CC-213L

Data Structures and Algorithms

Laboratory 10

Binary Search Tree and AVL

Version: 1.0.0

Release Date: 20-10-2023

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
- Binary Search Tree
- AVL

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 Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid @pucit.edu.pk
Teacher Assistants	Muhammad Nabeel	bitf20m009@pucit.edu.pk
	Abdul Rafay Zubairi	bcsf20a032@pucit.edu.pk

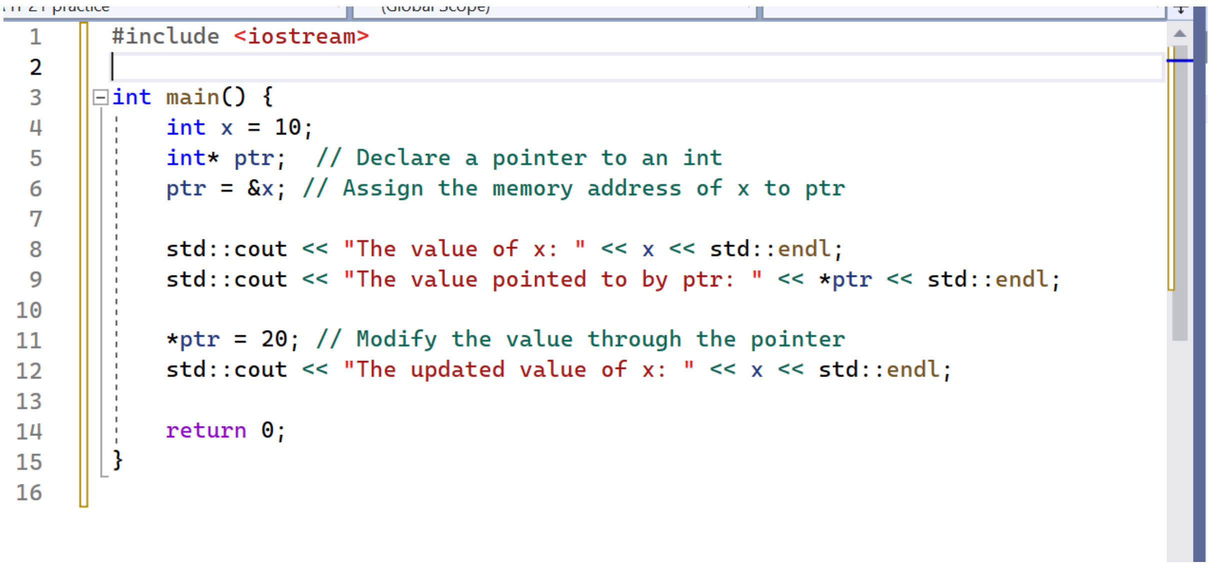
Background and Overview

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.

