CC-213L

Data Structures and Algorithms

Laboratory 10

Binary Trees

Version: 1.0.0

Release Date: 10-12-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Non-Linear Data Structure
- Array Binary Tree
- Linked Binary Tree

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

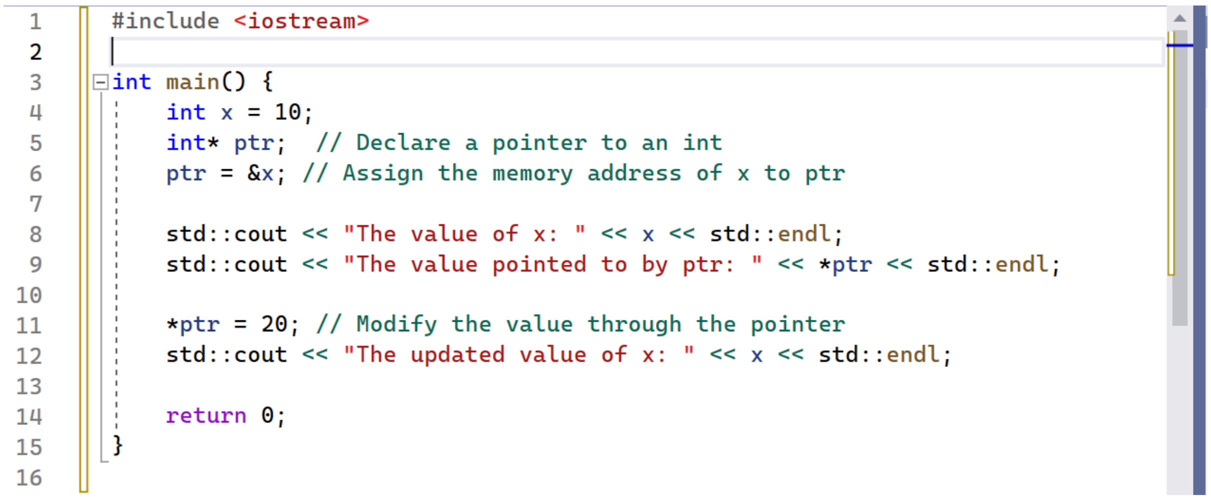
- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Tahir Mustafvi	bcs20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

Background and Overview

Pointers and Dynamic Memory Allocation: Pointers and dynamic memory allocation are important concepts in programming, particularly in languages like C and C++. Pointers allow you to work with memory addresses, while dynamic memory allocation allows you to manage memory at runtime.

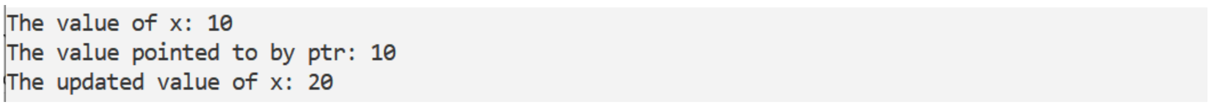
Pointers: A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access the value of the variable stored at that address. Pointers are often used for various purposes, such as dynamically allocated memory, working with arrays, and passing functions as arguments.



```cpp
#include <iostream>

int main() {
    int x = 10;
    int* ptr;  // Declare a pointer to an int
    ptr = &x; // Assign the memory address of x to ptr

    std::cout << "The value of x: " << x << std::endl;
    std::cout << "The value pointed to by ptr: " << *ptr << std::endl;

    *ptr = 20; // Modify the value through the pointer
    std::cout << "The updated value of x: " << x << std::endl;

    return 0;
}
```

Figure 1(Pointers)

