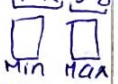
ARRAY

1) Finding min / max

@ By simple traversal :-
NO of comparison = n-1
TC = O(n) , SC = O(1)

ⓑ Tournament method (4 3 3 9)
NO of comparison = n-1
TC = O(n) , SC = O(n)

2) In an array of n ele, min no of comparisons needed to
@ Find min/max = (n-1)
ⓑ Find min & max = $\frac{3n}{2} - 2$
↳ same for LL



Tournament method
TC = O(n), SC = O(n)

3) Finding 2nd min
@ Find 1st then 2nd min
NO of comp = (n-1)+(n-2) = 2n-3
ⓑ Tournament method
Find min of list of ele compared with min.
NO of comp = (n-1)+(log n - 1) = n+log n - 2

4) Searching → Linear TC = O(n) ; Binary (sorted array) O(log$_2$n)

5) NO of elements in range LB-UB = UB - LB + 1

6) By default, elements of array are initialized by garbage value (if not initialized)

7) Array is static (size is prefixed) data structure, LL is dynamic.

8) int a[n]; int a[]; are wrong as size is unknown.

9) Array name is base address.

10) add(arr[i]) = Base add + (i - LB) * size of element

11) 2D array :- A[LB$_1$ UB$_1$][LB$_2$ UB$_2$]
NO of rows (m) = UB$_1$ - LB$_1$ + 1
NO of columns (n) = UB$_2$ - LB$_2$ + 1

12) Row major order (By default)
add(A[i][j]) = Base addr + [(i - LB$_1$) * n + (j - LB$_2$)] * sizeof 1 ele

13) Column major order
add(A[i][j]) = Base addr + [(j - LB$_2$) * m + (i - LB$_1$)] * sizeof 1 ele

14) All multidimentional arrays are stored in 1D array.

15) int a[][5][6] = {.....}; a[][3]
1st dimension is optional

16) 3D array :- A[LB$_1$ UB$_1$][LB$_2$ UB$_2$][LB$_3$ UB$_3$]
NO of 2D = p, NO of rows = q, columns = r

17) Using RMO
add(A[i][j][k]) = Base addr + [(i - LB1) * q * r + (j - LB2) * r + (k - LB3)] * W

18) using CMO
add(A[i][j][k]) = Base addr + [(i - LB1) * q r + (k - LB3) * q + (j - LB2)] * W

19) Lower triangular matrix :-
@ RMO : add(A[i][j]) = Base addr + [(i - LB1) natural no sum + (j - LB2)] * W
(if j ≤ i)
ⓑ CMO : add(A[i][j]) = Base addr + [n natural no sum - (UB$_2$ - j + 1) natural no sum + (i - j)] * W

20) Lower △ matrix RMO = Upper △ matrix CMO
LTM CMO = UTM RMO

HASHING

1) Searching technique
2) Direct address table : key is address
- Search time = O(1) (every case)
- Drawback : infinite size hash table
3) Types of hash functions :-
@ Division modulo method (K mod m)
- If m = 2^k , Key = (101000111001)$_2$
Key mod m = LSB k bits
- Prefer m value prime number, not close to 2^k.

ⓑ Digit extraction method

Probe : searching

© Mid square method : sq key then find address from middle elements

ⓓ Fold Boundary Method

ⓔ Fold shifting Method

4) Best hash fu" means having min no. of collision.

5) Collision resolution techniques:
- open addressing / closed hashing → Linear probing (default), Quadratic probing, Random probing / Double hashing
- closed addr. / open hashing (chaining)

6) Chaining:
- Max chain length with n keys = n (WC)
- searching time = O(1) (BC/AC); = O(n) (WC)
- insertion time = O(1) (Every case) (at beginning)
- Deletion time = O(1) (BC); = O(n) (WC)
- space is wasted for addresses.
- If no. of keys is unknown or greater than table size then use chaining.
- cache performance is not good as keys are not stored at contiguous memory location.

7) Max no. of keys open addressing can accomodate = Size of hashtable ∴ called closed hashing

8) Linear probing:
LP = (HF + i) mod m , i ≥ 0

9) Quadratic probing:
QP = (HF + i²) mod m , i ≥ 0
i → probe number

10) Double hashing:
DH = (HF1 + i * HF2) mod m

11) Random Probing:
RP = (HF + i * random no.) mod m

12) Double hashing is better among open addressing.

13) Load factor (α) = No. of keys / Total no. of slots
- Open addressing: 0 ≤ α ≤ 1
- closed addressing: 0 ≤ α (chaining)

14) Space utilization = No. of occupied slots / Total slots
0 ≤ S ≤ 1

15) Linear probing has problem of primary clustering ie. if 2 keys contains same hash value initially, both follow same path in linear manner.

16) Quadratic probing has problem of secondary clustering ie. if 2 keys has same initial hash value, both follow same path in quadratic manner.

17) Double hashing don't have clustering problem.

18) Linear probing: Search / Insert / Delete
TC = O(1) BC
O(m) WC, AC
m: hash table size

19) Quadratic probing: Search / Insert / Delete
TC = O(1) BC
O(m) WC, AC

20) While searching an ele if blank comes means ele not exist.

