Lab 9 - Binary Trees

CS 251, Fall 2024

Copyright Notice

© 2024 Ethan Ordentlich, University of Illinois Chicago
This assignment description is protected by U.S. copyright law. Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like chegg.com is explicitly prohibited by law and also violates UIC's Student Disciplinary Policy (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution). Material posted on any third party sites in violation of this copyright and the website terms will be removed and your user information will be released.

Learning Goals

By the end of this lab, you'll:

- Have implemented multiple recursive traversals over trees
- Implemented a memory-safe binary tree

Starter Code

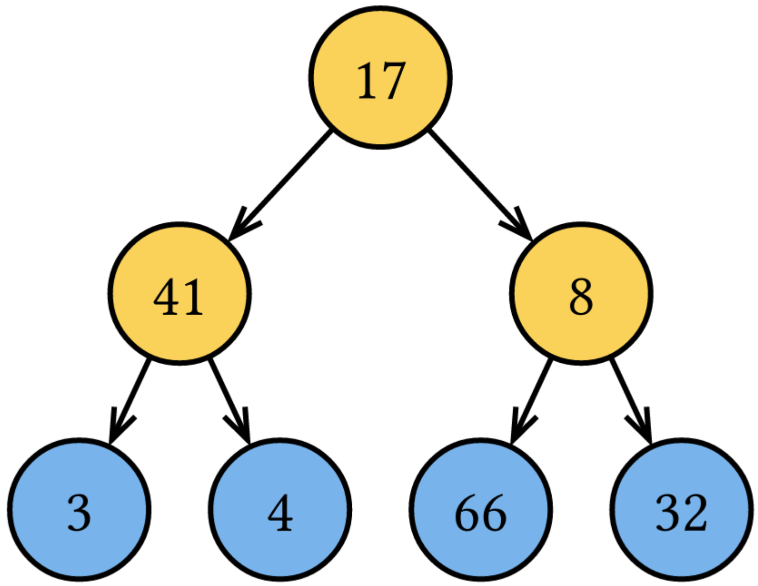
lab09-starter.zip

Tasks

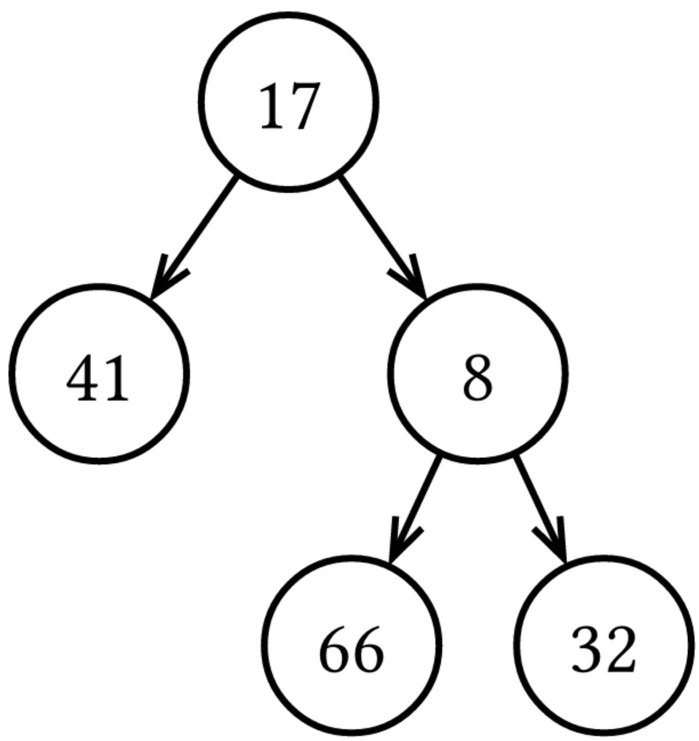
For this lab, work on the tasks in the **given order**. We strongly encourage collaboration in labs. Work with your peers, and ask your TAs questions!

Tree Terminology Review

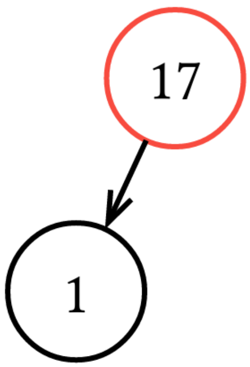
A **leaf node** is a node that doesn't have any children. An **internal node** is any node in a binary tree that is not a leaf. In the following example, leaf nodes are colored **blue** and internal nodes are colored **yellow**.

