NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

BINARY SEARCH TREES (BST)

INTRODUCTION

The purpose of this assignment is for you to implement a fully functional binary search tree. No auto-balancing is done, but the insertion and removal functions will return information that will be useful when adding it at a later time.

YOUR TASK

You are responsible for implementing all of the following functions:

- For general binary tree functionality (in `bintree.h`).
  - `preorder(root, fn)`
  - `inorder(root, fn)`
  - `postorder(root, fn)`
  - `levelorder(root, fn)`
  - `delete_tree(root)`
  - `height(node)`
  - `count(root)`
- For dealing with binary search trees (in `bst.h`):
  - `bst_find(root, value)`
  - `bst_insert(root, value)`
  - `bst_remove_value(root, value)`
  - `is_bst(root)`
  - `bst_minimum(root)`
  - `bst_maximum(root)`
  - `successor(node)`

Some of these were in a previous assignment. If you got them working, feel free to use them again. If not, write them now.

BINARY SEARCH TREES

You should have already covered binary search trees in your course's lecture, but I'm including a brief summary.

A binary search tree is a binary tree where nodes are organized based on a comparison in a specific way. The comparison function we will use for this assignment is < (less than), although it's possible to implement binary search trees in other ways.

Using less than as our comparison, the following must be true for every node in the binary tree for it to be a valid binary search tree.

- Every node in the node's left subtree must have a value less than the value of the node; None of the nodes in the left subtree of a node may be greater than or equal to the node in question.
- Every node in the node's right subtree must have a value greater than the value of the node; None of the nodes in the right subtree of a node may be less than or equal to the node in question.

Note that we do not allow duplicate values in our binary search tree.

PROVIDED FOR YOU

The functions you are implementing are templated. I've used a convention for naming the parameters.

- `NODE` is the type used for a binary tree node that will have a `data` member, and `left` and `right` pointers. The actual node passed in may actually have that pointer, but the functions using `NODE` as their template parameter should not require access to any parent pointers to function, and you shouldn't use them. This will ensure that these functions can still be applied to node types that don't have them.
- `NODEP` will be used as the type for a binary tree node like `NODE` but required to have an additional `parent` pointer present. This is because the functions using it either use the `parent` to do their job, or they make changes to the structure of the tree, and may need to change `parent` pointers in the process of that. An example of a type fitting this would be `NodeLRP` in `nodes.h`.

The type that the test programs will actually be using is `NodeLRP`, which is defined in `nodes.h`.

```
template <typename T>
class NodeLRP {
  public:
    T data; ///< The data element, of type `T`
    NodeLRP <T> * left;  ///< A pointer to the root of the left subtree, or `nullptr` if none exists.
    NodeLRP <T> * right; ///< A pointer to the root of the right subtree, or `nullptr` if none exists.
    NodeLRP <T> * parent; ///< A pointer to this node's parent, or `nullptr` if it is root or orphaned.
    NodeLRP(const T &data, NodeLRP *left=nullptr, NodeLRP *right=nullptr, NodeLRP *parent=nullptr)
      : data(data), left(left), right(right), parent(parent) { } };
```

FUNCTIONS YOU IMPLEMENT - GENERAL BINARY TREE

`inorder(root,fn)`, `preorder(root,fn)`, `postorder(root,fn)`, `levelorder(root,fn)`

- `root` - The root of the tree to traverse
- `fn` - A function or function-like object that is called to visit each node during the traversal.

Each of these functions performs the traversal it is named after on the tree with `root` as its root, calling `fn` **on the node pointer** as it visits each node.

`delete_tree(root)`

- `root` - A pointer to the node at the root of the tree from which all nodes will be deleted.

Deletes all of the nodes in the tree with `root` as its root, freeing up any dynamically-allocated memory used.

`height(node)`

- `node` - the node whose height we are calculating

This function must return the height of the node pointed to by the `node` parameter.

For the purposes of this assignment, the height of a node is defined as the number of *edges* in the path from the node to its most distant descendant. This means that the height of the root node of a tree with no descendants is `0`, and the height of an empty tree is `-1`.

Your section's professor may have used a different definition in their lecture, but you will need to be aware of this definition for your program to match the reference output.

`count(root)`

- `root` - A pointer to the node at the root of the binary tree for which we would like to count the nodes.

This function must return the number of nodes existing in the tree whose root node is pointed to by `root`.

FUNCTIONS YOU IMPLEMENT - BINARY SEARCH TREE

`bst_find(root, value)`

- `root` - A pointer to the root of the binary search tree to search.
- `value` - The value to search for

This function returns a pointer to the node on the binary search tree with `root` as its root in which the value `value` can be found. If the value is not found in the tree, return `nullptr` instead.

