GenericTree and Tree Traversal

Ordered Data Structures — Week 3 Project
University of Illinois

Introduction

You've seen how several types of trees can be implemented. This week, you'll be provided with a simple templated tree type, and you'll implement a few small functions that operate on these trees.

C++ Environment Setup

If you are new to C++, we recommend checking out the first MOOC in this course sequence, which has more information on getting your C++ programming environment set up. In particular, we show how to set up AWS Cloud9 to do your coding entirely through your browser, but it's also possible to do this assignment on your own computer in Windows (with WSL Bash), in macOS (with some additional tools installed for the terminal), or in Linux natively. Cloud9 itself hosts a version of a Linux.

In the Week 3 Programming Assignment item on Coursera, where you found this PDF, there is a download link for the starter files: **`GenericTree_starter_files.zip`**. If you are using a web IDE like Cloud9, you should still download the zip file locally to your computer first, then upload it to your Cloud9 workspace. With the default Cloud9 workspace configuration, make sure you've updated the compiler first, by typing this command in the terminal:

```
sudo yum install -y gcc72 gcc72-c++
```

Now, in the Cloud9 interface, click **File > Upload Local Files**, and select your local copy of the starter zip file to upload it to your workspace. After that, you'll see that the zip file appears in the environment file listing on the left of the screen.

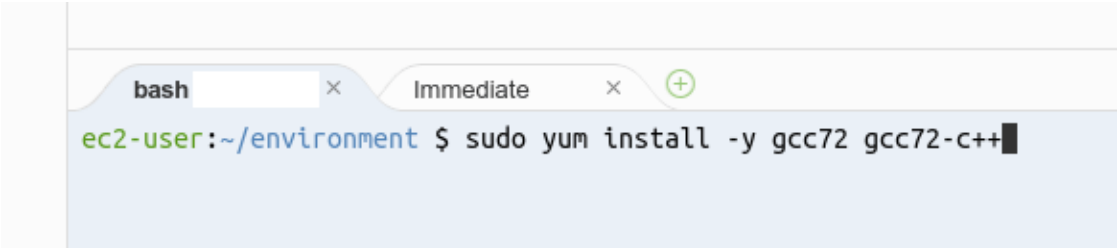

Fig. 1: Using a "yum" command in the Cloud9 terminal to update the compiler

Next, you'll want to extract the contents of the zip file to a new directory. Your terminal on Cloud9 already has some commands for this, **`zip`** and **`unzip`**. First, type **`ls`** in the terminal and make sure you see the zip file in the current directory listing. If not, then as described in the earlier readings, you should use the **`cd`** command to change directory until you see the file in the same directory. You can use the file listing on the left side of the window to help you see where you are. The **`pwd`** command ("print working directory") also reports where your terminal is currently browsing in the file system.

Once in the same directory as the zip file, enter the following command to extract the contents of the file, which contains a **`GenericTree`** subdirectory: (Be careful, because if you already have a subdirectory called **`GenericTree`**, this may overwrite files in that directory!)

```
unzip GenericTree_starter_files.zip
```

After you extract the code files, you can double-click them in the file list pane to see the editable document in the viewer.

Provided Files

You are provided with several files for the sake of this assignment. In particular, you are given a Makefile, and so you can compile your code from the **`GenericTree`** subdirectory just by typing "**`make`**" (for the main program) and "**`make test`**" (for the testing suite). The file that you need to edit is called **`GenericTreeExercises.h`**, but several other files have important clues about how to implement your solution. There are a lot of comments in the code, so you should try reading through the files.

About Templates in Header Files

You'll notice that much of the code for this assignment is contained in header files instead of cpp files. That's because *templated* C++ code generally needs to be written entirely in header files; when you use templates, the compiler generates specific versions of the templated code at compile time, depending on how you tried to apply the template in your code. In order for the compiler to use the template, it first needs to know about the entire template, which is usually done by putting all the relevant parts in the header. There are complicated tricks that programmers can use to move some parts of the template into cpp files for faster recompilation, but we'll overlook that for this assignment.

Files you should edit:

These are the *only* files you should edit, because the autograder will discard changes that you make to any other files.