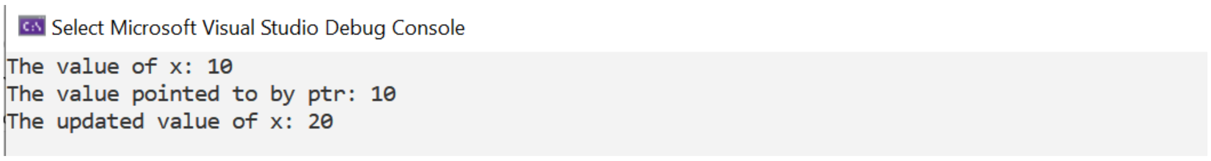


```cpp
#include <iostream>

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).

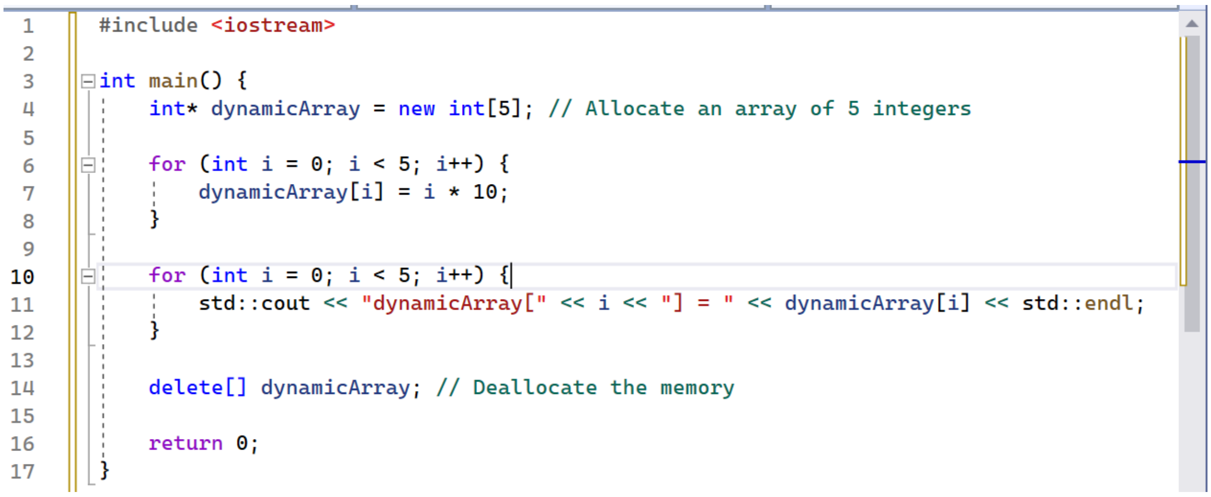


```
Select Microsoft Visual Studio Debug Console
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

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.



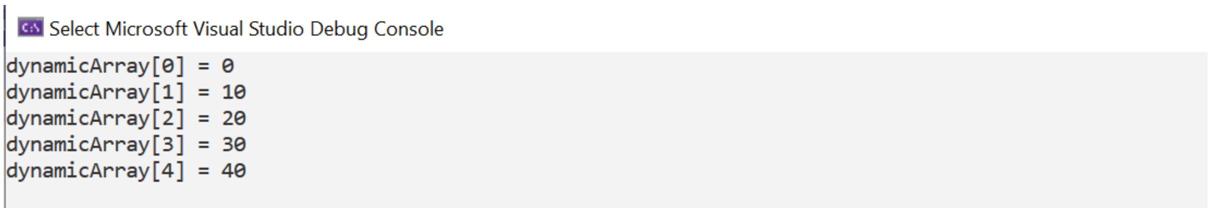
```cpp
#include <iostream>

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: 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.



Select Microsoft Visual Studio Debug Console

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

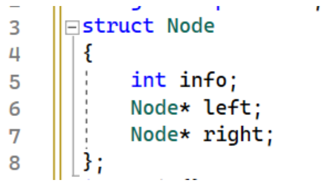
Figure 4(Output)

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;
};
```

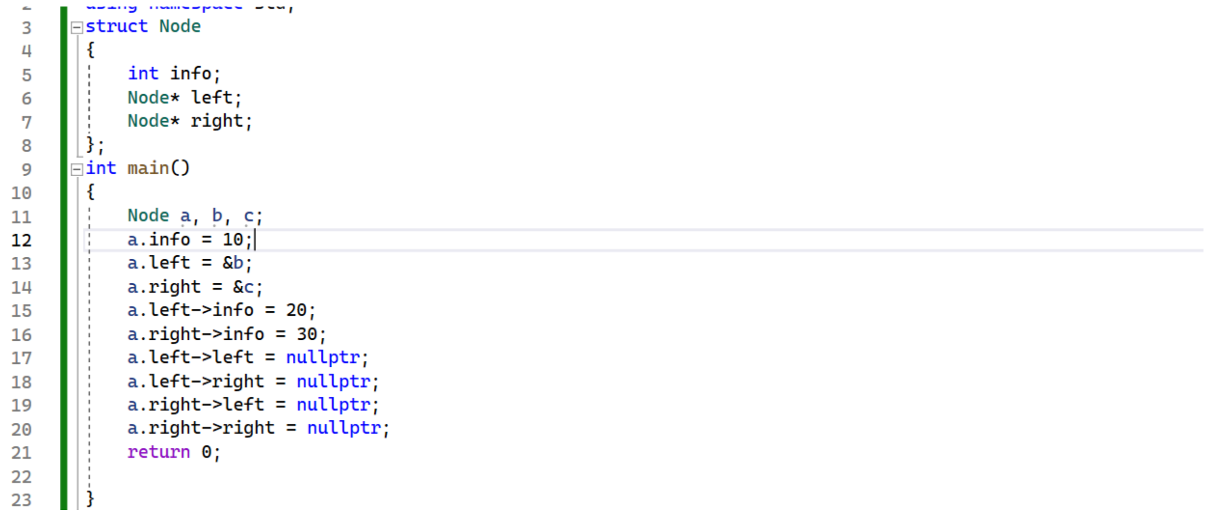
Figure 5(Self Referencing)

