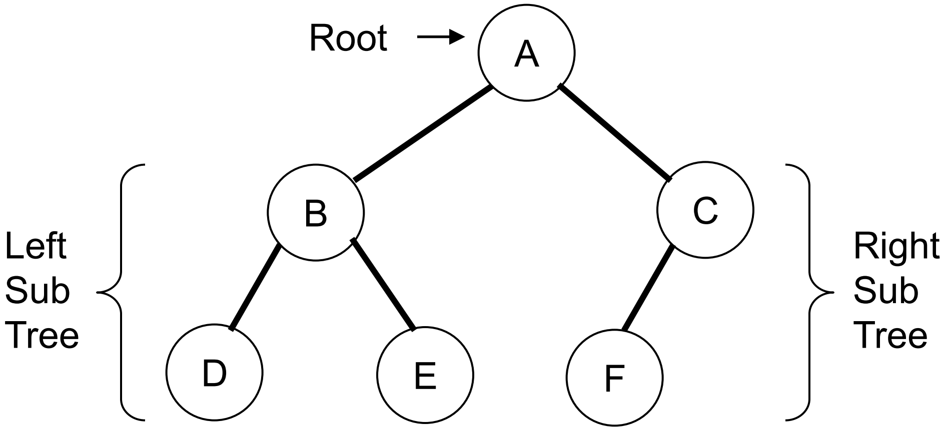
Tree – a Hierarchical Data Structure

Trees are non linear data structure that can be represented in a hierarchical manner.

- A tree contains a finite non-empty set of elements.
- Any two nodes in the tree are connected with a relationship of parent-child.
- Every individual elements in a tree can have any number of sub trees.



Binary Tree

-- A Tree is said to be a binary tree that every element in a binary tree has at the most two sub trees. (means zero or one or two)

Types of binary trees

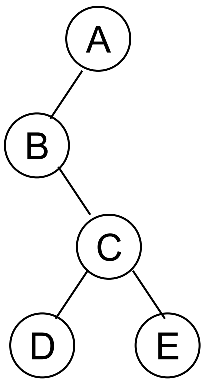
-- Strictly binary tree
-- Complete binary tree
-- Extended binary tree

Tree – Terminology.

- **Root :** The basic node of all nodes in the tree. All operations on the tree are performed with passing root node to the functions.(A – is the root node in above example.)
- **Child :** a successor node connected to a node is called child. A node in binary tree may have at most two children. (B and C are child nodes to the node A, Like that D and E are child nodes to the node B.)
- **Parent :** a node is said to be parent node to all its child nodes. (A is parent node to B,C and B is parent node to the nodes D and F).
- **Leaf :** a node that has no child nodes. (D, E and F are Leaf nodes)
- **Siblings :** Two nodes are siblings if they are children to the same parent node.
- **Ancestor :** a node which is parent of parent node (A is ancestor node to D,E and F).
- **Descendent :** a node which is child of child node (D, E and F are descendent nodes of node A)
- **Level** : The distance of a node from the root node, The root is at level – 0,(B and C are at Level 1 and D, E, F have Level 2 (highest level of tree is called **height** of tree)
- **Degree :** The number of nodes connected to a particular parent node.

Representation of binary tree.

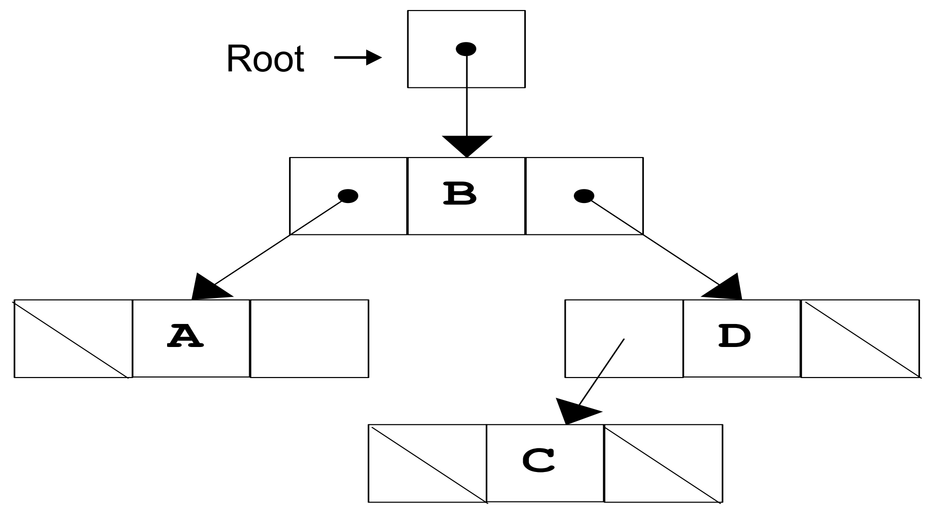
- **Sequential Representation :**
  - -- Tree Nodes are stored in a linear data structure like array.
  - -- Root node is stored at index ‘0’
  - -- If a node is at a location ‘i’, then its left child is located at 2 * i + 1 and right child is located at 2 * i + 2
  - -- This storage is efficient when it is a complete binary tree, because a lot of memory is wasted.



