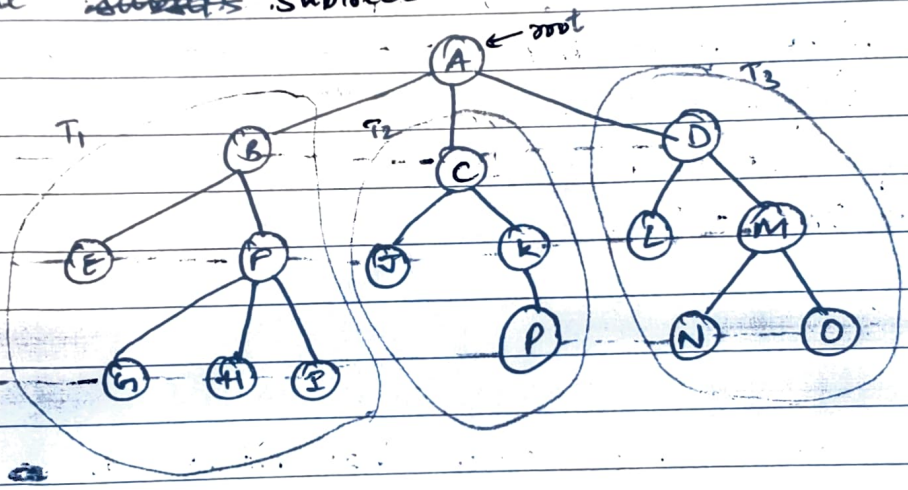
Trees

A tree is a non-linear, non-primitive data structure where in data is stored in a hierarchal fashion.

A tree is a finite set of one or more nodes that exhibits the parent-child relationship such that

1. There is a special node called the root node.
2. The remaining nodes are partitioned into disjoint subsets of nodes like T_1, T_2 upto n, termed as the subtrees.



(1) Siblings

Two or more nodes having a common parent.

eg: J & K are siblings of C.

(2) Ancestors

The nodes obtained in the path from a specific node say x while moving upwards towards the root node are termed as ancestors of x.

eg: Ancestors of node H are F and B.

(3) Degree of a node

The number of sub trees of a given node is termed as the degree of a node.

eg: degree of node N is 0. F is 3.

↳ have degree of 0

④ **Leaf Node**

The nodes in a tree which has degree of 0 is termed as leaf nodes.

eg: E, G, H, I, P, N, O are leaf nodes

⑤ **Level of a tree (node).**

The distance of a node from the root node is termed as the level of a tree.

eg: Level of root node is 0

⑥ **Height of a tree.**

The number of nodes processed to reach the leaf node present in last level from the root node is termed as the height of the tree.

eg: Here height is 4.

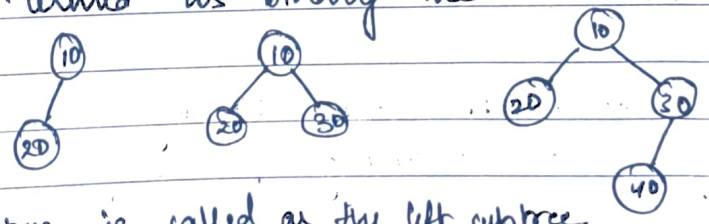
Height of root node is 1.

Ht = max level + 1.

• **Binary tree**

A tree in which each node has either 0, 1 or 2 subtrees is termed as binary tree.

eg:



The first subtree is called as the left subtree.

The second " " right subtree.

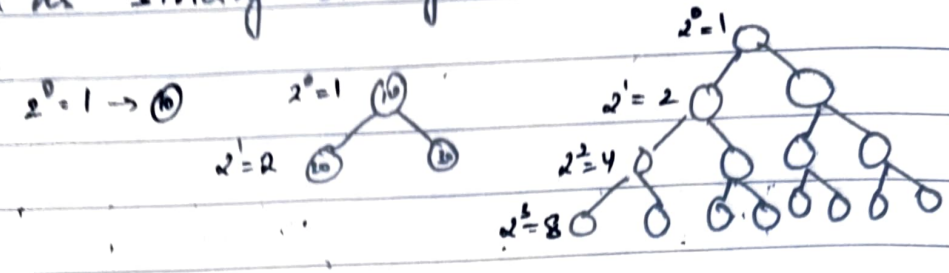
Types of Binary trees

① Strictly binary tree
② Complete binary tree
③ Skewed binary tree
④ Expression tree
⑤ Binary search tree

Strictly binary tree (Full bin tree)

A bin tree having 2^i nodes at any given level i is referred as strictly binary tree.

eg:

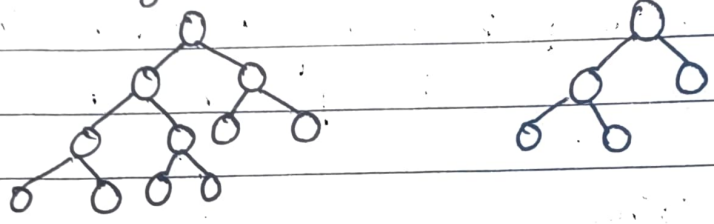


(2) Complete bin tree

A bin tree in which at every level apart from the last level if it is completely filled we term it as complete bin tree.

In the last level the nodes should be filled from left to right

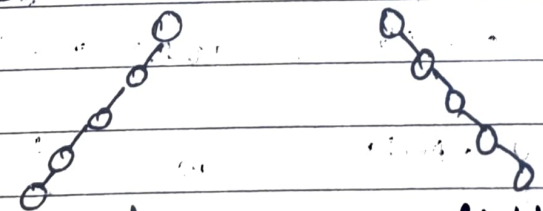
eg



(3) Skewed binary tree

If a bin tree is built only on one side its called as skewed bin tree.

eg:



Left skewed

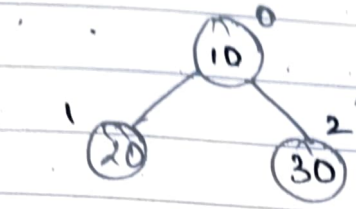
Right skewed.

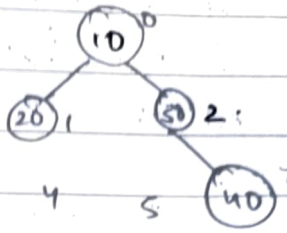
Representation of a binary tree

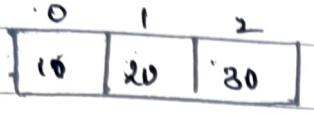
A bin tree can be represented in 2 ways:

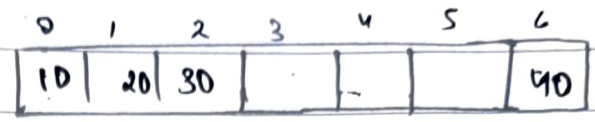
1. Array rep^n
2. List rep^n

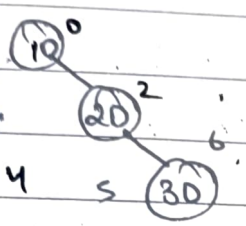
① Array representation

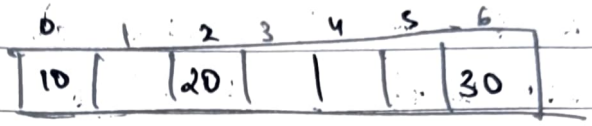












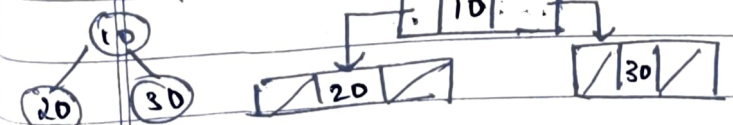
not useful cause wasting too much space.

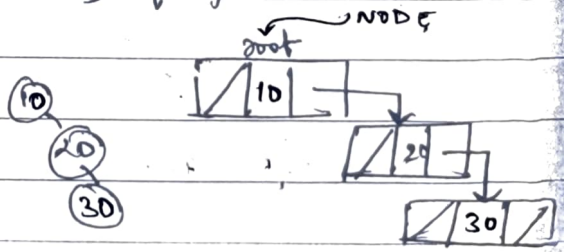
* If a tree is a complete binary tree then it is efficient to use arrays to represent a tree.

② List representation

We represent each node with the following structure def^n

```
struct node
{ int info;
  struct node *left;   //address of left subtree/child
  struct node *right;  //address of right child
};
typedef struct node *NODE;
```





Operations on Binary tree.

① Traversal operation

It is a method of visiting all the nodes of a tree exactly once.

Types of traversal:

1. Pre order traversal
2. Inorder "
3. Post order "

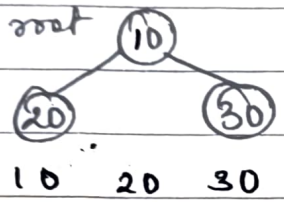
} can be used only if tree is binary.
If not binary, convert it into bin tree & then perform traversal.

1. Preorder traversal (NLR)

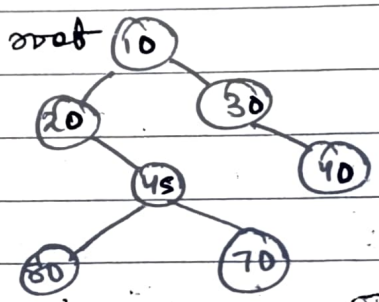
root left right

The recursive def^n of the preorder traversal is

1. Visit the root node.
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder.



10 20 30



10 20 45 80 70 30 40

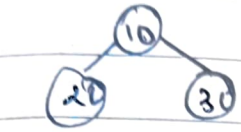
A recursive func^n for preorder:

```
void preorder (NODE root)
{
    if(root != NULL)
    { printf("%d \t", root->info);
      preorder(root->left);
      preorder(root->right);
    }
}
```

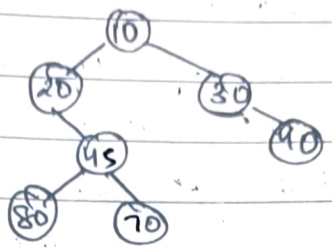
(2) Inorder traversal (LNR)

The recursive def^n of inorder traversal is:

1. Traverse the left subtree in inorder.
2. Visit the root node.
3. Traverse the right subtree in inorder.



20 10 30



20 80 45 70 10 30 40

Recursive def^n:

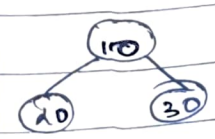
```
Void inorder (NODE root)
{  if (root != NULL)
     { inorder (root->left);
       printf ("%d \t", root->info);
       inorder (root->right);
     }
}
```

I should be at 4th position.

