Defne Betül Çiftci

21802635

CS 202 Section 01

Assignment 4

Question 2.1 – Explaining the Implementation

To implement the HashTable class, I needed to use an integer array implementation. To implement the delete function properly, i.e. to make the keys have Empty, Deleted, or Occupied states, I created a multidimensional array in which the first column of each row would correspond to the integer value of that row and the second column of the row would be either -1, 0, or 1 depending on the state of the row (-1 is for empty, 0 is for deleted, and 1 is for occupied.) I initialized the first columns as 0 on construction and upon deletion, the value reverts back to 0 again. I assumed that the HashTable object would never be full on my implementation.

These are the variables I used:

```
    int** items;
    int size; //this is initialized upon construction, not changed
later on
    int currItems; //incremented when item is added, decremented
when an item is deleted
    CollisionStrategy collStr; //initialized upon construction, not
changed later on
```

I used a multidimensional array for this reason: I needed to either use a multidimensional array or a new TableContent class to keep track of whether a node was deleted or was empty from initialization. This is important for search function because if we assume a scenario like this: table size is 13, deleted index is 9, and we want to search for number 22 (which corresponds to 22%13=9 from the first hash function). If we do not skip the deleted index as "not-empty" in a situation like this, it would return 'not found' for 22 when, in fact, it could have been located elsewhere by the probe function. My implementation for the multidimensional array goes like this:

```
      Number column     State column

Rows  #############     # for whatever state it is in
```

So, the implementation is a tableSize-row 2-column multidimensional int array implementation. As for the state integers, I put -1 for 'Empty' nodes (all elements are initialized as 1 for the second column in the constructor), 0 for 'Deleted' nodes, and 1 for 'Occupied' nodes.

I implemented a probe function for which this prototype applies:

```
const int HashTable::probe( int& address, const int item, const int
searchFor, bool& found, const bool insertion);
```

The function initializes a temporary integer i to count the number of probes (initializes as 1) and starts by checking for the probe enumeration with if statements. The integer parameter searchFor is used to substitute for whatever value we are looking for*.

*In the cases where we look for an empty/deleted node (in analyze function's unsuccessful searches where we look for an empty node and in insert where we look for an empty/deleted node), this is how I overcame the problem:

Upon initialization, the first column of every index is 0 and when deleted, the first column is, again, substituted as 0. When I looked for specifically an empty node, I resorted to the search function and did not call the probe function as search function already returns false upon probing through the whole array of integers and stumbling upon an empty node. Inside the insert function, however, I called the probe function instead of the search function. For the probe function's searchFor parameter, I substituted 0, as both deleted and empty nodes are 0 in their first columns of their respective indexes.

Pseudocode goes like this:

```
Initialize probation count as 1

initialize boolean found as true
initialize Boolean passForZero as equal to ( searchFor is zero AND
insertion parameter is false)
if( collision strategy of class is linear/quadratic)
    while passForZero is true OR the first column of the array isn't
equal to the parameter searchFor OR (the item in address is zero AND
it is 'Occupied')
        implement the probation method and update address accordingly
        increment probation count
        end loop prematurely when an ('Empty' node is reached AND
passForZero is false) OR when probation count is greater than table
size, set boolean found to false
return probation count
```

For double hashing, because there isn't any need to check for multiple things because of the insertion/deletion/search of 0, the algorithm is simpler:

```
Initialize probation count as 1

initialize boolean found as true
if( collision strategy of class is double hashing)
    (when strategy is double hashing, initialize int hash2, equating
to the return value of second hash function)
    while the first column of the array isn't equal to the parameter
searchFor
        implement the probation method and update address accordingly
        increment probation count
        end loop prematurely when an 'Empty' node is reached OR end
loop prematurely when probation count is greater than table size, set
boolean found to false
return probation count
```

The reason I end the loop when probation count exceeds the size is because at that point the loop repeats itself and we know that it will go in an infinite loop surely, so we shouldn't end up with an infinite loop inside quadratic and double hashing probe methods (I added this check inside the linear probe method too, just in case).

When the collision strategy is double hashing, one different thing I do is to check for an infinite loop situation, which happens when item to be added is 0 (parameter item is 0)

and (the result from second hash function) % (size of the table) is 0. I checked the first situation inside the insert function (explanation is below) and I checked for the second situation in this way:

```
inside the while loop, after end loop condition
    if(hash2 mod size is zero) {
        if(items[address][0] is not equal to searchFor)
            found = false;
            break;
```

When we run the remove (const int item) function, the probe function goes through the array by using the appropriate probation method and if an index found by the probation method is empty, it initializes found as true so as to make it false inside the loop if necessary. Then the remove function, if the Boolean *found* value is passed by reference as true, makes the first column of the address (again, passed by reference from the probe function) 0 (to "delete" the value, in which it makes it revert to its initial state) and the second column 0 also (here it is 0, which is different from the initialized -1. That is because I use the 0 value to determine if it is deleted and -1 for an empty node.) The delete function, when it calls the probe function, skips the indexes that are "deleted" when ending prematurely and only ends when an "empty" node is reached.

