PROJECT 4
Copying of Tree

Under the guidance of
Dr. P. Thiyagarajan
Associate Professor
Head Dept. of Computer Science
(Cyber Security)

DEPARTMENT OF COMPUTER SCIENCE
RAJIV GANDHI NATIONAL INSTITUTE OF YOUTH DEVELOPMENT,
SRIPERUMBUDUR – 602105

Made by

Name	Topic	Page No.
1. ATHAR OBAID KHAN	INSERTION IN TREE/ORGINAL TREE FORMATION	10-14
2. ANUPRIYA KUMARI	COPYING OF TREE	15-19
3. MUHAMMED ANSHAD M K	DELETION OF NODE	20-25
4. RUSTHAM SHAHAN. V	TRAVERSAL	26-28
5. VARADHA S AJITH	FINDING CHILD NODE	29-31
6. KARNATI VENKATA PRAVEEN KUMAR REDDY	FINDING PARENT NODE	32-34
7. ISHIKA MANDAL	FINDING SIBLING OF NODE	35-37
8. VENKATESWARLU NAGAM	ADDRESS OF NODE	38-41

Department of Computer Science
Rajiv Gandhi National Institute of Youth Development
Sriperumbudur, Tamil Nadu
India – 602105

CERTIFICATE

This is to certify that we have examined the project entitled "**TREE COPYING"**, submitted by **Anupriya Kumari**, **Varadha s Ajith**, **Ishika Mandal**, **Rustham Shahan. V**, **Athar Obaid Khan**, **Muhammed Anshad M k**, **Nagam Venkateswarlu**, **K V Praveen Kumar Reddy** (Roll number: *MSAI22R004, MSCS22R016, MSCS22R006, MSDS22R011, MSAI22R006, MSAI22R012, MSDS22R015, MSCS22R007)*, the post graduate students of **Department of Computer Science** in partial fulfillment of for the award of degree of **Master of Computer Science**. We hereby accord our approval of it as a study carried out and presented in a manner required for its acceptance in fulfillment for **MSCS101-Data Structures And Algorithms** course for which it has been submitted. The project has fulfilled all the requirements as per the regulations of the institute as well as course instructor and has reached the standard needed for submission.

Supervisor
Dr.P.Thiyagarajan
Department of Computer Science
(Cyber Security)
RGNIYD, Sriperumbudur

Place: Sriperumbudur

Date: 10/02/2023

ACKNOWLEDGEMENT

We would like to express our sincere and deep gratitude to our supervisor and guide **Dr.P.Thiyagarajan**, Associate Professor, HOD of computer science(Cyber Security), his kind and constant support during our post-graduation study. It has been an absolute privilege to work with Dr.P.Thiyagarajan for our project. His valuable advice, critical criticism and active supervision encouraged us to sharpen our research methodology and was instrumental in shaping our professional outlook.

CONTENT

INTRODUCTION

A "copying of a tree" refers to creating a duplicate of an existing tree data structure. This can be done in several ways, including a shallow copy, where only the references to the nodes are copied, or a deep copy, where new nodes are created with the same values as the original nodes.

When creating a deep copy, it is important to ensure that all nodes and any connected sub-trees are also copied. This can be done recursively, starting from the root node, and copying each connected sub-tree until all nodes have been copied.

The time and space complexity of the copying operation will depend on the specific implementation and data structure used for the tree. In general, the time complexity of a deep copy is O(n), where n is the number of nodes in the tree, since each node must be visited and copied. The space complexity is also O(n), since a new node must be created for each node in the original tree.

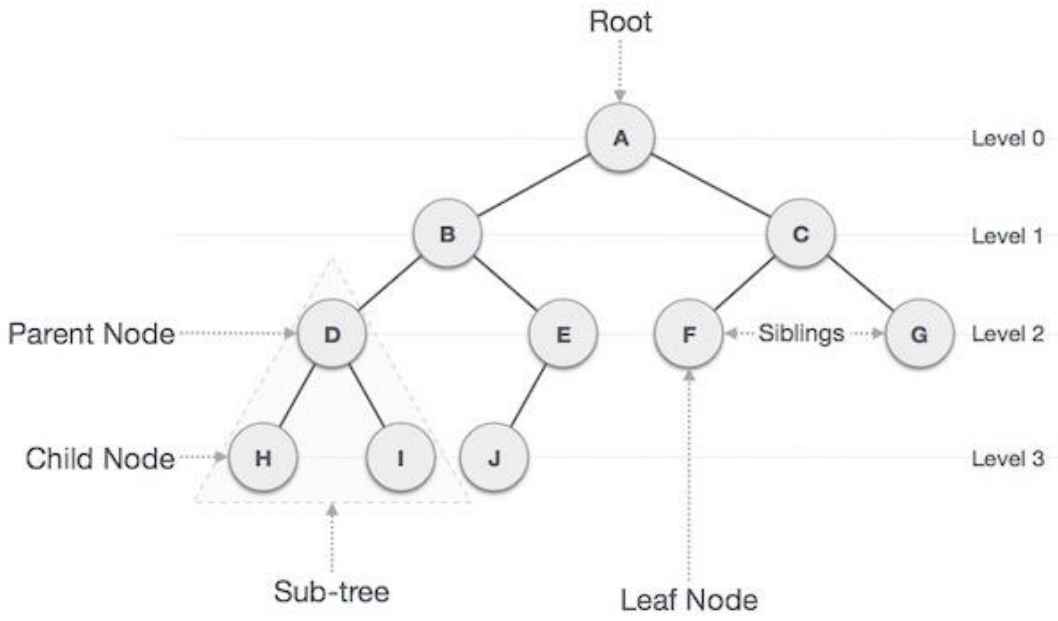
Overall, the process of copying a tree can be an important operation in tree algorithms and data structures, and careful consideration of the specific requirements and implementation should be taken to ensure correct and efficient results.