A	B	-	C	-	-	-	D	E	-	-					

```
/* a node in the tree structure */
struct node {
    struct node *lchild;
    int data ;
    struct node *rchild;
};
```

-- The pointer lchild stores the address of left child node.
-- The pointer rchild stores the address of right child node.
-- if child is not available NULL is strored.
-- a pointer variable root represents the root of the tree.



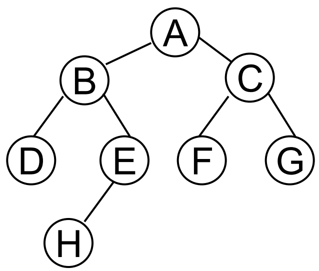
Tree structure using Doubly Linked List

Implementing Binary Tree

```
struct node {
  struct node *lchild;
  char data;
  struct node *rchild;
} *root=NULL;
struct node *insert(char a[],int index) {
   struct node *temp=NULL;
   if(a[index]!='\0')  {
      temp = (struct node *)malloc(
                        sizeof(struct node));
      temp->lchild = insert(a,2*index+1);
      temp->data = a[index];
      temp->rchild = insert(a,2*index+2);
   }
   return temp;
}
void buildtree(char a[])  {
   root = insert(a,0);
}
void inorder(struct node *r){
   if(r!=NULL) {
     inorder(r->lchild);
     printf("%5c",r->data);
     inorder(r->rchild);
   }
}
void preorder(struct node *r){
   if(r!=NULL) {
     printf("%5c",r->data);
     preorder(r->lchild);
     preorder(r->rchild);
   }
}
```

```
void postorder(struct node *r){
   if(r!=NULL) {
     postorder(r->lchild);
     postorder(r->rchild);
     printf("%5c",r->data);
   }
}
void main()
{
   char arr[] = { 'A','B','C','D','E','F','G','\0','\0','H','\0',
               '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0'};
   buildtree(arr);
   printf("\nPre-order Traversal : \n");
   preorder(root);
   printf("\nIn-order Traversal : \n");
   inorder(root);
   printf("\nIn-order Traversal : \n");
   inorder(root);
}
```

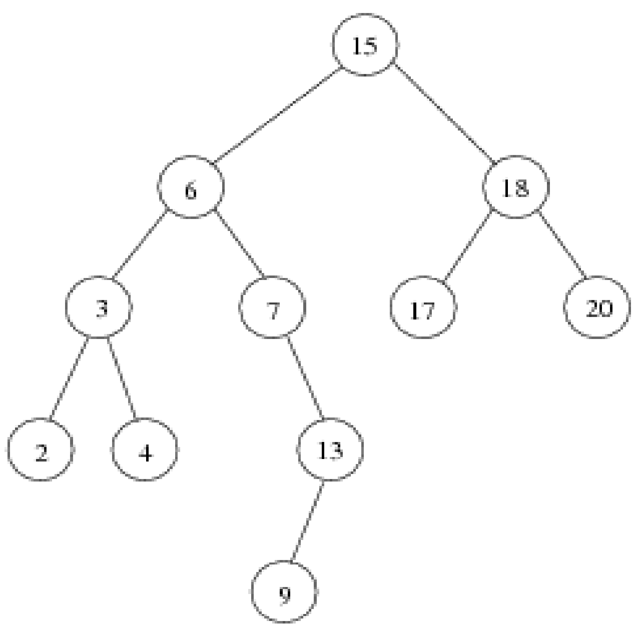
Tree Created :



Output :
Preorder Traversal : A B D E H C F G
Inorder Traversal : D B H E A F C G
Postorder Traversal : D H E B F G C A

