CSC A48 - Assignment 2 - A mini music sequencer

This assignment is intended to help you practice and master the material we have covered in Unit 4 relating to ***BSTs*** and all the operations we can carry out on them.

The goal is to implement the data storage component of a mini music ***sequencer***. A ***sequencer*** is a program that stores, organizes, and allows a user to manipulate ***musical notes***. Sequencers enable individual artists to arrange and perform songs that involve multiple instruments playing together - by recording the parts played by individual instruments one by one, and then having the sequencer play them back while the artists takes care of the main track (or the vocals). There are many free, open source sequencers you can play with if you're curious. For example, ***Hydrogen*** is a drum pattern sequencer you can install and play with (comes with a pretty high quality set of drum sounds):

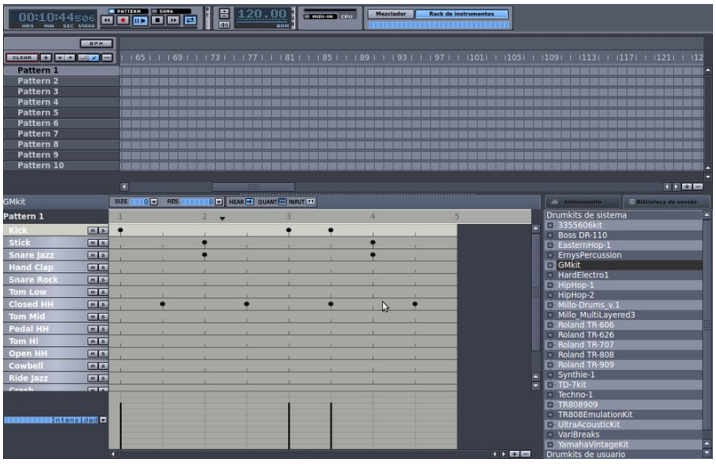


Figure 1: The Hydrogen sequencer main window. Image: Heyokha, Wikimedia Commons, CC-SA 3.0

Of course, the central component of a sequencer is a data structure for storing, managing, and manipulating musical notes. You're going to implement a small, but fully functional sequencer storage application based on ***BSTs***.

The data you will be manipulating consists of musical notes. In case you've never studied music ***don't worry - you do not need to know anything about music to work on, complete, and learn everything you are expected to learn with*** this assignment.

Your Task:

Implement all the functionality associated with a ***BST***.
- Creating and inserting nodes into the BST
- BST search
- Deleting nodes
- Traversals (all three versions)
- Deleting (releasing memory reserved for) a BST
- And a couple additional ***crunchy*** functions related to manipulating and playing musical notes

First step:

Download, unpack, and carefully study the starter code.

There is only one file where you have to implement code: 'BSTs.c'

The remaining .c files are:

- A2_test_driver.c - Your test driver for A2, to help you test and debug as you're working on the assignment. Same as you did in A1
- A2_interactive_driver.c - An interactive driver to help you manually test your ***BST*** and which you can use to manipulate musical notes and play-back the music stored by your sequencer.
- NoteSynth.c - This file implements a simple but fully-functional software synthesizer. It receives a list of notes from your sequencer, and produces the sounds corresponding to each of them, at the right point in time, to play the notes in your tree. ***You do not need to even look at this file if you don't want to***. It is not required of you, and you're not going to be tested on anything there. However, ***after you complete your BST***, if you're curious you can study this file to learn how sound is made so you can listen to the music you stored in the tree.

Additional to the .c files, there are several text files.

- note_frequencies.txt - Has a list of notes, with the name and frequency of each note. * DO NOT CHANGE THIS FILE * as it is required by NoteSynth.c to make music.
- All other .txt files - Contain song information you can use to test your BST. (the interactive driver can load data from any of these files into your tree once you've implemented the relevant functions in BSTs.c).

Compiling your code:

To compile your BST code with the test driver, use:

```
gcc -Wall A2_test_driver.c -lm
```

Note 1: If you try the above before you complete the compound data types for the ***BST_Node***, you will get a series of compiler errors.

Note 2: There is something new here! the '-lm' option tells the compiler you are using functions from the math library (math.h), and that it needs to take the code for those functions from the math library to add it to your executable file. The code won't compile without that! (it will complain it can't find code for math functions).

