CSCE 221 Cover Page
Programming Assignment # 4

FirstName: Eric LastName: Gonzalez

UIN: 322002089

User Name: egoftcp

E-mail address: egoftcp@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more in the Aggie Honor System Office http://aggiehonor.tamu.edu/

Type of sources	
People	
Web pages (provide URL)	stackoverflow.com
Printed material	
Other Sources	CSCE 221 slides

I certify that I have listed all the sources that I used to develop the solutions/code to the submitted work.
"*On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.*"

Your Name (signature) Eric Gonzalez Date 04/07/2015

Program Description

This program is designed to create a binary search tree, calculate the average search cost of each node in the tree, and then remove a designated node.

Purpose of the Assignment

The purpose of this assignment was to teach us how make use of binary search trees.

Data Structures Description

In this assignment, we learned to make use of the binary tree data structure in C++. For our implementation of the binary tree, we were tasked with developing functions for searching within the tree, and thus implementing a binary search tree. Each binary node has two pointers that point to the left and right, as well as a Key (represented by element) and a SearchCost. The BinarySearchTree class is a friend class of the BinaryNode class. BinaryNode has all the functions that are called in main.cpp.

Algorithm Description

1. insert(int x, BinaryNode *t, int& search_cost) The insert function compares the input with the current node and if greater than the current node's value, then the current node is moved to the left. If it is smaller than the current node, then the current node is moved to the right, instead. When the current node is empty (null), the function will insert the new node. The maximum number of comparisons for the insert function is the height of the tree, so the time complexity is O(h).

2. remove(int x, BinaryNode *t) The remove searches for a node x based on comparisons and when the designated node is found, it substitutes the value of the node with the minimum value of the right sub-tree. If the node has only one sub-tree, then it will delete the node and move the sub-tree up. Otherwise, it will just delete the single node. After deleting a node, the search cost for every node in the tree is updated. The time complexity for the deleting is O(h), for the updating of the search cost it's O(n). Total time complexity is O(n+h).

3. resetCost(BinaryNode* t, int i) This function updates the search cost of an individual node. Through its implementation, the function resets the search cost for all nodes in the tree. The time complexity is O(n).

4. preOrderTraversal(BinaryNode *t, int& total_sch_cost) This function will calculate and return the summing up of the search cost. The pre/post/in order traversal functions are all recursive functions and will go through every node in the tree. The time complexity is O(n).

5. The individual search cost of an element in a binary tree is O(n) for the worst case (a linear tree) and O(logn) for the best case (a perfect tree). For the worst case, the height of the tree is n-1 and the total search cost is (n(n+1)/2)/n, which equals O(n).

For the best case, the height of the tree is log(n). For each level k, there are 2^k nodes. The total search time is (log1+1)+2(log2+1)+ 2^2(log3+1) +2^3log4 +.....2^log(n+1)-1(logn+1)= (n+1)log(n+1)-n. So, the average search time is O(log(n)).

Program Organization and Description of Classes

```
class BinaryNode {

private:

    friend class BinarySearchTree;

    int element;

    int SearchCost;

    BinaryNode *left, *right;
```

```
public:
   BinaryNode(int el = 0, int search_cost=0, BinaryNode *lt = NULL, BinaryNode *rt = NULL)
://constructor
                element(el),SearchCost(search_cost), left(lt), right(rt){}// functions
   BinaryNode *getLeft() {return left; }
   BinaryNode *getRight() {return right; }
};
class BinarySearchTree {
private:
   int node_num; // total num in the tree
   BinaryNode *root;
   BinaryNode *insert(int x, BinaryNode *t, int& search_cost);//insert x into tree t
   BinaryNode *findMin(BinaryNode *t);
   BinaryNode *removeMin (BinaryNode *t);
   BinaryNode *remove(int x, BinaryNode *t);

   void inOrderTraversal( BinaryNode *t, int& total_sch_cost);
   void preOrderTraversal( BinaryNode *t, int& total_sch_cost);
   void postOrderTraversal( BinaryNode *t, int& total_sch_cost);
        void resetCost(BinaryNode* t, int i);
public:
   // constructor
   BinarySearchTree() { root = NULL; node_num=0; }
   bool isEmpty(){return root == NULL;}
   void remove(int x){ root = remove(x, root); }
```

```
void insert(int x) {
    int search_cost=0;
    root=insert(x, root, search_cost);
}
int inOrderTraversal(){
    int total_sch_cost = 0;
    inOrderTraversal(root, total_sch_cost);
    cout<<endl;
    return total_sch_cost;
}
int preOrderTraversal(){
    int total_sch_cost = 0;
    preOrderTraversal(root, total_sch_cost);
    cout<<endl;
    return total_sch_cost;
}
int postOrderTraversal(){
    int total_sch_cost = 0;
    postOrderTraversal(root, total_sch_cost);
    cout<<endl;
    return total_sch_cost;
}
int getNodeNum(){return node_num;}

void OutputTreeLevelByLevel();
```

```
    void OutputText(string filename);
};
```

Instructions to Compile and Run

Make

./Main

Input file name to run (test_example)

Enter node number to remove

Enter any key to output remaining files

Input and Output Specs

Input consists of a string for the file name when running the example test file, an integer for the value one wishes to remove from the tree, and the values passed by the test files.

Output consists of the input data, the Keys and Search Costs of the binary tree, the three traversal for the tree, the total number of nodes, the average search cost, the tree displayed as a level-by-level print out, and updated information after a node has been removed. The program outputs to both the console and separate result files for 1-4p, r, and l, as well as output_averages and example_output.

Logical Exceptions

There should be no logical errors.