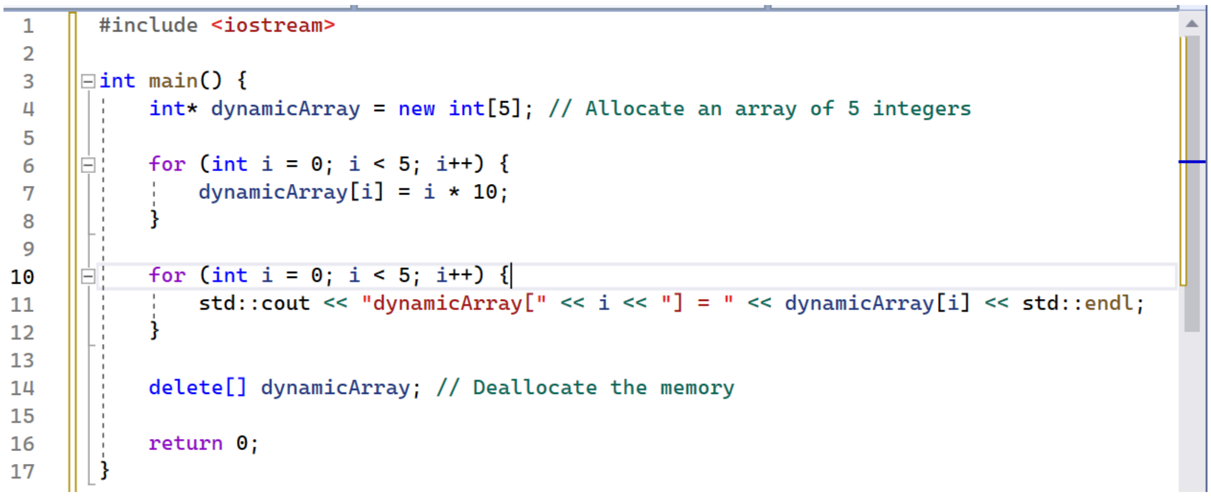
Explanation: In this example, **ptr** is a pointer to an integer, and it is assigned the memory address of the variable x. You can access and modify the value of x through the pointer using the dereference operator (ptr).



```
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

Figure 2(output)

Dynamic Memory Allocation: Dynamic memory allocation allows you to allocate memory for variables at runtime. In C++, you can use new and delete operators to allocate and deallocate memory for objects on the heap.

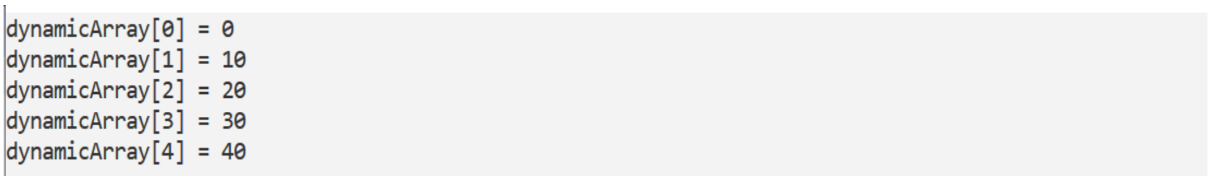


```cpp
#include <iostream>

int main() {
    int* dynamicArray = new int[5]; // Allocate an array of 5 integers

    for (int i = 0; i < 5; i++) {
        dynamicArray[i] = i * 10;
    }

    for (int i = 0; i < 5; i++) {
        std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
    }