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.

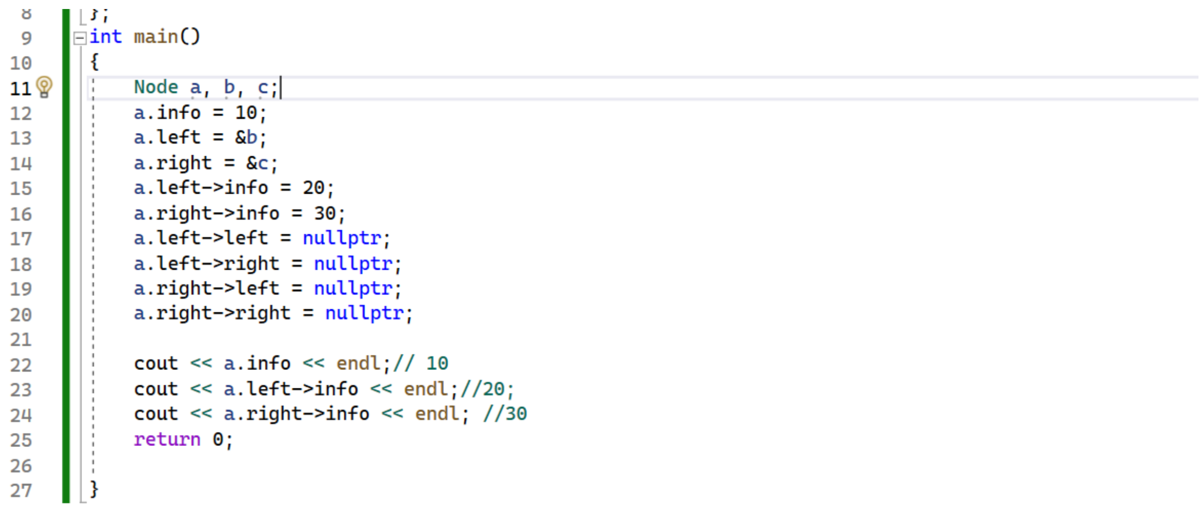
left and right Represents the link part. Enables the object to a self-referential object. There can be more than one such references used for different purposes in different applications /situations.



```
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;

}
```

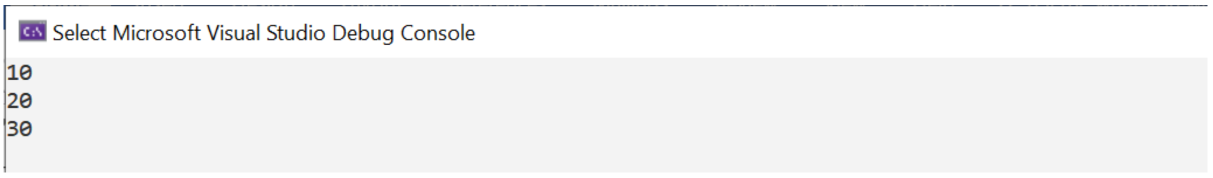
Figure 6(Self Referential Objects)

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



```
};
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)


Select Microsoft Visual Studio Debug Console

```
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.

