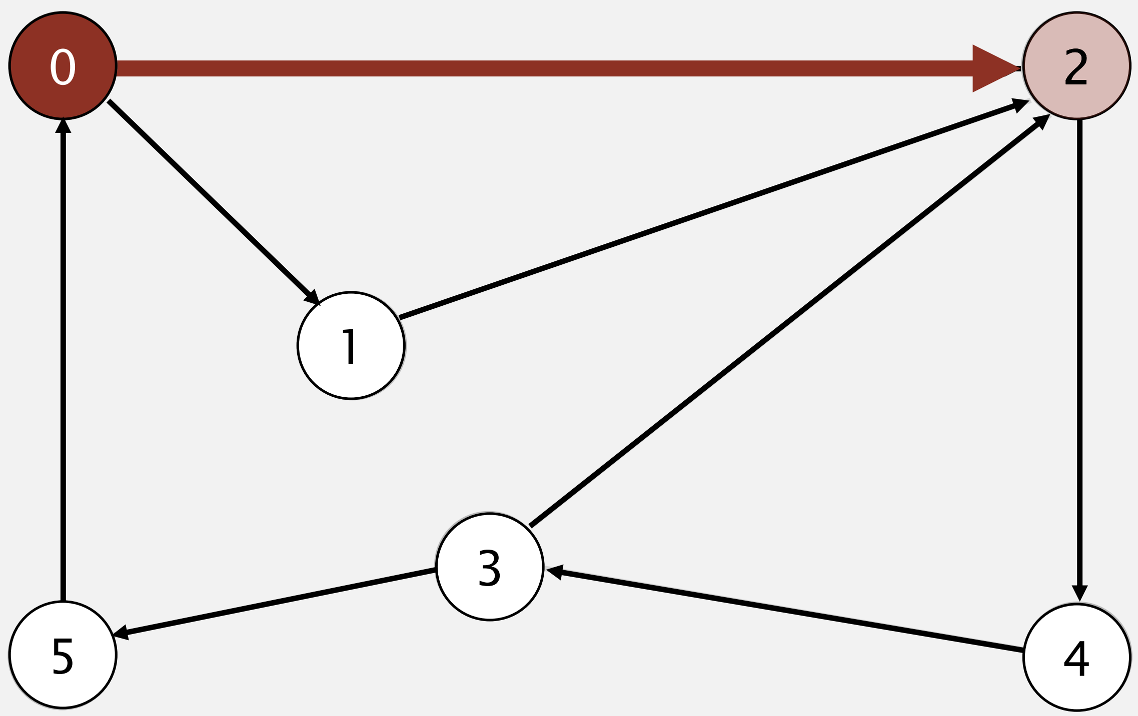
v	edgeTo[]	distTo[]
0	–	0
1	–	–
2	–	–
3	–	–
4	–	–
5	–	–

dequeue 0

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue

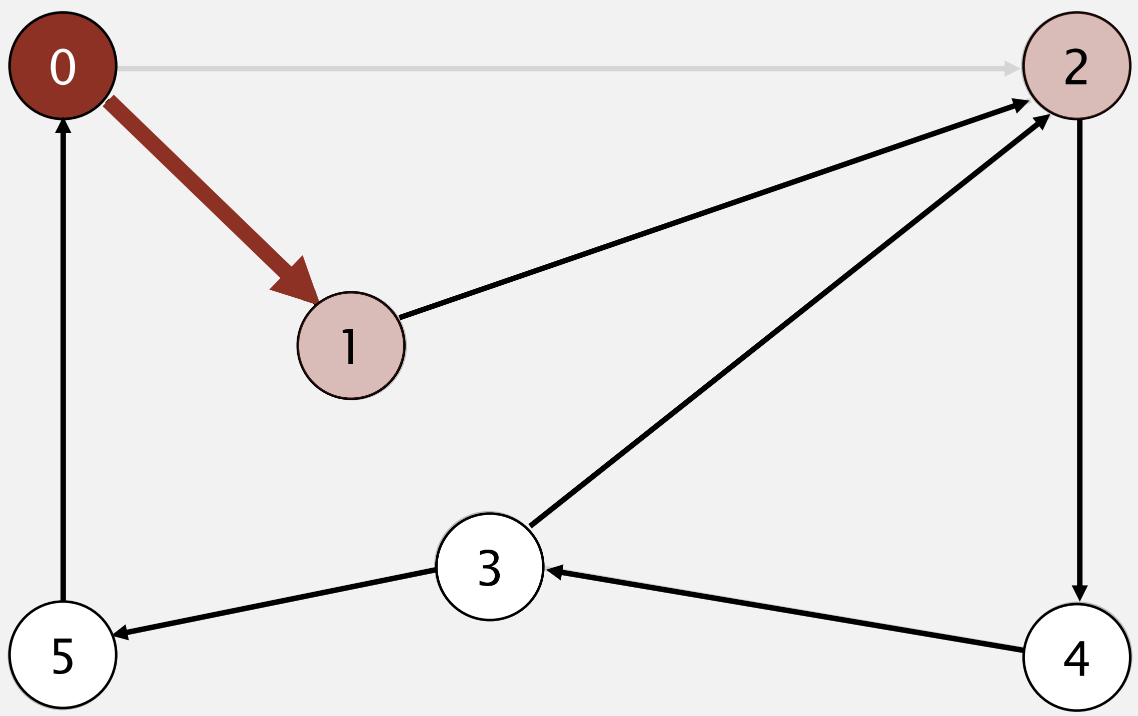
v	edgeTo[]	distTo[]
0	–	0
1		
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0: check 2 and check 1

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue

2

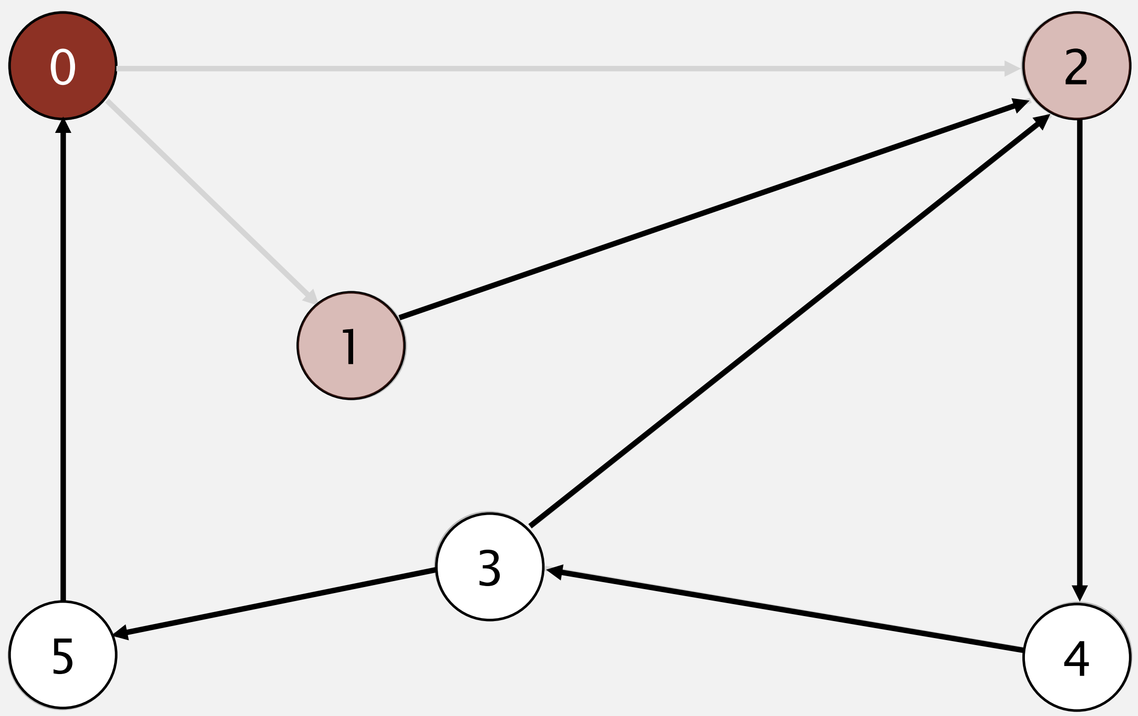
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0: check 2 and **check 1**

