Rohan Behera

11/9/20

UCR - CS 010C

Assignment 3

Deliverables: Create a single PDF file that contains your answers to the questions. Then create a zip file that contains this PDF file along with all your code source files. Submit this zip file on iLearn.

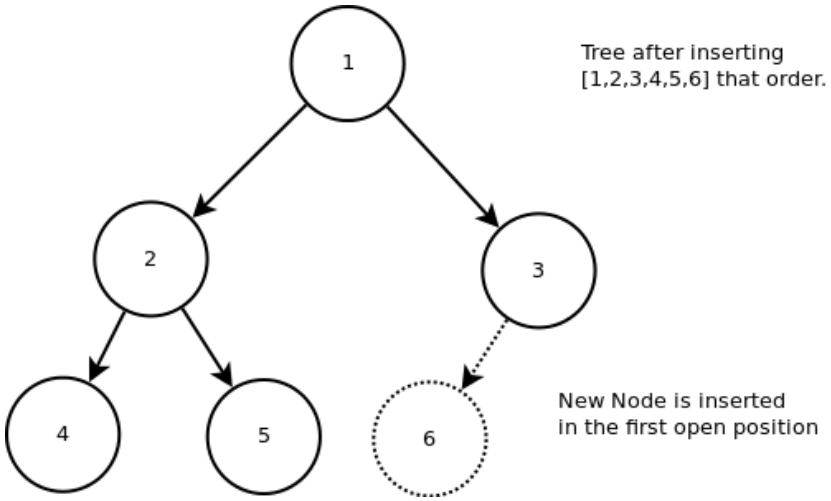
Deadline: 11/15/2020 11:59 pm.

Use provided C++ skeleton files (MyTree.cpp, MyTree.h and test.cpp) to insert your code.

1. Implement a binary tree class **MyTree** using pointers (define *struct* **BinaryNode** to store internal nodes), where each node has a pointer to each of its children and to its parent, and each node holds two variables, a string myString and an integer myInt.

   Write functions in **MyTree** class to do the following:

   a. **Insert(string, int)**: Insert new node into first available position (to keep the tree almost complete). E.g.,



   To get full points your Insert functions should be faster than $O(n)$, where n is the number of nodes in the tree. Hint: you may use additional private variable(s) in MyTree class.

```
BinaryNode *MyTree::Insert(int myInt, const string &myString) {
   return InsertRecursive(myInt, myString, root_);

}
```

```
BinaryNode *MyTree::InsertRecursive(int myInt, const string &myString, BinaryNode *currentNode) {
  // cout << " MyTree::InsertRecursive:: string : " << myString << " myInt: " << myInt << endl;
   if (currentNode == NULL) {
      cout << "current node NULL " << endl;
      root_ = newNode(myInt, myString);
      return root_;
   }
    //   cout << "MyTree::InsertRecursive::not null"<< endl;
      if (myInt < currentNode->myInt) {
         if (currentNode->left != NULL) {
            InsertRecursive(myInt, myString, currentNode->left);
         } else {
            currentNode->left = newNode(myInt, myString);
         }
      } else if (myInt > currentNode->myInt) {
         if (currentNode->right != NULL) {
            InsertRecursive(myInt, myString, currentNode->right);
         } else {
            currentNode->right = newNode(myInt, myString);
         }
      }
      return currentNode;
}
```

b. **Preorder()**: Output all strings in pre-order.

```
void MyTree::PreorderRecursive(BinaryNode* node) const {
   if (node != NULL) {
      cout << "String: " << node->myString << " Data: " << node->myInt << endl;
      PreorderRecursive(node->left);
      PreorderRecursive(node->right);
   }
}

void MyTree::Preorder() const {
   cout << "MyTree::Preorder " << endl;
   if (root_ == NULL){
      cout << "root_ null " << endl;
   }
   PreorderRecursive(root_);
}
```