- **`GenericTreeExercises.h`**: There are two exercises to complete, which are described in this document and in the code comments of the file itself.

Files you should read:

Among the other code files provided, some of them are meant for you to read through, even though you don't need to edit them.

- **`GenericTree.h`**: You can see how the **`GenericTree`** class implementation is done. There are comments throughout the class declaration and member function definitions.
- **`GenericTreeExercises.h`**: There are some more examples besides the exercises, along with some comments about syntax and design strategies.

- **`main.cpp`**: The main function that launches some examples and a few tests. The Makefile is set up for you to compile and run the **`main`** and **`test`** programs separately, although they do run some similar tests on your code. You should try them both and check the output in the terminal.
- **`tests/week3_tests.cpp`**: This configures the **`test`** program you can compile, and contains unit tests in Catch library syntax. You don't need to learn how to use this syntax, but this file shows what assertions the **`test`** program will check on your code, and the autograder on Coursera will also run these tests. If you are interested, the Catch documentation is here: https://github.com/catchorg/Catch2

Other files:

There are some additional files in the provided package, although you don't need to look at them. They're part of the mechanical inner workings of the assignment. For example, there are files containing Makefile definitions for the GNU Make program to use when it launches the compiler, and there is the source code for the Catch unit testing library. You are welcome to look at this code, but we won't discuss how it works.

Compiling and Testing Your Code

To compile the code, you must use the terminal to enter the same directory where the **`Makefile`** is stored; it's in the same directory where **`main.cpp`** is located. As explained in the readings, use **`cd`** to change to the appropriate directory, use **`ls`** to view the file list in that directory (just to make sure that you're in the right place), and then type **`make`** to automatically start the compilation. If you need to clear out the files you've previously built, type **`make clean`**. If you encounter any warnings or errors during compilation, study the messages in the terminal carefully to figure out what might be wrong with your code.

If your compilation is successful, you'll get an executable file simply called **`main`** in the same directory as **`main.cpp`**. You can try running it by typing **`./main`** while you're in that directory.

As the compilation message suggests, you should also run the unit tests included in the test program. To compile it automatically, just type **`make test`** at the same prompt. If this compilation is also successful, you'll see a file named **`test`** appear in the directory. Then, you can run it by typing **`./test`** similarly to before.

To compile and run the main program:

```
make clean && make && ./main
```

It is faster to just do "`make`" and not perform "`make clean`" first, but "`make clean`" can help you resolve certain compilation problems sometimes.

To compile and run the test suite:

```
make clean && make test && ./test
```

For this assignment, you can run the same test suite that the autograder will use on Coursera.

To make a package of your files for submission:

```
make zip
```

This automatically bundles only those files that have been specified for you to edit. For this assignment, only `GenericTreeExercises.h` will be collected.

The GenericTree Class

This project gives you "**`GenericTree`**," a class template. It is an "N-ary" tree, meaning each node can have many children, unlike a binary tree. There is no special balancing logic implemented for this tree, unlike an AVL tree or B-tree, although you could extend the class to add such a feature. The nodes are implemented to each hold an actual piece of data, as opposed to a pointer or reference to external data; however, the nodes are still connected by pointers, with each node having a pointer to its parent, as well as a **`std::vector`** of pointers to its own children nodes. (Recall that **`std::vector`** is a convenient container class similar to an array, provided by the Standard Template Library for C++.) You should try to read through the **`GenericTree`** header file and understand how it works.

The **`GenericTree`** class isn't as thoroughly constructed as a more mature general-purpose container like you would find in a library like STL. For example, we have disabled its copy constructor on purpose to simplify the implementation and also to prevent it from being misused for the sake of the assignment. This is especially important because the interface we provide to **`GenericTree`** exposes raw pointers to its internal nodes so you can work on it at a very low level of abstraction. A robust C++ library is usually designed to completely hide its internal memory structure, providing an interface of functions that allow users to operate on the object in an easy and safe way without knowing how exactly it is implemented.