21) To delete an ele, replace it with special symbol.

22) Double hashing: TC = O(1) BC, AC
O(m) WC

23) Perfect hashing has (m+1) hash table & hash function. (like 2D array)
searching time = O(1) (Every case)

24) Uniform hashing: each key has equal probability of going to any slot.

25) Universal hash fu" works independent of type of keys.

26) No. of possible probe sequence for a key
- Linear / quadratic probing: m permutations
- uniform hashing: m! permutations
- Double hashing: m² "

27) Expected no. of probes in an unsuccessful search (getting empty slot) of open addressing (assuming uniform hashing) = $\frac{1}{1-\alpha}$, $\alpha = \frac{n}{m}$

E(x) → (1-α) → Empty slot (1)
E(x) → α → filled slot (1 + E(x))

E(x) = (1-α)*1 + α(1 + E(x))

28) Cost of successful search of i^{th} item = cost of unsuccessful search of i^{th} item in $(i-1)$ items

29) Expected no. of probes for successful search in open addressing $= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1-\frac{i-1}{m}}$

$= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

30) An unsuccessful search in chaining takes average case time $O(1+\alpha)$ (assuming uniform hashing)

if $m \ll n \Rightarrow \alpha \approx n \Rightarrow O(1+\alpha)$ is not constant

if $m = O(n) \Rightarrow \alpha = O(1) \Rightarrow O(1+\alpha)$ is constant

31) Successful search for i^{th} item $\equiv$ inserting i^{th} item in table $\equiv$ unsuccessful search for i^{th} item with $i-1$ items in table

$\hookrightarrow \alpha = \frac{i-1}{m}$

32) Successful search in chaining takes average case time $O(1+\alpha)$

$= 1 + \frac{1}{n}\sum_{i=1}^{n} \frac{i-1}{m} = 1 + \frac{1}{mn}\sum_{i=0}^{n-1} i = 1 + \frac{n(n-1)}{2mn}$

$= 1 + \frac{\alpha}{2}$

33) In chaining, average case time is constant $O(1+\alpha)$.
α is constant when $m = O(n)$

34)

	Unsuccessful	Successful
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Open Addressing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

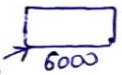
35) In chaining, deletion is easier compared to open addressing

36) Linear probing is default collision resolution technique in closed hashing.

#LINKED LIST

1) Dynamic linear data structure
- Accessing uninitialized pointer → Segmentation error
- malloc returns void pointer that's why typecasting needed.
- memory created using malloc is created in Heap area.

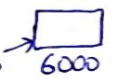
2) Binary search is possible on LL but its not efficient.

3) 

If we free(s), s is called dangling pointer

Pf(s) → 6000
Pf(s→data) → Garbage (or error)
Pf(s→next) → n

To remove dangling pointer do s=NULL;

4) 

s=NULL or s=some other addr. Memory is allocated but can't be accessed, its called Memory leak.

5) Union of 2 LL ⇒ $O(n_1 * n_2)$ (ensure no duplicate is present)

6) Intersection of 2 LL ⇒ $O(n_1 * n_2)$

7) Apending 1 LL at end of other ⇒ O(n)

8) Binary search on LL ⇒ O(n) (Every case)
$n + \frac{n}{2} + \cdots + \frac{n}{2^{\log n}}$
Binary search on array
= O(1) Best case
O(log n) worst, Avg case
Linear search on LL
= O(1) (Best C)
O(n) (WC, AC)

9) Merge sort on LL
TC = O(n log n)
SC = O(log n)

10) Merging two LL (sorted)
TC = O(m+n) (WC)
Inplace
TC = O(min(m,n)) (BC)

* LL can be implemented using 2 arrays.
* Finding all such ele in LL2 that are lesser than LL1→data ⇒ O(n)

11) Merge sort


	TC	SC	
LL:	n log n	(log n)	Inplace log n
Array:	n log n	(n+log n)	Outplace log n

- Mergesort on LL is better than array.

12) On LL, all sorting algo can be applied

13) Reverse a LL ⇒ TC = O(n) (EC) (3 pointer)

14) Finding loop ⇒ TC = O(n) (EC) using BFS/DFS
SC = O(n) ← extra flag field

TC = O(n)
SC = O(1)
using Floyd's algo (Rabbit, tortoise)

15) To delete a node pointed by given pointer ⇒ O(n) (WC) O(1) (BC)

16) To delete data at node pointed by given pointer ⇒ O(1)

17) To go back in circular LL ⇒ O(n)

18) Finding 2nd last ele in circular doubly LL ⇒ O(1) time

Singly LL:

	Insertion	Deletion
1) At beginning	O(1)	O(1)
2) After a given node	O(1)	O(1)
3) Before a given node	O(n)	O(n)
4) At the end	O(n)	O(n)
5) At end (last node address given)	O(1)	O(n)
6) Node in sorted list	O(n)	O(n)
7) Of a given node	—	O(n)

Doubly LL:

	Insertion	Deletion
1) At beginning	O(1)	O(1)
2) After a given node	O(1)	O(1)
3) Before a given node	O(1)	O(1)
4) At the end	O(n)	O(n)
5) At the end (last node address given)	O(1)	O(1)
6) Node in sorted list	O(n)	O(n)
7) Of a given node	—	O(1)

