# Assignment 2 README

Here's a description of the second programming assignment for CSE 100, Fall 2013. In this assignment, you will subclass your binary search tree class template from the previous assignment in order to implement a randomized search tree, and write a benchmarking program that experimentally investigates its performance.

**>>> Due Fri Oct 25 8:00 PM**

**>>> Checkpoint submission Mon Oct 21 8:00 PM**

**>>> Required solution files: BST.hpp BSTNode.hpp BSTIterator.hpp RST.hpp benchtree.cpp benchtree.pdf**

This means that the file that contains your solution to the assignment must be turned in by the due deadline. The turnin procedure is described below in the section"Turning in your assignment".

It's a good idea to plan to do the assignment well before the deadlines; terminals and tutors can get very busy at the last minute.

In doing this assignment, it's your responsibility to understand the course rules for integrity of scholarship.

The files that contain your solution to the assignment must be turned in by the due deadline.

You may develop your code anywhere, but you must ensure that it compiles and runs correctly on ieng6 (the lab machines), because that's where we'll be testing it.

## 1. Getting started

Read Aragon and Seidel's paper on randomized search trees, (available online in PDF: http://people.ischool.berkeley.edu/~aragon/pubs/rst96.pdf) and Drozdek section 6.10 on treaps. This assignment uses those concepts. You may find lecture notes and discussion sections helpful as well.

In this assignment, you will be extending a class template you wrote for the previous assignment. So for this assignment to work correctly, the BST, BSTNode, and BSTIterator class templates must be working correctly; fix them first if not.

If you worked with a partner on PA1 assignment, you can (and should) work with the same partner on this one, or you can work alone.

In case you mentioned that you will be working with a partner on PA1 but ended up working alone, then you are allowed to (but don't necessarily have to) pair up with the person that you chose initially. In this case, you can use either person's code from PA1 to use for this assignment.

**Github Set-up**

Your repositories for PA2 will contain the files Makefile, countint.hpp, countint.cpp and test_RST.cpp. The test_RST.cpp files is a partial test program for you test the correctness of your RST implementation. However, it is in no way a complete or exhaustive testbench which is why you should write your own test cases in order to ensure that your implementation is correct. We will certainly test your code very thoroughly. Your PA2 repository on github will not contain the files from PA1 initially. After you pull the four starter files provided to you in your repository, you will need to copy your solution files for PA1 and paste them into your working directory for PA2. You will then need to add those files to the staging area and commit/push them so that they are available on github.

For commands or instructions on how to use git, refer to PA1 README.

## 2. Defining the RST class template: RST.hpp

In this assignment, you will subclass the BST class template of the previous assignment to define a class template named RST, which implements a randomized search tree. In a file RST.hpp, you can start with

```
#ifndef RST_HPP
#define RST_HPP
#include "BST.hpp"

template <typename Data>
class RST : public BST<Data> {

public:

 virtual bool insert(const Data& item) {
    // TODO: implement this function!
  }

};
#endif // RST_HPP
```

(Note that the `insert` function is declared 'virtual' here, and in the parent class. This means that it can be overriden, and dynamic, polymorphic, virtual function

binding can happen at runtime, even when accessing a child class object via a parent class type pointer.)

Many improvements on ordinary BSTs, such as AVL trees, red-black trees, or randomized search trees, can be thought of as different kinds of BSTs; here we are explicitly setting up that "is-a" relationship with public class template derivation: the class template RST inherits from the existing BST class template.

Further, these more advanced kinds of BSTs usually require some additional information to be stored in each node; for example, randomized search trees require a random *priority*, which can be represented as an int value. For this assignment, you will need to change the BSTNode.hpp file from PA1 to include an integer data member (called `info` or `priority`) in the BSTNode class, which is basically the priority field required by the RST for inserting elements into the tree. Be sure that your constructor for your BSTNode does not require this field, as we do not want to break your BST implementation from last week. [As an aside, it is probably better design to subclass the BSTNode to create an RSTNode, but because this addition is so small we'll save you the overhead of creating a whole new class and just add the field directly to our BSTNode class. It won't hurt the functionality of the BST anyway.]

