DATA STRUCTURES

DR. RAMZI SAEFAN

DONE BY
SARAH KASASBEH

POWERUNIT

Part 1

- write code to do something / analyze the following code
  - written / code
  - given code

- 2 performance metrices or (objective methods) → there are metrics.
  - measurable metrics :
    1- Processing (execution) time
    2- used memory

function ↔ input size → Provide guarantees, Predict Performance, Comparison algorithms

- Processing time = $F_1(n)$
- memory usage = $F_2(n)$

- Provide guarantees → ex:- guarantee that the website will not be down → ex:- processing time is less than (Page not found) time.

Part 2

- Scientific method to analysis of algorithms:

1) observe :- depends on input size and how it affects performance, memory usage, CPU execution time ... etc.
   - these observations should be repeatable in the same platform specifications.
2) Hypothesize :- find function equation for execution time and memory usage in terms of input size. → Counter example
   - Hypothesize should be falsifiable :- one Proof is enough to say that the hypothesize is false.
   - we should make more observations to be more sure that the hypothesize is true.

3) Prediction :- Predict the result of certain input size according to the hypothesize.

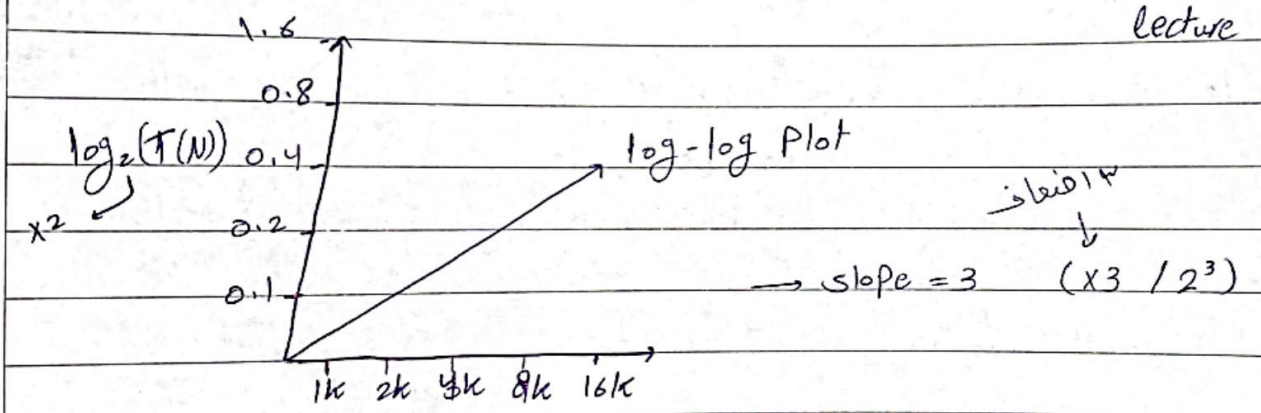
then ↓

4) Verify :- find experimental value (ex:- run code on certain input size and compare with the predicted value)

5) Validate :- edit the functionality to get more accurate result if needed.

note :- (more) Command → used to view the content of certain file.
(java) Command → run
(javaC) → Compile

week (2)
lecture 1-1



- $y = ax + b \longrightarrow \log_2 T(N) = a * \log_2(N) + b$

$\rightarrow T(N) = 2^{a * \log_2^2(N) + b} = (2^{\log_2 N})^a * 2^b$

$= (N)^a * 2^b$

$T(N) = (N)^a * 2^b \longrightarrow$ Power law

where a :- slope

$= N^3 * 2^b \rightarrow T(8000) = (8000)^3 * 2^b$

So $\rightarrow 2^b = \frac{T(8000)}{(8000)^2} \rightarrow b = \log_2 \frac{T(8000)}{8000}$

(from the table)

Lecture 1-2

- Mathematical model :-

analyze code → Cost for each operation
# of operations → frequency for each operation

$$\text{Total Cost} = \sum_{i=0}^{n} \text{Cost}_i * \text{Frequency}_i$$

Cost depends on: machine, compiler
Frequency depends on: algorithms / input

- note : floating points operations > integers operations

Slide (19)

$$\text{Mathematical Cost} = 2c_1 + 2c_2 + (N+1)c_3 + Nc_4 + Nc_5 + (N \text{ to } 2N)c_6$$

