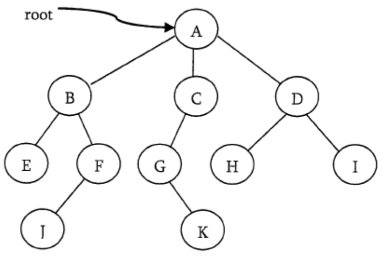
Binary Trees

Data structures

- Data structure is a specialized format for organizing and storing data.
- Types:
  1. Linear data structures: elements are accessed in a sequential order
  2. Non-linear data structures: elements are stored/accessed in a non-linear order.

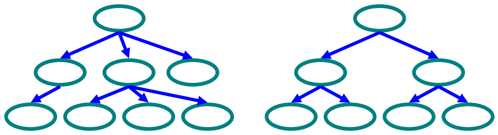
Tree

- Non-linear data structures
- Each node points to a number of nodes.
- Representing the hierarchical nature of a structure in a graphical form.



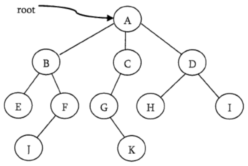
Hierarchical Data Structures

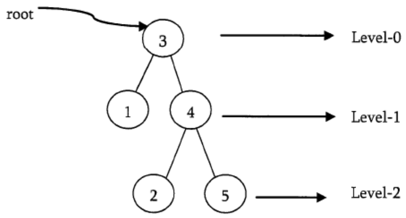
- One-to-many relationship between elements
- Tree
  - Single parent, multiple children
- Binary tree
  - Tree with 0–2 children per node



Tree Binary Tree

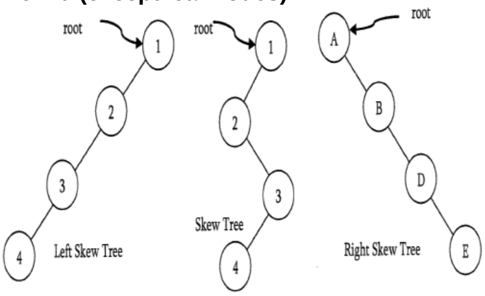
- Root: node with no parents
- Edge: link from parent to child
- Leaf: node with no children
- Depth: depth of a node is the length of the path from the root node to the leaf.
- Level: set of all nodes at a given depth. Root (level-0)





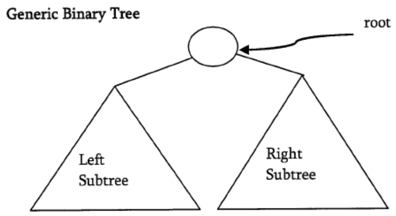
- Height (node): length of the path from the node to the deepest node.
- Height (tree): MAXIMUM height among all the nodes in the tree
- Size: size of a node is the number of descendents it has including itself.

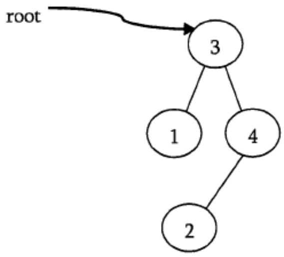
- Skew **trees: if every node of a tree has only one child (except leaf nodes)**



Binary tree

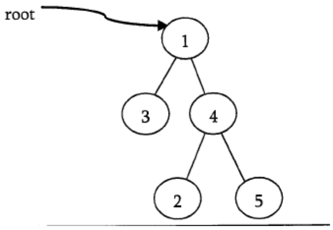
- **A tree is called as binary tree if each node has zero child, 1 child, 2 childs.**



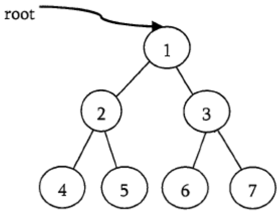


Binary tree types

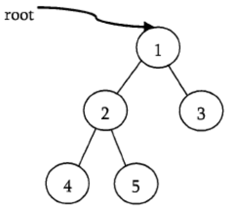
- **Strict binary tree: if each node has exactly 2 or no children.**



- Full binary tree: each node has exactly 2 children and all leaf nodes are at same level.



- Complete binary tree:if all leaf nodes are at height h or h-1

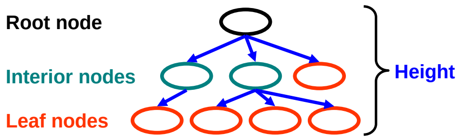


Properties of binary tree

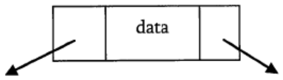
- Number of nodes in a full binary tree is 2^{h+1} -1
- Complete binary tree: 2^h to 2^{h+1} -1
- Leaf nodes (full binary tree): 2^h

Trees

- Terminology
  - Root ⇒ no predecessor
  - Leaf ⇒ no successor
  - Interior ⇒ non-leaf
  - Height ⇒ distance from root to leaf



Structure of binary tree




```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
};
```


Operations on Binary tree

Basic Operations

- Inserting an element in to a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding size of the tree
- Finding the height of the tree
- Finding the level which has maximum sum
- Finding least common ancestor (LCA) for a given pair of nodes and many more.