The basic idea of the insert operation for an RST is to first insert a new BSTNode using the usual BST insert algorithm, assign to it a randomly generated priority, and then to rotate that node up, as needed, to restore the heap ordering property governing the priorities in the tree. Random numbers can be generated in C++ using the `rand()` function from the `<stdlib.h>` header (C's standard library). Take a look at this article for some examples on how to call it.

**** CHECKPOINT Submission (worth 5 of the 25 possible points on this assignment)****

In order to get you started early, we require that you submit the RST.hpp with the implemented `insert` method, along with the BST.hpp, BSTIterator.hpp and BSTNode.hpp files to your repository in github by **Monday, October 21 at 8:00 PM**. This means that you will need to PUSH all of these files to github before the deadline with the commit message "CHECKPOINT SUBMISSION". We will grade these submissions for correctness of the `insert` method for the RST class template.

Also note that this checkpoint is less than half way and represents the BARE MINIMUM you should have completed by Monday. You should strive to be well beyond this point before the checkpoint deadline (i.e., strive to submit your checkpoint submission early). We will only grade up to this point at the checkpoint, but if you're done early with the part that you need to submit before the checkpoint deadline, you should move on and try to complete the rest of the project as soon as possible.

Finally, make sure that if you do go beyond the basic checkpoint requirement by the checkpoint deadline, that your checkpoint submission compiles and runs the RST portion of the assignment correctly. That is, when you tag your push with the commit message "CHECKPOINT SUBMISSION" it should not include any code that might break the RST implementation or cause the code not to run or compile. We will be testing your checkpoint submission directly.

## 3. Benchmarking: benchtree.cpp, benchtree.pdf

Red-black trees (which are used to implement the C++ STL `std::set` structure) provide guaranteed O(log N) worst case performance for add, remove, and contains operations, but the add and remove operations are challenging to code correctly. Aragon and Seidel claim that randomized search trees are approximately as efficient as highly optimized red-black trees (which are much better in the worst case than ordinary binary search trees), while being much easier to implement. Here, you will investigate these claims experimentally.

Write a C++ program `benchtree` that permits benchmarking the performance of a `RST` implementation, an ordinary `BST` implementation, and the C++ STL `std::set` structure. Interpret "performance" to mean "how many comparisons were required to do a successful `find` operation, in the average case". For this you may find it convenient to create your containers to hold instances of the countint class (provided to you in your github repository), which can count comparisons done on them.

Your `benchtree` must take four command line arguments:

1. the first is a string which must be either "bst", "rst", or "set", and indicates whether benchmarking should be done using a BST, a RST, or std::set.
2. the second is a string which must be either "sorted" or "shuffled", and indicates whether the keys inserted in the structure are inserted in sorted order, or shuffled (randomized) order.
3. the third is an integer which specifes the maximum size of the tree to be built in the benchmarking. Trees should be build with sizes N that are a power of 2, minus 1, in the range 1 up through this maximum size. This permits comparing results to the 'best possible' binary tree with size N, a completely filled tree, which will always have a size that is a power of 2, minus 1.
4. the last is an integer that specifies how many runs will be done, and averaged over, for each N.

It should then collect statistics on successful finds for the appropriate kind of structure over that range of sizes `N`.

You will get good, interpretable results here if you have `N`, the number of keys in the structure, take on values that are a power of 2, minus 1, in a range from 1 up through the maximum specified. For each `N`, run the number of trials specified. In each trial, create a vector of `N` distinct keys, for example by

```
std::vector<countint> v;
v.clear();
for(int i=0; i<N; ++i) {
  v.push_back(i);
}
```

Note the vector filled that way holds keys in sorted order. If it is required to insert keys in shuffled order, randomize the vector:

```
std::random_shuffle(v.begin(),v.end());
```

And insert the keys in the vector in an empty structure of the appropriate kind:

```
 std::vector<countint>::iterator vit = v.begin();
 std::vector<countint>::iterator ven = v.end();

 for(vit = v.begin(); vit != ven; ++vit) {
    s.insert(*vit);
 }
```

Now, find how many comparisons it takes, on average, to successfully find a key in that structure, averaging over all the possible N successful finds:

```
countint::clearcount();
for(vit = v.begin(); vit != ven; ++vit) {
  s.find(*vit);
}
double avgcomps = countint::getcount()/(double)N;
```

Note that `avgcomps` here is just the average number of comparisons per successful find in a structure of size `N`, for one particular run. Since randomness may be involved (if the keys are randomly shuffled, and/or the structure is a randomized one), it will be useful to also average over several runs (the number of runs to average over is specified on the command line). Then it will also be useful to compute the standard deviation of the measurements taken over those runs.

