Talanda Williams

Final Project

```
public:
    /***** Function Members *****/
    BST();
    bool empty() const;
    void insert(const int & item);
    void inOrder();
    void inOrder(BinNode * toCheck);
    BinNode* balanceTree(int arr[], int first, int last);
    void LeafHeights();
    void LeafHeights(BinNode * current, int height);
    void selectionSort(int list[], int size);
    int NewSearch(const int & item, BinNode * node, int greater) const;
    int NewSearch(const int & item) const;

}; // end of class declaration
```

Step 1:

The header file was modified to include the new functions needed and their types, including inOrder and it's overloaded function, balanceTree, LeafHeights and another overloaded function, a pull of selectionSort from a previous assignment, and NewSearch.

```
BST::BinNode* BST::balanceTree(int arr[], int first, int last){
    if (first > last) {
        return NULL;
    }
    int mid = (first + last)/2;
    BinNode * next = new BinNode(arr[mid]);
    insert(next->data);
    next->left = balanceTree(arr, first, mid - 1);
    next->right = balanceTree(arr, mid + 1, last);
}
```

Step 2: Balancing the Tree

The driver file gets the array, sorts it with selection sort (from the prior assignment), and then gives the sorted array to this balanceTree function. This will process the array essentially from the middle element outward, calling the insert function to build the actual tree. This allows the use of the given function while also ensuring that the tree is built balanced, rather than needing to be self balancing as items are added. The algorithm is essentially find middle element, add it as a node, return subset (smaller or larger) to the function, continue to add the left and right children until all items have been added of each 'sub'array created. Time complexity is O(n*d) as it relates directly to how many items are given in the array (n) and there is no nested loops but it is repeated m times based on the depth of the tree (in this case).

```
void BST::selectionSort(int list[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minNum = i; //minNum starts as first pos
        for (int j = i + 1; j < size; j++) {
            if (list[j] < list[minNum]) {
                minNum = j; //minNum becomes positio
            }
        }

        if (list[i] != list[minNum]) {
            int temp = list[i]; //save current list
            list[i] = list[minNum]; // swap value o
            list[minNum] = temp; //put value previou
        }
    }
}
```

Step 2.5:

This is the added selectionSort algorithm from a prior assignment we had. This just is used for the driver function to get the array in whatever order, and sort it. This adds $O(n^2)$ complexity due to the nested for loops, but is not necessarily a main part of this program- the sorting algorithm can be changed to a more efficient heapsort algorithm for instance.

```
void BST::inOrder() {
    inOrder(myRoot);
}

void BST::inOrder(BinNode * toCheck) {
    if (toCheck == NULL) {
        return;
    }
    inOrder(toCheck->left);
    cout << toCheck->data << ", ";
    inOrder(toCheck->right);
}
```

Step 3:

inOrder function is the same as from the prior assignment. It has a no parameter function to call from the main, and an overloaded function with a BinNode element to process the actual ordering and recursive call.

Time complexity requires all nodes to be traversed so is at least O(n) but adding the number of times recursively called is O(n*d)

```
void BST::LeafHeights() {
    LeafHeights(myRoot, 0);
}

void BST::LeafHeights(BinNode * current, int height) {
    if (current == NULL) {
        return;
    }

    if (current->left == NULL && current->right == NULL) {
        cout << "\nHeight of leaf '" << current->data << "' is: " << height;
    }

    int currentHeight = height + 1;
    LeafHeights(current->left, currentHeight);
    LeafHeights(current->right, currentHeight);

}
```

Step 4:

LeafHeights needs to get the current node, verified it's not null itself, and then find out if the node has left or right children. This determines a leaf, which it then prints out the value and height of that leaf. It's a recursive function which has to traverse the full tree to verify it has found all the notes, making it also O(n*d) worst in time case, even though it's just linearly going through all items.

```
int BST::NewSearch(const int & item) const
{
    BinNode * locptr = myRoot;
    return NewSearch(item, myRoot, 0);
}

int BST::NewSearch(const int & item, BinNode * node, int greater) const {
    if (node == NULL) {
        if (greater > item) {
            return greater;
        }
        else {
            return -999; //cannot be searched for
        }
    }
    else if (item == node->data) {
        return node->data;
    }
    else if (node->data > item) {
        return(NewSearch(item, node->left, node->data));
    }
    else { //node is smaller, do not want to lose the last smallest but bigger number
        return(NewSearch(item, node->right, greater));
    }
}
```

Step 5:

NewSearch requires traversing through the nodes similarly to preorder, checking the data and going left or right. In this case, it searches for the item while also holding the data for the last node checked. In this manner, I can compare what value to keep based on which direction the search is going. If the item is moving to the left child, then the item being searched for is smaller than the current node,

and that value needs to be saved so that it can be used as the smallest, greater number if the item itself is not found. If the search needs to move to the right child, then the item is larger than that last found number and the previous number saved should be retained as to not hold a number lower than the one being searched for.

```
BST intBST;

int numbers[] = { 2, 32, 100, 67, 55, 34, 1, 3, 19, 6, 7, 47, 9, 10, 39, 15, 5, 23 };
int size = sizeof(numbers) / sizeof(numbers[0]);

cout << "Numbers to Insert into Tree: " << endl;
for (int i = 0; i < size; i++) {
    cout << numbers[i] << ", ";
}

cout << "\n\nSelection Sorting Array... " << endl;
intBST.selectionSort(numbers, size);
for (int i = 0; i < size; i++) {
    cout << numbers[i] << ", ";
}

cout << "\n\nInserting and Balancing Tree..." << endl;
intBST.balanceTree(numbers, 0, size-1);

//Testing inOrder
cout << "\nRunning inOrder: " << endl;
intBST.inOrder();
cout << endl;

//Testing LeafHeights
cout << "\nRunning LeafHeights: ";
intBST.LeafHeights();
cout << endl;
```

Step 6 - Driver file:

The process of running these 4 parts includes creating the array, displaying the items in the array (just for programmer/user ease of explanation), and then running the selection sort array. Any sorting algorithm to sort the array in the driver is acceptable.

The balanceTree function is what inserts and balances the tree itself, with help from the insert function, covers part 1.

inOrder and LeafHeights runs next to cover parts 2 and 3.

```
    //Testing NewSearch
    cout << "\nNumber to Search For (equal or smallest greater than): ";
    int search = 0;

    do{
        cout << "\nItem to find (-999 to stop): ";
        cin >> search;
        int temp = intBST.NewSearch(search);
        if (temp == -999) { //because our search driver depends on -999 never being searched for
            cout << "No match found.\n";
        }
        else {
            cout << "Closest Match is " << temp << endl;
        }
    } while (search != -999);

}
```

continued…

NewSearch, lastly, covers part 4 and uses the search style of the prior assignment, except changing the empty for loop into a do-while loop. This keeps the assumption that -999 will not be searched for in the array, as given prior. I use -999 to return in the code if a value is not found since we know it cannot be included. Although searching for it, prior to the code stopping, will actually still give you the closest number per the design. I needed temp to act as a bool in the false case while returning the number in the true case so this is what the design ended up as.

End Result on next page-

Sample end results:

```
Numbers to Insert into Tree:
2, 32, 100, 67, 55, 34, 1, 3, 19, 6, 7, 47, 9, 10, 39, 15, 5, 23,

Selection Sorting Array...
1, 2, 3, 5, 6, 7, 9, 10, 15, 19, 23, 32, 34, 39, 47, 55, 67, 100,

Inserting and Balancing Tree...

Running inOrder:
1, 2, 3, 5, 6, 7, 9, 10, 15, 19, 23, 32, 34, 39, 47, 55, 67, 100,

Running LeafHeights:
Height of leaf '1' is: 3
Height of leaf '3' is: 3
Height of leaf '6' is: 3
Height of leaf '10' is: 4
Height of leaf '19' is: 3
Height of leaf '34' is: 4
Height of leaf '47' is: 3
Height of leaf '100' is: 4

Number to Search For (equal or smallest greater than):
Item to find (-999 to stop): 100
Closest Match is 100

Item to find (-999 to stop): 25
Closest Match is 32

Item to find (-999 to stop): -5
Closest Match is 1

Item to find (-999 to stop): 56
Closest Match is 67

Item to find (-999 to stop): -999
Closest Match is 1
Press any key to continue . . .
```

```
Numbers to Insert into Tree:
1, 2, 3, 4,

Selection Sorting Array...
1, 2, 3, 4,

Inserting and Balancing Tree...

Running inOrder:
1, 2, 3, 4,

Running LeafHeights:
Height of leaf '1' is: 1
Height of leaf '4' is: 2

Number to Search For (equal or smallest greater than):
Item to find (-999 to stop): 5
No match found.
```

This assignment was made in Visual Studio 2017, I have removed the #include "stdafx.h" lines but they would need to be re-added if testing in this IDE.