(1 time, largest value)

$$* \quad \sum_{i=0}^{N} i = \frac{N(N+1)}{2}$$

Slide (20)

j < n ... i=0 ⇒ j<n ⇒ N
(< less than) i=1 ⇒ j<n ⇒ N-1
for j i=2 ⇒ j<n ⇒ N-2
i = N-2 ⇒ j<n ⇒ 2
i = N-1 ⇒ j<n ⇒ 1

(i)

# of less than comp. = $(N+1) + N + (N-1) + \cdots + 2 + 1$

$$= \sum_{i=1}^{N+1} i = \frac{(N+1)(N+2)}{2} \quad \#$$

Slide (20)

equal to Compare → $i=0 \Rightarrow N-1$

$i=1 \Rightarrow N-2$

$i=2 \Rightarrow N-3$

$i=N-2 \Rightarrow 2$

$i=N-1 \Rightarrow 1$

So → $\frac{(N-1)(N)}{2}$

- array access → each time of (equal to) we access array two times.

Lecture 2-1

ext. loop → N times

int. loop → k times (for each ext. loop iteration the int. loop occurs k times)

ex:- for (i=0; i<N; i++) → N times

for (j=0; j<k; j++) → k times

if ( ) → N * k

- The int. loop has number of iterations that is independent from the external loop counter.

ex:- for (i=0; i<N; i++)

for (j=i; j<k; j++)

if ( )

i=0 → j → N times

i=1 → j → N-1 times

i=2 → j → N-2 times

} we should use summation because (j) depends on (i)

ex:- for (i = N ; i > 0 ; i--)
for (j = i ; j > 0 ; j--) → j depends on i
Count++

i = N (j) → N times
i = N-1 → N-1 times
i = N-2 → N-2 times
i = N-3 → N-3 times
⋮
i = 1 → 1

} جمع 2, 05 ع
j
= $\frac{N(N+1)}{2}$ times → Count++

ex:- for (i = N ; i > 0 ; i--)
for (j = k ; j > 0 ; j--)
Count++

ext. loop = N times
int. loop = k times
} = kxN
↳ for each ext. iteration

Lecture 2-2

for (i = 0 , i < N , i++)
for (j = i+1 ; j < N ; j++)

كم مرة تلف loop الـ $\binom{N}{2}$ = $\frac{N!}{(N-2)!\,2!}$ = $\frac{N*(N-1)*(N-2)!}{(N-2)!\,2!}$
↳ لا تهم

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

note:- $\binom{N}{\#\text{of loops}}$

(1) (2) (3)

- experimental → mathematical → simplification on mathematical

$$\sum_{i=0}^{N} C_i * Teq_i$$

- Simplification on mathematical

1) cost model :- find most frequent operation (most time consuming)
2) Tilde approximation (ignore lower power terms)
3) growth order (ignore lower power terms + leading coefficient

Tilde approx. → $T(N) \longrightarrow \sim T(N)$ →

$$\lim_{N \to \infty} \frac{\sim T(N)}{T(N)} \approx 1$$

large N

$N = 4 \rightarrow N = 8 \Rightarrow \frac{T(8)}{T(4)}$ → How much does the complexity increased.

---

order of growth

Lecture 2-3

constant (1) → without any loop

Logarithmic (log N) → while (N > 1) { N = N/2; ... }

linear (N) → loop (i=0, i++)

quadratic (N^2) → 2 loops (i=0, j=0, i++, j++)

Cubic (N^3) → 3 loops (i=0, j=0, k=0, i++, j++, k++)

exponential (2^N) → exhaustive search
↳ input size (power) → worst case

ex:

if Best Case = $f_1 = n$
if worst case = $f_2 = n \lg N$

} Average = $\frac{f_1 + f_2}{2} = \frac{1 + \log N}{2}$

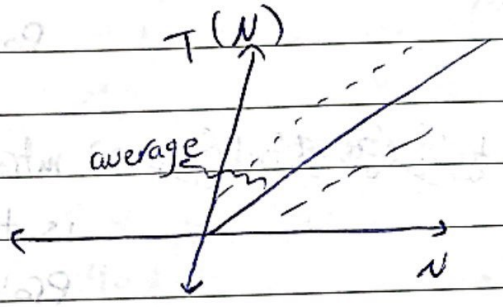
$= \frac{1}{2} \log N$

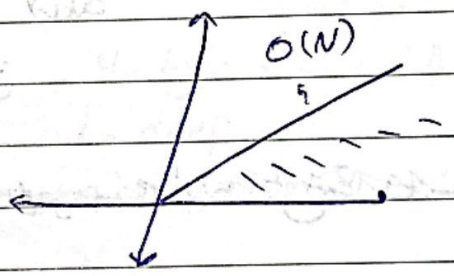
• we focuse on the worst case.

