CS 345 (Spring 17): Analysis of Discrete Structures

Project #2

Binary Search Trees and `.dot` Files

due at 6pm, Thu 2 Mar 2017

1 Introduction

In this project, we'll be practicing implementing a BST (Binary Search Tree). We'll also be learning to use a tool called `dot`, which generates pictures of graphs[1]. We'll be implementing the data structure in Java (or C), and using the `dot` graphs for debugging.

Later, as we implement more complex data structures, we'll use `dot` again. You'll find that the debug help will be invaluable.

1.1 What to Turn In

You will turn in either Java or C source files (not both). You may turn in one or more source files - but if you turn in multiple C files, you **must** include a header, which is `#include`'d by the `.c` files, and which declares all of the shared types and functions.

If your program is written in Java, then your main class must be named `Proj02_BST`. (If your program is written in C, you can name the files anything you want.)

In addition, you must turn in a README file (name it either `README` or `README.txt`). Include in the README a quick overview of your program (what works and what doesn't), any interesting notes about your design or what you learned, or anything else that will help your TA grade your code.

You may zip up all of these files into a single `.zip` file, or you may upload them individually to D2L.

2 Binary Search Trees

You've already seen BSTs in other classes, and we've reviewed them in this class (see the end of Deck 1). You should know how to search through a tree, how to perform a pre-order or in-order traversal, and how to insert. You probably know how to perform deletion - but if not, Russ will be posting a video (and slide deck) to review the material.

3 Program Overview

You will write a program, in Java or C, which implements a Binary Search Tree which stores (non-negative) integers. Your program will read from a file,

[1] Mathematical graphs - that is, vertices and edges.

which contains a series of commands; you will perform the required commands (updating a BST in memory). Some of the commands will change the tree; some will require you to print out something about the tree; some will require you to create a `.dot` file, for debugging.

Your program will take a single argument, which is the name of the file to read; if no file is given, then print out an error message and end immediately. Likewise, if you cannot open the filename give, print out an error message and end immediately. (Our automatic grading script won't test these error cases, so it doesn't matter what you print - although the TAs will check that you printed out **something.** Give a reasonable error message.)

Each line of the file represents a command. Each command is a single word; command names are case-sensitive, and begin on the first character of the line. Most commands also have a single argument, which is separated from the command by a single space.

3.1 No Need to Error-Check the Input

In our testcases, we may have blank lines (ignore these), but other than that, we will not have any spurious whitespace - no leading spaces, trailing spaces, tabs, or extra spaces in the middle. Likewise, we will only use valid commands (no invalid command names). Finally, every command will always have the right number of arguments, and the argument (if any) will have the right type and a reasonable value. **You are not required to check that our testcases are valid.** Instead, you can simply **assume** that they are in the correct form, and read them without much error checking. Notably, if you are writing in Java, you are welcome to use `Scanner.next(), Scanner.nextInt()` (or any other class) to read the input file.

3.2 The Example Executable

We have provided an example executable which you must use for checking your output, along with a grading script which will run testscases against both your code, and also against the example executable. (Unfortuantely, since the example executable is a compiled C program, you will need to run it on an x86 Linux machine - this can be Lectura or any x86 Linux machine that you have access to.)

Instead of specifying exactly the correct outputs for each command in this spec, we have provided a list of the required commands - and a few example testcases. Run the example executable to find out exactly what your program should print.

3.3 Required Commands

Your program must support all of the following commands:

- `insert <int>`

Inserts a value into the tree. If the insertion worked, then nothing should be printed. However, **duplicates are not allowed in the tree;** if the value already exists in the tree, then you must print out a message to `stdout` (check the example executable for the exact output required).

The value given will always be non-negative, but can be any non-negative value that can be stored in a 32-bit value (that is, from 0 to $2^{31} - 1$, inclusive).

- `search <int>`

  Searches the current tree to see if the value exists. Print out a message if it is found, and a different one if not (use the example executable we provide to see exactly what those messages should be).

  The valid arguments are the same as the `insert` command above.

- `debug <filename>`

  Prints out a `.dot` file, to the filename given. The filename given will be a valid filename, including the `.dot` extension at the end, and will not contain any spaces in the middle. The filename may be up to 128 characters in length.

  The file that you create must contain the specification for a graph, which the `dot` tool can use to generate a picture.

  While you are **required** to create a file with exactly the right name, you are **not** expected that the text inside it will exactly match the example executable. However, when we process this file with the `dot` tool, it should produce a picture which accurately represents the current shape of the BST. The grading script will check that you produced a file, and that `dot` was able to generate a picture; the TAs will check by hand to see that the picture looks correct.