Here, in our project we design our code on basis of Binary Tree. First, we should create Binary tree with insertion of several node.

Now, first of all, we should know about Binary Tree, here there is short description about Binary Tree.

Binary Tree

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

Following are the important terms with respect to tree.

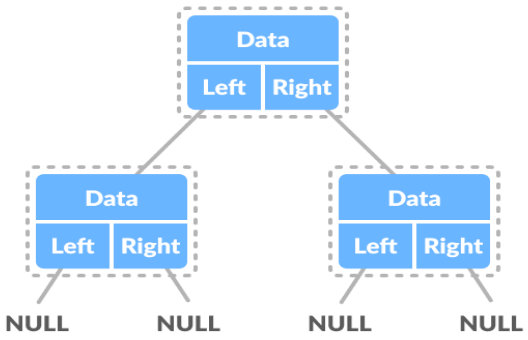
- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.

- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

CREATING THE BINARY TREE

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

Representation:



A binary tree is a data structure that consists of a root node and two subtrees, called the left subtree and the right subtree. Each node in a binary tree can have at most two child nodes. Here is a step-by-step explanation on how to create a binary tree:

1. Define a structure for a binary tree node: A binary tree node has three elements: data, pointer to the left child, and pointer to the

right child. Here's an example in C programming language:

```c
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

2. Create a function to create a new node: This function will allocate memory dynamically for a new node and assign the value of data to the node.

```c
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

3. **Create a function to insert data into the tree:** This function will insert a new node into the binary tree. The insertion logic is such that if the data is less than or equal to the current node, it will be inserted in the left subtree and if the data is greater than the current node, it will be inserted in the right subtree.

```c
// INSERTING A NODE IN TREE

struct Node* insertNode(struct Node* node, int data)
{
  if (node == NULL)
  {
    printf("Node is created\n");
    return createNode(data);
  }
  if (data < node->data)
  {  printf("it added to left of %d.  \n", node->data);
    node->left = insertNode(node->left, data);
  }
  else if (data > node->data)
  {  printf("it added to right of %d. \n", node->data);
    node->right = insertNode(node->right, data);
  }
  return node;
}
```

Create the root node: This is the starting point of the binary tree. You can create the root node using the createNode function.

This is a function to insert a node in a binary tree.

1)First, it checks if the given node is NULL, i.e., the tree is empty. If it is NULL, it calls the function **createNode** to create a new node with the given data, and returns it.

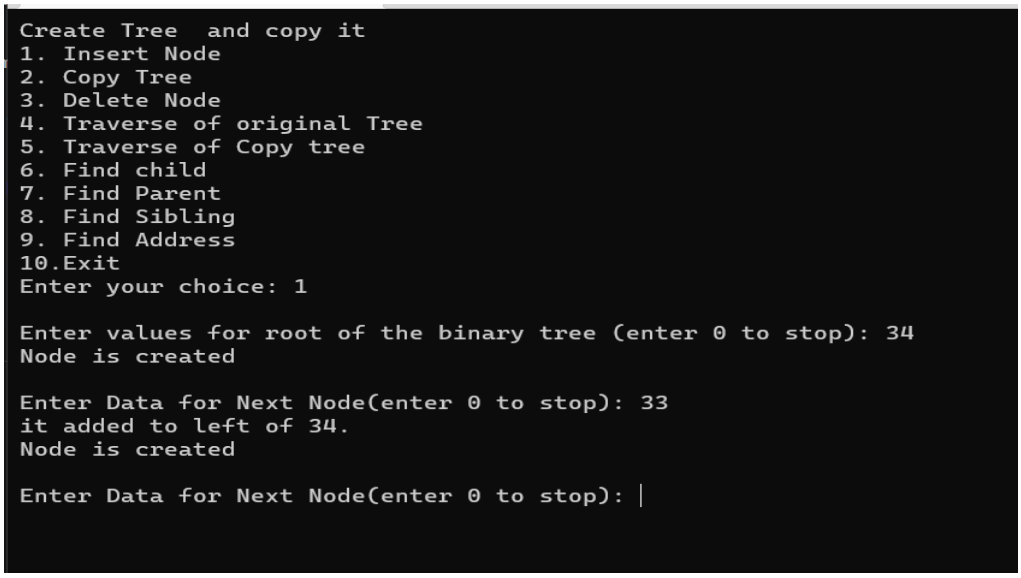
2) If the given node is not NULL, it checks whether the data to be inserted is less than the data of the node. If it is, it sets the left child of the node as the result of a recursive call to the **insertNode** function, passing the left child of the node and the data to be inserted.
3) If the data to be inserted is greater than the data of the node, it sets the right child of the node as the result of a recursive call to the **insertNode** function, passing the right child of the node and the data to be inserted.