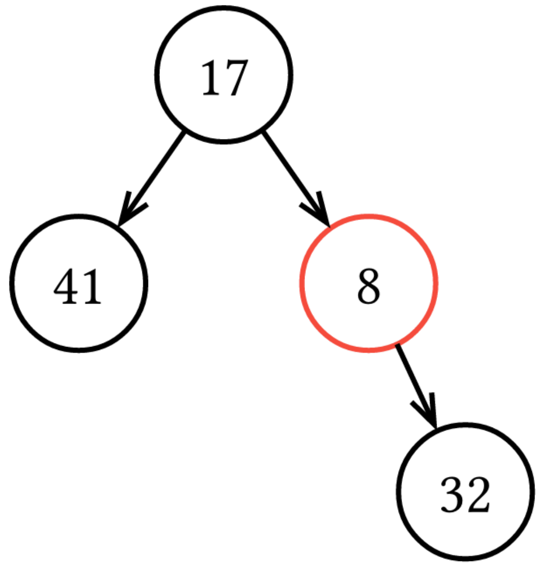


In a **full** binary tree (which is different from a *complete* binary tree), all nodes have either zero or two children. For example, both of these are **full** binary trees:



In contrast, neither of these binary trees are full, because the nodes highlighted **red** only have one child each:





StringTree

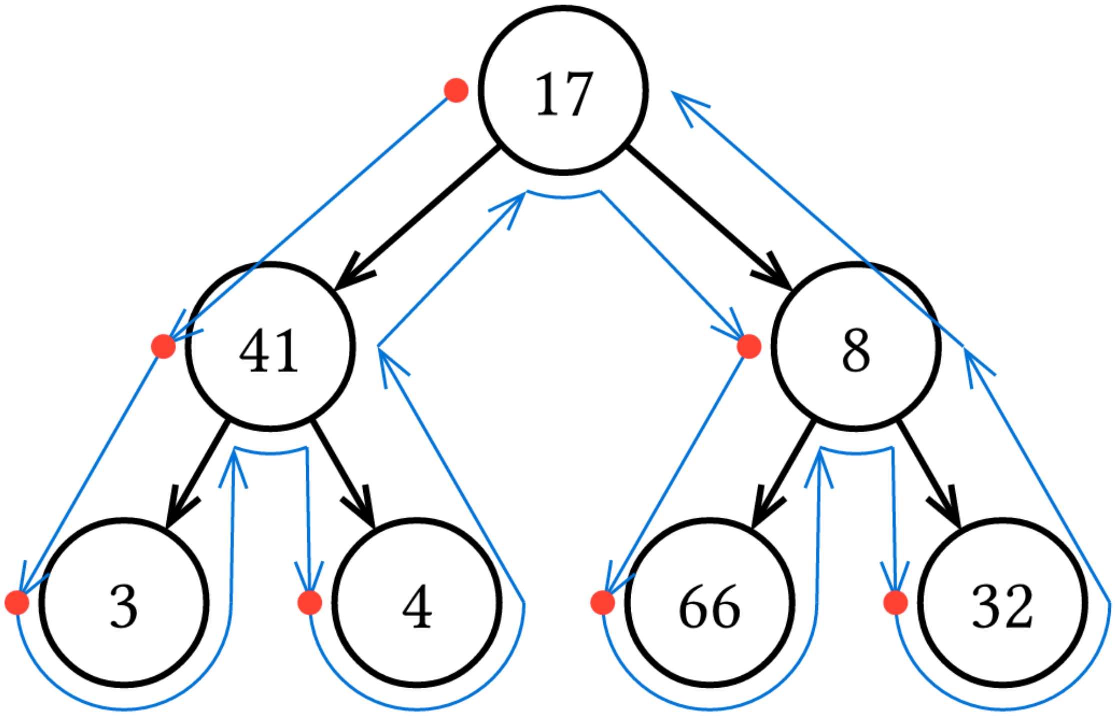
In this lab, we will implement a `StringTree` class, which represents a full binary tree that stores `string` data.

One common task is to save and load the data that's stored in a data structure. The process of converting a data structure to be stored (e.g. in a file) so that we can reconstruct it later is called *serialization*. The process of reconstruction is called *deserialization*.

In this lab, we will implement serialization and deserialization for a memory-safe binary tree data structure. To serialize a data structure, we need to choose a data format. For a tree, we also need to pick a specific order to save/load the nodes in. In lecture, we saw several different ways that we could traverse (iterate) the nodes of a tree – here, we'll use the *pre-order* traversal:

Starting from the root…

- Process the current node
- Recursively visit the left child until you reach a leaf
- Recursively visit the right child, until you reach a leaf



Following the blue arrows, we see that we reach each node for the first time (marked with a red circle) in the following order: 17, 41, 3, 4, 8, 66, 32. Even though our recursion leads us back to some nodes, it's not the first time we visit!

The file format we'll use to represent this is:

```
I: 17
I: 41
L: 3
L: 4
I: 8
L: 66
L: 32
```

That is, we have one line per element in the tree, where each line is prefixed with either `I:` (internal node) or `L:` (leaf node).

Constructor

```
StringTree::StringTree(istream& input)
```

Implement the constructor, which builds a new tree based on the contents of the given `istream` that contains a serialized tree. An `istream` can be any input stream, including `cin`, `ifstream`, or `istringstream`.

