Data-Structures

Data Structures

10	20	30	40	50
0	1	2	3	4

* pointer access of array

*(arr+0) *(arr+2) → added with scale factor

10 20

Time Complexities

	Best (Ω)	Average (Θ)	Worst
Day2 1) Selection sort	~~n~~ n^2	n^2	n^2
2) Bubble sort	n	n^2	n^2
3) Insertion sort	n	n^2	n^2
4) heap sort	nlog(n)	nlog(n)	nlog(n)
5) Quick sort	nlog(n)	nlog(n)	n^2
6) merge sort	nlog(n)	nlog(n)	nlog(n).
7) lineal	O(1)	O(n)	O(n)
8) Binary	O(1)	log(n)	O(log n)

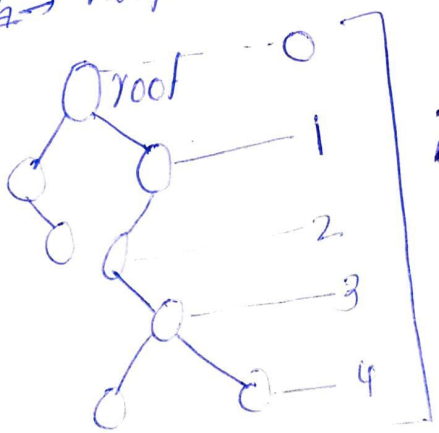
* Single loop → O(n), nested-loop → $O(n^2)$
* partitioning → O(logn), partitioning + iteration = nlog(n)
* BST → loorst → O(n).

Binary Tree:- every node other than leaf node has maximum of two childs

Leaves depth height:

leaf node → node with no childrene, depth: no of links from root to the node

Depth → height → no of links from its root to the furthest leaf



height = 4

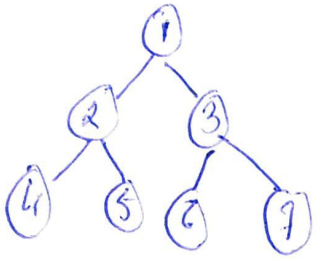
height of empty tree = -1

height = max(left, right) + 1

14

Breadth first search

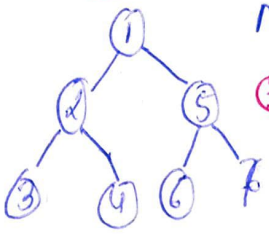
① Queue is used in traversal mechanism, level wise traversal is done



Depth - first Search

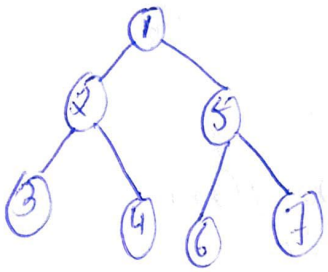
number represent traversal

② Stack is used in traversal mechanism



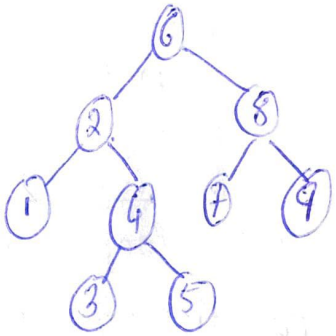
preorder - traversal

print Left Right



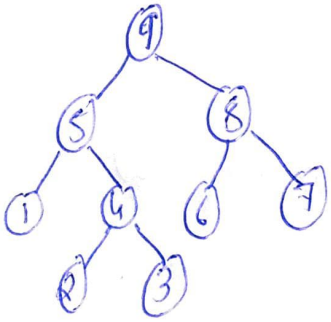
Inorder

Left print Right



post order traversal

Left Right print



① Full binary Tree.

ⓐ tree must have '0' or '2' childs

② perfect binary Tree.

ⓐ tree is full until height is ↑

③ Complete binary tree.

