

GROUP B

Assignment 03

Problem Statement:

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

Objectives:

1. To understand concept of Tree.
2. To understand to find the record of book which consist of no of chapters, sections and subsections.
3. To understand concept & features of creating nodes in object oriented programming.

Learning Objectives

To understand concept of tree.

To understand operations like insert nodes in tree.

Learning Outcome

Learn object oriented Programming features

Understand & implement concept of creating nodes with insert and display operation on it.

Theory

A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.
2. Each node has one parent only but can have multiple children.
3. Each node is connected to its children via edge.

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the “children”).

Basic Terminology In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree. The degree of the node {3} is 3.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the parent nodes of the node {7}
- **Descendant:** Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.
- **Sibling:** Children of the same parent node are called siblings. {8, 9, 10} are called siblings.
- **Depth of a node:** The count of edges from the root to the node. Depth of node {14} is 3.
- **Height of a node:** The number of edges on the longest path from that node to a leaf. Height of node {3} is 2.
- **Height of a tree:** The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is 3.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

Properties of Binary Tree

- At each level of i , the maximum number of nodes is 2^i .
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to $(1+2+4+8) = 15$. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to $h+1$.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right

pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
1. struct node
2. {
3.     int data,
4.     struct node *left, *right;
5. }
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Software Required: Dev C++ compiler- / 64 bit windows


Input: Book name, chapter name, section name, and subsection name.

Output: Create root node as book name having child chapters, sections and subsections.

Conclusion: This program gives us the knowledge of create and display nodes in tree.

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features for tree. 

GROUP B

Assignment 04

Problem Statement:

Construct an expression tree from the given prefix expression eg. $+-a*bc/def$ and traverse it using post order traversal (non recursive) and then delete the entire tree.

Objectives:

1. To understand concept of expression tree.
2. To understand the concept of converting the given expression into prefix and postfix order.
3. To understand concept & features of creating expression tree in object oriented programming.

Learning Objectives

To understand concept of expression tree.

To understand operations like create expression tree, delete tree.

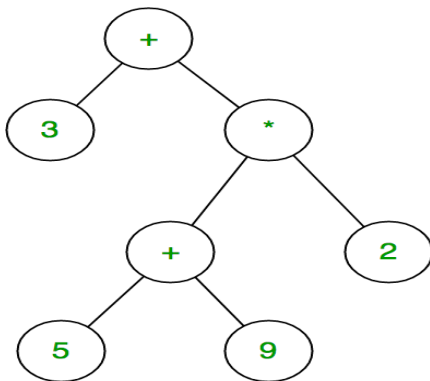
Learning Outcome

Learn object oriented Programming features

Understand & implement concept of creating creating expression tree from given expression, delete tree

Theory

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for $3 + ((5+9)*2)$ would be:



Evaluating the expression represented by an expression tree:

Let t be the expression tree

If t is not null then

 If t.value is operand then

 Return t.value

 A = solve(t.left)

 B = solve(t.right)

 // calculate applies operator 't.value'

 // on A and B, and returns value

 Return calculate(A, B, t.value)

Construction of Expression Tree:

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

Prefix to Postfix step by step

- Scan the given prefix expression from **right to left** character by character.
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.

Concatenate this operator with these two values (**operator+1st top value+2nd top value**) to get a new string.

- Now push this resulting string back into the stack.
- Repeat this process until the end of prefix expression. Now the value in the stack is the desired postfix expression.

How to convert Postfix to Prefix?

- Scan the given postfix expression from **left to right** character by character.
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.

Concatenate this operator with these two values (**operator+2nd top value+1st top value**) to get a new string.

- Now push this resulting string back into the stack.
- Repeat this process until the end of postfix expression. Now the value in the stack is the desired prefix expression.

Software Required: Dev C++ compiler- / 64 bit windows

Input: Expression in prefix order.

Output: Convert the given prefix expression into tree and write the post order traversal (non recursive) and then delete the entire tree.

Conclusion: This program gives us the knowledge of create and display expression tree.

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features for tree.

GROUP B

Assignment 05

Problem Statement:

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -

- i. Insert new node
- ii. Find number of nodes in longest path from root
- iii. Minimum datavalue found in the tree
- iv. Change a tree so that the roles of the left and right pointers are swapped at every node
- v. Search a value

Objectives:

1. To understand concept of binary search tree .
2. To understand the concept of Insert new node, Find number of nodes, Minimum datavalue found in the tree

Learning Objectives

To understand concept of binary search tree

Learning Outcome

Learn object oriented Programming features

Understand & implement concept of Insert new node, Find number of nodes, Minimum datavalue found in the tree

Theory

A node in Binary Search tree will be represented as follows:

Left	Data	Right
------	------	-------

- Declare a structure to represent a node in the binary tree
- ```
Struct BinTree
```

```
{

 struct BinTree *left;

 char data;
```

```

 struct BinTree *right;

};

```

- Display a menu with following options:
  1. Recursive Create
  2. Non – recursive create
  3. Insert New Node
  4. Find Height of the tree
  5. Smallest node value in the tree
  6. Mirror Image of the tree
  7. Search Value
- Read the choice & switch according to that
- If the choice is 1 or 2 create the root node & then give call to create function
- Otherwise pass the root node as the input parameter to the function

#### **Rcreate ( ) function:**

- Main ( ) function has created the root node.
- Display the data for root node.
- Ask user if the new node is to be added to the left.
- If choice is yes
  - Create a new node
  - Read the data for new node
  - Initialize left & right pointers of the new node to NULL.
  - Attach this new node to the left of the root
  - Give recursive call to Rcreate ( ) function with root->left as the new root.
- Display the data for root node.
- Ask user if the new node is to be added to the right.
- If choice is yes
  - Create a new node
  - Read the data for new node
  - Initialize left & right pointers of the new node to NULL.
  - Attach this new node to the right of the root
  - Give recursive call to Rcreate ( ) function with root->right as the new root.
- End of Rcreate()

#### **Nrcreate ( ) function**

- Main ( ) function has created the root node.
  - Let temp & new node be two node structures
- ```

while(1)

```

```

begin

```



```

initialize temp to point to Root
create a new node
initialize left & right pointer of new node to NULL
accept data for new node
while(1)
begin
    If newnode->data < temp->data
        If temp->left != NULL then
            temp = temp->left
        else
            temp->left = newnode
        break
    If newnode->data > temp->data
        If temp->right != NULL then
            temp = temp->right
        else
            temp->right = newnode
        break

    End

Ask user if more nodes are to be added to the tree
If the „no“ break;
End
➤ End of Nrcreate( )

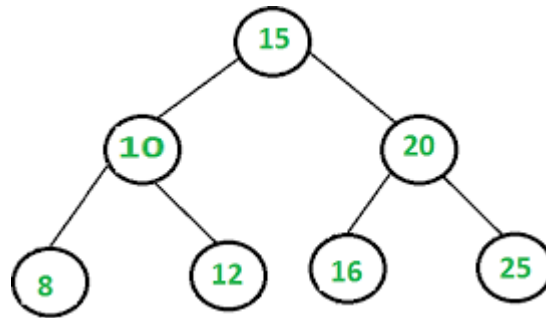
```

Find Height of the tree-

There are two conventions to define height of Binary Tree

- 1) Number of nodes on longest path from root to the deepest node.
- 2) Number of edges on longest path from root to the deepest node.

In this post, the first convention is followed. For example, height of the below tree is 3.



Recursive method to find height of Binary Tree is discussed [here](#). How to find height without recursion? We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level; stop traversing when count of nodes at next level is 0.

Recursive Function-

```
int btree::nheight(node *root)
{
    int i, j, max=0;
    i=1,j=1;
    if(root!=NULL)
    {
        i=i+nheight(root->left);
        j=j+nheight(root->right);
        if(i>j)
            max=i;
        else
            max=j;
    }
}
```

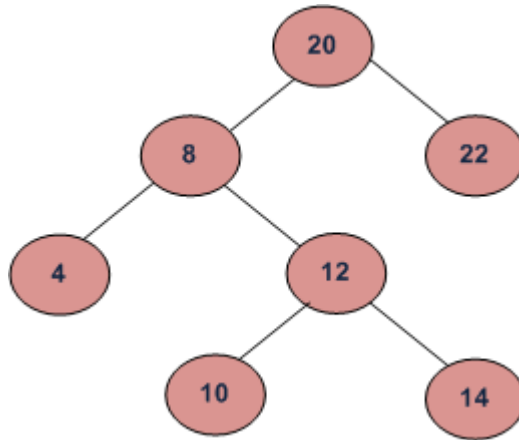
```
        return(max);  
    }  
}
```

Non-recursive function-

```
int BinaryTree :: TreeHeight(TreeNode *Root)  
{  
    int heightL, heightR;  
    if(Root == Null)  
        return 0;  
    if(Root->Lchild == Null && Root->Rchild == Null)  
        return 0;  
    heightL = TreeHeight(Root->Lchild);  
    heightR = TreeHeight(Root->Rchild);  
    if(heightR > heightL)  
        return(heightR + 1);  
    return(heightL + 1);  
}
```

Find the node with minimum value in a Binary Search Tree-

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

In binary search tree, the smallest node is in the left side and the largest node is in the right side. To find the smallest node, the process will check the parent node. In case that the parent node is not empty, if it doesn't have a left child node, the smallest node is the parent node; otherwise the smallest node is its left child node.

