Robert Barnes

DSA 20407

2013b

Maman 16

1) Assuming that the hash function uniformly distributes the keys answer the following.

A) Given an empty hash table of size m with chaining, what is the probability that the first 4 insertions all are hashed to the same value?

The first key may be placed in any of the m buckets. Each subsequent insertion has a probability of $\frac{1}{m}$ to be in the same bucket as the initial insertion, giving a probability of $\frac{1}{m^3}$.

B) Given an empty m entry hash table using open addressing we insert 3 values. What is the probability that inserting the third value will require 3 steps?

On the first attempt to insert the third value, the probability to conflict with on of the first two values is $\frac{2}{m}$. Assuming a conflict with one of the first two values the probability of a second conflict is $\frac{1}{m-1}$. On the third attempt the value is succesfully inserted. Thus we have a probability of $\frac{2}{m^2-m}$.

C) We are given a hash table whose load factor is related to the number of values currently in the table by $\alpha = 1 - 1/\lg n$. Assuming open addressing, what is the expected number of steps in a failed search as a function of n?

We take Theorem 11.6 and substitute α to get:

$$\frac{1}{1-(1-1/\lg n)} = \lg n$$

2) Assume an m entry hash table and a hash function $h(k)$. Show that:

1. Compute the value $i = h(k)$, and set $j = 0$.
2. Probe in position i for the desired key k. If you find it, or if this position is empty, terminate the search.
3. Set $j = (j+1) \mod m$ and $i = (i+j) \mod m$, and return to step 2.

is an instance of the quadratic probing scheme.

We see that the values of j cyclically runs from 0 to $\frac{m(m-1)}{2}$. Substituting we get that $i = (h(k) + (\frac{j(j-1)}{2}) \mod m) \mod m$ which is equivalent to the equation $h(k,i) = (h`(k) + 0.5i + 0.5i^2) \mod m$.

3) Given an arbitrary binary tree, add fields to each node in total $O(n)$ time such that it will be possible to determine if two arbitrary nodes from the tree have a ancestor / descendent relationship in O(1) time.

We add two fields to each node, PreOrder and PostOrder which contain the position of the node in a PreOrder or PostOrder walk, respectively. Given any two nodes, n1 and n2 we have that they share such a relationship if and only if `n1.PreOrder < n2.PreOrder XNOR n1.PostOrder > n2.PostOrder` is true.

```
PreOrderWalk(T, i)
T.PreOrder = i;
if T.left != NIL i = PreOrderWalk(T.left, i + 1 )
if T.right != NIL i = PreOrderWalk(T.right, i + 1 )
return i

PostOrderWalk(T, i)
if T.left != NIL i = PostOrderWalk(T.left, i + 1 )
if T.right != NIL i = PostOrderWalk(T.right, i + 1 )
T.PostOrder = i ;
return i

InsertOrdering(T)
PreOrderWalk(T, 0)
PostOrderWalk(T, 0)
```

Each walk takes $O(n)$ time and the relationship determination takes constant time.

4) Given a node z, we remove the node by finding it's successor y, then swapping the left nodes of z and y and then replacing z with it's right child r. Show that this is correct, what is the running time and what are it's pros and cons.

We have by the structure of a bst, that the successor y of z has no left child because it is the smallest element in the right subtree of z. If the left child of z is l and it's right child r, then we have that

$$l < z < y < r$$

Thus, swapping the left branch of z for the left branch of y will maintain this relation and the validity of the bst since all children of l are also less than z and thus less than y. Now z has only one child and by assumption can be deleted from the tree in the regular fasion.

It takes $O(h)$ time to find the successor of any element and swapping the branches takes constant time. By assumption $h = \lg n$ and thus the routine runs in $O(\lg n)$ time.

The primary advantage is that the element passed to the routine is the element deleted, and thus any pointers which the user may have to nodes in the tree will not be invalidated. I see no disadvantage as swapping two pointers will generally be quicker than copying over all the satelite data associated with a node except in the most trivial cases.

5) Proove that it is possible to reconstruct a binary search tree from an array of values created by a post order walk of the tree.

An array created by a post order walk of a binary search tree will have the following properties. First, given any arbitrary binary search tree, we have that all the children of the root in the left branch of the tree will be placed in a contiguous block at the beginning of the array in a unique way. These are all the elements less than the root. Then, all the elements in the right branch of the root will be placed in a contiguous block in a unique way. These are the elements that are greater than or equal to the root. The final position in the array is held by the root itself. Further, any arbitrary element within the array will be the root of a subtree such that these properties hold, i.e. to the immediate left of that element will be a contiguous block (possibly empty) forming the right branch of the subtree, and to the immediate left of that block will be a contiguous block (potentially empty) forming the left branch of the sub-tree. This construction is unique and deterministic for any bst. Thus we have that the tree can be rebuilt by simply calling insert on the elements of the array from right to left. This is clearly true for $n = 1$. By the properties above we have that the block of size less than n to the immediate left of the rightmost element forms a subtree all of whose elements are greater than that element and calling insert on these elements must place them in the right branch in a unique way. Similarly, the block to the left of this block contains elements all of which are smaller than that element and the size of this block is also less than n. It's elements must be placed the left branch of the tree in a unique way. Thus we have a unique left branch and right branch with the rightmost element in the array being the root.

Regarding an in order walk simple scanning from left or right cannot always rebuild the same tree. For example, given a simple three element tree with one element in each branch, an in order walk will place the root in the middle of the array. Clearly then scanning from left or right will place the incorrect element in the root.

Regarding a preorder walk a similar situation hold as for that of the post order walk, except that the root is always the left most element in a given sub block, followed by it's left branch and the it's right branch.