4. Hash Tables:

- Hash Map: A data structure that implements an associative array abstract data type, a structure that can map keys to values.
- Open Addressing: A technique in hash tables where collisions are resolved by finding the next open slot in the hash table.
- Separate Chaining: A technique in hash tables where each bucket (or slot) in the hash table holds a linked list of elements.

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:

1. 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.

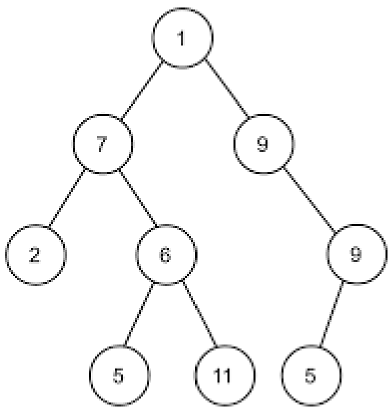


Figure 9(Binary Tree)

2. 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.

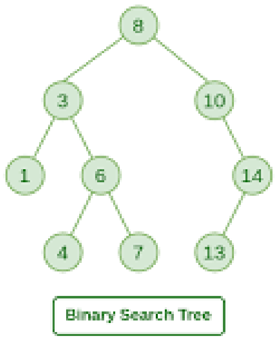


Figure 10(Binary Search Tree)

3.AVL

AVL refers to a type of self-balancing binary search tree named after its inventors Adelson-Velsky and Landis. AVL trees maintain balance during insertions and deletions to ensure that the tree remains relatively balanced, which helps to keep the search time for elements logarithmic.

The balance property of an AVL tree is defined by the heights of the two child subtrees of every node, which should not differ by more than one. If at any time during an insertion or deletion operation the balance factor violates this property, rotations are performed to restore balance.

There are four possible rotations in AVL trees: left rotation, right rotation, left-right rotation, and right-left rotation. These rotations are applied to adjust the balance factors of nodes and maintain the overall balance of the tree.

The self-balancing property of AVL trees ensures that the worst-case time complexity for basic operations such as search, insertion, and deletion remain logarithmic, making them efficient for dynamic sets and dictionaries where these operations are commonly performed.

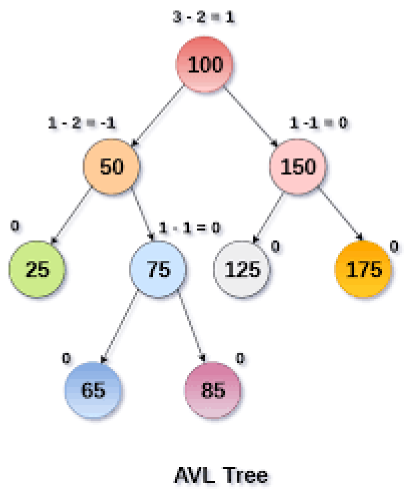
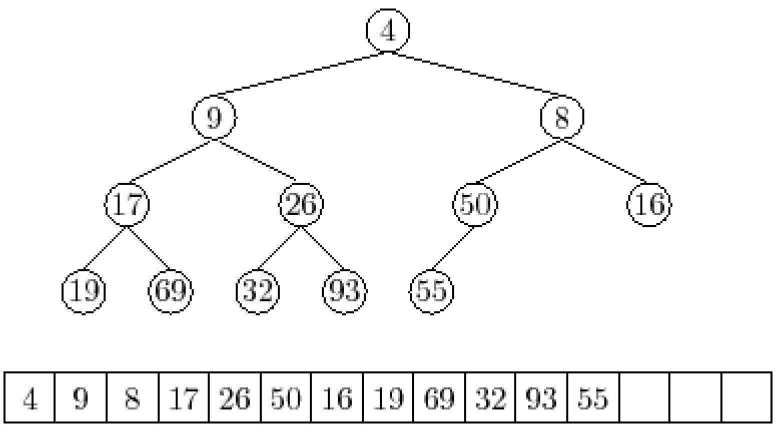


Figure 11(AVL)

Binary Tree Implementation:

1-Array based Binary Tree

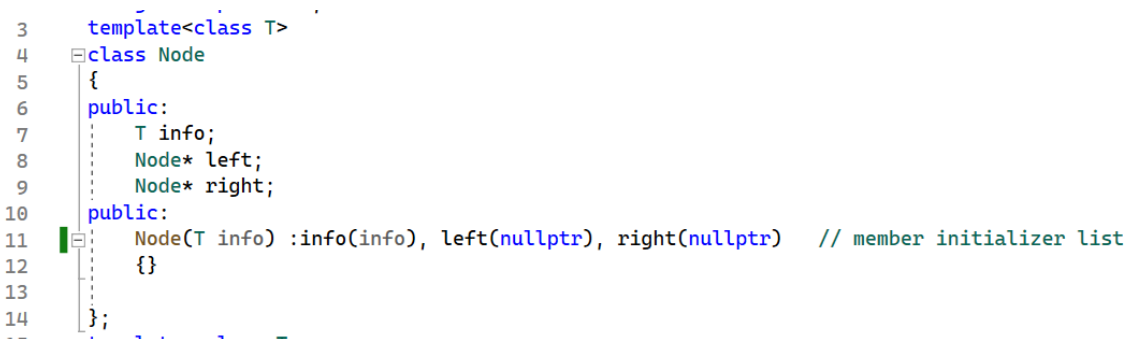
In an array-based implementation of a binary tree, the elements of the tree are stored in a one-dimensional array, and the relationships between nodes are determined by the indices of the array. The root of the tree is stored at the first position (index 0) of the array. For any node at index `i`, its 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.



4	9	8	17	26	50	16	19	69	32	93	55			

Figure 12(Array Based Binary Tree)

2-Linked Binary Tree



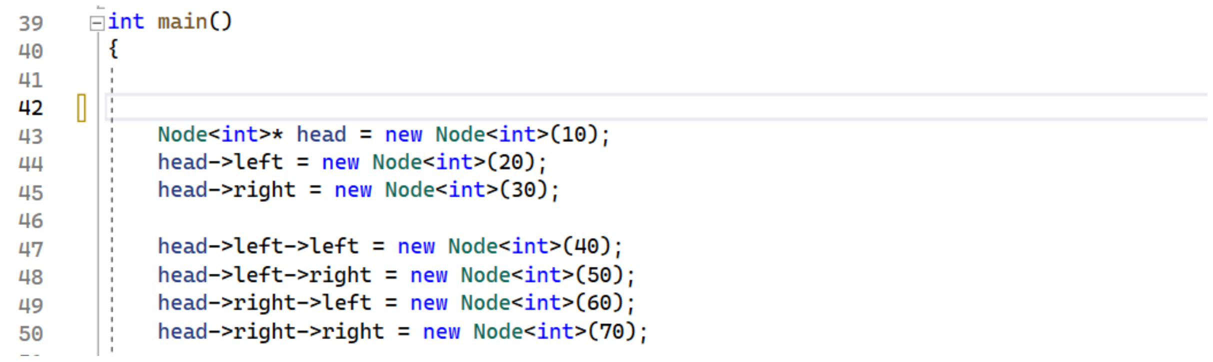
```
3   template<class T>
4   class Node
5   {
6   public:
7       T info;
8       Node* left;
9       Node* right;
10  public:
11      Node(T info) :info(info), left(nullptr), right(nullptr)    // member initializer list
12      {}
13
14  };
```

Figure 13(Linked Binary Tree)

Explanation

At line 4 a templated class has been declared that has a T type info variable. There are two other Pointer to nodes left and right that can save address of another Node type T in them.