ⓐ tree is complete if not than childrens must be to the left as possible

Balanced Tree

left height - Right heigh = 0, 1, -1

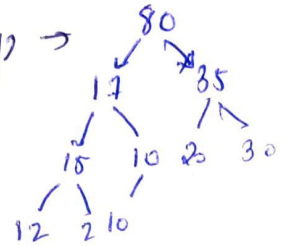
Data structure

heap → implementation of Complete binary tree into array.

- parent index = child/2
- left child = child × 2
- Right → parent × 2 + 1

Complete binary tree: all nodes are filled if partial then they must be to left

max heap →

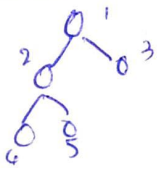
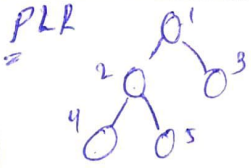


min heap → root node is smaller than both child

- used to implement priority Queue

push/pop → O(log(n))

PLR

BFS	DFS
Queue DS is used	Stack DS is used
	
1 2 3 4 5	1 2 4 5 3

Tree: nonlinear DS where data is represent in parent child relation

Graph: is non linear DS where Data is represented in the form of Node and Edges connect these Nodes

AVL Tree: it is self balancing Tree with balance factor -1, 0, 1, (BF = lH − RH)

operation → O(h) or O(logn).

Binary Search Tree: - binary tree with max two child having left child < parent Right ≥ parent

Tree traversal → travelling through tree all Nodes.

1. inorder → L (P) R
2. preorder → (P) L R
3. postorder → L R (P)

→ What is the max number of Node in a binary tree of height k?

$(2^{k+1}-1)$ $k >= 1$

Hashmap → No duplicate element are allowed if entered than it gets overwritten

Time Complexities:

get() put() in hashmap?

O(1)

what is requirement on key of Hashmap?

1. equals() & hashCode()
   - equals() → To retrieve from map
   - hashCode() → To insert into map

Array	Linked list
fixed size	Variable size
faster access (Random)	No random access. sequential access
used when searching is preff	used when insertion deletion is preffered

Circular Queue

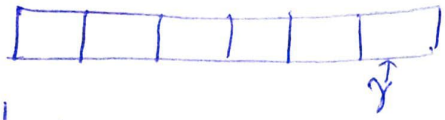
push: r = (r+1) % max

pop: f = (f+1) % max

peek:

```
next = (f+1) % max
return [next]
```

Queue empty/full



f=-1

r

(f==-1 && r==max-1)

r f

(f==r+1 && f!=-1) ||

Queue empty

(f==r+1 && f==-1)

push → r = (r+1) % max

pop → f = (f+1) % max

```
if(f==r){
    f=-1
    r=-1
}
```

---

peek → next = (f+1) % max

return [next]

---

comparisions → Binary Search

$(\log_2 N+1)$

---

① Sorting in liked list with minimum time complexity.

Muge sort.

Data Structure

order of growth →

→ constant < log(log n) < log n < $n^{1/3}$

• $n^{1/2} < n < n^2 < n^3 < n^4 < 2^n < n^n$

order of growth →

for → O(n)

for for → $O(n^2)$

single loop → O(n) | partitioning → O(log n)

nested loop → $O(n^2)$ | p+iteratn = (n log n)

linear search	binary search
best case → O(1)	worst case → O(log n)
avg → O(n)	avg → O(log n)
worst → O(n)	best → O(1)

binary search

```
m = (L+R)/2
if (key > mid)
  L = mid+1
else
  R = mid-1
```

Recursion

① prcen in terms of prcen

② base condition

selection sort (i<n-1)

8 3 4 5 6 8 — $O(n^2)$

i ⟷ ⟷ j

j = i+1

if (a[i] > a[j]) (i and j are con

swap

⊗ After first iteratn smallest element settled

Bubble sort

8 3 4 5 6 8

i →

j →

if (a[j] > a[j+1]) (only j)

⊗ first iteratn largest element geb settled

⊗ Improved bubble sort

```
for(i=0; i<n-1; i++)
  for(j=0; j<n-1-i; j++)
```

→ Improved Bubbl. Sort

$O(n^2)$

⊗ Improved bubble with flag

```
flag = 0
for(i=0; i<n-1; i++)
  for(j=0; j<n-i-1; j++)
```

→ check for comparision happening if don. set flag = 1

Insertion sort

$O(n^2)$

shift ahead to make place for temp

6 5 3 8 4

① we take element in temp according to i

i = start from 1 and j = i-1 to 0

⊗ Whatever a[i] > temp

a[j+1] = a[i].

