Elif Özer
21602495
CS202
Section: 3

HW4 - REPORT

PART 1

HashTable Class:

Private Variables:

CollisionStrategy collision: To determine collision strategy type

int tableSize: Size of hash table

int currentSize: Number of elements in the hash table

int * hashItems: Array that contains the items in the hash table with corresponding index number

bool * deleted This boolean array is like a flag such that if element in the index i deleted, ith index in deleted array will be true so that insertion and search operation can be done correctly.

bool * emptyCheck This boolean array is to control whether an index in the hash table is empty or not. For example, if we delete an item from index i, we make the ith index of emptyCheck array true so that we can insert another element to ith index if needed.

bool * occupied This boolean array is to control whether an index is occupied or not. If we insert an element to ith index of array hashItems than be make ith index of occupied array be true so that new element can not be inserted there and search operation can be done correctly. In addition, if we deleted an element from ith index of hashItems we make ith index of occupied array false so that new element can be inserted there and search operation can be done correctly.

Public Functions:

int primaryHash(int key)
This function is for help function for the function **int hash(int key, int i).** PrimaryHash function simply implement the modulo operation to the key and return the index variable that is used in the **hash** function.PrimaryHash simply does this: index = key mod tableSize and returns index.

int hash2(int key):
This function is used if collision resolution is doubleHashing. hash2 function reversed digits of the key such that if key is 123, this function returns 321.

int f(int index):

This function is for collision strategy and it is used in the **hash(int key, int i)** function. It decides which operation will be implemented accorging to collision strategy. If CollisionStrategy is LINEAR, f returns the index itself; else if CollisionStrategy is QUADRATIC, f returns (index*index). If CollisionStrategy is DOUBLE f returns (i*hash2(key)).

int hash(int key, int i)

This function is for determining the index of key. i represent the iteration number. This function does this operation. $h_i(key) = hash(key) + f(i) \bmod tableSize$. Function f(i) is for collision strategy. This f(i) implements its own operation according to collision resolution.

bool insert(const int item):

This function is to insert the parameter item to the hashTable. First, this function checks whether parameter **item** is already table or not to avoid inserting same number to the table and returns false if item is already in the table.Then, this function calls the **index=hash(item,0)** function since iteration number is 0 initially. Then, this function checks whether index is empty or not by if statement such that if(emptyCheck[index]==true) is satisfied, this means that we find the location and it is empty so we can simply insert the item there and we need to occupied[index]= true and emptyCheck[index] false.

SOLVING INFINITE LOOP:

To solve infinite loop, we should be aware of that hash function may give maximum 11 different results for each key when we have tableSize= 11 because hash function uses modulo operation, so it can not give different than 0,1,2,3,4,5,6,7,8,9 or 10 and this result of hash function may repeat itself after 11 iteration in the worst case. Therefore, I have used for loop which iterates 11 times. For 3 cases that are LINEAR, QUADRATIC and DOUBLE collision resolutions, this idea works because f depends on i(iterate number) and this proves that it can give max 11 differens results in the worst case but it does not have to give 11 different results in each case.,

STOPPING CASE:

If index = hash(item,0) (which computes the index for first iteration) is occupied we need to enter the for loop. The table size may be so huge and for loop can iterate too many times, so wee need to avoid unneccessary iterations. To do this, when appropriate index is found we should make the return parameter true and go out from the for loop by using break. At the end, we return that parameter.

bool remove(const int item):

To remove an item from the hashTable, this function use for loop to iterate the table. Before entering the for loop, I compute the index of item by calling the hash(item) function. Then, I enter the for loop. In the for loop to check whether that index is equal to paremeter item, I use this if case **if(occupied[index] &&**

hashItems[index]==item). If condition is true, function simply deletes the item by making deleted and occupied are false at that index and empty is true.

AVOIDING INFINITE LOOP:

I implement the for loop which iterates tableSize time so that it does not go to infinite loop. Reason for iterating tableSize time is that hash function uses modulo operation, so it can iterate maximum tableSize times. The reason is that tableSize is 11, so in the worst case it may give 11 different remaining number in the worst case and it will go back to beginning remaining number and repeats the hash function with same results. For 3 cases that are LINEAR, QUADRATIC and DOUBLE collision resolutions, this works because f depends on i(iterate number) and this proves that it can give max 11 differens results in the worst case but it does not have to give 11 different results in each case.

STOPPING CASE:

We enter the for loop to find the removed element but for loop can be so huge that iteration can take so long. So, when we enter the if statement in the for loop, we can should return true after making deleted and occupied are false at that index and empty is true.

bool search(const int item, int& numProbes):

This function searchs the given parameter item in the hash table and returns true if item is in the hash table. This function also returns how many probes can be done while searching the item parameter and returns this number in the parameter numProbes.