4) Finally, it returns the node.
    a. Note: The **printf** statements are there to print some output to the user to show how the insertion is taking place.

Insert data into the tree: You can insert data into the tree using the insert function and passing the root node and the data you want to insert as parameters.

In main function there are 10 choices, that we could select. So for insertion, we choose choice 1, then it ask to enter Root Node, if we enter the root node, then it will ask next node and according to features of Binary Tree it will add.

Demo Output:



COPYING OF TREE

```
// COPYING OF TREE

struct Node* copyTree(struct Node* node)
{
    struct Node* newNode;
  if (node == NULL) return NULL;
  newNode = createNode(node->data);
  newNode->left = copyTree(node->left);
  newNode->right = copyTree(node->right);
  return newNode;
}
```

This above function creates copy of the tree.

Working:

- This function is a recursive implementation to create a copy of a binary tree. The function takes as input a node of the binary tree, and returns a new node that is a duplicate of the input node.
- The function first checks if the input node is NULL. If it is, the function returns NULL, indicating that the copy of the tree has reached its end.

- Otherwise, the function creates a new node using the "createNode" function, passing the data stored in the input node as an argument. The data stored in the new node is identical to that of the input node.
- Next, the function recursively calls itself, passing the left and right children of the input node as arguments, to create a copy of the left and right subtrees of the input node. The left and right children of the new node are assigned the results of these recursive calls.
- Finally, the function returns the newly created node, which is a complete copy of the input node and its subtrees.

```
copy = copyTree(root);
if(copy==NULL){ printf("\nNo any Tree, 1st Create Tree\n");}
else{ printf("\nTree Copied\n");}
break;
```

The above code is the calling statement (in main function) to call copyTree function to copy the original tree.

The code creates a new variable **copy** and assigns to it the result of calling **copyTree** on the root of the original binary tree. This creates a copy of the original binary tree.

```
// Function for traversing the copyied tree
printf("\nInorder traversal of the copy tree: ");
inorder(copy);
printf("\nPreorder traversal of the copy tree: ");
preorder(copy);
printf("\nPostorder traversal of the copy tree: ");
postorder(copy);
break;
```

Calling traversal functions (in main) to traverse the copied tree.

the code uses three functions **inorder**, **preorder**, and **postorder** to traverse the copied tree and print its values in different orders.

- In-order traversal: In this traversal, the left subtree is visited first, then the root node, and finally the right subtree.

- Pre-order traversal: In this traversal, the root node is visited first, followed by the left subtree, and finally the right subtree.
- Post-order traversal: In this traversal, the left subtree is visited first, followed by the right subtree, and finally the root node.

By performing these three traversals, you can see the values of the nodes in the copied tree in different orders and verify that the copy is correct.

Output:

This is the output of the original tree:

```
Inorder traversal of the original tree: 10 20 30 35 40 50 60 70 80
Preorder traversal of the original tree: 50 30 20 10 40 35 70 60 80
Postorder traversal of the original tree: 10 20 35 40 30 60 80 70 50
```

```
1. Insert Node
2. Copy Tree
3. Delete Node
4. Traverse of original Tree
5. Traverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice: 2
Tree Copied
```

This is the output of the copied tree:

```
Inorder traversal of the copy tree: 10 20 30 35 40 50 60 70 80
Preorder traversal of the copy tree: 50 30 20 10 40 35 70 60 80
Postorder traversal of the copy tree: 10 20 35 40 30 60 80 70 50
```

Here we can see the output of the copied tree and the original tree is same.

Which shows copying of tree is successfully done.

Note: once the copying of tree is done, it will have no effect of modification (I.e., insertion or deletion) made in the original tree.