Applications of Binary Trees

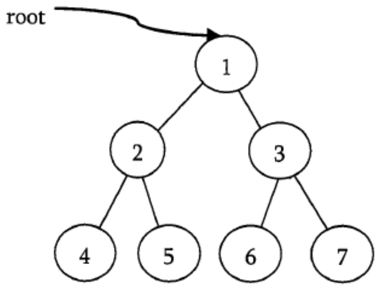
- **Expressions tree in compliers**
- **Binary Search Tree**
- **Priority trees**
- **Hoffman coding trees**

Binary Tree traversals

- **Process of visiting all nodes of a tree is known as tree traversal.**

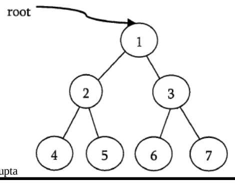
Traversal possibilities

- Current node/root (D)
- Left child node (L)
- Right child node (R)
- Traversal possibilities
  - LDR (Inorder traversal)
  - DLR (Preorder traversal)
  - LRD (Postorder traversal)

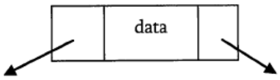


Preorder traversal

- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.
- 1 2 4 5 3 6 7



Structure of binary tree

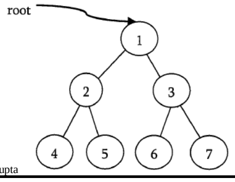



```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
};
```

```
void PreOrder ( struct BinaryTreeNode * root)
{
  if (root)
      {     printf (“%d”, root➔data);
            PreOrder (root➔left);
            PreOrder (root➔right);
      }
}
```


Inorder traversal

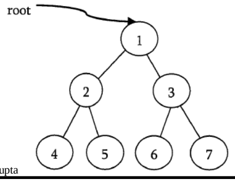
- Traverse the left subtree in Inorder.
- Visit the root.
- Traverse the right subtree in Inorder.
- 4 2 5 1 6 3 7




```
void InOrder ( struct BinaryTreeNode * root)
{
  if (root)
      {     InOrder (root➔left);
            printf (“%d”, root➔data);
            InOrder (root➔right);
      }
}
```


PostOrder traversal

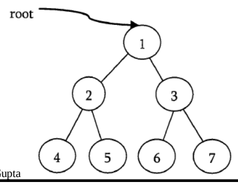
- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.
- 4 5 2 6 7 3 1



```
void PostOrder ( struct BinaryTreeNode * root)
{
   if (root)
        {       PostOrder (root➔left);
                PostOrder (root➔right);
                printf (“%d”, root➔data);
        }
}
```


Level order traversal

- Visit the root.
- While traversing level l, keep all the elements at level l+1 in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.
- 1 2 3 4 5 6 7



Binary tree array based implementation

```
#include<stdio.h>
int arr[100],count;
main()
{  int i,num,choice;
    count = 0;
    for(i=0;i<100;i++)
       arr[i] = 0;
    do
    {
       printf("enter your choice \n 1.Insert into tree \n 2.delete from tree \n 3. search for an element in tree \n 4. inorder traversal\n5. exit\n");
       scanf("%d",&choice);
```



```
switch(choice)
{
     case 1:
          printf("enter element");
          scanf("%d",&num);
          arr[count] = num;
          count++;
          break;
```

```
case 2:
    printf("\n enter the element to be deleted");
    scanf("%d",&num);
    for(i=0;i<count;i++)
    {
       if(arr[i]==num)   {
         count--;
         arr[i] = arr[count];
         arr[count] = 0;
         break;}
    }
    if(i==count)
    printf("\n element not found");
    break;
```

Bharat Gupta 33

```
case 3:
    printf("\n enter the element to be searched");
    scanf("%d",&num);
    for(i=0;i<count;i++)
    {  if(arr[i]==num)   {
              printf("\n element found");
              break; }
    }
    if(i==count)
       printf("\n element not found");
    break;
```

Bharat Gupta 34

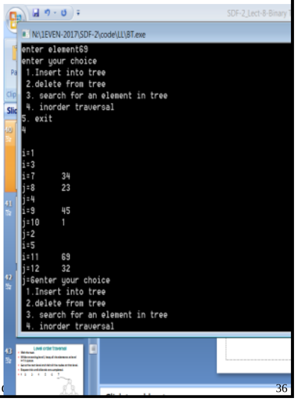
```
  case 4:
       inorder(0);
       break;
  }
 }while(choice != 5);

}
```