  HINT: Make sure that you close your files after you write them. Do this both because it flushes the data to disk - and also because if you don't close your files, you are using up resources - and eventually the OS won't allow your program to open up any more files.

  NOTE: To prevent overwrites, the example executable will add the string "example_" to the beginning of the filename. But otherwise, your code must produce a file like the example executable.

- `delete <int>`

  Removes a value from the tree, if it exists. If it doesn't exist (or if the tree is empty), prints out an error message.

  NOTE: Make sure that you handle all three delete cases:

  - **Case 1:** The node to delete has no children
  - **Case 1:** The node to delete has one child

- **Case 1:** The node to delete has two children

  In Case 3, make sure that you swap the node-to-delete with its **predecessor.** Swapping with the **successor** works just as well, in general - but if you do that, your output won't match the example executable.

- `count`

  Takes no arguments. Counts the number of elements in the tree, and prints out the answer. See the example executable for the output format.

  Make sure that you test to see what the example executable does when the tree is empty.

- `preOrder`

  Takes no arguments. Prints out a pre-order traversal of the nodes in the tree, all on one line, separated by spaces. If the tree is empty, print out a blank line.

  Note that (for simplicity), you will print out one **extra** space, after the last element, if the tree is **not** empty. See the example executable for the output format.

- `inOrder`

  See `preOrder`; this is identical, except that it prints out an in-order traversal of the tree.

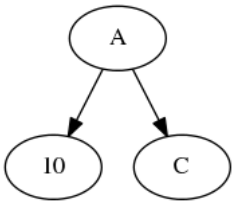
4 The `.dot` File Format

In Computer Science and Mathematics, a "graph" is a structure which contains a set of vertices (sometimes called "nodes") which are connected by edges. A binary tree, for instance, is simply a special type of graph - one where every vertex has (up to) two child edges pointed at other nodes (and a few other requirements).

The `.dot` file format describes a graph as a set of vertices and edges. A very basic dot file looks like this:

```
digraph
{
    A;         // this is a vertex named 'A'
    10;        // this is a vertex named '10'
    C;
    A -> 10;  // this is an edge going from A to 10
    A -> C;
}
```

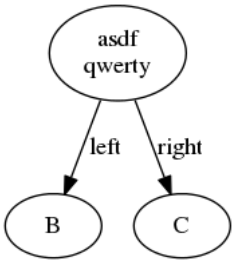
The code above produces the following picture:



`.dot` allows you to pass arguments to each vertex or edge. There are many different arguments allowed; however, the simplest one is `label`, which allows you to assign the printed label to a vertex or edge. Look at the next example:

```
digraph
{
    A [label="asdf\nqwerty"];
    B;
    C;
    A -> B [label="left"];
    A -> C [label="right"];
}
```

This produces the following picture:



In this project, you won't need to change the labels on your nodes (though we might require that in future projects). However, you **will** need to add labels to every one of your edges - you will label each link as either `left` or `right` [2].

4.1 Requirements for Debug Graphs

The `.dot` files that you generate must fulfill the following requirements:

- All of the vertices in the tree must be represented by vertices in the graph.
- The root vertex must be easily identifiable. An easy way to do this is to pass a `penwidth` argument, like this:

```
A [penwidth=4];
```

[2] You have to do this because `dot` doesn't always place the child nodes where you might expect. So you need to know, for each link, whether it is a left or a right child.

However, any obvious marker is allowed. (See below for more ideas.)

- Every edge in the tree must be present in the `.dot` file, and labeled as either "left" or "right." ("L" and "R", or any other easily-readable abbreviation, is acceptable.)

4.2 Optional Features of `.dot`

If you fulfill the requirements from the previous section, that's enough. But here are some pointers about how to make your graphs look much better, like the ones from the Project 1 spec.

4.2.1 `rank=same`

Typically, `.dot` will arrange your nodes pretty well; the root will be at the top, with links to children going further and further down. But occasionally, it may do something strange - particularly if the graph is very complex.

One way to make a binary tree look good is to make sure that each node has both of its children on the same level. To force **any** two nodes to be on the same level, you must tell `dot` to put them on the same "rank." Say that we wanted the nodes `123, 456` on the same row. We would write:

```
{ rank=same; 123 456 }
```

(You can do this any time after you have declared the two nodes - maybe even before, I'm not sure.)

4.2.2 `taildir, headdir`

Next, you can control where the endpoints of the arrow are. Each edge has properties "taildir" and "headdir," which indicate where the tail and head of the arrow should attach. To create an edge going from the right edge of vertex `123`, and going to the left edge of vertex `456`, you would do:

```
123 -> 456 [taildir=e headdir=w];
```

(You use the directions n,s,e,w - north, south, east, and west - to represent the edges of the vertices.)