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue
1
2

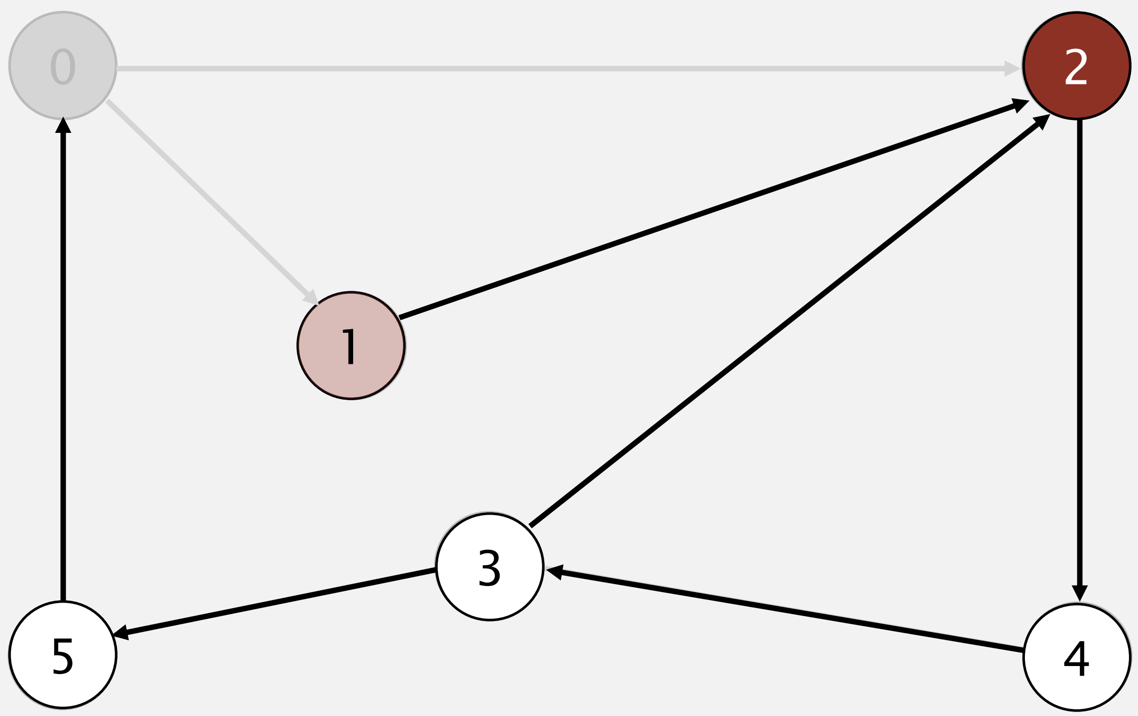
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

0 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue
1
2

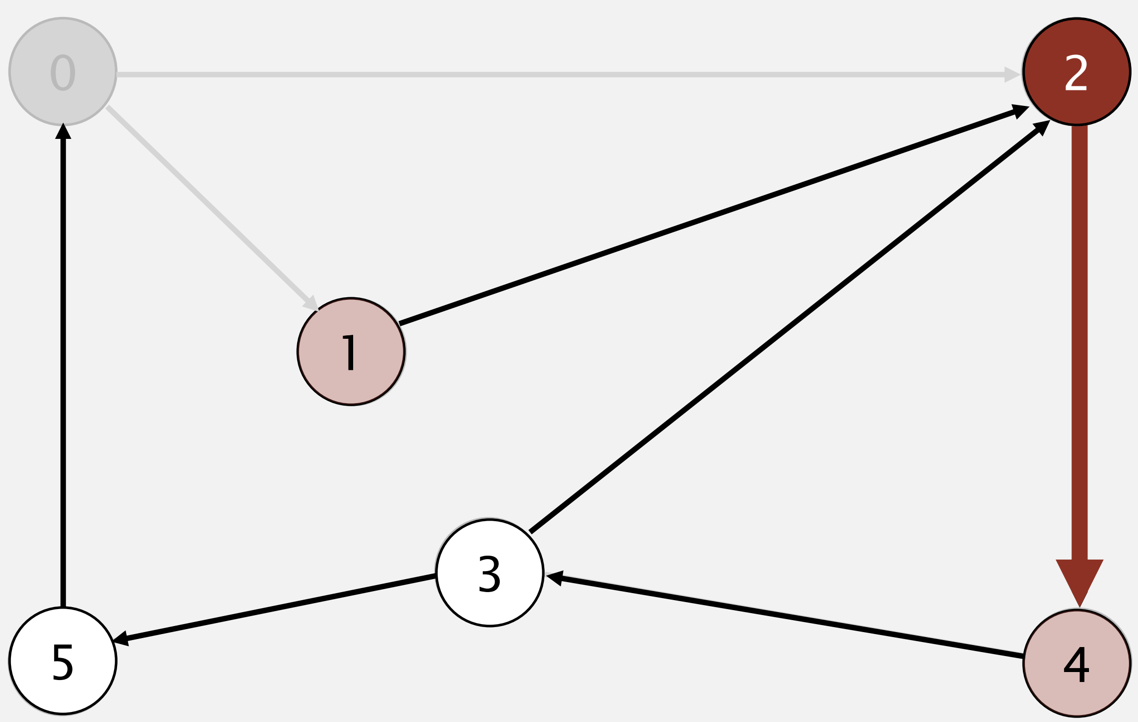
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 2

