Infix to postfix conversion

first operand – next operator

1) If the incoming symbol is operand, you need to print directly.

2) If the stack is empty, you can push the operator into stack based on its arrival.

3) If the incoming symbol (left parenthesis, push into stack

4) If the incoming symbol is Right parenthesis), perform the pop operation until you find the left paranthesis (then discard both parenthesis.

5) If the incoming symbol is lowest presidency than top of the stack we need to perform pop operation and again check with the next operator.

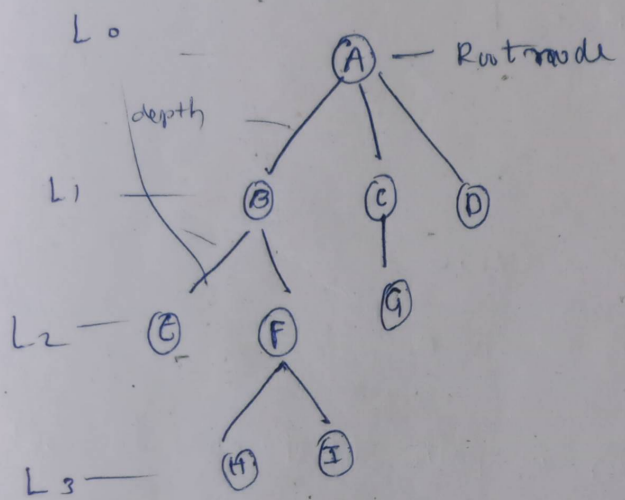
6) If the incoming symbol is highest presidency than top of the stack, we need to push operation.

7) If the operator has same presidency, we need to check for association. L-R ⇒ pop(), R→L : push()

Trees

Tree is a non-linear data structure. Collection of nodes connected with edges and there is a Hirarical relationship among nodes.

→ The first node of tree is "root node")

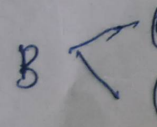


Predecessor: or parent:

Imediate predecessor of any node is known as parent.

parent: A, B, C, F

Child node: The Immediate successor of any node is known as child node.



leaf node: The nodes which are not having children known as leaf node.

E, H, I, G, D

non-leaf node: The nodes which have atleast one node children known as non-leaf node.

A, B, C, F

Ancestor:

G ⟶ A, C

F ⟶ B, A

predecessor on path from that node to root node

Decendent node: Successor node on the path from particular node to leaf node

B ⟶ F, H, I, G

Siblings: Children node of same parent.

A ⟶ B, C, D

Sub tree: A node with all its decendent nodes

Degree: Children of particular node

degree (A) — 3

B ⟶ 2

F ⟶ 2

C ⟶ 1

degree of tree = maximum degree of overall node
= 3 ↳ particular node

depth of a node: length of a path from root node to that node.

depth of E = 2 — Edges

I = 3

Height of a node: no. of edges in the path (longest path) from that node to leaf node

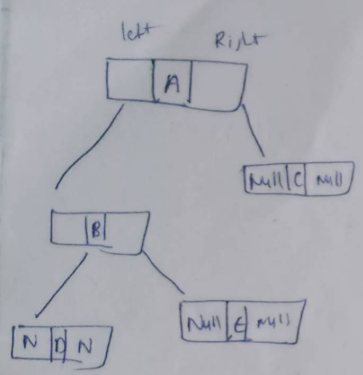
B — 2

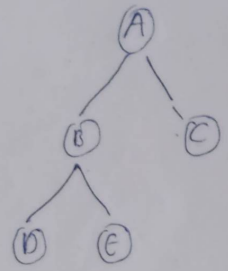
Height of Tree = height of root node

Level of node

height of ~~tree~~ = level of node

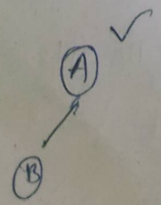
Depth of particular node = its level

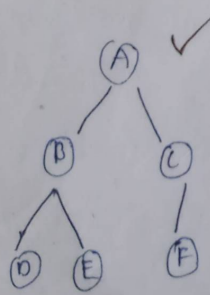




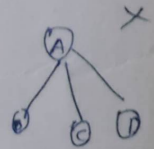
Binary tree

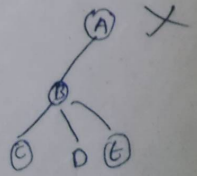
In Binary tree each node should have atmost two children.



