Dis mockingbird - He make da purty pet
His name be Greg. And oh, he like to peck.

the mockingbird, a sadhu

The BST & the Lazy BST

previously by Michael Loceff

In this quest, which is simply LOADED with rewards, you get to implement a Binary Search Tree (BST) and then generalize the class to support *lazy deletion*.

Roar or purr. It's up to you which you do first. But you get to do 'em both in this quest. It assumes that you already know how to make `BST`s. That was covered in CS 2B. The main focus of this quest will be its doozy cousin, the `Lazy_BST`.

You need to know how to work BSTs with little to no effort because you may need all the time you've got to debug `Lazy_BST` issues.

Those of you who saved up a ton of time by acing your previous easy quest way too soon (I hope that's all of you) - here is where you can put that saved time to good use.

A new notation

One of the things that this quest will make you accustomed to is a new notation: `*&p` (no I'm not swearing).

```
// parameter p is a reference to a pointer to a Bar object.
void foo(Bar *&p);
```

Remember how you had the luxury of having a head node in the Platypus quest and the head node always pointed to the first element of the collection?

Well, here's a great way to find out how to handle the situation when you don't have that luxury. If you don't have a fixed handle to a pointer that may end up getting changed in a function call, then you have to pass *a reference to the pointer*. The notation for this, though apparently awkward when you first encounter it, can quickly be internalized into an idiom you can (and should) get comfortable recognizing immediately.

This is important when you're being handed a tree's root to manipulate and you end up changing the root node itself. The reference to the root node allows you to reset the root pointer in the object within which your tree's root is held. Once you've done this a few times, it will stop looking mysterious. Yet, FWIW, I'd suggest that you try to implement a `BST` (in your own time) using the other way (a head node and a cursor) and see how that goes.

The BST

Not much to say here. Except that if you're not up to snuff yet on Binary Search Trees, this would be a good time to hit your reference material. Whether that's your text, video, interactive app, or discussion forum - go for it. Check in our sub if you're stuck.

Figure 1 shows a picture of the BST class. As you can see, it is a template class. I should be able to instantiate binary search trees of any type I want, as long as my type supports the less-than comparison operator.

You'll notice that a few of its public methods and all of its private helpers are missing implementations. Figure out which ones, and implement them.

Your entire implementation should be contained within the file `BST.h`, which you will submit.

Here are some *possibly* rewarding details to know about some of the methods:

- Note that a null tree is a tree. Thus it is a good sentinel. Methods that return node pointers may return the null pointer on failure.
- The `Node`'s deep copy method should return an exact (and disjoint) clone of the given node. Note that it is a static method which takes the source node as a parameter.

- `_insert(Node *&p, const T &elem)` should insert the given element into the subtree rooted at `p`. Note that `p` may be null. In that case, you would have to create a brand-new tree rooted at `p` (this would be the time when the reference parameter comes in handy). It is an error to insert a duplicate (return `false`).
- `_remove(Node *&p, const T &elem)` should remove the given element from the subtree rooted at `p`. Return `false` if the element does not exist in the subtree.

- `_find_min(Node *p)` should return a pointer to the node with the least element in the subtree rooted at `p`. It may be `p` itself. (When?)
- The public `find(const T &elem)` method should attempt to find `elem` starting at the root of the tree. If the element is found, it should return a *constant* reference to the element. Otherwise, it should throw an instance of `Not_found_exception`.
- `_recursive_delete(Node *&p)` should free up all heap space allocated for the subtree rooted at `p`. As the name suggests, save yourself a lot of effort by being recursive here.[1] Remember to set deleted nodes to null and to manage your `_size` correctly through the process.
- The public facing `to_string()` simply puts a wrapper around the serialized root node of the BST.
    - The first line should say "`# Tree rooted at [X]`"
    - The next line should say "`# size = [X]`"
    - This should be followed the result of `_to_string(_root)`
    - The last line should say "# End of tree"
- The private helper `_to_string(const Node *p)` should serialize the subtree rooted at `p` in depth-first left to right order:
    - Append a line with the given node (it's `_data` element), a space, a colon, and a space, then the left child's `_data` element, a space, and the right child's `_data` element.
    - Recursively append the left child's serialization
    - Recursively append the right child's serialization
    - If a node is null, then append the string "`[null]`" where it appears as one of the children of another node.
    - If a node has NO children (both left and right are null), then don't print a line for that node (i.e. You will not produce a line like "`X : [null] [null]`")
    - As an example, the following `to_string()` output describes the tree shown in Figure 2.

```
# Tree rooted at 19
# size = 9
19 : 9 29
9 : 4 14
4 : -1 [null]
29 : 24 34
34 : [null] 39
# End of Tree
```

[1] Any time the size of a data structure grows exponentially by a factor of 2 or more) along some dimension (e.g. the height of a tree), I think it makes sense to see if recursive methods might work. This is because if the dimension grows logarithmically with size (e.g. tree height), you can count on the fact that the maximum stack overhead of methods recursing in that dimension will diminish exponentially. (Discuss this).