To compile your BST code with the interactive driver use:

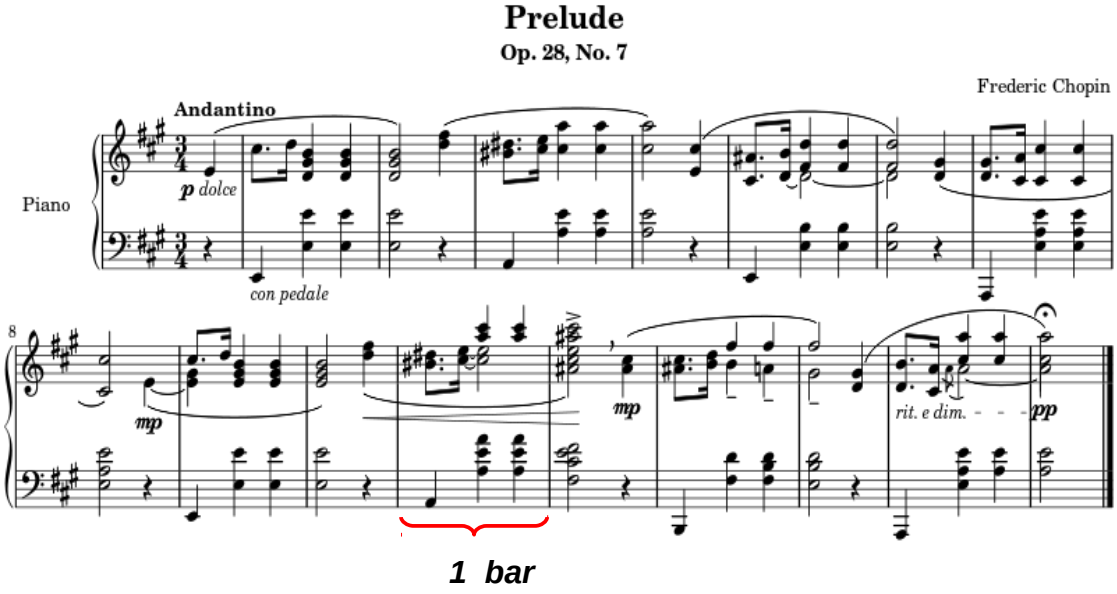
```
gcc -Wall A2_interactive_driver.c -lm
```

All the functions you need to implement are described in the starter code in BST.c. So read it carefully, and implement things according to the specifications in the function headers.

What is the data in the tree?

It consists of musical notes, each note is represented by:

- The frequency of the sound produced for this note (a floating point number)
- Its position in the song in terms of ***bar***, and ***index***.



Drawing 1: A sample musical score. Image: Prof.rick, Wikimedia Commons, Public Domain

Musical scores are split into ***bars*** (one of which is shown above). They are helpfully separated by... *you guessed it! vertical bars!* So we identify for each note which ***bar*** it belongs to. Then, within that ***bar***, we use the ***index*** to specify very accurately at which point the note is to be played. The ***index*** for our sequencer is a floating point number between 0.0 and 1.0. If the ***index*** is 0.0, the note plays exactly at the beginning of the bar, and if it's 1.0 it plays exactly at the end.

That is all the data you need to store for each note!

However, there is a catch: We know that ***BSTs*** and in general any data structures where you will support ***search*** require us to provide some ***key*** value that uniquely identifies an item. We will definitely find notes that have the same ***frequency,*** the same ***bar***, or the same ***index***, so none of these fields alone can work as key.

Instead, we will use a combination of them. For each note, we define its unique ***key*** to be:

key = (10.0*bar) + index

And we require that there be ***no duplicates***. That means your ***sequencer*** won't allow you to have two notes playing at exactly the same time. But that's ok, because the ***index*** is a floating point number, so if you have a musical score where two notes are playing together, all you need to do is add ***a very tiny amount*** to the ***index value*** for one of them before inserting them in the tree. That will give them a different ***key*** value in the tree.

Question: Are ***keys*** ordered in some way related to how the musical notes are organized?

How do I test this if I don't know music?

You can just add notes to your tree with made up ***frequency, bar, and index***. You will need to understand what their ***key*** value will be so you can tell where they should go in your tree, but that's all you need to worry about if you don't want to deal with music!

When testing: Write down the tree that would result from the data you're adding to it, and the operations you're carrying out on it! that's the only way to figure out if your ***BST*** is working properly. And along those lines --

Test, test, test!

We provide you with two tools for testing, but as you learned from A1, the tests we give you are only a ***subset*** of the tests we will run on your code once you submit it. Remember: ***one of the goals of the course is for you to develop your skill as a good software tester***. Therefore, we expect you to put your code through serious tests of your own, and to make sure every function works - first on its own, and later as a part of the sequencer.

Same as with A1, passing all the tests in the test driver ***does not mean your code is bug free and you will get 100%***. It only means your code seems to be doing the right thing with a couple of inputs.