GenericTree Text Output Format

We usually visualize trees with children nodes spreading out horizontally beneath their parent nodes, but **`GenericTree`** allows you to output the tree in text format to the terminal with a *vertical* orientation. In this style, what is usually the "leftmost" child of the root node (that is, the first child in the node's collection of children) will actually be displayed at the top, above the other children of the root node.

Here is a small example for comparison. Suppose the root node is A, its first child (the "leftmost" child) is B, and its rightmost child is C. In a typical tree diagram, that would be displayed as in Fig. 2, with A on top, B on the left, and C on the right. Nodes at the same level in the tree appear in the same horizontal row:

```
  A
 / \
B   C
```

Fig. 2: A horizontal tree diagram

However, if you create this tree as a **`GenericTree`** object and output it to **`std::cout`**, you will see the following vertical diagram (Fig. 3), which more closely resembles a file directory or nested list display style. The root node is still displayed at the top, but the leftmost child is then displayed topmost. Nodes at the same level in the tree appear in the same vertical column:

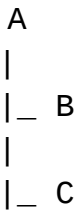

Fig. 3: A vertical tree diagram

Additional STL Data Structures

Besides the **`GenericTree`** class template that we provide, you'll notice that the provided code makes use of several Standard Template Library (STL) data structures, especially **`std::vector`**, **`std::stack`**, and **`std::queue`**. You can see how these are used in the provided code, and you might want to use them when you solve the problems. Additional documentation about those and other STL structures can be found readily on the web:

http://www.cplusplus.com/reference/vector/vector/
https://en.cppreference.com/w/cpp/container/vector

In particular, these member functions are quite useful: (for **`std::vector`**) **`push_back`**; (for **`std::stack`**) **`push`**, **`top`**, and **`pop`**; (for **`std::queue`**) **`push`**, **`front`**, and **`pop`**.

Programming Objectives

Study the Code

Since we have provided you with several files with examples and complete classes and functions to use, it's best to begin by reading the code that you are given. There are comments throughout and many hints about how to do the exercises. Some of the example functions and **`GenericTree`** member functions show coding techniques directly applicable to the exercises.

Complete the Exercises

Your task for this project is to complete the two exercises located in the **`GenericTreeExercises.h`** file.

Exercise 1: treeFactory

This function is passed a **`GenericTree`** object as input by reference (so no copying is done), and you need to modify that object in-place so that it becomes the tree illustrated in the code comments. Because an existing tree is passed in for direct modification, you should check if it has any existing contents and delete those contents properly first. There are examples in the provided code files showing how to construct a **`GenericTree`** using its member functions. The tree you want to recreate is shown here (you can also see this represented in the exercise source code):

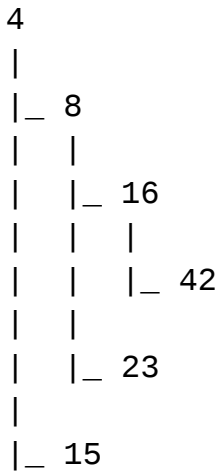

Fig. 4: The expected treeFactory results

Exercise 2: traverseLevels

This function needs to perform a level-order traversal of the input tree, and make a record of the data it finds (in order) using the std::vector class. The code comments provide some discussion about the design considerations that go into implementing this; for example, note that level-order traversal and breadth-first search are often natural to implement using a queue, while pre-order traversal and depth-first search are often implemented most simply using a stack. If you use a data structure to keep track of which nodes to traverse next and in what order, you should probably store pointers to nodes rather than full copies of the nodes themselves. There are examples of various similar functions in the provided code files.

You may also want to think about the different costs or benefits of writing functions like these recursively (the function making calls to itself on smaller and smaller problems) versus iteratively (looping explicitly). There are some notes about that in the provided code comments as well.

Submitting Your Work

When you're ready to hand in your work, you can type this command in the main subdirectory:

```
make zip
```

This will package the necessary files into a new zip file for submission. You can download the created zip file to your own computer from Cloud9 if necessary, and then submit the zip file on Coursera for autograding. (If you need any reminders about how to submit work on Coursera or use Cloud9, the first MOOC in this course sequence on Coursera has more detailed information about these topics in the lessons and homework assignments.)