    delete[] dynamicArray; // Deallocate the memory

    return 0;
}
```

Figure 3(Dynamic Memory)

Explanation:

In this example, dynamicArray is allocated on the heap with space for 5 integers. After using it, it is essential to deallocate the memory using **delete[]** to prevent memory leaks. Note that in modern C++ (C++11 and later), it is recommended to use smart pointers like **std::unique_ptr** and **std::shared_ptr** for better memory management, as they automatically handle memory deallocation.



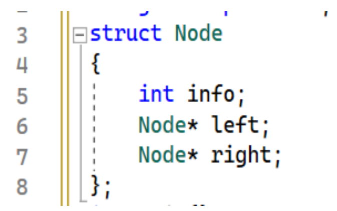
```
dynamicArray[0] = 0
dynamicArray[1] = 10
dynamicArray[2] = 20
dynamicArray[3] = 30
dynamicArray[4] = 40
```

Figure 4(Output)

Self-Referential Objects (Single Self Reference): Classes that have capability to refer to their own types of objects are called **Self Referential Classes/Structs.** Objects of such classes are called self-referential Objects.

Self-referential structure in C++ are those structure that contains one or more than one pointer as their member which will be pointing to the structure of the same type. In simple words, a structure that is pointing to the structure of the same type is known as a self-referential structure.

Example in C++



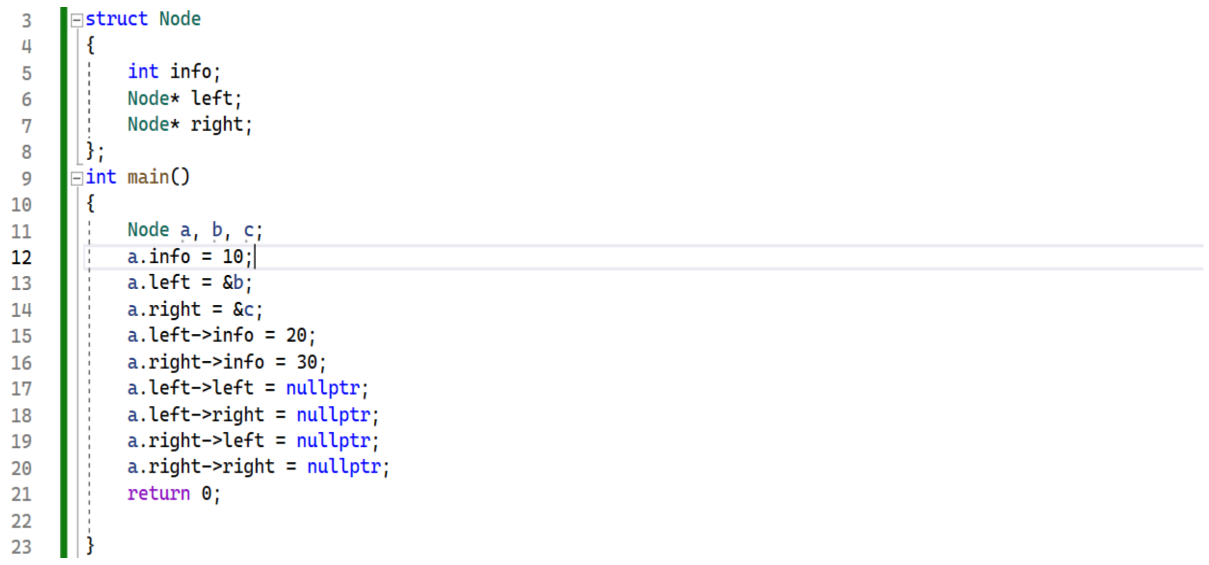
```cpp
struct Node
{
    int info;
    Node* left;
    Node* right;
};
```

Figure 5(Self Referencing)

Explanation:

In Figure 5 we have declared a struct Node. It has three data members info, left and right pointers to Node. **Info** Represents the information data part. Enables the object to store relevant information in it. There can be more than one identifier of same/different datatypes depending upon the application /situation.

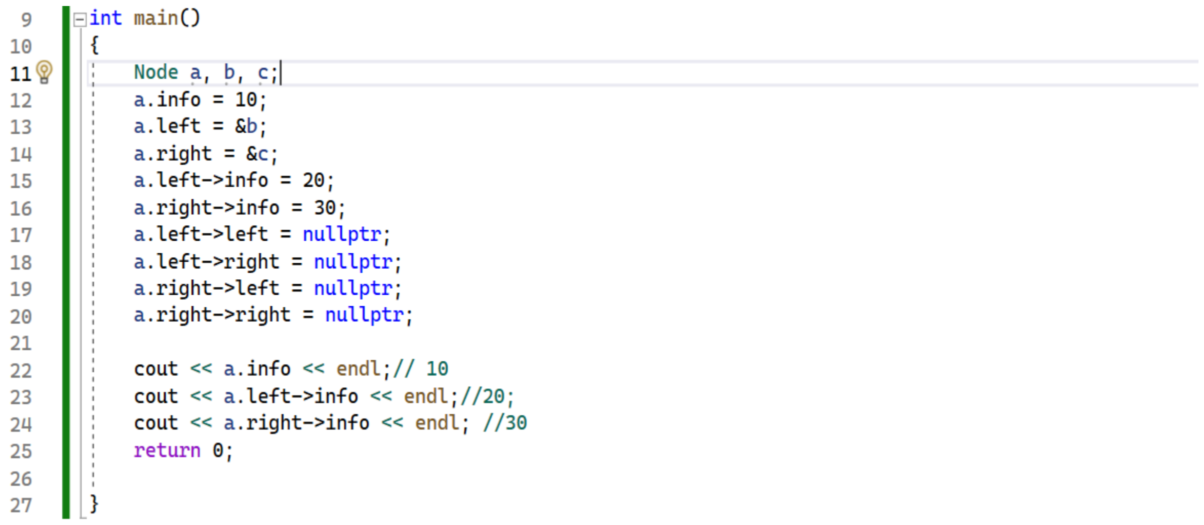
left and right Represents the link part. Enables the object to a self-referential object. There can be more than one such references used for different purposes in different applications /situations.



