Solutions to Review Questions for Midterm

CS409 - Spring 2000

Please let me know via email (chew@cs.cornell.edu) if you discover mistakes in my solutions. Past experience indicates that I may have made some.

Review Questions

1. Fill in the table below with the *expected* time for each operation. Use big-O notation. The operations are insert (place a new item in the data structure), member (test if a given item is in the data structure), getMin (return the value of the minimum item in the data structure and delete it from the data structure), and successor (given an item, return the successor of that item).

Data Structure	insert	member	getMin	successor
sorted array	*O(n)*	*O(log n)*	*O(1)*	*O(log n)*
unsorted array	*O(1)*	*O(n)*	*O(n)*	*O(n)*
balanced tree (red-black tree)	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*
hashtable	*O(1)**	*O(1)**	*O(n)*	*O(n)*
sorted linked list	*O(n)*	*O(n)*	*O(1)*	*O(n)*
unsorted linked list	*O(1)*	*O(n)*	*O(n)*	*O(n)*

**I am assuming that table doubling is being used for the Hashtable.*

2. Short Answer.
   a. Where is the smallest element in a red-black tree? In a 234-tree?
      In both cases, it's the leftmost item is the node you reach by starting at the root and always moving left.
   b. When the subject is balanced trees, what does *rotation* mean?
      This is the process of changing the root of a subtree. One of the children of this node becomes the new root of the subtree. The order of the data is not affected by this operation.
   c. What is *path compression* in the union/find algorithm?
      When doing the find operation, each node passed on the way to the root is made to point at the root.
   d. How long does it take to insert a new element into a heap? To return the smallest thing in a min-heap? To delete the smallest thing in a min-heap? To find the largest thing in a min-heap?
      insert = O(log n); findMin = O(1); deleteMin = O(log n). Finding the largest thing in a min-heap is expensive: O(n).
3. The following picture represents a 234-tree.

```
             24
          /      \
   2 , 22            30 , 40 , 48
 /   |     \      /    /      \      \
1  5,11,19  23  29  32,36  41,42,43  50
```

   a. Draw an equivalent red-black-tree.
      I'll use parens () to indicate a red node. The 3-nodes (2,22) and (32,36) can be represented in more than one way.

```
           24
       /        \
    22             40
   /  \         /    \
 (2)   23  (30)        (48)
 / \       /  \        /  \
1   11   29    36    42    50
   /  \       /     /  \
 (5)  (19) (32)  (41)  (43)
```

   b. Draw a picture of the 234-tree that results from inserting 31 into the *original* 234-tree.

```
             24      ,     40
          /         |         \
   2 , 22           30              48
 /   |     \      /    \          /    \
1  5,11,19  23  29  31,32,36  41,42,43  50
```

4. For each of the following problems, choose the best of the listed data structures and explain why your choice is best. Assume that the data set is going to be large unless otherwise indicated. Where several operations are listed, you should assume, unless stated otherwise that the operations occur with about equal frequency.
   a. Operations are Insert, DeleteMax, and DeleteMin.
      balanced tree or *sorted doubly-linked list*
      The balanced tree is better since all operations take O(log n) time. The sorted doubly-linked list is O(1) for DeleteMax and DeleteMin, but Insert is O(n); thus, the average time per operation is O(n).
   b. Operations are Insert and FindMedian. (The median is the item m such that half the items are less than m and half are greater than m.)

red-black trees or *sorted array*
You can use two red-black trees plus an additional variable to hold the median, one red-black tree for items less than the median and one for items greater than the median. When you insert, you can keep track of the median by moving items from one tree to the other. With this scheme, Insert takes O(log n) time and FindMedian take O(1) time. The sorted array takes O(n) time for Insert.

c. You have a dictionary containing the keywords of the Pascal programming language.
ordered array or *red-black tree*
In this situation the ordered array is best. Both data structures take time O(log n) to find an item. An ordered array takes longer to insert or delete, but we don't expect to be creating or destroying keywords, so there shouldn't be any insertion or deletion. The ordered array is simpler to program and takes less space.

d. You have a dictionary that can contain anywhere from 100 to 10,000 words.
unordered linked-list or *red-black tree*
The red/black tree is better, since the operations require O(n) time for the linked-list and O(log n) time for the red/black tree. For 10,000 words this could certainly be significant.

e. You have a large set of integers with operations insert, findMax, and deleteMax.
unordered array or *Hashtable*
Neither data structure is good for this problem. An unordered array is slightly better since it has less overhead and it's easier to program.