DELETION OF TREE

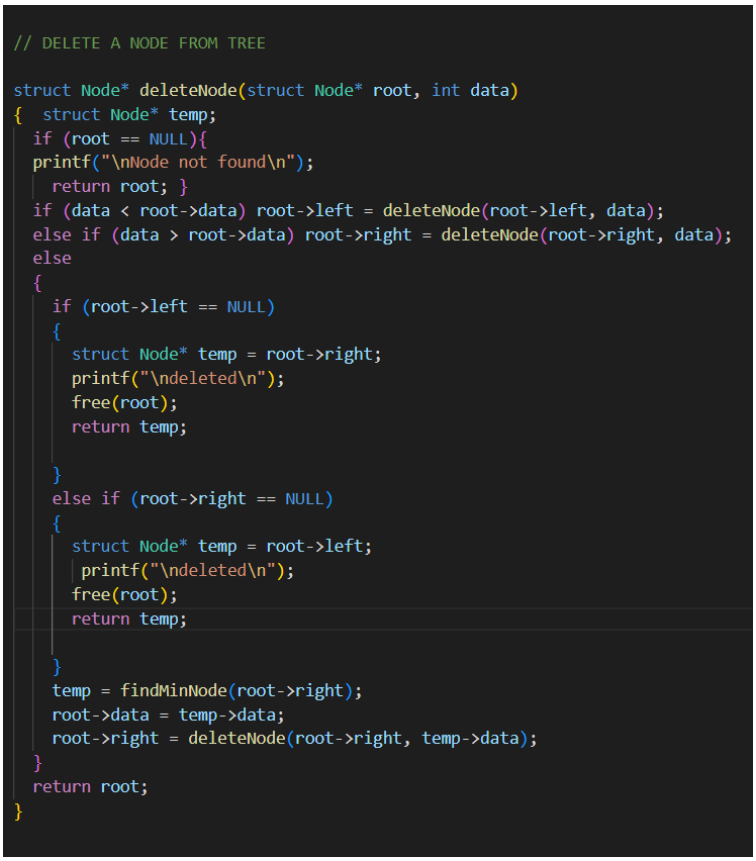
Delete function is used to delete the specified node from a binary tree. However, we must delete a node from a binary search tree in such a way, that the property of binary tree doesn't violate. There are three situations of deleting a node from binary tree.

Deleting a node in a binary tree can be done in the following steps:

1. Find the node to be deleted: You first need to locate the node that you want to delete from the binary tree.
2. Determine if the node is a leaf node: If the node is a leaf node (i.e., it has no children), then you can simply remove it from the tree.
3. Handle deletion of a node with one child: If the node has only one child, you can bypass the node and replace it with its child.
4. Handle deletion of a node with two children: If the node has two children, you need to find the in-order successor of the node (i.e., the next smallest value in the tree). Replace the node with the in-order successor and then delete the in-order successor.

5. Rebalance the tree: After deleting a node, you may need to rebalance the tree to maintain its binary tree properties.

Here the brief explanation of my code:



```c
// DELETE A NODE FROM TREE

struct Node* deleteNode(struct Node* root, int data)
{  struct Node* temp;
  if (root == NULL){
  printf("\nNode not found\n");
    return root; }
  if (data < root->data) root->left = deleteNode(root->left, data);
  else if (data > root->data) root->right = deleteNode(root->right, data);
  else
  {
    if (root->left == NULL)
    {
      struct Node* temp = root->right;
      printf("\ndeleted\n");
      free(root);
      return temp;

    }
    else if (root->right == NULL)
    {
      struct Node* temp = root->left;
       printf("\ndeleted\n");
      free(root);
      return temp;

    }
    temp = findMinNode(root->right);
    root->data = temp->data;
    root->right = deleteNode(root->right, temp->data);
  }
  return root;
}
```

This code is an implementation of a function to delete a node from a binary search tree. Here is an explanation of the code line by line:

1. **struct Node* deleteNode(struct Node* root, int data)**: The function takes two arguments, a pointer to the root node of the binary search tree and an integer **data** which is the value of the node to be deleted. The function returns a pointer to the root node of the binary search tree.
2. **if (root == NULL)**: If the root node is **NULL**, this means the tree is empty, so the function returns **root** (which is **NULL**) and prints a message indicating that the node was not found.
3. **if (data < root->data) root->left = deleteNode(root->left, data);** If the data to be deleted is less than the value of the root node, the function calls itself recursively with the left child of the root node. The result of this recursive call is then assigned to the **left** pointer of the root node. This continues until the node to be deleted is found.
4. **else if (data > root->data) root->right = deleteNode(root->right, data);** If the data to be deleted is greater than the value of the root node, the function calls itself recursively with the right child of the root node. The result of this recursive call is then assigned to the **right** pointer of the root

node. This continues until the node to be deleted is found.

5. **else**: If the data to be deleted is equal to the value of the root node, the node to be deleted has been found.
6. **if (root->left == NULL)**: If the left child of the node to be deleted is **NULL**, the node to be deleted has no left child. In this case, the right child of the node to be deleted (if it exists) becomes the root of the subtree. The node to be deleted is freed from memory using the **free ()** function.
7. **else if (root->right == NULL)**: If the right child of the node to be deleted is **NULL**, the node to be deleted has no right child. In this case, the left child of the node to be deleted (if it exists) becomes the root of the subtree. The node to be deleted is freed from memory using the **free()** function.
8. **temp = findMinNode(root->right);**: If the node to be deleted has both a left and right child, the minimum value node in the right subtree is found using the **findMinNode()** function. This node is stored in a temporary variable **temp**.

9. **root->data = temp->data;** The value of the node to be deleted is replaced with the value of the minimum node found in the previous step.
10. **root->right = deleteNode(root->right, temp->data);**: The minimum node found in step 8 is then deleted from the right subtree by calling the **deleteNode()** function recursively. The result of this call is then assigned to the **right** pointer of the root node.
11. **return root;**: Finally, the function returns a pointer to the root node of the binary search tree.