```cpp
struct Node
{
    int info;
    Node* left;
    Node* right;
};
int main()
{
    Node a, b, c;
    a.info = 10;
    a.left = &b;
    a.right = &c;
    a.left->info = 20;
    a.right->info = 30;
    a.left->left = nullptr;
    a.left->right = nullptr;
    a.right->left = nullptr;
    a.right->right = nullptr;
    return 0;

}
```

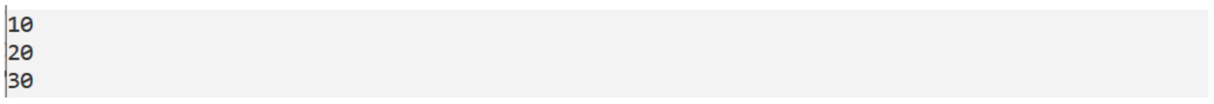
Figure 6(Self Referential Objects)

Here we have displayed the values that are stored in those variables that are self-referenced.



```cpp
int main()
{
    Node a, b, c;
    a.info = 10;
    a.left = &b;
    a.right = &c;
    a.left->info = 20;
    a.right->info = 30;
    a.left->left = nullptr;
    a.left->right = nullptr;
    a.right->left = nullptr;
    a.right->right = nullptr;

    cout << a.info << endl;// 10
    cout << a.left->info << endl;//20;
    cout << a.right->info << endl; //30
    return 0;

}
```

Figure 7(Self-Referencing)



