Assignement 4
MyTree and BST

**Part of the source code is included here, but all of it is in the zip folder*

Exercise 1

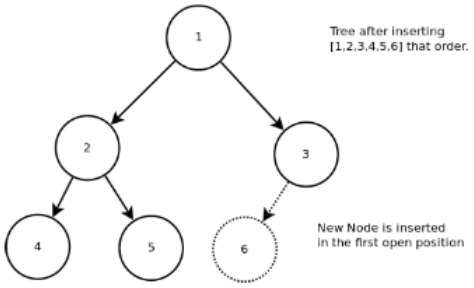
Use provided C++ skeleton files to insert your code.

A.

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

Write functions in MyTree class to do the following:

a. Insert (int,string): 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 # nodes in tree. Hint: you may use an additional private variable in MyTree class.

b. Preorder (): Output all strings in pre-order.
c. FindMax (): returns a pointer to the node with maximum myInt.
d. MakeBST (): converts 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 STL's sort() method)

B.

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?

Here is the code for the Insert function:

```
void MyTree::insert(int x, string s) {
    BinaryNode* to_insert = new BinaryNode(s, x);
    if(!this->root){
        this->root = to_insert;
        this->nextInsertRoot = root;
    }
    else{
        if(!this->nextInsertRoot->lchild){
            this->nextInsertRoot->lchild = to_insert;
            to_insert->parent = this->nextInsertRoot;
        }
        else if(!this->nextInsertRoot->rchild){
            this->nextInsertRoot->rchild = to_insert;
            to_insert->parent = this->nextInsertRoot;
            // if parent is a left child, next location is parent's right chi
ld
            if(this->nextInsertRoot->parent && this->nextInsertRoot->parent-
>lchild == this->nextInsertRoot){
                this->nextInsertRoot = this->nextInsertRoot->parent->rchild;
            }
            // if parent is right child, two scenarios
            else{
                // here we traverse parents until the first left parent is fo
und (if can be found)
                BinaryNode* temp = this->nextInsertRoot;
                while(temp->parent && temp->parent->lchild != temp)
                    temp = temp->parent;
                // if we're not at root, we find the first parent right child
                if(temp->parent)
                    temp = temp->parent->rchild;
                // traverse to deepest left child
                while(temp->lchild)
                   temp = temp->lchild;
                this->nextInsertRoot = temp;
            }
        }
    }
}
```

The first thing to note before the analysis, is that a parent pointer has been added to the BinaryNode class, and that a nextInsertRoot has been added to the MyTree class.

The nextLocationRoot is updated in each call to the Insert function and will help to determine the next node's location that we will want to be added to the binary tree. Thus, we will not traverse our whole tree in $O(n)$ every time we want to add a new node.

We follow by analyzing all the branches of our Insert function. The first if determines if the node we want to insert is going to be our actual root of the tree, it's time is thus constant $O(1)$. If the root already exists, we have two cases. If the nextInsertRoot doesn't have a left child, we insert the new node as its left child (insertion goes from left to right). We don't change the nextInsertRoot pointer as the next node's location will be its right child. This operation is also done in constant $O(1)$ time. If our nextInsertRoot doesn't have a right child, we insert it as a right child, and then determine the next location of the nextInsertRoot pointer. We have two cases, if our nextInsertRoot has a parent (is not the root) and is a left child, then the nextInsertRoot will always be our parent's right child. This operation is also done in constant $O(1)$ time. Otherwise, we traverse the tree up until either we find the root, or we find a parent which is a left child. That traverse operation could take $O(log(n))$ in the worst case, if we just inserted the bottom right node, with n being the number of total nodes in the tree. If we went back to the root, the nextInsert location must be the deepest left child, so we traverse to this child in the same $O(log(n))$ time and reset the nextInsertRoot pointer. If we found a parent that is a left child, we find the first parent's right child, and go to the deepest left node. This case is handled in constant $O(1)$ time. Overall, the worst case remains the next Insertion after the deepest right child, handled in $2 * O(log(n)) + k * O(1)$ for changing pointers, which yields a $O(log(n))$ time complexity for the Insert function. The space complexity is $O(1)$ for creating a new Node to insert, and a temp pointer to traverse the tree, which is $O(1)$, so the overall space complexity is constant. The algorithm is running in place since we only create a copy for traversing the tree, and we are not duplicating it.

Here is the code for the preorder function:

```
void MyTree::preorder() const {
    this->preorderHelper(this->root);
}

void MyTree::preorderHelper(BinaryNode* node) const {
    if(node){
        cout << node->getString() << " ";
        preorderHelper(node->lchild);
        preorderHelper(node->rchild);
    }
}
```

Since we go through all the n elements recursively to print them, the time complexity is $O(n)$. The space complexity is only determined by the stack frames issued by the recursive calls. In a perfect balanced tree, the maximum stack frames that could be stacked correspond to the height of the tree, thus $O(log(n))$. In the worst case however, if the tree is hardly unbalanced (only right or left childs), the space complexity the height of the tree, now being $O(n)$. Given the Insert function, this case is only encountered when we have a root and left child, we traverse the n elements, but since n=2, this case can be considered done in $O(1)$. The algorithm runs in place as we are not using an auxiliary data structure to process the tree.

