FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 11

Semester 1, 2019

Objectives of this tutorial

- To understand Hash Tables.

Exercise 1 *

Using the following hash function:

```
def hash(input_string):
        return ord(input_string[0]) % 11
```

and linear probing, calculate the hash value of the following names and insert them into a Hash Table of size 11.

```
Eva, Amy, Tim, Ron, Jan, Kim, Dot, Ann, Jim, Jon
```

Note that the ascii value for E is 69, for A 65, for T 84, for R 82, for J 74, for K 75, and for D 68.

Solution

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 *

Assume you have completed Exercise 1. Illustrate what happens, when you search for the names Jim, Jon and Joe.

Solution

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 *

Repeat Exercises 1 and 2 using Quadratic probing instead of linear probing.

Solution

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 *

Using the following function:

```
def hash2(input_string):
        return ord(input_string[0]) % 10 + 1
```

as the second hash function, repeat Exercise 1 and 2 using double hashing instead of linear probing.

Is the second hash function a good choice of function? Discuss in terms of the values provided for keys that are mapped to the same value by the first hash function.

Solution

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**	

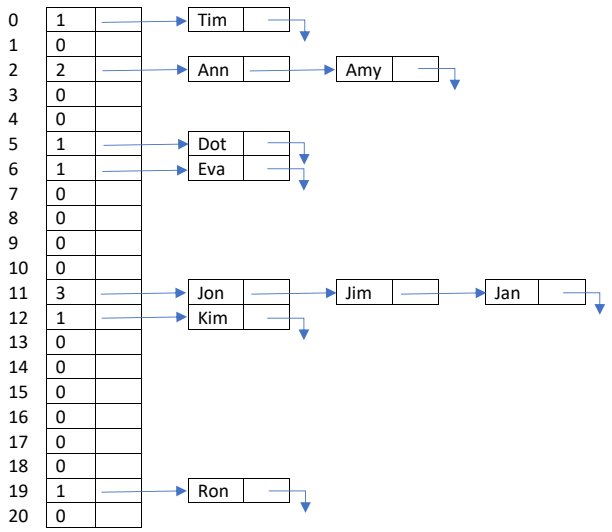
where Eva, Amy, Tim, Ron, Jan, Kim, and Dot are inserted – as before – without collisions (and thus, they appear in exactly the same position as before). When inserting Ann, there is again a collision (as before, Amy is already in the cell 2) and, thus, Ann uses as step the value of the second hash function - 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

One of the methods to delete an element at position N from a Hash Table that uses linear probing is to re-insert every element from N+1, N+2, N+3, etc, until an empty cell is found. Why is this necessary?

Explain by giving an example in which if we don't do this, things go wrong.

Solution

Reinserting every element from N+1, N+2, N+3, etc, until an empty cell is found is necessary because the search method relies on the following invariant: if the hash value of key K is position N, and we successfully insert K in the hash table, then K must appear in some cell between N and the first empty cell (wrapping to 0 when running out of bounds). Lets assume that K appears at position N+3, the first empty cell appears at N+5, and we erase an element at position N+1. If we simply leave N+1 empty, then we have broken the invariant and, therefore, if we now search for K, the search will start at position N, not find K there, look at N+1, found an empty position and fail.

For example, consider the hash table in Exercise 1 above: the hash value of key Jon is 9, and Jon appears in position 5, before the first empty cell 7. Let us assume we now erase Amy, which is in position 0, by simply creating an empty cell. Then, if we look for Jim (or Jon or Ann), the search will say the key does not appear in the Hash Table, when it does. What we need to to is to reinsert all elements from Jim to Ron included.

IMPORTANT: note that re-inserting does not mean simple shuffling up. Shuffling up does not work: in the above example you would be moving Dot, Eva, and Ron to the wrong places (e.g., Dot was hashed to 3 and you would be shuffling it to 2). Re-inserting means marking the position as empty and calling the insert method (thus the hash function etc) again.

Exercise 6

We said that, when performing double hashing, it is important for the second hash function not to return 0. Explain why. How does the above `hash2` function achieve this?

Solution

When performing double hashing the second hash function is used to compute the offset h used while searching either for a key, or for a place to insert the key (we first try N, then N+h, then N+2h, then, N+3h, etc, so N+S*h as mentioned in Exercise 2). Thus, if the second hash function returns 0, we never move from N. Do we stop? It looks like an infinite loop, but we will actually stop if our loop takes into account the size of the table (still wrong though).

The above `hash2` function avoids this by adding +1 to the result of the mod (%) function (in case the mod returns 0). Note that this is correct because the remainder will never return a negative number.