Stack & Queue

Linear Queue

front → pop

rear → push

init

f = r = -1

Circular Queue

init → push

f = r = -1, r = (r+1)% max

Queue full

(f == -1 && r == max-1)

Circular Queue

init: f = r = -1

Queue empty

f == r && f == -1

Que full

if(f == r && f != -1)

size of array

Pop

f = (f+1)% size

if (f == r)

(f = r = -1)

push

f = f

r = (r+1)(size)

a[r] = val

f → points to last elem

⑧ Peek

next = (f+1)%, size
return arr[next]

array is from

f+1 to r

↳ f always point to last poped posit

Dequeue

pushBack() popBack()
pushFront() popFront()

Stack

init = Top = -1.

Expression evaluatn

a+b → infix
ab+ → postfix
+ab → prefix.

priorities

( )
$ power
* / %
+ -

infix to postfix (greater or equal)

① start from left to write.
② if operand → put in expres
② if operator → put in stack provided that no operator higher than is not present in stack or equal

③ if (→ put in stack
) → pop all and append till (find.

④ at last pop all and apend to expressn

infix to prefix

post fix evaluation

① go from left to Right
② if operand (1,2,3) put in stack
③ if operator → pop two operands
a = secondpop b = first pop
④ push back
⑤ pop final result

Infix to prefix

Left ← Right

① if operand
append to prefix expressio
② if operator
pop till only greater
③ push to stack
④ pop all and append to expres
⑤ Reverse expression

Singlly linked list

display()

trav = head
while(trav != null)

add at position

for → trav = at 2n
before i = 2
trav points to 3

① ② ③ ④ ⑤ (i < pos-1)
pos = 4 ↑
We need to go to ③
for that i = should go till 2

delete at position

① ② ③ ④ ⑤ ⑥ i lags one posi than trav

del at ③

```
for(i=1; i<pos; i++)
{ prev= trav;
  trav= trav.next;
}
```

⊛ add at pos for del is same as that of add

for add → i < pos-1

del → i < pos

Hash table

↳ associative Data structure

load factor = $\frac{\text{no of entries}}{\text{no of slots}}$

Collision handling methods

① open addressing & chaining

② chaining

① Open addressing :-

used when load factor <= 1

↳ linear probing, quadratic probing.

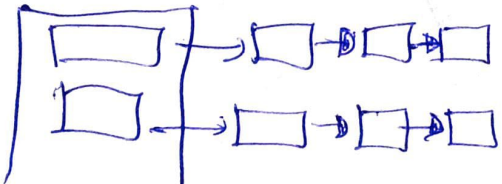
$h(k) = k\%SD$

$rh(k) = (prev+1)\%SD$

Chaining :-

⊛ linked list used to store data

⊛ list of linked list is stored in array list



Tree Definition

→ Tree is hierarchial data structure

~~Node : a~~

null Tree → Tree with no node

leaf-Node → Terminal node of a tree & does not have any node connected To it

Degree of node :- no of sub trees of node

Degree of tree :- max degree of node in a tree

ancenstors → all nodes from root to that node

Descendants → all nodes reachable from that node

level of node :- level of parent + 1

level of root is '1'

