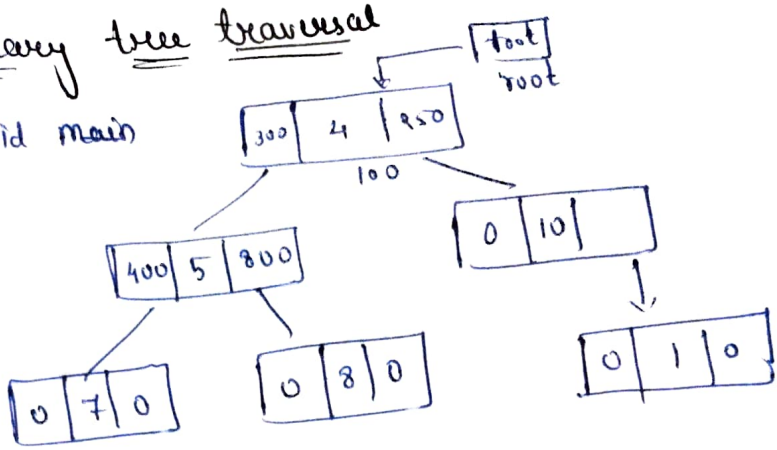
Binary tree traversal

Void main



Inorder: left root right

7, 5 8 4 10 1

Preorder: root left right

4, 5, 7 8, 10, 1

Postorder: left right root

7, 8, 5, 1, 10, 4

```
Void main(){
   struct node *root;
   printf("Preorder is: \n");
   preorder (root);
   Inorder (root);
   Postorder (root);
}

Void Preorder (structnode * root);
   if (root == '0'){
      return ;}
   }
   else
      printf("%d", root→data);
      preorder (root→left);
      preorder (root→right);
   }
   void fn
```

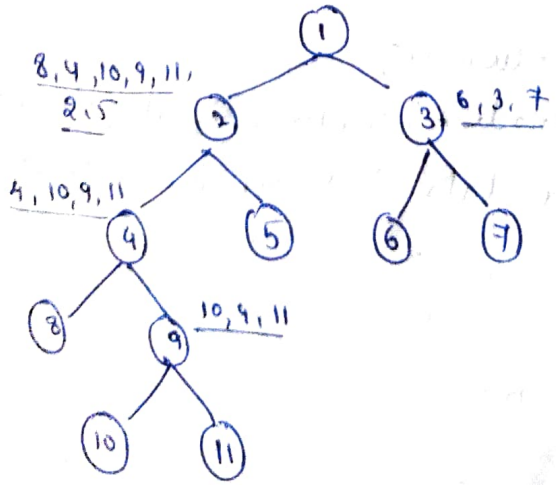
```
Void inorder ( structnode * root) {
    if (root == 0) {
        return 0;
    }
    else {
        inorder (root -> left);
        printf (" %d", root -> data);
        inorder ( root -> right);
    }
```

```
Void postorder ( structnode * root) {
    if (root = 0) {
        return;
    }
    else {
        postorder ( root -> left);
        postorder ( root -> right);
        printf (" %d", root -> data);
    }
```

Construct a B.T from pre order & Inorder

preorder :- 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7

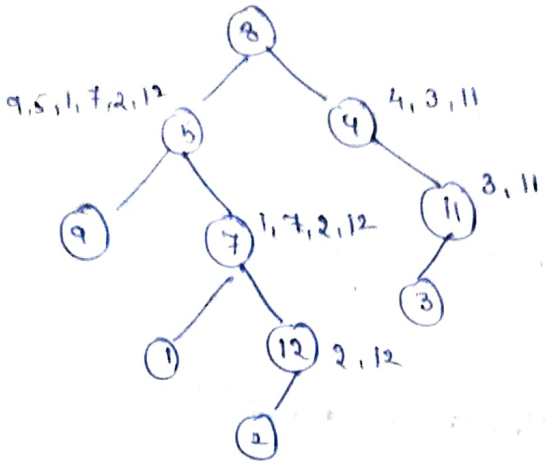
Inorder :- 8, 4, 10, 9, 11, 2, 5, (1), 6, 3, 7



