FIT1008 – Intro to Computer Science
Solutions for Tutorial 10

Semester 1, 2018

Exercise 1

The following table maps each key to its associated hash value (E has Ascii value 69, A 65, T 84, R 82, J 74, K 75, and D 68):

Name	Eva	Amy	Tim	Ron	Jan	Kim	Dot	Ann	Jim	Jon
Hash Value	6	2	0	19	11	12	5	2	11	11

Given that mapping, the resulting Hash Table with linear probing is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tim		Amy	Ann		Dot	Eva					Jan	Kim	Jim	Jon					Ron	

where Eva, Amy, Tim, Ron, Jan, and Kim, and Dot are inserted without collisions and, therefore, they appear in the same position as their hash value. In other words, for each of those keys, the cell in the array position returned by the hash function was empty. For the other three, the situation is different. When inserting Ann, there is a collision (Amy is already in the cell 2) and, thus, a linear probe (2+1, +1, +1, etc) is used until the next empty cell (which is 3) is found and Ann inserted there. Jim also results in a collision (Jan is already in cell 11), and is inserted in the next empty cell (which is found at 1 after going from 11 to 12, 13). Finally, Jon also gets a collision (Jan is already at cell 11) and is inserted in the next empty cell, which is 14 (after going from 11 to 12, 13, 14).

Exercise 2

When searching for an element in a hash table using linear probing, it will initially use the hash to determine the starting position in our hash table. It will keep going until one of these three conditions is meet:

Condtion 1: The keys match.
Condtion 2: We encounted an empty space.
Condtion 3: We have searched every element in the Hash Table.

In the case of Linear Probing, it will keep adding one to the position (wrapping around to the start) until one of the above conditions are met. If the keys match it signifies we have found the element in our hash table, otherwise it means we haven't found the item.

Jim: Initial position is 8 and then visits positions 11, 12 and 13 (Condtion 1).
Jon: Initial position is 8 and then visits position 11, 12, 13 and 14 (Condtion 1).
Joe: Initial position is 8 and then visits positions 11, 12, 13, 14 and 15 (Condtion 2).

Exercise 3

Inserting Eva, Amy, Tim, Ron, Jan, Kim, Dot are same as above.

Collision handling = (N + S*S) % TABLE_SIZE

Insert Ann: N = 2
= 2 (collision)
= (2 + 1) % 21 = 3

Insert Jim: N = 11
= 11 (collision)
= (11 + 1) % 21 = 12
= (11 + 4) % 21 = 15

Insert Jon: N = 11
= 11 (collision)
= (11 + 1) % 21 = 12
= (11 + 4) % 21 = 15
= (11 + 9) % 21 = 20

Given that mapping from Exercise 1, the resulting Hash Table with quadratic probing is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tim		Amy	Ann		Dot	Eva					Jan	Kim			Jim				Ron	Jon

Exercise 4

The following table maps each key to its associated second hash (`hash2`) value:

Name	Eva	Amy	Tim	Ron	Jan	Kim	Dot	Ann	Jim	Jon
Second Hash Value	10	6	5	3	5	6	9	6	5	5

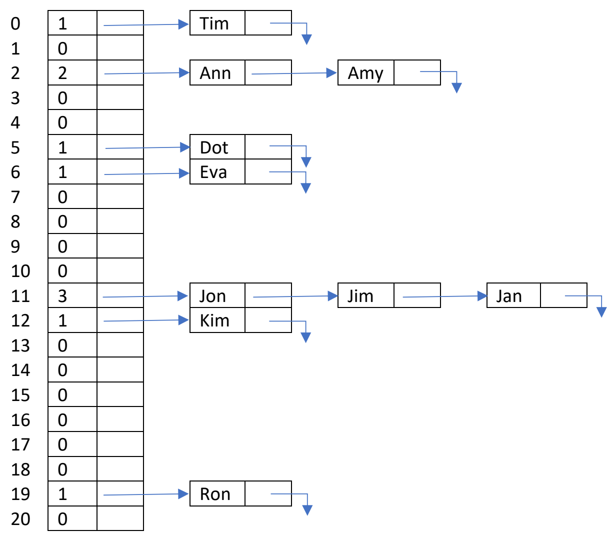
Given that mapping, the resulting Hash Table with double hashing is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tim		**Amy**			**Dot**	**Eva**		Ann		Jon	**Jan**	**Kim**				Jim			**Ron**	

where Eva, Amy, Tim, Ron, Jan, Kim, and Dot are inserted – as before – without collisions (and thus, they appear in exactly the same position as before). When inserting Ann, there is again a collision (as before, Amy is already in the cell 2) and, thus, Ann uses as step the value of the second hash function - 6 – probing first cell 8 (2+6) which is empty. Jim also results in a collision (Jan is already in cell 11), and thus it uses as a step the value of the second hash function – 5 – going from 11 to probing 16 which is empty. Finally, Jon also gets a collision and uses as step the value of the second hash function - 5 – going from 11 to 16, 0, 5, and 10 which is empty.
Note that what the second hash gives you isn't an index, but an offset – you don't go to `hash2(item.key)`, you go to `(hash1(item.key) + hash2(item.key)) % tablesize`, where the "modulus tablesize" is there so that if you fall off the end of the table, you'll wrap around to the top (the wrapping around mentioned before). Using double hashing means that you can handle multiple collisions easily: you just keep adding `hash2(item.key)` to the index until you've either found an empty spot or established that there aren't any.
You can also think of it as follows: given a hash function that returns value N for key K, all open address methods look first in N, and then in

- Linear Probing: N+1, then N+2, then N+3, then N+4, etc (N+S)
- Quadratic Probing: N+1, then N+4, then N+9, then N+16, etc ($N+S^2$)
- Double Hashing: N+h, then N+2*h, then N+3*h, then N+4*h, etc, (N+S*h) where h is the value returned by the second hash function

When using separate chaining, a possible resulting hash table is as follows:



Note that we have used unsorted lists and inserted the elements into the list using `addFirst`. Keep in mind, this does not prevent duplicates in our hash table. One could also have used a sorted list and inserted in order (therefore preventing duplicates). Dividing the array into two (the counter and the lists) is not needed, I have just done it to easily show the number of elements in each list.

Exercise 5

A hash function is considered a universal hash function if there's a chance it can return the same position given different keys. Due to the nature of randomness generated by the hash value there's a chance we could hash to the same position.
A perfect hash function guarantees that every key maps to a unique position, with no collisions. The main advantage here is that we have constant time insertion and retrieval and no need of collision resolution.

Exercise 6

Some possible **advantages:**

- Conceptually simpler

- Insertions and deletions are easy and quick
- Naturally resizable, allows a varying number of records to be stored

Some possible **disadvantages:**

- Requires extra space for the links
- Requires linear search for elements in a list