In statistics, an often used measurement of the dispersion of a set of observations is their empirical *standard deviation,* which is the square root of the empirical *variance*.

Here's a way to compute the standard deviation: You are doing R runs for a particular N. For each run *i*, compute the average number of comparisons for a successful find as shown above; call this $t_i$.

Now Adding up all those R times, and dividing by R, will give the overall average number of comparisons for that N:

$$\hat{t} = \frac{1}{R}\sum_{i=1}^{R} t_i$$

But also sum the *squares* of the average number of comparisons for each run; then dividing that sum by R will give the overall average squared number of comparisons for that N:

$$\hat{s} = \frac{1}{R}\sum_{i=1}^{R} t_i^2$$

Now take that average squared number of comparisons, minus the square of the overall average number of comparisons; That is the empirical variance for that N. Take the square root of the variance to get the standard deviation:

$$\sigma = \sqrt{\hat{s} - \hat{t}^2}$$

A small standard deviation means that the various observed averages in the R runs for a give N were relatively close to each other, and so are probably a reliable indication of what to expect in practice; a large standard deviation means the observations were spread out, and so the simple overall average is not so reliable. (Wikipedia has an article on the topic which explains more about it.)

Your benchtree program should print its results to standard output (that is, using `cout`). It should print some comments, each line starting with `#`, indicating parameters being used. Then should follow lines of data. Each line should have 3 numbers on it, separated by a tab character: the value of N, the overall average number of comparisons for a successful find with that size N, and the standard deviation of the average number of comparisons, computed as given above. For example,

```
 ./benchtree rst sorted 32768 5
```

might print to the screen data like

```
# Benchmarking average number of comparisons for successful find
# Data structure: rst
# Data: sorted
# N is powers of 2, minus 1, from 1 to 32768
# Averaging over 5 runs for each N
#
# N       avgcomps        stdev
     1          2            0
     3    3.33333     0.365148
     7    4.77143     0.714857
    15       6.08     0.243676
    31     8.2129      0.44726
    63     10.873       1.8052
   127    12.8882     0.790504
   255    14.6525     0.901901
   511    16.5119     0.623989
  1023    19.5752      1.28953
  2047    21.4011     0.735492
  4095    22.0275      0.75499
  8191    24.7844      1.07525
 16383    26.9748      1.29925
 32767    28.8381     0.546904
```

To save the data to a file named, say `rst.sort` you could redirect standard output:

```
 ./benchtree rst sorted 32768 5 > rst.sort
```

To visualize the number of comparisons, as a function of N, you will want to use a data plotting program. `gnuplot` is a good, free program available on the lab machines. To use `gnuplot`, start the program, and tell it to plot data with lines and errorbars:

```
$ gnuplot
Terminal type set to 'x11'
gnuplot> set style data yerrorlines
```

Now ask gnuplot to plot some of the data files you created, for example:

```
gnuplot>  plot "rst.sort","set.sort","bst.sort"
```

Look at the data and see if it agrees perfectly with the theoretical big-O time cost analysis. If not, can you suggest a reason why it disagrees?

If the time cost function being plotted is proportional to log(N), using logscale on the x axis will show a straight line:

```
gnuplot> set logscale x 2
```

If the time cost is a linear function of N, using logscale x will show exponential growth. Alternatively, you can use linear x scaling:

```
gnuplot> unset logscale x
```

and now linear time cost growth will show a straight line.

In a file benchtree.pdf, include a page or two summarizing and analyzing the data your benchtree program generated. (You can create your document using Microsoft Word or Open Office or any word processing program that you like, and then export it in .pdf format.) Does the data agree with the theoretical big-O time cost analysis? If there are places where the data diverges from the theoretical, can you suggest a reason why it disagrees?