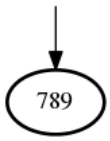
4.2.3 Invisible nodes

The way I like to draw my graphs (to mark the root clearly) is to have an incoming edge, pointing at the root of the tree. To do this, you can pass the argument `style=invis` to a vertex, like this:

```
digraph
{
  root [style=invis];
```

```
    789  [penwidth=2];
    root -> 789;
}
```

which gives a picture like this:



5 Using the `.dot` Files

So why are we bothering with `.dot` files at all? We're doing it because they are a **very** useful tool when you want to debug your code. You probably won't need this tool for the first few, simple testcases - but once you start deleting nodes (or, when you have lots and lots of nodes in the tree), then a visual debugging tool is hard to beat.

(This will be even more true in future projects, where you are changing the structure of the tree!)

5.1 What the Grading Script Does

The grading script runs your code, and compares it to the example executable. Part of this comparison is checking what files are generated by each of your tools. If there was a line

```
debug foo.dot
```

in the testcase, then the example executable will create `example_foo.dot`, and your code should generate `foo.dot` .

After both programs run, the grading script will attempt to convert all of the `.dot` files into `.png` files. Unfortunately, there's no way to check to see if the picture is correct - you have to check that by hand. However, the grading script **will** check to make sure that your `.dot` file worked (no syntax errors) - and of course, that you generated the same files as the example executable.

5.2 Running `dot` by Hand

While the grading script is cool, sometimes you want to run `dot` by yourself - so you can see something that the grading script didn't check, or debug a problem. There are several ways to do this.

To turn a `.dot` file into an image, you use the tool `dot`, which is part of an open-source package known as GraphViz. There are three basic ways that you can do this:

- Go to `http://sandbox.kidstrythisathome.com/erdos/`; you can put in a `.dot` file and have it generate the `.dot` file for you.
- Download GraphViz and install it on your own computer, and run it locally. You can download it from `http://www.graphviz.org`
- Test your program in one of the department computer labs - lectura (and most or all of the other CS machines) have `dot` installed. I've also provided a Makefile to make this easy for you. You can view the images with the `eog` (Eye of Gnome) tool. **This has the advantage that you can build and then see lots of pictures at once.**

5.3 How to Run `dot` Manually

(These instructions work if you have installed GraphViz on your local computer, or if you are running in a computer lab.)

To run `dot` to turn the file `abc123.dot` into `abc123.png`, use the following command:

```
dot -Tpng -o abc123.png abc123.dot
```

(The flag `-T` is for the type of output, and `-o` is for the name of the output file.)

6 Compilation

Your program may be written in Java or C. In both cases, the code must compile and run properly on Lectura or an equivalent CS department machine (`oxford.cs.arizona.edu` is another option).

The grading script will detect whether your code is written in Java or C, and compile and run it appropriately. In both cases, all of the files that you write (including headers, if you use multiple C files) must be in the current working directory when you run the grading script.

6.1 Java

The grading script will compile **all** of the `.java` files in the current directory, using `javac`. If any text is produced by the compilation process, the grading script will terminate and not run your code.

6.2 C

The grading script will compile **all** of the `.c` files in the current directory, linking them in the same step. It will compile with the following command:

```
gcc -Wall -std=gnu99 *.c -o proj02_bst
```

If `gcc` prints out anything, then the grading script will terminate and not run your code.

7 Comments and Style

You must comment your code. Make sure that your comments are clear; they need to express both what you are trying to do, and how you plan to do it.

Use good programming style, including good use of whitespace, consistent indentation, and meaningful variable names. If you write your program in Java, follow the Java variable naming conventions (start with lowercase, and camel case after that). If you write in C, there is more flexibility about style - but use a style that is clear, easy to read, and consistent.

Each file should include a header comment, which includes:

- File name or Java class name
- Our class and assignment name (CSc 345 Spring 17 - Project 2)
- Your name
- A description of the basics of the file or class

8 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

8.1 Testcases

You can find a set of testcases for this project on D2L, with the rest of the files. Each testcase is named `test_*`.

For some projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

8.2 Automatic Testing

We have provided a testing script named `grade_proj02`. Place this script, all of the testcase files, and your solution file(s) in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

Run the grading script - it will tell you which testcases passed and which failed, and will give you a numeric score. (When we grade your code, we may have additional "secret testcases" that you didn't see. So your score may be somewhat different than what the grading script gave you. But it's a reasonable estimate.)

8.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

9 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only the required files (see above); do not turn in any testcases that you've written, or other files.