Balanced Binary Search Tree Key Lookup Performance

Jake Pitkin

Introduction— Balanced binary search trees store keys in a sorted tree of nodes. This allows binary search to be performed when searching the tree for a key. Binary search uses a divide-and-conquer approach, every comparison allows half of the remaining tree to be ignored when searching for the desired key. This grants balanced binary search trees logarithmic properties when searching for keys. To explore this an experiment was conducted using Java's TreeSets, which are implemented using balanced binary search trees.

Setup— Eleven TreeSets of integers ranging in size from 2^{10} to 2^{20}, where each step is a power of 2, are constructed and populated with randomly-generated keys. Integers in the range [0, 10,000,000) are generated one at a time and if a TreeSet doesn't already contain that integer, it's added to the TreeSet as well as to an ArrayList of valid keys for that specific TreeSet. The case of searching a TreeSet for a key it doesn't contain will be ignored. By randomly-generating the contents of the TreeSets, an average case for a balanced binary search tree can be constructed.

Timing Considerations— Good timing practices were used in an attempt to avoid skewed data.

Avoiding cold starts– To allow for libraries to load, the JIT compiler to run and the cache to populate before timing code considerations setup code was ran. The setup code populates the eleven TreeSets with the random integer values. In additon though, this code takes a little over two seconds to run, giving the Java environment time to warm up.

Timing intervals of at least one second– The goal is to determine the average performance of searching a TreeSet. To avoid data anomalies, each TreeSet was searched millions of times and the average was taken. This allowed for large timing intervals of at least one second.

```
Keys: 1024      Contains calls: 10240000       Milliseconds: 1933
Keys: 2048      Contains calls: 10240000       Milliseconds: 969
Keys: 4096      Contains calls: 10240000       Milliseconds: 1082
Keys: 8192      Contains calls: 10240000       Milliseconds: 1376
Keys: 16384     Contains calls: 10240000       Milliseconds: 1554
Keys: 32768     Contains calls: 10223616       Milliseconds: 1734
Keys: 65536     Contains calls: 10223616       Milliseconds: 2416
Keys: 131072    Contains calls: 10223616       Milliseconds: 3422
Keys: 262144    Contains calls: 10223616       Milliseconds: 4570
Keys: 524288    Contains calls: 13107200       Milliseconds: 7657
Keys: 1048576   Contains calls: 10485760       Milliseconds: 6817
```

Figure 1: Experiment Run Time and Call Count

Taking averages of multiple runs– As seen in Figure 1, each TreeSet was searched about ten million times. This helps collect accurate data that is representative of the average run case.

```
startTime = System.currentTimeMillis();
// repeat 'iterations' times
for (int i = 0; i < iterations; i++) {
    // iterate through each key and look it up in the TreeSet
    for (int j = 0; j < keys.size(); j++) {
        int currentKey = keys.get(j);
        tree.contains(currentKey);
    }
}
endTime = System.currentTimeMillis();
// total time for 'iterations * keys.size' lookups with overhead
totalTime = endTime - startTime;
```

Figure 2: Repetitively Searching a TreeSet

Accounting for timing overhead– The code in Figure 2 that isn't searching the TreeSet needs to be negated out. To accomlish this, the same code was ran without the line that searches the TreeSet.

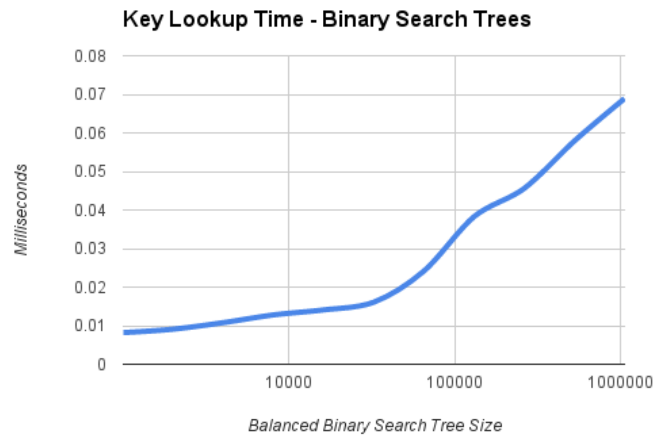
```
startTime = System.currentTimeMillis();
// repeat 'iterations' times without the lookup to calculate overhead
for (int i = 0; i < iterations; i++) {
    // iterate through each key and look it up in the TreeSet
    for (int j = 0; j < keys.size(); j++) {
        int currentKey = keys.get(j);
    }
}
endTime = System.currentTimeMillis();
// time for the overhead for 'iterations * keys.size'
overheadTime = endTime - startTime;
```

Figure 3: Overhead Code

Interference from computation heavy applications– To ensure that no other applications were using major CPU load, all other easily closable applications were closed. The Activity Monitor was consulted and the system use was at 3% before the experiment was ran.

Results— The results of the experiment were telling of balanced binary search trees lookup having logarithmic properties.

Tree Size	Lookup Time	Time Gain
1024	0.000082	
2048	0.000091	+0.000009
4096	0.000108	+0.000017
8192	0.000128	+0.000020
16384	0.000141	+0.000013
32768	0.000161	+0.000020
65536	0.000242	+0.000081
131072	0.000383	+0.000141
262144	0.000456	+0.000073
524288	0.000579	+0.000123
1048576	0.000689	+0.000092



The TreeSet demonstrates logarithmic behavior when looking up a key it contains. As seen in the table, when the size of the TreeSet is doubled the runtime of the experiment increases by a constant amount. This constant growth in runtime, when doubling the TreeSet size, indicates it has logarithmic properties on lookup. What was suprising at first is the jump the performace time takes when the TreeSet is doubled in size to 65,536 keys.

I believe this is due to the role the cache plays on the performance of the TreeSet. The TreeSet can best take advantage of the cache for TreeSets up to 32,768 keys. After that, the cache gets thrashed by new keys and causes performance issues. This can best be seen in the graph, as the constant gain the runtime sees each time the TreeSet is doubled goes up. This isn't to say it stops demonstrating logarithmic behavior on lookup, but rather the constant at which the runtime grows just increases.