Bharat Gupta 35

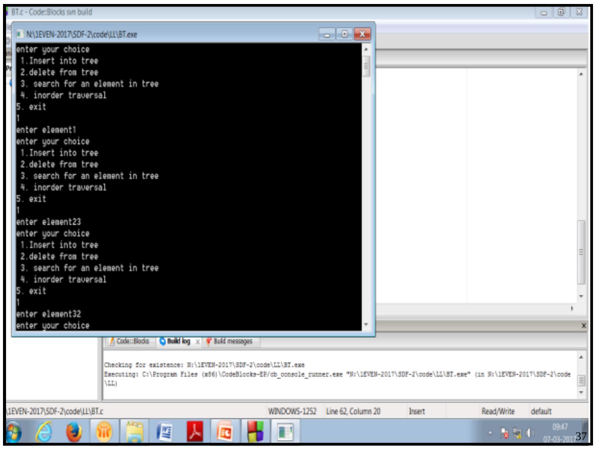
Binary tree array based implementation

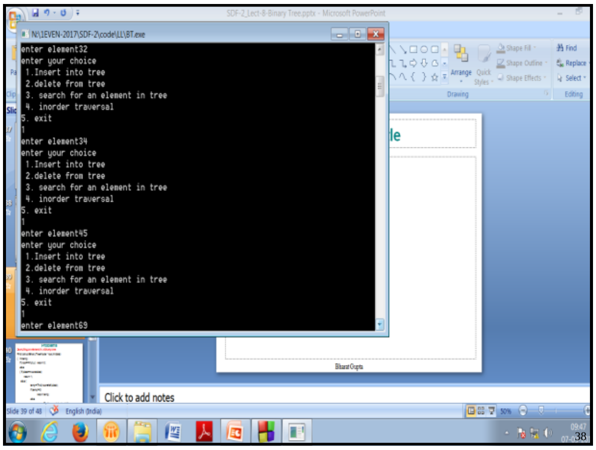
```
void inorder(int pos)
    { int i,j;
       i = 2*pos + 1;
    print("\ni=%d",i);
    if(arr[i] != 0)
            inorder(i);
       printf("\t %d",arr[pos]);
       j = 2*pos +2;
    print("\nj=%d",j);
    if(arr[j] != 0)
           inorder(j);
    }
```

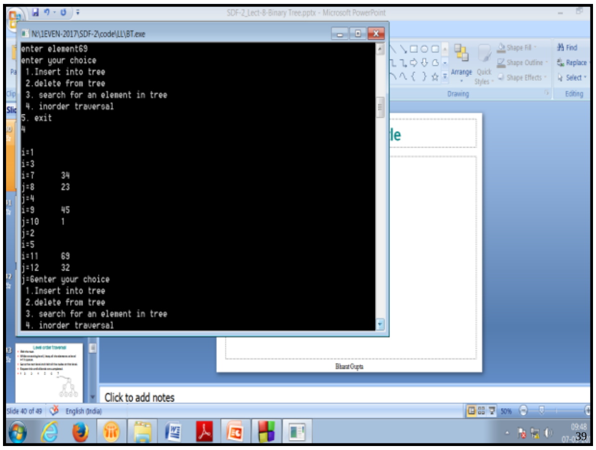


4 2 5 1 6

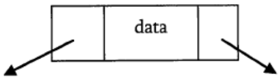
Bharat Gupta 36







Structure of binary tree




```
struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
};
```

Problems

Searching an element in a binary tree

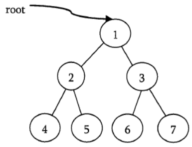
```
Find (struct BinaryTreeNode * root, int data)
{   int temp;
    if (root==NULL)    return 0;
    else
    { if (data==root➔data)
          return 1;
      else {
                temp= Find(root➔left, data);
                if (temp!=0)
                        return temp;
                else
                        return Find(root➔right, data);} }
          return 0;}
```


Deleting a binary tree (post-order traversal)

```
void deletebt(struct binarytreenode *root) {
If (root==NULL)
   return ;
deletebt(root➔left);
deletebt(root➔right);
free(root);
}
```


Level order traversal

- Visit the root.
- While traversing level l, keep all the elements at level l+1 in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.
- 1 2 3 4 5 6 7



Level-order traversal

```
void levelorder( struct binarytree *root) {
struct binarytree *temp;
struct queue *Q= createqueue();
If(!root)
   return;
enqueue(Q,root);
```

```
while(!isemptyqueue (Q)){
    temp=dequeue(Q);
    printf("%d", temp➔data);
    if (temp➔left)
      enqueue(Q, temp➔left);
    if (temp➔right)
      enqueue(Q, temp➔right);
    }
    deletequeue(Q);
    }
```


Insert in a binary tree (level-order traversal)

```
void insertinbinarytree( struct BTnode * root, int data){
    struct Queue *Q;
    struct BTnode *temp;
    struct BTnode   *newnode;
    newnode=(struct BTnode *) malloc (sizeof (struct
      BTnode));
```



```
if(!newNode) {
        printf("Memory Error");
        return;
}
if(!root) {
        root = newNode;
        return;
}
Q = CreateQueue();
EnQueue(Q,root);
```



```
while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(temp→left)
                EnQueue(Q, temp→left);
        else {  temp→left=newNode;
                DeleteQueue(Q);
                return;
        }
if(temp→right)
        EnQueue(Q, temp→right);
else {  temp→right=newNode;
        DeleteQueue(Q);
        return;
}
```

```
        }
        DeleteQueue(Q);
}
```