

**Kathmandu University**

**Department of Computer Science and  
Engineering**

**Dhulikhel, Kavre**



**A Lab Report  
On**

**“Implementing Non-linear Data Structures - Tree”**

**[Code No.: COMP 202]**

**Submitted by**

**Hridayanshu Raj Acharya (1)**

**Sankalp Acharya (2)**

**UNG-CE (II/I)**

**Submitted to**

**Dr. Rajani Chulyadyo**

**Department of Computer Science and Engineering**

**Submission Date: 2024/06/02**

# Lab Report 3

## Tree

A tree is a non-linear data structure where the data are organized in a hierarchical manner.

A tree is a finite set of one or more nodes such that

1. There is a specially designated node called the root.
2. The remaining nodes are partitioned into  $n \geq 0$  disjoint sets  $T_1, T_2, \dots, T_n$ , where each of these sets is a tree.  $T_1, T_2, \dots, T_n$  are called the subtrees of the root

## Binary Search Tree (BST)

A binary search tree (BST) is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Each node has exactly one key and the keys in the tree are distinct.
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

## Operations

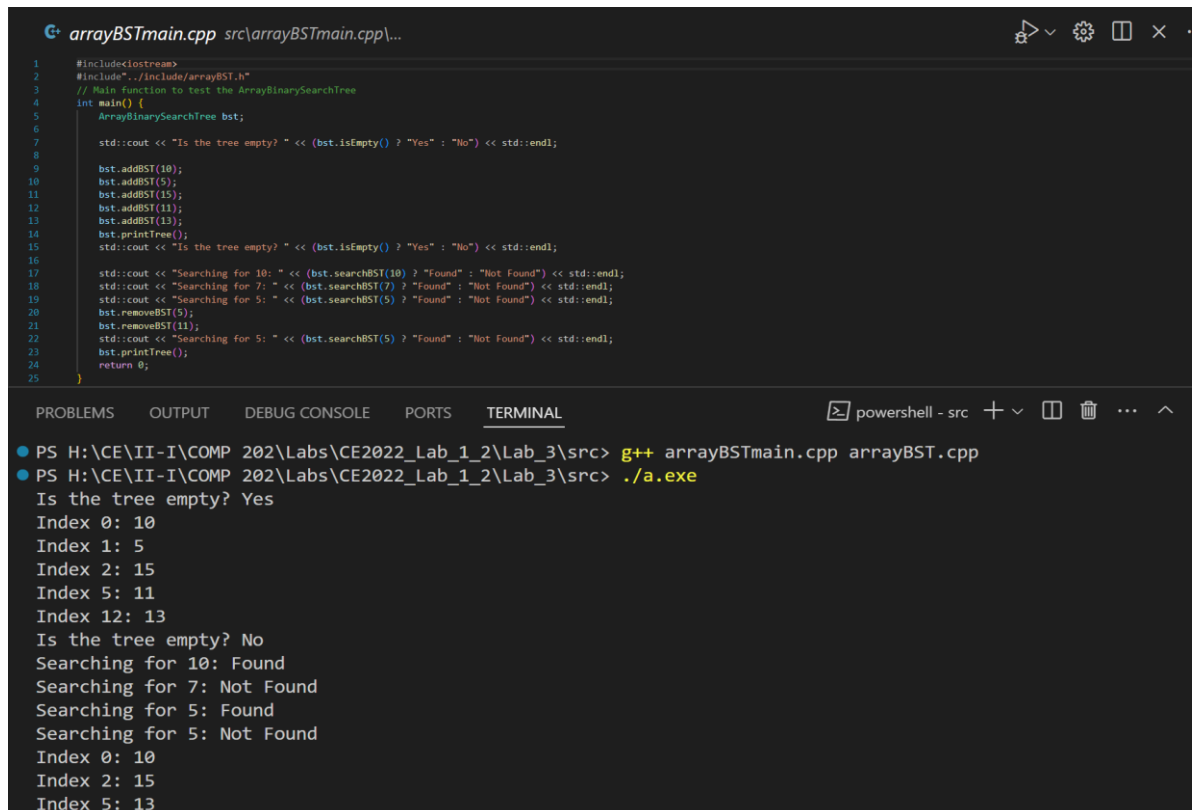
- isEmpty(): Returns true if the tree is empty, and false otherwise.
- addBST(newNode): Inserts an element to the BST.
- removeBST(keyToDelete): Removes the node with the given key from the BST.
- searchBST(targetKey): Returns true if the key exists in the tree, and false otherwise.

