DATA STRUCTURE

DR.RAMZI SAEFAN

BY:FARAH HATEM

POWERUNIT

[Analysis of Algorithms]

← هنا هنشوف كودين، كيف نقدر نحلل الـ performance - مثلاً نقدر نحلل الـ analysis

Remember: for any code, it will be asked about → excution time
→ memory usage

→ you are asked to write code to do something.
+ analyze the following code

you may have → written code (by you)
→ given code
to analyze

to analyze, you should think objectively

→ you have 2 performance metrics:
(objective method)

→ Basically, in general, you have a subjective way, or an objective way, to judge something, anything in life, subjective does not depend on any metrics, but objective does.
for algorithms we use objective

Back to the 2 performance metrics (measurable metrics)

① processing (excution time)
② used memory.

→ When trying to improve these two metrics sometimes, ~~they~~ there will be tradeoff between them, and sometimes each one will be independent

Why to analyze algorithms?

→ if you could find a relationship between, excution time and input size, memory usage and excution time

excution time = $f(n)$ → function of input size

memory usage = $f(n)$ → " " "

so you can :-

① predict performance — خصوصًا لما بدي اعرف أداؤه عند أعداد مستخدمين كثار

② Compare algorithms

③ provide guarantees — اعطيه ضمان انه ما يوقع لما يصير عليه استخدام كبير،

فمع ما وجد العلاقات اللي فوق، بحاول الاقي قديش راح يعطيني response و excution time و بتأكد حسبهم اذا ما ويب سايت ح يصير down اولا.

primary practical reason: avoid performance bugs

performance bugs: you get right results of the code, but not in the performance I wish

Because the programmer didn't understand performance characteristics

→ this means programmer didn't do the analysis correctly for the code, wrong prediction, wrong comparisons, and failing guarantees.

فمثلاً الكومبيوتر بيعطل لما نشغله ، او بياخذ memory كبيرة ، او وقت تنفيذ ضخم

Scientific method to analysis algorithms:-

① Observe (in terms of input size)
(memory usage and excution time interms of input size) for different input sizes

→ observations should be repeatable

repeatable : ~~having same results each time code is executed input~~

① platform specification

② all results are taken.

② Hypothesize : find function equation for excution time and memory in terms of input size (after observation results)

→ should be falsifiable

by one counter example the Hypothesis will be called false.

سوف نحل ١٩ خلال اعمل observations أكثر من مرة دي كتير اقرب للمرجع تماماً

③ predict (theoritical)

10,000 input size → ? 50 sec

④ verify : find (experimental) value

- run code on input size (50,000)

as in prediction, and compare

⑤ Validate (repeating till agreement)

Example (Slide 7) → 3-Sum :-

3-Sum is problem well-known in algorithms and data structures.

3-Sum is about how many triples sum to exactly zero

* we want to make analysis for 3-Sum

commands → ① more 8ints.txt → linux command opens the file you typed

② javac ThreeSum 8ints.txt
java class ↵ ↳ parameter

→ to do a compilation for some class in command. type: javac ThreeSum.java.

.class file الـ (bytecode) compilation بعمل

→ to run the class type:

java ThreeSum .
parameters بنبعتلو

parameters set from netbeans: Run → set proj config → customize → Arguments.

These parameters are given to the main method in main class ✓ (the first excuted code in main class)

→ Empirical analysis slide 10

→ Data analysis (Hypothesis) using log-log plot

→ linear equation principle

$$\log_2(T(N)) = b \log_2(N) + c$$

$$T(N) = 2^{b \log_2(N)} + 2^c$$

$$T(N) = N^b + 2^c$$

b = slope = 3

$$T(N) = N^3 + 2^c$$

from observation table.

$$T(8000) = (8000)^3 + 2^c$$

$$2^c = \frac{T(8000)}{(8000)^3} \rightarrow c = \log_2\left(\frac{T(8000)}{(8000)^3}\right)$$

So for log-log plot :-

found by observation table.

$$T(N) = 2^{\textcircled{c}} \times N^{\textcircled{b}} \rightarrow \text{slope} = c_1 N^b$$

(power law.