Circular LL : (Singly)

circular doubly LL

	Insertion	Del	Insertion	Deletion
1. At beginning	O(n)	O(n)	O(1)	O(1)
2. After a given node	O(1)	O(1)	O(1)	O(1)
3. Before a " "	O(n)	O(n)	O(1)	O(1)
4. At end	O(n)	O(n)	O(1)	O(1)
5. At end (last node address given)	O(1)	O(n)	O(1)	O(1)
6. Node in sorted list	O(n)	O(n)	O(n)	O(n)
7. Of a given node	-	O(n)	-	O(1)

Trick NOTE: In SLL, insertion before a given node & deletion of a given node can be done in constant time (by deleting node after it & swapping data)
* Its not deletion of node, its deletion of data.

#STACK

- TOP value when stack is empty is called Bottom index of stack.

1) LIFO / FILO property

2) Push, Pop ⇒ O(1) (EC) using array or LL with top at beginning

- Linear data structure

* Multiple stacks using single array :-

Kth stack initial TOP of stack = (K − 0) * N − 1 ← size of array
(N ← No. of stack; starting stack no.)

- Stack Si is full if T_i = initial TOP of stack Si+1
- Stack Si is empty if T_i = initial TOP of stack Si
- Push / POP ⇒ O(1) (EC)

* Multiple stacks using single array efficiently :

- Stack is full if $T_L + 1 = T_R$
- Left stack empty if $T_L = -1$
  Right " " if $T_R = N$

3) Insertion & deletion at TOP (at some end).

* Lifetime - Push kone ke baad, POP kone ke pehle ka time gap.

4) Avg Lifetime of stack & queue = $\frac{n(x+y) - x}{n}$
n → no. of ele in stack
x → 1 operation time (push pop takes same time is assumed)
y → time blw 2 operation.

- Stack used to check parenthesis expression balanced or not.

* Queue using stack :
For enqueue ⇒ 1 stack needed ⇒ TC = O(1)
For dequeue ⇒ 2 " "
↳ TC = O(n) WC
O(1) BC, AC

- Queue is underflow if both S1 & S2 are empty.

* Stack using Queue : (Linear queue)
- 2 Linear queue required
- Push ⇒ TC = O(1) EC
- POP ⇒ TC = O(n) EC
- If stack is implemented using 1 circular queue of size n or 1 linear queue of infinite size (like circular array)
  TO POP do : (n−1) DQ, (n−1) EQ, DQ

* Stack Permutation :
- No. of valid permutations with n inputs = Catalan no. = $\frac{2nC_n}{n+1}$
- No. of invalid perm = $n! - \frac{2nC_n}{n+1}$
- Catalan no. $T(n) = \sum_{k=0}^{n-1} T(k)\,T(n-k-1)$

* Recursion :
1. Tail recursion : No work after recursive call.
2. Non Tail " : Has work after recursive call
   - For the given recursive program, on writing equivalent non recursive program
     - if stack almost eliminated : Tail recursion
     - if stack is almost same : Non " "
3. Indirect recursion : A calling B and B calling A.
   - For TC consider TC of that fu" (A/B) having max no. of recursive call.
4. Nested recursion : Ex : A(m−1, A(m, n−1))

• Evaluation of prefix using stack → scan from right to left

Infix - Prefix - Postfix :

1) Prefix: Polish notation
Postfix: Reverse polish
→ Postorder traversal of expression tree

2) Conversion done as per priority and associativity.

NOTE: ^ (exponent) = associativity is right to left.

* Infix to postfix using stacks:

i) Operand : print

ii) Operator
- Priority > TOP : push
- Priority = TOP :
  if assoc R→L then push
  if assoc L→R then pop & push
- Priority < TOP : pop then push all higher priority

- TC = O(n) (operator stack needed)
- When (comes, push,
  when) comes, pop & print till (comes, & pop (.
3) Computers work with postfix expression.
4) Computer converts infix to postfix expression then evaluate using stack.
5) Evaluation of postfix using stack ⇒ TC = O(n)
   → operand stack
6) In postfix start making circle from LHS and in prefix from RHS.
7) No need of bracket in prefix & postfix expression.

Fibinocci series:

1) $T(n) = T(n-1) + T(n-2) + C$
   ⇒ TC = $O(2^n)$ EC
2) Stack space = O(n)
3) Fu" call → preorder
   Fu" completed → postorder
4) In f(n), after n+1 fu" call 1st addition takes place.
5) After last fu" call, n/2 addition there.
6) No of fu" calls, $f(n) = \begin{cases} f(n-1)+f(n-2)+1 & \text{if } n=0 \text{ or } 1 \\ 1 & \text{if } n=0 \text{ or } 1 \end{cases}$
7) No of additions, $f(n) = \begin{cases} f(n-1)+f(n-2)+1 & \text{if } n=0 \text{ or } 1 \\ 0 & \text{if } n=0 \text{ or } 1 \end{cases}$

Tower of Hanoi :

- It has unique solution for min moves.
- TOH(n, A, B, C)
  - n: No of disk
  - A: source tower
  - B: intermediatory tower
  - C: Destination tower

means move n disk from tower A to C.

