C212 Semester Project

Adrian German

December 6, 2019

Abstract

I wrote this to make sure you have enough information, and guidance to successfully complete the semester project for C212. Lab instructors are supposed to help you with this during the last two labs. Please let me[1] know if you have any questions or need any help.

1 Introduction

You semester project has three parts: first solve R17.10 on page 814 in your book. Then solve R17.11 on the same page. Finally, write a report in LaTeX(such as this one) in which you describe what you did and how you know it's correct.

1.1 A Guide to LaTeX

Look up a simple template online. Start from it. Look up a short introduction as well. I will provide the source code for this document, should you want to examine it. Then use either the online tools or ask for help in using the software that is available on the shared system.

Try to simplify as much as you can, for example, most simple templates assume you will include graphics in your document. You don't need to, so eliminate them and you will have one less thing to worry about.

2 Stage One

We need to establish a data representation. We then need to implement it. I will give an example of both. Understand though that my examples are meant to give an idea, not to provide you with a solution to your project.

Like in class, I will deliberately work with binary search trees that contain only numbers. You are supposed to be more general, especially now that we have discussed Generic Types (chapter 18 in your text).

[1] dgerman@indiana.edu

2.1 Project Rationale

First let's state our case: I will implement a data type `BST` and will provide a coherent and logically consistent experiment in which I will measure its performance. I will then improve the data type (the new and improved data type will be called `Boom`, which means tree in Dutch) and demonstrate clearly its superior performance as follows:

```
Welcome to DrJava. Working directory is C:\[...]\Project
> run BST 100
For 100 nodes 0 ms is the average search time
> run BST 1000
For 1000 nodes 16 ms is the average search time
> run BST 10000
For 10000 nodes 580 ms is the average search time
> run BST 20000
For 20000 nodes 2226 ms is the average search time
> run BST 30000
For 30000 nodes 5629 ms is the average search time
```

Now pay attention to how the second data type behaves:

```
> run Boom 100
For 100 nodes 56524 ns is the average search time
> run Boom 1000
For 1000 nodes 754144 ns is the average search time
> run Boom 10000
For 10000 nodes 953256 ns is the average search time
> run Boom 100000
For 100000 nodes 7667249 ns is the average search time
> run Boom 1000000
For 1000000 nodes 62291977 ns is the average search time
> run Boom 2000000
For 2000000 nodes 128771494 ns is the average search time
> run Boom 3000000
For 3000000 nodes 204007902 ns is the average search time
>
```

Please note the time is reported in nano seconds in the second batch of tests and in milli seconds in the first. A millisecond is 10^{-3} seconds, while a nanosecond is 10^{-9} seconds so a millisecond is a million times longer than a nanosecond. Clearly, the improved version is a lot more performant than the original one. Now let me show you how I achieved that.

2.2 Getting Started

I start with what I developed in class before Thanksgiving and what was said in class this week. I add to that a function that will allow me to create a perfectly

balanced tree. A binary tree of height h will have at most 2^{h+1} nodes, with the maximum number of nodes being attained when the tree is perfectly balanced[2]. For that I need to insert the numbers from 1 to 2^{h+1} in a certain order. Here's the `BST` data type:

```
public class BST {
  int value;
  BST left, right;
  public BST(int v, BST l, BST r) {
   this.value = v;
   this.left = l;
   this.right = r;
  }
  public int find(int k) {
    if (k == 1 + Utilities.size(this.left)) return this.value;
    else if ( k <= Utilities.size(this.left)) {
      return this.left.find(k);
    } else {
      return this.right.find( k - 1 - Utilities.size(this.left));
    }
  }
}
```

Here's the `Utilities` class:

```
public class Utilities {
  public static int size(BST node) {
    if (node == null) return 0;
    else return 1 + Utilities.size(node.left) + Utilities.size(node.right);
  }
  public static BST fun(int lo, int hi) {
    if (hi < lo) return null;
    int middle = (hi + lo) / 2;
    return new BST(middle,
                   Utilities.fun(lo, middle-1),
                   Utilities.fun(middle+1, hi));
  }
  public static BST create(int lo, int hi) {
    if (hi < lo) return null;
    int middle = (hi + lo) / 2;
    return new BST(middle,
                   Utilities.fun(lo, middle-1),
                   Utilities.fun(middle+1, hi));
  }
} // end of class Utilities
```

[2]This is easily proved by induction, like in C241.

In the interest of keeping things simple so I can include them in a legible format here, I have refactored (reorganized) the code and placed all the `static` methods we developed in class (plus the new one I mentioned) in a separate class of their own. I also eliminated the `insert` method since I don't need it[3] to make my point here. Here's the tester program:

```
public class One {
  public static void main(String[] args) {
    int size = Integer.parseInt( args[0] );
    BST a = Utilities.create(1, size);
    int s = Utilities.size(a);
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < s; i++) {
      int k = a.find(i+1);
    }
    long endTime = System.currentTimeMillis();
    System.out.println("For " + Utilities.size(a) + " nodes " +
      (endTime - startTime) + " ms is the average search time" );
  }
}
```

Here's what I get when I run it:

```
Welcome to DrJava.  Working directory is C:\Users\dgerman\Desktop\project
> java One 100
For 100 nodes 0 ms is the average search time
> java One 1000
For 1000 nodes 0 ms is the average search time
> java One 10000
For 10000 nodes 581 ms is the average search time
> java One 20000
For 20000 nodes 2229 ms is the average search time
> java One 30000
For 30000 nodes 5646 ms is the average search time
>
```

So for 2^{15} nodes we have an average time of more than 6 seconds.

