MIT World Peace University

ADVANCED DATA STRUCTURE

Second Year B.Tech, Semester 2

Implement binary tree using C++ and perform following operations: Creation of binary tree and traversal (recursive and non-recursive)

Assignment 2

Prepared By

P15. Parth Zarekar

Batch A1

February 8, 2023

Contents

1 Problem Statement

Implement binary tree using C++ and perform following operations: Creation of binary tree and traversal (recursive and non-recursive)

2 Objective

1. To study data structure: Tree & Binary Tree
2. To study different traversal in Binary Tree
3. To study recursive and non-recursive approach of programming

3 Theory

A tree data structure is a hierarchical structure that consists of nodes connected by edges. It has a root node and zero or more child nodes. Each child node may have its own child nodes, forming a parent-child relationship between nodes. This relationship creates a tree-like structure, with the root node at the top and leaf nodes at the bottom with no child nodes. Trees are commonly used to represent hierarchical relationship and are used in algorithms for searching and sorting data.

3.1 Binary Tree:

Binary Trees are graphs of tree data structure where each node (shown as circles in the graph to the left) has up to a possible two branches ('Children'). These are called the left branch and right branch, or, sometimes, the left child and right child.
Although they are relatively simple structure, binary trees are extremely useful for modeling data. For example, rational numbers can be represented with a variant of the binary tree. They can also used to represent hierarchies, and reflect structural relationship in data. They are flexible; we can move 'subtree' around easily within a tree as needed. They can also be very efficiently searched and analyzed by computer.

3.1.1 Binary Tree Definitions:

1. **Root:** The topmost node in a binary tree is called the root node.
2. **Leaf node:** A node with children is called a leaf node.
3. **Parent node:** A node with children is called a parent node.
4. **Child node:** A node that is connected to a parent node is called a child node.
5. **Sibling node:** Two or more nodes with the same parent are called sibling nodes.
6. **Ancestor:** A node that is farther from the root node is called an ancestor.
7. **Descendant:** A node that is closer to the root node is called a descendant.
8. **Depth:** The distance from a node to the root node is called the depth of the node.
9. **Height:** The number of edges from a node to the deepest leaf node is called the Height of the node.
10. **Full binary tree:** A binary tree where every node has either 0 or 2 children is called a full binary tree.
11. **Complete binary tree:** A binary tree that is completely filled, except for the rightmost elements of the last level is called a complete binary tree.
12. **Perfect binary tree:** A binary tree that is both full and complete is called a perfect binary tree.

3.2 Different Traversals

In tree data structure, different traversal are used to visit every node of a tree in a specific sequence. The most commonly used tree traversals are:

3.2.1 Inorder Traversal:

Inorder Traversal is a recursive algorithm for traversing or searching tree data structures. It starts at the tree's root node (or another designated node of a sub-tree), and explores the sub-tree's left branch until it reaches the left-most node (i.e., a node without a left child), which it then visits. It then explores the right branch in the same manner, i.e., it recursively visits the left-most node in that sub-tree, and so on, backtracking to the root node once all nodes have been visited.

3.2.2 Preorder Traversal:

Preorder traversal is a recursive algorithm for traversing or searching tree data structures. It starts at the tree's root node (or another designated node of a sub-tree), visits the left subtree, then the right subtree. Printing the value of the root node after visiting the left and right subtrees. Preorder traversal is a depth-first traversal.
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

3.2.3 Postorder Traversal:

Linear data structures such as stack, array, queue, etc., only have one way to traverse the data. But in a hierarchical data structure such as tree, there are multiple ways to traverse the data. So, here we will discuss another way to traverse the tree data structure, i.e., postorder traversal. The postorder traversal is one of the traversing techniques used for visiting the node in the tree. It follows the principle LRN (Left-right-node). Postorder traversal is used to get the postfix expression of a tree.

3.3 Platform

Operating System: Garuda Linux x86-64
IDEs or Text editor Used: Visual Studio Code
Compiler: g++ and gcc on linux for c++

4 Input

1. The Nodes of the Binary Tree

5 Output:

1. The traversal of the binary tree in different ways.

6 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

7 Pseudo Code

7.1 Inorder Traversal - Iterative

```
void inorder_traversal_non_recursive(treenode*temp)
{
  temp =root;
  Stack st;
  while (1)
  {
    while(temp!=NULL)
    {
      st.push(temp);
      temp = temp->left;
    }
    if(st.isempty()==1)
    {
      break;
    }
    temp=st.pop();
    cout<< temp->data<<endl;
    temp = temp->right;
  }
}
```

7.2 Preorder Traversal - Iterative

```
void preorder_traversal_non_recursive(treenode*temp)
{
  temp = root;
  Stack st;
  while(1)
  {
    while(temp!=NULL)
    {
      cout<<temp->data<<endl;
      st.push(temp);
      temp =temp->left;
    }
    if(st.isempty()==1)
    {
      break;
    }
    temp=st.pop();
    temp = temp->right;
  }
}
```

