Part 3

Chained hash tables have many unique advantages. They are more memory efficient than linear hash tables for large data type sizes. They technically never need to be resized (although it is a good idea to do so). And, as the book says, “ChainedHashTable supports the operations add(x), remove(x), and find(x) in O(1) expected time per operation” (pg. 114). The efficiency here is shown by the average run time of 9 milliseconds on my random remove method.

Despite its advantages, chained hash tables are by no means flawless. They run very slow, as shown by their average completion time of over 1500 milliseconds for both the sequential add and sequential find functions. They are also harder to implement when compared to linear hash tables, due to using two kinds of data structures. Also, “the performance of a hash table depends critically on the choice of the hash function,” (pg. 109) so it is possible to that a bad hash function could cripple the complexity and run time.

Linear hash tables are another data structure that has many draws to them. Most obviously, they are much faster than chained hash tables, with its highest average run time being 6 milliseconds in my program. They are much easier to implement that chained hash tables, due to only using one kind of data structures. Ideally they will only have a “O(1) expected time per operation” (pg. 121).

Linear hash tables also have their fair share of problems. When it comes to using large data types, chained hash tables are much more useful. Linear hash table are much less memory efficient than chained hash tables. This is because linear hash tables “maintain the invariant that t.length ≥ 2q,”(pg. 115) which means that non-null elements can never be more than half the length of the hash table, so they grow to huge sizes very quickly.

Binary search tree are a data type that offer many advantages to the user. They are useful because “the binary search tree property is extremely useful because it allows us to quickly locate a value” (pg. 140). To back this claim up, the average time on my random find method for this data type was 45 milliseconds. The simple logic of the search tree (bigger objects to the right, smaller ones to the left) also make the binary search tree very easy to work with.

Binary search trees unfortunately also offer many disadvantages to the user. Binary search trees can quickly become unbalanced, due to a bad root or to only adding sequential numbers. As a result, it can “have O(log n) time operations”(pg. 146). This is shown by a runtime of 9224 milliseconds for sequential add.

Red-black trees give the users many advantages in regards to other data types. Red-black trees are self-balancing, so “A red-black tree storing n values has height at most 2 log n.” They are also very quick, operating “in O(log n) worst-case time”. This is backed up by my average run time of 13 milliseconds being the highest runtime for red-black trees.

Red-black trees also come with many drawbacks. “The nice properties of red-black trees come with a price: implementation complexity” (pg. 185). This means that red-black trees are no walk in the park to code. If the user isn’t going to doing a lot of work with the red-black tree, it is almost always worth it to just make something simpler.

Part 4

These operations change how many “child” nodes a parent node will have. For example, Pushblack(u) will take a parent with 4 “children” node and will break it up into two separate parent nodes each with two of the original “children”. Meanwhile, Pullblack(u) will do the opposite. It will take 2 parent nodes and combine them so that all of their “children point to the one parent node.