1) After (n+1) fu" call, 1st move took place.
2) TC = $O(2^n)$ EC , Stack space = O(n)
3) No of fu" call, $f(n) = \begin{cases} 2f(n-1)+1 & \text{if } n=0 \\ 1 & \text{if } n=0 \end{cases}$ ⇒ $f(n) = 2^{n+1} - 1$
4) No of moves, $f(n) = \begin{cases} 2f(n-1)+1 & \text{if } n=0 \\ 0 & \text{if } n=0 \end{cases}$ ⇒ $f(n) = 2^n - 1$

QUEUE

1) Insertion and deletion at different end.
2) FIFO / LILO property

   Deletion ← Front ——— Rear ← Insertion

3) if (F == -1) Queue empty
   if (R+1 == N) " full
   if (F == R) only 1 element present
4) Enqueue, Dequeue ⇒ TC = O(1) EC

➤ Circular Queue :
- if (R+1) mod N == F → Full
- if F == -1 → Empty

➤ Priority Queue :
- Ascending priority Queue (By default)
  smallest ele → highest priority
- Descending priority Queue
  largest ele → highest priority

5) Implementing Ascending priority Queue

using	Enqueue	Dequeue
(i) unsorted array	O(1)	O(n)
(ii) sorted array	O(n) WC, O(1) BC	O(1)
(iii) Minheap	O(log n) WC, O(1) BC	O(log n) WC, O(1) BC

6) Minheap best data structure to implement Asc Priority Queue, & Maxheap for Desc Priority ".

7) Priority Queue & o/p restricted, Deque i/p restricted Queue does not obeys FIFO property.

8) o/p restricted Queue
insertion : Rear
Deletion : Front & Rear

9) i/p restricted Queue
insertion: Front & Rear
Deletion: Front

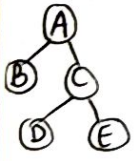
10) Double ended Queue (Deque)
insertion : Front & Rear
Deletion : Front & Rear

TREE

Binary Tree :

1) Every node has max 2 children.

2) has no children → Leaf node
Remaining nodes → internal node



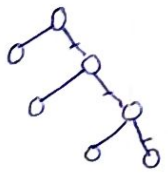
External node/Leaf node : B, D, E
Non leaf node/Internal node : A, C
No of nodes = Leaf nodes + internal nodes

3) Self Referential Data structure: pointer pointing to node of same structure.

4) if (ROOT == NULL) → Empty

5) Make sure termination condition has condition of ROOT = NULL.

6) To count no of leaf nodes (or non leaf nodes) in a binary tree (or total no of nodes):-
TC = O(n) EC
Stack space = {O(log n) BC, AC
O(n) WC

7)


Height = 3
Level = 4
Level = Height + 1

8) Height of empty tree = " " leaf node = 0

9) To find height of given binary tree
TC = O(n) EC
Stack space = {O(n) WC
O(log n) BC, AC

10) Height : max distance from node to leaf.
Level : Number of parent corresponds to given node.

11) Strict Binary tree : either 2 or 0 child.

12) To verify given binary tree is strict BT :
TC = {O(n) WC
O(1) BC root has only 1 child

13) Disconnected acyclic graph → Forest where each connected component is a tree.

14) To verify given 2 Binary tree equal or not :
TC = {O(n) WC
O(1) BC

15) To convert given Binary tree into its mirror image
TC = O(n) EC

NOTE: For every recursion, non-recursion possible.

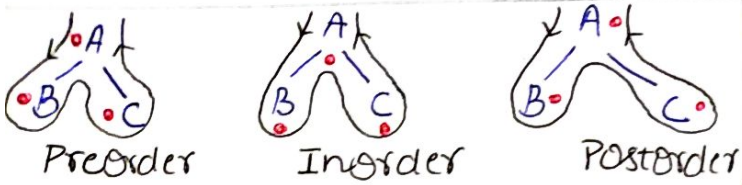
Tree Traversal :

TRICK: draw dot & traverse

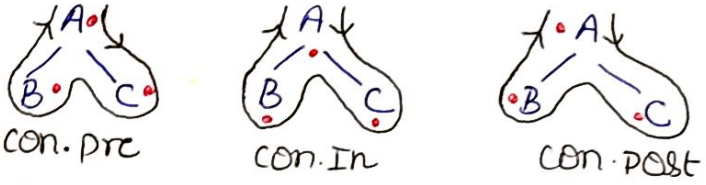
1) Preorder : N L R
Postorder : L R N
Inorder : L N R
⇒ TC = O(n) EC
Stack = {O(log n) BC
O(n) WC

2) Preorder sequence 1st node & Postorder sequence last node is ROOT node.