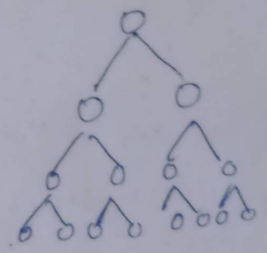


Binary





L-0 → 1
L-1 → 2
L-2 → 4
L-3 → 8



2^n

maximum no. of nodes in a level i = 2^i

h = 3

Max = $2^{h+1} - 1$

$2^{3+1} - 1$

$2^4 - 1 = 15$

minimum no of nodes in a level

min = $h+1$

h = 2

$2+1 = 3$

Types of binary tree

Fully binary tree : It this each node should have exactly two children except leaf nodes.

(or)

Each node should have 0 (or) two children

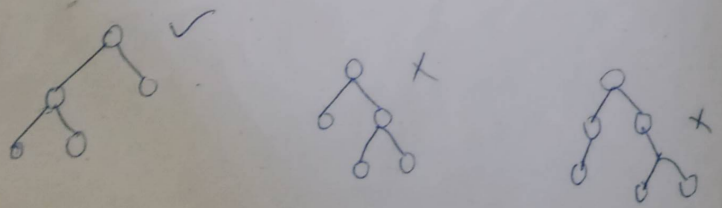
Types of Binary tree :

- fully binary tree
- complete B.T
- perfect B.T
- Degenerate B.T

max = $2^{n+1} - 1$

min = $2n + 1$

Complete binary tree : In this complete binary tree the all the levels should be filled completly except last level it should be filled from left to right



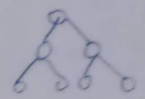
max = $2^{n+1} - 1$

min = 2^n

perfect binary tree : All internals nodes should have two children except leaf nodes

all the nodes should be in same level.

✗ P.B.T
✓ F.B.T
✓ C.B.T

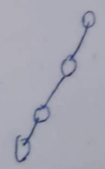


✓ P.B.T
✓ C.B.T
✓ F.B.T

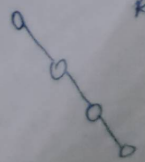
→ If a tree is perfect binary tree than it is complete & full binary tree.

Degerate binary tree : All the internal node should have one child in this it is divided into left skewed binary tree & Right skewed binary tree.

left skewed B.T : It have only left child.



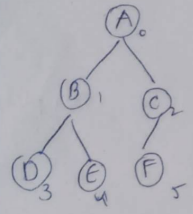
Right skewed B.T : It have only Right child.



&& — logical And (true if both operands are true)

|| (true if either one operand is true)

degenerate: Combinations of left & Right



It is for only complete binary tree

left child = $(2*i)+1$

Right child = $(2*i)+2$

~~Parent~~ = $\frac{(i-1)}{2}$

$i=2$, left child = $(2\times2)+1 = 5$

Right child $(2\times2)+2 = 6$ (No right child.

parent = $\left(\frac{2-1}{2}\right) = \frac{1}{2} = 0$ (A)

If index start from (1)

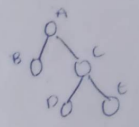
A	B	C	D	E	F
1	2	3	4	5	6

left child = $(2*i)$

Right child = $(2*i)+1$

Parent = $\frac{i}{2}$

if it is not complete B T



	A	B	C	-	-	D	E
0	1	2	3	4	5	6	

	A	B	C	-	-	D	E
1	2	3	4	5	6	7	

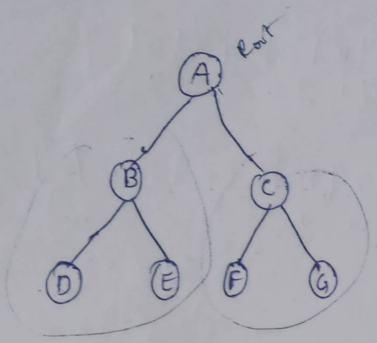
$(2*i)+1$

$(2*i)+2$

$\left(\frac{i-1}{2}\right)$

$\left(\frac{2-i}{2}\right)$:

B.tree traversals



1) Inorder → left, root, right

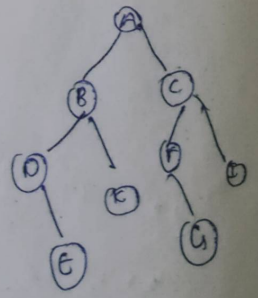
2) preorder → root, left, right

3) post order → left, right, root

1) D, B, E, A, F, C, G

2) A, B, D, E, C, F, G

3) D, E, B, F, G, C, A



1) D, E, B, K, A, F, G, C, I

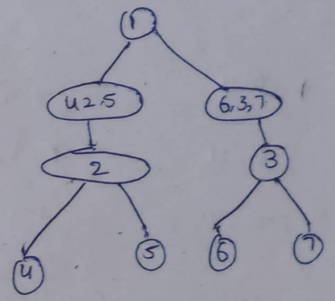
2) A, B, D, E, K, C, F, G, I

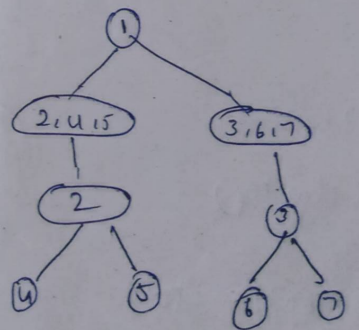
3) E, D, K, B, G, F, I, C, A

1) Inorder — 4, 2, 5, 1, 6, 3, 7

2) pre order — 1, 2, 4, 5, 3, 6, 7

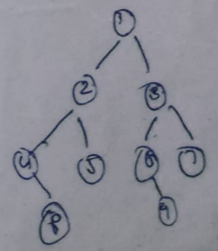
3) ?





from pre-order

1, 2, 4, 5, 3, 6, 7
root left Right



pre - 1, 2, 4, 8, 5, 3, 6, 9

in order 5

5, 2,

heap

heap is also a tree data structure. in this heap is divided into two types

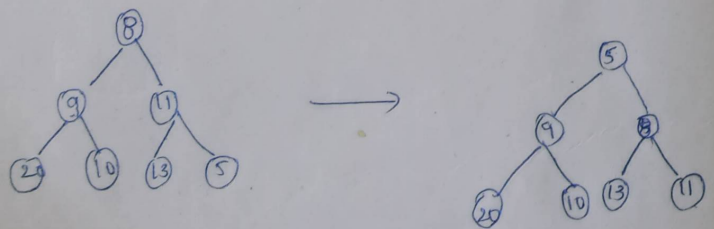
① min heap and ② max heap

① Min heap

The element at root must be less than or equal to its children. and recursively true for all subtrees.