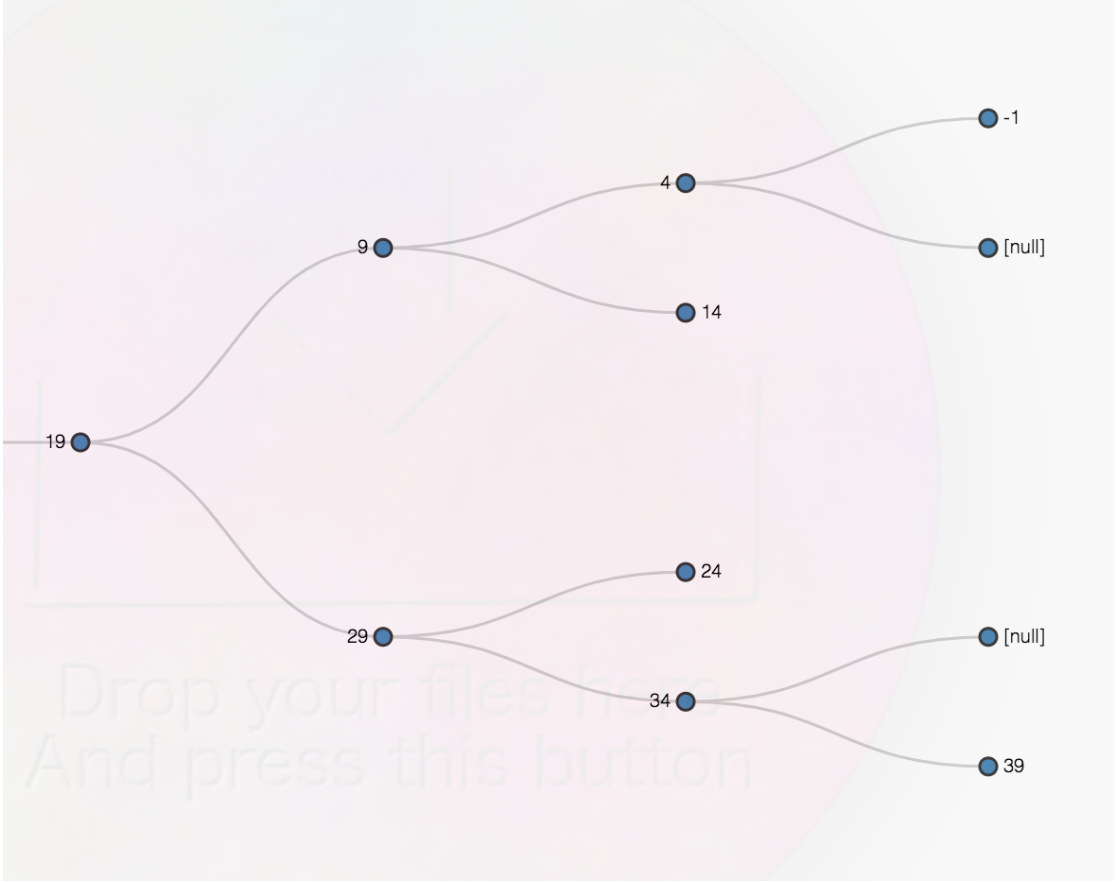

Figure 2. The Tree Rooted at 19

That's it. Let me know if you find out how many of these are tested for rewards in the quest.

Caution: Those of you who are familiar with the Loceff lab version of this quest may remember using an extra class member called `mRoot` to code tree membership into each node (every node knows the exact tree to which it belongs). This is so your tree's methods can check before accidentally operating on nodes from another tree. You are not required to do that check in this quest. So, beware!

```
// T must be Comparable. That is, must support ordering via <
template <typename T>
class BST {
private:
    struct Node {
        T _data;
        Node *_left, *_right;
        Node(const T &d, Node *l = nullptr, Node *r = nullptr) : _data(d), _left(l), _right(r) {}
    };
    Node *_root;
    size_t _size;

    // Helpers
    static Node *_deep_copy(const Node *p);
    bool _insert(Node *&p, const T &elem);
    bool _remove(Node *&p, const T &elem);
    bool _recursive_delete(Node *&p);
    const Node *_find_min(Node *p) const;
    Node *_find(Node *p, const T &elem) const;
    string _to_string(const Node *p) const;

public:
    BST() : _root(nullptr), _size(0) {}
    virtual ~BST() { _recursive_delete(_root); }

    virtual size_t get_size() const { return _size; }

    virtual bool insert(const T &elem) { return _insert(_root, elem); }
    virtual bool remove(const T &elem) { return _remove(_root, elem); }
    virtual bool clear();

    virtual bool contains(const T &elem) const { return _find(_root, elem) != nullptr; }
    virtual T &find(const T &elem) const;
    virtual string to_string() const;

    class Not_found_exception : public exception {
    public:
        string to_string() { return "Not found exception"; }
    };

    friend class Tests; // Don't remove
};
```

Figure 1. The `BST` Class

The Lazy_BST

What exactly is lazy about this BST? Deletion is.