7.3 Postorder Traversal - Iterative

```
void postorder_traversal_non_recursive(treenode*temp)
{
  temp = root;
  Stack st;
  while(1)
  {
    while(temp!=NULL)
    {
      st.push(temp);
      temp = temp->left;
    }
    if (st.data[st.top]->right == NULL)
    {
      temp=st.pop();
      cout<<temp->data<<endl;
    }
    while(st.isempty()==0 && st.data[st.top]->right==temp)
    {
      temp=st.pop();
      cout<<temp->data;
```

```
      }
      if (st.isempty()==1)
      {
        break;
      }
      temp = st.data[st.top]->right;
    }
  }
```

8 Time Complexity

Time Complexity of different Traversal in their Iterative Approach are as follows:

- Inorder Traversal O(n)
- Preorder Traversal O(n)
- Postorder Traversal O(n)

The time complexity is linear.

9 Code

9.1 Program

```
  #include <iostream>
using namespace std;
class treenode {
  public :
  string data;
    treenode *left;
    treenode *right;
    friend class tree;
};

class Stack{
  int top;
  treenode *data [30];
  public:
  Stack()
  {
    top=-1;
  }
  void push (treenode *temp)
  {
    top++;
    data[top] = temp;
  }
  treenode*pop()
  {
    return data[top--];
  }
  int isempty(){
    if (top == -1)
    {
      return 1;
    }
    else
    {
      return 0;
    }
  }
  friend class tree;
};

```

```
class tree {
  public:
  treenode *root;
  tree(){
    root = NULL;
  }
  void create_r()
  {
    root = new treenode;
    cout<<"Enter ROOT Data:"<<endl;
    cin>>root->data;
    root->left = NULL;
    root->right = NULL;
    create_r(root);
  }

  void create_r(treenode * temp){
    char choice1, choice2;
    cout << "Enter whether you want to add the data to the left of \""<< temp->data <<"\" or not (y/n): ";
    cin >> choice1;
    if (choice1 == 'y'){
      treenode *curr1 = new treenode();
      cout << "Enter the data for the left of \""<< temp->data << "\" node: ";
      cin >> curr1 -> data;
      temp -> left = curr1;
      create_r(curr1);
    }
    cout << "Enter whether you want to add the data to the right of \""<<temp->data<<"\" or not (y/n): ";
    cin >> choice2;
    if (choice2 == 'y'){
      treenode *curr2 = new treenode();
      cout << "Enter the data for the Right of \""<< temp->data <<"\" node:";
      cin >> curr2 -> data;
      temp -> right = curr2;
      create_r(curr2);

    }
  }
  //inorder recursive
  void inorder_traversal_r (treenode * node){
    if (node == NULL){
      return;
    }
    inorder_traversal_r(node->left);
    cout << node->data << "\n";
    inorder_traversal_r(node->right);
  }
  //preorder recursive
  void preorder_traversal_r(treenode* temp)
  {
    if (temp!= NULL)
    {
      cout<< temp->data << endl;
      preorder_traversal_r(temp->left);
      preorder_traversal_r(temp->right);
    }
  }
  // postorder recursive
  void postorder_traversal_r(treenode*temp)
  {
    if(temp!=NULL)
    {
      postorder_traversal_r(temp->left);
      postorder_traversal_r(temp->right);
      cout<<temp->data<<endl;
    }
  }
  //Inorder Non recursive
```

```
109 void inorder_traversal_non_recursive(treenode*temp)
110 {
111   temp =root;
112   Stack st;
113   while (1)
114   {
115     while(temp!=NULL)
116     {
117       st.push(temp);
118       temp = temp->left;
119     }
120     if(st.isempty()==1)
121     {
122       break;
123     }
124     temp=st.pop();
125     cout<< temp->data<<endl;
126     temp = temp->right;
127   }
128 }
129
130 //Postorder non recursive
131 void postorder_traversal_non_recursive(treenode*temp)
132 {
133   temp = root;
134   Stack st;
135   while(1)
136   {
137     while(temp!=NULL)
138     {
139       st.push(temp);
140       temp = temp->left;
141     }
142     if (st.data[st.top]->right == NULL)
143     {
144       temp=st.pop();
145       cout<<temp->data<<endl;
146     }
147     while(st.isempty()==0 && st.data[st.top]->right==temp)
148     {
149       temp=st.pop();
150       cout<<temp->data;
151     }
152     if (st.isempty()==1)
153     {
154       break;
155     }
156     temp = st.data[st.top]->right;
157   }
158 }
159
160 //Preorder Non recursive
161 void preorder_traversal_non_recursive(treenode*temp)
162 {
163   temp = root;
164   Stack st;
165   while(1)
166   {
167     while(temp!=NULL)
168     {
169       cout<<temp->data<<endl;
170       st.push(temp);
171       temp =temp->left;
172     }
173     if(st.isempty()==1)
174     {
175       break;
176     }
177     temp=st.pop();
178     temp = temp->right;
```

```
    }
  }
};

int main() {
  tree bt;
  int ch;
  char c;
  do {

  cout<<"\nWhat you want to do:\n1.Enter Tree\n2.Inorder Recursive\n3.Preorder Recursive\n4.
    Postorder Recursive\n5.Inorder Non Recursive\n6.Preorder Non Recursive\n7.Postorder Non
    Recursive\nchoose:"<<endl;
  cin>>ch;
  switch (ch)
  {
    case 1:
        bt.create_r();
        break;
    case 2:
        cout << "The inorder display of the tree is: " << endl;
        bt.inorder_traversal_r(bt.root);
        break;
    case 3:
        cout<<"The preorder display of the tree is:" << endl;
        bt.preorder_traversal_r(bt.root);
        break;
    case 4:
        cout<<"The postorder display of the tree is:"<<endl;
        bt.postorder_traversal_r(bt.root);
        break;
    case 5:
        cout<<"The Inorder Non Recursive Display of the tree is:"<<endl;
        bt.inorder_traversal_non_recursive(bt.root);
        break;
    case 6:
        cout<<"The Preorder Non Recursive Display of the tree is:"<<endl;
        bt.preorder_traversal_non_recursive(bt.root);
        break;
    case 7:
        cout<<"The Postorder Non Recursive Display of the tree is:"<<endl;
        bt.postorder_traversal_non_recursive(bt.root);
        break;
    default:
        cout<<"Invalid choice";
      break;
    }

    cout << "Do you want to continue(y/n):";
    cin>> c;
  }while (c == 'y');


  return 0;
}
```