* لو ع انا run الكود على computer مختلف،
مش شرط يعطي تيم مطابقة لل observation، لكن راح يعطيك
نفس ال relationship. وال slope، وال constant
نفسهم

حتى لما أعيد ال run ع نفس الجهاز، ع كل تيم ال observation
تتغير بصورة طفيفة
حسب انواع ال inputs، لكن بالملاحظة ال slope ← بظل
ثابت.

How slope and constant (2^c) are affected?

→ System independent effects.
[determines exponent, (slope) in power law]
* Algorithm   * Input data.

→ System dependent effects
[determines constant ($a=2^c$), in power law]
Hardware, Software, System.

→ Till now this is called Experimental method.

→ Mathematical models for running time

→ Total running time = $\sum_{i=1}^{n}$ cost x frequency.

cost: affected by : machine, compiler.

frequency: " " : algorithm, input data.

Hint:- - primitive operation → constant time.
- ~~a no~~ non-primitive operations:
depends on frequency that can't be determined while mathematically analyzing.
like arrays accesses in loops, with unknown iterations number.

Example / slide 19

increament (N → 2N)

لأنه ممكن يكون جوا الـ input وكمان، وممكن يكون لا

increament here is = (i++) + (count++)

↓ ↓

N times + (0 → N)

N + (0 → N)

N → 2N

Remember → $\sum_{i=1}^{N} i = \frac{N*(N+1)}{2}$

Geometric series

Rule :- Use some basic operation as a proxy for running time / most frequented

→ example / slide 22

array accesses :-
Depends on external loop.

i → i=0	j<N	j → N-1
i=1	j<N	j → N-2
i=2	j<N	j → N-3
i=N-2	j<N	j → 2
i=N-1	j<N	j → 1

→ 1 ⟶ N-1

½ N(N-1) → 2 arrays to access each iteration

then frequency = N(N-1)

Rule:- The internal loop has number of iterations that is independent from the external loop, then use $(N * K)$

Otherwise:- analyze

Example:-
```
for(i=N, i>0, i--)
    for(j=i, i>0, j--)
        count++;
```

dependent N : 3✓ 2✓ 1✓ 0✗
3✓ 2✓ 1✓ 0✗

i	j
N	N
N-1	N-1
N-2	N-2
2	2
1	1

→ frequency: for internal loop iterations

$$= \frac{N(N+1)}{2}$$

* example :-

```
for (i=N; i>0, i--)
    for(j=k, j>0, j--)
        count++
```

independent : inter

frequency for internal loop = k * N

Special way to find the frequency :-

$$\binom{N}{\text{loops number}} = \frac{N!}{(N-2)! * 2!}$$ → assuming loops number = 2 in a nested loop ✓

example slide 20 ← * Depending on each other

→ Simplifications of mathematical:-

① Cost model: find most frequent operation
② Tilde approximation: ignore lower power terms.
   $T(N) \rightarrow \sim T(N)$
   $\lim_{N \to \infty} \frac{\sim T(N)}{T(N)} \approx 1$
③ growth order → next section

still remember that:
Total running time $= \sum_{i=0}^{n} C_i \times f_i$

Example slide 25:-

Approximately array accesses as a function of input size N (cost model)
approximately → Tilde

→ Dependent internal loop:-

$$\binom{N}{2} = \frac{N!}{(N-2)!\,2!} = \frac{N(N-1)}{2}$$

→ $N(N-1) = N^2 - N \approx N^2$ array accesses.

Example slide 26:-

using cost model, Tilde approximation

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

$$\frac{1}{6}N^3 + \ldots\ldots \text{ (lower terms) / ignored.}$$

$\rightarrow \frac{1}{2}N^3$ = array accesses.

→ Order-of-growth classification

→ Coefficients varies from compiler to compiler, then they are ignored here.

→ Set of functions of algorithms

→ ① constant is chosen to be 1, for coefficients are ignored.

→ ② linearithmic is NlogN

why it is called growth order and why the coefficient is ignored.