When you try to delete a node in a `Lazy_BST`, it is not unlinked and released into the heap immediately, but simply marked as deleted. You need to store and manage this extra bit of information within each node.

What do you gain by doing this? Do any of your tree operations improve in complexity because of it? Under what circumstances does using a `Lazy_BST` make sense? All worthy questions to discuss in our sub.

As you implement the `Lazy_BST`, if you have any questions regarding its public interface - simply let the following guide your decision making process:

The fact that your `Lazy_BST` is lazy is something that should only be known to you, the developer of the class. No matter what its underlying implementation, the public interfaces for methods `Lazy_BST` and `BST` share should be identical. In other words, the `Lazy_BST` should present a successful illusion to its user that it is a plain old binary search tree. The only exceptions are: `collect_garbage()` and `to_string()` - both discussed below.

Figure 2 shows you a picture of the `Lazy_BST` class.[2]

Here are some *possibly* rewarding things to know about it:

[2] `Lazy_BST`, your lazy version of the `BST`, should not derive from `BST`. It should be a completely separate class. You may find it helpful to do a whole bunch of copy/pastes here.

insert(), remove(), find_min(), and find()

These are similar to their `BST` relatives. But they must now account for the fact that a node might be marked as deleted. For example, when inserting an element, you may find the element already in the tree in a deleted state. You would then have to unset its deleted flag.

_find_real_min() and _really_remove()

These correspond to your `BST`'s `find_min()` and `remove()` because now you can't believe their public versions (above) any more. They may now say a node has been removed, when *in reality*, it's still lurking in there!

In the node removal algorithm for a lazy tree, a node to be deleted must be replaced by the node with the real minimum in the tree (even if it has been marked as deleted), not with the minimum undeleted element returned by the regular `find_min()` (Why?).

The `_really_remove()` method really removes a node from the tree, not just mark it as deleted. This is the method that should be called from the lazy tree's garbage collector.

_collect_garbage(Node *p)

This is the private helper for the new public-facing garbage collector in `Lazy_BST`.

When it is invoked, it must scan the subtree rooted at `p` and release marked nodes into the heap. It's ok for this helper method to be recursive. If you code it systematically, you'll find it's short and simple:

- If the node is null, you don't have to do anything.
- Otherwise, recurse first on the left sub-tree, then on the right, and finally check `p`. In one of the basis cases, `p` would be marked as deleted. In this case, you would simply (really) remove p from the tree and return true. (What about other basis cases?)
- In any case, this method should return true if *at least one node* was released into the heap. That is, if collecting garbage resulted in a structural change to the tree. (When might you use this boolean value? Are there other, better. ways to tell if a tree needs a cleanup?)

`to_string()`

This is almost identical to that of a `BST` with one simple difference:

If a node is marked as deleted, then its element should have an asterisk appended to its name in the serialization.

```
class Lazy_BST {
protected:
    struct Node {
        T _data;
        Node *_left, *_right;
        bool _is_deleted;

        Node(const T &d) : _data(d), _left(nullptr), _right(nullptr), _is_deleted(false) {}
    };
    Node *_root;
    size_t _size, _real_size;

    // Private helpers
    bool _recursive_delete(Node *&p);
    bool _insert(Node *&p, const T &elem);
    bool _remove(Node *&p, const T &elem);
    bool _collect_garbage(Node *&p);
    const Node *_find_min(const Node *p) const;
    const Node *_find_real_min(const Node *p) const;
    const Node *_find(const Node *p, const T &elem) const;
    bool _really_remove(Node *&p, const T &elem);
    string _to_string(const Node *p) const;

public:
    Lazy_BST() : _root(nullptr), _size(0), _real_size(0) {}
    ~Lazy_BST() { _recursive_delete(_root); }

    size_t get_size() const { return _size; }
    size_t get_real_size() const { return _real_size; }
    bool insert(const T &elem) { return _insert(_root, elem); }
    bool remove(const T &elem) { return _remove(_root, elem); }
    bool collect_garbage() { return _collect_garbage(_root); }
    bool contains(const T &elem) const { return _find(_root, elem) != nullptr; }
    const T &find(const T &elem) const;
    string to_string() const;
    bool clear();

    class Not_found_exception : public exception {
    public:
        string what() { return "Element not found exception"; }
    };
    friend class Tests; // Don't remove this line
```

Figure 2. The `Lazy_BST` Class

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files[3] into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questing,

&

[3] `BST.h` and `Lazy_BST.h`