You must implement the constructor **recursively** using a public/private function. Declare your private function in `string_tree.h`.

Test your implementation with `make test_ctor`. You will leak memory, but this is fine for now.

Serialize

```
void StringTree::serialize(ostream& output)
```

Serializes the `StringTree`, and outputs the contents into the given `ostream`. You can use an `ostream` like `cout`, but make sure that you're outputting to `output` (not `cout`)!

You must also implement the serializer **recursively** using a public/private function. Declare your private function in `string_tree.h`.

Test your implementation with `make test_serialize`. You will still leak memory, but we're almost ready to fix that!

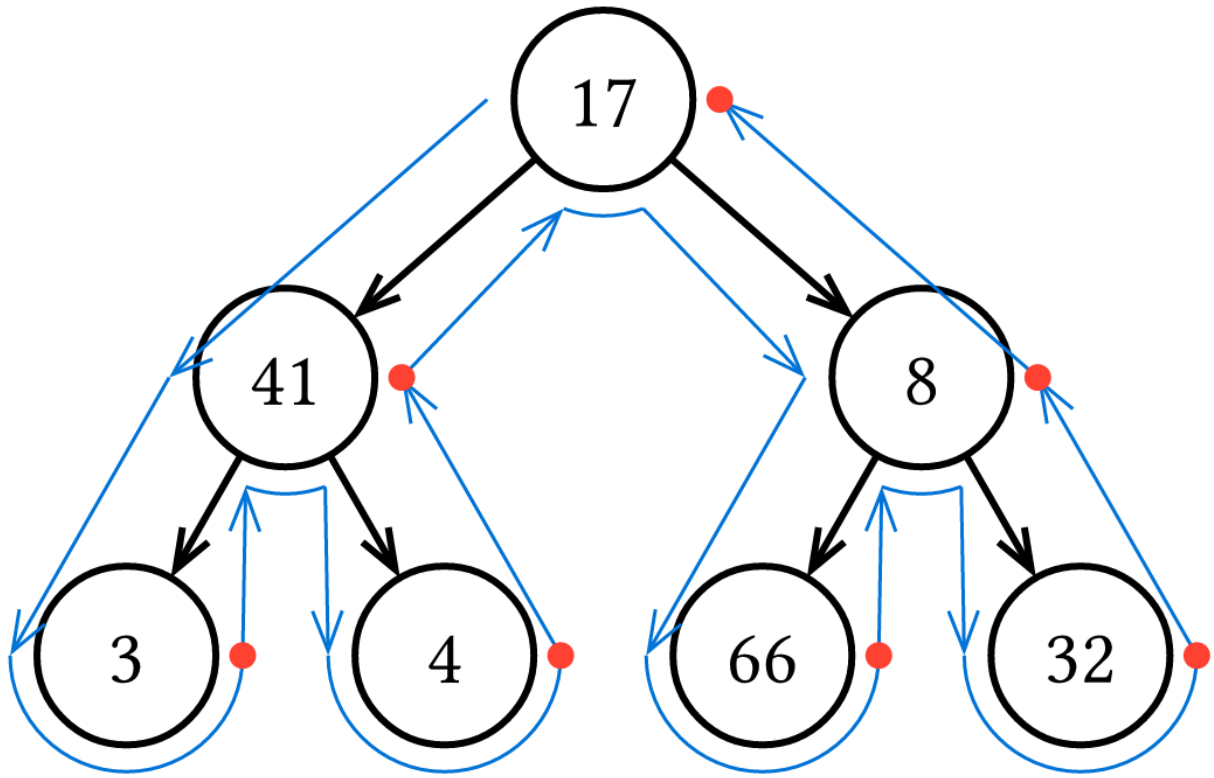
Destructor

Finally, to prevent memory leaks, we'll implement the destructor. While serialization and deserialization used pre-order traversal, the destructor should use **post-order** traversal.

This is only slightly different from a pre-order traversal: the "process" step happens *after* we recurse through both children.

Starting from the root…

- Recursively visit the left child until you reach a leaf
- Recursively visit the right child, until you reach a leaf
- Process the current node



Following the blue arrows, we see that we reach each node for the last time (marked with a red circle) in the following order: 3, 4, 41, 66, 32, 8, 17. The blue arrows are exactly the same as for pre-order! We just changed when we "process" the node, marking it with the red circle.

`StringTree::~StringTree()`

Recursively frees all allocated nodes in the tree.

You must also implement the destructor **recursively** using a public/private function. Declare your private function in `string_tree.h`.

Run `make test_all`. At this point, all your tests should pass, and you should have 0 memory errors or leaks.

Deliverables

- A completely passing test suite with no memory leaks.

Acknowledgements

Content by Adam Blank, adapted for C++ and CS 251 by Ethan Ordentlich