Construct a binary tree from Postorder & Inorder

Postorder : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

Inorder : 9, 5, 1, 7, 2, 12, 8, 4, 3, 11



Implementation of B.T

```
struct node {
    int data;
    struct node * left, * right;
};
struct node *create()
void main(){
    struct node * root;
     root = NULL;
     root = Create();
}
struct node * Create() {
    int x;
    struct node *newnode;
newnode = (structnode *) malloc(sizeof (struct node));
printf (" Enter data (-1 for no node):");
scanf ("%d", &x);
if (x==-1)
    return 0;
}
else {
 newnode -> data = x;
 printf(" Enter left child of %d", x);
  newnode -> left = create();
```

```
struct node * create() {
    int i;
  new = (struct node *) malloc (size of (struct node));
    new → left = NULL;
    new → right = NULL;
    printf (" enter the data ");
    scanf (" %d", &i);
    if (i == -1)
      return NULL;
    new → data = i;
    printf (" enter the data for left child %d", i);
    new → left = create();
    new → right = create();
      return new;
}
```

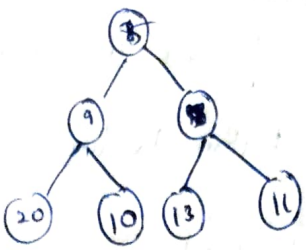
Heap : Heap is also a tree data structure in this we heap is divided into 2 types. It is complete binary

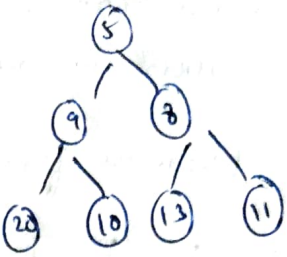
(i) min heap

(ii) max heap

(i) **Min heap**: The element at root must less than or or equal to its children. It is recursively true for all its sub trees.

8	9	11	20	10	13	5	



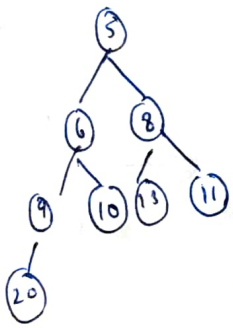


8	9	11	20	10	13	5	6

before inserting

5	6	8	9	10	13	11	20

After inserting



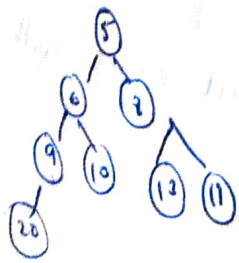
⇒ In Insertion we need to traverse from bottom to top

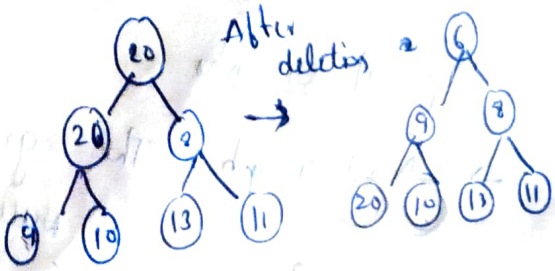
⇒ In deletion we need to traverse from top to bottom

⇒ while deleting a node we delete root node and insert last level node at root & heapify algorithm (checking heap property).

5	6	8	9	10	13	11	20

Ex:

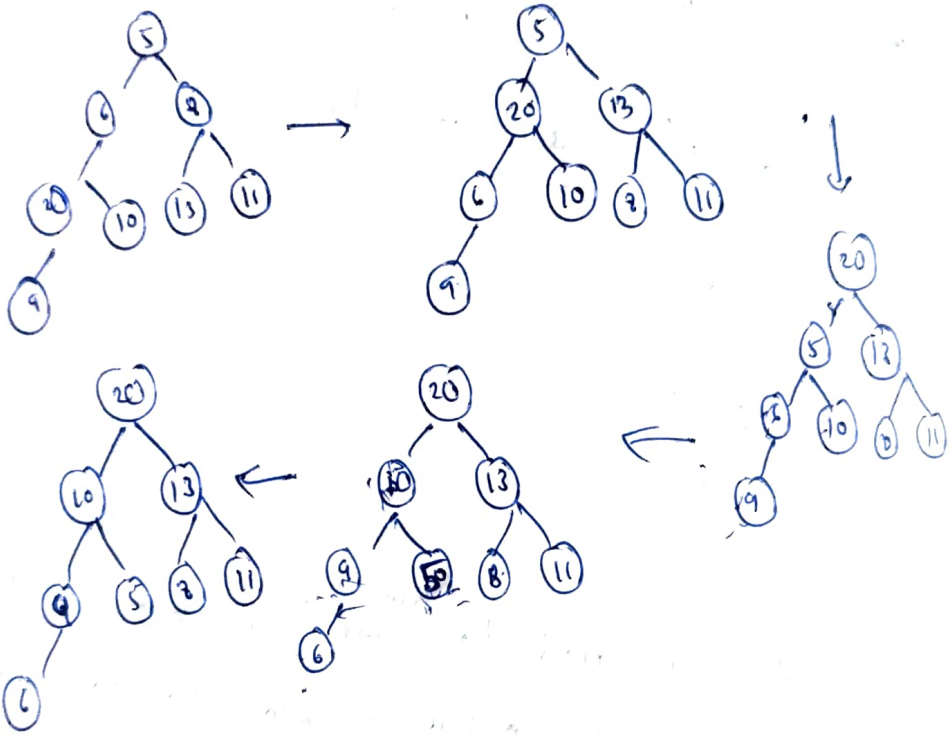




Max heap : Max heap : The element at root must greater than or equal to its children. It is recursively true for all its sub trees.

Insertion : (traverse from bottom to top)

5	6	8	9	10	13	11	20



Graph: Graph is a non-linear data structure it consists of set of vertices & edges. If a graph G consists of E edges, V vertices we can represent the graph as

$$G(E, V)$$

Types of Graph

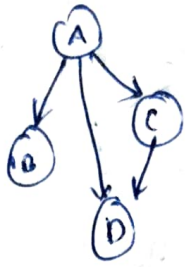
→ Null graph : The graph without edges, with vertices is null graph

Eg: (A) (B) (C)

→ Trivial graph: A graph with only 1 Vertex is known as trivial graph.

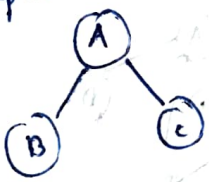
→ Directed graph (DI-graph): A graph with a particular direction from one ~~node~~ vertex to another vertex is directed graph

Eg



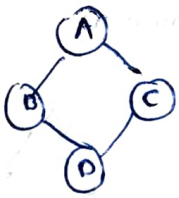
→ Undirected graph : A graph without direction is Undirected graph.

Eg:



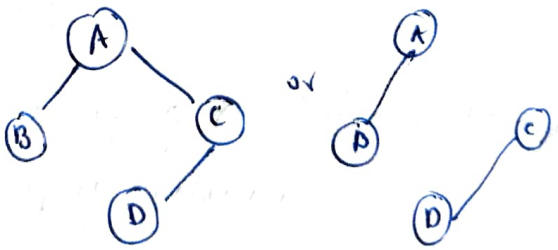
→ Connected graph :- In a connected graph, from one node you can visit any other node is known as connected graph

Eg



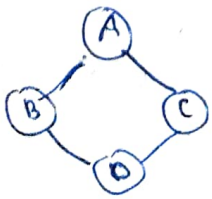
→ Disconnected graph Atleast one node is not reachable from any node

Eg



→ Regular graph : In regular graph degree (no. of Edges connected) of each ~~node~~ vertex is equal

Eg



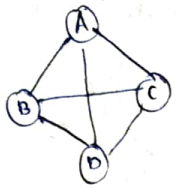
D(A) = 2
D(B) = 2
D(C) = 2
D(D) = 2
} Same

Complete graph: The graph ~~in which~~ do with 'n' vertices and with (n-1) degree for each vertex in a is known as Complete graph.