Here is the code for the findMax function:

```
BinaryNode* MyTree::findMax() const {
    return findMaxHelper(this->root);
}

BinaryNode* MyTree::findMaxHelper(BinaryNode* node) const {
    if(node){
        int current = node->myInt;
        BinaryNode* leftMax = findMaxHelper(node->lchild);
        BinaryNode* rightMax = findMaxHelper(node->rchild);
        int lMax = INT16_MIN, rMax = INT16_MIN;
        if (leftMax)
            lMax = leftMax->getInt();
        if (rightMax)
            rMax = rightMax->getInt();
        if (lMax > rMax && lMax > current)
            return leftMax;
        else if(rMax >= lMax && rMax > current)
            return rightMax;
        return node;
    }
    return nullptr;
}
```

For the time complexity it's the same case as the preorder function, we must traverse all the elements to find the maximum, and we can't do better that $O(n)$. The recursive calls will return the max of the left and right subtree and will go through the n elements. For the space complexity, it's affected by the two recursive calls to findMaxHelper. If the tree is perfectly balanced, the stack frames will account for $O(log(n))$ of the tree (its height), but if it's not, the complexity falls in the same category as the one of preorder, $O(n) \sim O(1)$ for the root and its left child. This algorithm runs also in place for the same reasons.

Here is the code for the makeBST function:

```
void MyTree::makeBST() {
    // since inorder traversal of BST is a sorted array, we inorder our BT, s
ort its nodes and change their values inorder
    // store tree inorder
    vector<tuple<int, string>> tree;
    this->storeInorder(this->root, tree);
    // sort in ascending order
    sort(tree.begin(), tree.end());
    // store it back
    int start_index = 0;
    this->resetValuesInorder(tree, &start_index, this->root);
```

```
void MyTree::storeInorder(BinaryNode* node, vector<tuple<int, string>> &curr
{
    if (node){
        this->storeInorder(node->lchild, curr);
        curr.push_back(make_tuple(node->myInt, node->myString));
        this->storeInorder(node->rchild, curr);
    }
}

void MyTree::resetValuesInorder(vector<tuple<int, string>> &tree, int* currIn
dex, BinaryNode* node){
    if (node){
        this->resetValuesInorder(tree, currIndex, node->lchild);
        node->myInt = get<0>(tree.at(*currIndex));
        node->myString = get<1>(tree.at(*currIndex));
        *currIndex = *currIndex + 1;
        this->resetValuesInorder(tree, currIndex, node->rchild);
    }
}
```

Making a BST from a Binary tree takes three steps. First, store in an ordered data structure the nodes of our tree, then sort them from smallest to largest and finally, since the inorder traversal of a BST is a sorted array, we traverse our tree inorder, and replace its string and int with the values contained in our data structure. We start by creating a vector of tuples (int, string). We store all of the nodes of our tree in it in $O(n)$ time and $O(n)$ space, since we allocated a vector of size n, and the rest is the same case as the preorder function, assuming that the push_back function takes constant time (actually amortized $O(1)$). Then, we sort our vector in $O(n * log(n))$ time, with the built-in sort function. Finally, we traverse our tree inorder, and place the values (int and string) contained in our vector into the tree, without modifying its structure. The get() and at() function are also in constant time. Thus, this part is $O(n)$ in time and $O(1)$ in space. Our final algorithm is thus bounded in time by the sort function: $O(n * log(n))$ time, and by the size of the vector, $O(n)$ in space. It's not in place, we created a vector to hold our int and string values. It would had been worst if we would have stored all of the nodes, since the root contains the whole tree, its left and right children contain the right and left root's subtree and so on.

C.

Test and measure the performance of your functions.

Create sequences of 100, 1000, 10000, 100000 random (int, string) pairs and insert them into MyTree (using Insert(.,.) function). Measure the times to (a) build the tree, (b) execute Preorder(), (c) execute FindMax(), (d) execute MakeBST().

Report the times for each tree size in a table.

Here is the table for the tests (values in seconds):

N	Build()	Preorder()	findMax()	makeBST()
100	0	0.034001	0	0
1000	0.005074	0.36103	0	0.003001
10000	0.053993	3.77018	0.000228	0.042993
100000	0.485125	38.2786	0.002002	0.305034

Few observations:

As expected the preorder and findMax() functions operate in $O(n)$ time, but the preorder function takes a lot of time to print to the console, that's why its relative time taken is relatively larger than the one of findMax(). If the preorder wouldn't print the value, its time taken to execute would have been similar to the findMax function. The makeBST function seems to be in $O(n)$ as well, and we expected $O(n * log(n))$. One of the explanations possible is that the time complexity for the sort function from STL is close to $O(n)$ for our input sizes, and thus our function approaches $O(n)$. The build function is in $O(n)$, and it comes from the n values we are inserting in our tree. Since the insert function is in the build function, we can deduct that the insert function is almost always made in constant time (as we expected), since the overall complexity of build() is $O(n)$.