CSCE 221 Cover Page
Assignment #4
Due November 1st at midnight to CSNet

First Name Raymond Last Name Zhu UIN 923008555

User Name rawrbyte E-mail address rawrbyte@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. According to the University Regulations, Section 42, scholastic dishonesty are including: acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion read more: Aggie Honor System Office

Type of sources			
People	Peer Teachers		
Web pages (provide URL)	Stackoverflow		
Printed material	textbook		
Other Sources	CSCE221 Slides		

I certify that I have listed all the sources that I used to develop the solutions/codes 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 Raymond Zhu Date November 1st, 2015

1. Program Description

   (a) For this lab, a binary search tree or BST was created using implementations in C++ programming to calculate the average search cost of each node in the binary tree, and remove nodes from the tree.

2. Purpose of the Assignment

   (a) The purpose of this programming assignment is to learn how to use our previous knowledge over nodes and pointers from the douby linked list assingment and create structured binary trees, how to search the trees for values based on an algorithm, and the efficiency of a BTS on how it is structured.

3. Data Structures Description

   (a) Binary Search Trees were implemented in this assignment. During the implementation of this abstract data type, the most important thing that stood out was that every node has a key and a search cost. Simlarly to a doubly linked list, instead of a head and a tail, a binary search tree has nodes pointing to the left and right. One for each childe essentially. For the BinarySearchTree class, it is a friend class of the BinaryNode, which inherits the functions that are called in the main function.

4. Alogorithm Description

   - insert(int x, BinaryNode *t, int& sc){} $O(h)$: this function compares a input value and values of a node to determine where to insert a new node. If the input value is greater than the current node, the insert function will branch out to the right. Whereas if the input value is smaller than the node, the insert function will branch out towards the left. When there is no node, or when the node reaches and points to a value of NULL, that will be where the new node is inserted in the binary search tree. The maximum number of comparisons comes from the height of the tree and whether the tree is a balanced or unbalanced tree.
   - remove(int x, BinaryNode *t) $O(n + h)$: Similarly to the insert function described above, the remove function will search for the designated node based comparing values and decending the height going either left or right of each parent. Once the node is found, to delete the node, the remove function substitues the value of the node with the minimum value of the left sub-tree. If the node has a smaller child, the function would remove the current node and move the child up. Otherwise, the current node will be removed if no child is present. Once a node is removed from the binary search tree, the search cost of every node is updated and so given the time-complexity of h, the update adds an additional runtime of n.

- resetSearchcost(BinaryNode* t, int i) $O(n)$: As stated in the remove function of updating the search cost of every node when a node is deleted/removed from the tree, this function will update the search cost of an individual node. The reset search cost function will reset search cost for every node in the tree. The given time-complexity of the function will be linear.
- preOrderTraversal(BinaryNode *t, int& sc_total) $O(n)$: This function returns the sum of the search cost. Because pre/post/in order traversal functions are recursive functions, they run on a linear time-complexity.
- Considering the formulas

$$\sum_{d=0}^{log_2(n+1)-1} 2^d(d+1) \approx (n+1) * log_2(n+1) - n \text{ and} \sum_{d=1}^{n} \approx n(n+1)/2$$

- The individual search cost of an element in a binary tree is $O(n)$ given that is the worst case where the tree is structured linearly. On the other hand, when the binary tree has a balanced structure (the left side of the tree is the same as the right side). This balanced tree runs on a time-complexity of $O(log(n))$. For the worst case, similar to an unbalanced tree and being the worst case we consider a linear tree, the height of the tree is n-1. Like a linked list, the search cost is n(n+1)/2, divide that by n, the overall big-O is O(n). For the best case, considering a balanced tree as described above, because the height of the tree is $log(n)$, each level of the binary tree the sum of nodes is $2^k + 1$. The toatal search time is the summation of the levels, $log(1) + 1 + 2(log(2) + 1) + 4(log(3) + 1) + 8(log(4) + 1) + ... + 2^log(n + 1) - 1(log(n) + 1) = (n + 1)(log(n + 1) - n$. Where dividing by n, the average search time is $O(log(n))$.

