CS146 - Homework 5

Frank Mock

April 6, 2014

For 5.1 parts a - d, I used an initial hashtable size of 10 considering the input size is 7.

5.1 a.

Using Seperate Chaining	
0	
1	→4371
2	
3	→6173→1323
4	→4344
5	
6	
7	
8	
9	→1989→9679→4199

5.1 b.

Using Linear Probing	
0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199

5.1 c.

Using Quadratic Probing	
0	9679
1	4371
2	
3	1323
4	6173
5	4344
6	
7	
8	1989
9	4199

5.1 d.

Using $h_2(x) = 7 - (x \bmod 7)$	
0	
1	4199
2	9679
3	4344
4	4371
5	
6	1989
7	1323
8	1673
9	

5.14

Describe a procedure that avoids initializing a hash table (at the expense of memory)
After much thought, I came up with a method that uses a linked list and splay tree. Assuming we want to keep the constant time O(1) insertion of a hash table, we could make all insertions at the head of a linked list of nodes of key and value pairs. The key is what the data mapped to after being hashed and the value is the data item. All the keys would not necessarily be unique since collisions can occur. The splay tree would only be built when an item needs to be accessed. When an access request is made, the tree that is built should be somewhat balanced due to the relatively uniqueness of the keys produced during hashing. The initial building of the tree would have the greatest cost, O(n log n) since the whole list is iterated over as each item is placed in the tree. Therefore, the first access would incur a O(n log n) hit. However, with each additional access operation the splay process will float the item being accessed to the top of the tree (root of the tree) and in the process balance the tree. Deep access will have a greater affect than shallow ones. If the same item is accessed again (consecutive access), the time cost would be a constant time operation.

Future insertions would build a new linked list of nodes containing key and value pairs, which again guarantees constant time O(1) insertion. However, there is a difference with future access operations. If the item being requested is already in the tree, it is not necessary to "dump" the items of the linked list into the splay tree. You simply traverse the tree and return it. The splay tree operations will float this item to the top. This does not break the log(n) after m accesses guarantee of splay trees. In fact, even if the item is not already in the tree and the items of the linked list must be added to the splay tree it still does not break the $mlog(n)$ guarantee of splay tress as long as there is a fairly good balance of insertions and access operations. The added constant of building each new linked list is amortized over all operations and you still get a $mlog(n)$ for access operations and O(1) for insertions. My method to deal with the problem presented would definitely be taxing on the memory since it would require lots of pointers, but it does not initialize a hash table (array).

5.19

a.

$$\frac{1}{(1-\lambda)-\lambda-ln(1-\lambda)}$$

This is the same as the given cost for insertion, because insertions and unsuccessful searches require the same number of probes.

b.

Expected cost of a successful search, is the average value of the computed time for each of the n keys in the hash table using the given insertion time as a function of lamda.

let $y = f(\lambda) = \dfrac{1}{(1-\lambda)-\lambda-ln(1-\lambda)}$

$$y_{avg} = \frac{\int_{m-n}^{m} f(\lambda)d\lambda}{\lambda} \qquad \text{n is the number of keys in the hash table and m is the hash table size}$$

$$= \frac{1}{\lambda}\int_{m-n}^{m} f(\lambda)d\lambda$$

$$= \frac{1}{\lambda}\int_{m-n}^{m} \frac{1}{(1-\lambda)-\lambda-ln(1-\lambda)}d\lambda$$

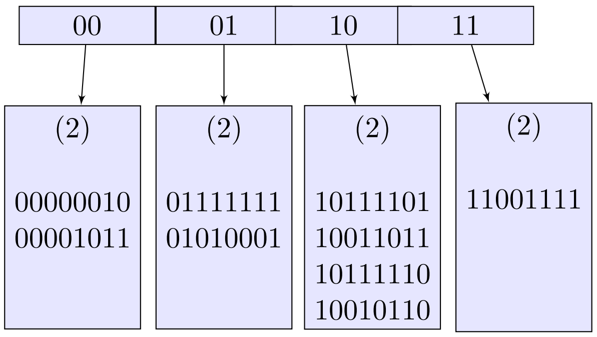
My calculus is rusty. I have tried to integrate this numerous times, but nothing I come up with looks reasonable (all messy).

I suspect that the cost of unsuccessful searches/insertions is greater than successful searches

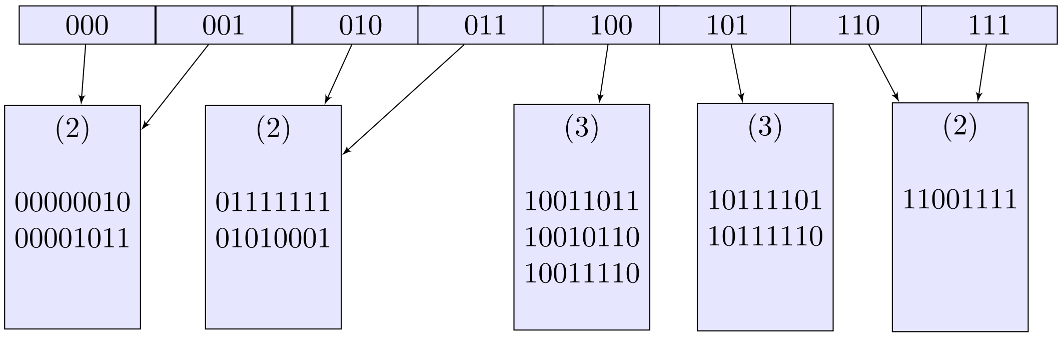
$$\frac{1}{(1-\lambda)-\lambda-ln(1-\lambda)} > \frac{1}{\lambda}\int_{m-n}^{m} \frac{1}{(1-\lambda)-\lambda-ln(1-\lambda)}d\lambda$$

5.27

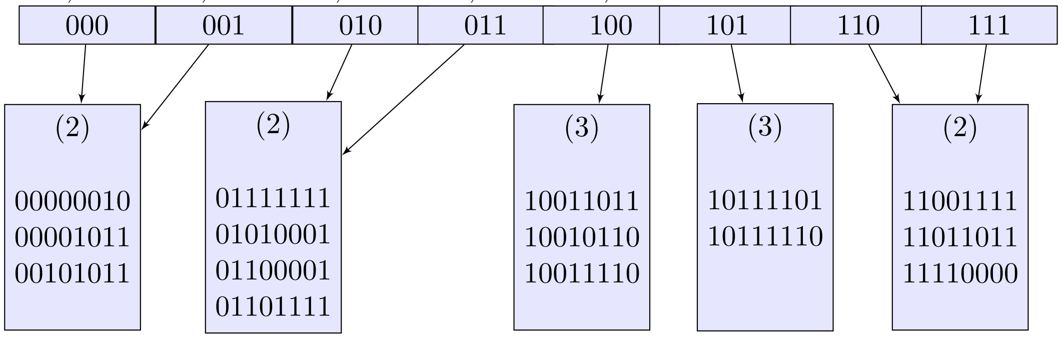
Extendible hashing data structure with M = 4. After inserting 10111101, 00000010, 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111. The third leaf is full.



Next 10011110 is inserted which causes the third leaf to split.



Next, 11011011, 00101011, 01100001, 11110000, 01101111 are inserted.


000
001
010
011
100
101
110
111
(2)
00000010
00001011
00101011
(2)
01111111
01010001
01100001
01101111
(3)
10011011
10010110
10011110
(3)
10111101
10111110
(2)
11001111
11011011
11110000