Θ (theta) → average

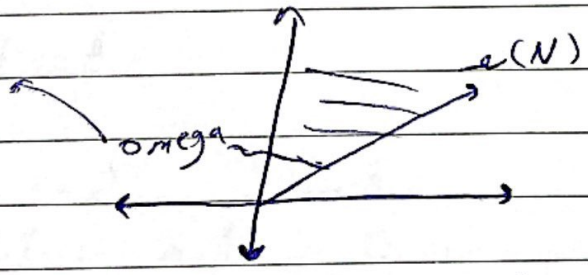
O (oh) → worst

Ω (omega) → Best





theoritical lower bound (optimal lower target)



↳ not easy, you may not find a solution with Θ or $O = \Omega$, so Ω is the target we try to achieve

if $\Omega = O$ → optimal solution

Ω → proof that you have to do these steps (at least) to solve the problem

- develop an alg. $\Rightarrow$ Complexity $O(f_1(N))$
  Ω for this problem $\Omega(f_2(N))$
  where $f_1(N) > f_2(N)$

Computer scientests $\Rightarrow$ 1) find a new algorithm that takes $O(f_3(N))$ where $f_3(N) < f_1(N)$ as close as possible to $f_2(N)$

more practical

2) introduce a proof that $\Omega(f_4(N))$ is the lower bound for such problem, where $f_4(N) > f_2(N)$ and if $f_1(N) = f_4(N)$ then your solution is optimal

Lecture 2-2

Kilobyte → 2^{10} , Megabyte → 2^{20} , GiGabyte → 2^{30}

Padding → عدد الـ bytes الي بنضيفهم ليصير
المجموع يقسم على 8

```
Public class String
{
    Private char[] value;
    private int offset;
    private int count;
    private int hash;
}
```

16 bytes → object overhead
8 bytes → reference (value)
3*4 bytes → integers
36 bytes
+ 4 bytes (Padding)
40 byte → shallow
↳ without array size
40 byte + 24 + 2N = 64 + 2N
↳ deep

Lecture 2-1

stack → last in, first out
Queue → first in → first out

Push = insert , PoP = read → stack
enQueue = insert , deQueue = read → Queue

stack ex:- "Hello My Name is Ramzi"

```
St.Push("Hello");
St.Push("My");
St.Push("Name");
St.Push("Ramzi");
```

1) System.out.println(St.Pop())
2)
3)
4)

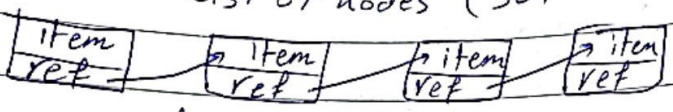
output (Ramzi / is / name / My / Hello)