Binary Search Tree

Binary search tree
- Values in left subtree less than parent
- Values in right subtree greater than parent
- Facilitates duplicate elimination
- Fast searches - for a balanced tree, maximum of log n comparisons
- Inorder traversal – prints the node values in ascending order



Tree traversals:

Inorder traversal – prints the node values in ascending order
1. Traverse the left subtree with an inorder traversal
2. Process the value in the node (i.e., print the node value)
3. Traverse the right subtree with an inorder traversal

Preorder traversal
1. Process the value in the node
2. Traverse the left subtree with a preorder traversal
3. Traverse the right subtree with a preorder traversal

Postorder traversal
1. Traverse the left subtree with a postorder traversal
2. Traverse the right subtree with a postorder traversal
3. Process the value in the node

Traversals of Tree
- In Order Traversal (Left – Origin – Right): 2 3 4 6 7 9 13 15 17 18 20
- Pre Order Traversal (Origin – Left – Right) : 15 6 3 2 4 7 13 9 18 17 20
- Post Order Traversal (Left – Right – Origin) : 2 4 3 9 13 7 6 17 20 18 15

Implementing Binary Search Tree

```
struct node {
   struct node *lchild;
   int data;
   struct node *rchild;
};
struct node *fnode=NULL,*parent=NULL;
void insert(struct node **p, int n) {
   if(*p==NULL)    {
     *p=(struct node *)malloc(sizeof(struct node));
     (*p)->lchild=NULL;
     (*p)->data = n;
     (*p)->rchild=NULL;
     return;
   }
   if(n < (*p)->data)
        insert(&((*p)->lchild),n);
   else
        insert(&((*p)->rchild),n);
}
void inorder(struct node *t) {
   if(t!=NULL)   {
     inorder(t->lchild);
     printf("%5d",t->data);
     inorder(t->rchild);
   }
}
void preorder(struct node *t)  {
   if(t!=NULL)    {
     printf("%5d",t->data);
     preorder(t->lchild);
     preorder(t->rchild);
   }
}
void postorder(struct node *t)  {
   if(t!=NULL)    {
     postorder(t->lchild);
     postorder(t->rchild);
     printf("%5d",t->data);
   }
}
void search(struct node *r,int key){
    struct node *q;
,   fnode=NULL;    q=r;
    while(q!=NULL)  {
        if(q->data == key) {
            fnode = q; return;
        }
        parent=q;
        if(q->data > key) q=q->lchild;
        else   q=q->rchild;
    }
}
void delnode ( struct node **r,int key)  {
    struct node *succ,*fnode,*parent;
    if(*r==NULL)  {
      printf("Tree is empty"); return;
    }
    parent=fnode=NULL;
```

Implementing Binary Search Tree (continued)

```
search(*r,key);
if(fnode==NULL) {
  printf("\nNode does not exist, no deletion");
  return;
}
if(fnode->lchild==NULL && fnode->rchild==NULL) {
     if(parent->lchild==fnode)
          parent->lchild=NULL;
     else
          parent->rchild=NULL;
     free(fnode);
     return;
}
if(fnode->lchild==NULL && fnode->rchild!=NULL) {
    if(parent->lchild==fnode)
       parent->lchild=fnode->rchild;
    else
       parent->rchild=fnode->rchild;
    free(fnode);
    return;
}
if(fnode->lchild!=NULL && fnode->rchild==NULL) {
    if(parent->lchild==fnode)
       parent->lchild=fnode->lchild;
    else
       parent->rchild=fnode->lchild;
    free(fnode);
    return;
}
```

```
if(fnode->lchild!=NULL && fnode->rchild!=NULL)  {
   parent = fnode;
   succ = fnode->lchild;
   while(succ->rchild!=NULL){
       parent=succ;
       succ = succ->rchild;
   }
   fnode->data = succ->data;
   free(succ);
   succ=NULL;
   parent->rchild = NULL;
 }
}
void inorder(struct node *t)  {
   if(t!=NULL)
   {
     inorder(t->lchild);
     printf("%5d",t->data);
     inorder(t->rchild);
   }
}
void preorder(struct node *t)
 {
   if(t!=NULL)  {
     printf("%5d",t->data);
     preorder(t->lchild);
     preorder(t->rchild);
   }
}
```

