CSCE 221 Cover Page

First Name Last Name UIN

User Name E-mail address

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 on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

Type of sources				
People				
Web pages (provide URL)				
Printed material				
Other Sources				

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 Date

CSCE 221 Assignment 3 (100 pts)

Due Dates: See the Calendar

Objective

Write a C++ class to create a binary search tree. Your program should empirically calculate the average search cost for each node in a tree and output a tree, level by level, in text format.

Programming (100 points)

Write code for a binary search tree (BSTree). The **BSTree** class has already several functions implemented that you can use. You must implement the following functions:

- **Destructor:** Deallocates the memory of the tree using the `delete` keyword on each node of the tree. This can be done in multiple ways (e.g. use a recursive function, BFS or DFS).
- **Copy constructor & assignment operator:** given a BSTree object, create a copy of the tree without modifying the given tree. For the copy assignment operator, you must also check if you are trying to copy the tree into itself, in which case you do nothing. If there is a tree in the destination you need to delete it (using the destructor).
- **Move constructor & assignment operator:** given a BSTree object, move the contents of the tree in $O(1)$ time, and make the original tree empty. For the move assignment operator, you must also check if you are trying to move the tree into itself, in which case you do nothing. If there is a tree in the destination you need to delete it (using the destructor).
- **insert():** This function adds a new node with a given value to the tree, increments the size of the tree, and returns a pointer to the new node. The new node must be given a search cost, which is the number of comparisons required for searching a node (i.e. the number of comparisons = the search cost for the node = 1+ the depth of the node). Do not use **update_search_times()** for this. You may assume that all the values inserted are unique.
- **search():** This function returns a pointer to the node with a given value. If no node contains such a value, return a null pointer. You may assume that all the values in the tree are unique.
- **update_search_times():** This function updates the search costs for all the nodes in the tree. The search cost is the number of comparisons required for searching a node (i.e. the number of comparisons = the search cost for the node = 1+ the depth of the node). Do not call this function when inserting an element as this will change the time complexity of insertion.
- **inorder():** Traverse and print the nodes of the tree on an in-order fashion, i.e. first print the left subtree of a node, then the value of the node and finally the right subtree. If this is done correctly, it should display the values in ascending order. Use the output operator for nodes and add a single space between nodes (it should have no newlines). See the example below for reference.
- **print_level_by_level():** Traverse and print the nodes of a tree in a level by level fashion where each level of the tree is printed in a new line. Use the output operator for nodes and add a single space between nodes of the same level. See the example below for reference.

In addition to these functions, you must also ensure that there are no memory leaks.

The files to be submitted to Gradescope are: BSTree.cpp and BSTree.h

- Use **BSTree_main.cpp** to test your code on the data files provided.

1. The files *1p*, *2p*, ..., *12p* contain $2^1 - 1$, $2^2 - 1$,..., and $2^{12} - 1$ integers respectively. The integers make 12 **perfect binary trees** where all leaves are at the same depth. Calculate the average search cost for each perfect binary tree.
2. The files *1r*, *2*r, ..., *12r* contain same number of integers as above. The integers are randomly ordered and make 12 **random binary trees**. Calculate the average search cost for each tree.
3. The files *1l*, *2l*, ..., *12l* contain same number of integers as above. The integers are in increasing order and make 12 **linear binary trees**. Calculate the average search cost for each tree.
4. (Optional – no additional points) You may include a table and a plot of the average search costs you obtain. The x axis should be the size of the tree and the y axis should be the average search cost. Compare the curves of search costs with your theoretical analysis that is derived above.

Examples:

Input data:

```
5
3
9
7
10
11
```

Create a binary search tree using keys and provide information about each node when you display the tree.

```
Key  Search Time
 5        1
 3        2
 9        2
 7        3
 10       3
 11       4
Total number of nodes is 6
```

The in-order traversal for this particular tree is:

`3[2] 5[1] 7[3] 9[2] 10[3] 11[4]`

The format of each node is `value[search_time]`.

Sum of the search cost over all the nodes in the tree is: $2 + 1 + 3 + 2+3 + 4 = 15$.

Average search cost: $15/6 = 2.5$.

Output the tree level-by-level to a file (missing elements are denoted by . (dot):

```
5[1]
3[2] 9[2]
. . 7[3] 10[3]
. . . . . . . 11[4]
```

Hints

1. Besides using links/pointers to represent a binary search tree, you may store the binary tree in a vector. This implementation might be useful, especially for the printing of a tree level by level.

2. You can add your own recursive functions in the header file (BSTree.h) as long as you define them in the cpp file and you don't remove or change any already defined functions in the header.

3. You may use the **std::queue** and **std::stack** classes to perform BFS or DFS respectively

4. The pseudocode here is one way of implementing the level-by-level function. You can create your own version if you find it more convenient.

```
level_by_level(BinarySearchTree T)
    Queue q // contains elements from a level and its children
    q.enqueue(T.root)
    elementsInLevel = 1 // elements in the current level
    nonNullChild = false
    while (elementsInLevel > 0) do:
        TreeNode* node = q.dequeue()
        elementsInLevel--
        if node is not null:
            print node
            enqueue the children of node into q
            if at least one child is not null:
              nonNullChild = true
        else:
            print '.'
            enqueue null //  represents the descendants of the empty node
            enqueue null
        if elementsInLevel == 0 // the end of the current level
            print newline
            if nonNullChild == true: // the next level is non-empty
                nonNullChild = false
                elementsInLevel = q.size
```