3) To create unique binary tree we require either (Preorder + Inorder) or (Postorder + Inorder).
TC = {O(n²) WC
O(n) BC

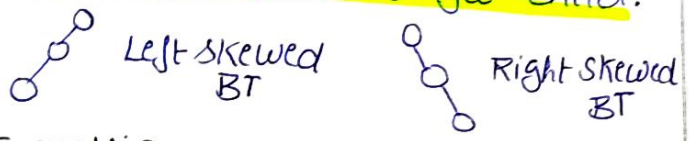


4) converse preorder: N R L
converse Inorder: R N L
converse postorder: R L N

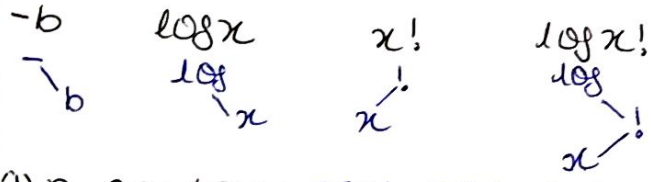


5) converse pre ⟷ post
converse post ⟵ Reverse ⟶ pre
converse In ⟷ In

6) From (Pre + Postorder), we will get unique binary tree only iff there is no node with single child.

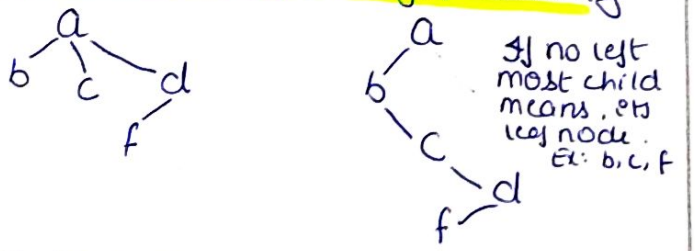
7) 

8) Expression tree:
a * b + c



9) n-ary tree: each node has max n - children.

10) converting n-ary tree into binary tree:
Left most child - Right sibling



11) If pre & postorder of BT given in ques & we need to find inorder, verify from options.

12) We can have unique BT from following combinations:
(In + Pre), (In + Post),
(In + levelorder)

13) Inorder traversal of ternary tree is
Left → root → Middle → Right

Q: A n-ary tree in which every internal node has exactly n children has x internal nodes, y leaf nodes.
degree of internal nodes except root is (n+1)
degree of leaf node is 1

Σ degree = 2E
$x(n+1) + y*1 - 1 = 2(x+y-1)$

* Perfect/complete BT can be uniquely build given any 1 traversal.
* Full BT → any 2 trav (in/pre/post

Heap

1) Almost complete binary tree. (with logn levels)

2) Min heap: Parent ≤ child
Max heap: Parent ≥ child

3) Array representation of heap:
Parent (i) = $\lfloor \frac{i}{2} \rfloor$
left child (i) = 2i
Right child (i) = 2i+1
(index starting from 1.)

4)

	Min heap	Max heap
Min ele	ROOT O(1)	Leaf O(n)
Max ele	Leaf O(n)	ROOT O(1)

5) It is a height balanced tree with logn levels.

6) In complete BT (heap), no. of leaf nodes = $\lceil n/2 \rceil$
Non leaf node (Internal node) = $\lfloor n/2 \rfloor$

NOTE: Array representation is better for CBT & for remaining LL is better.
So represent heap using array.

7) No. of comparisons needed to find min in maxheap with n elements or max in minheap = $\lceil \frac{n}{2} \rceil - 1$

8) In maxheap, for finding (constant)th max
In minheap, for finding (constant)th min
TC = O(1) constant no. of comparisons

9) Insertion in Maxheap/Minheap
TC = { O(log₂n) WC, AC ; O(1) BC }
Algo: insert at last & do heapify up

10) Deletion Algo:
- Swap root node with last ele.
- Delete last node (reduce size)
- Perform heapify down from root

TC = { O(log₂n) WC, AC ; O(1) BC }

11) In array representation,
- Maxheap gives ascending order
- Minheap gives descending order.

12) By default sorted means ascending order.

13) Heapsort: do deletion n times
TC = O(n log n)

14) Heap sort is inplace (no extra space required) and not stable (order of repeted ele changes).

NOTE: Heapify up: Heapify bottom
Heapify down: Heapify top

15) For a given array of n size, Building heap by one by one insertion TC = O(n log₂n)

16) For a given array of n size, building heap using Build-Heap method TC = O(n) (EC)

Algo: - construct CBT from given array.
- For all non leaf node (starting from last)
for (i = ⌊n/2⌋ ; i >= 1 ; i--)
- do heapify down till all ele below are perfect.

* No. of swap = $\frac{n}{2}*0 + \frac{n}{2^2}*1 + \cdots + \frac{n}{2^{\log n}}(\log n - 1)$
= O(n/2)
* No. of comparisons = O(n/2 * 2) = O(n)

17) If all elements same
Heapsort TC = O(n)
∵ each deletion takes O(1) time.

18) In maxheap, delete (constant)th max TC = O(log n)
- deleting Kth (not constant) max = O(K log n)

19) In maxheap, to find Kth min
TC = O(kn) = O(n) (↳ constant)
Apply K passes of selection sort.

20) In maxheap, to find Kth max
⇒ K ele sighting, K-1 comparison
To find 10th max, we need to find 1 to 9th max.
Total comparison = $\sum^{K-1} n = \frac{K(K-1)}{2}$
= O(K²) If K is constant = O(1)

21) To convert given maxheap into minheap. TC = O(n)
Apply Buildheap method.