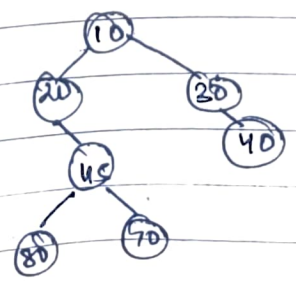
(3) Postorder traversal (LRN)

The recursive def^n to perform postorder traversal is:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root node



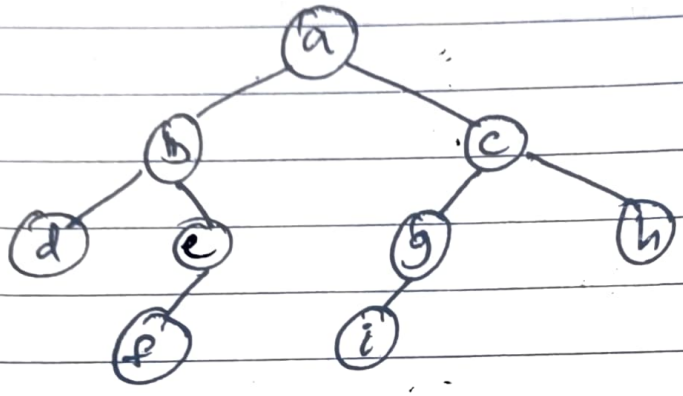
20 30 10



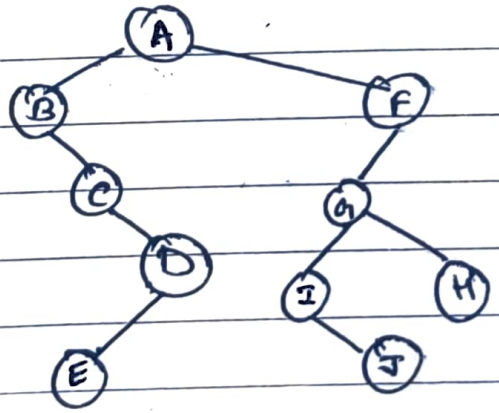
80 70 45 20 40 30 10

```
void postorder (NODE root)
{   if (root != NULL)
    {   postorder (root->left);
        postorder (root->right);
        printf ("%d\t", root->info);
    }
}
```

Q Perform the traversal for given graph.



preorder → a b d e f c g i h
inorder → d b f e a i g c h
postorder → d f e b i g h c a

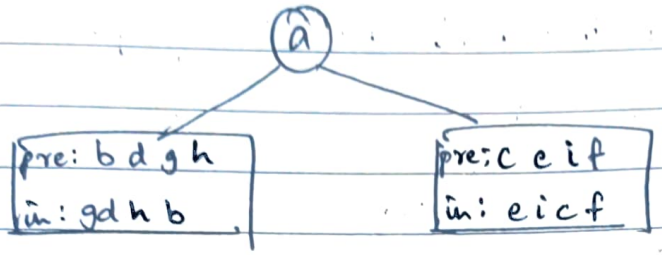


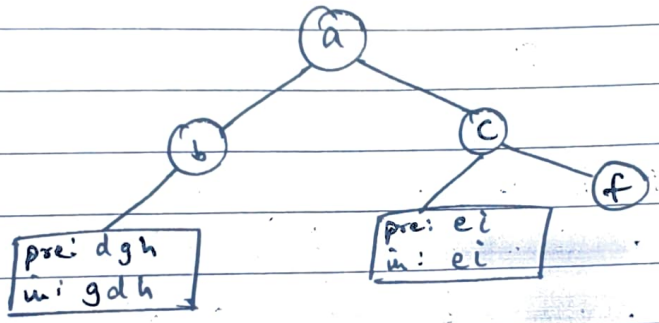
NLR pre → A B C D E F G I J H
LNR in → B C E D A I J G H F
LRN post → E D C B J I H G F A

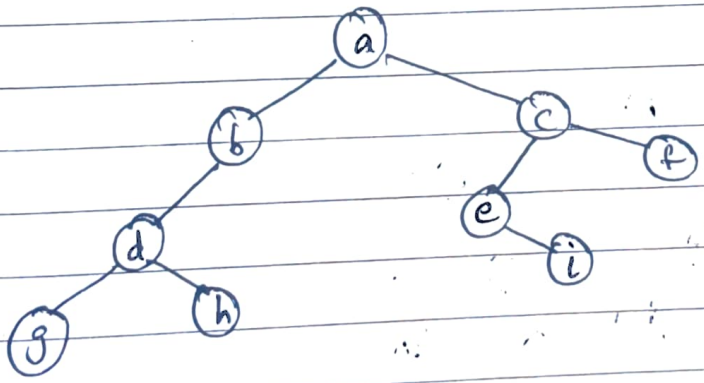
comb^n of pre & in
↑ or post & in

Q Construct a bin tree from a given traversal.

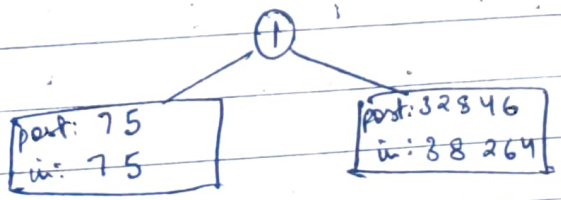
root
↑
LR preorder → a b d g h c e i f
NR inorder → g d h b a e i c f



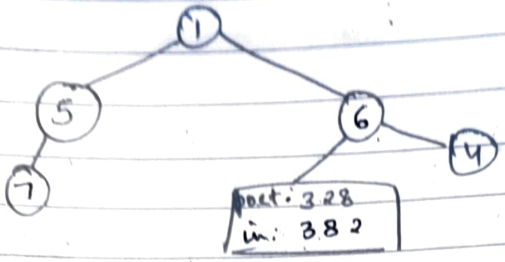


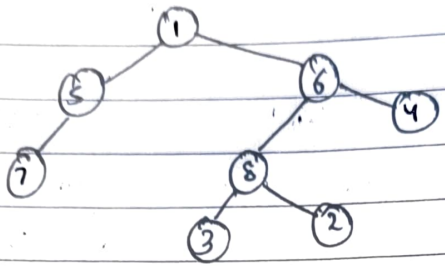


Q post: 7 5 3 2 8 4 6 1 LRN
in: 7 5 1 3 8 2 6 4 LNR



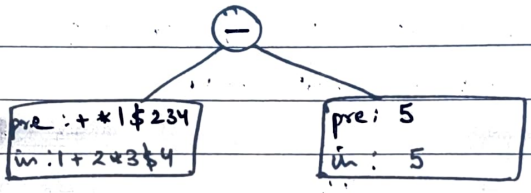
others are operators

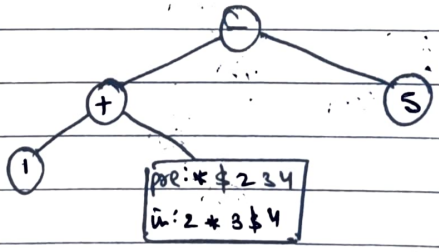


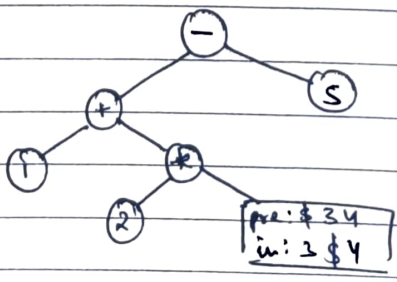


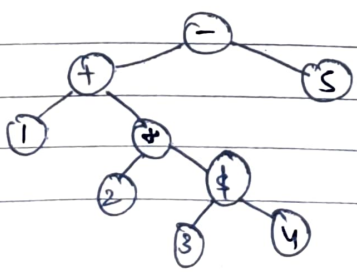
8 preorder: − + * 1 $ 2 3 4 5 NLR

inorder: 1 + 2 * 3 $ 4 − 5 LNR







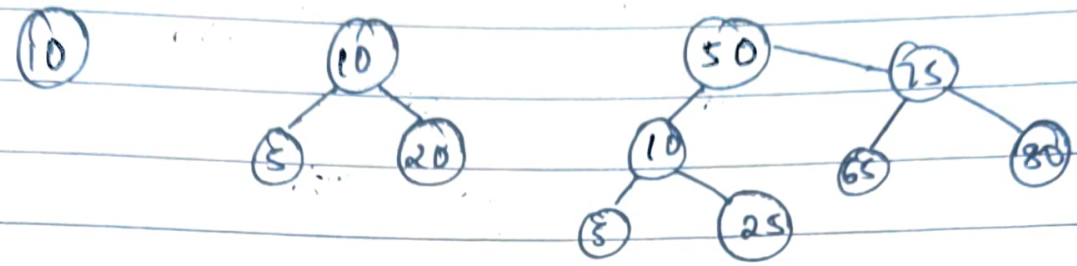


(1 + (2 * (3 $ 4)) − 5

Binary search tree

It is a binary tree such that for each node x in the binary tree, the elements in the left subtree are less than data of x and the elements in the right subtree are greater than / equal to data of x.

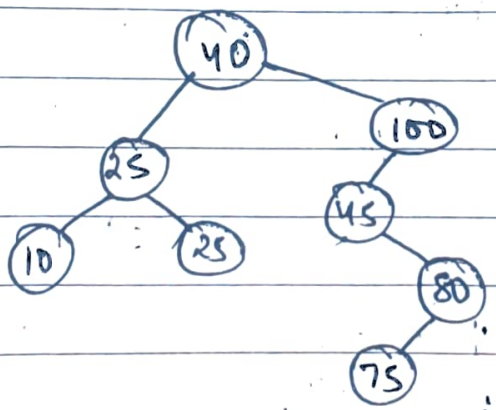
eg:



Q Construct a BST for a given set of elements:

40 100 25 45 25 80 10 75

→ 1st element always root node



Compare 100 with 40

25 < 40

45 with 40

45 < 100

25 < 40

25 < 25

80 > 40

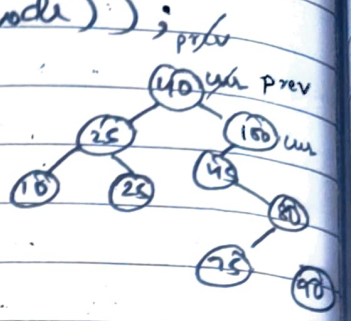
80 < 100

80 > 45

10 < 40

Function to create a bin search tree.