```
10
20
30
```

Figure 8(Output)

Non-Linear Data structures

Non-linear data structures are data structures in which elements are not arranged in a sequential, linear manner. Unlike linear data structures (e.g., arrays, linked lists) where elements are stored in a linear order, non-linear data structures allow for more complex relationships among elements. Here are some examples of non-linear data structures:

1. **Trees**
   - Binary Tree: Each node has at most two children.
   - Binary Search Tree (BST): A binary tree where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only nodes with keys greater than the node's key.
   - AVL Tree: A self-balancing binary search tree where the height of the two child subtrees of every node differs by at most one.

2. **Graphs**
   - Directed Graph (Digraph): A graph in which edges have a direction.
   - Undirected Graph: A graph in which edges do not have a direction.
   - Weighted Graph: A graph in which each edge has an associated weight.

3. **Heaps**
   - Binary Heap: A complete binary tree where the value of each node is greater than or equal to (or less than or equal to) the values of its children.
   - Max Heap: A binary heap where the value of each node is greater than or equal to the values of its children.
   - Min Heap: A binary heap where the value of each node is less than or equal to the values of its children.

These non-linear data structures are essential in various applications and are chosen based on the specific requirements and characteristics of the data and the operations to be performed on them.

Binary Trees

Binary trees are a type of tree data structure in which each node has at most two children, which are referred to as the left child and the right child. These children are distinguished as being either the "left" or "right" child. The topmost node in a binary tree is called the root, and nodes with no children are called leaves. Here are some common types of binary trees:

General Binary Tree: In a general binary tree, each node can have at most two children. However, there are no strict rules about how the children are organized, making it a more general form.

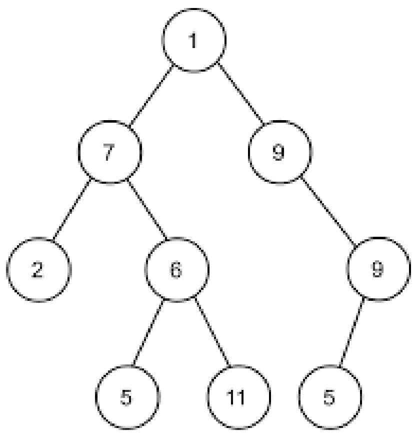


Figure 9(Binary Tree)

Binary Search Tree (BST): A binary search tree is a binary tree in which each node has a value, and the values of nodes in the left subtree are less than the value of the root, while the values in the right subtree are greater. This property makes searching for a specific value more efficient compared to a general binary tree.

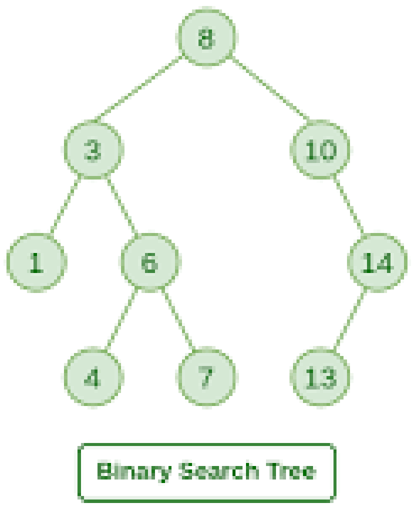


Figure 10(Binary Search Tree)

Complete Binary Tree: A complete binary tree is a binary tree in which all levels are filled, except possibly for the last level, which is filled from left to right. This makes it a balanced tree structure.

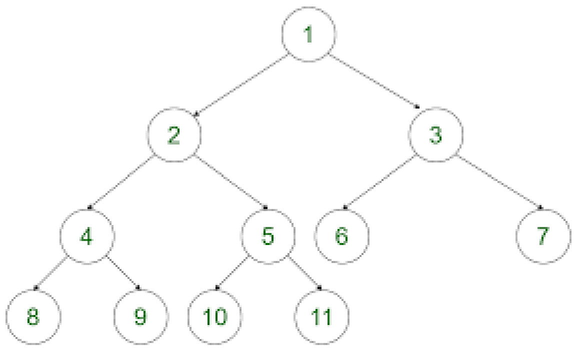


Figure 11(Complete Binary Tree)

