Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 3: Data structures for lists and sets

SOLUTIONS

1. (a) *Number of copyings for 5000 appends.* The first 1000 append operations are "free" meaning that we don't need to copy any elements to a new array. Once the 1001st append operation is performed, then we have exceeded our capacitity and we need to perform an expansion. More precisely, we will copy all the 1000 elements of our original array to a new array, and add the new element as well. Therefore at this point we will be "charged" with 1000 copyings. Continuing like this, the next trigger that forces an expansion happens when we append element 1101, in which case we will perform 1100 more copyings. In general, we have the following number of copyings: $1000 + 1100 + 1200 + \ldots + 4900$ (the 5000th append doesn't trigger an expansion). That is

$$\sum_{i=10}^{49} 100i = 100\left(\sum_{i=1}^{49} i \;-\; \sum_{i=1}^{9} i\right) \;=\; 100((49\times 50)/2 \;-\; (9\times 10)/2)$$
$$=\; 100(1225-45) \;=\; 118000.$$

   (b) *Number of copying for n appends.* We generalise the reasoning above. We perform an expansion on the $(1000+100j+1)$th append operation for $j = 0, 1, \ldots$, copying the $1000 + 100j$ elements already present. So, the resulting formula for the total number of copyings is

$$\sum_{i=10}^{\lfloor (n-1)/100 \rfloor} 100i \;=\; 50\lfloor (n-1)/100 \rfloor \, \lfloor (n-1)/100 + 1 \rfloor \;-\; 4500$$

   Asymptotically, this is clearly $\Theta(n^2)$. So over a run of n appends, amortized cost is $\Theta(n)$ per append. Bad idea.

   (c) *What bad behaviour might result from Q's policy?* Suppose we start with 1001 appends, forcing an expansion to capacity 2000. We then do two deletes, bring number of elements down to 999 and forcing a contraction to capacity 1000. We then do two appends, forcing an expansion to 2000 again. We can repeat this cycle as often as we want, forcing 1999 copyings per 4 list operations (after the initial run of appends). Bad.

   A better contraction policy would be e.g. to wait till the proportion of the array in use dips to 0.25 before contracting by 0.5. This will ensure that following any expansion/contraction, if the list size is n then there will be at least $n/2 - 1$

list operations before the next expansion/contraction, leading to amortized $\Theta(1)$ cost per operation. For a detailed analysis, see the CLRS textbook, chapter 17.

2. (a) *With a bucket-style hash table, what does the given sequence of operations do?*
Each insert should add an element at the head of the relevant bucket. So after the five insertions, we have

$$\text{Bucket } 3: \ [143, 93]. \qquad \text{Bucket } 7: \ [777, 17, 47]$$

with all other buckets empty. Evaluating contains(93) will locate the 93 in bucket 3 and return True; contains(7) will search all the way to the end of bucket 7 and return False.

(b) *What happens with the given hash/probe scheme?*

- insert(47): table is empty, so first probe will succeed and put 47 in slot 7.
- insert(93): slot 3 is free so 93 goes there.
- insert(17): first-choice slot is 7 which is taken. For next choice, we have $F(17) = 3$ so $g(17, 1) = 0$ which is free: 17 goes there.
- insert(143): again, first choice (3) is taken. Since $F(143) = 7$, the successive probe values are 3,0,7,4,1,... (each time we add 7 and reduce mod 10). Of these, slots 3,0,7 are already taken, but $g(143, 3) = 4$ is free, so 143 goes there.
- insert(777): first choice (7) is taken. Have $F(777) = 3$, so successive probes are 7,0,3,6,9,.... Of these, 6 is the first free one, so 777 goes there.

Table now looks like this:

0	1	2	3	4	5	6	7	8	9
17	−1	−1	93	143	−1	777	47	−1	−1

- contains(93): first probe is 3 and we find 93 there, so return True.
- contains(7): we have $F(7) = 1$. First probe is at 7 which contains $47 \neq 7$. Second probe is at 8 which is blank. So we may conclude that 7 is not in the table: if it had been inserted, it would have found a home on the second probe if not before.

(c) *Why not $F(-) = 5$?*
Suppose we wish to insert n. If $F(n)$ were 5, the probe function $g(n, i)$ would only take two distinct values as we vary i. If these two slots are already taken, our probing will continue forever, and we'll never discover a free slot for n, even if plenty of free slots exist.

3. (a) *Red-black tree insertions*

- insert(5): the 5 replaces the leftmost leaf: it is coloured red and gets two trivial black leaves. The red-uncle rule doesn't apply, but we are in the 'endgame scenario' where black node 14 essentially have 4 black offspring. We rebalance this subtree, so that 8 (black) is where 14 was. Left child is 5 (red), right child is 14 (red), and each of these has two trivial black leaves.
- insert(17): we add 17 (red) as the right child of 14. Since 14 and 5 are red, the red-uncle rule applies and the black on 8 is pushed down to 5 and 14; 8 itself turns red. Now since 23 and 71 are red, the red-uncle rule applies again and we push the black on 47 down to 23 and 71. 47 turns red. But now the red has reached the root node, so we may simply turn it black again, adding 1 to the black-length of *all* paths through the tree.

(b) *Red-black tree deletions*

- delete(76): This is easy. We remove the node 76 and its two children, replacing it with a trivial node. Since the removed node was red, there is no floating black, and we are done.
- delete(59): We remove 59 and its children, replacing it with a trivial node. Since the removed node was black, there is now a floating black on this trivial node. Fortunately, having already removed 76, we are in the situation of the wandering black rule, and we may move the floating black up to 71, turning 87 red. But now the floating black is on the red node 71, so it takes up residence there turning 71 black, and all is well.

(c) *How would we use red-black trees to implement lists?*

The main point is that each internal node should store, in addition to its key value, the number of list elements in the subtree rooted at that node. This will enable us to tell, when computing e.g. $get(10)$, whether the element at position 10 in the list is in the left or the right subtree, or indeed at the current node. This count information will of course need to be updated every time the tree changes – but since **insert** and **delete** only make changes to $O(\lg n)$ subtrees (those on the path from the point of change back to the root), this can be done in $O(\lg n)$ time.