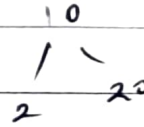
```
NODE createBST (NODE root, int item)
{  NODE temp, cur, prev;
   temp = (NODE) malloc (sizeof (struct node));
   temp -> data = item;
   temp -> left = NULL;
   temp -> right = NULL;
   if (root == NULL)
      return temp;
   cur = root;
   prev = NULL;
   while (cur != NULL)
    {  prev = cur;
       if ( item < cur -> data )
              cur = cur -> left;
       else
              cur = cur -> right;
    }
   if (prev -> data > item)
       prev -> left = temp;
   else
       prev -> right = temp;
   return root;
}
```



LAB 7 Create BST, traverse & delete a node.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{  int data;
   struct node *left;
   struct node *right;
};
```

```
typedef struct node *NODE;
// create BST
// write preorder, inorder, postorder
NODE delete(NODE root, int key)
{ NODE temp;
  if(root == NULL)
     return root;
  if(key < root->data)
     root->lft = delete(root->lft, key);
  else if(key > root->data)
     root->right = delete(root->right, key);
  else
  { if(root->lft == NULL)
    { temp = root->right;
      free(root);
      return temp;
    }
    if(root->right == NULL)
    { temp = root->lft;
      free(root);
      return temp;
    }
    temp = inordersuccessor(root->right)
    root->data = temp->data;
    root->right = delete(root->right, temp->data);
  }
  return root;
}
```

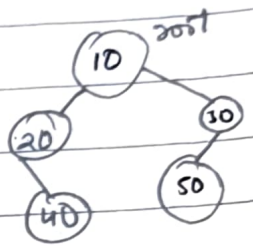


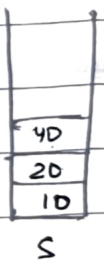
```
int main()
{
    NODE root = NULL;
    int item, key, ch;
    for(;;)
    { printf("1. Insert \n 2. Delete \n 3. Preorder \n
              4. Inorder \n 5. Postorder \n 6. Exit \n");
      printf("Read choice \n");
      scanf("%d", &ch);
      switch(ch)
      { case 1: printf("\n Enter element to be inserted: ");
                scanf("%d", &item);
                root = creatBST(root, item);
                break;

        case 2: printf("\n Enter the node to be deleted \n");
                scanf("%d", &key);
                root = delete(root, key);
                break;

        case 3: printf("\n Preorder traversal is \n");
                preorder(root);
                break;
        case 4: printf("\n Inorder traversal is \n");
                inorder(root);
                break;
        case 5: printf("\n Postorder traversal is \n");
                postorder(root);
                break;
        default: exit(0);
      }
    }
}
```

```
NODE inordersuccessor (NODE root)
{ NODE cur = root;
  while (cur->left != NULL)
      cur = cur->left;
  return cur;
}
```

A C function to perform iterative preorder traversal

```
void preorder_iter (NODE root)
{ NODE cur, s[20]; int top = -1;
  if (root == NULL)
    { printf ("\n Empty tree");
      return;
    }
  cur = root;
  while (1)
  { while (cur != NULL)
      { printf ("%d\t", cur->data);
        s[++top] = cur;
        cur = cur->left;
      }
    if (top != -1)
    {
      cur = s[top--];
      cur = cur->right;
    }
    else
      return;
  }
}
```





Write a iterative func to perform the iterative inorder traversal

```
void inorder_iter(NODE root)
{   NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {  printf("\n Empty tree");
       return;
    }
    cur = root;
    while(1)
    {  while(cur != NULL)
       {    s[++top] = cur;
            cur = cur -> left;
       }
       if (top != -1)
       {  cur = s[top--];
          printf("%d\t", cur -> data);
          cur = cur -> right;
       }
       else
          return;
    }
}
```

Q Write an iterative func to perform postorder traversal

```
void postorder_iter(NODE root)
{   NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {  pf("\n Empty tree");
       return;
    }
```

```
cur = root;
while(1)
{   while (cur != NULL)
    {  s[++top] = cur;
       cur = cur -> left;
    }
    if (top != -1)
    {  cur = s[top--];
       cur = cur -> right;
    }
    else
        printf ("%d\t", cur -> data);
}
}
```

2) To write a C func to check or to find 2 binary trees are same or not.

```
int issameBT (NODE root1, NODE root2)
{

  if (root1 == NULL && root2 == NULL)
      return 1;
  if ((root1 != NULL && root2 != NULL)
      return root1->data == root2->data
    && issameBT (root1->left, root2->left)
    && issameBT (root1->right, root2->right))
}
```

3) To check whether 2 BST are same or not.

```
int issameBST (NODE root1, NODE root2)
{  if (root1 == NULL && root2 == NULL)
      return 1;
   if (root1 != NULL && root2 != NULL)
```

8 To count the no. of nodes in a binary tree.

```
int count = 0          // is a global variable
count_nodes (NODE root)
{
    if (root == NULL)
        return 0;
    count++;
    count_nodes (root->left);
    countnodes (root->right);
}
```

8 To count the no. of leaf nodes in B.T

```
int count = 0;
void countleaf (NODE root)
{
    if (root == NULL)
        return;
    countleaf (root->left);
    if (root->left == NULL && root->right == NULL)
        count++;
    countleaf (root->right);
}
```

9 To seach a key in BST

```
NODE search (NODE root, int key)
{
    if (root == NULL)
        return NULL;
    if (key == root->data)
        return root;
```

```
    if (key < root->data)
        return search (root->left, key);
    return search (root->right, key);
}
```

Q To find the height of a given BT

```
int max (int a, int b)
{
    if (a > b)
        return a;
    return b;
}

int height (NODE root)
{
    if (root == NULL)
        return -1;
    if (root != NULL)
        return 1 + max (height(root->left), height(root->right));
}
```

Q To find the max element in BST

```
NODE max ( NODE root )
{
    NODE cur = root;
    while (cur->right != NULL)
        cur = cur->right;
    return cur;
}
```

Expression Tree

Expression tree is a binary tree wherein a give infix expression can be represented as a binary tree with operands as leaf nodes and operators as internal nodes.

Algorithm to construct an exp tree.

1. Scan the given expression from left to right.
2. Initialize 2 stacks namely a tree stack and operator stack.
3. If the scanned symbol is
   (a) an operand - construct a node for the operand & push the node onto the tree stack.
   (b) an operator - if operator stack is empty or the precedence of the operator on the top of the operator stack is less than the precedence of the scanned operator, construct a node for the operator & push it onto the operator stack.
   else pop 2 nodes from the tree stack & attach them as the right & the left child & push the operator node onto the tree stack and push the scanned operator onto the operator stack.
   (pop an operator node from the operator stack).
4. Until the operator stack becomes empty, pop an operator node from the operator stack and 2 nodes from the tree stack, attaching them as the right and the left child and push the operator node onto the tree stack.
5. Return tree stack to get your exp. tree.

Expression Tree

Expression tree is a binary tree wherein a give infix expression can be represented as a binary tree with operands as leaf nodes and operators as internal nodes.

Algorithm to construct an exp tree.

Step 2 → Scan the given expression from left to right.

Step 1 → Initialize 2 stacks namely a tree stack and operator stack.

Step 3 If the scanned symbol is

(a) an operand - construct a node for the operand & push the node onto the tree stack.

(b) an operator - if operator stack is empty or the precedence of the operator on the top of the operator stack is less than the precedence of the scanned operator, construct a node for the operator & push it onto the operator stack.

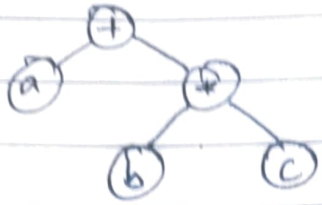
else pop 2 nodes from the tree stack & attach them as the right & the left child & push the operator node onto the tree stack and push the scanned operator onto the operator stack.

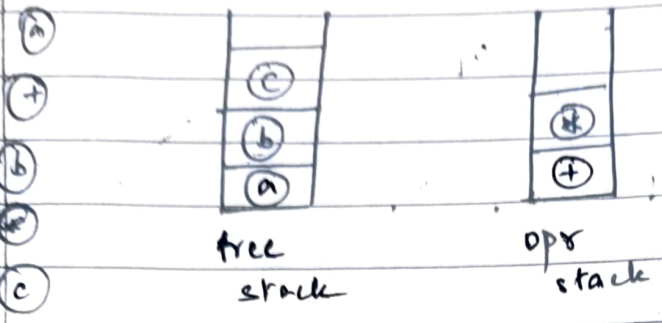
(pop an operator node from the operator stack).

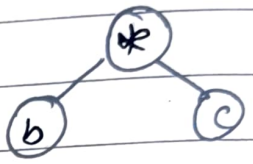
Step 4 Until the operator stack becomes empty, pop an operator node from the operator stack and 2 nodes from the tree stack, attaching them as the right and the left child and push the operator node onto the tree stack.

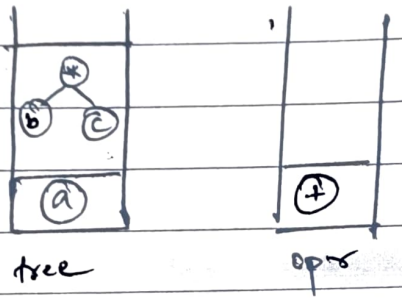
Step 5 Return tree stack to get your exp. tree.

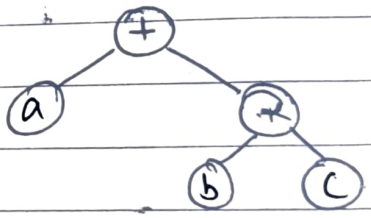
a + b * c

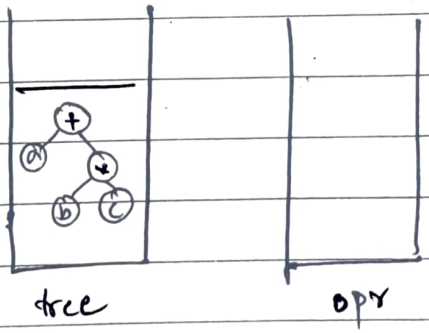




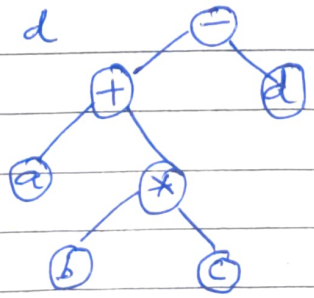








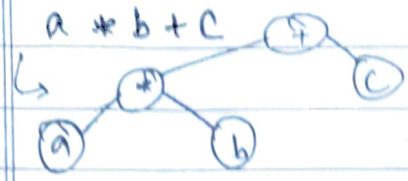
a + b * c - d

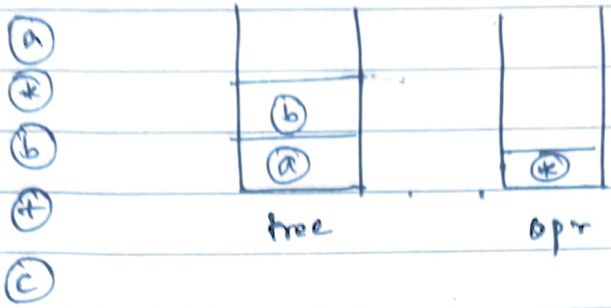


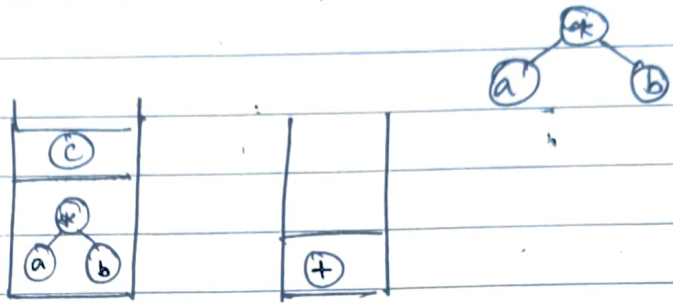
pre → - + a * b c d

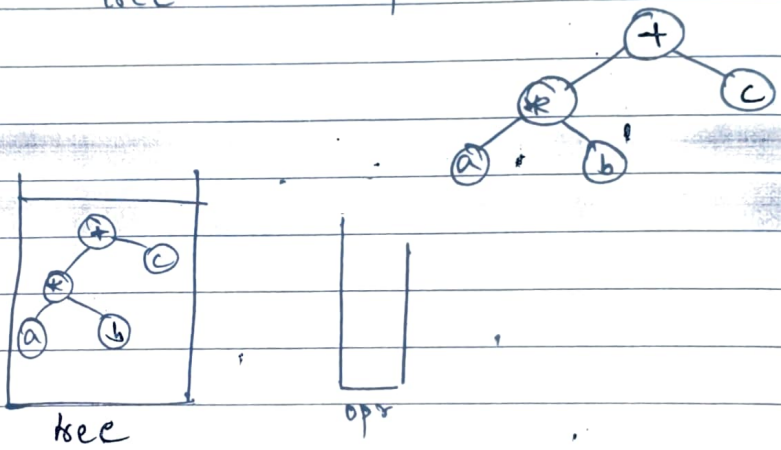
post → a b c * + d -

Q a * b + c









Q LAB-8

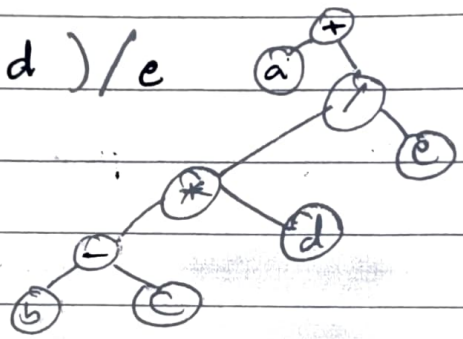
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct node
{ char data;
  struct node *left;
  struct node *right;
};
typedef struct node
```

```
NODE create_node (char item)
{
  NODE temp;
  temp = (NODE) malloc (sizeof (struct node));
  temp -> data = item;
  temp -> lft = NULL;
  temp -> right = NULL;
  return temp;
}

int precede (char c)
{
  switch (c)
  { case '$': return 5;
    case '*':
    case '/': return 3;
    case '+':
    case '-': return 1;
  }
}

NODE createexptree (char infix[15])
{  char symbol
   NODE treestack[20], opstack[20], temp, t, l, r;
   int top1 = -1, top2 = -1, i,
   for (i = 0; infix[i] != '0'; i++)
     {
       symbol = infix[i];
       if(isalnum(symbol))
         {
           temp= create_node(symbol);
           treestack[++top1] = temp;
         }
```

```
else
 {
    temp = creat_node (symbol);
    if (top2 == -1) || preced(symbol) > preced(opstack[top2]->data))
        opstack [++top2] = temp;
    else while (preced(symbol) <= preced(opstack [top2]->data))
    { t = opstack [top2--];
      r = treestack [top1--];
      l = treestack [top1--];
      t->right = r;
      t->left = l;
      treestack [++top1] = t; }
    opstack[++top2] = temp;
 }
}
while ( top2 != -1)
{
    t = opstack [top2--];
    r = treestack [top1--];
    l = treestack [top1--];
    t->right = r;
    t->left = l;
    treestack [++top1] = t;
 }
 return treestack [top1];
}
// write preorder, inorder, postorder
// change %d to %c

int main()
{
   NODE root = NULL;
   char infix[20];
```

```
printf("\n Read the expression \n");
scanf("%s", infix);
root = createexptree(infix);
printf("\n Preorder traversal is \n");
preorder(root);
printf("\n Inorder traversal is \n");
inorder(root);
printf("\n Postorder traversal is \n");
postorder(root);
}
```

