COMP 2611 – Data Structures

Lab 5

Part 1: Binary Trees

Previously, the preorder, inorder, and postorder traversals of a binary tree were obtained using recursion. We will now write non-recursive functions for the inorder and preorder traversals.

Recall that an inorder traversal goes left → root → right. When we go down the left subtree of the root, we will need some way to come back up to the root. This can be repeated at many levels and we must come back up the same way we went down to the subtrees.

A stack can be used to store the nodes that were met on the way down, but not visited. On the way back up, the nodes are popped from the stack in the reverse order that they were met. The following is an algorithm for a non-recursive inorder traversal using a stack (Kalicharan, 2008):

```
initialize stack S to empty
set curr = root
set finished = false
while (not finished) {
	while (curr != null) {
		push curr onto S
		set curr = left (curr)
	}
	if (S is empty)
		set finished = true
	else {
		pop S into curr
		visit curr
		set curr = right (curr)
	}
}
```

Download the `Lab5-Part1.zip` file from myElearning. The zip file contains a Dev-C++ project, `Lab5-Part1`, which already contains the code for the stack in `Stack.cpp`. The `BinaryTree.cpp` code is similar to the code that was written during Lab 4.

(a) Write the code for the non-recursive inorder traversal in `BinaryTree.cpp`.
(b) Write the code for the non-recursive preorder traversal in `BinaryTree.cpp`.

Part 2: Binary Search Trees

A binary search tree (BST) is a binary tree where the keys stored at each node satisfy the *binary-search-tree property*:

- Let *x* be a node in a BST.
- If *y* is a node in the left subtree of *x*, then y.key ≤ x.key.
- If *y* is a node in the right subtree of *x*, then y.key ≥ x.key.

Download the `Lab5-Part2.zip` file from myElearning. The zip file contains a Dev-C++ project. You must unzip the files to a different folder from Part 1 since it contains files with the same names.

Write the code for each problem below in the `BinarySearchTree.cpp` file.

(a) In the *main* function of `BinarySearchTree.cpp`, write code to create the BST in Figure 1 (this is the binary search tree that was used in the lecture of October 8, 2020):

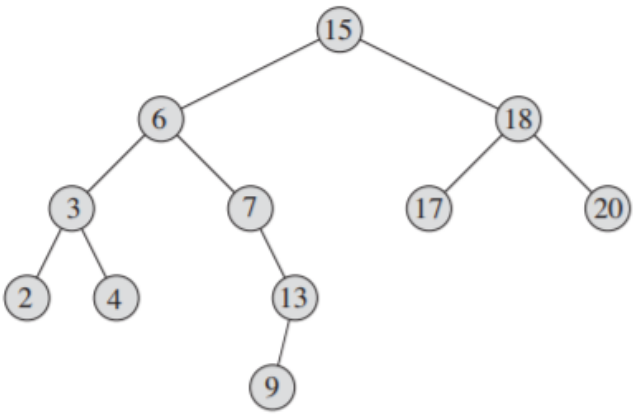

Figure 1: Binary Search Tree

(b) Test the functions that were written in Lab 4 using the BST shown in Figure 1 and verify that the correct answers are obtained.

(c) Write recursive code for the *treeSearch* function with following prototype:

```
BTNode * treeSearch (BTNode * root, int key);
```

(d) Write non-recursive code for the *treeMinimum* function with the following prototype:

```
BTNode * treeMinimum (BTNode * root);
```

(e) Write non-recursive code for the *treeMaximum* function with the following prototype:

```
BTNode * treeMaximum (BTNode * root);
```

Part 3: Binary Search Tree with Parent Pointers

Suppose we wish to include another field in each node of a BST which will contain the address of that node's parent.

(a) What changes would be necessary in the *BTNode* struct and *createBTNode* functions?

(b) Given any node, how do we find its parent? What is the parent of the root node?

(c) The *depth* of a node is the number of branches that must be traversed on the path to the node from the root. Write the code for the function *nodeDepth* with the following prototype which finds the depth of the node passed as a parameter:

```
int nodeDepth (BTNode * node);
```

(d) Suppose *treeSearch* (Part 2(c)) returns the address of a node which must be deleted if it is a leaf. Write a function with the following prototype to delete this node (but only if it is a leaf):

```
bool deleteLeafNode (BTNode * node); // returns true if deleted & false otherwise
```