stack

Ramzi
is
name
My
Hello

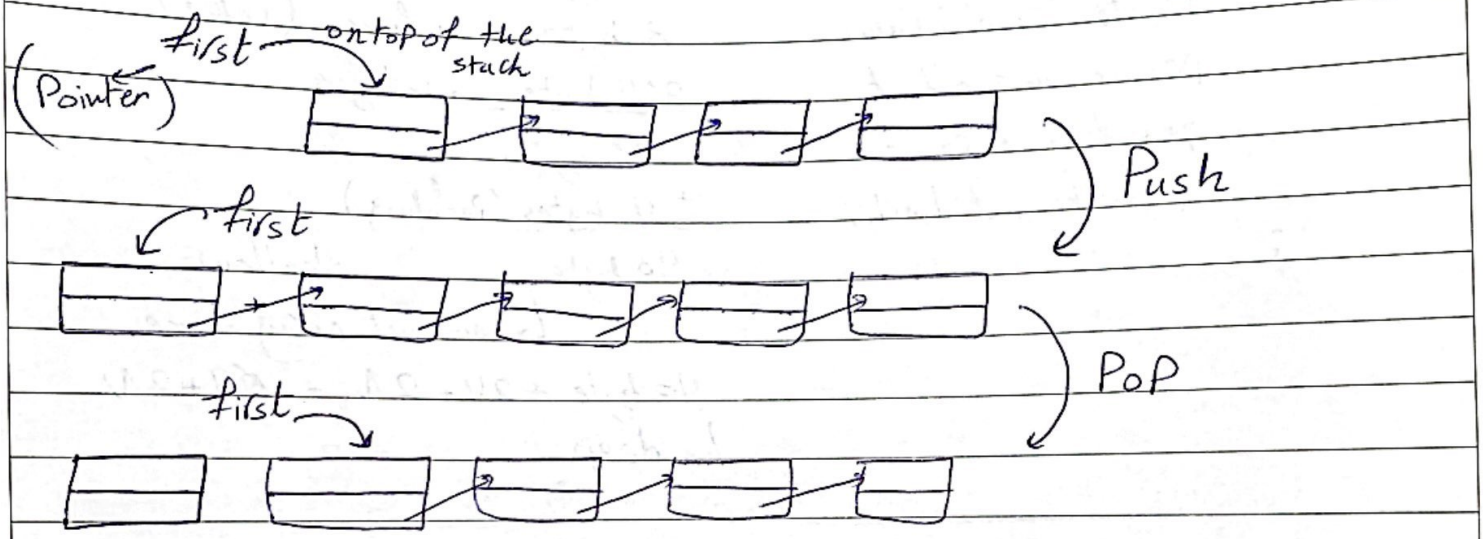
- Linked - List implementation

list of nodes (sequential)



nodes composed of 2 things :-

1. item
2. reference to the next node



↓

read this then remove.

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

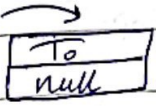
```
LinkedStackofStrings LS = new LinkedStackofStrings;
```

- first → null

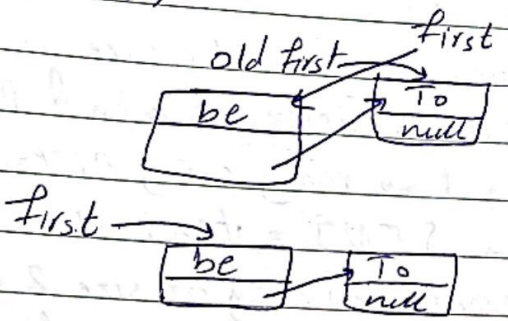
LS.push("To"); → oldfirst = null

first → New node

first →



LS.Push("be");




```
Public String Pop()
{
  String item = first.item;  // be
  first = first.next;  // To
  return item;  // be
}
```

String st = LS.Pop();

- (be) node will be deleted by the Garbage collector.

- Stack complexity (memory usage) $\approx 40N$, where N = # of nodes
- each Node takes 40 bytes

Fixed-capacity Stack :- array implementation lecture 2-2

Pointer

Push() :- add new item at s[N] and N will be at s[N+1]

Pop() :- remove item from s[N-1]

- Problem → Poped elements are not deleted, because they are within the array. ⟹ loitering :- a non needed object that is not cleaned by G.C.

So →
```
String st = s[--N];
s[N] = null;
return st;
```

week 4

Lecture 1-1

- resizing array (1st approach)
  - ↳ for each push → increase the size and push
  - ↳ for each pop → decrease size and pop.
- to push N elements, how many array access are needed ?
  - ele. 1 → S[N] = item; N++; (1 array access)
  - ele. 2 → create new array of size 2 and copy the element from the old array to the new array and make the stack points to the new array, then push the element.
    - ↳ 2 array access (for copy 1 element) + 1 push
  - ele. 3 → 4 array access to copy 2 elements + 1 push
  - ele. 4 → 1 + 6
  - ele. 5 → 1 + 8
  - ele. N → 1 + 2(N-1)

$$= N + 2 \cdot (1+2+3+4+\cdots+(N-1))$$
$$= N + 2\,\frac{(N-1)(N)}{2} = N^2 \rightarrow \text{to push } N \text{ ele.}$$