Submitting your work:

As usual, submit your .c program file on Quercus, with the name:

BSTs_studentNo.c

Have fun playing music with your BSTs!

On the 'crunchy' functions:

You can test them from the interactive driver, but you need to know what they do:

- The reverse song function. This one is interesting – it takes whatever song is currently in the tree and modifies the song so that it plays backwards (from the end to the beginning, with each note in the correct reversed order). There are several ways to do this, and you have to find one that you are confident does the right thing. The header for the function gives you the details you need to complete this function.

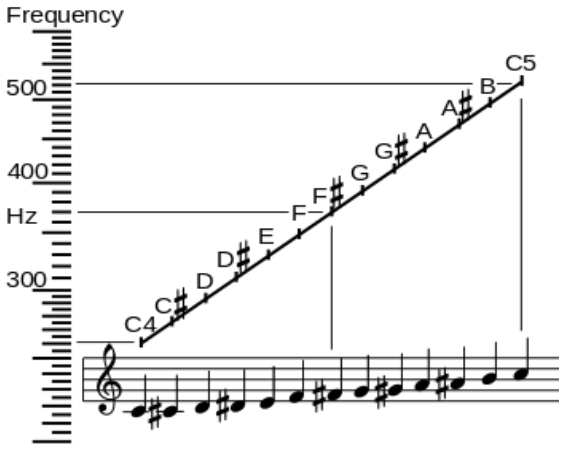


Figure 2: A subset of musical notes on a score, their name, and their frequency. Image: Jono4174, Wikimedia Commons, Public Domain

How do you test this?

You can do an *in order traversal* of the tree and print out the contents ***before*** the reversal, and then ***after*** the reversal, and check that notes are backwards and at the correct bar and index. Of course you can also output the original and the reversed songs and listen to that, if you have a musical ear, this may be more informative (if you use a cleverly designed test input!). Once you're happy your function works, try it out on one of the test songs provided!

- The ***Harmonize*** function: This is a bit more fun, and you should test it on the interactive driver. What it does is, go into your ***BST***, and for every existing note, insert a new one that is shifted by a certain number of ***semitones*** in ***frequency***, and by a specific amount of time in their ***index value***.

It will take a bit of thought to implement, but once you have it, you should be able to create all kinds of wacky sounds by taking an input set of notes and adding notes shifted by different amounts in terms of time, and in terms of frequency.

How to test this?

Add a few notes to the tree, then call harmonize with ***k*** semitones, and ***t*** time shift, and check that:

- Notes were added for each existing note
- Their ***frequency*** corresponds to the note that is ***k*** entries away in the note table
- Their ***index*** is the ***index of the original + t***.

Once you are satisfied this is working with a couple of notes, try it out by loading a song (start with the BST test inputs! they are short) and trying it out on those. You should have completed the code that allows you to play back a song, so you can hear what happens when you harmonize your tree!

Music output:

The code will produce a standard .wav file with the sound corresponding to notes in your tree - but only after you have implemented the relevant code!

Several test files are provided that you can play back - try them out on their own, and then mess with them using the frequency shift, and/or the harmonize functions you built.

Problem solving:

The crunchy functions are the part of the assignment that is intended to help you build and practice your ability to come up with a solution to a problem. Therefore:

- Do not ask us to confirm if your idea is right or wrong: Try it out, improve it, or change it if it doesn't look promising. You're expected to come up with your solution without us telling you at each step that you're allowed to try things.

- Do come to talk to us in office hours if you're stuck with something, or if you want to discuss your plans (not your code) for implementing either of these functions. We'll be happy to help!

Marking scheme:

[5 marks] - Completing the node structure for the BST
[10 marks] - Creating a new node, and inserting it into the BST (in the right place)
[5 marks] - BST search
[5 marks] - BST delete case a) (node without children)
[10 marks] - BST delete case b) (node with one child)
[15 marks] - BST delete case c) (node with two children)
[10 marks] - All tree traversals (for printing)
[5 marks] - Make play list
[5 marks] - Delete BST

That's up to ***70 marks*** for the basic functions that your BST must perform. Then:

[10 marks] For the the reverse-song function
[20 marks] For the harmonize function

For a total of ***100 marks***.

That's it for A2 - You have built a mini sequencer!

Done with all the required functions for your A2? If you like, you can expand the functionality of the synthesizer component:
- Add different waveforms – square, sawtooth, triangle, for a good-ol 8-bit sound!
- Make the Karplus-Strong plucked string sound like an electric guitar
- Add support for multiple instruments (note data needs to have instrument index, the playback function needs to select an appropriate waveform to play it)
- Add more advanced sound shaping (feel free to come chat with me if you want to try this out)