22) To delete ith position ele from Max/Min heap TC = O(logn)
O(1) BC
By random access goto ith position replace by last node. Perform heapify up & down.

NOTE: In a sorted array TC to find pair of no. whose sum is >1000. (or sum <1000)
TC = O(1)

23) Number of heap with n distinct keys.

$T(n) = {}^{n-1}C_K * T(K) * T(n-K-1)$

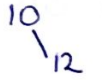
(K → No. of ele in left subtree; $n-K-1$ → to find K)

(First construct CBT with n-unlabeled nodes. ROOT is fix: min or max)

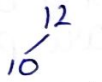
---

*Optimal BST: (Weighted BST) is a BST that minimizes search cost (no. of comparisons required to search for a given key)

Ex: i/p: 10, 12 freq = 34, 50



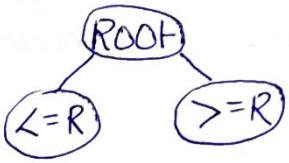
cost = 34*1 + 50*2
= 134



cost = 34*2 + 50*1
= 118
Min

*Optimal BST construction can be performed efficiently by using dynamic programming.

BST:

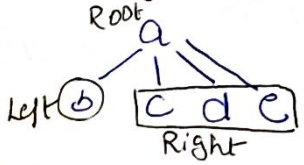


Handshaking lemma in Binary tree:-
#leaf = #internal node with 2 children + 1
$L = I_2 + 1$
$N = I + L$
$I = I_1 + I_2$
$\Rightarrow N = 2I_2 + I_1 + 1$

1) Inorder of BST is ascending order.
2) TO get sorted order from BST $\Rightarrow$ TC = O(n) EC
3) TO construct BST we require either preorder or postorder. DO its asc order to get inorder.
   TC = O(nlogn) + O(nlogn) = O(nlogn)
   (TO sort ↓, Binarysearch ↓ each root in inorder) EC
   - Better algo: using stack (save(min,max) for each node)
     TC = O(n) (EC) for every node constant no. of comparison with top of stack.
4) For a given Preorder or Postorder to create BST will take O(n) time (EC)
5) For a given preorder, directly start constructing BST from starting. For postorder, start from last.
6) Max in BST / Min in BST

Q: Identify correct sequence for searching 80 in BST.
9, 120, 12, 58, 60, 50
9 < 80 so all ele after 9 must be greater than 9 bcoz we will go right of 9 to search 80.
Its wrong seq, after 58 all must be > 58 but 50 < 58.

* N-ary Tree:



In: b a c d e
Pre: a b c d e
Post: b c d e a

7) NO. of BST possible with 'n' nodes = catalan no. = $\frac{2nC_n}{n+1}$

$BST(N) = \sum_{L=0}^{N-1} BST(L) * BST(R) = \frac{2nC_n}{n+1}$

$L + R = N - 1 \Rightarrow R = N - 1 - L$

8) NO. of unlabeled Binary trees with 'n' nodes = catalan no. = $\frac{2nC_n}{n+1}$
9) TO convert an unlabeled BT into BST there's only 1 unique way.
10) TO convert an unlabeled BT into binary tree = n! ways
    = n^n ways if repetition allowed
11) NO. of binary trees possible with 'n' nodes = no. of unlabeled BT * n!
    $= \frac{2nC_n}{n+1} * n!$
12) In BST
    - Max ele → Right most node
    - Min ele → Left most node
    - 2nd max ele → ROOT of max ele or max of left subtree of max ele
    - 2nd min ele → ROOT of min ele or min of right subtree of min ele.
13) TO find min/max ele in BST
    TC = O(n) WC, O(1) BC, O(logn) AC
14) TC to find smallest element:

	BC	WC	AC
BT	n	n	n
BST	1	n	logn
AVL	logn	logn	logn
Minheap	1	1	1
Maxheap	n	n	n ← have to compare all $\frac{n}{2}$ leaf nodes

15) TC to search an element:

	BC	WC	AC
BT	1	n	n
BST	1	n	logn
AVL	1	logn	logn
Minheap	1	n	n
Maxheap	1	n	n

16) TO confirm ele x not present:

	BC	WC	AC
BT	n	n	n
BST	1	n	logn
AVL	logn	logn	logn
minheap	1 (ele < root)	n	n
maxheap	1 (ele > root)	n	n

17) Insertion of a node in BST

$TC = \begin{cases} O(1) & BC \\ O(n) & WC \\ O(\log n) & AC \end{cases} \Rightarrow O(n), \Omega(1)$

18) TO create BST from n nodes

$TC = \begin{cases} O(n^2) & WC \\ O(n\log n) & AC, BC \end{cases}$

19) Since AVL is balanced BST, TO create AVL from n nodes

$TC = O(n\log n)$ EC

20) Inorder successor: min of right subtree
Inorder predecessor: max of left subtree
(found from tree, not inorder)

TO find succ/pred:

$TC = \begin{cases} O(1) & BC \\ O(n) & WC \\ O(\log n) & AC \end{cases}$ (we are traversing only 1 path)

21) Insertion also: search for data to be inserted, where null found insert.

