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

Assignment 3 (100 pts)

Program: Due March 29 at 11:59 pm

Objectives: Programming

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 a text format.

1. **Programming (80 points)**.

   Write code for a Binary Search Tree implementation and submit it to Mimir. The BSTree class already has several functions that you can use in your own functions and **you may create additional helper functions if you find them convenient**. You must implement the following operations:

   - 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, or if there is a tree in the destination and if so, delete it.
   - Move constructor & assignment operator: given a BSTree object, move the contents of the tree in $O(1)$ time, and empty the original tree. 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, or if there is a tree in the destination and if so, delete it.
   - Insert: This function adds a new node to the tree with the value given, 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 the update_search_costs for this. You may assume that all the values inserted are unique.
   - Search: This function returns a pointer to the node of the tree with the value given. If no node contains such value, return a null pointer. You may assume that all the values on the tree are unique.
   - Update search costs: This function updates the search costs for all the nodes on 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 hurt the time complexity of insertion.
   - Inorder traversal: Traverse and print the nodes of the tree on an inorder 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 and it should have no newlines. See the example below for reference.
   - Level-by-level traversal: Traverse and print the nodes of a tree in a level by level fashion where each level of the tree is printed on 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 and you must provide a makefile. This makefile should create an executable called "run-trees" and use the BSTree_main.cpp and BSTree.pp source files. The files to be sumbitted to Mimir are:

   - BSTree.cpp
   - BSTree.h
   - Makefile (The makefile must use BSTree_main.cpp, but you should NOT include it in your Mimir submission. The final executable must be called "run-trees").

Report (20 points)

Write a brief report and submit it to Canvas. The report should include the following sections:

1. A description of the assignment objective, how to compile and run your program, and an explanation of your program structure (i.e. a high level description of the functions or classes in your code).

2. A brief description of the data structure you create (i.e. a theoretical definition of the data structure and the actual data arrangement in the classes).

3. A description of the implementation of (a) individual search cost and (b) average search cost. Analyze the time complexity of the functions that (a) calculate the individual search cost and (b) sum up the search costs over all the nodes. The recurrences/runtime functions should be from your functions (insert() for an individual search cost).

4. **Give an individual search cost in terms of n using big-O notation**. Analyze and give the average search costs of a perfect binary tree and a linear binary tree using big-O notation, assuming that the following formulas are true (n denotes the total number of input data). To be clear, part 3 asks you to analyze the running time of the functions implemented to compute the individual and average search cost, while here you must analyze the asymptotic behavior of the values of the search cost itself (Hint: depth of the tree affects the individual search cost.)

   Formula for perfect binary tree: $\sum_{d=0}^{\log_2(n+1)-1} 2^d(d+1) \simeq (n+1)\cdot\log_2(n+1) - n$

   Formula for linear binary tree: $\sum_{d=1}^{n} d \simeq n(n+1)/2$

   where d represents the depth of the tree.

5. Use BSTree_main.cpp to run your code to analyze the data files provided. In case you are unable to compile and run code outside of Mimir, contact your TA for assistance.

   (a) 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 and record the average search cost for each perfect binary tree.

   (b) 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 and record the average search cost for each tree.

   (c) 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 and record the average search cost for each tree.

   (d) 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. In your discussions of experimental results, 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 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 inorder traversal for this particular tree is:
```
3[2] 5[1] 7[3] 9[2] 10[3] 11[4]
Here 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: $15/6=2.5$.
```
Average search cost is 2.5
Output the tree level-by-level to a file (missing elements are denoted by X):
5[1]
3[2] 9[2]
X X 7[3] 10[3]
X X X X X X 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 on the header file as long as you define them in the cpp file and you don't remove or change any of the functions that are defined already on the header.
3. You may use the std::queue and std::stack classes to perform BFS or DFS repectively
4. The pseudocode here is one way of doing the level-by-level function. You can create your own version if you find it easier.

```
level_by_level(BinarySearchTree T)
   Queue q // This queue 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 'X'
            enqueue null // these nulls represent the descendants of the empty node
            enqueue null
        if elementsInLevel == 0 // We have reached the end of the current level
            print newline
            if nonNullChild == true: // The next level is non-empty
                nonNullChild = false
                elementsInLevel = q.size
```