Yippee! Look. I found a tree. How very high the top is!
I hope I found another one. A yummy yooka laptus

General Tree

This is a really cool quest because it shows how a simple perspective change can make a data structure fundamentally different from another one to which it is structurally identical. Consider the linked lists you implemented in Quest 1. Remember how each node has a pointer to the next node. Suppose that each node had 2 pointers instead. Then your linked structure would look more like a binary tree, yes? See Figure 1.

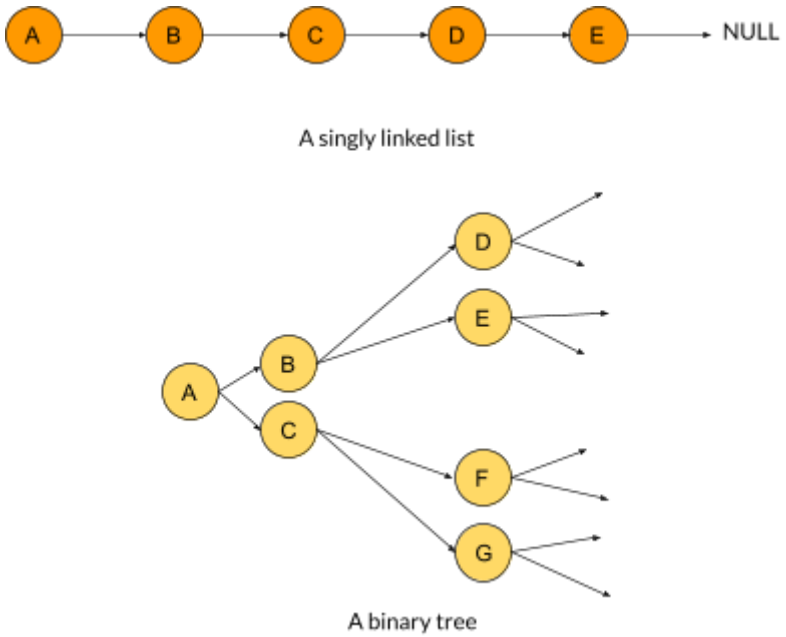


Figure 1: Two structurally and functionally different data structures

Now suppose you wanted a data structure in which every node had 5 children instead of 2, how might you implement it?

There are a bunch of different alternatives a programmer might consider (each has its own pros and cons - discuss them). Among them are these:

- Make each node have 5 separate pointers (`_child_1, _child_2,` etc.)
- Make each node have a linked list of children (5-children is now just a special case)
- Make each node have a vector of children (ditto)

#1 seems like a tailor-mode option for 5 children. But most of the time we realize pretty quickly that it's also its weakness. It will seem that it's not the right level of generality to code into your logic. What is?

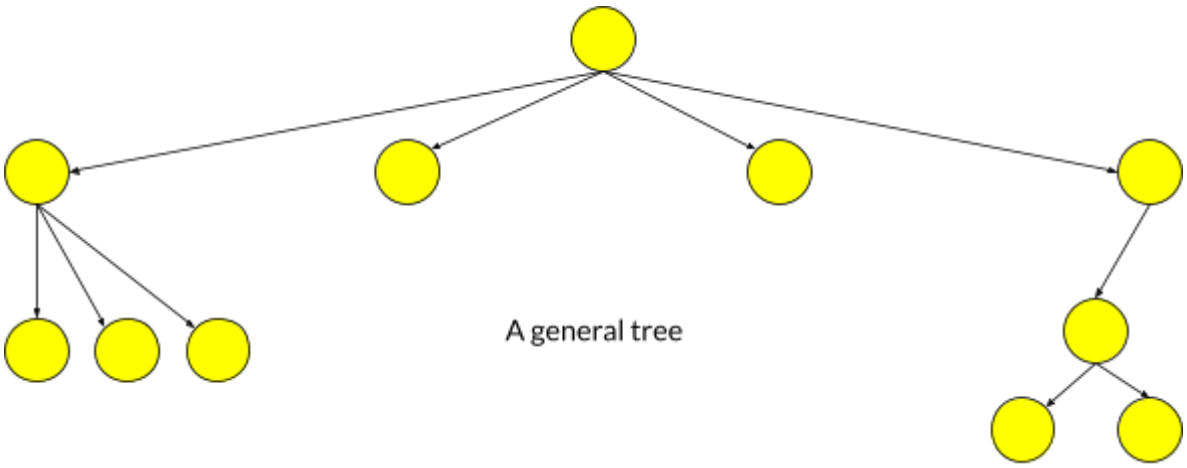
Ideally, you want a general tree in which nodes can have arbitrary numbers of children. How would you implement it?

