Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 3: Data structures for lists and sets

This sheet covers material from Lectures 7, 8 and 9. The question marked $\star$ may be more challenging than the rest.

1. In lectures we discussed an implementation of lists via *extensible arrays*, whose capacity could be expanded when necessary by some factor r (e.g. 1.1).

   Programmer P is concerned that this policy may be wasteful on memory when lists are large, and suggests the following alternative: use an array with initial capacity 1000, and increase by adding 100 each time an expansion is needed.

   (a) For this scheme, calculate the total number of copyings involved in performing a sequence of 5000 *append* operations, beginning from an empty list. Recall from Lecture 6 that *append* adds a single element at the right-hand end of the list. By a 'copying' we mean the copying of a single element from one array to another, i.e. an execution of an instruction of the form 'B[i] = A[i]'.

   (b) More generally, give a precise formula for the number of copyings required by a sequence of n *append*s (beginning from an empty list), with a clear explanation of how this formula is derived. What is the asymptotic growth rate of this function? What is the amortized cost of a single *append* over a long sequence of n *appends*?

   (c) Programmer Q wants to use the original 'factor r' approach to expansion with $r = 2$, but suggests adding a *contraction* policy: if the current array capacity is 2000 or more, and the proportion of the array in use dips below 0.5, move the contents to an array of half the size, freeing up some memory.

   What bad behaviour might result from this policy? Suggest a better contraction policy.

2. (a) Suppose we implement sets of non-negative integers via a hash table of size 10, with buckets stored as linked lists outside the table itself. Our hash function is $h(n) = n \bmod 10$ (this is chosen to make the arithmetic easy).

   With the help of pictures, show what happens when we perform the following sequence of set operations (starting from a table representing the empty set).

   insert(47), insert(93), insert(17), insert(143), insert(777),
   contains(93), contains(7)

   (b) Alternatively, we can represent such sets by storing the elements within the hash table itself, using *probing* to resolve collisions. Since the elements of our sets are non-negative integers, we may use -1 to indicate a blank entry in the table. We use the following hash-probe function (again designed to be easy to calculate):

$$g(n,i) \;=\; (n + (i \times F(n)) \bmod 10 \;,$$

where F is some magical function such that

$$F(47) = 1, \quad F(93) = 3, \quad F(17) = 3, \quad F(143) = 7, \quad F(777) = 3, \quad F(7) = 1$$

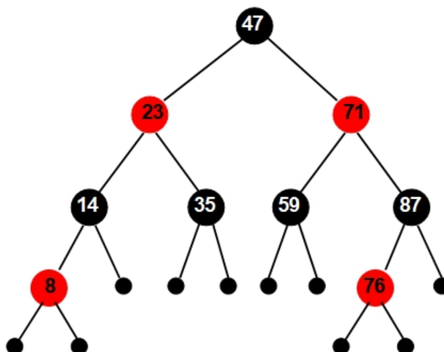
For example, the successive hash-probe values for 143 are

$$g(143,0) = 3, \quad g(143,1) = 0, \quad g(143,2) = 7, \quad \ldots$$

Using this scheme, work through what happens when the sequence of set operations from part (a) is performed.

(c) What could go wrong if for some n we had $F(n) = 5$?

3. (a) Consider the following red-black tree used to represent a set of integers. (The black nodes have their integer written in white, the red nodes have it in black.)



Work through what happens when the following are performed in sequence:

insert(5), insert(17)

(b) Starting afresh from the tree depicted above, work through what happens when we do:

delete(76), delete(59)

(c) ⋆ We've discussed how red-black trees can be used to implement sets or dictionaries. Explain how they could also be used to implement *lists* (a.k.a. *vectors*), with operations $get(i)$, $set(i,x)$, $insert(i,x)$, $delete(i)$ and $length()$, all with worst-case time $O(\lg n)$ (where n is the length of the list). What extra information needs to be stored on nodes to facilitate this?

(Note: Your scheme should support the representation of arbitrary lists, not just sorted ones. However, it need not support an efficient membership test.)