3 Stage Two

I define a new abstraction that is just like the old one except in one significant aspect: this time each node stores the sizes of their respective left and right subtrees. When we need those values we simply look them up, instead of calculating them again and again in a generative recursive approach reminiscent of the naïve implementation of the Fibonacci numbers (see Homework 01).

[3] But you do need to provide it in your code, like I did in class, along with `remove` and the standard `find` so you can actually be able to talk about a binary search tree ADT.

So here's the new and improved data type:

```
public class Boom { // Tree in Dutch
  int value;
  Boom left, right;
  int before, // size of the left subtree
      after;  // size of the right subtree
  public Boom(int value, Boom left, Boom right) {
    this.value = value;
    this.right = right;
    this.left = left;
    this.before = Boom.size(left);
    this.after = Boom.size(right);
  }
  public static int size(Boom boom) {
    if (boom == null) return 0;
    else return 1 + Boom.size(boom.left) + Boom.size(boom.right);
  }
  public static Boom make(int lo, int hi) {
    if (hi < lo) return null;
    int middle = (hi + lo) / 2;
    return new Boom(middle, Boom.make(lo, middle-1), Boom.make(middle+1, hi));
  }
  public String print() {
    return "(" + this.value + " " +
      ((this.left == null) ? "." : this.left.print()) + " " +
      ((this.right == null) ? "." : this.right.print())  + ")";
  }
  public int find(int k) {
    if (k == 1 + this.before) return this.value;
    else if ( k <= this.before) {
      return this.left.find(k);
    } else {
      return this.right.find( k - 1 - this.before);
    }
  }
  public static void main(String[] args) {
    int size = Integer.parseInt( args[0] );
    Boom boom = Boom.make(1, Integer.parseInt(args[0]));
    System.out.println( boom.print() );
  }
}
```

I've put everything in one class this time, for convenience (everything is on one page, above). With the `main` provided we can test, visualize and confirm

that the kind of trees we are producing are exactly the kind we're expecting[4].

Try, for example:

```
Welcome to DrJava.  Working directory is C:\[...]\Project
> java Boom 1
(1 . .)
> java Boom 2
(1 . (2 . .))
> java Boom 3
(2 (1 . .) (3 . .))
> java Boom 4
(2 (1 . .) (3 . (4 . .)))
> java Boom 5
(3 (1 . (2 . .)) (4 . (5 . .)))
> java Boom 6
(3 (1 . (2 . .)) (5 (4 . .) (6 . .)))
> java Boom 7
(4 (2 (1 . .) (3 . .)) (6 (5 . .) (7 . .)))
> java Boom 8
(4 (2 (1 . .) (3 . .)) (6 (5 . .) (7 . (8 . .))))
> java Boom 9
(5 (2 (1 . .) (3 . (4 . .))) (7 (6 . .) (8 . (9 . .))))
>
```

Now here's the tester:

```
public class Two {
  public static void main(String[] args) {
    int size = Integer.parseInt( args[0] );
    Boom boom = Boom.make(1, Integer.parseInt(args[0]));
    int s = Boom.size(boom);
    long startTime = System.nanoTime();
    for (int i = 0; i < s; i++) {
      int k = boom.find(i+1);
    }
    long endTime = System.nanoTime();
    System.out.println("For " + Boom.size(boom) + " nodes " +
      (endTime - startTime) + " nanoseconds is the average search time" );
  }
}
```

If you run the tester above and examine carefully the code we developed you will realize (and notice) that the creation process now is a bit slower than before (although not much slower) since it's creating and initializing the instance variables that R17.11 is mentioning (and is doing that in a quick and dirty,

[4]That is, fully balanced (to the extent possible, since the number of nodes may not always be a power of 2).

perhaps simple-minded way). That's why you should in fact carefully consider (and address) all aspects mentioned in the text of your two problems.

4 Stage Three

The last stage is to write a short paper (shorter than this one) in which you should indicate how you designed your programs and what their relative performance seems to be.

```
Welcome to DrJava.  Working directory is C:\Users\dgerman\Desktop\project
> java Two 10
For 10 nodes 5106 nanoseconds is the average search time
> java Two 100
For 100 nodes 58347 nanoseconds is the average search time
> java Two 1000
For 1000 nodes 173219 nanoseconds is the average search time
> java Two 10000
For 10000 nodes 1287295 nanoseconds is the average search time
> java Two 100000
For 100000 nodes 6619179 nanoseconds is the average search time
> java Two 1000000
For 1000000 nodes 59931448 nanoseconds is the average search time
> java Two 3000000
For 3000000 nodes 191647311 nanoseconds is the average search time
> java Two 10000000
For 10000000 nodes 671806732 nanoseconds is the average search time
>
```

We note that for 10 million nodes the average search time is a little more than half a second. In contrast the first method has an average search time of 6 seconds (so 12 times longer) for a tree that is $\frac{1}{300}$ smaller.

5 Conclusion

You can find the source code for this file here[5].

[5] `http://silo.cs.indiana.edu:8346/c212/fall2019/project/`