Implementing Binary Search Tree (continued)

```
void postorder(struct node *t) {
   if(t!=NULL)   {
     postorder(t->lchild);
     postorder(t->rchild);
     printf("%5d",t->data);
   }
}
int main()  {
   struct node *root=NULL;
   int n,i,num,key;
   printf("Enter no of nodes in the tree : ");
   scanf("%d",&n);
   for(i=0;i<n;i++)  {
      printf("Enter the element : ");
      scanf("%d",&num);
      insert(&root,num);
   }
   printf("\nPreorder traversal :\n");
   preorder(root);
   printf("\nInorder traversal :\n");
   inorder(root);
   printf("\nPostorder traversal :\n");
   postorder(root);
  printf("\nEnter the key of node to be searched : ");
   scanf("%d",&key);
   search(root,key);
   if(fnode==NULL)
      printf("\nNode does not exist");
   else  printf("\nNOde exists");
```

```
      printf("\nEnter the key of node to be deleted : ");
      scanf("%d",&key);
      delnode(&root,key);
      printf("\nInorder traversal after deletion :\n");
      inorder(root);
   }
```

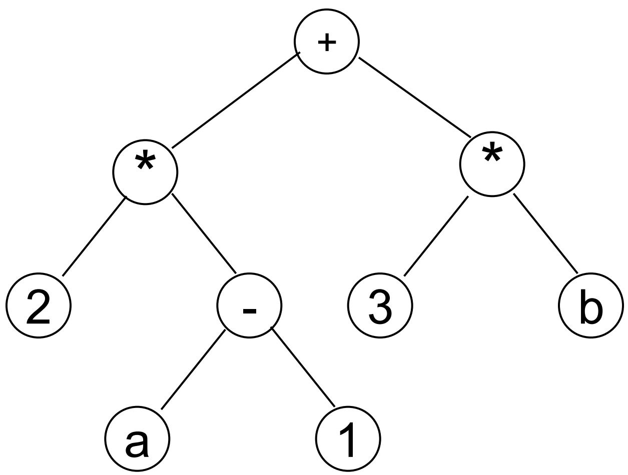
```
Output :
Enter no of nodes in the tree : 5
Enter the element : 18
Enter the element : 13
Enter the element : 8
Enter the element : 21
Enter the element : 20

Preorder traversal :
   18   13    8   21   20
Inorder traversal :
    8   13   18   20   21
Postorder traversal :
    8   13   20   21   18
Enter the key of node to be searched : 13

Node exists
Enter the key of node to be deleted : 20

Inorder traversal after deletion :
    8   13   18   21
```

Arithmetic Expression Tree

Binary Tree associated with Arithmetic expression is called Arithmetic Expression Tree

- All the internal nodes are operators
- All the external nodes are operands
- When Arithmetic Expression tree is traversed in in-order, the output is In-fix expression.
- When Arithmetic Expression tree is traversed in post-order, We will obtain Post-fix expression.

Example: Expression Tree for (2 * (a − 1) + (3 * b))



In-Order Traversal of Expression Tree Results In-fix Expression.

2 * (a – 1) + (3 * b)

Post-Order Traversal of Expression Tree Results Post-fix Expression.

2 a 1 – * 3 b * +

To convert In-fix Expression to Post-fix expression, First construct an expression tree using infix expression and then do tree traversal in post order.

Graphs

A Tree is in fact a special type of graph. Graph is a data structure having a group of nodes connecting with cyclic paths.

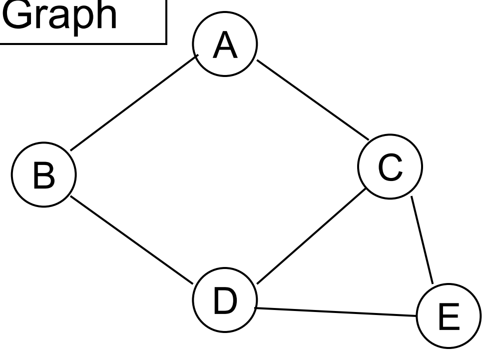
A Graph contains two finite sets, one is set of nodes is called vertices and other is set of edges.

A Graph is defined as G = (V, E) where.

i) V is set of elements called nodes or vertices.

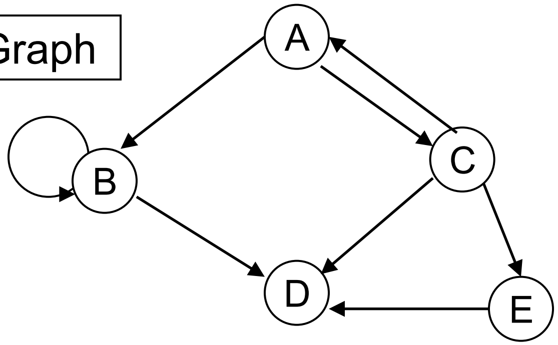
ii) E is set of edges, identified with a unique pair (v1, v2) of nodes. Here (v1, v2) pair denotes that there is an edge from node v1 to node v2.