Q a + ((b-c) *d)/e

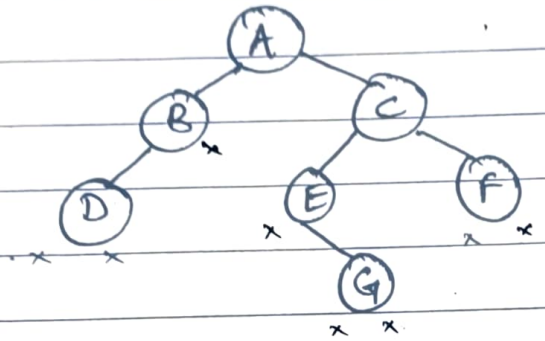


Q Write a short note on threaded binary tree.

Threaded Binary Trees

Drawback of bin tree

- 50% of the address are stored NULL.
- time consumed is more for traversal operation
- To find the successor / predecessor node it takes more time.



To overcome this, we go for threaded bin trees. (based on traversal)

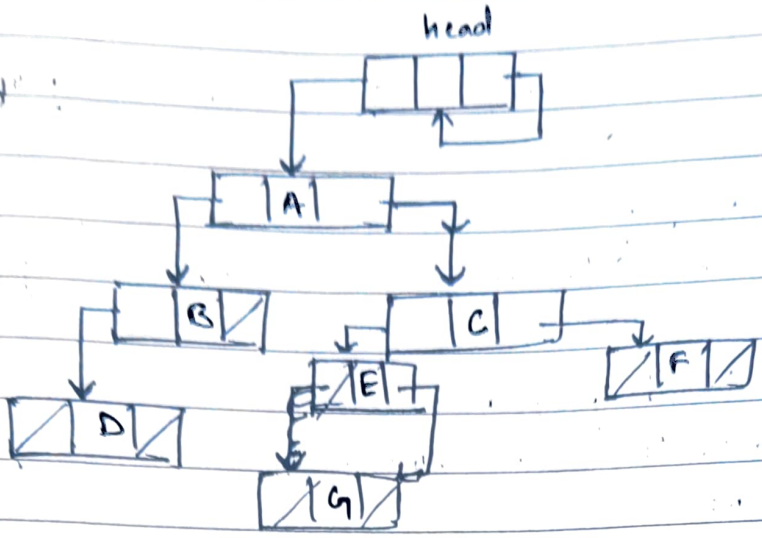
consists of head node which contains addr of root node.

1. In-threaded bin tree
2. Post - " "
3. Pre- " "

2 types

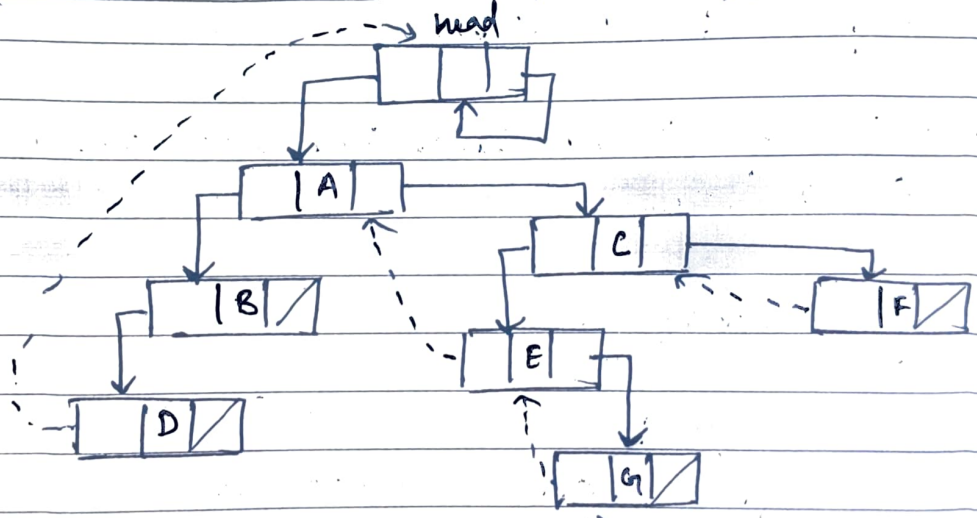
left → predecessor

right → successor

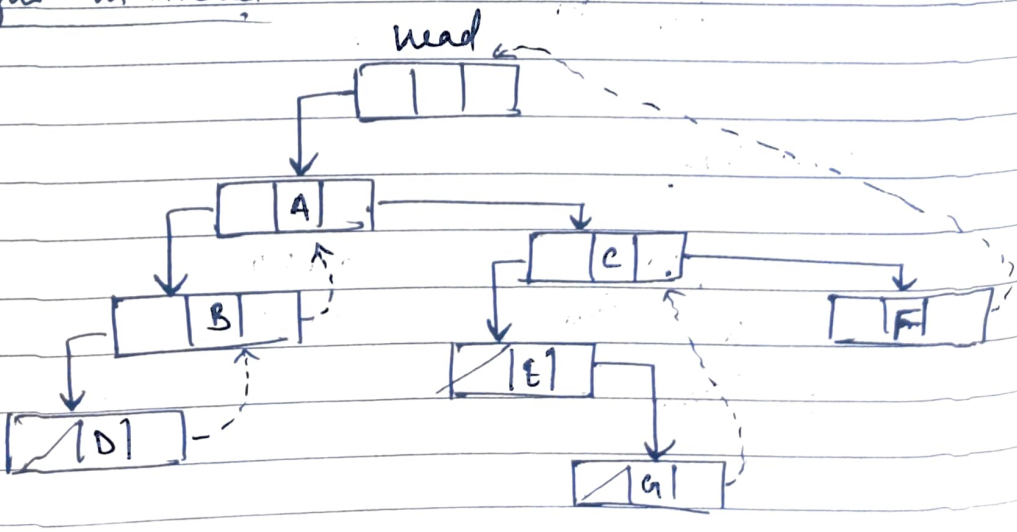


Inorder : D B A E G C F

Left in-threaded bin tree (predecessor)



Right in-threaded bin tree (successor)



According to inorder

Heap

Heap is a binary tree with 2 additional properties :

(1) Structural property - → except last level all levels 2^i node.
The tree should be almost complete binary tree.