T.n of vertices = 4, n-1 = 3

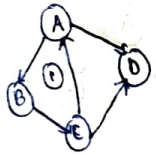
D(A) = 3
D(B) = 3
D(C) = 3
D(D) = 4

Eg:



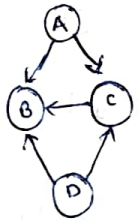
Cyclic graph A graph with (closed loop) or at least one Cycle is known as Cyclic graph

Eg:



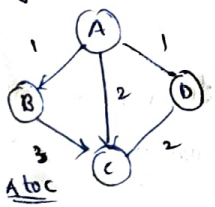
Acyclic graph: A graph without a Cycle (closed path) is known as Acyclic graph

Eg:



Weighted graph: A graphs with weights is known as weighted graph

Eg:



A to C

A → C = 2

A → B → C = 4 3+1 = 4

BFS: (Breadth First Search):

→ BFS follows Queue (FIFO)

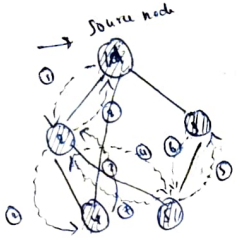
→ 1. First we need to select any node as source so Node
2. Insert into queue (source node)
3. Mark it as visited
4. Remove source node & print, Increasment front
5. ~~Insert into queue~~ Visit adjacent vertices & remove it & print and mark it as visited

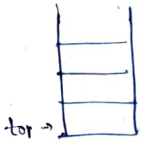
DFS: Depth First search

(20) DFS follows Stack (LIFO)

Algorithm

⇒ Select a node as source node
⇒ Mark it as visited, and display
⇒ Check for any Unvisited adjacent nodes following a particular path
⇒ Insert the adjacent node if it is unvisited into the stack
⇒ If there is no Unvisited adjacent nodes back tracking will be done (removing the ~~node (pop out)~~ and print it. & then (pop out)
⇒ decrement the top (top--)
⇒ ~~agai~~ch again check for any Unvisited adjacent nodes, mark it as visited, if not visited else visited pop out the node
⇒ follow the same process untill stack is empty





1	2	4	5	3

visited

print : | 4 | 3 | 5 | 2 | 1 |

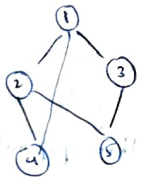
Graph Representation (1. Adjacency Matrix & Adjacency List)

1) Adjacency Matrix :- In adjacency matrix boolean for values are considered. '0' indicate

'0' → No edge nodes between nodes

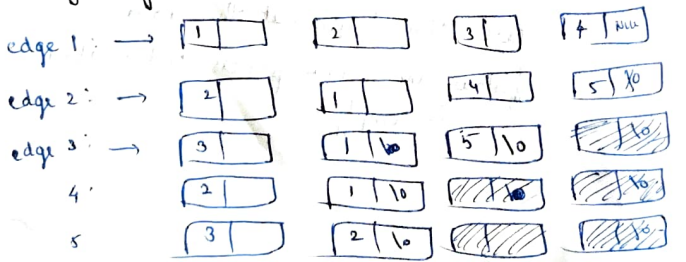
'1' → edge between nodes

Eg:



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

2) Adjacency list :



Assignment BFS & DFS code with Adjacency Matrix.

DFS : with Adjacency list)

Min & Max Create min heap & insert

Implement DFS & BFS with Adjacency matrix.

Construct min & Max heap & perform Insertion & deletion operation (~~Array~~ with array)

Hashing

Hashing is a technique or a process which is keys 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 the type of Hash function is used.

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

size = 10

H(key) = key mod size (or) H(k) = k mod m

H(11) = 11 mode 10 = 1

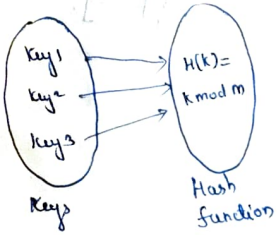
H(12) = 12 mod 10 = 2

H(13) = 13 mod 10 = 3

H(14) = 14 mod 10 = 4

H(15) = 15 mod 10 = 5

Value	keys
0	
1	11
2	12
3	13
4	14
5	15
6	
7	
8	
9	



Key	Value
key1	0
key2	1
key3	2
⋮	⋮

Hash table

Hashing Terminology

Hash key, Hash function, Hash table

Key: Key is integer or a string which is given as input to hash function.

Hash function:- Hash function is a mathematical function that uses key as a input & it will generate value or index. where we can store keys, based on the value generated by hash function.

Hash Table: Hash table is a type of data structure which is used to mapp.

Types of Hash functions

i) Division Method
ii) Mid-square method
iii) Multiplication Method
iv) Digit-folding Method

i) Division Method:

$H(key) = key \bmod m$

key value — hash table size

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

H(10) = 10 mod 10

H(11) = 11 mod 10

H(13) = 13 mod 10

H(17) = 17 mod 10

0	10
1	11
2	
3	13
4	
5	
6	
7	17
8	
9	

ii) Mid-Square Method:

You need to find out the square of key and then find middle or digits.