AVOID INFINITE LOOP WHILE SEARCHING:

As explained in the remove and insert methods, 11 iterates is enough while searching the given item where 11 is the tableSize. Reason for iterating 11(tableSize) times is that hash function uses modulo operation which can give maximum 11 different results in the worst case. For LINEAR, QUADRATIC and DOUBLE collision strategies, this idea works since f function depends on the iterate number and this proves that it can give max 11 results in the worst case.

STOPPING CASE:

We have 2 if conditions in the for loop. When we enter one of these if conditions that are **if(emptyCheck[index]== true && deleted[index] == false)** and **if(hashItems[index]== item && deleted[index] == false),** we should boolean found be true and go out from the for loop by using break. At the end of function, we return the found.

void display():

To display the hashTable, I simple use for loop and this for loop iterates tableSize times. In the for loop, I checked each index whether it is empty or not. If ındex is empty, I just print the index number; else I print index number and the element at that index.

void analyze(double& numSuccProbes, double &numUnsuccProbes)

This function analyze the LINEAR, QUADRATIC and DOUBLE collision resolutions and returns average number of successful searches in the parameter **numSuccProbes** and returns average number of unsuccessful searches in the parameter **numUnsuccProbes.**

For Successful Search

To compute the average number of probes for successful search, I simply iterate the whole array and call search(searchedItem, numProbe) function for each item in the hashTable (if hashItems[i] is occupied which means that index i is not empty) where searchedItem is hashItems[i] int and numProbe is probes which done while searching the hashItems[i]. To find the average, I use a variable sumSucc to compute the sum of numProbe variable for each item in the array by doing (sumSucc += numProbe) operation. When for loop is over, I find the average by doing the operation (average = sumSucc / counter) where counter is incremented in the if statement that is (if hashItems[i] is occupied which means that index i is not empty)

For Unsuccessful Search

To compute the average number of probes for successful search, this function returns -1 if collision resolution strategy is double hashing. For LINEAR or QUADRATIC probing, this function initiate a search that starts at each array location (index), and count the number of probes needed to reach an empty location in the array for each search as it is explained in the HW4 description. To do this, we need for loop for starting the search in each array location. In the for loop, we need a while loop to control whether we reach empty location or not such that **while(!emptyCheck[tempI] && numberOfProbes <= tableSize)** where tempI is i(index). We need tempI because we call tempI = hash(i,numberOfProbes) hash function in the for loop so we need another index called tempI. In the while loop I increment the number of probes with parameter numProbes++. When for loop is over I add numProbes by (sumUnsucc += numberOfProbes) so that I can find the average at the end.

PART 3:

For LINEAR probing:

α= **(current number of items) / tableSize** where α is load factor

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$ *for a successful search*

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$ *for an unsuccessful search*

In my implementation, there are 9 items in the hash table and table size is 11 so load factor is 9/11.

Theoratical result for successful search is **3.25,** empirical result is **1.88889**

Theoratical result for unsuccessful search is **15.625,** empirical result is **3.81818**

For QUADRATIC PROBING and DOUBLE hashing:

α= **(current number of items) / tableSize** where α is load factor

$$\left[\frac{1}{\alpha}(\log_e \frac{1}{1-\alpha})\right] = \frac{-\log_e(1-\alpha)}{\alpha}$$ *for a successful search*

$$\frac{1}{1-\alpha}$$ *for an unsuccessful search*

In my implementation, there are 9 items in the hash table and table size is 11 so load factor is 9/11.

Theoratical result for successful search is **2.083,** empirical result is **1.55556** for quadratic probe and **3.11111** for double hashing.

Theoratical result for unsuccessful search is 1/(1-9/11) = 11/2 = **5.5,** empirical result is **4.54545** for quadratic probe and -1 for double hashing.

For Unsuccessful Search:

I have found different average number of search results from the theoratical results which can be derived from my “input.txt” file or my table size. My results are less than the expected values according to the theoratical formulas which means that my hash tables’ implementation’ performance is less than the average. The reason for this can be that my hash tables’ number of probes of unsuccessful searches are less than expected. These empirical results may be closer to the theoratical results when the table size become larger and inputs is changed such that every single possibility is tried. However, I create my input text file with hand as it is expected and stated in the HW4 clarification, so it is not possible to determining every case can be occurred in the table. I have tried to test many cases but catching the exact result is not possible since theoratical formulas give the result which can be at most. I have checked my results with hand as possible as can be and I found results which is closer to outputs of my program.

For Successful Search:

For linear and quadratic successful searches I have found the values less than expected values according to the theoratical results. Reason for this is same as unsuccessful searches’ results explained above.

For double hashing, I have found the result greater than expected result which can be derived from that my hash table’s performance. I didn’t try all possibilites because it is impoessible when I create my input.txt file with hand. Therefore, some differences are normal compared to theoratical results.

-In addition, in each LINEAR, QUADRATIC and DOUBLE collision strategy, my program’s ouput is exactly the same as expected tables.