Programming Assignment 3

Binary Search Tree

Part 1 (of 1)

Objectives

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.

Please Note: you are asked to implement a generic binary tree, not a balanced (AVL, Red-Black) binary tree

Report & Turn In

There is no written report for this assignment. There will be an oral report over the contents of this assignment in lab the week following the due date.

`!IMPORTANT` By submitting code to Mimir and/or Canvas you acknowledge and are bound by the Aggie Honor Code:

> *On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work*
>
> *An Aggie does not lie, cheat, or steal, or tolerate those who do.*

Your submission will be taken in place of a digital/physical signature.

Turn in your code to Mimir. You should submit the following files:

- TODO: add list here

Please also submit a zipped or tarball version of these files to Canvas.

Provided Materials

Starter code is provided on Github: `https://github.tamu.edu/csce221/pa3`

Additionally, sample test files are provided. You will need to create your own main and makefile to use them. See the code snippet below for some ideas.

```
#include <fstream>

//...

int main(void){

    BSTree bst;
    std::ifstream inFile(DATA_FILE_NAME);

    inFile >> bst;

    // do some checks to convince yourself things are running correctly
}
```

The Binary Search Tree Node

The binary search tree node is provided in the start file. It maintains two pointers, one to the left child and one to the right child. A constructor is provided for you. There should not be a need to modify this class, but if you do so, be sure that your changes do not alter the node class or deviate from the expected behavior.

The Binary Search Tree

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 *most likely* will need to create additional helper functions where you find them convenient. You must implement the following operations:

- *copy constructor* - ensure this is a deep copy
- *copy assignment operator* - ensure this is a deep copy, do not leak memory
- *move constructor*
- *move assignment operator* - do not leak memory
- *destructor* - do not leak memory
- `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 `update_search_costs` for this. You may assume that all the values inserted are unique.
- `search` - 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). Although the logic is similar, 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` - 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 `operator<<` for nodes and add a single space between nodes of the same level. If a space is empty, use a `X` character. See the example below for reference.

For the input: 5 3 9 7 10 11

5[1]

3[2] 9[2]

X X 7[3] 10[3]

X X X X X X 11[4]

Note: Again, notice this is NOT a balanced tree.

A couple helper functions have been implemented for you. You probably should not change their implementation, or you are likely to not pass mimir test cases. If you find yourself making significant modifications to these functions, we suggest reconsidering the remainder of your program for correctness and simplicity.

The already implemented helper functions are:

- `BSTree()` - default constructor for `BSTree`. Sets `root` to `nullptr` and `size` to `0`

- `operator<<(ostream&, BSTree&)` - simply calls `print_level_by_level` on the tree
- `operator<<(ostream&, Node&)` - prints a node to the stream in the format `a[b]` where `a` is the node value and `b` is the node search cost
- `operator>>(istream&, BSTree&)` - loops over the provided input stream and calls insert for each item extracted from the stream by `operator>>(istream&, int)`
- `BSTree::copy_helper(Node*&, const Node*)` - takes a reference to a pointer to a `Node` (**param0**) and a pointer to a `const Node` (**param1**). Creates a new `Node` with the same value as **param1**, assigns it to **param0**, and then applies recursively to the children of **param1** as children of **param0**
- `BSTree::get_total_search_time(Node*)` - iterates over the `Node` and all its children recursively. Returns the sum of search time for all `Node` s visited
- `BSTree::get_average_search_time()` - returns total search time divided by number of nodes. Returns `-1` if the tree is empty.
- `BSTree::get_root() const` - returns a pointer to the root node
- `BSTree::get_size() const` - returns the number of nodes in the tree

> *Learning Moment: the last two functions listed above are declared `inline`. This is because the functions are declared in the `.h` file and thus may appear in multiple compilation units. By making the functions inline we avoid the multiple declaration error which typically occurs with function bodies declared inside the `.h` file. This is not a magic solution for all cases; at compile time, `inline` functions are inlined where they occur in the machine code. If the function is very large (in terms of machine code) and used in may places, the same machine code is inlined in multiple places, potentially significantly increasing the size of the compiled file.*
>
> *Additionally, when compiling with optimizations, the compiler may automatically inline small functions. This can provide massive performance improvements due to cpu caching, instruction pipelining, and potentially even in the memory saved from saving a stack frame.*
>
> *This will not be on the test for this class, but computer science students will have plenty of opportunities to see the impact of this and other compiler optimizations as they progress through later courses.*

Feel free to implement and additional helper functions that you deem necessary.

Hints

You may (should) use `std::queue` `std::stack` and/or `std::vector` where necessary in your implementation.

`print_level_by_level` is quite complex. You need to keep track of the layer you are on and wether or not there is another layer. Additionally, you need to track the position in the layer where the item goes (see node 7 in the example above). One solution is to push the nodes of a layer into a queue (including some placeholder for empty nodes). If there is a non-empty item in the queue, print this layer and repeat the process. Stop once the queue contains only empty nodes. Notice that as you iterate across a layer, you can easily access the children in the order you need for the layer. If only we had a First In First Out data structure…