List = {10, 11, 13}

$H(K) = K^2$

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

0	10
1	
2	11
3	
4	
5	
6	13
7	

]0[0
]2[1
]6[9

∴ This only works for large numbers (i.e after 2 or more digit numbers)

iii) Multiplication Method

$H(k) = floor(m(kA \bmod 1))$

m → table size, k → key value, A → constant

A = ~~0.68103~~

A = 0.618033

$0 < A < 1$

iv) Digit folding Method:-

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

H(12345) → k_1 k_2 k_3

$S = k_1 + k_2 + \ldots k_n$

= 12 + 34 + 5

= 51

0	
51	12345
100	

Collisions:

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

$H(K) = K \bmod m$

list = {20, 10, 11, 21, 33}, m = 10

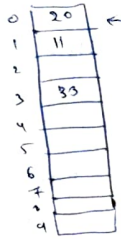
H(20) = 20 mod 10 = 0

H(10) = 10 mod 10 = 0

H(11) = 11 mod 10 = 1

H(21) = 21 mod 10 = 1

H(33) = 33 mod 10 = 3



but (21 & 10) (which is Collision)

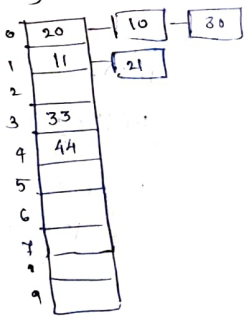
Collision resolving Techniques

- Separate chaining (open hashing) (Linked list is used in separate chaining)
- Open Addressing (closed hashing)
  - → linear probing
  - → quadratic probing
  - → Double-hashing

- Separate chaining

List = {20, 10, 11, 21, 33, 44}

H(10) = 10 mod 10 = 0
H(20) = 20 mod 10 = 0
H(11) = 11 mod 10 = 1
H(21) = 21 mod 10 = 1
H(33) = 33 mod 10 = 3
H(44) = 44 mod 10 = 4



Open addressing

- Linear probing: $(u+i) \% m$
- quardatic: $(u+i^2) \% m$
- Double hashing: $(u+vi)$

9 — 0 to m-1
u — collision location

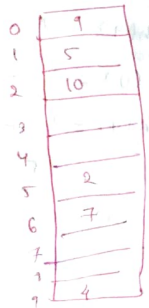
① Linear probing

1. $h(k) = k \% m$
2. If you have free space in hash table at a particular location then we have to store the key value in the hash table.
3. If you don't have empty space, we have to find empty space based on $(u+i) \% m$.
   u → collision location in hash table
   i → 0 to m-1 where m is the size
4. Repeat 2 & 3 Until (size-1). ie (m-1) and store.

given, 5, 10, 4, 2, 7, 9

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

$h(5) = (2(5)+1) \% 10 = 1$
$h(10) = (2(10)+1) \% 10 = 1$
$h(4) = (2(4)+1) \% 10 = 9$
$h(2) = 5$
$h(7) = 5$
Gt $h(9) = 9$



Key	location	Probes
5	1	1
10	1 (collⁿ)	2
4	9	1
2	5	1
7	5	1
9	(9) coll	2

for 10 $(u+i)\%m$

for 2 { $(1+0)\%10 = 1$, $(1+1)\%10 = 2$

for 7 2 { $(5+0)\%m = 5$, $(5+1)\%10 = 6$

for 9 3. $(9+10)\%10 = 9$, $(9+1)\%10 = 0$

Note: The element k will be stored at first three location from $(u+i)\%m$ in has table.

