KATHMANDU UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DHULIKHEL, KAVRE

A

LAB REPORT

ON

"BINARY SEARCH TREE"

Subject Code No: COMP 202

SUBMITTED BY:

Sanjib Dahal (18)

Arjit Chand (17)

Group: CE

Level: II/I

SUBMITTED TO:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Date of Submission: - 2024/06/02

Lab Report 3

In this lab, we implemented binary search tree with array and linkedlist. This report aims to demonstrate the functionality and applications of binary search tree using array and linked list through code demonstrations and outputs.

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 T1, T2, . . . , Tn, where each of these sets is a tree. T1, T2, . . . , Tn 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 on BST

The major operations performed in stack are as follows:

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

GitHub Link to Code

The GitHub link to the repository for the implementation of stack and queue linked list is:
GitHub Repository

Binary Search Tree

Implementation using LinkedBST:

Output:

```cpp
// main.cpp

#include "linked_bst.h"
#include <iostream>
using namespace std;

int main() {
    BST* bst = new LinkedBST();
    cout << "Is the tree empty? " << (bst->isEmpty() ? "Yes" : "No") << endl;

    bst->addBST(5);
    bst->addBST(3);
    bst->addBST(7);
    bst->addBST(2);
    bst->addBST(4);
    bst->addBST(6);

    cout << "Does the tree contain 4? " << (bst->searchBST(4) ? "Yes" : "No") << endl;
    cout << "Does the tree contain 9? " << (bst->searchBST(9) ? "Yes" : "No") << endl;

    bst->removeBST(7);
    cout << "Does the tree contain 7 after removal? " << (bst->searchBST(7) ? "Yes" : "No") << endl;

    delete bst;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE  GITLENS  COMMENTS

Dahal on ~/Desktop/CE2022_Lab3_17_18 main
# g++ -o main main.cpp linked_bst.cpp
Dahal on ~/Desktop/CE2022_Lab3_17_18 main
# ./main.exe
Is the tree empty? Yes
Does the tree contain 4? Yes
Does the tree contain 9? No
Does the tree contain 7 after removal? No
```

Operations:

Short descriptions of the functions for the LinkedBST implementation:

(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) **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.

Implementation using ArrayBST:

Output:

```cpp
#include "array_bst.h"
#include <iostream>
using namespace std;

int main() {
    BST* bst = new ArrayBST();
    cout << "Is the tree empty? " << (bst->isEmpty() ? "Yes" : "No") << endl;

    bst->addBST(5);
    bst->addBST(3);
    bst->addBST(7);
    bst->addBST(2);
    bst->addBST(4);
    bst->addBST(6);
    bst->addBST(8);

    cout << "Does the tree contain 4? " << (bst->searchBST(4) ? "Yes" : "No") << endl;
    cout << "Does the tree contain 9? " << (bst->searchBST(9) ? "Yes" : "No") << endl;

    bst->removeBST(7);
    cout << "Does the tree contain 7 after removal? " << (bst->searchBST(7) ? "Yes" : "No") << endl;

    delete bst;

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE  GITLENS  COMMENTS

Dahal on ~/Desktop/CE2022_Lab3_17_18 main
# g++ -o main main.cpp array_bst.cpp
Dahal on ~/Desktop/CE2022_Lab3_17_18 main
# ./main.exe
Is the tree empty? Yes
Does the tree contain 4? Yes
Does the tree contain 9? No
Does the tree contain 7 after removal? No
```

Operations:

Short descriptions of the functions for the ArrayBST implementation:

(a) **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.

(b) **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 * index + 1). Otherwise, it moves to the right child (index = 2 * index + 2). This process continues until an empty node is found, where the new node is inserted.

(c) **removeBST(int keyToDelete):** This function removes the node with the given keyToDelete value from the ArrayBST, if it exists. It starts from the root node (index 0) and traverses the tree in a similar manner as addBST, comparing the keyToDelete value with the current node's value and moving to the left or right child accordingly. If a node with the keyToDelete value is found, its isOccupied flag is set to false, effectively removing it from the tree.

(d) **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.

Conclusion:

In this lab, we implemented Binary Search Trees (BSTs) using a linked list structure (LinkedBST) and an array-based structure (ArrayBST). The LinkedBST implementation demonstrated dynamic memory allocation and efficient operations with an average time complexity of O(log n) for a balanced tree. The ArrayBST implementation used a fixed-size array and provided similar time complexity, but with a hard limit on the maximum number of nodes.

Both implementations showcased the fundamental properties and operations of BSTs, including insertion, deletion, and search.