Depth of node :- no of nodes from root to node

root Depth = 0

Level = depth + 1

Height of node = no of nodes from the node to its deepest leaf

H = max of child height + 1

height of empty / null node = -1

Binary tree → each node with max 2 child or tree with degree 2

BST → nodes are arranged according to values

left child < parent / Right child ≥ parent

Inorder = LPR

Pre-order → PLR

post order → LRP

BST preorder (PLR) non rec

↳ stack is used
↳ start with root
↳ put right in stack so that its not lost.
↳ trav moves till it reaches null
↳ then pop new element

BST search Inorder LPR

↳ trav = root;
↳ while(trav != null || !s.isEmpty)
↳ put on stack that may be lost

DFS ⇒ Inorder traversal (pLR)

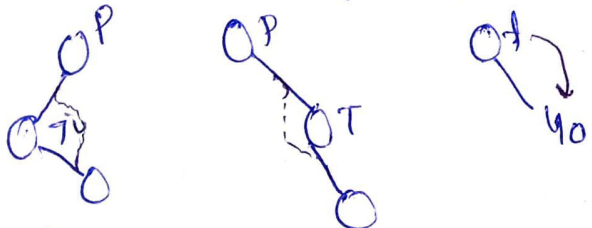
BFS ⇒ is independant level wise traversal

DFS if to → stack
① match key with all values
② if found return
③ put right on stack go to left
④ pop Right put its

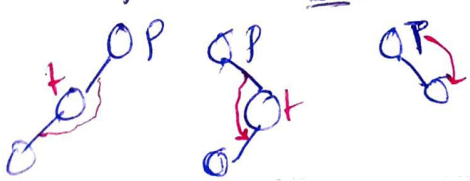
BFS → Queue

Delete node

Case when temp.left == null

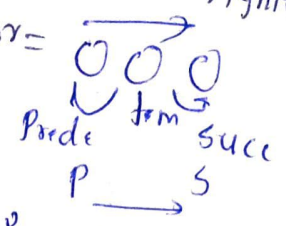


temp.right == null



Case 3 temp.left != null && temp.right = nul

Inorder successor = (LPR)



Succ Parent = temp
= temp.right
While (Succ.left != null) {
Parent = Succ;
Succ = Succ.left
}
temp.data = Succ.data
parent.left = Succ.right)

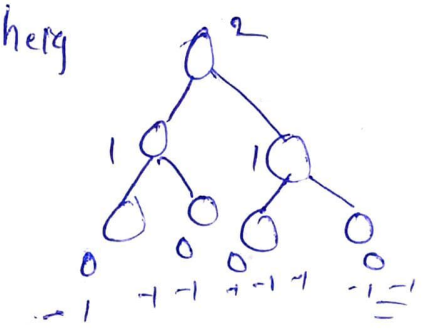
Skewed binary tree time Complexity
O(h) → height

Balanced BST

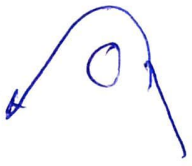
↳ for speed search height should be minimum as possible
↳ If node in BST are arranged so that its height is kep as less as possibl.

Balance factor = Hight of left subtree − Hight of Right subtree

BF = −1 0 1



Right Rotation
left



Right =

AVL Tree

- is self balancing Binary Search Tree
- → Balance factor ≤ 1 → $-1, 0, 1$
- most of Tree operation $O(\log n)$

Red & black Tree

1. Root is always black
2. parent should be different than child
- most of insert/delete → $O(\log n)$

Strict binary tree

$2^h \times 2 + 1$

↳ each node exactly 2 child nodes in non-leaf node (all).

perfect binary Tree

a tree which is full for the given height or contains max node

No of nodes = $2^{(h+1)} - 1$

no of leaf nodes = 2^h

non-leaf node = $2^h - 1$

Complete binary Tree

all level are filled except last which has max nodes to left as possible

- heap is array impl of Complete binary tree

$p = \frac{\text{child index}}{2}$, Left = $2p$, Right = $2p+1$

max heap

parent is greater than both childs

min heap

parent is smaller than both the childs

kth largest element

1. delete max element k' times

$O(k \log n)$

heap Sort

1. make max heap
2. delete max from heap and add to end of array
3. repeat till heap is empty