→ It must be complete binary tree

8	9	11	20	10	13	5

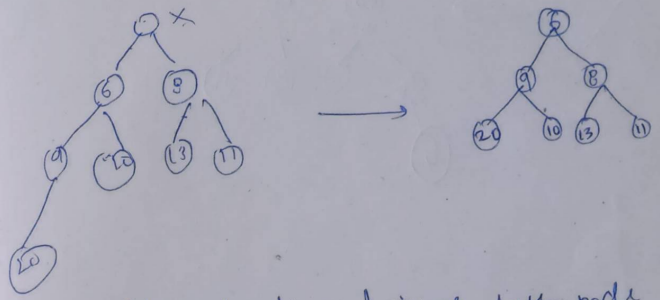


Insertion — bottom to top

delete — top to bottom

5	9	8	20	10	13	11

Deletion



delete root node and insert last node

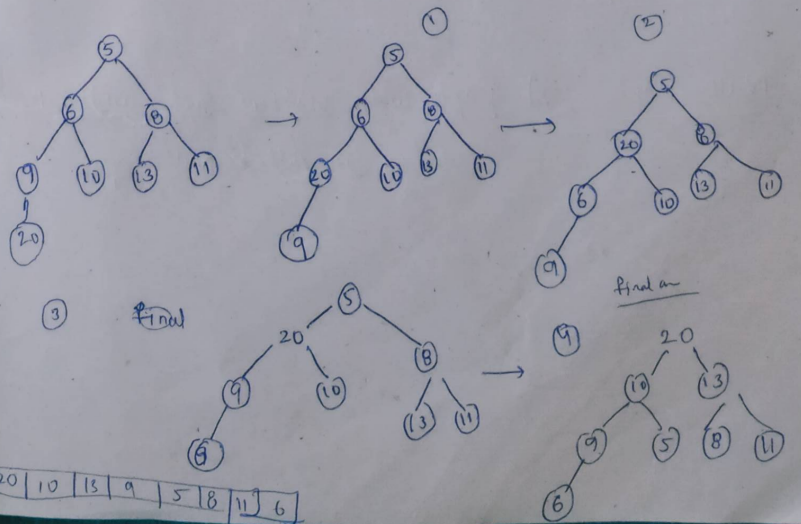
finally apply heapify algorithm

② max heap

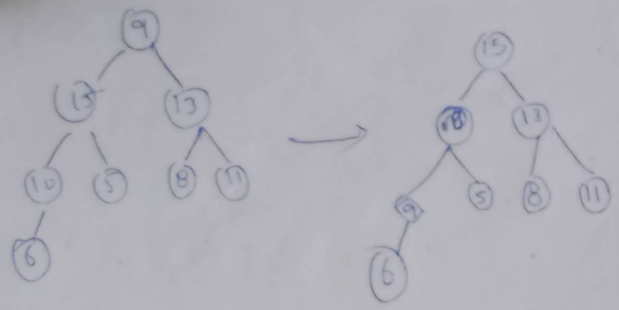
The element at root must be greater than or equal to its children and recursively true for all subtrees.

Ex

5	6	8	9	10	13	11	20



20	10	13	9	5	8	11	6



Graphs

Graph is a non-linear data structure. It consists of set of Edges and vertices.

→ If a graph G consists of E edges V (vertices) we can represent it as

G (E, V)

Types of graphs

① Null graph : A graph with vertices but without edges is called null graph.

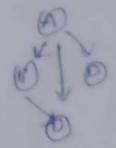
O

O O

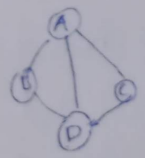
② Trivial graph A graph with only one vertice is known as Trivial graph

→ O

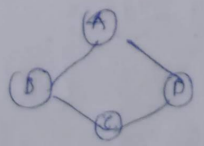
③ directed : There is a particular direction from one node to node.



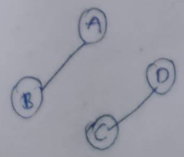
④ Undirected : The graph without direction



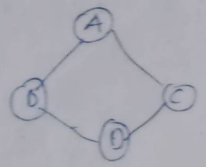
⑤ Connected graph : From one node you can visit to any other node is called Connected graph



⑥ Disconnected graph atleast one node is not reachable from any node.

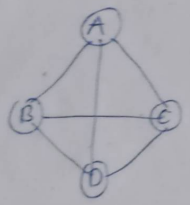


(7) regular graph: In regular graph degree of each vertex is equal.



(8) Complete graph: ~~In complete graph degree~~

A graph with n vertices, if there is $(n-1)$ degree for each vertex is called complete graph.



no of vertices = 4

$n-1 = 3$

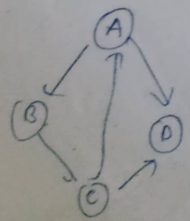
degree (A) = 3

degree B = 3

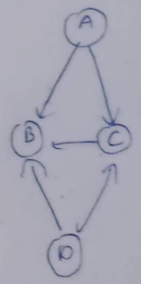
C = 3

D = 3

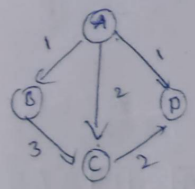
(9) cyclic graph: A graph with atleast one cycle is known as cyclic graph.