Insert into Node



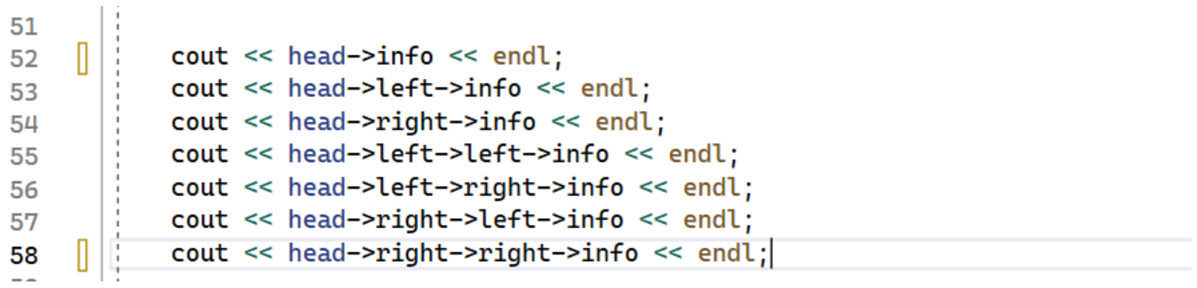
```
int main()
{


    Node<int>* head = new Node<int>(10);
    head->left = new Node<int>(20);
    head->right = new Node<int>(30);

    head->left->left = new Node<int>(40);
    head->left->right = new Node<int>(50);
    head->right->left = new Node<int>(60);
    head->right->right = new Node<int>(70);
```

Figure 14(Linked Binary Tree)

Explanation:

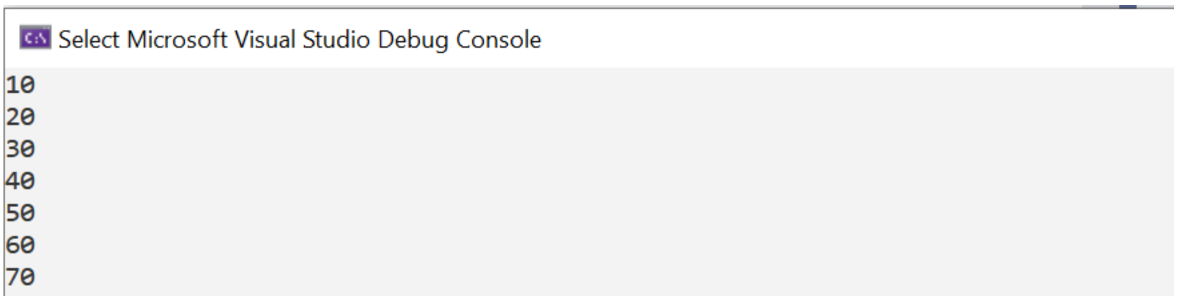
At line 44 a pointer to Node type int declared. head identifier can save address of integer type Node objects.

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 15(Display Nodes)

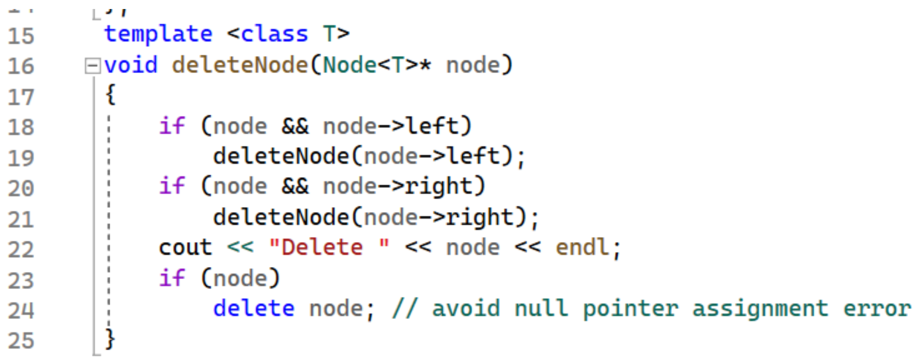


Select Microsoft Visual Studio Debug Console

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

Figure 16(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 17(Binary Tree Deletion)

Binary Search Tree

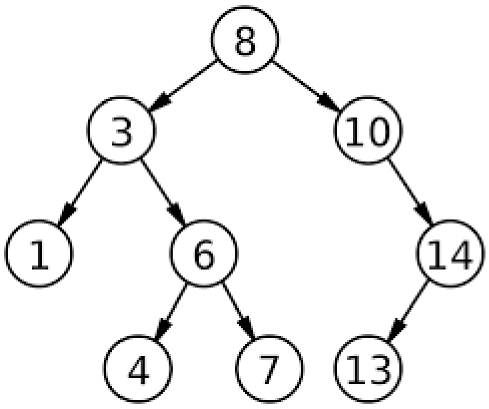


Figure 18(BST)