## Code

Github Link: [https://github.com/hridayanshu236/CE2022\\_Lab\\_1\\_2](https://github.com/hridayanshu236/CE2022_Lab_1_2)

## Binary Search Tree using array

When implementing a Binary Search Tree (BST) using arrays, we typically allocate a fixed-size array where each index represents a node in the tree. This approach is straightforward and memory-efficient, but it requires careful management of the array indices to maintain the BST property. Insertions and deletions might involve shifting elements in the array to accommodate the changes, which can be computationally expensive, especially if the array is large. However, searching in an array-based BST is relatively efficient, as it offers constant-time access to elements.



```
1 #include<iostream>
2 #include "../include/arrayBST.h"
3 // Main function to test the ArrayBinarySearchTree
4 int main() {
5     ArrayBinarySearchTree bst;
6
7     std::cout << "Is the tree empty? " << (bst.isEmpty() ? "Yes" : "No") << std::endl;
8
9     bst.addBST(10);
10    bst.addBST(5);
11    bst.addBST(15);
12    bst.addBST(11);
13    bst.addBST(13);
14    bst.printTree();
15    std::cout << "Is the tree empty? " << (bst.isEmpty() ? "Yes" : "No") << std::endl;
16
17    std::cout << "Searching for 10: " << (bst.searchBST(10) ? "Found" : "Not Found") << std::endl;
18    std::cout << "Searching for 7: " << (bst.searchBST(7) ? "Found" : "Not Found") << std::endl;
19    std::cout << "Searching for 5: " << (bst.searchBST(5) ? "Found" : "Not Found") << std::endl;
20    bst.removeBST(5);
21    bst.removeBST(11);
22    std::cout << "Searching for 5: " << (bst.searchBST(5) ? "Found" : "Not Found") << std::endl;
23    bst.printTree();
24    return 0;
25 }
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL

```
PS H:\CE\II-I\COMP 202\Labs\CE2022_Lab_1_2\Lab_3\src> g++ arrayBSTmain.cpp arrayBST.cpp
PS H:\CE\II-I\COMP 202\Labs\CE2022_Lab_1_2\Lab_3\src> ./a.exe
Is the tree empty? Yes
Index 0: 10
Index 1: 5
Index 2: 15
Index 5: 11
Index 12: 13
Is the tree empty? No
Searching for 10: Found
Searching for 7: Not Found
Searching for 5: Found
Searching for 5: Not Found
Index 0: 10
Index 2: 15
Index 5: 13
```

## Operations:

- isEmpty():** This function checks if the ArrayBST is empty or not. It iterates through the array of nodes and checks if any node is occupied (i.e., isOccupied flag is true). If no occupied node is found, it means the tree is empty, and the function returns true. Otherwise, it returns false.
- addBST(int data):** This function inserts a new node with the given data value into the ArrayBST. It starts from the root node (index 0) and traverses the tree by comparing the data value with the current node's value. If the data value is smaller than the current node's value, it moves to the left child (index =  $2 * \text{index} + 1$ ). Otherwise, it moves to the right child (index =  $2 * \text{index} + 2$ ). This process continues until an empty node is found, where the new node is inserted.
- printTree():** This prints the tree with the index.
- searchBST(int targetKey):** This function searches for a node with the given targetKey value in the ArrayBST. It starts from the root node (index 0) and traverses the tree by comparing the targetKey value with the current node's value, moving to the left or right child accordingly. If a node with the targetKey value is found, it

returns true. Otherwise, it returns false if the key is not present in the tree.

- e. **removeBST(int keyToDelete):** This removeBST function in the ArrayBinarySearchTree class is designed to delete a node from the binary search tree implemented using an array. It follows three main cases:

**Case 1:** Node to be deleted has no children (leaf node):

If the node has no children, it is simply deleted from the array by setting its index to nullptr.

**Case 2:** Node to be deleted has one child:

If the node has only one child, the child node replaces the deleted node in the array. This adjustment ensures that the BST property is maintained.

**Case 3:** Node to be deleted has two children:

If the node has two children, it finds the minimum node in its right subtree. This minimum node's value replaces the value of the node to be deleted, preserving the ordering of the BST.

After replacing the value, the minimum node is recursively deleted. If the minimum node has children, they are handled according to the first two cases.

## Binary Search Tree using Linked List

On the other hand, implementing a BST using a linked list involves dynamically allocating memory for each node. Each node contains pointers to its left and right children, allowing for efficient insertion and deletion operations without the need for shifting elements. However, searching in a linked list-based BST might be less efficient compared to an array-based BST due to the overhead of traversing pointers. Additionally, linked list-based implementations can consume more memory than array-based implementations due to the memory overhead of pointers.