C++ Object Oriented or Generic Programming Features

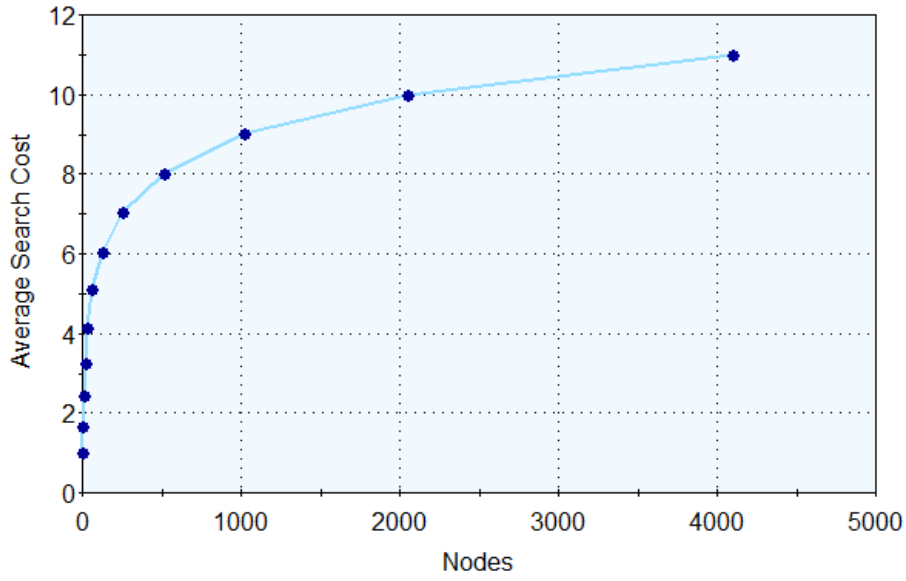
-Auto

Tests (Graphs)

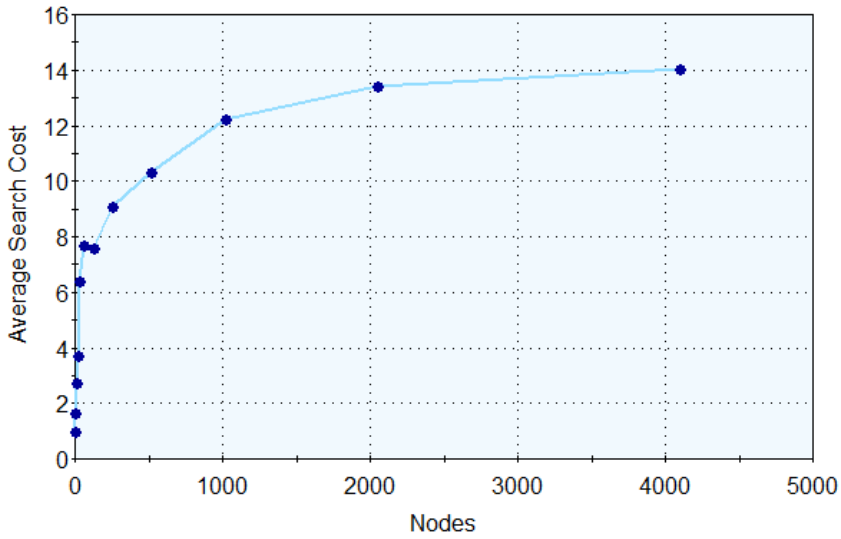
Nodes	Perfect	Random	Linear
1	**1**	**1**	**1**
3	**1.66667**	**1.66667**	**2**
7	**2.42857**	**2.71429**	**4**
15	**3.26667**	**3.73333**	**8**
31	**4.16129**	**6.3871**	**16**
63	**5.09524**	**7.66667**	**32**
127	**6.05512**	**7.59055**	**64**
255	**7.03137**	**9.06667**	**128**

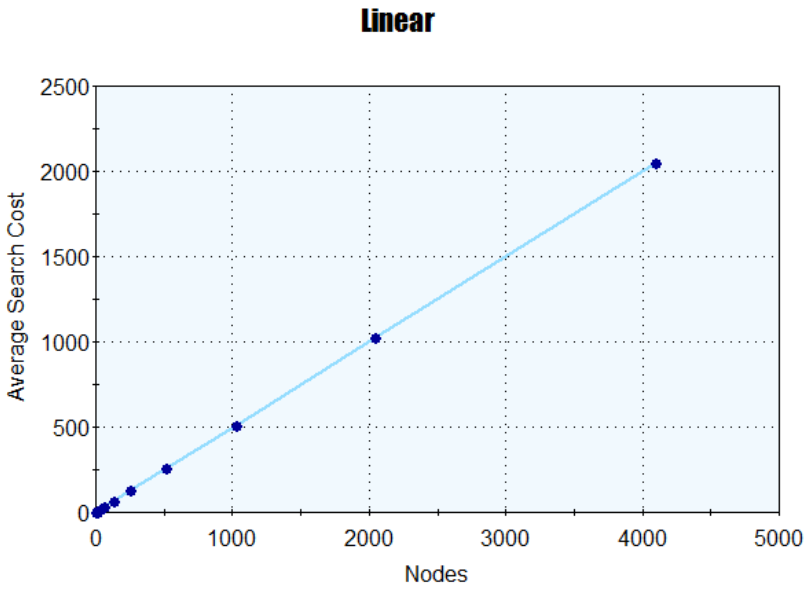
511	**8.01761**	**10.3033**	**256**
1023	**9.00978**	**12.2463**	**512**
2047	**10.0054**	**13.3972**	**1024**
4095	**11.0029**	**14.0237**	**2048**

Perfect



Random




Linear
Average Search Cost
Nodes
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000