**Note: These are essentially just linked lists, and have been implemented as such**

Make-HeapA()

\[ X \leftarrow \text{head}[L] \]
\[ \text{// run insertion sort on the list} \]

InsertA()

\[ X \leftarrow \text{next}[\text{nil}[L]] \]
\[ \text{While } x \neq \text{nel}[L] \text{ and key}[x] \neq k \]
\[ \text{Do } x \leftarrow \text{next}[x] \]
\[ \text{// insert } x \text{ here} \]

MinimumA()

\[ X \leftarrow \text{head}[L] \]
\[ \text{Return } x; \]

Extract-MinA()

\[ X \leftarrow \text{head}[L] \]
\[ \text{List-Delete}(L, x); \]
\[ \text{Return } x; \]

UnionA()

\[ X1 \leftarrow \text{head}[L] \]
\[ X2 \leftarrow \text{head}[L2] \]
\[ \text{// loop through both lists, comparing the values and inserting the lesser value into} \]
\[ \text{// newList until one of the lists is empty, then append remaining elements in non-empty} \]
\[ \text{// list} \]
\[ L = \text{newList}; \]

Make-HeapB()

\[ \text{// do nothing; list is unsorted and has no property to maintain} \]
InsertB()
// search list for repeats. If none exist, insert element at head as follows
next[x] <- head[L]
if head[L] != Nil
    then prev[head[L]] <- x
head[L] <- x
    prev[x] <- Nil
}

MinimumB()
Min = Max_Num
// traverse list in order, comparing values and updating min
return min
}

Extract-MinB()
Min = MinimumB
// traverse list until match is found, then delete
}

UnionB()
// for each element of the other list, sequentially search this list to see if the element
// already exists. If not, add to the head of this list. If so, quite searching and move on
// to the next element
}

Make-HeapC()
// same as Make-HeapB()
}

InsertC()
// same as InsertB()
}

MinimumC()
// same as MinimumC()
}

Extract-MinC()
// same as Extract-MinB()
}

UnionC()
// tack the list to be merged onto the end of this list; ordering doesn’t matter and we
// know that there are no repeats to check for
Analysis:

a. Lists are sorted.

Make-HeapA – \( \theta(n^2) \) since each of the \( n \) elements must be compared to, on average, \( n/2 \) other elements before their ordered position is found.

InsertA – on average, \( n/2 \) elements will have to be checked before the correct spot for insertion is found, and then the insertion will happen in constant time. The running time is \( \theta(n) \).

MinimumA - since the minimum is, by definition, the first element in the list, the happens in \( O(1) \) time.

Extract-MinA – the element to be deleted is the first element. The running time is the time to return and delete the element, which is \( O(1) \).

UnionA – assuming \( n \) is the length of this list and \( m \) is the length of the list to be merged, the running time is \( \theta(n + m) \) since there is at most one comparison for each element.

b. Lists are unsorted.

Make-HeapB – no time; nothing needs to be done

InsertB – \( \theta(n) \) because the list has to be searched for repeats before an element can be inserted. Insertion is \( O(1) \) since the element can simply be inserted at the head

MinimumB – \( \theta(n) \) because the entire list must be searched to find the minimum

Extract-MinB – \( \theta(n) \) because the entire list must be searched to find the minimum and then the extraction happens in \( O(1) \) time.

UnionB – assuming \( n \) is the length of this list and \( m \) is the length of the list to be merged, the running time is \( \theta(m * n) \) since there entire list must be searched for repeats for each element in the second list.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

Make-HeapC - same as above

InsertC - same as above
MinimumC - same as above

Extract-MinC - same as above

UnionC – depends on how long it takes to append the lists together, but most likely O(1) since no searching needs to be done.

11.2-1

Step 1: Find the probability that the slot that an element gets hashed to is empty.

The probability that an element hashes to a particular slot out of m slots is 1/m. The probability that an element does not hash to a particular slot is 1 – 1/m. The probability that the i – 1 elements before a given element were not hashed to a particular slot (i.e. the slot is empty) is (1 – 1/m)^(i – 1) since all the individual probabilities of the previous elements must be multiplied together. Therefore, the probability that the slot an element is hashed to is empty is (1 – 1/m)^(i – 1).

Step 2: Find the probability that the slot that an element gets hashed to is not empty (i.e. there is a collision).

This is equal to 1 – Pr(slot is empty) = 1 - (1 – 1/m)^(i – 1)

Step 3: Sum the above result over all n keys to compute the total number of collisions.