Full Binary Tree: A full binary tree is a binary tree in which every node has either 0 or 2 children. In other words, every node is either a leaf or has two children.

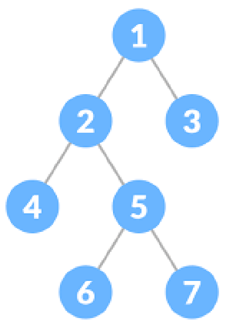


Figure 12(Full Binary Tree)

Perfect Binary Tree: A perfect binary tree is both full and complete, meaning all levels are completely filled, and all nodes have either 0 or 2 children.

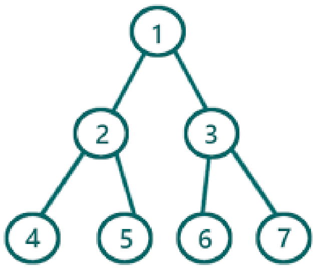


Figure 13(Full Binary Tree)

Balanced Binary Tree: A balanced binary tree is a tree in which the height of the two child subtrees of any node differs by at most one. This property helps maintain a relatively balanced structure, ensuring efficient operations.

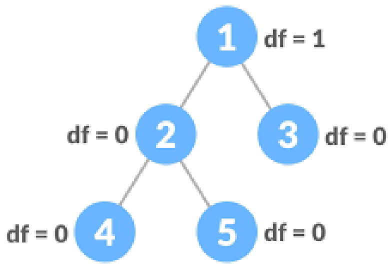


Figure 14(Balanced Binary Tree)

Degenerate (or pathological) Tree: A degenerate tree is a tree where each parent node has only one associated child node. In this case, the tree essentially becomes a linked list.

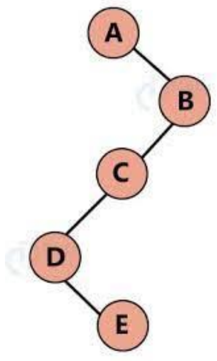


Figure 15(Pathological Tree)

Binary trees find applications in various algorithms and data structures. Binary search trees, in particular, are widely used for efficient searching and sorting operations. The specific type of binary tree chosen depends on the requirements of the application and the desired characteristics of the tree structure.

Binary Tree Implementation

Array based Binary Tree: In an array-based implementation of a binary tree, the elements of the tree are stored in a one-dimensional array, and the relationships between nodes are determined by the indices of the array. The root of the tree is stored at the first position (index 0) of the array. For any node at index `i`, its left child is located at index 2×i + 1, and its right child is at index 2×i + 2. This mapping follows a level-order traversal of the binary tree. The array is dynamically resized as needed to accommodate new elements, ensuring that there is enough space for the growing tree structure. This approach provides a simple and memory-efficient representation of a binary tree, although it may not be as suitable for scenarios involving frequent insertions and deletions, as these operations may require resizing the array and updating indices, potentially leading to performance overhead.

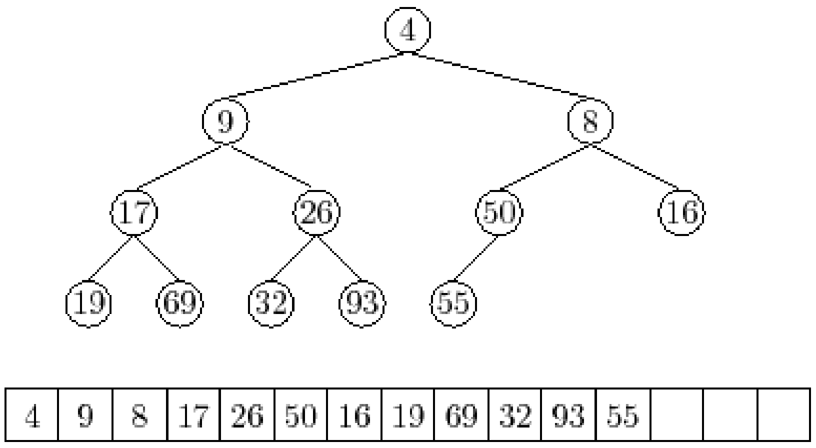
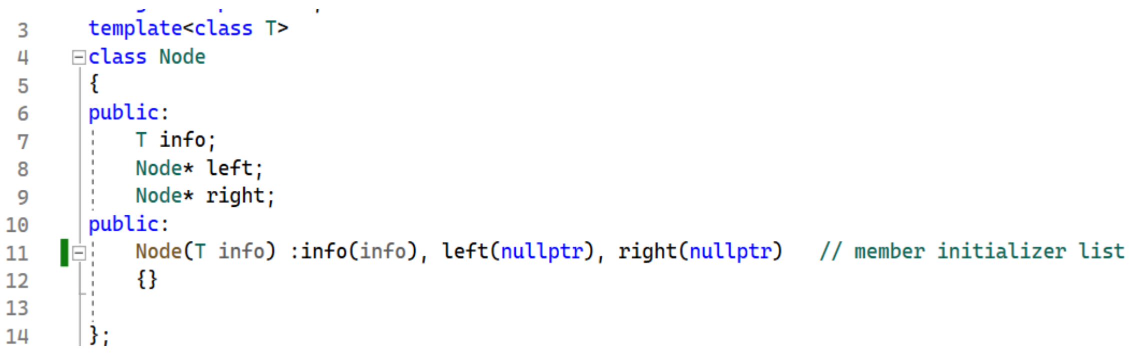


Figure 16(Array Based Binary Tree)

2-Linked Binary Tree

A linked list-based implementation of a binary tree involves using nodes and pointers similar to how a linked list is structured. In a binary tree, each node has at most two children, often referred to as the left child and the right child. The linked structure is used to represent these relationships between nodes. Here's a basic structure for a node in a linked list-based binary tree:



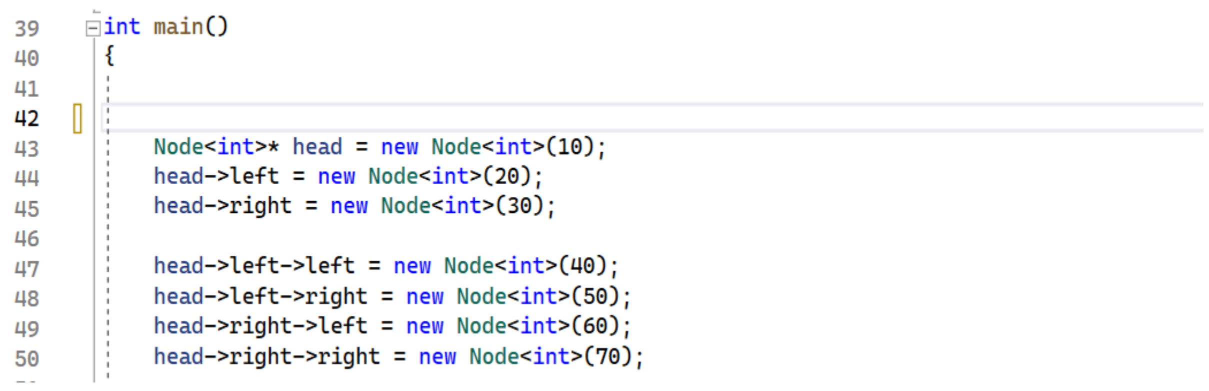
```
template<class T>
class Node
{
public:
    T info;
    Node* left;
    Node* right;
public:
    Node(T info) :info(info), left(nullptr), right(nullptr)   // member initializer list
    {}