Find Smallest Node function-(Non-recursive)

```

void btree::smallest(node* root)
{
    node *temp;

    temp=root;

    while(temp->left!=NULL)

        temp=temp->left;

    cout<<"\nMinimum data value="<<temp->data;
}
  
```

Find Largest Node function-(Non-recursive)

```

void btree::largest(node* root)
{
    node *temp;

    temp=root;
  
```

```

while(temp->right!=NULL)

    temp=temp->right;

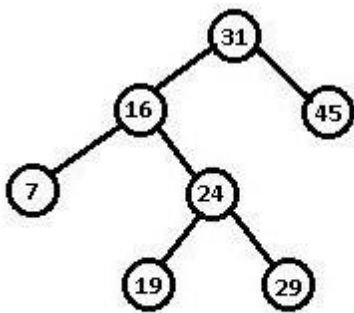
cout<<"\nMaximum data value="<<temp->data;

}

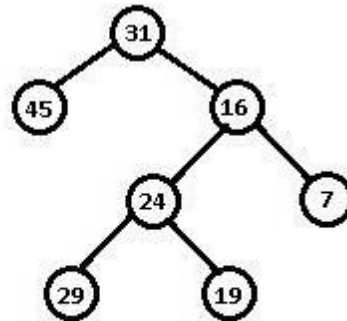
```

Getting Mirror, Replica, or Tree Interchange of Binary Tree

The Mirror() operation finds the mirror of the tree that will interchange all left and right subtrees in a linked binary tree.



Original Tree



Mirror Tree

Recursive Function

```

void BinaryTree :: Mirror(TreeNode *Root)
{
    TreeNode *Tmp;

    if(Root != Null)
    {
        Tmp = Root->Lchild;
        Root->Lchild = Root->Rchild;
        Root->Rchild = Tmp;

        Mirror(Root->Lchild);
        Mirror(Root->Rchild);
    }
}

```

```

    }
}

```

Mirror() Function (Non-Recursive) using Queue-

Initialize a queue of nodes as empty

Initialize a pointer temp = root

Add(temp) to queue

while(queue not empty)

begin

temp = delete

if (temp->left != NULL)

add(temp->left)

if(temp->right != NULL)

add (temp->right)

change = temp->left

temp->Left = temp->right

temp->right = change

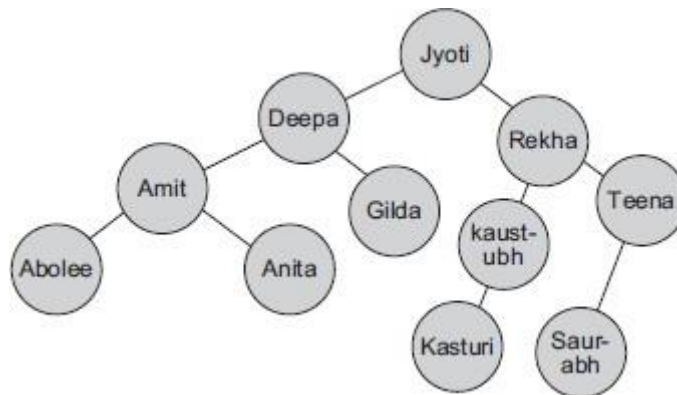
end

end

Searching for a Key-

To search for a target key, we first compare it with the key at the root of the tree. If it is the same, then the algorithm ends. If it is less than the key at the root, search for the target key in

the left subtree, else search in the right subtree. Let us, for example, search for the key „Saurabh“ in following Figure



We first compare „Saurabh“ with the key of the root, „Jyoti“. Since „Saurabh“ comes after „Jyoti“ in alphabetical order, we move to the right side and next compare it with the key „Rekha“. Since „Saurabh“ comes after „Rekha“, we move to the right again and compare with „Teena“. Since „Saurabh“ comes before „Teena“, we move to the left. Now the question is to identify what event will be the terminating condition for the search. The solution is if we find the key, the function finishes successfully. If not, we continue searching until we hit an empty subtree.

Program Code shows the implementation of search () function, both nonrecursive and recursive implementations.

Non-recursive-

```
TreeNode *BSTree :: Search(int Key)
```

```
{
```

```
    TreeNode *Tmp = Root;
```

```
    while(Tmp)
```

```
    {
```

```
        if(Tmp->Data == Key)
```

```
            return Tmp;
```

```

        else if(Tmp->data < Key)
            Tmp = Tmp->Lchild;
        else Tmp = Tmp->Rchild;
    }
    return NULL;
}

```

Recursive-

```

TreeNode *BSTree :: Rec_Search(TreeNode *root, int key)
{
    if(root == Null)
        return(root);
    else
    {
        if(root->Data < Key)
            root = Rec_Search(root->Lchild);
        else if(root->data > Key)
            root = Rec_Search(root->Rchild);
    }
}

```

Software Required: Dev C++ compiler- / 64 bit windows

Input: node names

Output: implement concept of Insert new node, Find number of nodes, Minimum data value found in the tree

Conclusion: Successfully implemented Binary search tree as an ADT using linked list in C++ language.