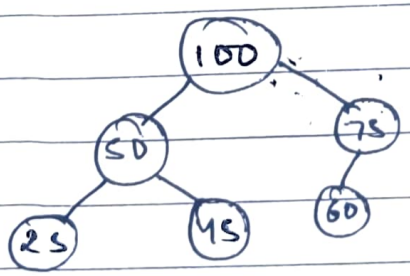
(2) Parent dominant property -
Parent node should be more dominant when compared to its children.

There are 2 types of heaps :

(1) Max heap - Parent should have higher value when compared to its children.

(2) Min heap - Parent should have lesser value compared to its children.

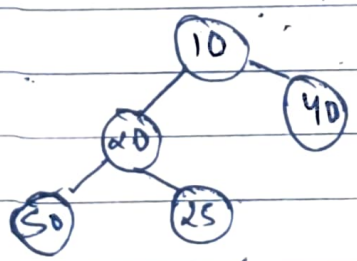
eg: Max heap



50 & 75 < 100

25 & 45 < 50

Min heap



10 & 40 > 10

50 & 25 > 20

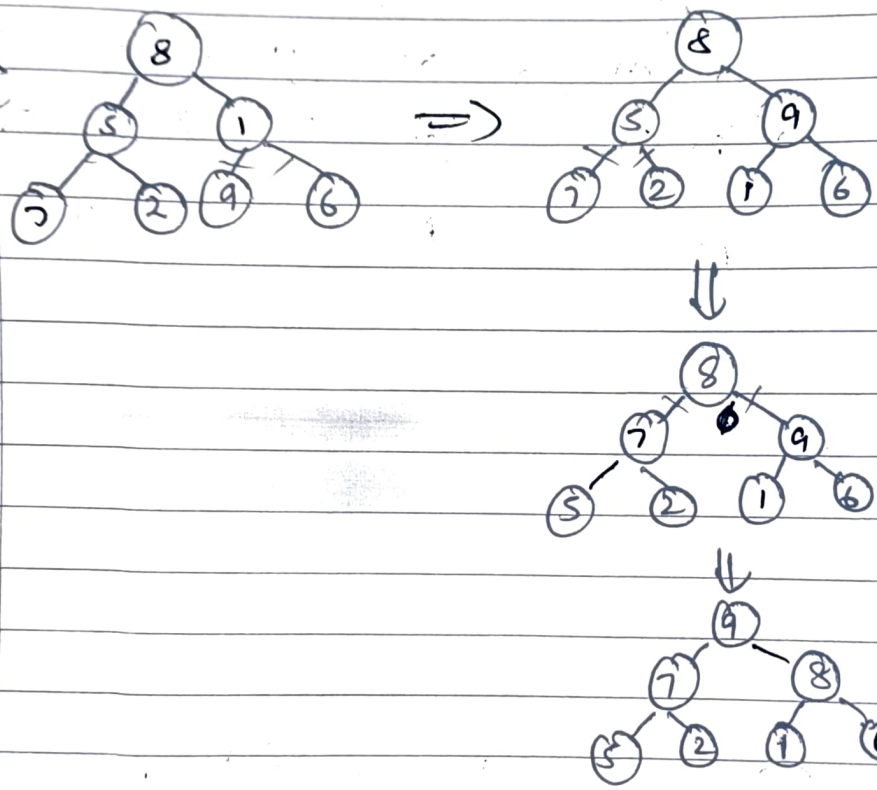
There are 2 methods to construct a heap.

1. Bottom-up method
2. Top down method

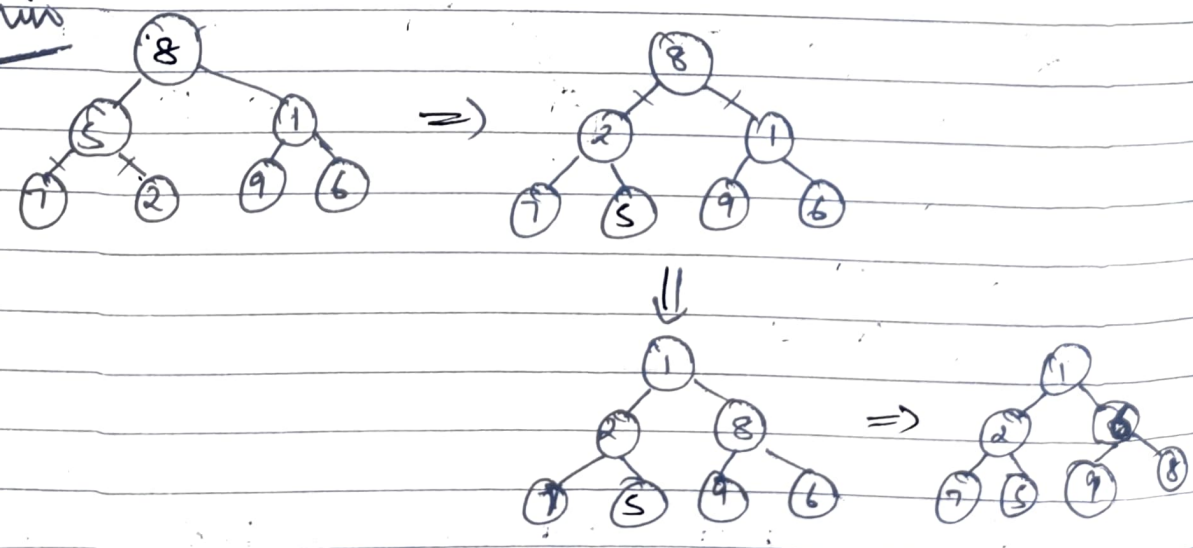
Bottom up.

Construct a max heap for the following elements:
8 5 1 7 2 9 6

Max



Min



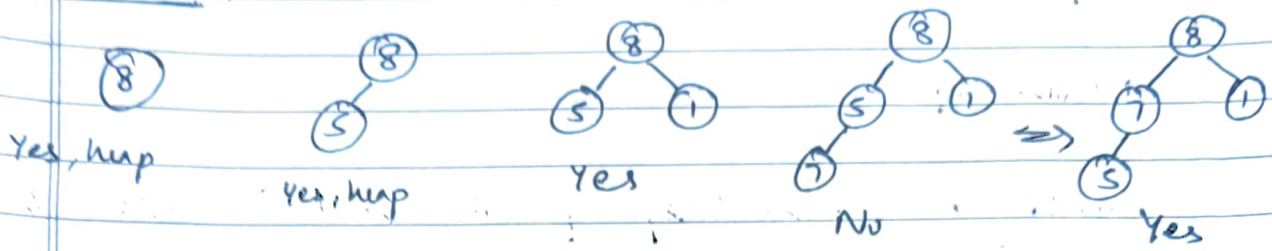
* Not a compulsion that you'll get same heap using both methods.

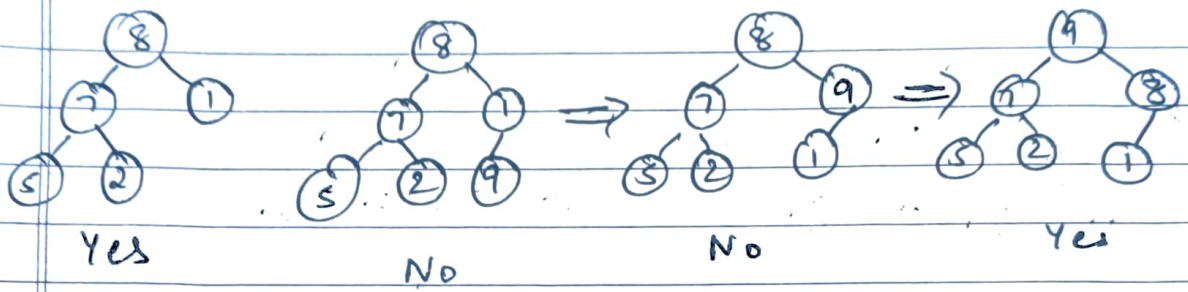
classmate
Date
Page

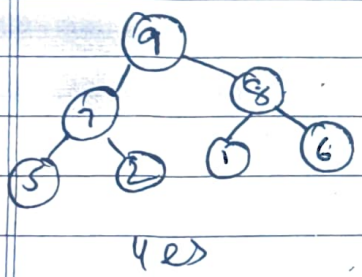
Top down.

8 5 1 7 2 9 6

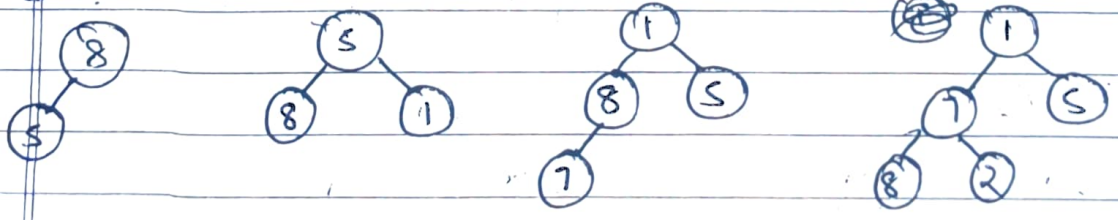
Max

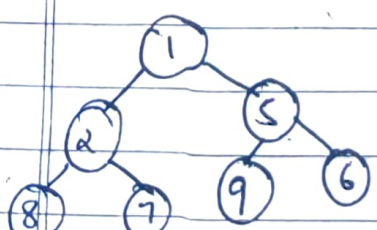






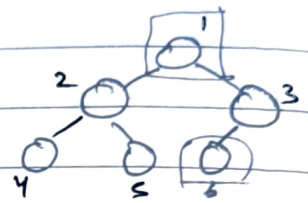
Min





Program to implement priority queue using heap.

```
#include <stdio.h>
#include <stdlib.h>
int n;
int extractmax(int a[10])
{ int max
  if (n == 0)
  {
     printf("Heap is empty");
     return -1;
  }
  else
  { max = a[1];
     a[1] = a[n];
     n = n-1;
     heapify(a,1);
     return max;
  }
}
void buildheap(int a[10])
{ int i
  for (i = n/2; i >= 1; i--)
     heapify(a,i);
}
void heapify(int a[10], int i)
{ int left, right, largest;
  left = 2*i;
  right = 2*i+1;
  if (left <= n && a[left] > a[i])
     largest = left;
  else
     largest = i;
  if(right <= n && a[right] > a[largest])
     largest = right;
  if (largest != i)
  {
     swap(&a[i], &a[largest]);
     heapify(a,largest);
```