$\sim T(N) = \frac{1}{6} N^3 \rightarrow$ by growth order $\rightarrow N^3$

N is input size: when $N = 4 \rightarrow N = 8 \rightarrow \frac{T(8)}{T(4)}$

or $N = 16 \rightarrow N = 32 \rightarrow \frac{T(32)}{T(16)}$

So from $N \rightarrow 2N \rightarrow \frac{T(2N)}{T(N)} = \frac{\frac{1}{6}(2N)^3}{\frac{1}{6}(N^3)}$

$= 8$

→ when we duplicate the input, then will result the 2 to the power 3, regardless what the coefficient is, it is gonna be ignored for this reason

3-Sum with ~~sorting~~ sorting:-

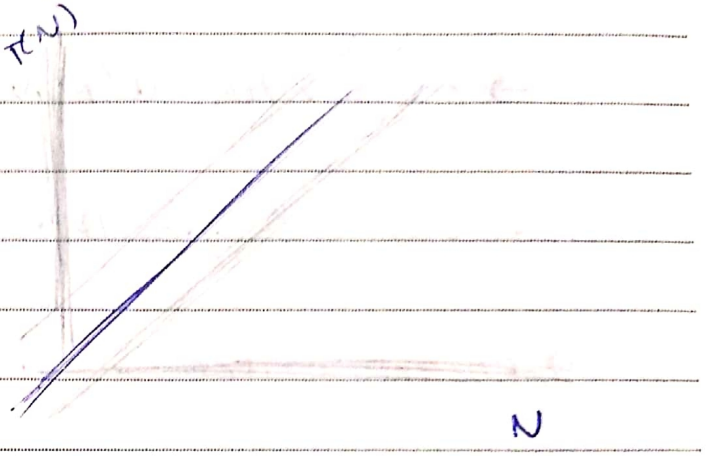
→ complexity: $N^2 + N^2 \log N$

→ order of growth: $N^2 \log N$

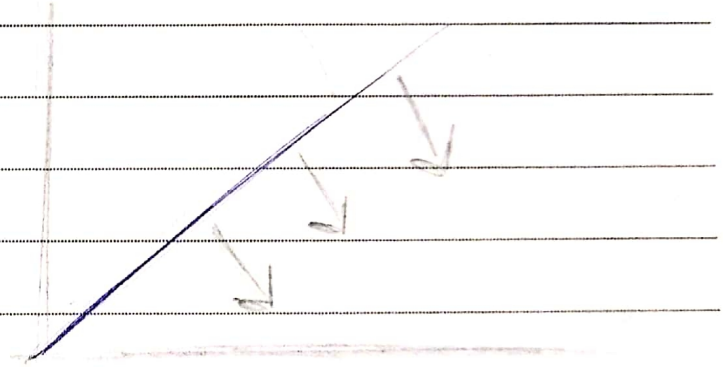
→ Better order of growth, faster in practice ✓

Theory of algorithms:-

Θ → average :-



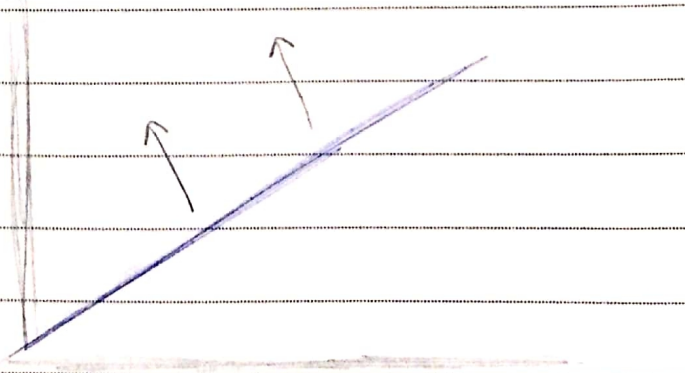
O → worst



→ $\Omega = O$: optimal solution.

Ω → Best

Theoritical, not easy, you may not find a solution with Θ or $O = \Omega$



Memory:-

→ kilobytes:- 2^{10} bytes

← كل objects دائمًا بتكون مضاعفات ال 8bytes
لأن الكومبيوتر بيقسم الـ blocks على 8 لأنه ال addressing بتاعه كل block واحد 8 bytes

→ bytes of arrays always +24

* Metadata → array بالكومبايل بتخزن فيها معلومات عن ال one dimentional

→ two-dimensional arrays is Tikle approximately viewed.

→ Memory usage for objects:-

→ Reference → 8 bytes / for 64-bit computers

→ padding → completing bytes up to a number that is a multiple of 8

→ Stacks and Queues :-

→ Stack : last in, first out.
→ the last element added is the first element read.

→ Queue : first in, first out.

→ for stack :-
pushing is inserting
poping is removing

→ for queue :-
enqueue is inserting
dequeue is ~~poping~~ removing.

