Reid Roberts
Feb 9, 2025

Assignment 3 - Hashing Answers:

1.
   - $h_1(k) = k \ mod \ 13$
   - $h^t(k) = 1 + (k \ mod \ 11)$
   - $h(k, i) = (h_1(k) + i * h_2(k) \ mod \ m$
   - Table Results:

Slot	Item
0	A
1	I
2	N
3	Q
4	E1
5	S1
6	T
7	U
8	S2
9	E2
10	-
11	Y
12	O

Collision Table				
Item	h1(k)	ht(k)	h(k, i)	i
E1	4	(not needed)	4	0
A	0	(not needed)	0	0
S1	5	(not needed)	5	0
Y	11	(not needed)	11	0
Q	3	(not needed)	3	0
U	7	(not needed)	7	0
E2	4	5	9	1
S2	5	8	8	2
T	6	9	6	0
I	8	9	1	5
O	1	4	12	6
N	0	3	2	5

2. The time complexity for insertion, delete, and search value operations in a hash table **assuming standard independent uniform hashing and a doubly linked list** are as follows:

Worst-Case Time Complexity		
	Unsorted Linked Lists	**Sorted Link Lists**
Successful Search	$O(1 + \alpha)$	$O(1 + \alpha)$
Unsuccessful Search	$O(1 + \alpha)$	$O(1 + \alpha)$
Insert	$O(1)$	$O(1 + \alpha)$
Delete	$O(1)$	$O(1)$

$$\alpha = \frac{n}{m}$$

By maintaining a sorted linked list in each of the hash table slots, the worst-case time complexity is slightly worse than in an unordered LL. Linked lists are not random-access data structures like arrays so we can't use a more efficient algorithm such as binary search to speed up the search time. Additionally, by maintaining a sorted order it requires the algorithm to potentially update the entire LL upon inserting a new element since it is no longer sufficient to just insert the new element at the beginning of the list and leave it as-is.

It is possible that sorting the linked list *slightly* speeds up the search function in the average-case, but those differences would be marginal, and the worst-case complexity remains $O(1 + \alpha)$. Since sorting the LL makes the time complexity worse for insertions and doesn't provide any significant benefits, this change would not be recommended

Reid Roberts
Feb 9, 2025

3. Using the following strategies, and given a hash table with 7 entries, draw the hash table and its contents after inserting the following numbers: 7, 11, 71, 42, 13, 49. Calculate the load factor α for each of the strategies:

Based on the instructions I am only showing the hash tables with table size of 7. I am not rehashing based on the load factor. I only rehashed on the quadratic probing method because it led to a failed hash. If we were to rehash based on load factor I would rehash each of the other tables when the load factor reaches around 0.6 - 0.7.

a) Separate Chaining
   a. This method results in 3 empty slots and a linked list with 3 elements at index 0. T**he load factor for chaining is 6/7 or ~0.86.**

Entry/Slot	Item
0	7 --> 42 --> 49
1	71
2	-
3	-
4	11
5	-
6	13

b) Linear Probing
   a. This method results in 6 occupied slots. **The load factor is 6/7 or ~0.86.**

Entry/Slot	Item
0	7
1	71
2	42
3	49
4	11
5	-
6	13

c) Quadratic Probing
   a. This method gets interesting because the quadratic probing equation leads to a continuous loop when attempting to assign X to an empty slot. Therefore, the table needs to be rehashed by expanding the table size. The method I chose is to select the nearest prime number greater than double the initial table size, which turns out to be 17. After resizing the table, the hash values were recalculated for every element using the larger table size value. **The final resulting load factor is 6/17 or 0.35.**

Entry/Slot	Item

0	-
1	-
2	-
3	71
4	-
5	-
6	-
7	7
8	42
9	-
10	-
11	11
12	-
13	13
14	-
15	49
16	-

d) Double Hashing

- For this method, we need to specify the secondary hash function to use when collisions occur. **I will use $h^t(k) = 1 + (k \bmod 5)$.** I chose 5 as the secondary mod value because it is a prime number smaller than the initial table size. The resulting table using this secondary hash function is shown below. **The load factor is 6/7 or ~0.86.**

Entry/Slot	Item
0	7
1	71
2	-
3	42
4	11
5	49
6	13