Demo Output:

Traverse before deletion:

```
1. Insert Node
2. Copy Tree
3. Delete Node
4. Traverse of original Tree
5. Traverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice: 4
Inorder traversal of the original tree: 11 12 13 14
Preorder traversal of the original tree: 12 11 13 14
Postorder traversal of the original tree: 11 14 13 12
```

Deletion:

Traverse after deletion:

```
Tree Copied

1. Insert Node
2. Copy Tree
3. Delete Node
4. Traverse of original Tree
5. Traverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice: 3

 Enter Data to Delete :12

deleted
```

TRAVERSAL

Source Code:

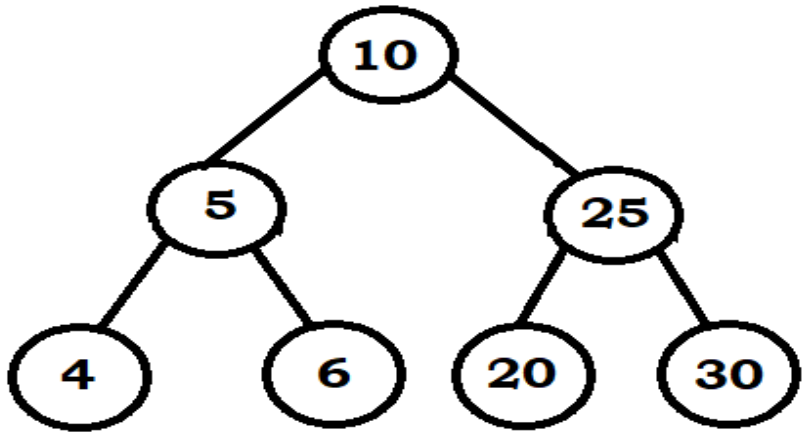
```c
// Function to perform in-order traversal of the tree
void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Function to perform pre-order traversal of the tree
void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}


// Function to perform post-order traversal of the tree
void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

Output:

```
Enter Data for Next Node: 0
1. Insert Node
2. Delete Node
3. Tranverse
4. Copy of tree
5. Find child
6. Find Parent
7. Find Sibling
8. Find Address
9.Exit
Enter your choice: 3
Inorder traversal of the original tree: 4 5 6 10 20 25 30
Preorder traversal of the original tree: 10 5 4 6 25 20 30
Postorder traversal of the original tree: 4 6 5 20 30 25 10
```

Explanation:

The above shown output gives the Inorder, Preorder, and Postorder Traversal of a tree that I created. The structure of the tree is shown below.



For performing various traversal methods like Inorder traversal, preorder traversal and postorder traversal, I created 3 separate functions for each of them.

The function ‘inorder’ takes a pointer to the root node of the tree (struct Node*) as an argument and prints the data of each node in the tree as it visits it in the Inorder Traversal method. In an Inorder traversal, the left subtree of a node is visited first, then the node itself is processed, and finally the right subtree is visited in the format left, root, right. This function implements this by calling itself recursively on the left child of the node (node->left) if it exists, printing the

data of the node (printf (“%d”, node->data)), and then calling it recursively on the right child of the node (node->right) if it exists. If the node is NULL, the function returns immediately, as this means it has reached a leaf node or an empty subtree.

In preorder traversal the root node is processed first, followed by the left subtree and then right subtree. The function ‘preorder’ takes a pointer to the root node of tree as an argument and check if the node is NULL . If it is NULL it returns. If it is not, it prints the value of the node and then recursively calls the ‘preorder’ function with the ‘left’(node->left) and ‘right’(node->right) child nodes.

In Postorder traversal, the left and right subtrees are processed first, and then the root node. The ‘postorder’ function first checks if the node passed as an argument is NULL or not, if it is NULL, the function is returned. If not, the function recursively calls itself on the left (node->left) and right(node->right) child nodes respectively. Finally, the data of the current node is printed. This process will continue until all the nodes in the tree are traversed and printed.

FINDING CHILD NODE

SOURCE CODE:

```
int findChildren(struct Node* node, int data)
{
    if (node == NULL)
    {
        printf("Node not found\n");
        return 0;
    }
    if (node->data == data)
     {
            if (node->left != NULL)
        {
                printf("Left child: %d\n", node->left->data);
            }
            if (node->right != NULL)
            {
                printf("Right child: %d\n", node->right->data);
            }
            return 0;
        }
    findChildren(node->left, data);
    findChildren(node->right, data);
}
```

OUTPUT:

```
Inorder traversal of the original tree: 1 2 3 4 5 7 8 9 11
Preorder traversal of the original tree: 5 3 2 1 4 8 7 9 11
Postorder traversal of the original tree: 1 2 4 3 7 11 9 8 5
1. Insert Node
2. Delete Node
3. Tranverse
4. Copy of tree
5. Find child
6. Find Parent
7. Find Sibling
8. Find Address
9.Exit
Enter your choice: 5
Enter Data of which you want to know child :5
Left child: 3
Right child: 8
```

EXPLANATION:

Here we are doing the searching for a node in a binary tree that has a value equal to the input data and then printing the values of its left and right children, if they exist.

The function takes a pointer to a node of the tree (struct Node*node) and the data that the function is searching for(int data) and here it checks if the node is NULL and if it is so it prints “node not found” and return 0. If node is not NULL and the value is equal to input data then it checks if the left and right children of the node exist,if so prints their values. If the node’s value is not equal to the data, then the function recursively searches for the node by calling itself with the node’s left and right children as inputs.

FINDING PARENT NODE

SOURCE CODE:

```
// FINDING PARENT OF THE GIVEN NODE

struct Node* findParent(struct Node* node, int data)
{
  struct Node* left;
  if (node == NULL) return NULL;
  if (node->left && node->left->data == data) return node;
  if (node->right && node->right->data == data) return node;
  left = findParent(node->left, data);
  if (left) return left;
  return findParent(node->right, data);
}


void findParentNode(struct Node* node, int data) {
  struct Node* parent = findParent(node, data);
  if (parent == NULL) {
    printf("Node not found or it is the root node\n");
    return;
  }
  printf("Parent of node with data %d: %d\n", data, parent->data);
}
```

Initialize a pointer "current" to the root node of the binary tree.

This step sets the starting point for our search to the root node of the tree.

Compare the value of the current node with the target node.

In this step, we compare the value of the current node to the target node, which is the node we want to find the parent of.

If the target node value is less than the current node value, move the current pointer to the left child.

If the target node is less than the current node, it means that the target node is located in the left subtree of the current node. So, we move the current pointer to the left child.

If the target node value is greater than the current node value, move the current pointer to the right child.

If the target node is greater than the current node, it means that the target node is located in the right subtree of the current node. So, we move the current pointer to the right child.

Repeat steps 2 to 4 until the current node is either the target node or is null.

Repeat these steps until the current node is either the target node or until it becomes null.

If the current node is null, return null as the parent node could not be found.

If the current node is null, it means that the target node is not present in the binary tree, so we return null.

If the current node is the target node, return the parent node.

If the current node is the target node, it means that we have found the target node, and the parent node is stored in the "parent" variable. So, we return the parent node

OUTPUT:

```
it added to left of 99.
Node is created

Enter Data for Next Node(enter 0 to stop): 0

1. Insert Node
2. Copy Tree
3. Delete Node
4. Traverse of original Tree
5. Traverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice: 7
Enter Data of which you want to know Parent :44
Parent of node with data 44: 99

1. Insert Node
2. Copy Tree
3. Delete Node
4. Traverse of original Tree
5. Traverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice:
```

FINDING SIBLING NODE

SOURCE CODE:

```
// FINDING SIBLING OF THE GIVEN NODE


void findSibling(struct Node* node, int data)
{
  struct Node* parent = findParent(node, data);
  if (parent == NULL) {
    printf("Node not found\n");
    return ;
  }
  printf("Sibling of %d: ", data);
  if (parent->left && parent->left->data == data) {
    if (parent->right) printf("%d\n", parent->right->data);
    else printf("NULL\n");
  } else {
    if (parent->left) printf("%d\n", parent->left->data);
    else printf("NULL\n");
  }
}
```

Figure 1

This function is finding sibling of the given node..

EXPLANATION:

The function works as follows:

- The **findSibling** function takes as input a pointer to the root node of a binary tree, **node**, and an integer **data** which represents the value of a node in the binary tree.

- The function starts by calling another function **findParent**, which takes the root node and the value of a node as input, and returns a pointer to the parent node of the node with the given value. If the **findParent** function returns **Null**, it means that the node with the given value does not exist in the binary tree, and the function prints "Node not found" and returns.

- If the **findParent** function returns a valid pointer, the function continues by printing "Sibling of [data]:" where [data] is replaced with the input value. Then, it checks if the node with the given value is the left child of its parent. If it is, the function checks if the parent has a right child. If it does, the function prints the value of the right child. If the parent does not have a right child, the function prints "NULL".

- If the node with the given value is not the left child of its parent, it must be the right child. The function then checks if the parent has a left child. If it does, the function prints the value of the left child. If the parent does not have a left child, the function prints "NULL".

- In essence, this function finds the sibling of a node in a binary tree by finding its parent and checking if the parent has a left or right child that is not equal to the node with the given value.

OUTPUT:

If exist it will print the sibling of the given node otherwise it prints **NULL.**

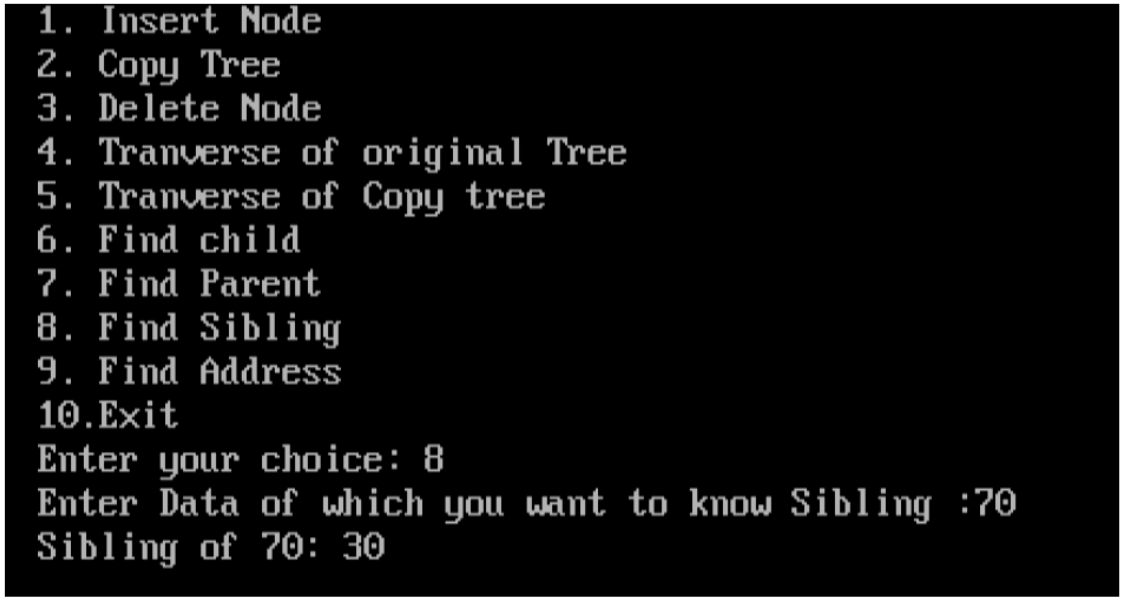


Figure 2

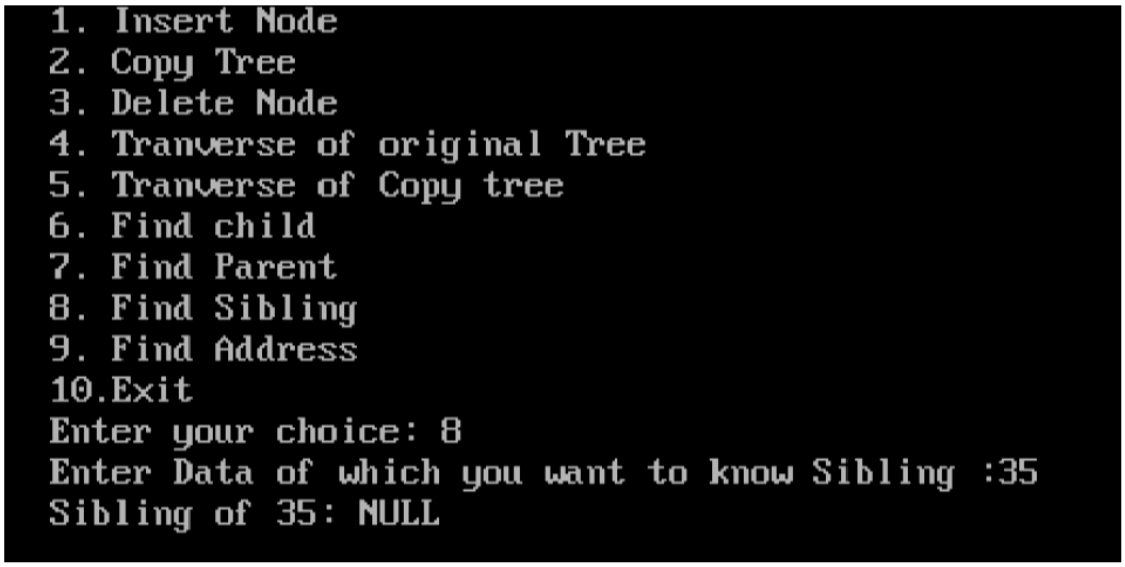

1. Insert Node
2. Copy Tree
3. Delete Node
4. Tranverse of original Tree
5. Tranverse of Copy tree
6. Find child
7. Find Parent
8. Find Sibling
9. Find Address
10.Exit
Enter your choice: 8
Enter Data of which you want to know Sibling :35
Sibling of 35: NULL

Figure 3

FINDING ADDRESS OF GIVEN NODE

SOURCE CODE:

```
// FINDING ADDRESS OF GIVEN NODE


struct Node* findNode(struct Node* node, int data)
{
   struct Node* left;
  if (node == NULL)  return NULL;
  if (node->data == data) return node;
  left = findNode(node->left, data);
  if (left) return left;
  return findNode(node->right, data);
}

void findAddress(struct Node* node, int data)
{
  struct Node* target = findNode(node, data);
  if (target == NULL) {
    printf("Node not found\n");
    return ;
  }
  printf("Address of node with data %d: %p\n", data, (void*)target);
}
```

This function is finding address of the given node.

This code contains two functions, **`findNode`** and **`findAddress`**.

The function works as follows:

★ FindNode function

The **findNode** function is a recursive function that takes a binary tree represented by **node** and an integer **data** as input. The function searches for a node with data **data** in the binary tree starting from the node **node**.

1. If the current node is **NULL**, it returns **NULL**.
2. If the data of the current node is equal to **data**, it returns the current node.
3. If the data of the current node is not equal to **data**, the function searches for the node with **data** in the left subtree by calling **findNode** on **node->left** and **data**.
4. If the function finds the node in the left subtree, it returns the node.
5. If the node is not found in the left subtree, the function searches for the node in the right subtree by calling **findNode** on **node->right** and **data**.

★ FindAddress function:

- The **findAddress** function takes a binary tree represented by **node** and an integer **data** as input.

- It calls the **`findNode`** function to search for the node with **`data`** in the binary tree.
- If the node is found, the function prints the address of the node with data **`data`** in the format **`"Address of node with data %d: %p"`**.
- If the node is not found, the function prints **`"Node not found"`**.

Note: The **`%p`** format specifier is used to print a pointer as a hexadecimal value.