5. Program Organization and Description of Classes

```
class BinaryNode {
private:
    friend class BinarySearchTree;
    int Key;
    int SearchCost;
    BinaryNode *left, *right;
public:
    BinaryNode(int key = 0, int sc=0,BinaryNode *lt = NULL,
        BinaryNode *rt = NULL) ://constructor
    Key(key),SearchCost(sc), 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& sc);//insert
         x into tree t
    BinaryNode *findMin(BinaryNode *t);
    BinaryNode *removeMin (BinaryNode *t);
    BinaryNode *remove(int x, BinaryNode *t);

    void resetSearchcost(BinaryNode* t, int i);
    void inOrderTraversal( BinaryNode *t, int& sc_total);
    void preOrderTraversal( BinaryNode *t, int& sc_total);
    void postOrderTraversal( BinaryNode *t, int& sc_total);
public:
    // constructor
    BinarySearchTree() { root = NULL; node_num=0; }
    bool isEmpty(){return root == NULL;}
    void remove(int x){ root = remove(x, root); } // remove
        Key = x
    void insert(int x) {
        int sc=0;
        root=insert(x, root, sc);
    }
    int inOrderTraversal(){
        int sc_total = 0;
        inOrderTraversal(root,sc_total);
        cout<<endl;
        return sc_total;
    }
    int preOrderTraversal(){
        int sc_total = 0;
        preOrderTraversal(root,sc_total);
        cout<<endl;
        return sc_total;
    }
    int postOrderTraversal(){
        int sc_total = 0;
        postOrderTraversal(root,sc_total);
        cout<<endl;
        return sc_total;
    }
    int getNodeNum(){return node_num;}

    void OutputTreeLevelByLevel();
    void Outputxt(string filename);
};
```

6. Instructions to Compile and Run your Program

   - First part
     - make
     - ./Main
     - The console will prompt the user to input a filename, the assignment comes with a file “example”, when asked to input a filename, type in “example”.
     - The console will prompt the user to delete a node, to delete any of the nodes, input the number of the node to delete

  - Each time a new tree will be printed
- Second part
  - To run the second part of the code, please go into Main.cpp and remove the comment parameters for the body of code.
  - Type “make” in console to recompile the code
  - ./Main
  - A file, AverageSC, will be created. Please check the contents for the correctness.

7. Input and Output Specifications
   - When deleting a node in the binary tree, input the key of the node.
   - When asked to input a file name, please use “example”

8. Logical Exceptions

   (a) No logical error has been found in testing from the program itself

   (b) During the process of implementing the code, logical errors were discovered that had to be fixed in order for the code to compile and run properly. For example, the file output of the binary tree with proper white spacing and new lines.

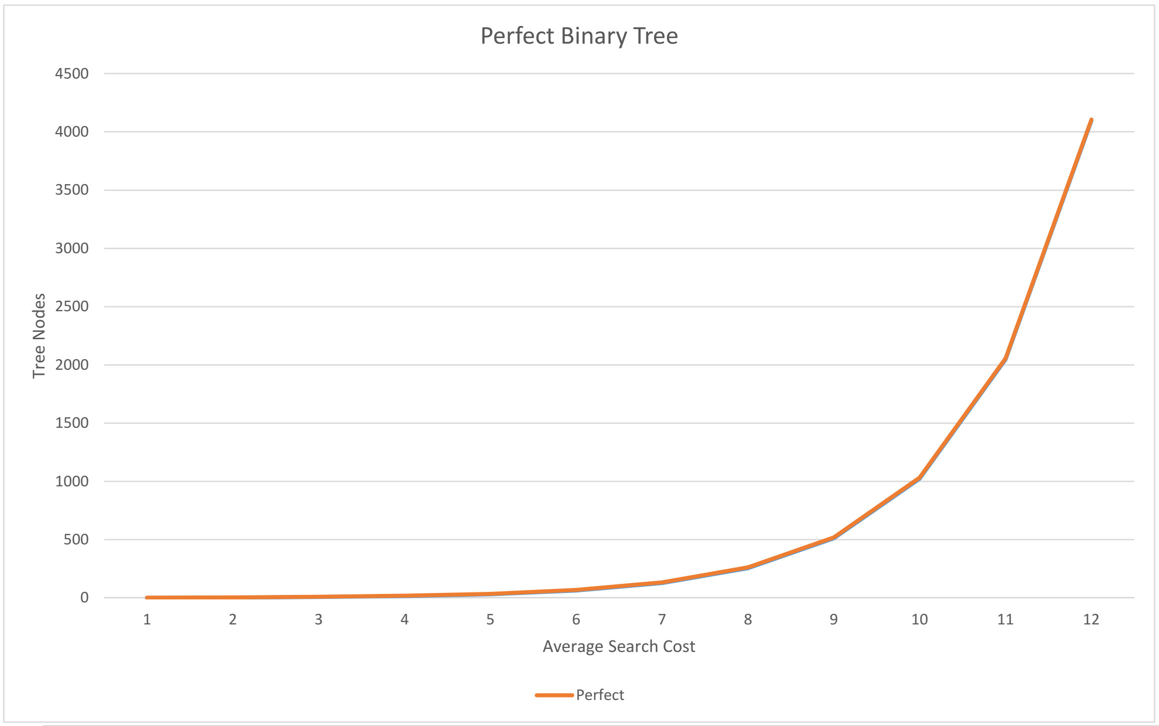
9. C++ object orientated or generic programming features