Interface : API ↓
Aplication Programming Interface.

حالياً اوصل بين 3 اشياء :-

① Interface ② Implementation ③ client →

→ main functionality on this datastructure or algorithm
→ only a description for the function, a prototype for it, what it is, what its parameters and returning type

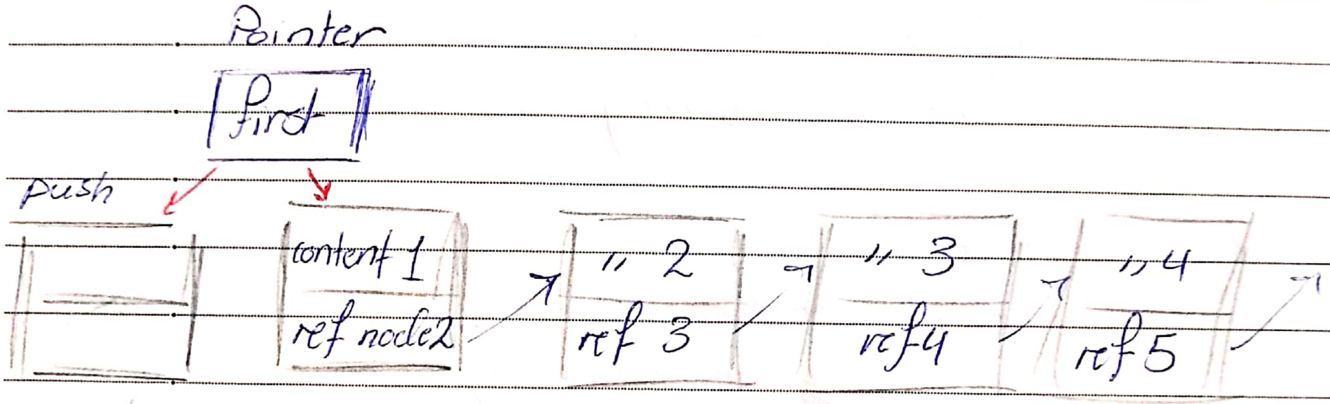
The class + the code for each function of those described in API

→ The client code calls the class by creating an object and calling methods of that class.

Stack:-

مثلاً حالة لو بدي ادخل string وارجع الاوبجكت

Stack: linked list implementation:



① add a new item on the top of the stack, and let pointer (first) point on it.

② push the items.

③ pop (read the element) (then remove it)

(written in slide 5)

fixed capacity implementation

→ you have to know the size of the stack before you create it

The problem of loitering: a non needed object that is not cleaned by garbage. it can be fixed by assigning null instead of removed item.
(the pop item is not deleted)

The problem of underflow
when you want to pop an element from stack, and there are no element left.

← linked-list implementation performance.
for every item we use 40 bytes.

Resizing Array (method-1)

push(): → increase by 1
pop(): → decrease by 1

calculating excution time for this operation:

element #1	1	array access
" #2	1	+ 2 (to copy and write)
" #3	1	+ 4 "
" #4	1	+ 6 "
" #5	1	+ 8 "
element #N →	1	+ 2(N-1)

$= N + 2(1+2+3+4+\cdots+(N-1))$

$= N + 2 * \frac{(N-1)N}{2}$

$= N + N^2 - N$

$= N^2$

How to grow array without resizing frequently? method - 2

array full? → duplicate

calculating excution time:-

element 1	→	1	
〃 2	→	1	+ 2
〃 3	→	1	+ 4
〃 4	→	1	
〃 5	→	1	+ 8
〃 6	→	1	
〃 7	→	1	
〃 8	→	1	
〃 9	→	1	+ 16

N = 2 3 4 5 6 7 8
size = 2 4 8 16

notice: when (elements = $\frac{N}{2}$)
(size = N)

so:- $\frac{N}{2}$: 1 + N

$\frac{N}{2}$ + 1 : 1

$\frac{N}{2}$ + 2 : 1

⋮

$N + 2(1 + 2 + 4 + \cdots + \frac{N}{2})$ → [illegible] أقل من N

$= 3N$ approximately

[Shrinking]

first try:-

→ double when full
→ halve when half.