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features for binary search tree.

GROUP C

Assignment 06

Problem Statement:

Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

Objectives:

1. To understand concept of adjacency matrix/list.
2. To understand the concept of DFS and BFS

Learning Objectives

To understand concept of adjacency matrix/list to perform DFS and using adjacency list to perform BFS

Learning Outcome

Learn object oriented Programming features

Understand & implement concept of adjacency matrix/list to perform DFS and using adjacencylist to perform BFS

Theory

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before backtracking.

Depth-first search in Graph

A Depth-first search (DFS) is a way of traversing graphs closely related to the preorder traversal of a tree. Following is the recursive implementation of preorder traversal:

procedure preorder(treeNode v)

```
{  
    visit(v);  
    for each child u of v  
        preorder(u);  
}
```

To turn this into a graph traversal algorithm, replace “child” with “neighbor”. But to prevent infinite loops, keep track of the vertices that are already discovered and not revisit them.

```

procedure dfs(vertex v)
{
    visit(v);
    for each neighbor u of v
        if u is undiscovered
            call dfs(u);
}

```

Iterative Implementation of DFS

The non-recursive implementation of DFS is similar to the non-recursive implementation of BFS but differs from it in two ways:

- It uses a stack instead of a queue.
- The DFS should mark discovered only after popping the vertex, not before pushing it.
- It uses a reverse iterator instead of an iterator to produce the same results as recursive DFS.

Breadth-first search (BFS)

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a „search key“) and explores the neighbor nodes first before moving to the next-level neighbors.

Breadth-first search (BFS) is a graph traversal algorithm that explores vertices in the order of their distance from the source vertex, where distance is the minimum length of a path from the source vertex to the node as evident from the above example.

Applications of BFS

- Copying garbage collection, Cheney’s algorithm.
- Finding the shortest path between two nodes u and v , with path length measured by the total number of edges (an advantage over depth-first search).
- Testing a graph for bipartiteness.
- Minimum Spanning Tree for an unweighted graph.
- Web crawler.
- Finding nodes in any connected component of a graph.
- Ford–Fulkerson method for computing the maximum flow in a flow network.
- Serialization/Deserialization of a binary tree vs. serialization in sorted order allows the tree to be reconstructed efficiently.

Iterative Implementation of BFS

The non-recursive implementation of BFS is similar to the non-recursive implementation of DFS but differs from it in two ways:

- It uses a queue instead of a stack.
- It checks whether a vertex has been discovered before pushing the vertex rather than delaying this check until the vertex is dequeued.

Software Required: Dev C++ compiler- / 64 bit windows

Input: Graph

Output: implement concept of traveling BFS,DFS

Conclusion: Successfully implemented DFS and BFS using adjacency matrix and list in C++ language.

.

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features for DFS and BFS.

GROUP C

Assignment 07

Problem Statement:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.

Check whether the graph is connected or not. Justify the storage representation used

Objectives:

1. To understand concept of Graph

Learning Objectives

To understand concept of Graph for flight path between cities.

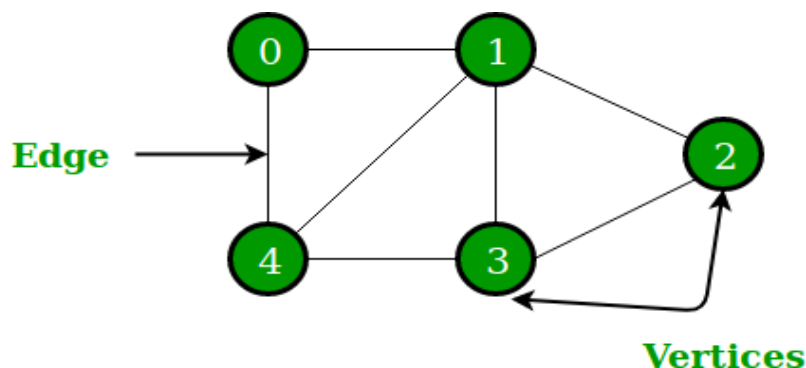
Learning Outcome

Learn object oriented Programming features

Understand & implement concept of Graph.

Theory

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



In the above Graph, the set of vertices $V = \{0,1,2,3,4\}$ and the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also

used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

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

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an $array[]$. An entry $array[i]$ represents the list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