Include some plots of your data in your benchtree.pdf document. Both Microsoft Word and the Open Office word processing program (available on the lab machines) can import images in Enhanced Metafile format (.emf). To get gnuplot to save a plot in a file `plot1.emf`, first plot it to the screen as discussed above, then do:

```
gnuplot> set term emf
gnuplot> set output "plot1.emf"
gnuplot> replot
gnuplot> set term pop
```

Images included in your benchtree.pdf file will be viewed on a computer monitor, so they don't need to be high-resolution. The total size of your turned-in files is limited. If your benchtree.pdf file is too large to turn in, try using lower-resolution or smaller images for your plots.

**Detecting memory leaks**

Memory management bugs are a potential problem in C++ programs, and it can be hard to track them down. Fortunately, tools exist that can help with this.

Two tools available in the ieng6 environment, `purify` and `valgrind`, can automatically analyze a compiled C++ program for memory leaks and other bugs such as double deletes, unitialized variables, etc. `purify` is (arguably) more powerful, while `valgrind` is (arguably) easier to use. Manual pages and tutorials for these tools exist online. Here is just one easy way to use `valgrind`. Suppose you have a compiled C++ executable program `foo` in the current working directory. Then running

```
valgrind ./foo
```

will run `foo`, printing the results of the checks `valgrind` did. If `foo` was compiled using the `-g` flag to `g++`, `valgrind` will be able to print the line numbers of statements that are involved in bugs that it has found. If want to see more detailed information about memory leaks, run

```
valgrind --leak-check=full ./foo
```

The output from `valgrind`'s analysis can be lengthy, and especially if the program `foo` is also printing to the terminal, it can be hard to read. Running it as

```
valgrind --log-file=out ./foo
```

will write `valgrind`'s output to the file `out` instead of to the terminal screen, and you can then read the file in an editor, or search in it with `grep`.

**Turning in your assignment**

When you have completed the checkpoint requirements or the whole assignment, you must submit it. You will do this by pushing your changes to Github. Follow these steps (which are more or less the same as the steps from the "pushing changes to Github" section above:

1. Run a '`git status`' to see what files have been changed
2. Use '`git add <files>`' to stage certain files for a commit, use '`git add .` ' to add all files in the working directory to the commit (hopefully you will not be doing this at this late stage--all files should already be in your repository!)
3. Use '`git commit -m "FINAL SUBMISSION"` ' for your final submission (or '`git commit -m "CHECKPOINT SUBMISSION"` ' for checkpoint submission) to commit the changes to your repository and indicate that you are done and this is your final submission.
4. To push your changes onto your repository on Github run '`git push origin master`' Don't forget this push! You must do this to get credit.
5. As an added safety check, you should log on to Github on the web and verify that your push went through.

A couple of notes about this process:

- If you are submitting with a partner, just push to the joint repository on behalf of both partners. That's your joint submission.
- Use the commit message "FINAL SUBMISSION" for your final submission (or "CHECKPOINT SUBMISSION" for checkpoint submission) in ONLY ONE of your individual and partner repositories. You shouldn't be using both anyway, but if you happen to have pushed changes to both, just make sure only one contains the "FINAL SUBMISSION" (or "CHECKPOINT SUBMISSION") commit message.
- If you decide you want to resubmit your assignment before the deadline, just do another commit with the message "FINAL SUBMISSION" (or "CHECKPOINT SUBMISION") (and don't forget to push to Github!!). We will grade the last commit before the deadline (or the slip day deadline if you use slip day(s)) that is tagged as the "FINAL SUBMISSION" (or "CHECKPOINT SUBMISSION").
- If you submit the assignment within 24 hours after the deadline you will be charged a slip day. If it is more than 24 but within 48 hours, you will be charged 2 slip days. If you are out of slip days, or after 48 hours, we'll roll back to the last commit tagged as final that was submitted before the deadline.

**Grading**

There are 25 possible points on the assignment. If your solution files do not compile and link error-free, you will recieve 0 points for this assignment.

```
10 points for your RST implementation. 5 of these points will only be awarded based on the checkpoint submission. The remaining 5
earned based on your final submission.
10 points for your benchtree.cpp benchmark program
5 points for your analysis writeup in benchtree.pdf
```

## Comments

You do not have permission to add comments.