For Efficient solution:
→ double when full
→ halve when one-quarter full

25% → 100% full array

← إذا كتير وهمتم بال memory، إلجأ لل resizing array
← إذا كتير وهمتم إني أضمن إنه كل العمليات تاخد نفس ال average time
إلجأ لل linked-list.

Queus :-

linked- list implementation:-

← في ال enqueue بطلع عال last - فقط بستخدم ال first عالماشي

اتأكد انه هاد هو اول item

← في ال dequeue بشتغل عال first . فقط بستخدم ال last عالماشي

اتأكد انه هاد هو آخر item.

Just do a little change to your code

class student implements comparable<student>

.... name;

.... studentID;

.... DoB;

.... major;

public in compareTo (Student that st){

returns -1 (this < that)

" +1 (this > that)

" 0 => equal (this / that)

}

}

a full example in slide 12

* Comparable is an Interface, not a class

Example:-

```
class Student ( ){
    public String name;
    ,,      ,,  studentID;
    Date DoB;
    String mejor;
}
```

```
Student [ ] SA= new Student[50]
```

Array.sort(SA) → this is wrong and can't work, Java can't identify the natural order by the built-in function

→ So we need a way to tell Java the natural order (A≤B)

Chapter // Elementary Sorts

→ we always use here ascending order for sorting

اذا بدنا العكس بنجي نعدل بسط

Note:-
to be able to sort, only for data structures having a total order.

Types of sorting functions in Java:-

① Arrays.sort(array) → sorts arrays

② collections.sort(collection → sorts any thing that implements collections interface like array list, or linked list

Builtin functions

→ Worst case

→ $\frac{1}{2}N^2 + \frac{1}{2}N^2$ = complete time complexity

compares exchanges.

→ array is in descending order with no duplicates

In comparison with selection sort:-

Selection compares $\frac{N^2}{2}$ exch N

as random ← Insertion compares $\frac{N^2}{4}$ exch $\frac{N^2}{4}$

For insertion, compares are better, exchanges are worse.

[2] Insertion Sort

iteration i ↓

بقارن ال i مع الي قبلها ع يسارها، اذا اصغر منه، بعمل swap

← بضلني اعمل swap لحد ما الاقي الي ع يساره swap بيستقر ترتيبه الصحيح

← لكن اذا المقارنة حالت false، يعني i مش اصغر من الي ع يسارها

بوقف ↓، وبروح عال element # الي بعده، يعني بنشئ new iteration

Break;

the class:-

① method sort (static)

② method less } same as

③ method exch } selection sort.

* all methods only take comparable types as parameters.

Time complexity.

→ randomly-ordered / distinct keys array

→ $\frac{1}{4}N^2 + \frac{1}{4}N^2$ = complete complexity.

compares (first term), exchanges (second term)

→ Best case

→ $N-1$ = time complexity (only for compares)

0 exchanges

* the array is basically ordered (ascending)

Algorithms of sorting:

[1] Selection sort

complexity in field = $\frac{N^2}{2}$

complexity in Growth order = N^2

complete complexity = $\frac{N^2}{2} + N$

$\frac{N^2}{2}$ → comparisons → Quadratic time

N → exchanges → linear number

principle : Swap a[i], a[min]

* class name is Selection
* the method sort is static to be able to call it by class name.
* when using parameters of type comparable in methods (less), and (exch), it means only accept parameters that can be compared.
  → and insert as well

→ it always takes the same growth order (N^2) which is Quadratic time, even if the array is sorted, or partially sorted before.

In selection sort, in the best case, or worst case, or on average it takes the same time complexity.

Memory with [mergesort]

merge sort uses extra space proportional to N, due to recursions, and copying the array to aux (Not inplace)

→ NOTE!!

A sorting algorithm is in-place if it uses $< c \log N$ extra memory (overhead), like insertion and selection sort.

improvements for memory usage

① use aux[] array of length $\frac{1}{2} N$ instead of N

② In-place merge (very hard.)

Practical improvements:-

technique ① Use insertion sort for small subarrays.
→ Cutoff to insertion sort for ≈ 10 items

technique ② Stop if already sorted: that is when the largest item in first half ≤ smallest item in second half.

Chapter :- Merge and Quick Sort

[1] Merge Sort

Basic plan ① Divide array into two halves.
② recursively sort each half
③ Merge two halves

→ * when having two equal elements in the two halves, we take one from the first half
→ * we copy the array to (aux) for the operation, but the final result will be in the original array.