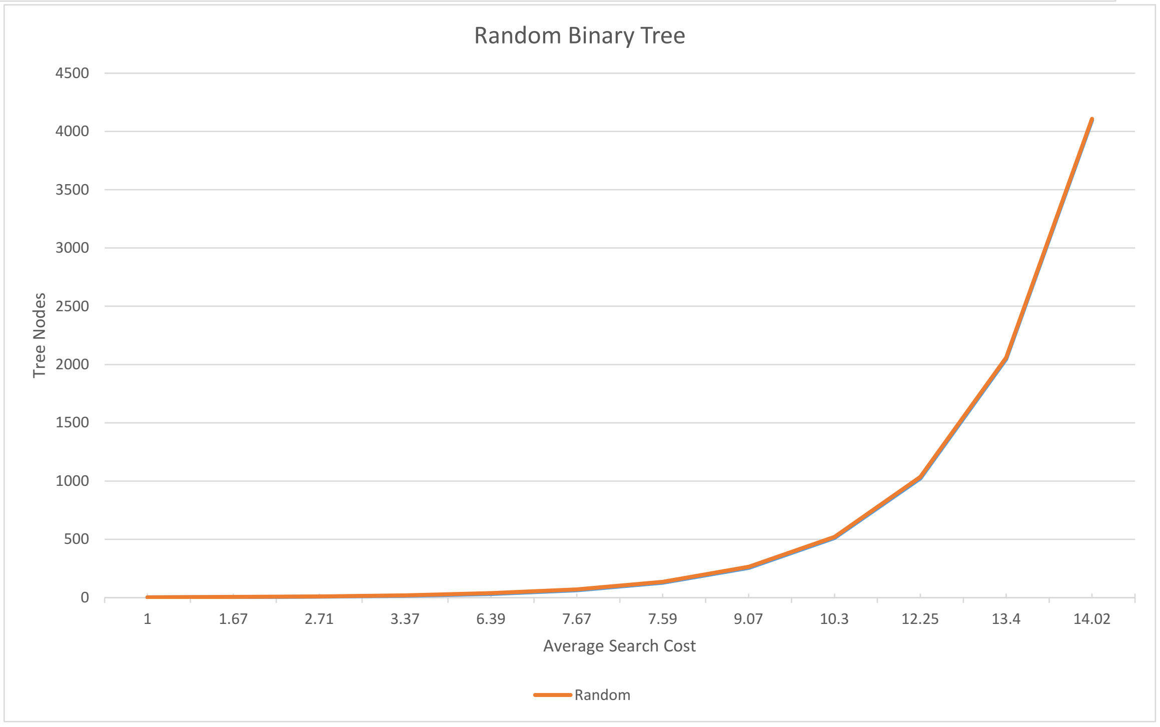
   (a) In this lab, only object oriented programming features were used. The classes in this lab were shown in the code attached above.

