CS280 Homework 8

Emmet Blassingame

Hash tables

Problem 1

Problem 1.a (10 points):

What is the value of h(ABCDZ)?

The hash function is h(ABCDZ).

==> floor(decimal(sum(ABCDZ) * 11/30)* 10)

sum(ABCDZ) = 65 + 66 + 67 + 68 + 90 = 356

decimal (sum(ABCDZ) * 11/30) * 10) = decimal(356 * 11/30) * 10)

= .53 * 10

= 5.3

h(ABCDZ) = floor(5.3) = 5

Problem 1.b (10 points):

What is the value of h(CMNQRY)?

The hash function is h(CMNQRY).

==> floor(decimal(sum(CMNQRY) * 11/30)* 10) sum(CMNQRY) = 67 + 77 + 78 + 81 + 82 + 89 = 484

decimal (sum(ABCDZ) * 11/30) * 10) = decimal(484 * 11/30) * 10)

= .46 * 10

= 4.6

h(CMNQRY) = floor(4.6) = 5

Problem 1.c (10 points):

How large of a hash table do we need to accommodate all possible indices generated by ? Explain how you arrived at your answer.

The hash table will need an array of 11 elements which will need to store values from 0 to 10 since the function only returns values from 0 to 10.

Problem 2

Problem 2.a (20 points):

Insert the following strings into the hash table using linear probing. When a collision happens, use a step size of 3. In other words, if you collide at index j, try inserting the element at index j+3, then at j+6, etc.. Show the hash table after each insert.

After inserting **"ABCA"**, the hash table would look like this:

H = [-, -, -, ABCA, -, -, -, -, -, -]

After inserting **"DDZ"**, the hash table would look like this:

H = [-, -, -, ABCA, -, -, -, -, DDZ, -]

After inserting **"CZXY"**, the hash table would look like this:

H = [-, -, -, ABCA, CZXY, -, -, -, DDZ, -]

After inserting **"MNOP"**, the hash table would look like this:

H = [-, -, -, ABCA, CZXY, -, -, MNOP, DDZ, -]

After inserting **"WWY"**, the hash table would look like this:

H = [-, -, -, ABCA, CZXY, -, WWY, MNOP, DDZ, -]

After inserting **"AAC"**, the hash table would look like this:

H = [AAC, -, -, ABCA, CZXY, -, WWY, MNOP, DDZ, -]

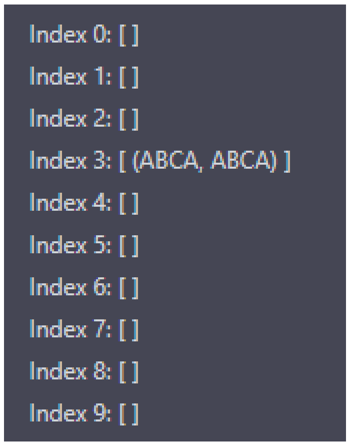
After inserting **"BBA"**, the hash table would look like this:

H = [AAC, -, -, BBA, CZXY, -, WWY, MNOP, DDZ, -]

Problem 2.b (20 points):

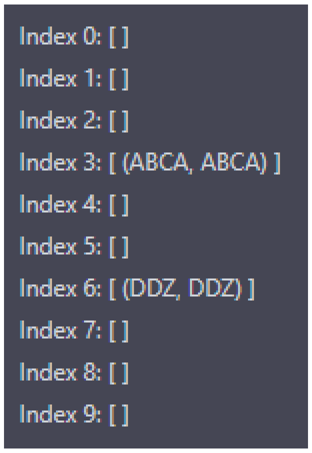
Insert the following strings into the hash table using chaining. Show the hash table after each insert.

After inserting **"ABCA"**, the hash table would look like this:



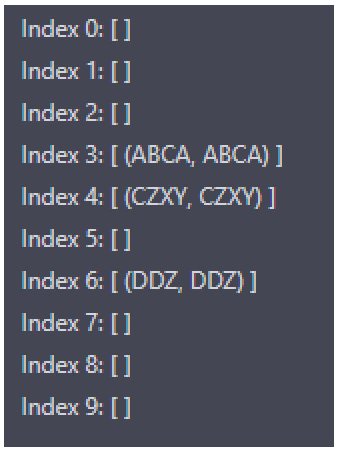
```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA) ]
Index 4: [ ]
Index 5: [ ]
Index 6: [ ]
Index 7: [ ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"DDZ"**, the hash table would look like this:



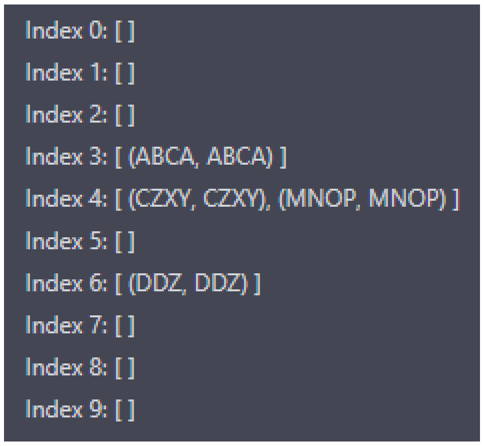
```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA) ]
Index 4: [ ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"CZXY"**, the hash table would look like this:



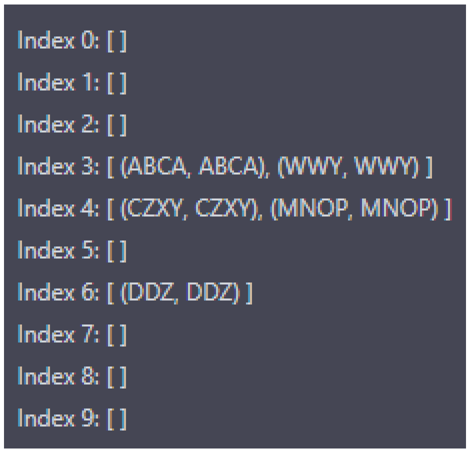
```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA) ]
Index 4: [ (CZXY, CZXY) ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"MNOP"**, the hash table would look like this:



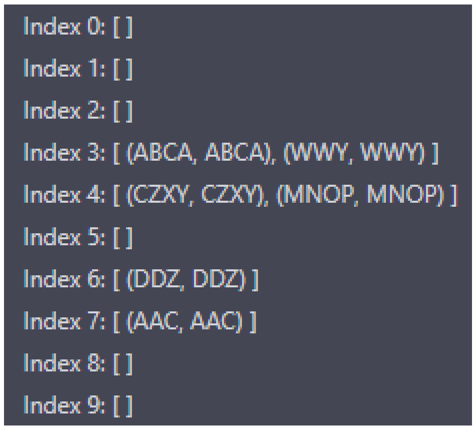
```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA) ]
Index 4: [ (CZXY, CZXY), (MNOP, MNOP) ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"WWY"**, the hash table would look like this:



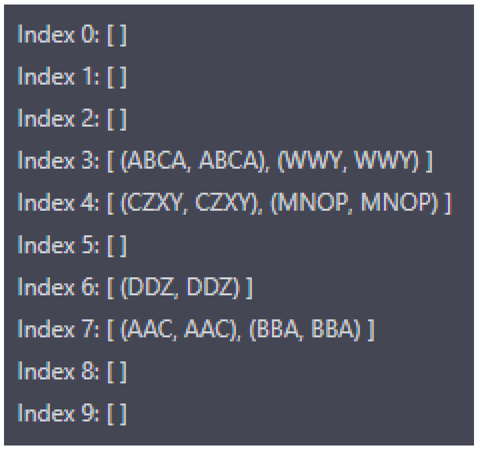
```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA), (WWY, WWY) ]
Index 4: [ (CZXY, CZXY), (MNOP, MNOP) ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"AAC"**, the hash table would look like this:



```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA), (WWY, WWY) ]
Index 4: [ (CZXY, CZXY), (MNOP, MNOP) ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ (AAC, AAC) ]
Index 8: [ ]
Index 9: [ ]
```

After inserting **"BBA"**, the hash table would look like this:



```
Index 0: [ ]
Index 1: [ ]
Index 2: [ ]
Index 3: [ (ABCA, ABCA), (WWY, WWY) ]
Index 4: [ (CZXY, CZXY), (MNOP, MNOP) ]
Index 5: [ ]
Index 6: [ (DDZ, DDZ) ]
Index 7: [ (AAC, AAC), (BBA, BBA) ]
Index 8: [ ]
Index 9: [ ]
```

Problem 2.c (10 points):

I want to insert 20 elements into the hash table described at the beginning of Problem 2. For collision resolution, should I use chaining or linear probing? Justify your choice.

It is not possible to determine which collision resolution technique is best for a given situation without additional information about the characteristics of the data that will be stored in the hash table, such as the number and distribution of the keys, the size of the hash table, and the requirements for time efficiency.

Problem 3

Problem 3.a (5 points):

What attribute will you use as the key? In other words, what attribute would you suggest we apply the hash function to in order to identify an array index to store the dog's information at? You must justify your answer.

The attribute we can use as the key is the dog id. This is a good choice for a key because it is a four-digit unique identifier that can take on any value from 0000 to 9999. This means that it is unlikely that there will be any collisions in the hash table and the key can be used to index the array efficiently.

Problem 3.b (5 points):

Define a hash function on the attribute you chose to use as a key.

The hash function we can use is just a modulo function, where the key is divided by the size of the array, and the remainder is used as the index. For example, if the dog id is 8888 and the array size is 10, the index will be 8888 % 10 = 8.

Problem 3.c (5 points):

What kind of collision resolution will you use in your hash table. You must justify your answer.

We can use linear probing as the collision resolution method. This is a good choice because it is simple and efficient; it works by checking the next position in the array if the index obtained using the hash function is occupied. If a collision is found, the next position is checked until an empty slot is found or the end of the array is reached.

Problem 3.d (5 points):

How large will you make your hash table? You must justify your answer.

We can make the hash table large enough to fit all the dogs in the list. This can be done by making the array size equal to the size of the list (i.e. 400). This ensures that we have enough buckets to fit each of the 400 dogs. Additionally, we can also use a good hashing function to hash the data efficiently and minimize collisions.