position

```
int main()
{  int a[10], i, ch, del
   printf("\n Read no. of elements
   printf("1. To creati heap \n 2. Delete \n 3. Exit \n");
   printf("Read choice \n");
   scanf("%d", &ch);
   switch(ch)
   {  case 1: printf("Read no. of elements \n");
              scanf("%d", &n);
              printf("Read elements \n");
              for(i=1; i<=n; i++)
                   scanf("%d", &a[i]);
              buildheap(a);
              printf("Elements of the constructing heap \n");
              for(i=1; i<=n; i++)
                   printf("%d \t", a[i]);
              break;

      case 2: del = extractmax(a);
              if(del != -1)
                   printf("Element deleted is %d \n", del
              printf("\n Elements after deletion \n");
              for(i=1; i<=n; i++)
                   printf("%d \t", a[i]);
              break;
      default: exit(0);
   }
}

void swap(int *a, int *b)
{  int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

efficiency is log n

AVL tree (Adelson / Velski / Landis) (Balanced BST)

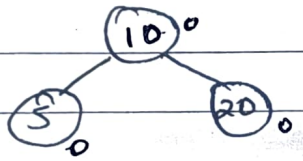
→ In a binary search tree to perform any operation the worst case efficiency (time taken) is order of n. O(n).

→ An AVL tree is a BST wherein in worst case to perform any operation we take O(log n) time.

AVL tree is a BST wherein the balance factor of each node is either 0, +1 or −1.

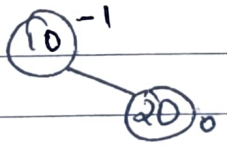
Balance factor = ht. of left sub tree − ht. of right sub tree

eg:



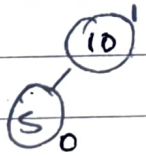
(5) Left subtree = 0
Right " = 0
Bal factor = 0

eg

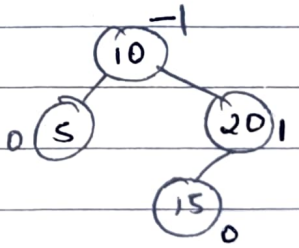


20
10 left = 1 right = 1
Bal = 1 − 1 = 0

eg



eg



∴ AVL.

→ If a BST is not an AVL tree, to convert a BST to an AVL tree we perform an operation called rotation.

→ There are 2 types of rotations:

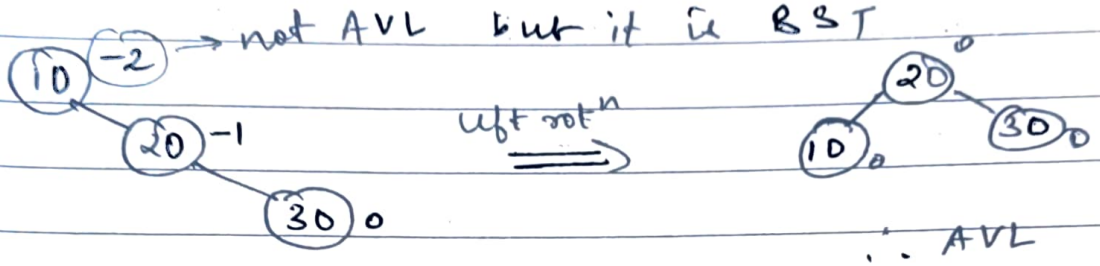