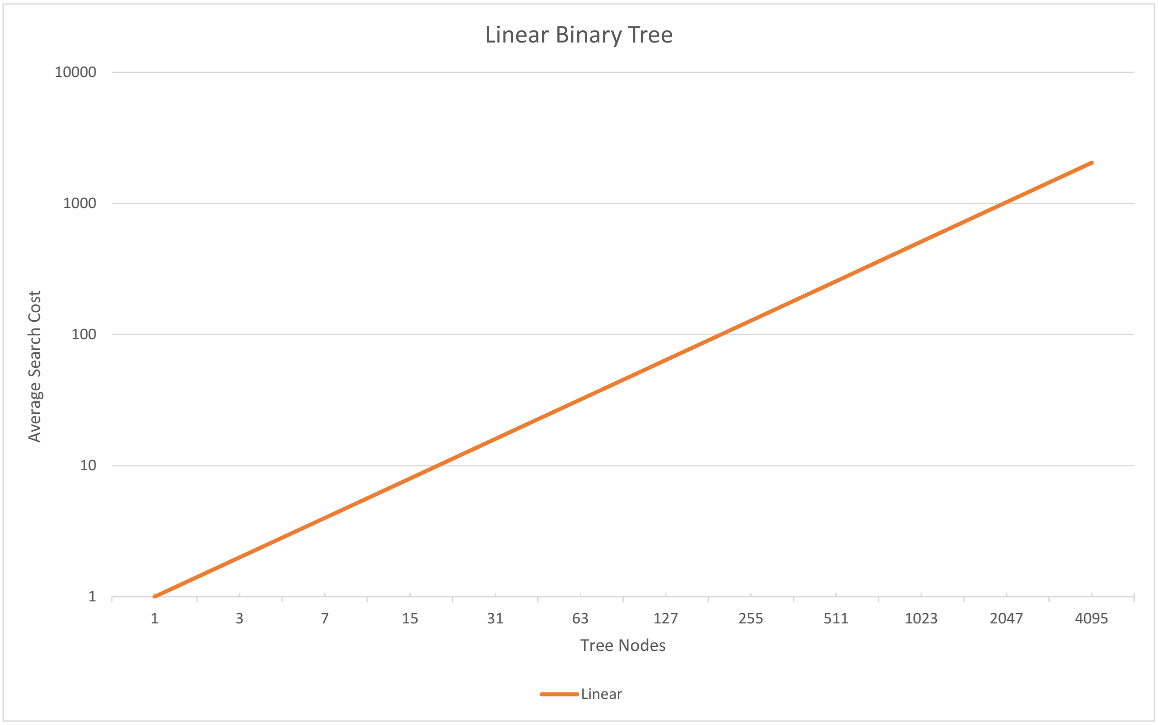
10. Charts and graphs

    (a) From the charts and the graphs, we can see that the average search time for linear search trees are $2n$, or in big-O notation $O(n)$. The average search time for balanced binary search trees are running on a time-complexity of $O(log(n))$.

Nodes	Perfect	Random	Linear
1	1.00	1.00	1.00
3	1,67	1.67	2.00
7	2.43	2.71	4.00
15	3.27	3.37	8.00
31	4.16	6.39	16.00
63	5.10	7.67	32.00
127	6.06	7.59	64.00
255	7.03	9.07	128.00
511	8.02	10.30	256.00
1023	9.01	12.25	512.00
2047	10.01	13.40	1024.00
4095	11.00	14.02	2048.00


Perfect Binary Tree
Tree Nodes
0
500
1000
1500
2000
2500
3000
3500
4000
4500
1
2
3
4
5
6
7
8
9
10
11
12
Average Search Cost
Perfect


Random Binary Tree
Tree Nodes
0
500
1000
1500
2000
2500
3000
3500
4000
4500
1
1.67
2.71
3.37
6.39
7.67
7.59
9.07
10.3
12.25
13.4
14.02
Average Search Cost
Random


Linear Binary Tree
10000
1000
100
10
1
Average Search Cost
1
3
7
15
31
63
127
255
511
1023
2047
4095
Tree Nodes
Linear