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue
1

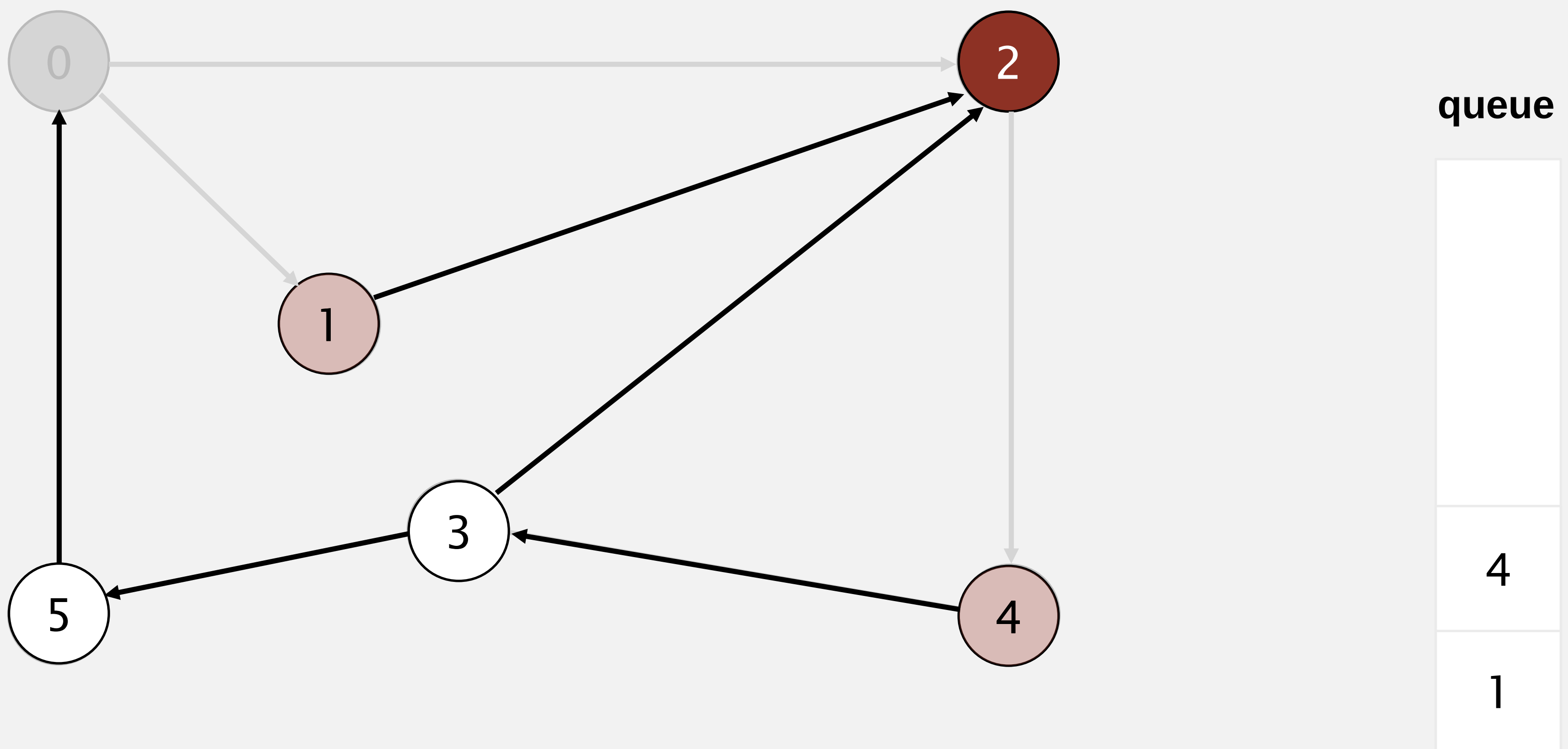
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	–	–
5	–	–

dequeue 2: check 4

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



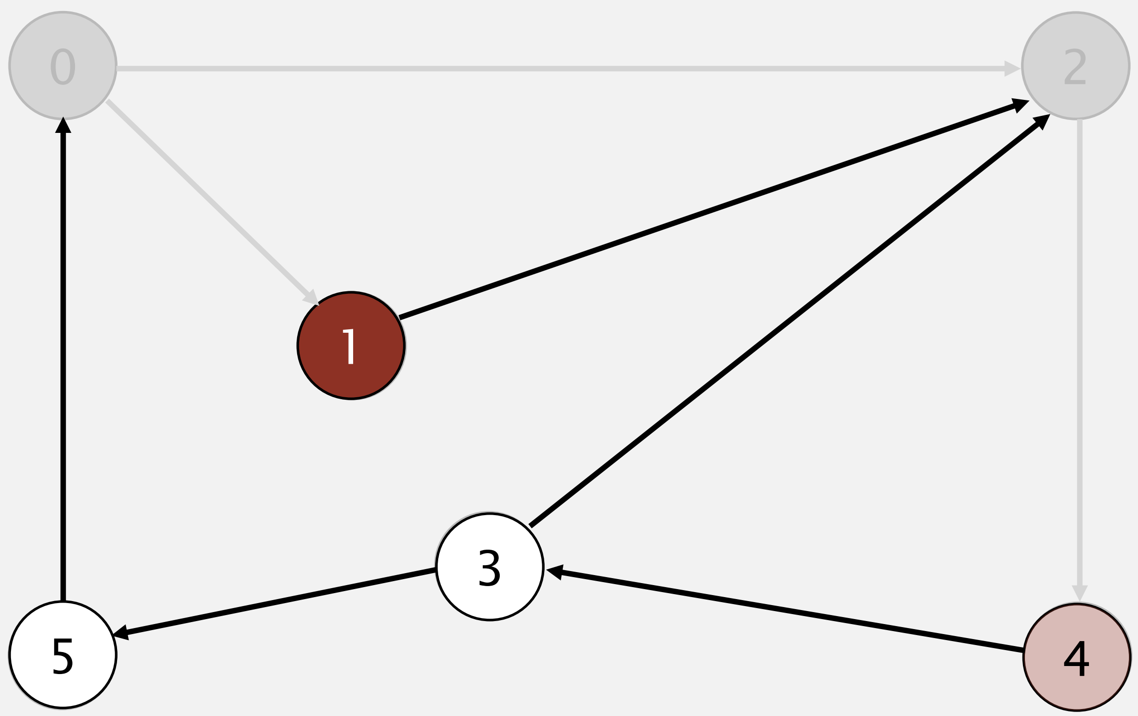
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