1. Single rotation
2. Double rotation

(1) Single rotation

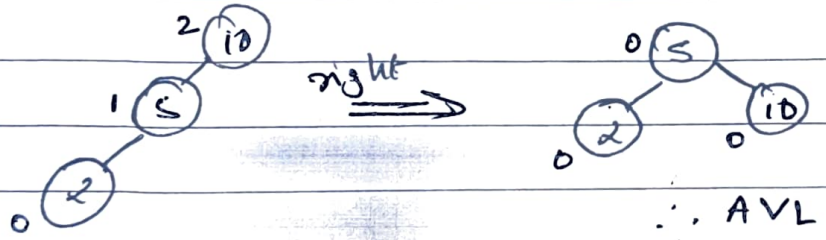
In a single rotation, there are 2 types:

1. Left rotation
2. Right rotation

Left rotation



Right rotation

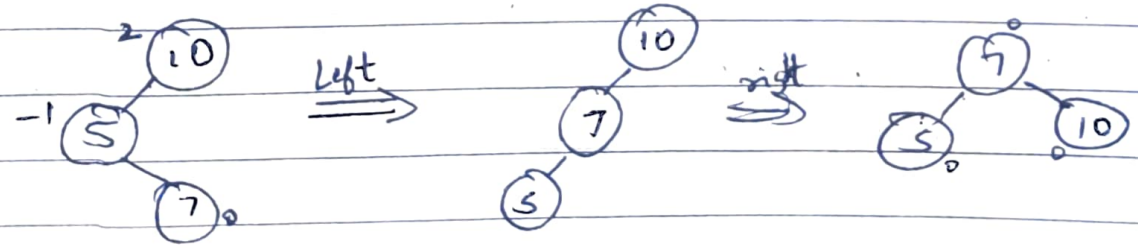


(2) Double Rotation

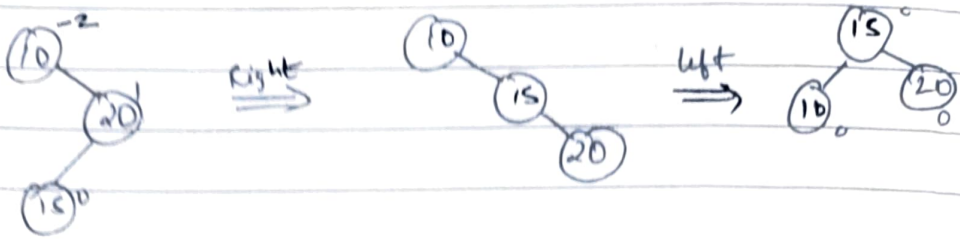
There are 2 types:

1. Left - right rotation (LR)
2. Right - left rotn (RL)

LR

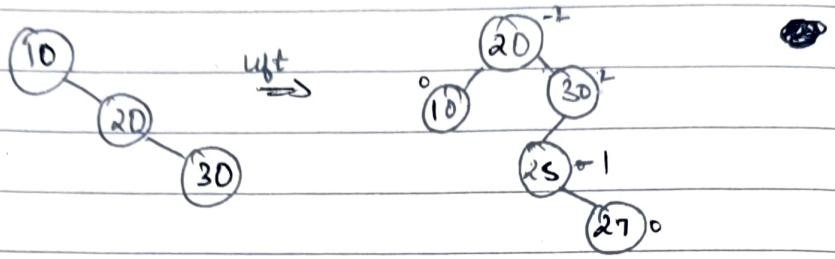


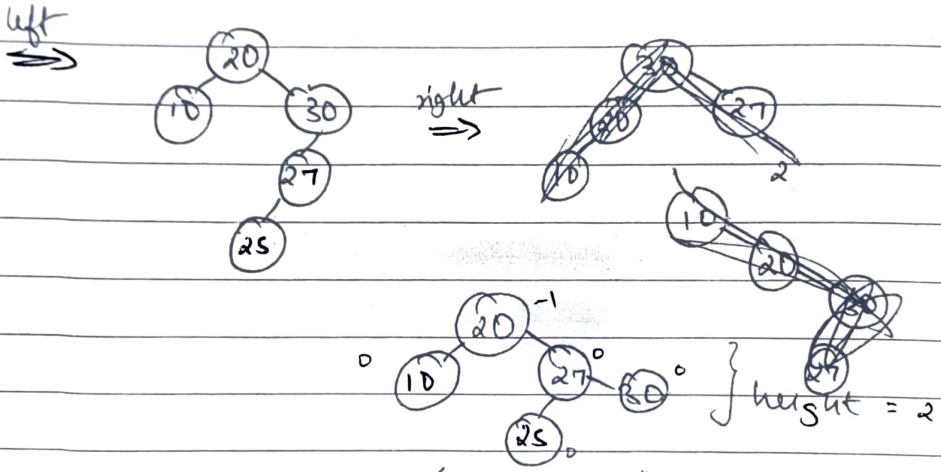
RL

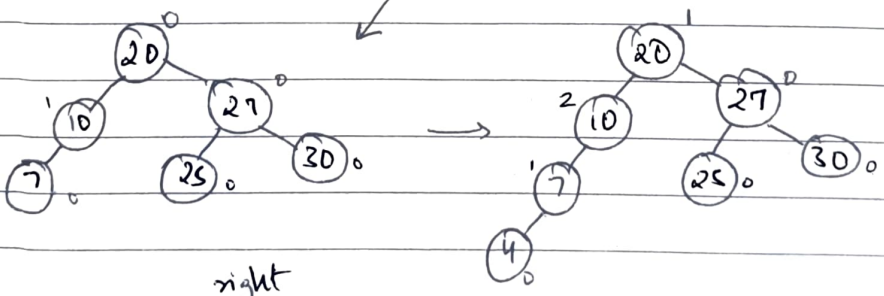


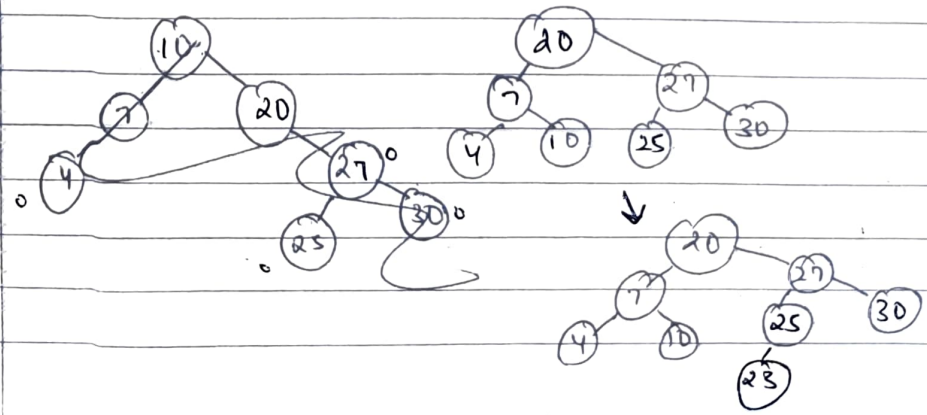
8. Construct an AVL tree for the elements

10 20 30 25 27 7 4 23

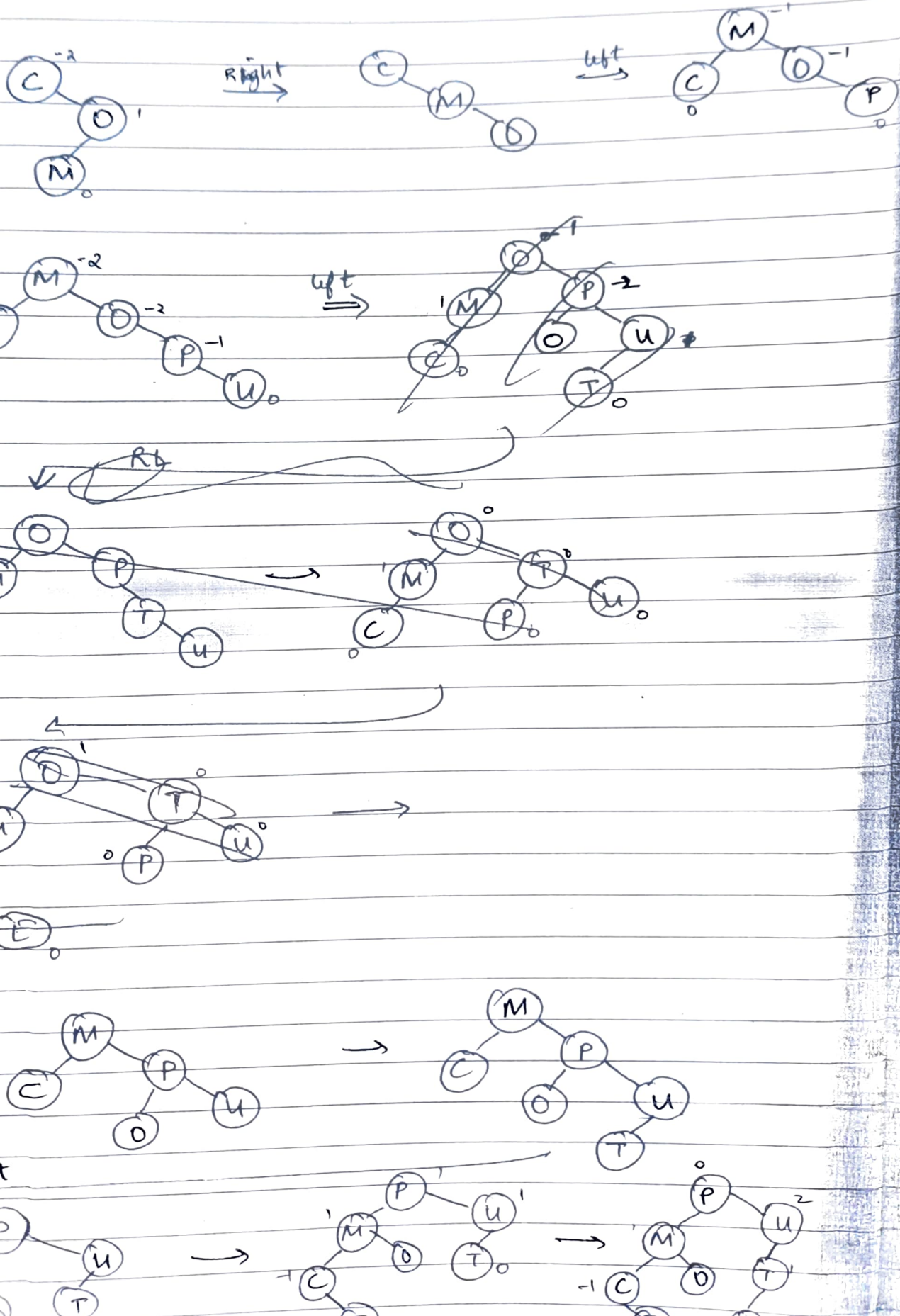




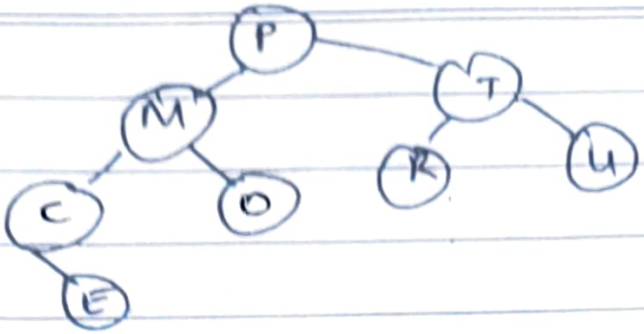




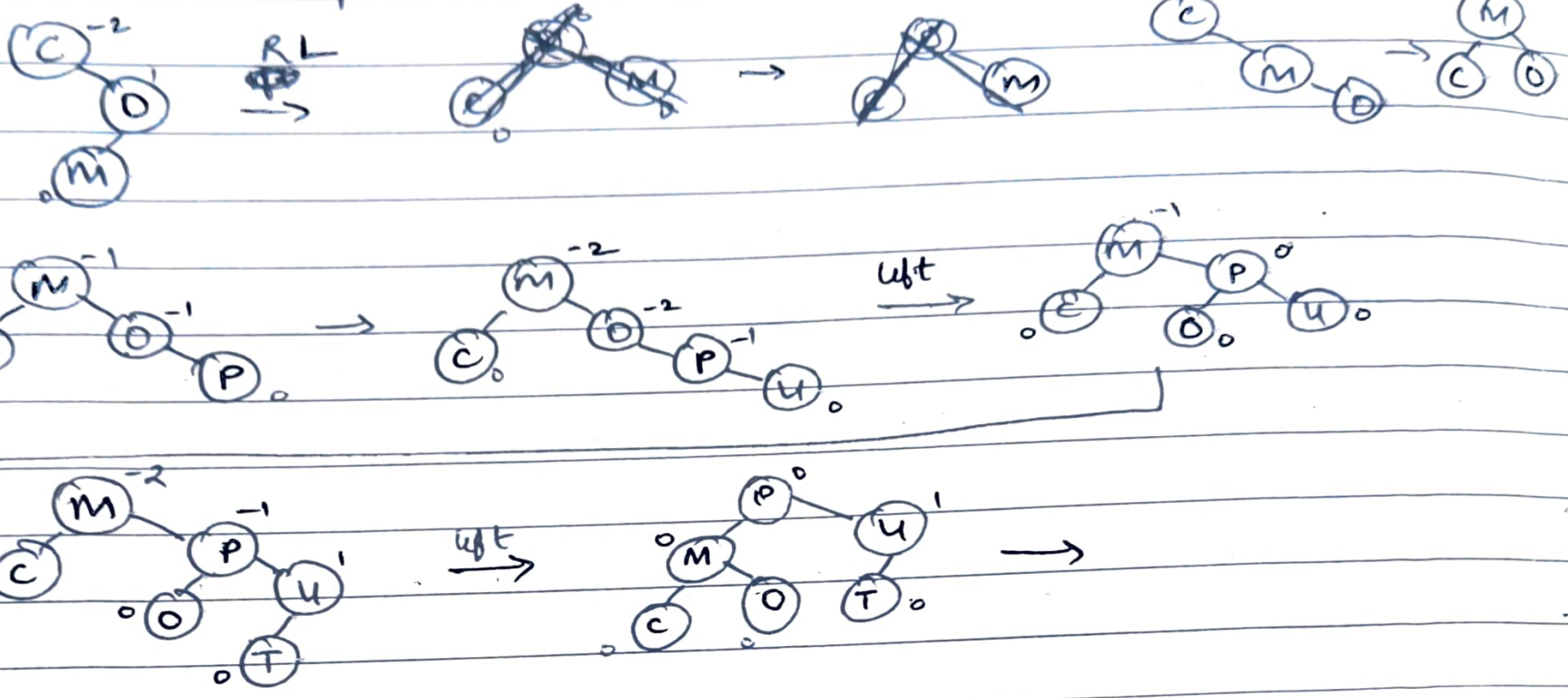
Q Create an AVL tree for characters of word COMPUTER



right →



8 COMPUTING



Splay tree

Splay tree is a self organizing BST with a property that recently accessed elements will be the root node.

Splay tree was invented by Daniel Sleator and Robert Tarjan.

Applications of splay tree are:

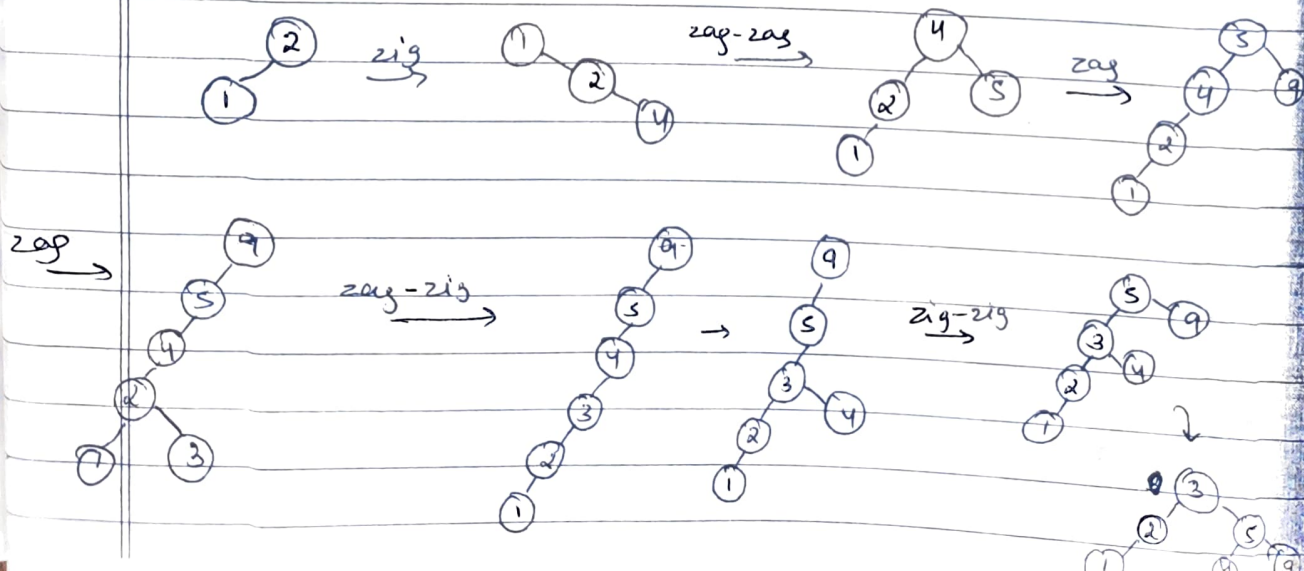
1. Cache memory
2. Virtual memory mgmt
3. Routing tables

To construct a splay we do the following rotations

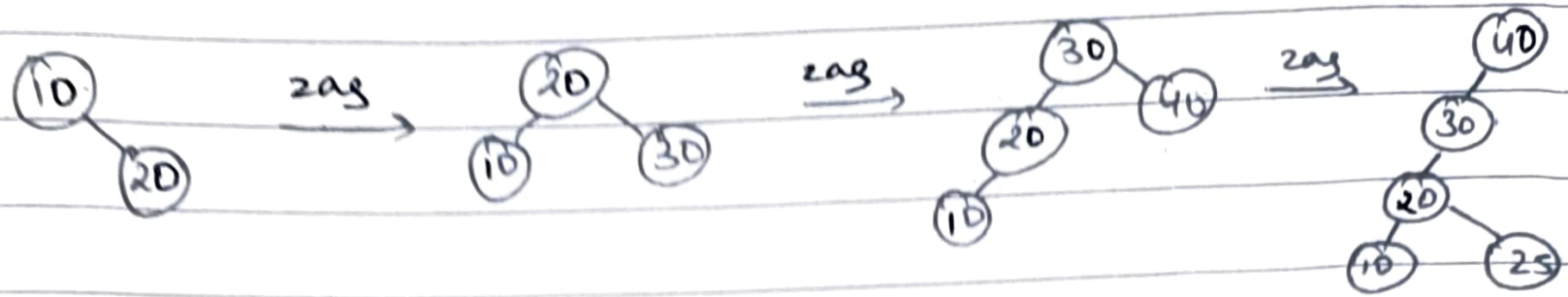
1. Zig (right rotn)
2. Zag (left)
3. Zig - Zag
4. Zag - Zig
5. Zig - Zig
6. Zag - Zag

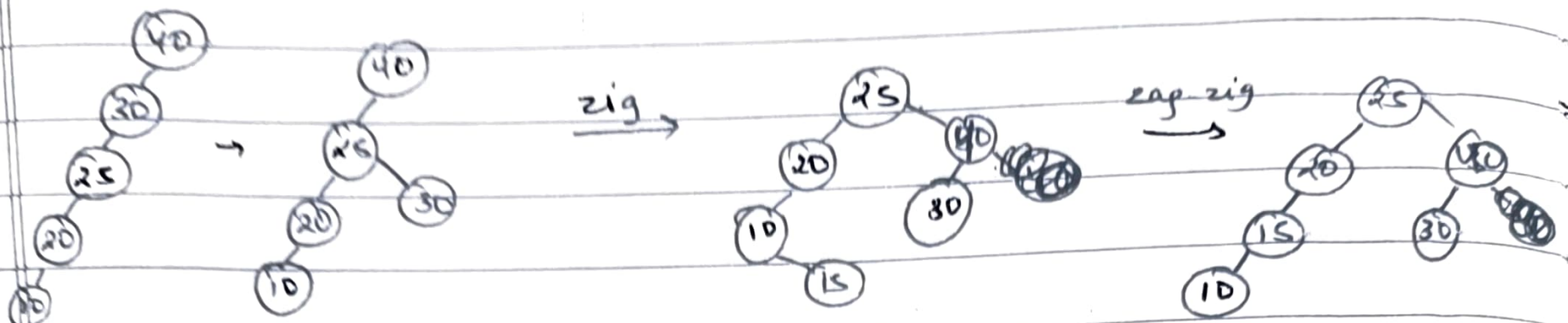
Q Construct a splay for the given elements:

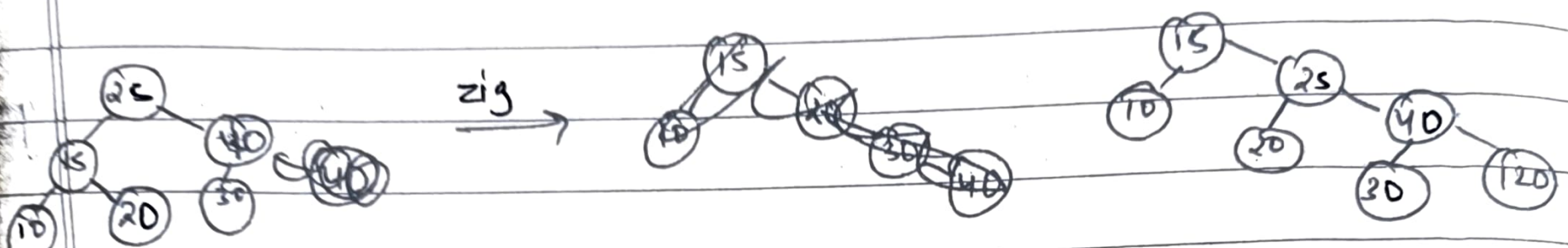
2 1 4 5 9 3



10 20 30 40 25 15 120







taken from word retrieval

TRIE - data structure to implement dictionary

The most popular data structure used to represent a set of strings.

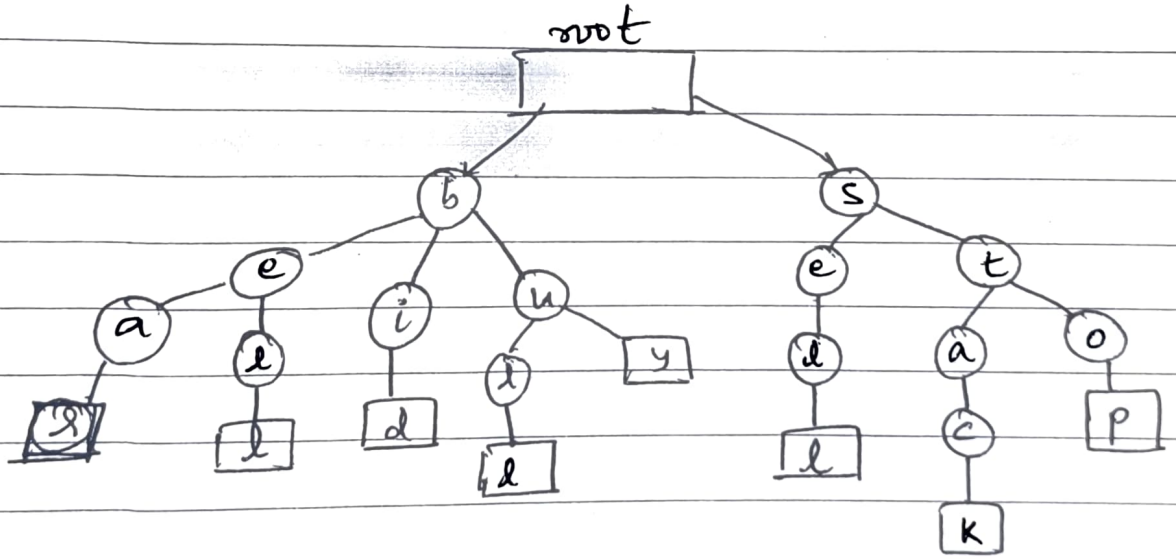
There are 2 types of TRIEs:

1. Standard TRIE
2. Compressed TRIE

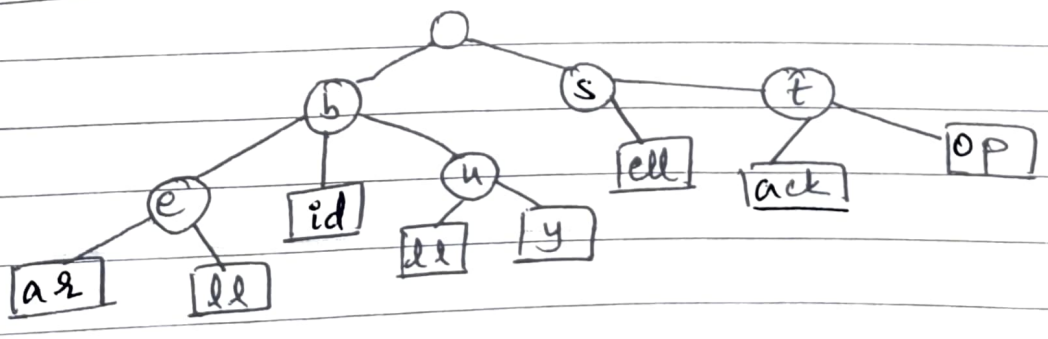
TRIE is a multiway search tree.