3 6 9 11 4 7 1 Quad $(u+i^2)\%m$

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

$h(3) = 9$

$h(6) = 5$

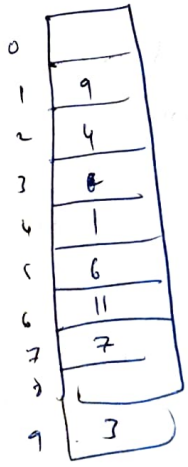
$h(9) = 1$

$h(11) = 5 \rightarrow (6)$

$h(4) = 1 \rightarrow (2)$

$h(7) = 5 \rightarrow (7)$

$h(1) = 5 \rightarrow (4)$



$5+0 = 5$ $5^2+2^2 = 9$

$5+1 = 6$ $5+3^2 = 4$

$1+0 = 1$

$1+1 = 2$

Quadratic Probing

3, 6, 9, 11, 4, 7

$m = 10$, $h(k) = 2k+3$, $(u+i^2)\%m$.

$h(3) = (6+3)\%10 = 9$

$h(6) = (12+3)\%10 = 5$

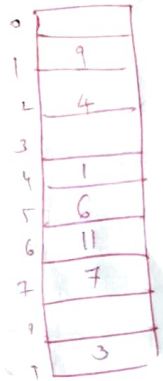
$h(9) = (18+3)\%10 = 1$

$h(11) = (22+3)\%10 = 5$

$h(4) = (8+3)\%10 = 1$

$h(7) = (14+3)\%10 = 7$

$h(1) = (2+3)\%10 = 5$



Key	loc	Probly
3	9	
6	5	
9	1	
11	5(1)	
4	1(2)	
7	7	
1	5(4)	

Collision at 5

$(5+0^2)\%10 = 5$

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

$(5+4)\%10 = 9$

$(5+9)\%10 = 4$ free

Collision at 1

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

$(1+i^2)\%10 = 2$

$(1+2^2)\%10 = 5$ free

Double Hashing

3, 9, 4, 7, 8, 2

$h_1(k) = 2k+3$, $h_2(k) = 3k+1$, $m = 10$

$(u + v*i) \% m$

1) $3 \Rightarrow (2\times3)+3) \% 10$

$= 9$

2) $9 \Rightarrow (2\times9+3) \% 10$

$= 1$

3) $4 \Rightarrow (2(4)+3) \% 10$

$= 1$ but collision occurs

apply second function

$v = (3\times4+1) \% 10$

$= (13) \% 10$

$= 3$

$(u + v*i) \%$

$\rightarrow (1+3\times0) \% 10 = 0$ (i

~~$(1+3\times1) \% 10 = 4$~~

$h(7) = (2(7)+3) \% 10 = 7$

$h(13) = (2(8)+3) \% 10 = 9$ $u=9$, v

$v = (3(8)+1) = 22$

$v =$

$h(2) = (2(2)+3) \% 10 = 7$ $u = 7$

$v = 3k+1$ $w = 7$

$= 3(2)+1 = 7$

$(u+v*i) \% 10 = (7+7\times0) \% 10 \neq 4$

key	location	Prob.
3	9	
9	1	1
4	1 [v=((3×4 +1)%10 v=3]	2

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

u = starting collision

$v =$

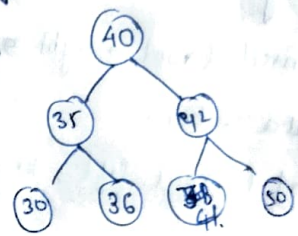
key	Location	Prob

100 150 200 300

BST:

Binary search tree is a non-linear data structure in this left side element must be less than its parent & right side element must be greater than its parent. these condition should be applied for its sub tree recursively.

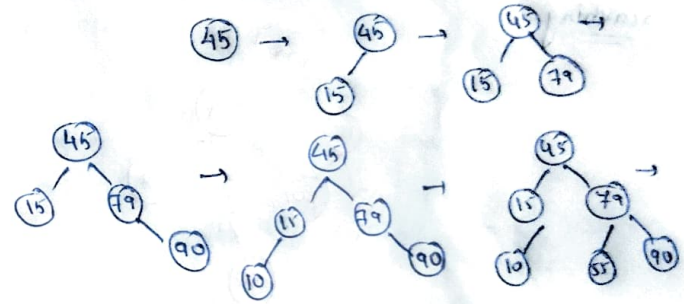
Eg:

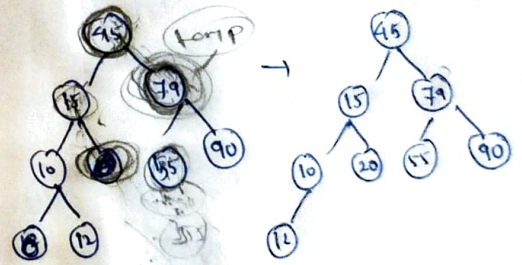


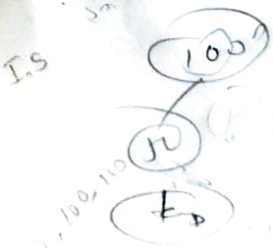
If right child must be less than or greater than its

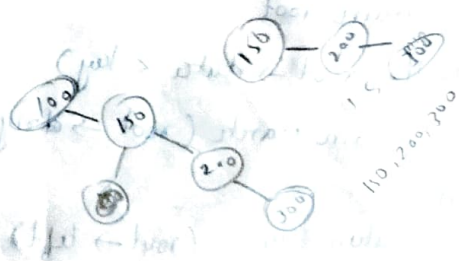
i) Searching
ii) Insertion
iii) Deletion
iv) traversals

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







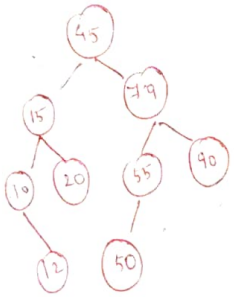


Algorithm :

Insert (root, key)

1) 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.

Searching



Algorithm:

search (root, key)

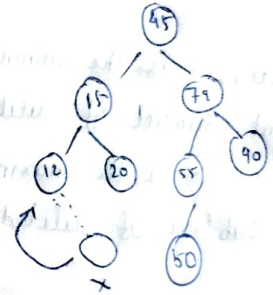
1) if (root → data == key)
   return root
2) else if (root → data < key)
   return search (root → right), key)
3) else
   return search (root → left), key).

Deletion:

1) The node to be deleted is a leaf node (without any children).
2) The node to be deleted is having one child
3) The node to be deleted is having two children.

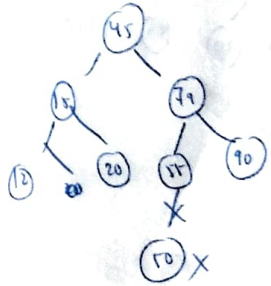
② Node with one child:-

Replace its child with node with its child
remove its child.

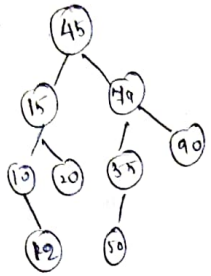


if (root → left == NULL) {
   temp = root → right;
   root → data = temp → data;
   free (temp);

① if (root → left == NULL && root → right == NULL)
   free(root)
   return NULL;



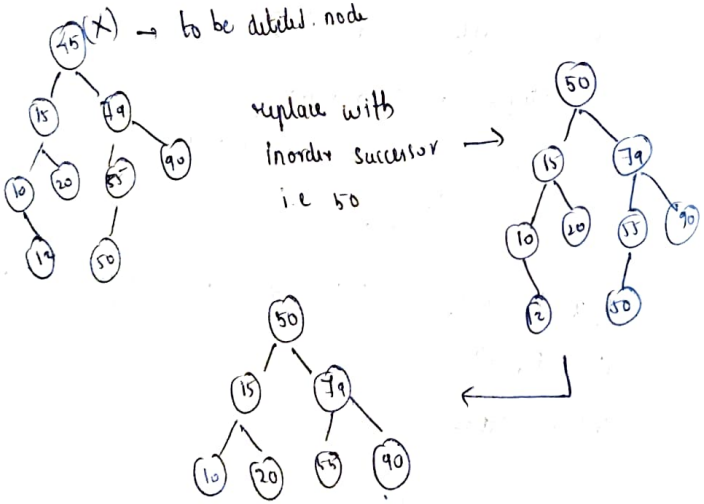
(3) A Node with 2 children

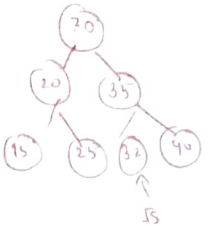


i) Find out inorder successor or inorder predecessor

ii) Replace with the deleted node with inorder successor

iii) Remove inorder successor (is the minimum element in a right subtree of deleted node) (or) inorder predecessor (is the maximum element in left subtree of deleted node)






```
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);
```