2 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue

4
1

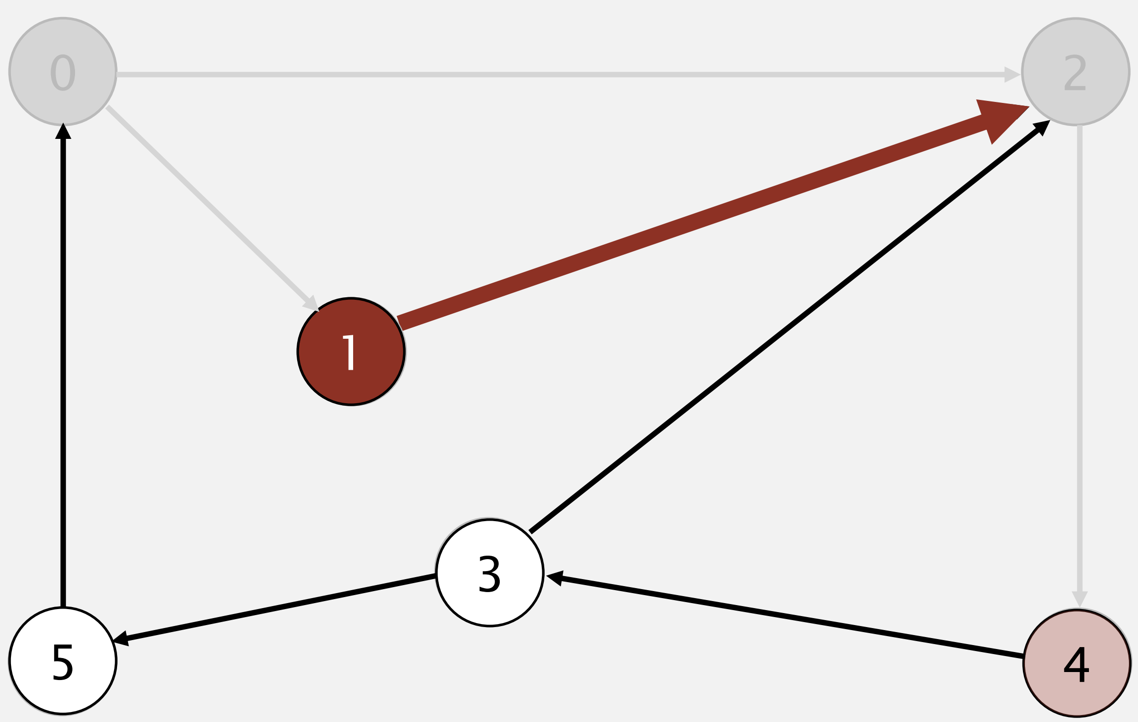
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 1

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue
4

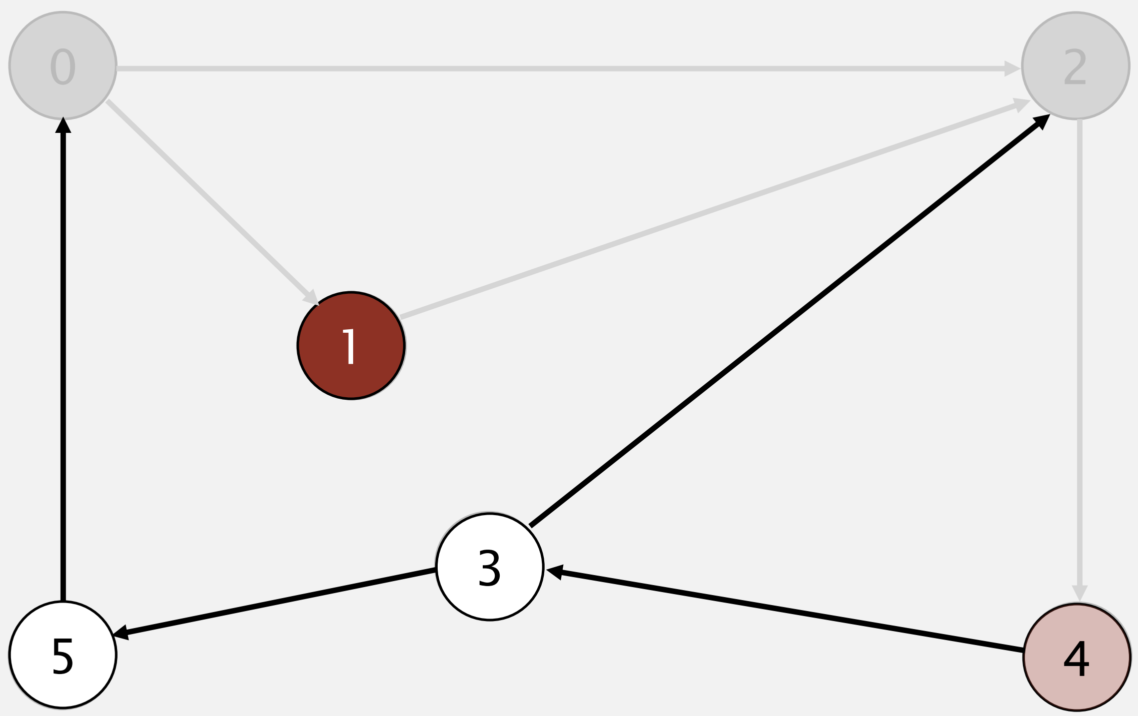
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 1; check 2

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue
4

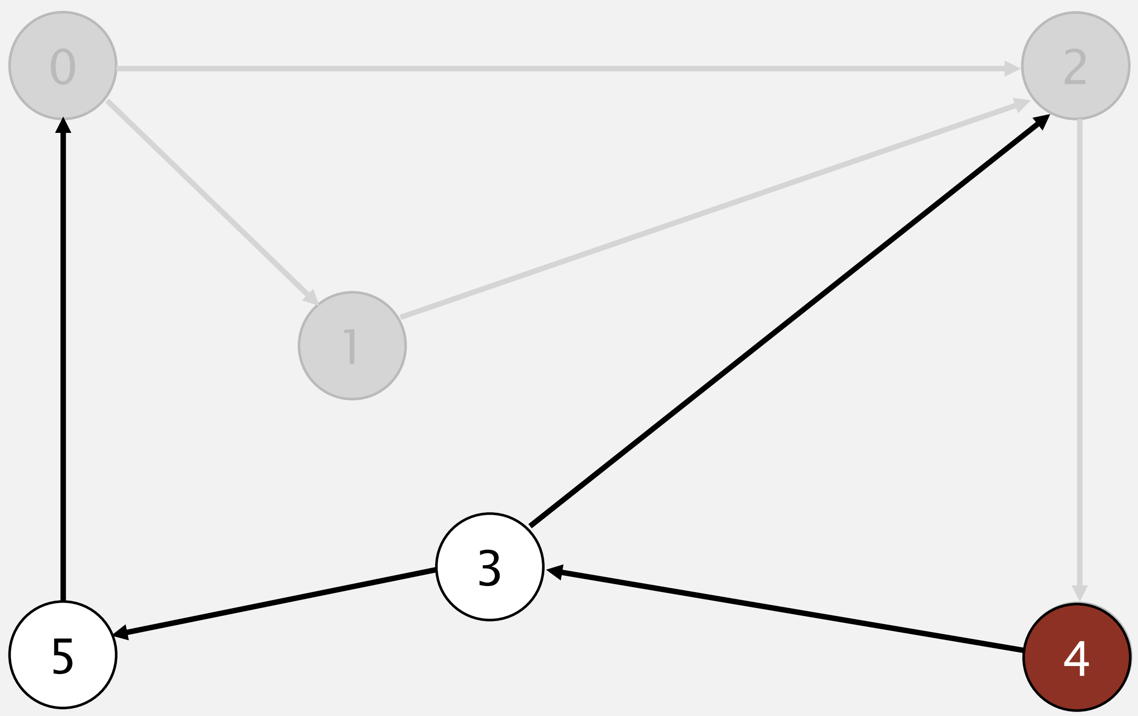
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