eg: TRIE for the words bear, bell, bid, bull, buy, sell, stack, stop.

Standard TRIE.



Compressed trie



Hashing

Hashing is a process of mapping huge amount of data into a smaller (table) with the help of a hash function.
The table which stores the data is called hash table.

To convert the data into a hash value we use the hash function. If h is the hash function and x is the data then the hash function can be denoted as $h(x)$.
A simple hash function can be represented as

$h(x) = x \bmod 10$ where 10 is the size of the hash table.

54 63 25 78 11

$54 \bmod 10 = 4$
$63 \bmod 10 = 3$

0	
1	11
2	
3	63
4	54
5	25
6	
7	
8	78
9	

If x_1 and x_2 are the data to be stored and if $h(x_1) = h(x_2)$
↓
"collision"

eg: $h(x) = x \bmod 10$

$x_1 = 77$ $\quad h(x_1) = 7$

$x_2 = 27$ $\quad h(x_2) = 7$

Collision can be avoided using 2 techniques:

1. open addressing
2. separate chaining.

Open addressing

(1) Linear probing (2) Quadratic probing (3) Double hashing

~~$h(x) = x \bmod 10$~~

(1) Linear probing

search for next free slot

$h(x) = x \bmod 10$

0	19
1	28
2	39
3	
4	
5	55
6	75
7	
8	78
9	89

78
89
19
28
39
55
75

$i = 0, 1, 2, 3, 4$

$h(2) = (x + i) \bmod \text{SIZE}$ (10)

$0 \rightarrow 19$
$1 \rightarrow 28$
$2 \rightarrow 39$
$3 \rightarrow 0$

Quadratic probing

SIZE-1

$i = 0, 1, 2 \ldots n-1$

$h_i(x) = (h(x) + i^2) \bmod 10$

$h(x) = x \bmod 10$

78
89
19
28
39

0	19
1	
2	28
3	39
4	
5	
6	
7	
8	78
9	89

~~$h_0(78) = 8 \bmod 10 = 8$~~

~~$h_1(89) = (9+1) \bmod 10 = 0$~~

19 collides

so (1) $(9+1) \bmod 10 = 0$

28 collides

(1) ~~$(8+1) \bmod 10 = 9$~~ ✗

$(8+4) \bmod 10 = 2$

39 collides

(1) $9+1 \bmod 10 = 0$ ✗

$(9+4) \bmod 10 = 3$

(3) Double hashing

We use 2 hash functions

$h(x) = x \bmod 10 + (x+5) \bmod 10$

Separate chaining - uses concept of L.L to avoid collision.

$h(x) = x \bmod 10$

78 89 19 28 39 54 63 70 44