Time complexity ✓

number of compares = S

and S is $\leq \lceil N \log N \rceil$ → folder. at most.

and that is N = number of compares for each level

logN = levels number.

number of array accesses = R

and R is $\leq \lceil 6N \log N \rceil$ → folder. at most.

Note that anything follows the divide and conquer principle will end with something log N time complexity.

Topic (Shuffling):-

approach ①:- Shuffle sort.

→ Generate a random real number for each array entry

→ sort array.

result: produces a uniformly random permutation

all the operation depends on: sorting

* so complexity (Growth) = N^2, Quadratic

approach ②:- Knuth shuffle.

→ iteration i : pick integer r randomly between 0 and i, uniformly at random.

→ swap a[i], and a[r]

result: produces a uniformly random permutation of input array in linear time.

* complexity (Growth) = N, linear

Good Algorithms are better than super-computers, Great algorithms are better than both.

Details:-

① partioning in place :-
making extra arrays mates it easier + stable but is not worth cost

② terminating loop :-
testing pointer cross is tricky

* you either use shuffling or picking a random partitioning item as alternative in each sub array.

Time complexity :-
Best case : $N \log N$
Average case $1.39 N \log N$
Worst case $\frac{1}{2} N^2$

Characteristics of performance :-
✓ Randomized algorithm.
✓ Guaranteed
✓ Running time depends on random shuffle
✓ faster than Merge sort in practic because of less data movement
✓ Not stable
✓ partioning takes constant extra space
✓ recursion depth takes logarithmic extra space / can be guaranteed by recurring on smaller sub arrays.

[2] Quick Sort

→ time complexity for it is higher than merge sort, but experimentally is faster.

Basic Plan:-

① Shuffle array (partially sorted) عشوائيا لتكون

② Partition

→ Scan i from left to right so long as (a[i] < a[lo]) if wrong stop

→ Scan j from right to left so long as (a[j] > a[lo]) if wrong stop

→ Exchange a[i] with a[j]

when pointers cross: exchange a[lo] with a[j] then you have partitioning item (a[j])

③ Sort each subarray recursively

Stability:-

a stable sort preserves the relative order of items with equal keys

Selection sort not stable long-distance exchange can move one equal item past another

Insertion sort stable Equal items never move past each other

Merge sort stable Takes from left subarray if equal key.

→ loitering

solved by adding this part for delMax()

```
key item = a[--N]
item[N] = null;
return item.
```

This Priority Queue can be implemented

① ~~or~~ non-ordered array.

added elements will be at the end of array

② ordered array

inserting element in the correct order

Chapter 11 Priority Queues:-

Remember to distinguish between:-

Interface   API implementation   Client Code

Priority Queue deletion:-

which item to remove?
smallest or largest.

Priority Queue API

→ Generic items are comparable:-
→ key must be comparable
two types MaxPQ , MinPQ
MaxPQ → delMax()
MinPQ → delMin()

Priority Queue problems:-

→ should know capacitance previouslly
solved by resizable array.
→ underflow
solved by exception throwing
→ over flow
resizing array

← حاجزين مساحة بمقدار ال capacitance ، مش ضروري تكون محتاجينها
← عملية ال del Max بتاخد وقت

practical improvement idea:-

→ Insertion sort small subarrays.
cutoff to insertion sort for ≈10 items.

Application using Quick sort: selection.
other examples: - order statistics
- find (top k)

→ Quick Selection.
→ it is first sorted
→ then tracked to determine (k)
لما k الأكبر من j ، ادور جهة اليمين.
وبالعكس صحيح
partitioning item. هيبقى الـ k هي الـ

→ Quick select takes linear time on average.
2N compares.

We use binary heap if you want to remove max key (used to build priority queue for more efficiency)

Queue used to remove first element entered

Stack " " " last " "

Heap implementation

insertion cost: $\log N$ at most due to swimming up

deleting Max cost: $2 \log N$ at most due to sinking down.