The two surviving options from the previous list are now:

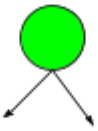
- Make each node have a linked list of children
- Make each node have a vector of children

Note that node objects from each of these options are structurally distinct from each other. The Nodes defined by these classes are packed differently even though they represent the exact same abstract data structure (a tree where nodes could have arbitrary numbers of children). These objects don't have equally sized sub-objects at corresponding byte locations.

Now you can ponder the central challenge in this quest: Can you use the binary tree's node structure as your node in a general tree? That is, can you make a general tree using just binary branching nodes?



A general tree



A single node from a binary tree

Incredible as it may sound, it is possible.

STOP READING THIS SPEC NOW.

Think up a solution.

Come back to this spec in a little while...

Welcome back.

See if you have the same solution as I do. I was very surprised when I first saw the equivalence: Leave the concrete data structure alone, but switch your own perspective!

In other words, *look at your binary branching node differently*. Instead of imagining the two node pointers as children of the current node, imagine them as the first child and next sibling.

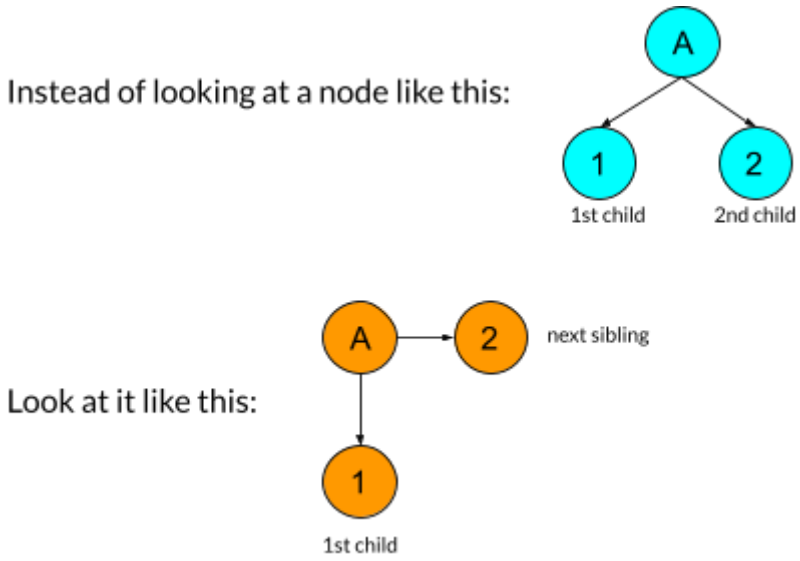
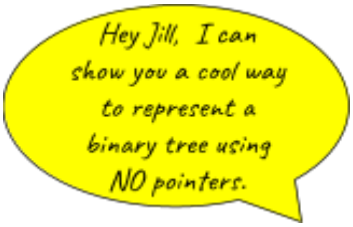


Figure 2: How to use a binary tree node to represent a general tree



Now take a break again to marvel at the incredibly transformative power of a simple perspective shift. Or not.

Either way, if you want, this is a good time to read your reference material on General Trees. They're also discussed at length in the recommended text. But if you understand the general concept, you can feel free to continue with this quest spec.

In fact, you may even be thinking whether you can implement a general tree using a node that has ONLY ONE pointer in it. Awesome, but wrap up this quest before experimenting further along those lines.

Now you get to implement a general tree using the 1st-child next-sibling (2-pointer) representation.

Starter Code

Here is part of the Tree class def. You need to find out what's missing and fill it in.[1]

```
class Tree {
private:
    struct Node { // Inner class
        std::string _data;
        Node *_sibling, *_child;
        static bool is_equal(const Node *p1, const Node *p2);

        Node(std::string s = "") : // TODO
        Node(const Node& that);    // TODO
        const Node& operator=(const Node& that); // Deep clone
        ~Node();

        std::string get_data() const { return _data; }
        void set_data(std::string s) { _data = s; }

        Node *insert_sibling(Node *p);
        Node *insert_child(Node *p);

        std::string to_string() const;

        bool operator==(const Node& that) const;
        bool operator!=(const Node& that) const;
    };

    Node *_root;

public:
    Tree();
    ~Tree();

    Tree(const Tree& that) { *this = that; }
    Tree& operator=(const Tree& that); // Deep clone

    std::string to_string() const;
    void make_special_config_1(const std::vector<std::string>& names);

    bool operator==(const Tree& that) const {
        // TODO
    }
    bool operator!=(const Tree& that) const {
        // TODO
    }

    friend std::ostream& operator<<(std::ostream& os, const Tree& tree) {
        // TODO
    };

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

[1] If you don't remember what a `struct` is (from 2A), look it up for more detail. It's simply a class where everything is `public` by default.

Before you proceed any further, review the `Tree` class definition until you have many questions (or not). You can also check out our subreddit (r/cs2b) to pick up questions from your classmates if they have extra ones to spare.

More detail on the individual methods can be found in the following miniquests. Please read them armed with the above questions. If you ask any that are still unanswered after reading the miniquests. I can use that info to improve this spec for your successors.

Your first miniquest - Node constructor