1 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue: 4

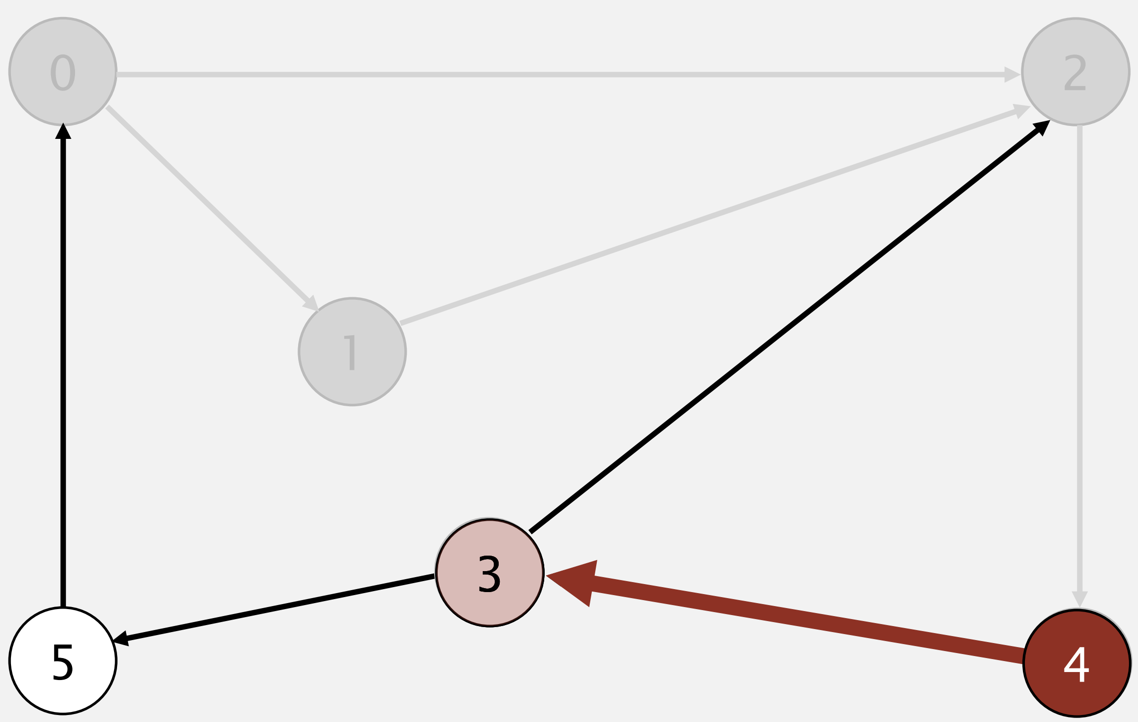
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 4

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue

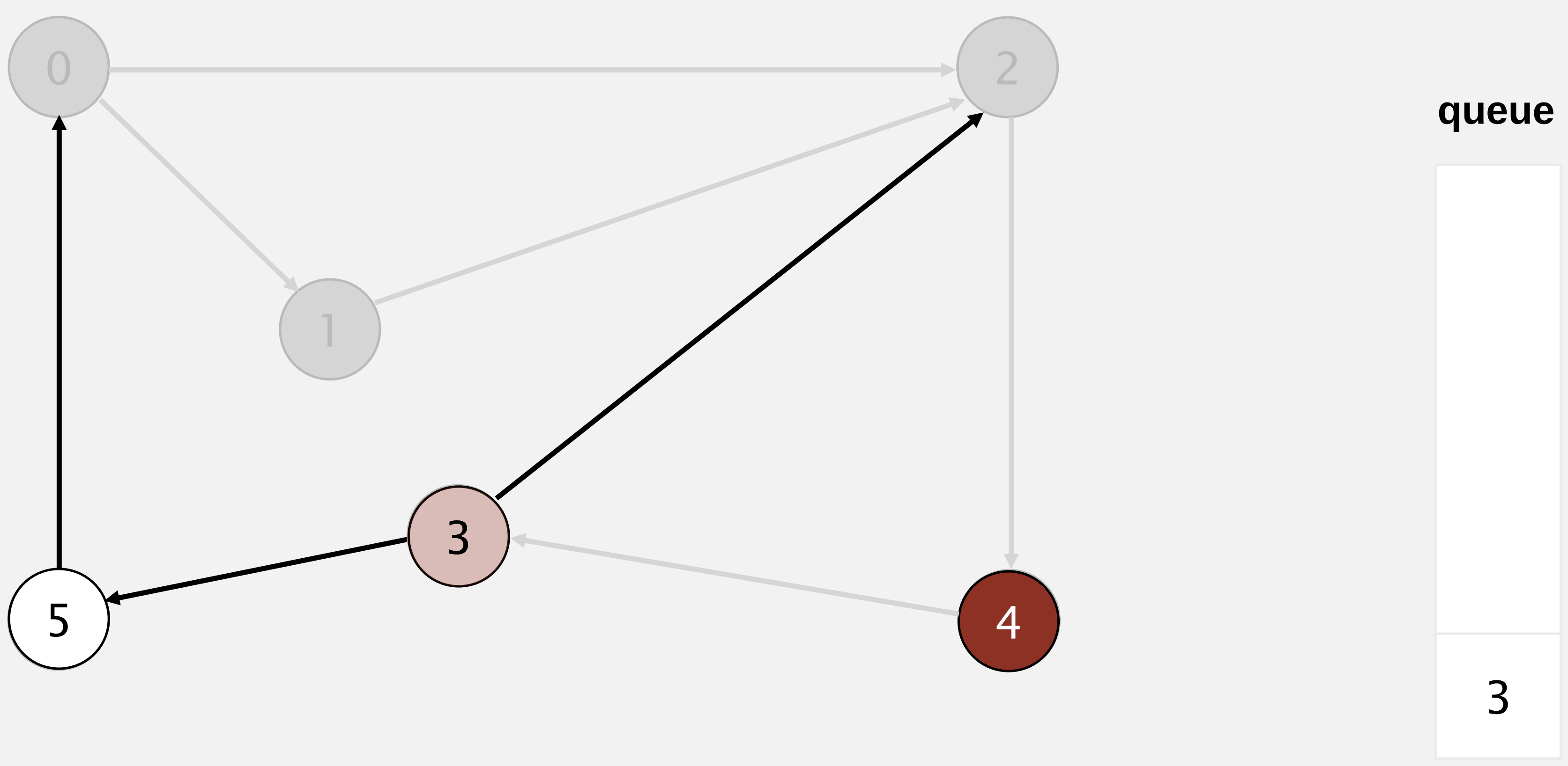
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2		
	4	3
3	–	–
4	2	2
5	–	–