22) Deletion also:
- leaf node ⇒ directly delete
- node with 1 child ⇒ delete & link 1 child.
- node with 2 child ⇒ replace with inorder successor or predecessor, delete suc/pred

$TC = \begin{cases} O(1) & BC \\ O(n) & WC \\ O(\log n) & AC \end{cases}$ (we are traversing only 1 path)

* In BST & AVL, insertion is done at NULL (it may or not be leaf node)

NOTE: If asked to delete node, delete that particular node. If asked to delete data, u can replace data and delete another node.

AVL Tree: self balancing tree
- Height balanced BST.
- AVL tree is BST, each node having Balancing factor −1, 0 or +1
- Height = logn (EC)
- Balancing factor = Height (Left subtree) − Height (Right subtree)

1) For max height with given no. of nodes find manually using relation in point (4)

Max Height	0	1	2	...
Min nodes	1	2	4	...

2) TO find min no of nodes required to form AVL tree of height 'H', Make left skewed tree of height 'H', then balance it & count no. of nodes.

3) Min Height of BinaryStree (BST or AVL) with 'n' nodes = $\lfloor \log_2 n \rfloor$

4) Min no. of nodes required to have AVL tree of height H

$MNN(H) = \begin{cases} 1 & ; H=0 \\ 2 & ; H=1 \\ MNN(H-1) + MNN(H-2) + 1 & ; H>1 \end{cases}$

* Rotation:
- LL } single rotation
- RR
- LR } Double rotation
- RL

} TC = O(1)

5) Insert an ele in AVL tree
TC = O(logn) EC

6) Insertion is always at NULL.

7) After insertion, while going back in path checking for problem,
Max 1 problem, min 0
Max 2 rotation, min 0

8) TO create AVL tree with n nodes
TC = O(nlogn) (EC)

9) In AVL tree,
Finding a node with 0 or 1 children (delete no leaf node)
⇒ TC = O(logn) EC
Finding a node with 2 children
⇒ TC = O(1) BC O(logn) WC AC

10) Deletion similar as in BST.

11) After deleting, going towards top & problem came solve it & continue going towards top.

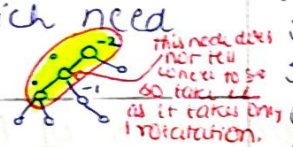
WC → $\log n$ problem in way.

BC → no problem

12) TC to delete an ele in AVL tree = $O(\log n)$ EC

13) In WC, max no of rotation after deleting an ele = 2 * max problem = $2\log n$ (assume note)

14) If at a time 2 problems come, go with one which need less rotation.



B & B+ Tree

1) These are search tree like BST & AVL.

2) Balanced trees.

3) If order = m

max no of children = m

min no of children = $\lceil \frac{m}{2} \rceil$

max no of keys = $m-1$

min no of keys = $\lceil \frac{m}{2} \rceil - 1$

4) Insertion is done at NULL place (leaf node).

5) All leaf nodes are at same level.

6) If there is overflow → split.

7) If split happens at root, height increases by 1.

8) TC to insert 1 ele = $O(m \log_m n)$ (EC)
Basis of order

9) B+ tree takes more space than B tree.

10) In B+ tree, all keys are present in leaf nodes so sequential search can also be performed.

11) While splitting leaf node maintain a copy at inorder suc/pred place in B+

12) Height of B/B+ tree = $O(\log_m n)$ Border

13) TO search an ele in B/B+ tree

$$TC = \begin{cases} O(m \log_m n) & WC \\ O(1) & BC \end{cases}$$

NOTE: If $m\log_m n$ not there in option, choose $\log_m n$.

14) Inorder is ascending order.

15) Deletion Algo:

If ele is part of leaf node, delete it. If underflows borrow from sibling or else combine.

If ele is not part of leaf node then replace it with inorder successor or predecessor then delete

* Array representation of Bt tree:

Root index: 1

Left child of node i = $2i$

Right " " " i = $2i+1$

Parent of node i = $\lfloor \frac{i}{2} \rfloor$

* complete Binary tree: nodes are inserted from left to right. (every level full except possibly last)

* Full Binary tree: every node has exactly 0/2 children.

* Perfect BT has every level full.

complete BT Full BT Perfect BT

Max no of nodes = $2^{h+1} - 1$ for height 'h'

* No of leaf nodes = No of internal nodes with 2 children + 1.

* For a binary tree of height h

Max no of nodes = $2^{h+1} - 1$

Min no " " = $h+1$

* On level L of binary tree, $n_{max} = 2^{L-1}$, $n_{min} = 1$

* For a complete BT of height h,
  Max no. of nodes = $2^{h+1}-1$
  Min no. of nodes = 2^h

* Total no. of BT with a given pre/post or inorder sequence
  = catalon no. = $\frac{^{2n}C_n}{n+1}$
  Given orders can be filled in 1 way in an unlabelled BT

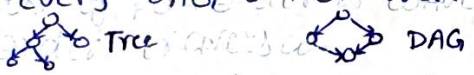
* weighted path length of BT
  = Σ weight$_i$ * length$_i$

* To represent BT of n nodes in array:
  Best case: array size required = n for complete BT.
  worst case: array size = 2^n-1 for Right skewed BT.