Practical improvements:

→ d-way tree array instead of binary tree.

like 3-way tree array (each node has 3 children)

→ Swim takes $\log_d N$ compares

→ sink " $d \log_d N$ "

Binary Heap representation:-

→ Array representation

→ Heap ordered:

→ Not totally ordered but the first element should be the biggest (largest key at a[1])

→ parent's key no smaller than children's keys

→ indices start in 1

→ take nodes in level order

→ use array indices to move through:-

node at k → parent is at $\left(\frac{k}{2}\right)$ integer division

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

→ Inserting, and removing max key

Insert: → add node at end, then swim it up to be at the right position, considering heap representation.

Remove Max: → exchange root with node at the end, then sink it down, always in the direction of the largest children at each level.

Topic:- Binary Trees

Binary tree

↓

empty or node with links to left and right binary trees

↓

Height is Highest level

Complete binary tree
perfect balanced
(each node has two children)
except for bottom level

↓

Height is Highest level or $\text{floor}(\log_2 N)$

↓

increases when only N is a power of 2

Heap-Sort in-place

plan

→ view input array as a complete binary tree

→ build the max heap

→ sort down

complexity

→ 2N compares + N exchanges for construction

uses $2N \log N$ (in-place) → worst case

problems:-

- Inner loop longer than Quick sort
- poor use of cache
- Not stable

but is optimal for both time and space

HeapSort

✓ complexity = N log N

✓ extra array of length N is used.

✓ Not stable

plan:- (additional array used)

→ ~~that~~ keys in arbitrary order

→ re-order them to heap representation by sinking down.

→ re-sort array ~~to~~ by using removing max.

Binary heap problems

Uneder flow: exception
over flow : resize array.

→ Can be used for:-
→ removing arbitrary item.
→ change priority of same item.

Can be implemented for Minimum-oriented priority queue with few change.

Chapter // Binary search Trees

→ remember binary trees.

→ each node has a key with this requierment:-

Key is larger than all keys in its left

key is smaller than all key in its right

* A BST is a reference to a root Node

* Node has four fields:-

key, value, reference to left subtree, reference to right subtree

* restrictions on key type

should be comparable.

→ no restrictions on values

Operations:-

search: if less → go left

if greater → go right.

if equal → search hit

compares cost = 1 + depth of node.

Insert: if less → go left

if greater → go right

if null → insert

compares cost = 1 + depth of node

Tree shape depends on order of insertion,

best case / typical case / worst case

Note for equality test

you can use equal() for generic types
→ for user-defined types, it's different.

Override the method with modifications:-

↳ standard recipe for implementation

① First check if this equals that.
② check if ~~this~~ the passed object is null
③ check if this and that are from the same type.
cast the passed object to the class type you override equals() in.

④ compare significant fields.
primitive types → ==
(Object) type → equals()
array → apply to each element

Best practices:-
① compare mostly likely fields to differ.
② compareTo() consistent with equals()
both return true if x equals y

chapter // Symbol tables

have a lot of names, and is like dictionaries in python (maps, dictionaries, associative arrays)

→ key-value pairs
you insert a value with specific key,
or search for a value of some key.

Note:-
other arrays are Generalized dictionaries, indices are keys, values are values.

[Symbol table API]

→ you cannot insert any null value. (it is only allowed for delet
because null is returned when get() finds nothing
* put() overwrites old value with the new one

→ it accepts any generic type.
for comparison
→ when keys are Comparable → use compareTo())
→ when " " generic → use equal to test equality

For best practices: use immutable types for symbol table keys

*Deletion:-

→ Delete the maximum key:-

→ go left till you find a node with null left link

→ replace that node by its right link to upper max node

→ update subtree counts

→ Hibbard deletion (delete a node with key k)

→ search for node of key (k)

case ①

0 children

delete t by

ting parent

nk to null

case ②

1 child

→ delete t by replacing parent link to the grand child.

case ③

2 children

→ find successor x

→ replace t with x

slide (40)

→ ~~delete~~ update counts

problems of Hibbard:-

→ not symmetric

can be solved by successor and its opposite.

$\text{cost} = \sqrt{N}$

Topic: ordered operands

→ maximum ✓

→ minimum ✓

→ size(): how many node in the subtree rooted at given key

→ rank(): how many keys < k

→ select(): key of given rank

** prepare node count

count = size()

Inorder traversal

left, enqueue key root, right

Get: return the value of key, if exit
other wise return null
compares cost = 1 + depth of node

Put: Associate value with key
compares cost = 1 + depth of node
↳ if key in tree → reset value
if not → add new node