dequeue 4: check 3

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



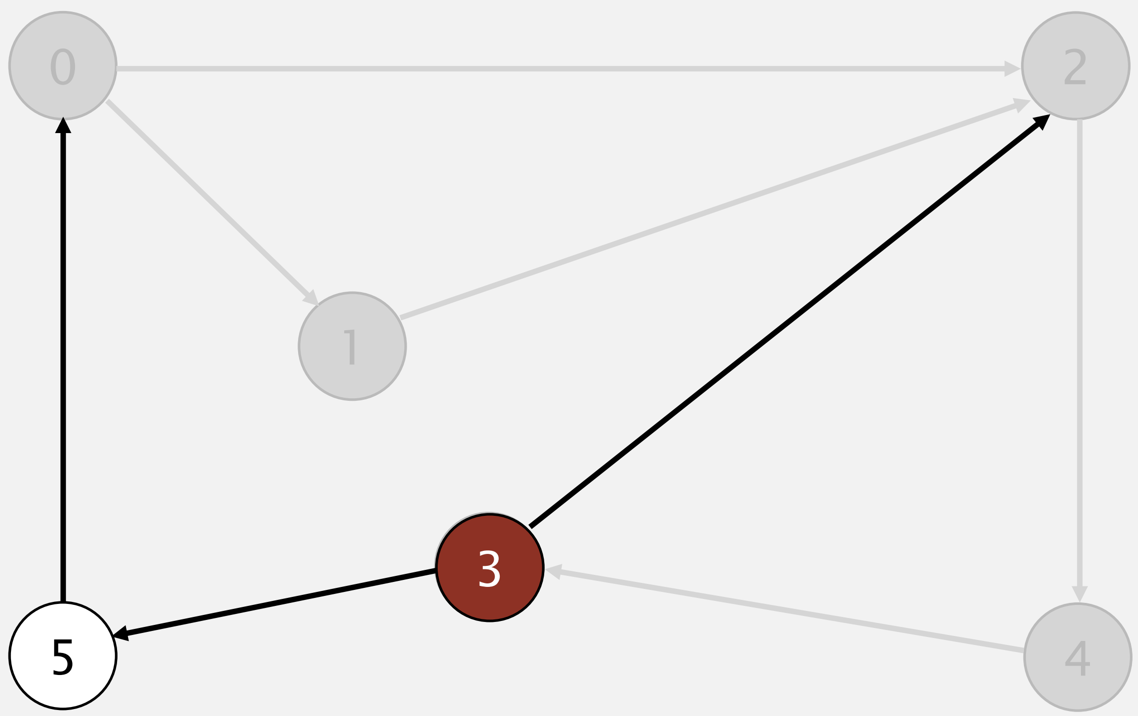
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	–	–

4 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue
3

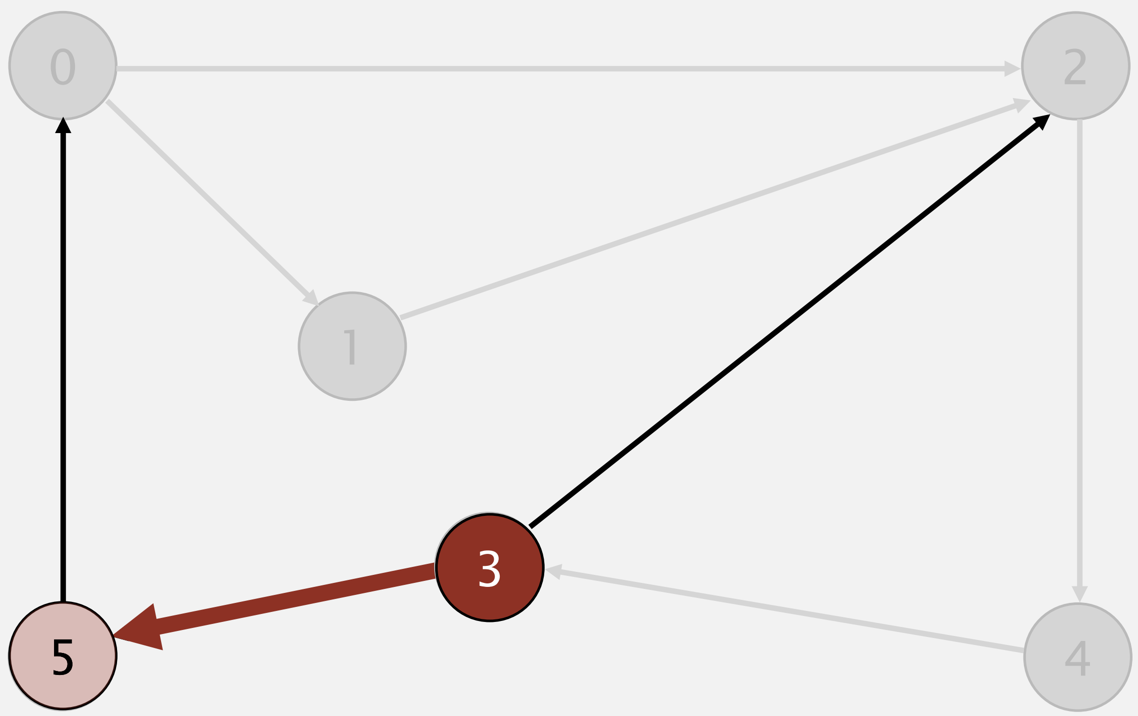
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	–	–

dequeue 3

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue

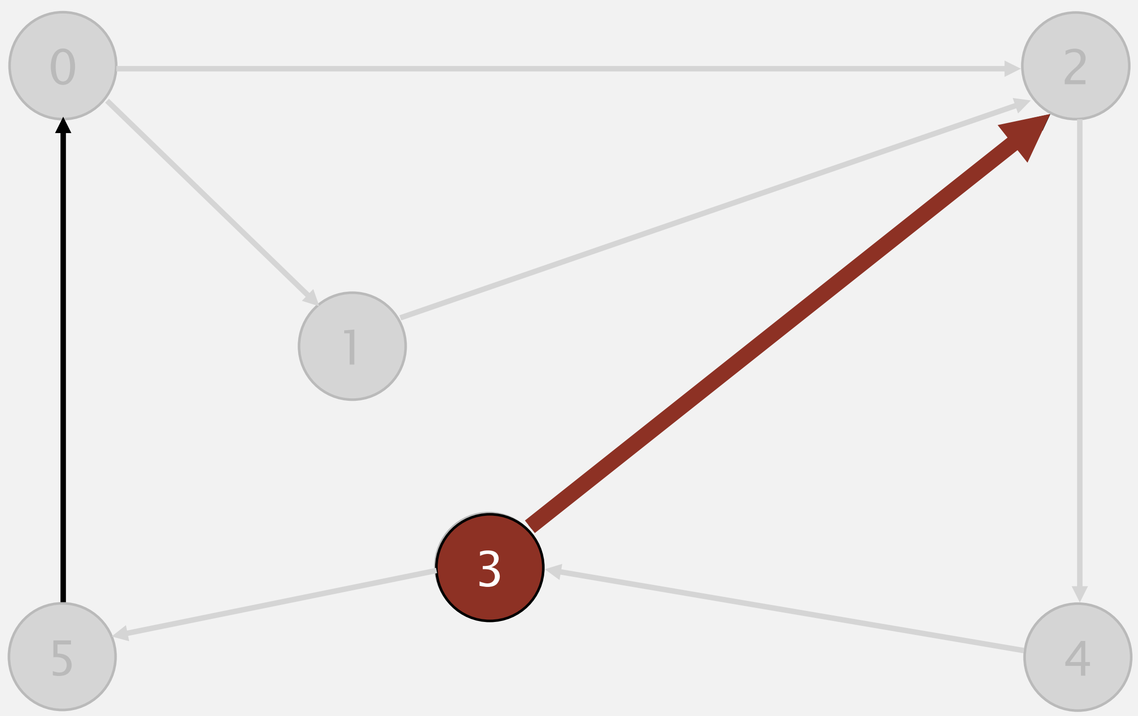
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	3	4
5	–	–