The way search(const int item) function finds the item through the probe function is the same as the way remove function does it. The only difference is that we initialize another integer variable here, called numProbes, and then equate it into the return value of probe function. We did not use the return integer value of probe function inside remove function. This also skips "deleted" indexes. If an empty node is re

Here are all the things insert function checks for:

```
if( item is NOT zero AND search result for item is true)
    return false
if( item IS zero AND collision strategy is double hashing)
    return false
if( item IS 0 AND zeroAdded)
    return false
if( item IS zero) which means that zeroAdded is false at this point
    zeroAdded = true
```

The aim of the first 'if' condition should be obvious – to not add a value to the table twice. The second if condition prevents adding 0 to the double hashing table – so the insertion of item value 0 to the double hashing HashTable is not possible as it would cause an infinite loop situation in the first place and adding it would not be possible with or without the second hash function even without this if condition. We are simply preventing an infinite loop from happening. Other things it checks for are if zero is already added. This is similar to the first condition as it is there to prevent adding the same number twice to the table but since it is harder for this implementation to check if a node is zero because it is empty or if it is zero and occupied, I assumed that this Boolean would better integrate what I have in mind.

After these checks, insert function uses probe function in this way: It first initializes an address that is equal to (with item mod tableSize), and checks whether the index is empty/deleted or not. If not, it calls the probe function, initializing a throwaway Boolean

for *found*, and calls the function's searchFor value as 0 (since we look for the first empty node basically). This works because the while loop will go through the array as long as an index for which the searchFor value does not exist, and will end when it is reached. Whether the index is deleted or empty becomes irrelevant as long as it is in its initial state (which is 0 in this case). Then it adds the value to the address's first column and equates the second column to 1 to make it in its "Occupied" state.

The insertion of number 0 was problematic for all of these, because I initialized the values for all indexes as 0 and make them 0 again when deleted; so, it was kind of hard for the insert function to look for empty values (substituting 0 for searchFor parameter of the probe function) and keep the newly inserted 0 at the same time. For this, I added a new Boolean variable for the class, bool zeroAdded, and initialized it as false inside the constructor. The value of it is changed inside the checks inside insert function. Another thing I did to insert it was to add another condition to the while loop, other than one that checks for empty indexes, which checks if the value of the address index is 0 and also if it is in 'Occupied' state, meaning that it has a value attached to the index. I prevent the addition of 0 into the table using double hashing collision strategy altogether because it causes an infinite loop situation to happen.

The algorithm for second hash function is simply an algorithm to reverse the integer. It initializes an integer as 0, multiplying it by 10 and adding (parameter item mod 10) to the integer. Then it returns the integer. This second hashing function is called only once.

I've implemented the analyze function in this way:

First, for the analysis of the successful searches, I went through each occupied item in the array and searched them in the array, adding probation counts to another double value I initialized as I went on. When no more occupied item is left, I divided the double value by the count of current items. This will give us the value for average successful probe count. I constructed a helper function for this with prototype const int totalSucc().

For the analysis of unsuccessful searches, I went through each index with a for loop, creating an int address variable and equating it into i value each time in the loop, and if the item in that address is occupied, I called the probe function with these values: address for the address, items[address][0] for the item, 0 for int searchFor (as we will try to find an empty node here), and a throwaway Boolean *found*. As it goes through trying to find an empty node, I add it to another integer I initialized as 0 and when there is no occupied item left, I divide this value by the item count. This will give us the value for average unsuccessful probe count. For unsuccessful searches, for the first index which is 0, since the search of 0 is actually quite hard and probably requires more probes than any other normal number, I instead searched (0+size) for this index. I also incremented the indexes, performing a search function and based on its result I decided if I needed to increment it by size or not. I increment it inside a loop which has a condition of

File reading is based on this: I create an fstream object and going through each line I look at the first indexes (I assume the input file is correct) and based on the char value of the first line ('S', 'D' or 'I') I first extract the number next to the char value by using another helper method which goes through spaces and/or tabs to get to the number and

using stringstream class, turns the string number into an integer. Then the number turned into integer is called inside the function (whichever one we need to use, again based on the first char of the line) and the printing for this part of the program is done inside this function.

Question 2.2 – Testing with Text File

Size of the HashTable I created was initialized as 13 (a prime number).