};
```

Figure 17(Linked Binary Tree)

Explanation:

In this class: Node type **info** represents the value stored in the node. **left** and **right** are pointers to the left and right children of the current node, respectively. The absence of a child is indicated by a NULL pointer. To build a binary tree, you create nodes and link them together using these pointers. The root of the tree is the first node, and each subsequent node is added as a left or right child of another node.

Insert into Node:



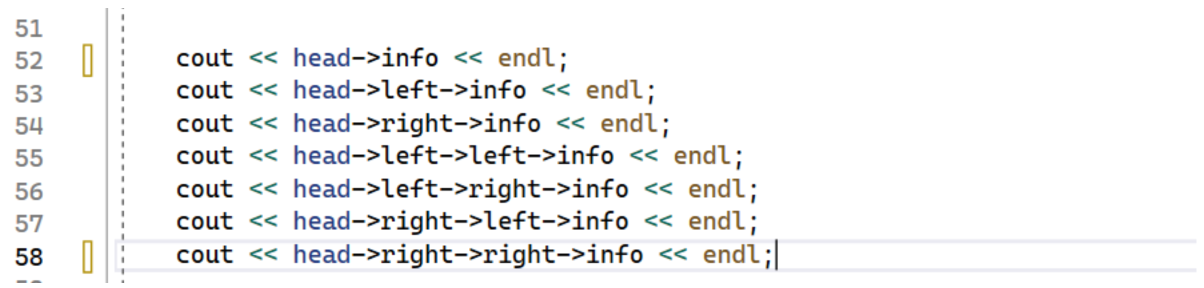
```
39  int main()
40  {
41  
42  
43      Node<int>* head = new Node<int>(10);
44      head->left = new Node<int>(20);
45      head->right = new Node<int>(30);
46  
47      head->left->left = new Node<int>(40);
48      head->left->right = new Node<int>(50);
49      head->right->left = new Node<int>(60);
50      head->right->right = new Node<int>(70);
```

Figure 18(Linked Binary Tree)

Explanation:

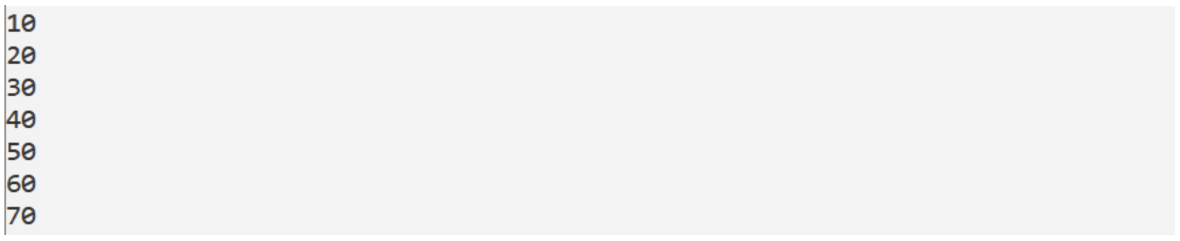
At line 44 a pointer to Node type int declared. head identifier can save address of integer type Node objects.

Display Node