dequeue 3: check 5 and check 2

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue

5

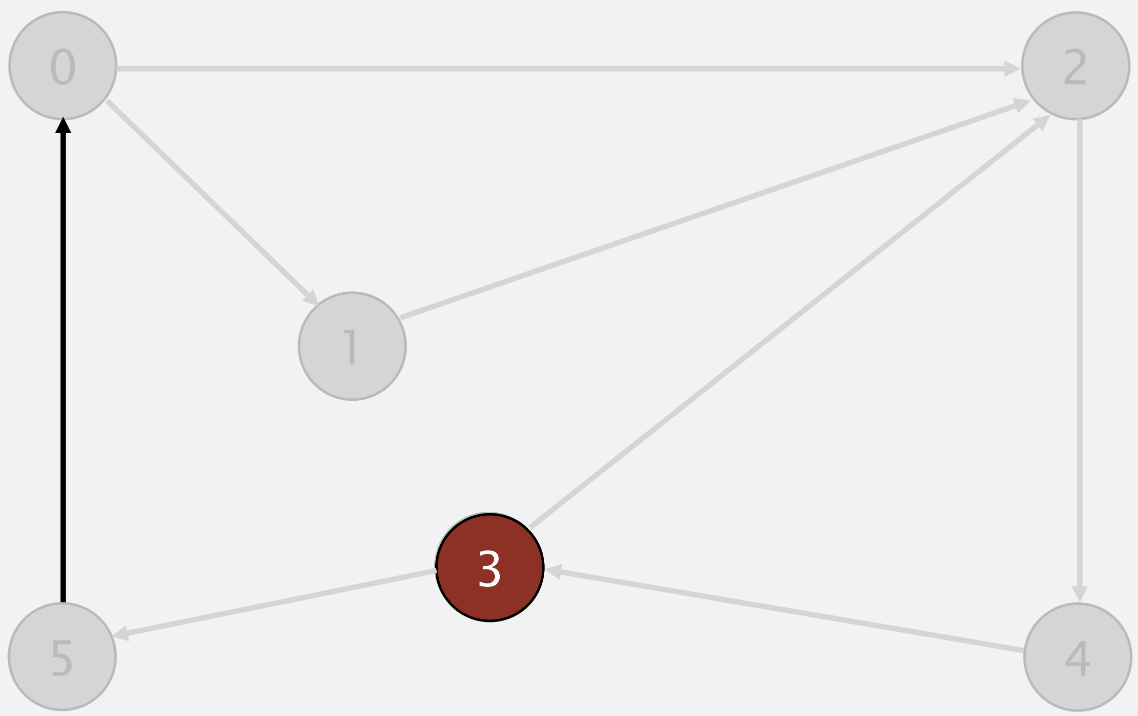
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 3: check 5 and **check 2**

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue

5

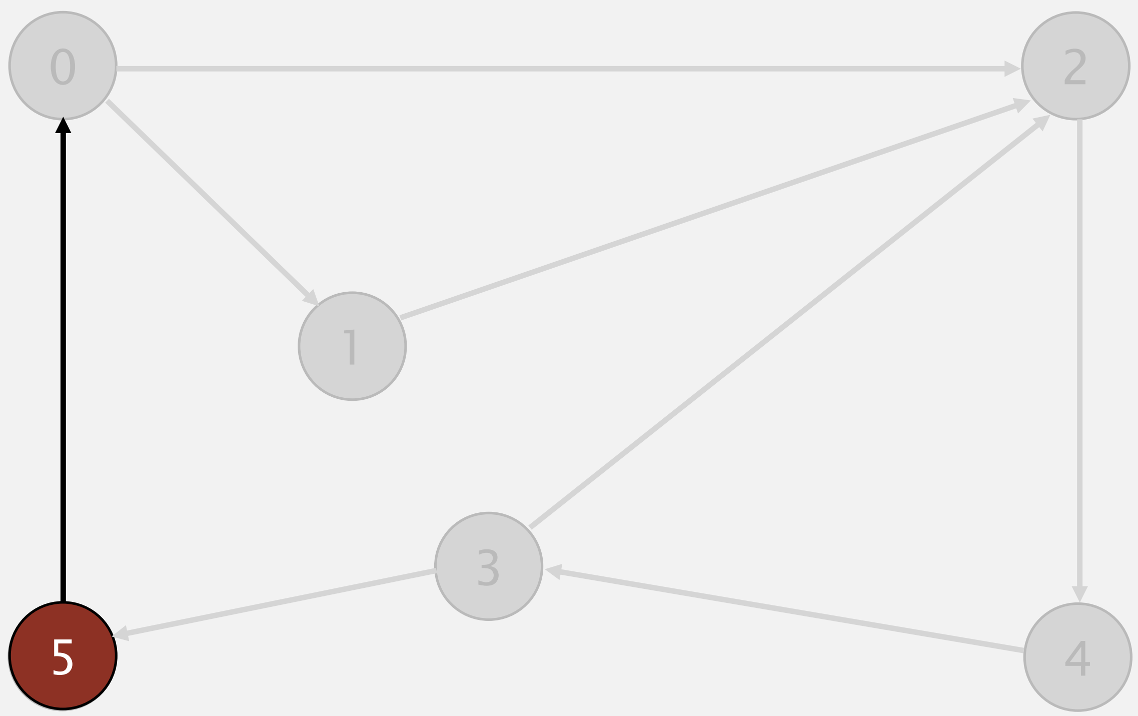
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