```
LinkedListBSTmain.cpp src\LinkedListBSTmain.cpp\...
1  #include <iostream>
2  #include "../include/LinkedListBST.h"
3  using namespace std;
4
5  // Main function to test the LinkedListBST
6  int main()
7  {
8      LinkedListBST bst;
9
10     cout << "Is the tree empty? " << (bst.isEmpty() ? "Yes" : "No") << endl;
11
12     bst.addBST(10);
13     bst.addBST(5);
14     bst.addBST(15);
15     bst.addBST(3);
16     bst.addBST(7);
17     bst.addBST(12);
18     bst.addBST(18);
19     cout << "Is the tree empty? " << (bst.isEmpty() ? "Yes" : "No") << endl;
20     bst.display();
21     cout << "Searching for 10: " << (bst.searchBST(10) ? "Found" : "Not Found") << endl;
22     cout << "Searching for 7: " << (bst.searchBST(7) ? "Found" : "Not Found") << endl;
23     cout << "Removing 5: " << (bst.removeBST(5) ? "Success" : "Fail") << endl;
24     bst.display();
25     cout << "Searching for 5: " << (bst.searchBST(5) ? "Found" : "Not Found") << endl;
26     cout << "Removing 10: " << (bst.removeBST(10) ? "Success" : "Fail") << endl;
27     bst.display();
28     cout << "Searching for 10: " << (bst.searchBST(10) ? "Found" : "Not Found") << endl;
29     return 0;
30 }
31
```

powerShell - src + -

Is the tree empty? Yes  
Is the tree empty? No

Node data: 3  
Node data: 5  
Node data: 7  
Node data: 10  
Node data: 12  
Node data: 15  
Node data: 18

Searching for 10: Found  
Searching for 7: Found  
Removing 5: Success

Node data: 3  
Node data: 7  
Node data: 10  
Node data: 12  
Node data: 15  
Node data: 18

Searching for 5: Not Found  
Removing 10: Success

Node data: 3  
Node data: 7  
Node data: 12  
Node data: 15  
Node data: 18

Searching for 10: Not Found

## Operations:

- a. **isEmpty():** This function checks if the Binary Search Tree (BST) is empty or not. It returns true if the root of the BST is nullptr (null pointer), indicating that the tree is empty. Otherwise, it returns false.
- b. **addBST(int data):** This function inserts a new node with the given data value into the BST. It follows the rules of a BST, where the value of each node in the left subtree is less than the node's value, and the value of each node in the right subtree is greater than the node's value. The function recursively traverses the tree, comparing the data value with the current node's value, and moving to the left or right subtree accordingly until it reaches a null pointer, where it creates a new node and inserts it.
- c. **removeBST(int keyToDelete):** : This function removes the node with the given keyToDelete value from the BST, if it exists. It first searches for the node with the target key by traversing the tree recursively. If the node is found, the function removes it based on the following cases:
  1. If the node has no children (leaf node), it is simply deleted.
  2. If the node has one child, the child node is connected to the parent of the deleted node.
  3. If the node has two children, the function finds the in-order successor (the smallest value in the right subtree) or the in-order predecessor (the largest value in the left subtree), replaces the node's value with the successor/predecessor value, and then removes the successor/predecessor node recursively.
- d. **inorder(Node \*node):** This function performs an inorder traversal of the BST, which means it visits the nodes in the following order: left subtree, current node, right subtree. It starts from the given node (usually the root of the tree) and recursively traverses its left child, then prints the data of the current node, and finally recursively traverses its right child. The base case is when the node is nullptr, indicating an empty subtree, in which case the function returns.
- e. **display():** This function serves as a wrapper for the inorder traversal function. It initiates the traversal from the root node of the tree. It prints a newline to start a new line for better readability, then calls the inorder function with the root node. After the traversal completes, another newline is printed to separate the output from any subsequent text.
- f. **searchBST(int targetKey):** This function searches for a node with the given targetKey value in the BST. It recursively traverses the tree, comparing the targetKey with the current node's value, and moving to the left or right subtree accordingly. If a node with the targetKey value is found, it returns true. Otherwise, it returns false.

## Conclusion:

Both array-based and linked list-based implementations of BSTs have their pros and cons. Array-based implementations offer efficient search operations but can be inefficient for insertions and deletions, especially in large arrays. Linked list-based implementations, while more flexible in terms of dynamic memory allocation, may suffer from slower search operations and higher memory overhead. The choice between the two implementations depends on the specific requirements of the application, such as the expected size of the tree, the frequency of insertions and deletions, and the importance of memory efficiency.