```

    cout << head->info << endl;
    cout << head->left->info << endl;
    cout << head->right->info << endl;
    cout << head->left->left->info << endl;
    cout << head->left->right->info << endl;
    cout << head->right->left->info << endl;
    cout << head->right->right->info << endl;
```

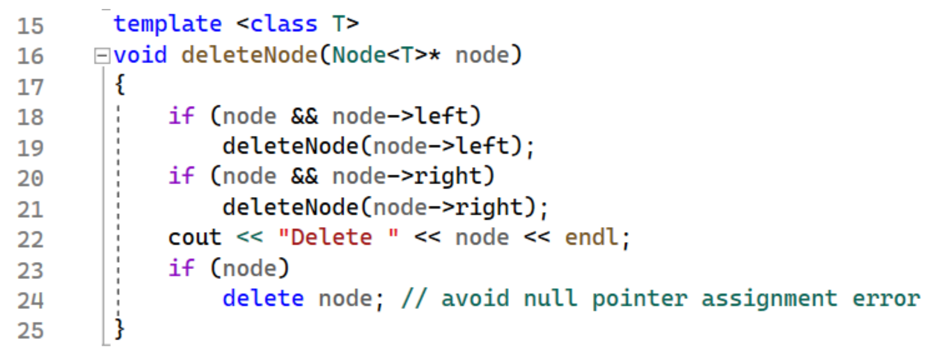
Figure 19(Display Nodes)



```
10
20
30
40
50
60
70
```

Figure 20(Output)

Delete Node



```
template <class T>
void deleteNode(Node<T>* node)
{
    if (node && node->left)
        deleteNode(node->left);
    if (node && node->right)
        deleteNode(node->right);
    cout << "Delete " << node << endl;
    if (node)
        delete node; // avoid null pointer assignment error
}
```

Figure 21(Binary Tree Deletion)

Activities

Pre-Lab Activities:

Task 01:Array Binary Tree

Implement Array Based Binary Tree ADT and add these members function to that ADT.

1. Constructor
2. Destructor
3. Copy-Constructor
4. void setLeftChild (T parentKey, T value)
5. void setRightChild (T parentKey, T value)
6. T getParent ()

Task 02:Heap implementation

Implement Array Based MIN-HEAP ADT and add these members function to that ADT.

1. Constructor
2. Destructor
3. Copy-Constructor
4. void insert (T value)
5. void deleteMin ()