(10) Acyclic graph: A graph without a cycle is known as acyclic graph



(11) weighted graph: A graph with weights



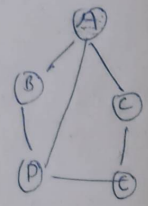
A → C = 2

A → B → C = 4

BFS - Breadth first search

→ BFS follows Queue = FIFO

1. Select source node (any node)
2. Insert source node into queue
3. Mark it as visited
4. pop out visited node
5. find adjacent node



display A C D B E

queue: A C D B E

visited

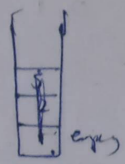
A C D B C

A B A C D B E

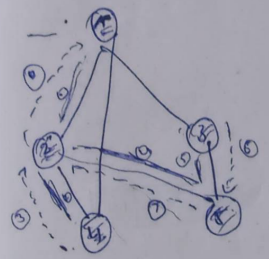
DFS Depth first search

→ It follows stack - LIFO

1. Select any node as source node
2. Insert source node into the stack
3. mark it as visited.
4. find adjacent node with any one direction
5. If there is no unvisited node.
6. pop out
7. and go back track.
8. check any unvisited nodes
9. "There is unvisited node pop out"



visited 1 2 4 5 3



1 2
1 3
1 4
2 5
3 5
2 4

4 3 5 2 1

1, 2, 4, 5, 3

Graph representation

→ Adjacency matrix
→ Adjacency list

(1) In the Adjacency matrix only 0 and 1 taken

0 indicates there is no edge

1 indicates there is an edge

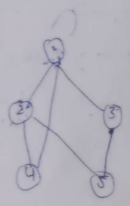
bing values consider

&& — logical And (true if both the operands are true)

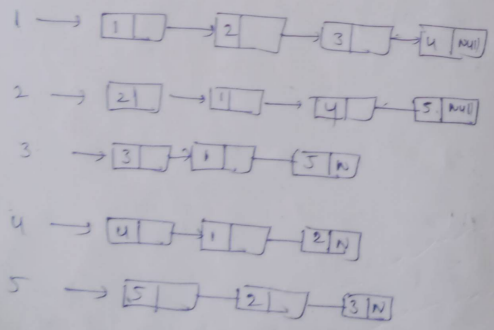
|| — logical OR (true if either one operand is true)

Adjency matrix

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	0	1
4	1	1	0	0	0
5	0	1	1	0	0



Adjency list



Hashing

Hashing is a technique or process which is used to map key with the values in a Hash table

→ with the help of Hashing we can efficiently access the elements from the hash table

→ The efficiency depends upon type of hash function is used.

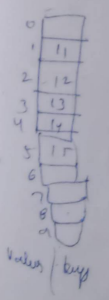
H(key) = key mod size

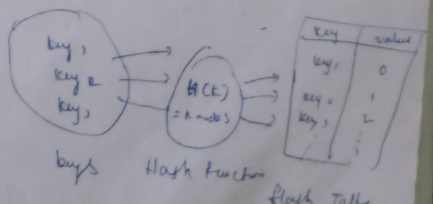
ex list = { 11, 12, 13, 14, 15 }

Size = 10

H(11) = 11 mode 10 —— 1

H(12) = 12 mode 10 —— 2





Hashing terms

→ Hash table
→ Hash function
→ key

→ key is a integer or string which will be given as input to the Hash function.

→ Hash function is a mathematical function that uses key as a input and it will generate value or index.

→ where we can store keys based on the value generated by the hash function

→ Hash table : Hash table is type of data structure which is used to map the keys with value

Types of Hash function

1. Division method — mostly used.
2. mid square method.
3. multiplication method.
4. digit folding method

① Division method

$H(key) = key \bmod m$

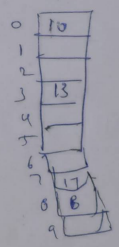
List = {10, 11, 13, 17}, m = 10

$H(10) = 10 \bmod 10 = 0$

$H(11) = 1$

$H(13) = 3$.