Un-directed Graph



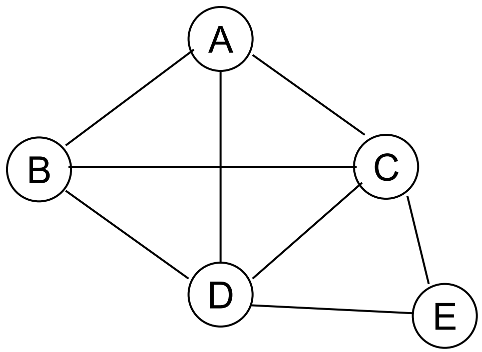
Set of vertices = { A, B, C, D, E }
Set of edges = { (A, B) , (A, C) , (B, D), (C, D),(C, E) , (D, E) }
A graph in which every edge is undirected is known as Un directed graph.
The set of edges are unordered pairs.
Edge (A, B) and (B, A) are treated as same edge.

Directed Graph



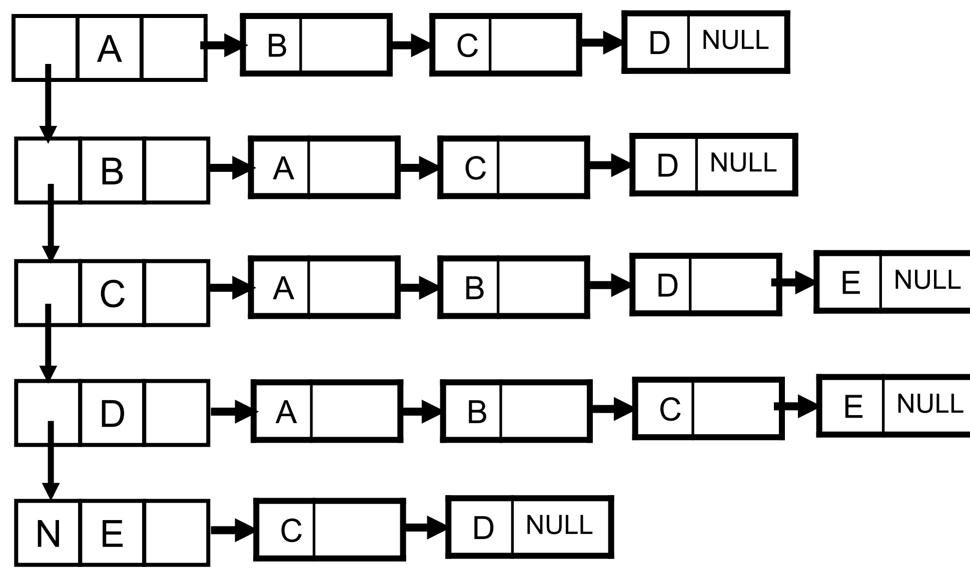
Set of vertices = { A, B, C, D, E }
Set of edges = { (A, B) , (A, C) , (B, B), (B, D), (C, A), (C, D), (C, E) , (E, D) }
A graph in which every edge is directed is known as Directed graph.
The set of edges are ordered pairs.
Edge (A, B) and (B, A) are treated as different edges.
The edge connected to same vertex (B, B) is called loop.
Out degree : no of edges originating at a given node.
In degree : no of edges terminating at a given node.
For node C - In degree is 1 and out degree is 3 and total degree is 4.

Representation of Graph



Set of vertices = { A, B, C, D, E }
Set of edges = { (A, B) , (A, C) , (A, D), (B, D), (B, D),
(C, D),(C, E) , (D, E) }
There are two ways of representing a graph in memory.
i) Sequential Representation by means of Adjacency Matrix.
ii) Linked Representation by means of Linked List.

Linked List



Adjacency Matrix

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	1	0
C	1	1	0	1	1
D	1	1	1	0	1
E	0	0	1	1	0

Adjacency Matrix is a bit matrix which contains entries of only 0 and 1
The connected edge between to vertices is represented by 1 and absence of edge is represented by 0.
Adjacency matrix of an undirected graph is symmetric.