Graph

- connected graph → some path
  - → from vertex path exist for every other vertex
  - ↳ can traverse entire graph starting from any vertex

↳ Complete graph

each vertex of graph is adjacent to every other vertex

Undirected → Edges = $\frac{n(n-1)}{2}$

Directed graph = Edge = $n(n-1)$

Spanning Tree (weighted)

→ is connected ~~graph~~ sub-graph of given graph that contains all vertices and subset of edges $(V-1)$

- In weighted ~~sub~~graph spanning tree who has minimum weight (sum of edges) such Tree is MST
- diagonal is always symmetric across diagonal

Graph traversal

DFS algorithm

- stack is used
1. choose start vertex
2. push on stack & mark it.
3. pop the vertex print it
4. put all adjacent nodes on stack & mark them (non marked)
5. 4 are repeated till stack is empty

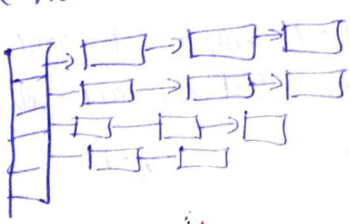
BFS traversal (Queue is used)

1. put start vertex in Queue mark it
2. pop it print & push all adjacent in Queue & mark it
3. pop it and print it
- repeat above steps till Queue is not empty

DFS and BFS spanning tree are same as above

Adjacency list

- Each vertex holds → its all neighbour vertex



Greedy approch

- building solution step by step (piece by piece)
- in each iteration such a piece is chosen that is most obvious & immediate benifit

greedy ○ 10 → choose complete
greedy ○ 20 → choose complete } 50
○ 30 → 1/3

Union find algorithm

it is used for detecting cycles

Krushkal MST greedy

1. sort all edges in ascending order of thier weigth
2. pick smallest check if forms cycle if yes discard else include
3. repeate above til all (V-1) nodes are not formed

Time Complex
sort Edges → O(E log E)
pick edges → O(E) (E edge)
Union find → O(log V)

Prim MST Greedy

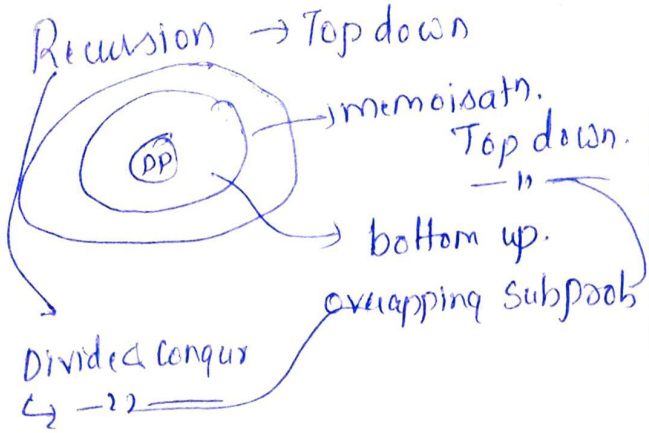
- set is used to keep track of vertices included in MST
- init each parent to '-1'
- init key to '∞', start key 0

Dikstra MST

memoisation → top down app
is used to speed up programs
- we store calculated result in memory
- store result in map / array

DP → not greedy → bottom-up
is used when
1. Overlapping sub problem
2. optimal sub-structure

- to solve a problem we need to solve sub problem
- used array to store state



Rodd cutting → DP

Bellman ford → Dikstra dont work for -ve' edges
we use Bellman ford

Warshal floyed →

Quick Sort (Day 12 → 3)

① selet pivot → left most

while (i<j) {
from left find element > pivot
from right element ≤ pivot.
if (i<j)
swap(a[i], a[j])

a[i] ⟷ pivot

Woorst case →
partitioning → O(n)
compar → O(n)
O(n²)

avl/best
logn
n
nlogn

Merge
→ merging
→ Partitio
O(nlogn)