3 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue

5

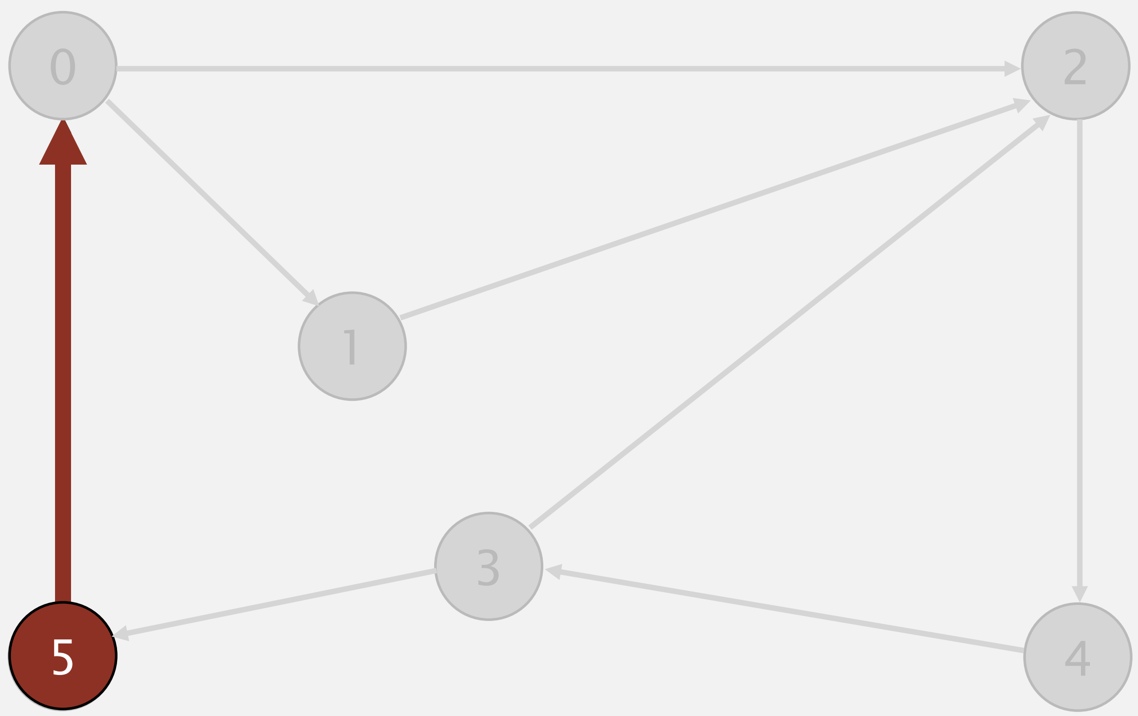
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 5

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices pointing from *v* and mark them.



queue

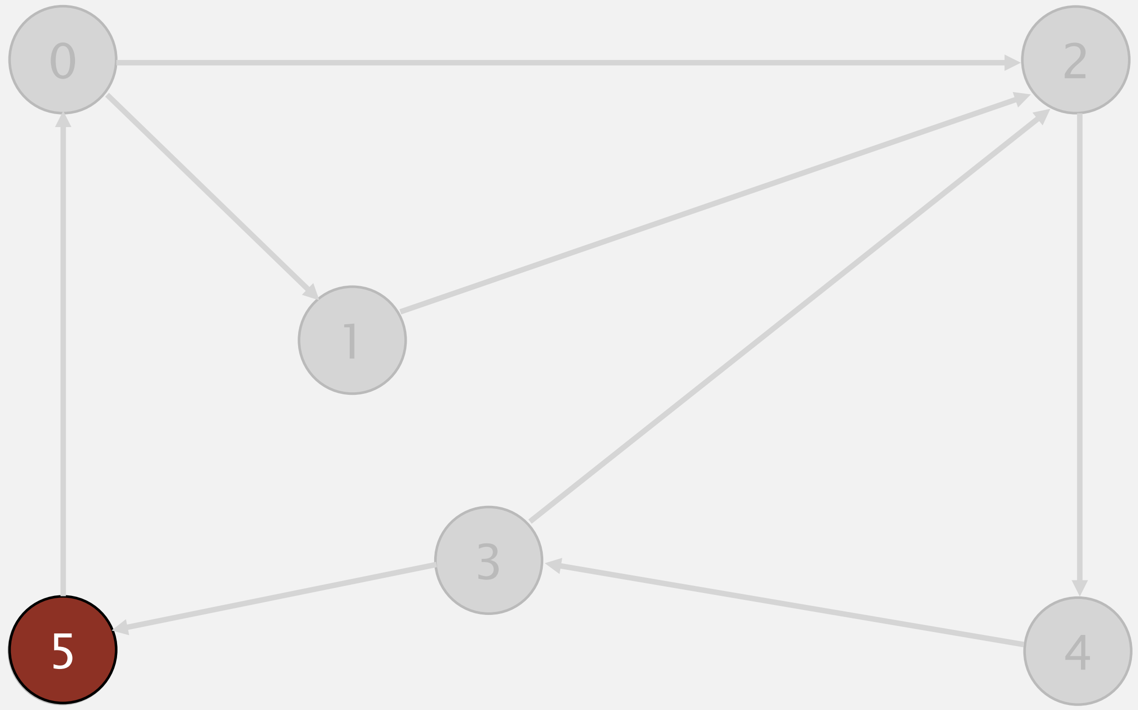
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 5: check 0

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



queue

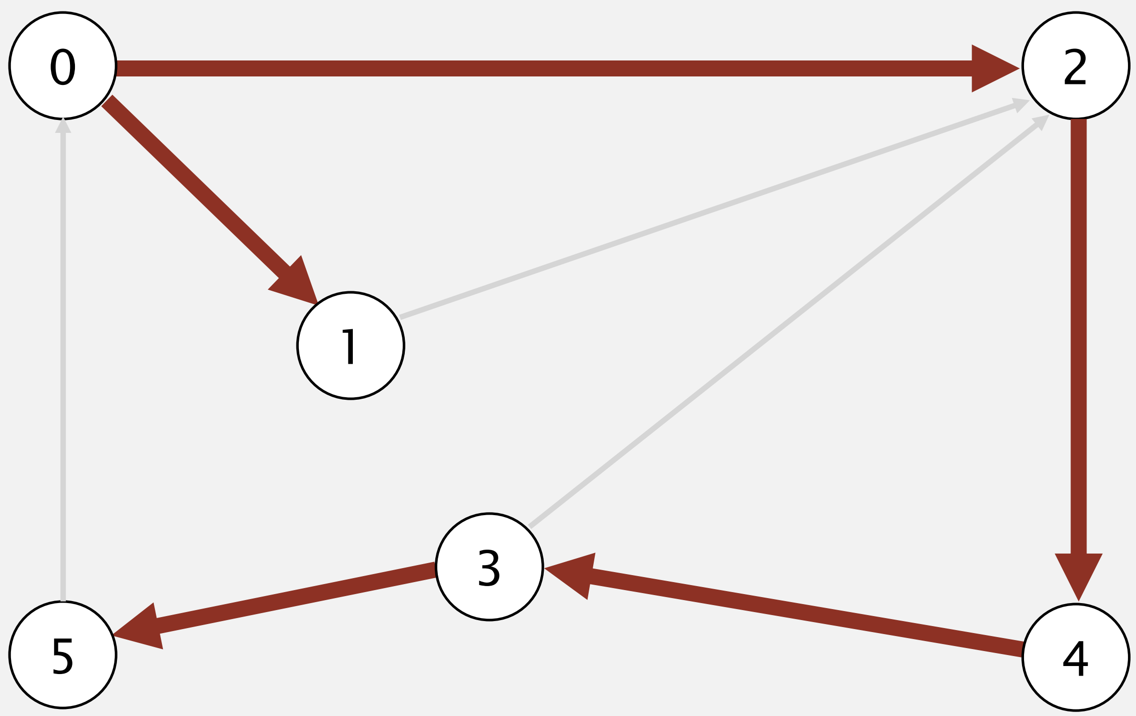
v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

5 done

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

done