Sum(i = 1 to n) { 1 - (1 – 1/m)^(i – 1) }
= n – Sum(i = 0 to n – 1) { ((m – 1)/m)^i }
= n – (((m – 1)/m)^n – 1) / (((m – 1)/m) – 1)

11.2-3

Chaining in sorted order

Successful searches
The average running time for a successful search will be unaffected. On average, half of the linked list (n/2) will be traversed before the matching element is found, independent of the ordering of the list. Since linked lists cannot be randomly accessed, there is no way to get away from doing a sequential search to get to an item that exists. Therefore the running time remains theta(1 + alpha).
Unsuccessful searches
The running time for an unsuccessful search will be cut in half since once value is found in the list that is greater than the value we’re looking for, we know that all values further down the list are also greater than the value, therefore there is not match and the search need not continue. This allows the search to be short-circuited. Whereas with an unsorted list, all n elements had to be compared, with sorting only n/2 elements need to be compared on average. The asymptotic behavior isn’t changed, but the constant is. The asymptotic running time is still theta(1 + alpha).

Insertions
The running time of insertions would be increased from O(1) to theta(n) since with sorted chaining the linked list must be traversed until the appropriate spot for the new element to be inserted, whereas with unsorted chaining the new element can simply be inserted at the front of list without concern for ordering.

Deletions
The running time for the deletion of an existing element is unchanged and depends on whether the lists are double or singly linked. If they are doubly linked, the running time is still O(1), if they are singly linked then the lists must be searched. Assuming a successful search, the deletion would have essentially the same running time as the successful search described above.

11.3-1
Since the strings are of variable length and are known to be long, we would likely be much more efficient to first hash a given key to get its hash value h(k), then search through the linked list looking for hash values that match. If the hash values match, then character strings are compared to make sure that the element is actually the same element and not a different element that happens to have the same hash value. This alleviates the need to do string comparisons for every comparison. This should improve efficiency since on many architectures string comparisons require each 8-bit character to be compared in sequence, whereas as numerical comparisons only need to compare two typically 32-bit numerical values. Thus number comparisons should have constant time, whereas string comparisons may not. This efficiency boost will likely be enough to offset the extra time to calculate the hash value, which should happen in constant time.

Graphs:
Analysis:

The two hash functions that I used are the division and multiplication functions as explained in the book. I used the division function because, since I expect pseudo-random values when I run hashcode on Objects in java, I can expect that by moding the hashcode value by the number of array "slots" available, I will end up with key values that are fairly evenly distributed within the range of available slots. Division using a modulus seems to be the most common and straightforward method of hashing a value to a given range, so it seemed like a logical choice.

In comparing the two hash functions, the variables that I chose to examine were 1) Number of entries hashed, 2) Load Factor, 3) Number of “gets” performed on the given entries. I set up my experiments with the assumption that the worse hash function of the two would result in longer, sparser linked lists in the hash table. One way to examine how even or uneven the distribution of elements is for a given hash function is to perform a series of “gets” on a hash table. The better hash function should get values more quickly on average since the linked lists should be shorter. I wasn’t sure if the maximum load factor would have an impact, but I chose to try a load factor of 1 and 10 just in case. I chose to set the number of gets performed to be 100 and 1000. Consequently, I got four graphs.

I chose to print out all of the values that I used get(Object key) to retrieve both so that I could make sure I was getting expected values and so that the time was large enough for
me to measure. Without printing the values, the time to perform a series of gets was often 0, which wasn't helpful to my analysis. I assumed that printing the values would only change the constant and not the asymptotic behavior of the algorithm.

I found that the time to get n elements did not generally increase as the number of hashed elements increased. This is because the load factor was maintained so that even though the table and the number of entries grew, the linked lists did not. One thing I would change if I were to rerun this data would be to time several sets of gets before the table is resized so that the load factor would vary instead of being constant. I don’t believe that this affected the comparison of the two hash functions, but it may have given more insight into the running time of the get function at different points before the table is doubled.

I expected that the hashing by division would be slower because, in my implementation, all the values of m were powers of 2 beginning with $2^{10} = 1024$. Consequently the hash function was only depending on the p (where p is the power of 2) lowest-order bits of the key to be hashed. By that logic, the second hash function should have been better because it operates independently of the size of the hash table. However, in my data I found very little difference in the running time using the two different hash functions. The second hash function appears to have resulted in slightly lower running times on average, but the difference is not significant. I assume then that the values returned by hashcode() had low-order bits that were sufficiently random to result in a fairly even distribution even when using the division method.