Activities

Pre-Lab Activities:

Task 01: Binary SearchTree Implementation

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

```
template <class T>
class BST;
template <class T>
class BSTNode
{
	friend BST<T>;
	T data;
	BSTNode<T>* left;
	BSTNode<T>* right;
	// Methods...
};

template <class T>
class BST
{
	BSTNode<T>* root;
	// Methods...
};
```

1.Constructor, Destructor, Copy-Constructor

2.void setRoot(T value);

3.void insert (T value);

4.BSTNode<T>* getLeftChild(BSTNode<T>* node);

5. BSTNode<T>* getRightChild(BSTNode<T>* node);

6.BSTNode<T>* search (T value);

7.void deleteNode (BSTNode<T>* node);

8.void printNodes (T parentKey, T value) ;// use in-order traversal

In-Lab Activities:

Task 01: Dry run

Tell this is a BST or not. What will be the final shapes of the tree if 30,80 ,70 and 20 nodes are removed step by step. Apply for each individual node. Draw the shapes of the trees after the removal of these nodes in such a way that still the tree should be BST.

(a)

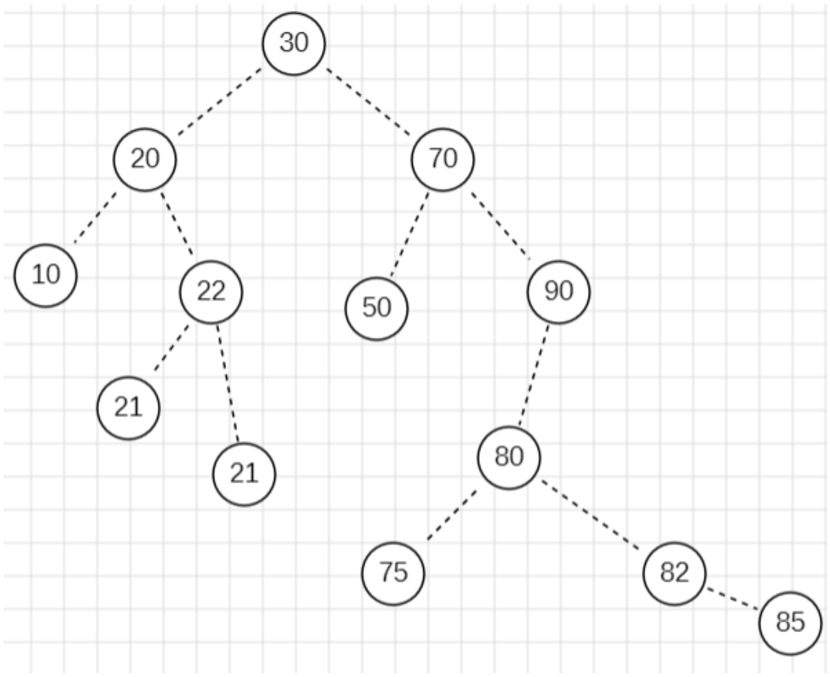
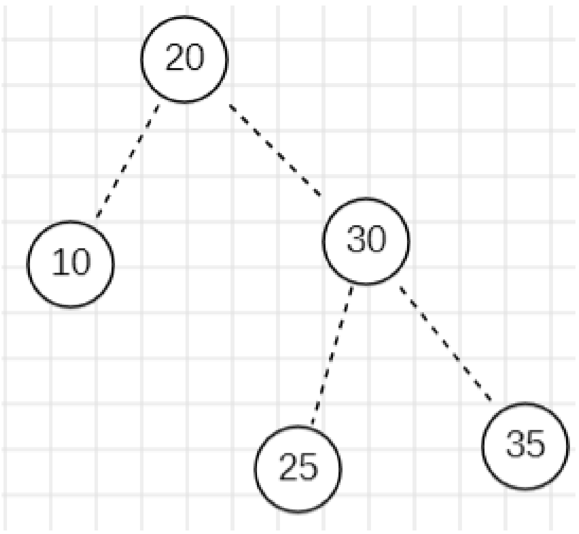