* In any Binary tree
  Σdegree = 2 * NO of edges
  NO of edges = n−1

* Descendent: वंशज Ancestor: पूर्वज

* Level order traversal (BFS) of BT
  TC = O(n) } using queue
  SC = O(n)
  TC = O(n²) ← for every level run for } using recursion
  Stack: SC = O(n)

* Tree is a Directed acyclic graph but every DAG is not tree.
  

** In BST to print all elements in range [a, b] → having k elements in range
  In WC, 2(k+n) extra nodes will be visited not n.
  Another way: Do inorder then find. TC = O(n) (EC)

* Full binary tree / strict BT / Proper BT / 2-Tree
  L = I + 1 , N = I + L
  ⇒ N = 2I + 1 = 2L − 1
  N is always odd.

* K-Tree: each node has 0 or k children.
  L = (K−1)I + 1
  N = KI + 1

* CBT is always height balanced.

Graph

1) Σ degree in Undirected graph = 2 * no. of edges

* BFS / DFS
  TC = O(V+E) adj list
  O(V²) adj matrix

* BFS / Level Order traversal:
  i) uses Queue of size V ⇒ SC = O(V)
  NOTE: Graph traversal is not unique but tree traversal is unique
  ii) For every BFT order, BFT tree exist

* connected acyclic graph is Tree. E = V−1

* Spanning tree: connected acyclic graph having all vertices.

* Applications of BFT: (TC of all same)
  1) TO check graph is connected or not
     TC = O(V+E) adj list
     O(V²) adj matrix
  2) TO check a vertex is reachable or not
     TC = O(V+E) adj list
     O(V²) adj matrix
  3) Identifying no. of connected component or no. of strongly connected component
     TC = O(V+E) adj list
     O(V²) adj matrix
  4) TO check given graph contains cycle or not
     Just find BFS tree → or in weighted graph will all weight equal
  5) Finding shortest path from a given vertex to every other vertex in an unweighted graph.
  6) TO check given graph is Bipartite or not. → add all vertices at a level in one grp vertices of next level in other grp

NOTE: By default consider adj list.

o Strongly connected component - Path exist from every node to every other node connected component with directions

* DFS:
  1) uses stack
     Stack = #levels of DFT
     TC = O(V+E) adj list
     O(V²) adj Matrix
     SC = O(V)
  2) NO. of PUSH = NO. of POP = V

o In BFS, NO. of insertion in queue = NO. of deletion = V

Max stack size: if no pop in blw ; Min stack size: if max no. of pop in blw

* Applications of DFT :

1) TO check given graph is connected or not.
2) TO find no. of connected component
NOTE: DFT can't be used to find shortest Path in unweighted graph.
3) TO detect cycle
4) TO check a vertex is reachable or not from a given vertex
5) TO check given directed graph is strongly connected or not.
→ apply DFT, reverse edges in graph; apply DFT
- Articulation point: Cut vertex
6) TO find all articulation point (BFT can't be used).

- For every connected graph, spanning tree possible.

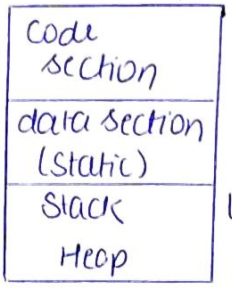
* Edge classification:

- Tree edge: Edges present in BFT/DFT tree or forest.
- Family: In tree, set of nodes from root to each leaf belong to a family
- Prior vertices are ancestors, next vertices are descended.
- Back edge: In a family edge going from a vertex to its ancestor.
- If back edge is present, there is cycle.
- Forward edge: In a family edge from a vertex to its descended.
- Cross edge: Neither forward nor backward edge.

1) DFT directed graph has 4 types of edges:
Tree edge, Back edge, Forward edge, Cross edge.
2) DFT undirected graph has 2 edges
Tree edge & Back edge.
3) BFT directed graph has 3 edges
Tree edge, Back edge & cross edge
4) BFT undirected graph has 2 edge
Tree edge & cross edge
5) In BFT undirected graph if cross edge is there, cycle is present.

- a → b means b is adjacent to a

#PROGRAMMING



1) Global variables allocated at compile time, local " " " run time.
2) Within a fu" 2 variables can't have same name.
3) Static scoping: (By default)
- done by compiler at compile time
- take global variable
4) Dynamic scoping:
- done by processor at run time
- take closest one in stack, if not there then take global variable.
If global val also not there → error
5) Every char can be stored as a int but not via versa. (char=1B, int=2B)
6) Global & static var is destroyed at end of program, having max lifetime.
7) Static variable: local variable in static area.
8) If static keyword is used with local variable, memory is allocated at compile time in static area.
9) Line containing static keyword is executed only once at compile time
10) a = f(x), first f(x) is executed then assigned to a.
11) In int 0, char '\0', float 0.0, pointer NULL.
12) Zero is false, all non zero is True
13) Operation is done blw operands of same data type.
14) Right to left associativity
?: , =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=
unary !, ~, ++, --, +, -, *, (type), sizeof