`bst_insert(root, value)`

- `root` - the root of the binary search tree to insert into
- `value` - the value to insert

This function must find the position on the binary search tree with `root` as its root where a node with `value` would belong, and:

- If the `value` is not already there, insert a new node where it belongs and return a pointer to the new node.
- If the `value` was already present, return `nullptr`.

NOTE: Your other functions will depend on the `parent` pointer for the inserted node, so make sure that you set it appropriately.

`bst_remove_value(root, value)`

- `root` - a *reference* to a pointer to the root of the binary search tree to search and remove the value from. The node will be of a type that has a `parent` pointer.
- `value` - the value of the node you'd like to delete from the BST

This function searches for a node in the tree with `root` as its root that has the value given, `value`. It then proceeds to remove the node from the tree.

Remember, there are three cases for BST removal: - No children - delete the node and make sure its parent knows it's gone. Grab the `parent` pointer from the node before you delete it, and return that pointer. - One child - make the parent point to the child, make sure to set the child's `parent` pointer to reflect the change. Return the position of the child that moved up to take the removed node's place. - Both children - swap with the successor, which is guaranteed to have at most one node, and delete from the new position (which will involve one of the other two cases).

You can use the `parent` pointer as needed, and you will need to remember to set it when your node removal affects the parents of a node.

The purpose of the return code in this function is to point to the position on the tree closest to where the change occurred. The one child and no child cases tell you what you need to return, and the case with both children turns into the other cases once you've swapped with the successor. Knowing where the change occurred will be useful when working with AVL trees later.

NOTE: `root` is a **reference** to a pointer, so if the root node is removed or another node moves into its place, you will need to assign a new value to `root` to change the pointer directly.

`is_bst(root)`

- `root` - pointer to the root of the binary tree to check

This function must return `true` if, and only if, the binary tree with `root` as its root qualifies as a binary search tree. Everything in the left subtree of a node must be less than it. Everything in the right subtree of a node must be greater than it. This must be true for every node.

NOTE: It is possible to have an empty binary search tree – not having any nodes is not a violation of the rules for a BST.

`bst_minimum(root)`

- `root` - pointer to the root of the BST to find the minimum in

This function must return a pointer to the node containing the minimum value in the binary search tree with `root` as its root. Assume that the tree is a valid binary search tree.

`bst_maximum(root)`

- `root` - pointer to the root of the BST to find the maximum in

This function must return a pointer to the node containing the maximum value in the binary search tree with `root` as its root. Assume that the tree is a valid binary search tree.

`successor(node)`

- `node` - a pointer to the node in the binary search tree whose successor we would like to find.

This function must return a pointer to the node in the binary search tree that would be the *inorder successor* to the supplied node, `node`. If the node has no successor, then return `nullptr`. Remember that the inorder successor is the node that has the value that would appear *next* in an inorder traversal. This will be found in the node with the smallest value that is greater than the current node's value.

NOTES

Your work should be done in `bintree.h` and `bst.h`.

You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission.

TESTING

There are a number of testing programs included. These will be used to evaluate the functionality of your implementations of the required functions. Typing `make` will attempt to compile them all, and will succeed to the degree it can with whatever you have implemented at that point.

The table below has a list of the tests available, and they are shown in order from least complex to most complex.

Order	Test	Purpose
1	`test1`	Testing your simpler functions on a known tree. No BST insertion or removal.
2	`test2`	Testing your `bst_insert`
3	`test3`	Testing your `bst_remove_value` on a known tree.
4	`test4`	Interactive program where the BST can be thoroughly tested.

The expected output is contained in the `*.refout` files in the `output/` directory.

HOW TO SUBMIT

In the autograder, submit the following files (and only these files):

- `bintree.h` - contains implementations of the required functions for binary trees. You may have implemented portions of this for a previous 340 assignment, and it's ok to reuse those portions here. Do *NOT* use other people's implementations.
- `bst.h` - contains the implementations of the required functions that are specific to binary search trees.

GRADING CONSIDERATIONS

- Does it compile? Does it run? All of the tests should compile and run on turing/hopper with the `Makefile` provided, and points will be deducted for each test that will not compile.
- Does the output match for all of the tests? The autograder will check this, but I have provided reference output for each of the test programs where such a thing is relevant, and you can compare your program's output to what is expected. This output can be found in the files ending in `.refout`.
- Did you indent your code?
  - Indentation aids in the readability of source code, and if you're not indenting your code blocks, the grader will legitimately dislike you for it. I'm authorizing them to mark you off if you subject them to reading that.
- Did you document your code?
  - You need a docbox at the top of every one of the files you're required to change including:
    - Your name
    - Your zid
    - Your course section
    - A description of what the program does
  - You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
  - Add other comments inside your code blocks describing what you're doing and why.
  - The use of `doxygen` style comments is encouraged, but not required.