Problem session 12

Dibran Dokter 1047390

December 5, 2019

12

12.1

To find the maximum element in the direct-address table we need to walk through the entire table to find the maximum value. We start with a value of $-\infty$ and then find the maximum value in the address table. When we do this we need to make a distinction between chaining and open addressing. In the case of chaining we may need to go through a chain for every index, for open addressing we just need to go through the number of indices in the table.
For open addressing we can be sure of a time complexity of $\mathcal{O}(m)$ since we need to run through all the indices.
In the case of chaining we would need to add the number of values that are chained. So we get something like $\mathcal{O}(m * 1 + \alpha)$ where α is the number of values that are chained in the addressing table.

12.2

In this case $h_1(k)$ would be better since we want to uniformly distribute the possible numbers of the keys. And for h_1 this is the case since when we use modulo 10 we distribute evenly between indices 0 and 9. If we were to choose $h_2(k)$ instead we would end up putting all the values in the indices 0 through 2 since $29/10 = 2.9$ and floor$(2.9) = 2$.

12.3

When we add the values we first run the hashing function on the value, namely $h(k) = (2k+5) mod 11$. Then we add the value at the index given by this hashing function. If there is already a value at this index, we add the value behind the value already there using a pointer to connect the two.

Doing this gives us the table below describing the final hash table. Note that the index 5 has the most elements because there are alot of element which have $2k \mod 11 = 0$.

Index	**list**
0	NIL
1	20
2	NIL
3	NIL
4	$16 \rightarrow 5$
5	$44 \rightarrow 88 \rightarrow 11$
6	$94 \rightarrow 39$
7	$12 \rightarrow 23$
8	NIL
9	13
10	NIL

12.4

12.4.1

To do this we first hash the value and then try to insert at that index. If that index is not available we look at the next index and so forth. Doing this gives us the following table:
Here the list values indicate a pair with the actual value and the number of probes needed to place te value.

Index	list
0	(77,2)
1	(13,0)
2	NIL
3	NIL
4	(40,0)
5	(88,0)
6	(45,0)
7	(23,0)
8	NIL
9	(20,0)
10	(29,0)
11	(12,0)
12	(8,0)

Here we have that every insertion except for 25 and 77 succeeded right away. For the values 25 and 77 we needed to probe twice. This give a total of 4 probes necessary.
This is assuming that the initial probe for insertion is not counted. Otherwise we would need to add a single probe for every insertion.
Note that we placed the value 77, which is hashed to 11 at index 0, this is because we probe linearly modulo 13, and after 12 we try 13, which is 0 when we take $13 \mod 13$.

12.4.2

To do this we do the same as the previous exercise, except that we use double hashing to solve the collisions.

Index	list
0	(23,1)
1	(13,0)
2	(77,2)
3	NIL
4	(40,0)
5	(88,0)
6	(45,0)
7	NIL
8	NIL
9	(20,0)
10	(29,0)
11	(12,0)
12	(8,0)

As can be seen in the table above we now only have 3 collisions and thus have reduced the number of collisions by 1.

12.5

To insert a value into a hashTable we need to find an empty place to insert. Here an empty place is either a index that has NIL or it has a TombStone. To find this place we first look at the index that the hash gives for hash(v,0) and check whether we can place it there. After that we enter a while loop that looks at the next index and checks whether we can place it there. When we have found a place to insert we insert the value.

Insert:

```
Insert(v, hashTable) // v is the value we wish to insert
{
    hashTable; // The hashtable to insert into
    probeCount = 0;
    hash(value, probeCount); // The hashing function for the hashmap
    index = hash(v,probeCount); // Find the index to insert into
    if(hashTable[index] == NIL || hashTable[index] == TombStone)
    {
        hashTable[index] = v;
    }
    else
    {
        while (hashTable[index] != NIL || hashTable[index] != TombStone)
           // Find the next empty index
        {
            probeCount++;               // Increase the probecount
            index = hash(v,probeCount); // Find the new index
        }
        hashTable[index] = v;           // The index is empty or tombstone
           , so we insert
    }
}
```

To delete a value from the hashtable we do the same as above but instead of looking for a empty index we look for the index containing the value we want to delete. When we find this index we delete the value by placing a tombstone.

Delete:

```
Delete(v, hashTable) // v is the value we wish to delete
{
    hashTable; // The hashtable to delete in
    hash(value, probeCount); // The hashing function for the hashmap
    probeCount = 0;
    index = hash(v,probeCount); // Find the index to delete from
    if(hashTable[index] == v)
    {
        hashTable[index] = TombStone;
    }
    else
    {
        while (hashTable[index] != v) // Find the right index
        {
            probeCount++;               // Increase the probecount
            index = hash(v,probeCount); // Find the new index
        }
        hashTable[index] = TombStone;   // The index is found, so we
           delete
        }
    }
```

When we want to search for a value in the hashmap we look for the value itself and when we find it we return it.

Search:

```
    Search(v, hashTable) // v is the value we wish to find
    {
        hashTable; // The hashtable to search
        hash(value, probeCount); // The hashing function for the hashmap
        probeCount = 0;
        index = hash(v,probeCount); // Find the index to search from
        if(hashTable[index] == v)
        {
            return hashTable[index]; // Return the value
        }
        else
        {
            while (hashTable[index] != v)   // Find the right index
            {
                probeCount++;               // Increase the probecount
                index = hash(v,probeCount); // Find the new index
            }
            return hashTable[index];        // Return the value
        }
    }
```