CSC263: Problem Set 4

Angus Fung,[1] Syed Kamran,[1] Michelle Leung[1]

[1]Division of Engineering Science, University of Toronto,

Question 1

(a) Assuming two hash tables H_1 and H_2 with the same hash function, of array size m, and with n_1 and n_2 keys-value pairs stored, respectively, then an algorithm (pseudocode) of merging H_2 into H_1 is the following:

```
def mergehash(hash_table1,hash_table2):

    currNode = None
    currList = None
    for i in range(m): #H1.size=H2.size=m
        currList = hash_table2[i] #pointer to the ith linked list in
            H2
        currNode = hash_table1[i]
        while currNode != None:
            currNode = currNode.next
        currNode = currList #adding the linked list to the end of the
            corresponding linked list in H1
```

This algorithm inserts all the (key, value) pairs in H_2 into H_1 by passing the entire linked list at each array element to the end of the linked list of the corresponding index in H_1. Since they both share the same hash function, then all (key, value) pairs that are mapped to the kth index in H_2 must also be mapped into the kth index in H_1. This algorithm, therefore, is independent of n_2.

(b) To prove an upper bound on the worst-case running time, we define the run-time as the number of array and node accesses. The first for loop iterates through each index in the arrays `hash_table1` and `hash_table2`. Since the iteration always goes to the end of the arrays the outer for loop runs for at most m iterations. The inner while loop iterates over all the keys at the particular index of the array. Over the entirety of the m indices of the arrays, the inner while loop executes at most n_1 times. Therefore the worse case run time complexity of the algorithm is $\mathcal{O}(n_1 + m)$.

(c)
```
def mergehash(hash_table1,hash_table2):

    currNode = None
    for i in range(m): #H2.size=m
        currNode = hash_table2[i] #pointer to the ith linked list in
           H2
        while currNode != None:
            InsertH1(hash_table1, currNode.key, currNode.value)
            currNode = currNode.next
        #adding the linked list to the end of the corresponding
           linked list in H1

#From course notes
def InsertH1(table, key, value):
    hash = h(key)
    linked_list = table[hash]
    insert key, value at the head of linked_list
```

(d) The above algorithm is similar to the algorithm described in part a) so again we define the run-time as the number of array and node accesses. The outer for loop runs for at most m iterations (the size of the `hash_table2` array. The inner while loop is executed for all the nodes in the linked list at a specific index of the hash table, and therefore for the entirety of the outer loop the inner loop runs at most n_2 times (the number of keys in `hash_table2`). The insert function taken from the course notes executes in constant time. Therefore we can say that the upper bound on the worst-case running time of our algorithm is $\mathcal{O}(n_2 + m)$ and is not dependent on n_1

Question 2

(a) We begin by considering the probability of a single element being inserted into the array. The first element will always be consecutive. The probabilities for the second and third element being inserted consecutively are as follows:

$$P_1 = 1$$
$$P_2 = (1) \cdot \frac{3}{m}$$
$$P_3 = (1) \cdot \frac{3}{m} \cdot \frac{4}{m} = \frac{12}{m^2}$$

We can justify this by assuming one key has already been inserted into the hash table at some location i, and in order for the next key to be consecutive, it needs to be inserted either before $(i-1)$ or after $(i+1)$, but by the linear probing sequence, if the key is inserted into index i where the first key resides, resulting in a collision, then the key will be hashed to the next available spot at $i+1$. Hence, there are three ways to hash the second key into an array of m size, hence a probability of $\frac{3}{m}$ for inserting the second key. With the third key, a similar situation is true with the exception that we assume two consecutive keys already exist, and we want to hash the third key either before, after, or at the location of the two keys (so that linear probing will hash it to the location after the two consecutive keys), hence giving the third key 4 out of m spaces to be hashed into in order to retain the consecutive key sequence.

(b) Given that n keys have already been inserted into the hash table, the probability that the next key k will be inserted into an unoccupied spot on the first try will be $\frac{m-n}{m}$ since there will be $m-n$ empty spots in the array. Let there be a random variable A that represents the number of array locations visited by the function when key k is inserted.Then $A = 1$ if k is inserted into an unoccupied array location, since the key will not need to be moved elsewhere. If k is inserted into the last location of the already-inserted consecutive keys, then k will be hashed into the next spot which will be empty, resulting in $A = 2$. If k is inserted into the location of the first of the consecutive keys, then $A = n+1$ since k will continually be hashed into the next spot, which will be occupied, until it gets to the first unoccupied spot, accessing all n keys along the way. The probability that k is inserted into an occupied spot is $\frac{1}{n}$, which applies to all cases from $A = 2$ to $A = n+1$ hence, we can write the average number of array spots visited as the expectation value of the number of array spots accessed, which is the summation of the probability of inserting k into a particular location multiplied by the number of array accesses (A) corresponding to that location, as follows:

$$E[A] = \frac{m-n}{m} \cdot 1 + \frac{n}{m} \cdot \sum_{a=2}^{a=n+1} a$$

(c) Let there be a random variable A that represents the number of array locations visited by the function when key k is inserted. Since the existing keys in the array are all surrounded by empty spots, each time k is inserted into an empty location, $A = 1$, and each time it is inserted into an occupied location and is hashed over into the next unoccupied location, $A = 2$. Choosing an occupied array location has a probability of $\frac{n}{m}$, while choosing an unoccupied spot has a probability of $\frac{m-n}{m}$. Hence, similar to the previous question, the average number of array spots visited can be represented by the expectation value of the number of array spots accessed, hence:

$$E[A] = \frac{m-n}{m} \cdot 1 + \frac{n}{m} \cdot 2$$