$H(17) = 17$

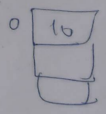


② mid. square method

list = {10, 11, 13}

$h(k) = k^2$

$= 100 \bmod 0$



you need to find a square of a key, second you need to find middle or digits of a square number

$K^2 = \{100, 121, 169\}$

Long Exercise Book

(3) multiplication method

$H(K) = \text{floor}(m\,(KA \bmod 1))$

size, key, 0.618033

618033

$= \text{floor}(10\,(123 \times 0.618033 \bmod 1))$

(4) Digit folding method

$H(k) = H(k_1, k_2 \ldots k_n)$

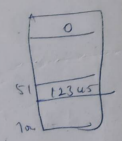
$H = (\underset{k_1}{12}\underset{k_2}{34}\underset{k_3}{5})$

$S = k_1 + k_2 + \ldots + k_n$

$S = (12+34+5)$

$S = 51$

$H(12345) = 51$



Collision

Collision is problem when two or more keys are mapped with same location in hash table.

eg

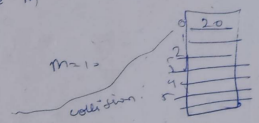
$H(k) = k \bmod m$

$k = \{20, 10, 11, 13\}$

$m = 10$

$H(20) = 0$

$H(10) = 0$

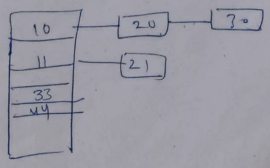


Collision resolving techniques

1. Separate chaining – (open hashing)
2. Open addressing – (closed hashing)
   - linear probing – $(u+i) \% m$
   - quadratic probing – $(u+i^2) \% m$
   - double hashing – $(u + v * i)$

Size >= no. of elements

→ Separate chaining



(1) Linear probing $(u+i)\%m$

Algorithm

1) $h(k) = k\%m$

2) we have to search for the
If you have free space in hash table - then
we have to store key value be assigned
into hash table

3) else if you not having empty space, then
you need to find empty place based
on the formula $(u+i)\%m$

u - position occurs / collision occurs
i - index 0 to m-1

(4) you need to Repeat 2nd and 3rd steps upto
to size is m.

probes - how many times search is done

$(u+i)\%m$

Ex 5, 4, 10, 2, 7, 9

$m=10$, $h(k) = 2k+1$

$h(5) = (2\times5+1)\%10 = 1$

$h(4) = 9\%10 = 9$

$h(10) = 11\%10 = 1$

$h(2) = 5\%10 = 5$

$h(7) = 15\%10 = 5$

$h(9) = 19\%10 = 9$

key	location	probs
5	1	1
4	9	1
10	1 coll.	2
2	5 coll	1
7	5 coll	2
9	9 coll	2

0	9
1	5
2	10
3	
4	
5	2
6	7
7	
8	
9	4

10
$(1+0)\%10 = 1$
$(1+1)\%10 = 2$

$(5+0)\%10 = 5$
$(5+1)\%10 = 6$

$(9+0)\%10 = 9$
$(9+1)\%10 = 0$

NOTE = ki stored at $(u+i)\%m$
k stored at

The Element ki is will be stored at first
$(u+i)\%m$ in hashtable.

Long Exercise Book

② quadratic probing

3, 6, 9, 11, 4, 7, 1

m = 10 , h(k) = (2k+3)