- in linked-list implementation → constant time for each push
- resizing array (1st approach) → N^2 to push N elements

(2nd approach)

Push ele. 1 → 1 (size = 1)
Push ele. 2 → 1 + 2 (size now = 2) } x2
Push ele. 3 → 1 + 4 (size now = 4)
Push ele. 4 → 1 } x2 (doubling)
Push ele. 5 → 1 + 8 (size now = 8)
Push ele. 6 → 1
" " . 7 → 1
" " . 8 → 8
" " . 9 → 1 + 16 (size now = 16)

ele. $\frac{N}{2}$ → 1 + N (size = N)

$N + 2 * (1+2+4 \cdots + \frac{N}{2}) < N$

$N + 2 \times N = \boxed{\sim 3N}$ Compared with N^2 in 1st approach which is much bigger.

ex:-

lecture 1-2

10^6 push on the stack.

↳ on doubling, takes long time (worst case)

↳ O.W takes Constant time

how many times, doubling happens? $\log 10^6 = 20$

how many times, constant time? $10^6 - 20 = 999,980$

So we use averag analysis.

ex: Stack with size = 16 → and # of elements (N) = 16

which means → full capacity

if we make M Pop → size = 16, N = 5

↳ Capacity ≈ 0.25

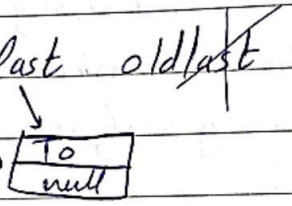
metrics	Linked-list impl.	resizing-array impl.
execution-time	Constant time (worst case)	$\sim N$ (worst)
	// // (average)	Constant (average)
	// // (best case)	Constant (best)
	↳ assume Constant = C_1	↳ assume Constant = C_2
	$C_1 > C_2$	
memory usage	$\sim 40N$	$\sim 8N \leftrightarrow \sim 32N$
	where $\sim 40N > \sim 8N \leftrightarrow \sim 32N$	

• in Queue we need 2 pointrs (on top, on tail)
enqueue ↓ dequeue ↓

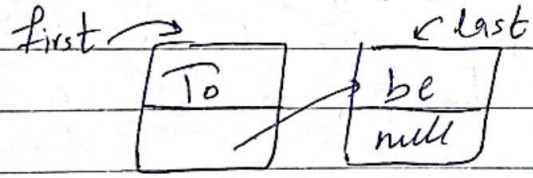
• LinkedQueueOfStrings LQ = new LinkedQueueOfStrings();

1) first ↓ null last ↓ null

2) LQ.enqueue("To"); → first element.



3) LQ.enqueue("be");



4) String S = LQ.dequeue(); // To

5) String S = LQ.dequeue();

(List) is an interface → we can make object from this interface, but we can't make instantiation from an interface → we just have instance objects from concrete classes (new concrete class)

• Queue q1 = new Queue → false X
because Queue is an interface
So → Queue q1 = new LinkedList();
↳ implement Queue

week 5
lecture 1

- Total order → can be sorted
  - Antisymmetry :- if $V \leq W$ and $W \leq V \rightarrow V = W$
  - Transitivity :- if $V \leq W$ and $W \leq X \rightarrow V \leq X$
  - Totality :- $V \leq W$ (or) $W \leq V$ (or) both

  } we can find total order

- To sort any data, it should have Total order.
  - anti-symmetry
  - Transitivity
  - Totality

in java → Arrays.sort(array)
↳ Collection.sort(collection)
↳ something that implements Collection interface.

- we need a way to tell java the natural order ($A \leq B$)
  ↳ by (implements Comparable) → then method (compare to)

---

lecture 2-D

insertion sort → Compare each element with the entry to its left / swapping between 2 adjacent elements only.