9.2 Input and Output

```
  What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
1
```

```
Enter ROOT Data:
1
Enter whether you want to add the data to the left of "1" or not (y/n): y
Enter the data for the left of "1" node: 2
Enter whether you want to add the data to the left of "2" or not (y/n): y
Enter the data for the left of "2" node: 3
Enter whether you want to add the data to the left of "3" or not (y/n): y
Enter the data for the left of "3" node: 4
Enter whether you want to add the data to the left of "4" or not (y/n): y
Enter the data for the left of "4" node: 5
Enter whether you want to add the data to the left of "5" or not (y/n): n
Enter whether you want to add the data to the right of "5" or not (y/n): n
Enter whether you want to add the data to the right of "4" or not (y/n): n
Enter whether you want to add the data to the right of "3" or not (y/n): n
Enter whether you want to add the data to the right of "2" or not (y/n): n
Enter whether you want to add the data to the right of "1" or not (y/n): n

Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
2
The inorder display of the tree is:
5 4 3 2 1
Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
3
The preorder display of the tree is:
1 2 3 4 5
Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
4
The postorder display of the tree is:
5 4 3 2 1
Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
```

```
5
The Inorder Non Recursive Display of the tree is:
5 4 3 2 1
Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
6
The Preorder Non Recursive Display of the tree is:
1 2 3 4 5
Do you want to continue(y/n):y

What you want to do:
1.Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
7
The Postorder Non Recursive Display of the tree is:
5 4 3 2 1
Do you want to continue(y/n):n
```

10 Conclusion

Thus, learnt about the different kinds of traversals in binary tree and also learnt about the recursive and non-recursive approach of programming.

11 FAQs

11.1 Explain any one application of binary tree with suitable example.

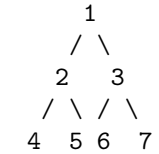
Routing Tables: A routing table is used to link routers in a network. It is usually implemented with a trie data structure, which is a variation of a binary tree. The tree data structure will store the location of routers based on their IP addresses. Routers with similar addresses are grouped under a single subtree.

To find a router to which a packet must be forwarded, we need to traverse the tree using the prefix of the network address to which a packet must be sent. Afterward, the packet is forwarded to the router with the longest matching prefix of the destination address.

11.2 Explain sequential representation of binary tree with example

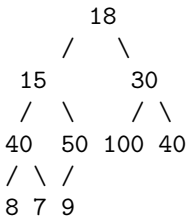
Sequential representation of binary trees refers to the way in which a binary tree is stored in an array. In this representation, the nodes of the binary tree are stored in a breadth-first order, i.e., the nodes are stored level-by-level from left to right. If a node is missing, it is represented by a special value, such as null or a sentinel value.

For example, consider the following binary tree:



Its sequential representation would be an array as follows: [1, 2, 3, 4, 5, 6, 7]. In this representation, the root of the tree is stored at the first index (1), the left child of the root at the second index (2), the right child of the root at the third index (3), and so on.

11.3 Write inorder, preorder and postorder for following tree



Answer:

1. Inorder: 8, 40, 7, 15, 9, 50, 18, 100, 30, 100, 40
2. Preorder: 15, 40, 8, 7, 50, 9, 18, 30, 100, 40, 100
3. Postorder: 8, 7, 40, 9, 50, 15, 100, 40, 100, 30, 18