```
Node(std::string s = "") : // TODO
```

The Tree Node supports both default and non-default constructors. The non-default constructor takes a string parameter to use as the data element of the node. While you don't have to implement it inline, I recommend you take this opportunity to play with and learn the inline constructor definition syntax. You'll be coding a single line to go in the above red // TODO part in the header file.

Your second miniquest - Node insertions

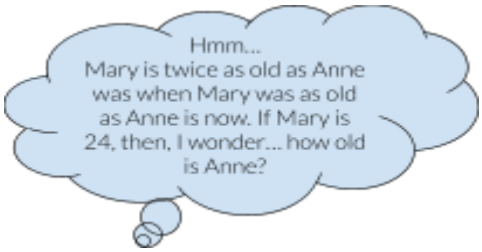
Now is probably a good time to refer back a few pages to read about how a general tree is represented using child and sibling pointers.

Implement:

```
Node *insert_sibling(Node *p);

Node *insert_child(Node *p);
```

In both cases, return the node pointer parameter back to the caller. You may NOT assume that `p` or the current node have no siblings or children of their own.



In `insert_sibling()`, first navigate to the last node in the list of siblings and insert `p` at the end.

In `insert_child()`, insert `p` as the child of the current node if it doesn't already have children. Otherwise add `p` as a sibling of your first child using the `insert_sibling()` method.

Your third and fourth miniquests - Node assignment

This one's a biggie. It's only ten lines or so, but it's a biggie. Make the assignment do the work of your copy constructor and your next miniquest will be a cake-walk.

Implement:

```
const Tree::Node& Tree::Node::operator=(const Tree::Node& that);
```

There's two parts to this miniquest. In the first part you have to simply create a copy of the original. You can easily get these points (even by cheating - how?)

To pass the second part, you have to make sure that your copy is unaffected even if your original gets clobbered. It's harder to cheat and pass this one. The straight solution is probably easier.

To make it even easier, here's some starter code for this method. Why do you think the IF statement in this method is important (if it is)?

```
const Tree::Node& Tree::Node::operator=(const Tree::Node& that);
    if (this != &that) {
        // TODO
    }
    return *this;
}
```

Your fifth miniquest - Node copy

Implement the copy constructor:

```
Node(const Node& that);
```

Hopefully you have node assignment working well by now. Collect this reward and move on.

Your sixth miniquest - Node comparisons

Implement:

```
static bool is_equal(const Node *p1, const Node *p2);
```

```
bool operator==(const Node& that) const;

bool operator!=(const Node& that) const;
```

It is ok to implement `!=` in terms of `==` (or vice-versa). The equals operator checks for deep equality defined as follows:

Two nodes are the same if they have the same set of siblings in the same order AND the same set of children in the same order.

Be recursive here and save yourself a ton of time. The is_equal() method is your private utility method that compares two nodes for equality, which you can invoke recursively on a node's children and siblings.

Your seventh miniquest - Node to string

Biggie. But just a bookkeeping biggie. When invoked on a `Node` object, it should return a string containing, in order:

1. The `Node's` data element, followed by a space and a colon (no newline yet)
2. The data-element of each of its children (if any) with exactly one space before each
3. A newline
4. If the node has a first child, the following additional strings:
    a. `"# Child of [X]\n"`
    where the red `X` in square brackets has been replaced by the node's data element.
    b. A string that is the result of a recursive invocation of `to_string()` on its first child.
5. If the node has a sibling, the following additional strings:
    a. `"# Next sib of [X]\n"`
    where the red `X` in square brackets has been replaced by the `node`'s data element.
    b. A string that is the result of invoking `to_string()` on its next sibling.

`to_string()` should not print anything. It must simply return the string that looks like the above output description.

Getting this right is largely a matter of taking care of little details, and remembering how the tree is structured (e.g. how do you traverse all of your children?)

If you're not passing this miniquest when your output *looks* identical to the reference output, it's possible that you just can't see the difference. Wow! That's a whopping big hint.

Your eighth miniquest - Node destructor

This miniquest isn't worth a lot of loot.[2] So don't futz around trying to implement an iterative sibling/child deleter like you had to do in the platypus quest. Just do a simple 5-line recursive version to score these easy rewards and move on.

However, you must remember to assign `nullptr` to any pointer you `delete`.

pandolas

Your ninth miniquest - Tree constructor and destructor

On to `Tree` methods. Implement the default constructor and destructor:

```
Tree();
~Tree();
```

An empty tree is defined as a Tree with no siblings and no children, whose own data element is equal to the string "ROOT". This is the tree a default constructor should make.

These are relatively straightforward methods. But again, remember to assign `nullptr` to any pointers you have to `delete` (e.g. `_root`)

[2] Ssshh! Here's a secret: The quester tester won't try and delete a node with an ENORMOUS number of siblings. (But why does that matter?)

Your tenth miniquest - Tree copy

Implement the copy constructor and the assignment operator, though not necessarily in that order.