```
printf("Enter Data of which you want to know address :");
scanf("%d", &data);
    printf("\nIn Original Tree\n");
findAddress(root, data);
printf("\nIn Copied Tree\n");
findAddress(copy, data);
break;
```

The above figure is the calling statement (in main function) to find address of the given node:

It takes input **data** from the user and calls the **findAddress** function twice with two different binary trees, original and copied.

- The first call to **findAddress** is performed on the **original** binary tree to find address of given node in original tree
- the second call is performed on the **copied** binary tree to find address of given node in the copied tree.

The **findAddress** function will search for a node with data equal to **data** in the binary tree and print its address if it is found. If the node is not found, it will print **"Node not found"**.

OUTPUT:

```
Enter Data of which you want to know address :30
In Original Tree
Address of node with data 30: 0x562cd45b5ae0

In Copied Tree
Address of node with data 30: 0x562cd45b5c00
```

Here as expected it gives address of "30" in original tree as well as copied tree.

APPLICATIONS

1. Data Backup & Recovery: Creating a copy of a tree data structure can be useful for backup & recovery purposes, allowing you to restore the original tree in case of data loss.

2. Graph algorithms: In graph algorithms, it is often necessary to make a copy of the graph to avoid modifying the original graph while exploring different solutions.

3. Tree Traversals: Tree traversals, such as depth-first search and breadth-first search, can make use of tree copying to traverse a tree without modifying to the original structure.

4. Search Algorithms: Tree copying can be used in search algorithms to store multiple states of the tree while exploring different solutions, allowing you to keep track of the best solution found so far.

5. Artificial Intelligence & Machine Learning: In Artificial Intelligence & Machine Learning, Tree Copying is used to explore different solutions for problems, such as decision trees, search trees & Monte Carlo trees.

CHALLENGES

This project was very fun to make but not so easy. From every function execution to documentation, it was very interesting as well as challenging too, that is why we would like to call it as STEP OF IMPROVEMENT rather than CHALLENGES, as every time we were not only rectifying the errors or simply executing the function but tried to make it more user friendly more intuitive and more presentable.

1. As everyone are using different compilers so combing and make it executable with every possible compiler (like turbo C++, visual Studio, online compiler) was the most difficult task as every compiler have some changes while execution.
2. Insertion: We will not list this as challenge but as improvement. After designing the tree, making it user friendly by explaining them where the node is going to be inserted, so that they can keep tracking the tree during insertion also (before traversing it).
3. Copying: after copying the tree, initially coping and its traversal are merged together. Therefore, program used to copy and traverse it as well .so

it become difficult to Analyse if modification in original tree is affecting the copied tree or not and every time traversal make the output more tedious. later we modified it and give user the freedom of choice if they only want to copy it or also want to traverse it after modification to check the changes.

4. Unsolved challenge:

- we wanted to give user the choice of syncing the tree or make it independent but we could not achieve it but we will keep on trying and under the guidance of our professor. hopefully we will come up with its solution also.
- we wanted to give user the option if they want to completely copy the tree or just some portions of it but due to time constrain, we could not try it.

5. Deletion: in this earlier when we call the delete function it used to delete the whole tree. But as per our principle of this project we made deletion as per user input of node.
6. Finding parent of node: it was little bit challenging as we must keep tracking the parent of every node in order to find it.
7. Finding address of given node:

- it was confusing as it gave different output in different compilers, so to verify if our output is correct or not was confusing.
- We tried to make this more interactive. Now, it is not only finding the given node address in original tree but also in copied tree.

CONCLUSION

Here we are making binary tree and trying to copy the tree. Here we mainly use four function find child of the node, find address of the given node, find parent of the given node, find sibling of the given node. Here in the child function, it actually takes the input from the user to which data that they have to find the child and if there exist any child of the node it will print the left and right child of the node. In the address it takes input from the user for which node they must find the address and the function find address will return the address of the given node in original tree as well as in copied tree. In sibling function, it takes user input to which they want to find the sibling and if there exists any sibling it will print the sibling of the tree. The main thing is if we are doing any modifications or changes like insertion or deletion in original tree doesn't affect the copied tree.

REFERENCES

1) Online C Compiler

2) Turbo C++

3) geeksforgeeks.com

4) learn-c.org

5) programiz.com

6) visual studio

THANK YOU NOTE

We would like to thank our professor of data structure and algorithm Dr. P.Thiyagarajan sir for his continuous guidance and support.

We would like to thank God and congratulate every member (Anupriya, Athar, Praveen, Venkateshwarlu, Ishika, Vardha, Ansad and Rustam) for their continuous effort to make the execution of this project successful.

Thank you everyone!