Hashtable

Sorting

The hashtable I created is really just an array of Linked Lists. Given the parameters of the assignment, any integers inserted will go into the Linked List at the index equal to the last digit of the integer being added. This takes the problem of insertion sort being not particularly efficient, and breaks it into 10 separate insertion sorts.

```
    for (int i = 0; i < 25; i++) {
        int newNumber = generateRandomNumber();
        hashtable[newNumber % 10].newNode(newNumber);
    }
```

The sorting on the Hashtable is simple and elegant, but the internals of the sorting within the hashtable get a little messier, being the previously addressed insertion sort:

```
void LinkedList::insert(Node* newNode) {
    //inserts nodes in ascending order, sorted by data value
    if (Head == nullptr) {
        Head = newNode;
    }
    else if (*(Head) > *(newNode)) {
        newNode->setNext(Head);
        Head = newNode;
    }
    else if (Head->getNext() == nullptr) {
        Head->setNext(newNode);
    }
    else {
        Node* next = Head;
        while (*(next->getNext()) < *(newNode)) {
            next = next->getNext();
            if (next->getNext() == nullptr) {
                next->setNext(newNode);
                return;
            }
        }
        newNode->setNext(next->getNext());
        next->setNext(newNode);
    }
}
```

The result is that in the absolute best case, there will be a completely even distribution of integers across each of the lists, *and* the data will be reverse-sorted as it reaches each list, requiring just one operation per insertion. This will be able to sort the list in $\Omega(n)$ time. In the worst case, all data will come in pre-sorted, and have the same output when run through the `newNum % 10` filter, appending to the same list each time. This will give a worst-case complexity of $O(n^2)$. On average, the results will be spread across the lists at least somewhat evenly, and will typically require the same average complexity as the insertion sort itself. This gives a complexity of

$\Theta(nlog_{10}(n))$. This is faster than the more typical $nlog_2(n)$, but does not actually approach complexity of 1.

- $O(\sum_{i=0}^{n} n(n-i)+5i) \sim O(n^2)$
- $\Omega(n)$
- $\Theta(\sum_{i=0}^{log_{10}(n)} \frac{n(n-i)}{2}) \sim \Theta(nlog_{10}(n))$

Thus the sorting of this hashtable is better than a linked list, typically reaching $nlog_{10}(n)$ time complexity.

Searching

My Hashtable

Searching is where hashtables are nearly always faster, even with a quick and dirty implementation with a high collision rate. In order to find a number sorted into our hashtable, we need to determine which list it would fall into within the hashtable, and then find it in the list at that index. Given an input to search for, this is as simple as:

```
Node* found = hashtable[input % 10].findByData(input);
```

Which will call the `findByData` method of that sublist, as seen here:

```
Node* LinkedList::findByData(int data) {
    //linearly searches for a given node by Data value
    Node* currentNode = Head;
    while (currentNode != nullptr) {
        if (currentNode->getData() == data) {
            return currentNode;
        }
        else {
            currentNode = currentNode->getNext();
        }
    }
    throw std::invalid_argument("No item found with data " +
std::to_string(data));
}
```

This is still a linear search, which we have previously analyzed. But the results are strongly modified by the use of the hashtable to break this into a series of 10 separate linear searches.

As with the sorting, the worst case is that all data was filtered into a single sublist of the hashtable, in which case we are simply running a linear search on a single linked list. As previously discussed in Lab 1, this is of complexity $O(n)$. However, the best case search time for the hashtable is when the data is evenly spread over all 10 sublists. This gives us a linear search over a much smaller portion of the data, similar to breaking the data in half repeatedly for a binary search, resulting in a best case complexity of $\Omega(log_{10}(n))$. The average case will still likely have the data broken across all 10 lists, but not perfectly. Still, being broken into 10 pieces has the most dramatic effect, leaving us with an average time complexity of $\Theta(log_{10}(n))$.

In a table of no more than 10 items, this could be equivalent to $O(1)$ time complexity; but even with as few as 25, we have guaranteed collisions, and our average search time will be closer to 2 operations each. And from the asymptotic view, will continue to grow, albeit quite slowly.

An Optimized Hashtable

Sorting

Given a more-perfectly made hashtable, where there will be 0 collisions on the data, there is no need to sort at all. The data is simply entered into the hashtable at the appropriate location. Entering each new item in the hashtable will be of time complexity $O(1)$; generating a hashtable from an input of n items will take $O(n)$ time as each item is inserted into its place.

Searching

Finding each item in this collision free hashtable is likewise as simple as accessing the data. There would be a key based on the entered data item, corresponding uniquely to the desired item. There is no need to iterate at all, and the item is found in $O(1)$ time.