```
Tree(const Tree& that);
Tree& operator=(const Tree& that); // Deep clone
```

The assignment operator should clone the RHS tree (perform a deep copy). The copy constructor should simply invoke the assignment operator. Again, make sure to check for the special case when someone accidentally assigns a tree to itself. What would happen then if you were careless?

Your eleventh miniquest - Tree comparisons

Implement:

```
bool operator==(const Tree& that);
bool operator!=(const Tree& that) const;
```

The == operator returns true if the two trees are structurally identical. Actual node pointers may be different. By *structurally identical*, I mean that they have the equal root nodes, where node equality is defined as in the earlier miniquest (I think 7).

Your twelfth miniquest - Tree to string

Implement:

```
to_string() const;
```

This is a very easy miniquest if you've managed to ace the `Node::to_string()` miniquest. Otherwise... not so much. It is also an optional miniquest (return "" to skip).

Simply return the stringified version of the root wrapped within *comment* lines as shown below:

```
# Tree rooted at [X]
# The following lines are of the form:
#    node: child1 child2...
[...]
# End of Tree
```

There is exactly one space before each word, except the word "node" on the last line where there are 3 spaces (gray rectangle). As before, the red X in square brackets must be replaced by

the name of the root node (which should be ROOT if everything is set up right). ↵ stands for a newline.

Your thirteenth miniquest - Special tree

This is a fun miniquest. All you have to do is to construct a Tree that looks like Fig 3. That's all.

Implement the method:

```
void make_special_config_1(const vector<string>& names);
```

Make sure to `delete` non-null sibling or children pointers before you start assembling the tree. Then insert nodes in the right order at the right places to recreate the tree in Figure 3.

Here is the list of node names[3] you must use for the three rows of nodes in the picture, top to bottom, left to right:

- `ROOT, AABA, ABAB, ABBA, BABA,`
- `COBO, COCO, CODO, COFO, COGO, COHO, COJO, COKO,`
- `DIBI, DIDI, DIFI, DIGI, DIHI, DIJI, DIKI, DILI`

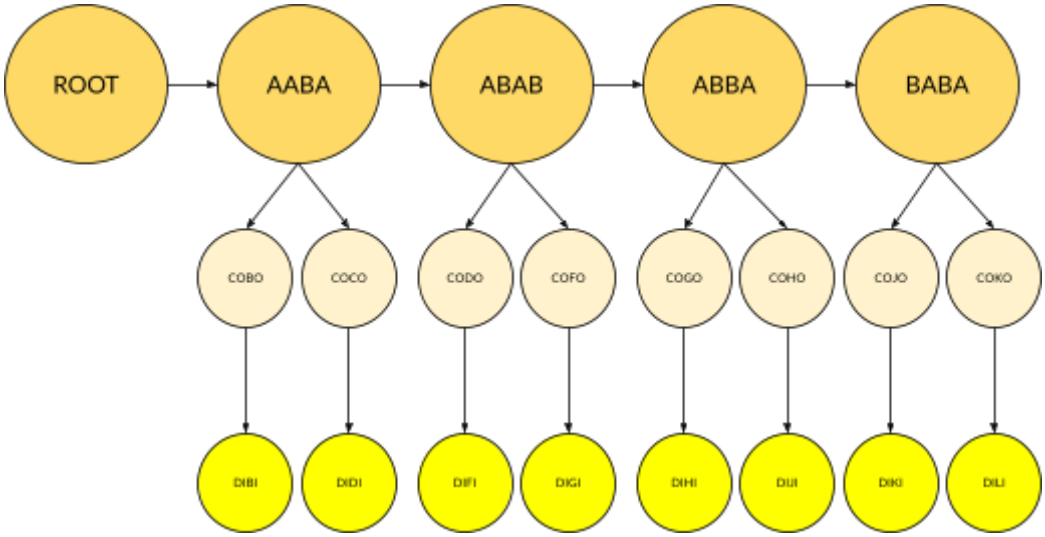


Figure 3: Special Configuration 1 (ROOT has siblings but no children).

In this abstract representation, horizontal arrows point to siblings and non-horiz ones to children (unlike in the UI) The leftmost-child is the first child of the parent. Absent arrows stand for an absent target node (e.g. no siblings or children).

[3] The vector param, `names = { AABA, ABAB, ABBA, BABA, COBO, COCO, CODO, COFO, COGO, COHO, COJO, COKO, DIBI, DIDI, DIFI, DIGI, DIHI, DIJI, DIKI, DILI }`

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 300 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

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 code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

Happy hacking,

&