jnguye36
Lab 2, Hashing Schemes

The implementation I've provided relies heavily on both standard integer arrays, and the Java library ArrayList object to contain the resulting hash table, a reference table, a stack to keep track of free space, and as a way to maintain input handling. Using an array or ArrayList object was a simple choice because of how hashing works. The random nature of the hash functions lends itself to the random access capability presented here as opposed to a linked list, which would require us to follow through the list in order to find the correct location. This also made the chaining portion easier, by having a reference table in parallel to the hash table. It would be possible to create an object that contains both an a value for the hash and a pointer, but I decided on this parallel table approach to make it simpler and more flexible for the other two collision methods which did not rely on the reference table.

When designing this implementation, I broke out four categories of functions that would be broken down further. This lab and hashing in general lent itself well to modularity, so I had three different functions for each probing method, and two functions for the two different probing methods. The output of the hash function could be placed directly into the probing functions because of this. This implementation will run through the dataset in linear time. The hash functions have a constant cost, but the probing cost can vary depending on free space. An empty table would have constant time, but in the worst case, the full table must be checked, giving a O(n*size of table). Bucketing adds a small cost, but will be equal to O(bucket size * n). In short, efficiency is dependent on the probing method. These functions were fairly short and standard, but I struggled with bucketing. I wanted buckets to be a parameter, so that I could use the same functions for all cases regardless of bucket size. This provides the advantage of being able to be used for future cases with different bucket and table sizes, but implementation was difficult for me. Because I wanted to use a standard array to maintain a table, I had to keep track of bucket size for multiple functions, and ensure I was acting at the level of the bucket before moving down to the level of the table. If I were to do this lab again, I would consider creating a new object that could handle both chaining and bucket, or create structures for each specific hashing scheme. Another change might be to address how chaining works. This implementation has it picking free space in descending order, whereas a random slot would work better towards reducing collisions.

As stated earlier, efficiency is tied to the probing algorithm, which in this implementation can be measured as the number of collisions. In small datasets there is very little difference in terms of collisions, as the load factor is so low. The given input for the lab faces at most 1 collision through all hashing schemes. Bumping up the entries to 30, and a load factor of 0.25, brings out differences between the schemes. Using the first three schemes as a base case, the number of collisions ranges between 5-7, which means it increased at least 5x when the input was tripled. This shows evidence of collisions becoming much more frequent as input grows. As expected, changing the divisor to a prime number, and adding buckets decreased the number of collisions to about half the amount. An unexpected change was the multiplying method from CLRS performing significantly better than the division method. This method of hashing depends on the constant A given, and I coincidentally chose a value close to what CLRS suggests.

Although not implemented here, hashing would need to address deletion. Deletion of items is significant for determining a stopping point during searches. For examples, a deleted link in the chaining method may make it so that algorithm stops searching, and loses its pointer to where to follow the chain. Our lecture material suggests putting a flag to indicate deletion, and thus keep the pointer while removing the value itself.

A bioinformatics example is given in our lecture by Dr. Rogozin, which explains how nucleotide searches in BLAST take advantage of hash tables, which when combined with scoring matrices are used to quickly search for the highest scoring match.

Overall, I learned the importance of finding a good load factor as well as different schemes to better minimize the number of collisions.