**Chained Hash Table:**

Average C. Hash operation run times:

Seq. ADD: 1446 milliseconds

Seq. FIND: 1352 milliseconds

Seq. REMOVE: 872 milliseconds

Random ADD: 934 milliseconds

Random FIND: 0 milliseconds

Random REMOVE: 2170 milliseconds

The strength of a Chained Hash Table lies in its hash function, which makes it very quick to find items distributed random, although apparently it is fairly weak at other operations when storing 50,000 integers.

As seen above, this particular implementation of a chained hash table was (relatively) slow compared to the other data structures. I believe this is because of the hash buckets being ArrayStacks instead of a DLL. Each time the bucket grows, it has to be copied over into an array twice as large (as seen in the textbook code), even though items are being added on to the end every time. This leads to a definite decrease in performance when adding and removing. Surprisingly, sequential find also takes a long time despite the ArrayStack’s O(1) find operation complexity.

**Linear Hash Table:**

Average L. Hash operation run times:

Seq. ADD: 3 milliseconds

Seq. FIND: 0 milliseconds

Seq. REMOVE: 3 milliseconds

Random ADD: 4 milliseconds

Random FIND: 0 milliseconds

Random REMOVE: 0 milliseconds

In my tests, a linear hash table was the fastest data structure, and for good reason. Integers are small and not very memory-intensive so copying over thousands of 4 byte integers can still be faster than copying over one single large object. Because linear hash tables are essentially faster arrays, they have all the advantages of O(1) retrieval and lookup (lending speed to find and remove operations) while adding the performance of hashing to improve insertion speed. Sequential finding and random finding/removing were so fast the clock could not represent its performance in milliseconds, which is quite impressive.

The only weakness of linear hash tables comes in when larger objects are being stored, as mentioned above. Copying data into an expanded array is still expensive, especially when these objects must be rehashed according to the resize() method. As it stands right now, add() uses linear probing to find an empty spot, which is the reason both sequential and random adding take the longest to complete.

**Binary Search Tree:**

Average B.S. Tree operation run times:

Seq. ADD: 1391 milliseconds

Seq. FIND: 1308 milliseconds

Seq. REMOVE: 6 milliseconds

Random ADD: 19 milliseconds

Random FIND: 0 milliseconds

Random REMOVE: 2 milliseconds

Binary Search Tree was the third-fastest performing data structure, behind Linear Hash Tables and Red/Black Trees. Randomly generated BSTrees were quicker to add, find and remove items than a comparable red/black tree because of simpler policies (only 2 child nodes, left < root < right) which make the balanced random tree’s operations perform in O(log n) time.

However, to achieve those faster times, a BSTree has to be balanced, which is never a guarantee. Random adding can help create a tree that is very close to balanced, but there will always be some room for error. Sequential adding, thanks to the binary nature of the data structure, effectively creates a doubly-linked list which drastically slows down performance. “DLList” performs only a few milliseconds faster than Chained Hash Table in this case.

**Red/Black Tree**

Average Red/Black Tree operation run times:

Seq. ADD: 13 milliseconds

Seq. FIND: 6 milliseconds

Seq. REMOVE: 12 milliseconds

Random ADD: 20 milliseconds

Random FIND: 3 milliseconds

Random REMOVE: 4 milliseconds

Finally, Red/Black trees have the 2nd fastest performance, losing out to Linear Hash Table. The reason for its almost identical performance between sequential and random operations is the tree’s self-balancing “fixup” methods which are called after every addition or removal. Both rotateLeft()/Right() work to switch the root node for a candidate closer to the median value so the tree doesn’t begin to be skewed in any particular direction. When compared to BSTree, a Red/Black tree performs on par but often better in real-world applications where the ability to balance itself ensures O(2 log n) complexity.

There are few downsides to using R/BTrees is that they are tougher to maintain compared to other data structures. Both hash tables use a hashing function as a “policy” to organize stored items, and BSTree has two rules all items must follow. In contrast, a Red/Black Tree must obey 5 separate limitations, all of which can slow down performance. Also, a Red/Black tree still cannot keep up with the O(1) runtime of linear hash tables, although O(2 log n) is still quite good.

**2-4 Trees and Red/Black Trees**

Red-black trees are the binary representation of 2-4 trees. Each red node with two black nodes is the “extra” part of a 2-4 tree. A branch with a black parent and 3 child nodes would turn into a black RBNode parent with one black child and a red child (having two children of its own), effectively simulating a parent with 3 child nodes in a binary fashion.

A call to pushBlack(u) simulates an addition of 3rd or 4th nodes to a 2-4 parent node, and a call to pullBlack(u) simulates their removal (since red nodes signify its children should be included with the red parent).