- ↳ Best case → sorted array (without any exchanges)
- ↳ worst case → array sorted in reverse order

- shuffle sort → Complexity (growth order) = N^2
- Knuth shuffle → " " = N → much better.

week 6

Lecture 1-1

• mergesort

• recursion :- function calls itself.

ex:- Public void func1 (int a, int b)
{ int x, y;
func1 (x, y); → recursive call (infinite loop)
statement 4; so we must have exit
} condition

Lecture 1-2

Mergesort.

if each half is sorted → compl. = N
if each quarter is sorted → " = N
⋮
= N log N compares at most

• 6N → for each 2 halves merging
• $6\frac{N}{2}$ → " " 2 quarters merging } → each level takes 6N array access
• $6\frac{N}{4}$ → " " "

⇓

6N log N array accesses at most

• disadvantage (mergesort) → takes additional array of size N (aux)

• improvement
1) aux half array ($N \rightarrow \frac{N}{2}$ additional memory)
2) Cut-off = 10 ⇒ 30% enhanced
3) don't do unnecessary merge.

Lecture 2-1

- stability → stable if it achieve relative order.

B_1 B_2 A_3

Class {
letter
number
}

according to letter (A_3 B_1 B_2)

↓

if B_1 before B_2 → stable + sorted

. (A_3 B_2 B_1) → sorted + not stable

- any long distance exchange can break the relative order.

selection sort → not stable
insertion sort → stable
~~merge~~ mergesort → stable

Quicksort
- best case → totally random data
- worst case → fully sorted.

week7, Lecture 1-1

↳ faster than mergesort because in merge sort we need to copy data

. Quick sort is not stable.

Lecture 2-1

. Complete tree → each node has two children.
↳ Perfectly balanced (except the bottom level)

. height = $\lceil \log_2 N \rceil$ where N = # of nodes
of binary tree
complete

- heap-ordered
  - keys inside nodes
  - each parent key > children keys

if these two conditions are true then heap-ordered

- Parent of any Node = $\frac{\text{Node index}}{2}$
- largest element = a[1] which is root element.

Lecture 2-2

- binary tree
  - if parent < children → sink the parent ↓
  - if children > parent → swim (child) ↑

- if N = # of node then array size = N+1 (first ele. is empty)
- Children of any element in the array = 2x(index of parent) and 2x(" " " +1)

- Sink → Cost = 2 lg N compares where lg N = levels and in each level there are 2 compares.
- Swim → Cost = lg N at most.

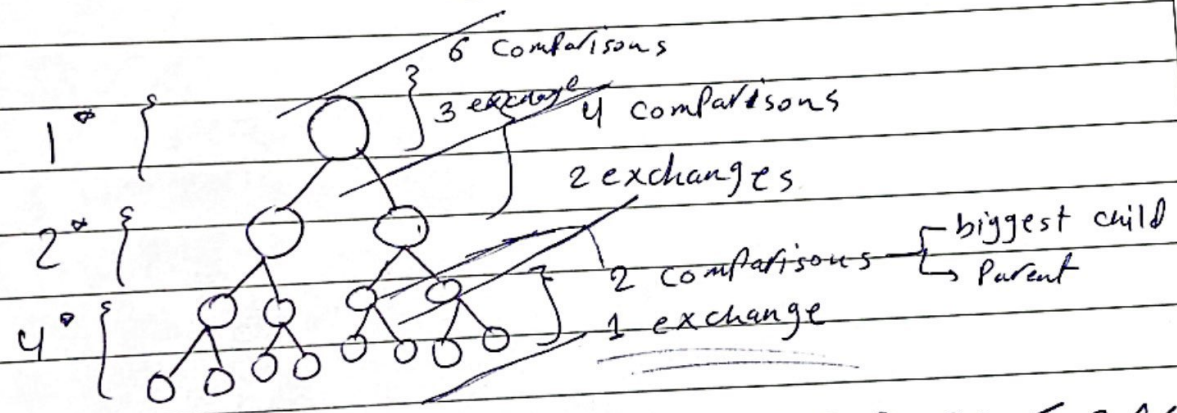
week 8
lecture 1-1

- Sorting with binary heap
  - $N \log N$ → to create the heap
  - $2N \log N$ → sorting

- disadvantages of sorting with binary heaps
  - 1) additional array (b_9)
  - 2) not stable
- Complexity = $N \log N$ at most, which is good

- Heapsort (in-place)

2 N Compares at most and N exchange (Heap Construction)



# of compares = $6*1 + 4*2 + 2*4 = 6+8+8 = 22 < 2N$

# of exchanges = $11 < N$

- N nodes sink = 2 log N comparisons

{ 2N log N compars
  N log N exchange } → Heapsort

- heapsort performance is much worse than Quicksort.