HW04 Exercises – Hashmaps

Prasun Surana
CS5008, Summer 1

1. The stringHash function in our program takes the string length and takes the remainder after it is divid2ed by the number of buckets in the hashmap. The remainder is then the bucket number, so if the remainder for a particular string is 2, it is put into the bucket 2. The problem with this is that this type of hash function creates a lot of collisions, or several keys that may end up in the same bucket. For a normal array, the access time is O(1) and the search time is O(n). However, if we have many collisions, i.e. several buckets that have linked lists attached to them (since there are several keys that produce the same remainder), the time complexity would increase since we not only have to search each element in the array, but also down the associated linked lists. This would be especially cumbersome if the node we were looking forward was closer to the end of the linked lists, meaning that we would have to iterate through more nodes.

   This would not be helped even if we increased the number of buckets. Most words in the English language are probably under 20 characters long. Even if we have 120 buckets in an attempt to avoid many nodes being in the same bucket, we would end up with linked lists in the first 20 buckets, and less frequent collisions thereafter, but also a lot of wasted space once the string lengths get longer and longer, since there are probably not many words which are 50 letters long.

   If we examine the program we wrote, for many of the functions that were intended to be written in constant time, they involved looping through the linked list for a particular bucket. For example, the hashmap_hasKey function had to find the right bucket, then check if the first node contained the key that we were looking for. If it did not, we had to loop down the linked list for that bucket until we found the correct key, which meant a worst-case time complexity of O(n) instead of constant time. Similarly, for the hashmap_delete function, as well as the hashmap_printKeys function, we had a while loop nested within a for loop, creating a time complexity of $O(n^2)$ when we were aiming to do it in O(n). Hence, we can say that the stringHash function is not optimal in terms of minimizing time complexity due to numerous collisions.

2. Instead of using the modulo operator on the string lengths to sort the keys into buckets, we could instead write our program to use the ASCII numbers for the string entered into the function as a parameter. Each character has an associated ASCII number. If the key entered was ‘fish’, we could take the individual ASCII characters of all 4 letters. To avoid getting collisions resulting from permutations of the same word (e.g. ‘sifh’), we could multiply each character’s ASCII by a constant raised to a progressively higher power. For instance:

   $102 * 5^1 + 105 * 5^2 + 115 * 5^3 + 104 * 5^4$

   This would give a largely unique bucket to each key. This would result in a huge number of buckets, but would reduce the number of collisions.

3. The time complexity for resizing is O(n). If we have an array with n elements in it, we cannot just take each element and mod it on the number of buckets again, since for the new resized array, the number of buckets will be different, so the element may end up in a completely different bucket. So for each element, we have to get it's key and value and store it in the appropriate place in the new array, so the time complexity is O(n).

4. Open addressing, like using linked lists, is a way of resolving collisions. Open addressing involves having at most one element in each bucket, so at all times, the size of the array has to be greater than the number of elements, or keys. When a new key is added, the program, instead of chaining it to a bucket via a linked list, will simply look for an open position in the array itself and place the key in that position. The three types of open addressing methods are linear probing, quadratic probing and double hashing.