h(3) = (2×3+3) % 10 = 9

h(6) = 15 % 10 = 5

h(9) = 21 % 10 = 1

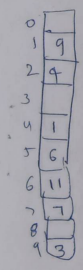
h(11) = 25 % 10 = 5

h(4) = 11 % 10 = 1

h(7) = 17 % 10 = 7

h(1) = 5 % 10 = 5

key	location	probes
3	9	1
6	5	1
9	1	1
11	5 coll	2
4	1 coll	2
7	7	1
1	5 coll	4



-, 9, 4, -, 1, 6, 11, 7, -, 3

4+i²

(5+0) % 10 = 5

(5+1) % 10 = 6

(1+0) % 10 = 1

(1+1) % 10 = 2

(5+0) % 10 = 5

(5+1) % 10 = 6

(5+4) % 10 = 9

(5+9) % 10 = 4

Double Hashing

3, 9, 4, 7, 13, 12

h₁(k) = 2k+3 , h₂(k) = 3k+1 , M = 10

3 — (2×3+3) % 10 = 9

9 — 21 % 10 = 1

4 = 11 % 10 = 1

7 = 17 % 10 = 7

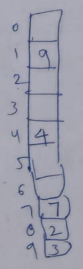
13 = 29 % 10 = 9

2 = 7 % 10 = 7

key	location	probes
3	9	1
9	1	1
4	v = (3×4+1) % 10 = 3	2
7	7	1
2	2	4

v = 2

4 + v×i



3k+1 =

3(13)+1 =

40 % 10 = 0

v = 9+

9 + 0×0 = 9

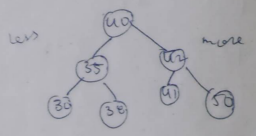
Long Exercise Book

BST – It follows inorder

Binary search Tree is a non-linear data structure in this left side element must be less than its parent and right side element must be greater than its parent. These conditions apply for its all sub trees recursively.

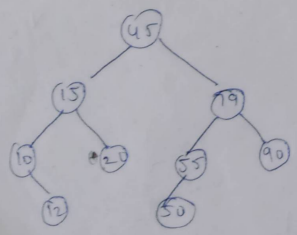
1. Searching
2. Insertion
3. deletion
4. Traversals

left → maximum
Right → minimum



45, 15, 79, 90, 10, 55, 12, 20, 50

45 →
15 →
79 →
90 →
10 →
55 →
12 →
20 →
50 →



① Algorithm — Intution

struct node* struct node

1) insert (root, key)

```
if(root == NULL)
    return root;
```

2) if (root → data < key)

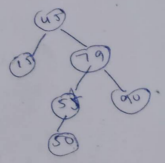
```
root → right = insert (root → right, key)
```

3) else if (root → data > key)

```
root → left = insert (root → left, key)
```

4) return root ;

55



② Search

```
Search (root, key)

① if (root → data == key)
③     return root;
  else if (root → data < key)
      return search (root → right, key)
⑤ else
      return search (root → left, key)
⑥ return root;
```

deletion

1) The node to be deleted is a leaf node (without having children)

2) The node to be deleted is having one child,

3) The node to be deleted is having two children.

i) replace its child to node.
ii) remove its child.

```
if (root->left == NULL)
{
  temp = root->right;
  root->data = temp->data;
  free(temp);
}

if (root->right == NULL)
{
  temp = root->left;
  root->data = temp->data;
  free(temp);
}
```

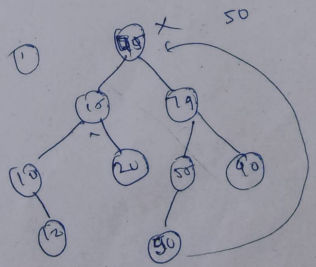
```
if (root->left == NULL && root->right == NULL)
{
  free(root);
  return NULL;
}
```

(3)

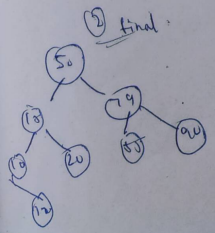
i) find out inorder successor
ii) Replace the node with inorder successor
iii) Remove inorder successor.

Inorder successor is the minimum element in a right sub tree of deleted node.

Inorder predessor is the maximum element in a left sub tree of deleted node.



(2) find.



Long Exercise Book


```
int min = Inordermin(root->right)
    root->data = min;
    delete(root, min);

Inordermin(struct node *root)
{
    temp = root->right;
    while(temp->left != NULL)
    {
        temp = temp->left;
    }
        return temp->data;
}
```

Linear probing

AVL

1. LR
2. RR
3. Left-Right
4. Right-left