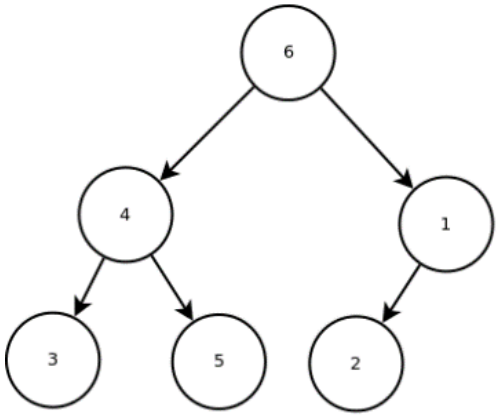
c. **FindMax()**: Return a pointer to the node with maximum myInt. (If multiple nodes have the same maximum myInt, return any node.)

```
BinaryNode *MyTree::FindMax() const {
  //  if (root_ != NULL) cout << " root not null" << endl;
  //  if (root_->left != NULL) cout << " root left not null" << endl;
  //  if (root_->right != NULL) cout << " root right not null" << endl;
    cout << "FindMax::max nodes = " << NumNodes() << endl;

    FindMaxHelper(root_);
}

BinaryNode *MyTree::FindMaxHelper(BinaryNode *node) const {
    if (node == NULL) {
        return NULL;
    }
    BinaryNode *currNode = node;
    BinaryNode *left = NULL;
    BinaryNode *right = NULL;
    left = FindMaxHelper(currNode->left);
    //cout << left->myInt << endl;
    right = FindMaxHelper(currNode->right);
    //cout << right->myInt << endl;
    if (left != NULL && right != NULL) {
        if (currNode->myInt >= left->myInt && currNode->myInt >= right->myInt) {
            cout << " FindMaxHelper::data = " << currNode->myInt << endl;
            return currNode;
        }
        else if (left->myInt >= currNode->myInt && left->myInt >= right->myInt) {
            cout << " FindMaxHelper::data = " << left->myInt << endl;
            return left;
        }
        else return right;
    }
    if (left != NULL && right == NULL) {
        if (currNode->myInt >= left->myInt) return currNode;
        else return left;
    }
    if (left == NULL && right != NULL) {
        if (currNode->myInt >= right->myInt) return currNode;
        else return right;
    }
    return currNode;
```

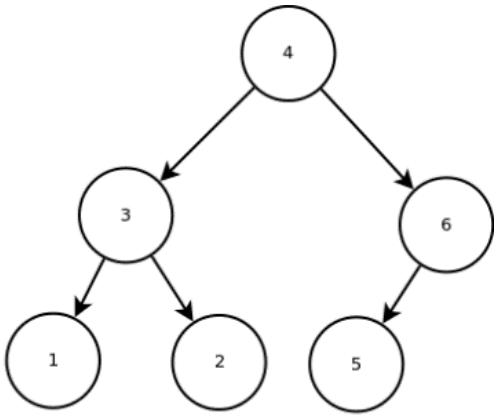
}

d. **MakeBST()**: Convert the binary tree into a binary search tree (BST) with respect to myInt. That is, move around node values (myString and myInt) to satisfy the BST property. Do not change the structure of the tree (i.e. the pointers) but only swap myString and myInt values. (Hint: if you need to sort an array, you can use std::sort method; use std::swap to exchange the values of 2 variables)

e.g.,



Original Tree after inserting [6,4,1,3,5,2]



BST: Notice we keep the ORIGINAL structure

```
void MyTree::MakeBST() {
    MakeBSTHelper(root_, numNodes_);
}
void MyTree::MakeBSTHelper(BinaryNode* node, size_t node_size) {
  if (node == NULL) {
    return;
  }
  //Copy values from root to a vector of pairs
  vector<pair<int, string>> *items = new vector<pair<int, string>>[node_size];
  items = copyToVector(node, items);
  cout << "MakeBSTHelper::vector size = " << items->size() << endl << endl;
  sort(items->begin(), items->end());
```

```
    // build the BST tree

    BinaryNode *bstNode = sortedArrayToBST(items);

    delete bstNode;

    delete [] items;

}
```

2. What is the big-Oh complexity of your functions above? Also, what is the space complexity of your functions? Are they all in-place? If not, how much extra space do they need?
   The big-Oh complexity for Insert(string, int) is O(1) because I am using a recursive helper function to traverse the nodes and the space complexity would be O(1). The big-Oh complexity for findMax would be O(n) and the space complexity would be O(1). The big-Oh complexity for MakeBST() is O(nlogn) and the space complexity is O(n). The big-Oh complexity of Preorder() is O(n) and the space complexity would be O(1). Insert, findMax, and Preorder are in place but MakeBST is not in place.

3. Test and measure the performance of your functions.
   Create sequences of 100, 1000, 10000, 100000 random (string, int) pairs and insert them into MyTree (using Insert() function). Measure the times (in **nanoseconds** or **microseconds**) to
   a) build the tree,
   b) execute Preorder(),
   c) execute FindMax(),
   d) execute MakeBST().

   Report the times for each tree size in a table.

   Hint: To generate *N* random unique numbers, you can first create an array or a vector with *N* unique numbers (e.g., 1 to *N*), then use std::shuffle to rearrange them in a random order.

Number of Pairs	Build Tree Time	Preorder() Time	FindMax() time	MakeBST() time
100	36602 ns	580410 ns	9544 ns	63780 ns
1000	408230 ns	13627754 ns	49430 ns	1352174 ns
10000	22463150 ns	117132974 ns	353064 ns	426601053 ns
100000	378630682 ns	3488812329 ns	10687164 ns	800853769 ns