Here is the content of my input text file:

```
Operation  Meaning
I 30
I 30
I 17
S 17
I 17
I 0
I 2
I 13
I 26
I 28
I 14
I 9
S 9
D 9
S 9
S 24
S 32
S 8
```

Here is the content of my output on Dijkstra machine:

```
LINEAR PROBING
30      inserted
30      not inserted
17      inserted
17      found after 2 probes
17      not inserted
0       inserted
2       inserted
13      inserted
26      inserted
28      inserted
14      inserted
9       inserted
9       found after 1 probes
9       deleted
9       not found after 2 probes
24      not found after 1 probes
32      not found after 3 probes
8       not found after 1 probes

Final table:
0:      0
1:      13
```

```
2:      2
3:      26
4:      30
5:      17
6:      28
7:      14
8:
9:
10:
11:
12:
Successful average:     2.9
Unsuccessful average:   3.8

QUADRATIC PROBING
30      inserted
30      not inserted
17      inserted
17      found after 2 probes
17      not inserted
0       inserted
2       inserted
13      inserted
26      inserted
28      inserted
14      inserted
9       inserted
9       found after 5 probes
9       deleted
9       not found after 6 probes
24      not found after 1 probes
32      not found after 1 probes
8       not found after 1 probes

Final table:
0:      0
1:      13
2:      2
3:      28
4:      30
5:      17
6:
7:
8:
9:      26
10:     14
11:
12:
Successful average:     2.1
Unsuccessful average:   3.7

DOUBLE HASHING
30      inserted
30      not inserted
17      inserted
17      found after 2 probes
17      not inserted
```

```
0       not inserted
2       inserted
13      inserted
26      inserted
28      inserted
14      inserted
9       inserted
9       found after 1 probes
9       deleted
9       not found after 2 probes
24      not found after 1 probes
32      not found after 2 probes
8       not found after 1 probes

Final table:
0:      13
1:      14
2:      2
3:
4:      30
5:
6:      28
7:      26
8:
9:
10:     17
11:
12:
Successful average:     1.6
No unsuccessful average calculated
```

Question 3.3 – Successful and Unsuccessful Probe Averages

Load factor $\alpha = \frac{(current\ number\ of\ items)}{tableSize} = \frac{8}{13}$ in this case.

For linear probing, approximate average probes are:

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)\ for\ a\ successful\ research$$

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)\ for\ an\ unsuccessful\ research$$

Then for the case we tested:

$$\frac{1}{2}\left(1 + \frac{1}{1-\frac{8}{13}}\right) = 1.8$$

$$\frac{1}{2}\left(1 + \frac{1}{\left(1-\frac{8}{13}\right)^2}\right) \cong 3.9$$

These are the theoretical results. Empirical results for linear probes are as follows:

```
Successful average:     2.1
Unsuccessful average:   3.1
```

For quadratic probing, approximate average probes are:

$$-\frac{\ln(1-\alpha)}{\alpha}\ for\ a\ successful\ research$$

$$\frac{1}{1-\alpha}\ for\ an\ unsuccessful\ research$$

Then for the case we tested:

$$-\frac{\ln\left(1-\frac{8}{13}\right)}{\frac{8}{13}} \cong 1.6$$

$$\frac{1}{1-\frac{8}{13}} \cong 2.6$$

These are the theoretical results. Empirical results for quadratic probes are as follows:

```
Successful average:     2.1
Unsuccessful average:   3.1
```

For double hashing, approximate average probes are:

$$-\frac{\ln(1-\alpha)}{\alpha}\ for\ a\ successful\ research$$

We did not add 0 in this case, which means that loading factor is different. Then for the case we tested:

$$-\frac{\ln\left(1-\frac{7}{13}\right)}{\frac{7}{13}} \cong 1.4$$

This is the theoretical results. Empirical result for double hashing is as follows:

```
Successful average:     1.6
```

In general, the empirical results seem close enough to the theoretical results, however some differences were present. In particular, the empirical results seemed to result in a much greater average than the theoretical results and I think the reason for that is that the numbers inserted were generally numbers that required many probes to be inserted – for linear probing, the items and their probes are as follows: 30 – 4; 17 – 4, 5; 0 – 0; 2 – 2; 13 – 0, 1; 26 – 0, 1, 2, 3; 28 – 2, 3, 4, 5, 6; 14 – 1, 2, 3, 4, 5, 6, 7. So we can see that the average probe count should be $\frac{1+2+1+1+2+4+5+7}{8} \cong 2.9 > 1.8$. It is reasonable by looking at the count of probes that the empirical probes are greater than the theoretical probes. Again for quadratic probing, the probes are 30 – 4; 17 – 4, 5; 0 – 0; 2 – 2; 13 – 0, 1; 26 – 0, 1, 4, 9; 28 – 2, 3; 14 – 1, 2, 5, 10. Now we can see that although it is less than the count of linear probing, $\frac{1+2+1+1+2+4+2+4}{8} \cong 2.1 > 1.6$. For double hashing with the second hash

function we used, probes are: 30 – 4; 17 – 4, 10; 2 – 2; 13 – 0; 26 – 0, 10, 7; 28 – 2, 6; 14 – 1 and 0 is not inserted. For double hashing, $\frac{1+2+1+1+3+2+1}{7} \cong 1{,}6 > 1.4$ and both double hashing and quadratic probing methods require fewer average cases than linear probing – this also aligns with what we have learnt about HashTables.