Figure 19(Binary Search Trees)

(b)

Given the BST. Find whether this is AVL or not. After that insert these values and write balance factor for each insertion in the tree and balance your Tree too. The final tree should be AVL

Values: 5,40,45,34,



Task 02: Add public member functions

In your pre-lab you have implemented BST ADT. Add these public members function to Binary Search Tree ADT.

1.void deleteNode (BSTNode<T>* node);

2.boolean isBST (BSTNode<T>*root); // this function can be overloaded like this can tell about the calling instance whether it is BST or can take another BST object pointer too.

3. boolean isEqual(BSTNode<T>* r1,BSTNode<T>*r2);

Takes roots of two trees and as input parameter and returns true if they are equal.

4.boolean isInternalNode(BSTNode<T>* node);

Returns true if given node is an internal node. Where, internal Node is one which has degree greater than zero.

5.bool isExternalNode(BSTNode<T>* node);

Returns true if given node is an external node. External Node is one which has degree equal to zero.

6 int getHeight ( );

Returns the height of tree. If u want to write recursive function for height calculation then call your recursive function in this function and make it helper/driver of recursive function.

7. T getKSmallestNode(int k);

Given an integer k, return *the* k_{th} *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

8. T getMinimumDiff();

Return *the minimum absolute difference between the values of any two different nodes in the tree*.

Post-Lab Activities

Task 01: Add Member Functions

Add these public members function to BST class.

1. void displayDescedents (T val);

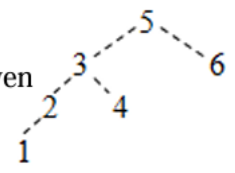
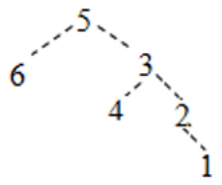
Display decedents of the node containing given value.

2. void displayAncestors (T val);

Display Ancestors of the node containing given value.

3. BST getMirrorImage ( );

Returns the mirror image of *this tree.

For Example, given  you should return 

4.int getNodeCount(BSTNode<T>* node);

This function returns the node count of BST.

5.T findMin();

Returns the min value stored in your BST.

6.T findMax();

Returns the max value stored in your BST.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's

Division of Pre-Lab marks:	**[50 marks]**
Task 01: BST Implementation	[50 marks]
Division of In-Lab marks:	**[60 marks]**
Task 01: Tree Balance	[15marks]
Task 02: BST methods	[45marks]
Division of Post-Lab marks:	**[30 marks]**
Task 01: Add member Functions	[30 marks]

References and Additional Material:

Binary Search Tree

https://www.geeksforgeeks.org/binary-search-tree-data-structure/

AVL

https://www.geeksforgeeks.org/introduction-to-avl-tree/

Lab Time Activity Simulation Log:

Slot – 01 – 02:00 – 00:15:	Class Settlement
Slot – 02 – 02:15 – 02:30:	In-Lab Task 01
Slot – 03 – 02:30 – 02:45:	In-Lab Task 01
Slot – 04 – 02:45 – 03:00:	In-Lab Task 01
Slot – 05 – 03:00 – 03:15:	In-Lab Task 02
Slot – 06 – 03:15 – 03:30:	In-Lab Task 02
Slot – 07 – 03:30 – 03:45:	In-Lab Task 02
Slot – 08 – 03:45 – 04:00:	In-Lab Task 02
Slot – 09 – 04:00 – 04:15:	In-Lab Task 02
Slot – 10 – 04:15 – 04:30:	In-Lab Task 02
Slot – 11 – 4:300 – 04:45:	In-Lab Task 02
Slot – 12 – 04:45 – 05:00:	Discussion on Post-Lab