5. You have a hashtable of size m=11 and a (not very good) hash function h:

   h(x) = (sum of the values of the first and last letters of x) mod m

   where the value of a letter is its position in the alphabet (e.g., value(a)=1, value(b)=2, etc.). Here are some precomputed hash values:

word	ape	bat	bird	carp	dog	hare	ibex	mud	koala	stork
h	6	0	6	7	0	2	0	6	1	8

   Draw a picture of the resulting hashtable (using chaining) after inserting, in order, the following words: *ibex, hare, ape, bat, koala, mud, dog, carp, stork*. Which cells are looked at when trying to find *bird*.

chaining			
0	ibex	bat	dog
1	koala		
2	hare		
3			
4			
5			
6	ape	mud	
7	carp		
8	stork		
9			
10			

6. Suppose you are given the following information about a hashtable.

Space Available (in words)	10000
Words per Item	7
Words per Link	1
Number of Items	1000
Proportion Successful Searches	1/3

   What is the expected number of probes for a search operation when hashing with chaining is used?

   We need to compute the value of the load factor, alpha. The items themselves take up 7000 words and their links would use another 1000; this leaves 2000 words for the hashtable (the table of list headers). Thus the load factor alpha = (number of items)/(number of table positions) is 1000/2000 = 0.5

   *For chaining, the expected number of probes for an unsuccessful search is alpha or about 0.5. The expected time for a successful search is 1+alpha/2 or about 1.25. Using the data on the proportion of successful searches, we conclude that the expected number of probes for a search is (2*0.5 + 1*1.25)/3 = 0.75.*

7. Consider a tree implementation for the union/find problem in which the smaller set is merged to the larger and the name of the set is taken to be the element stored at the root of the tree. Suppose we initialize our sets so that each integer between 1 and 8 (inclusve) is contained within its own set.
   a. Give a sequence of seven unions that produces a tree whose height is as large as possible. Your answer should be a sequence of procedure calls of the form `Union(a,b)` where a and b are integers between 1 and 8. Draw the resulting tree.
   There are lots of possible sequences that would work here. One of them is: U(1,2), U(3,4), U(5,6), U(7,8), U(1,3), U(5,7), U(1,5).
   b. Give a sequence of seven unions, on the original eight sets, that produces a tree of minimum height. Draw the resulting tree.
   Again, there are many correct answers. Here's one: U(1,2), U(1,3), U(1,4), U(1,5), U(1,6), U(1,7), U(1,8).
   c. Explain why both the min- and max-height trees use *seven* unions.

max height	**min height**

```
   1
  /|\\
2 3 5
 /  | \\
4   6  7
       /
     8
```
	```
	      1

	 /| | | | |\\
	2 3 4 5 6 7 8
	```

We are building trees on 8 nodes and all such trees have 7 edges. (You should be able to prove by induction that all n-node trees have n-1 edges.) Each Union operation creates a single edge, so to build an 8-node tree, we always need at least 7 Union operations.

8. The following questions refer to an implementation of an ADT with operation Insert, Delete, and isMember. Note that these are the *only* operations, so for this problem it is not an advantage for a data structure to allow more operations.
   a. Under what conditions would you use a red-black tree instead of hashing with chaining?
   If I needed a guaranteed worst-case time of O(log n) for Dictionary operations. The good results for hashing are only expected time.
   b. Under what conditions would you use an unordered array instead of a red-black tree?
   If I were short of space or if the set to be stored is small.
   c. Under what conditions would you use a binary search tree instead of a heap?
   Always. A BST does each of these operations in O(log n) expected time. A heap does these operation no better than using an unordered list.
   d. What implementation would you use to get the best expected time for a search?
   Hashing with table doubling. Expected search time is O(1).
   e. What implementation would you use to get the best worst-case time for a search?
   Any balanced tree scheme (234-tree or red-black tree) or even a sorted array. These all provide worst-case time O(log n) for a search.

9.

234 Tree (Dictionary)	**Max-Heap (Priority Queue)**

```
      (3 , 5)
     /   |   \\
(1,2)   (4)   (6)
```
	```
	     6
	    /   \\
	  3      4
	 / \\
	1   2
	```

For each of the preceding trees, find a sequence of appropriate operations that will produce it starting from an empty tree.
There are lots of sequences that will do this. Here's one possible sequence for the 234-tree: Insert 2,3,4,5,6,7,1 then Delete 7. Here's one that doesn't use Delete: Insert 4,5,6,3,2,1. And here's one for the Max-Heap: Insert 6,3,4,1,2.

10. We know that a sequence of n union/find operations using weighted union and path compression takes time O(n alpha(n)) where alpha(n) grows extremely slowly. What if all the union operations are done first? Show that a sequence of n unions followed by m finds takes time O(n + m). [Hint: The time for a find operation is proportional to the number of links that it changes. How many links are there?]

First recall that each Union operation takes time O(1); thus, all we have to show is that not too much time is spent on Finds. Observe that each edge of a tree has to be created by a Union operation; thus, the total number of tree edges is O(n). Suppose we want to count only edges that are deep in a tree, i.e., just those edges that are not edges from the root. There are O(n) such deep edges.

A Find operation on item i is slow only when i is deep in a tree; then it takes time proportional to the depth of i. The Find operation also converts a bunch of deep edges into root edges. As a matter of fact, the Find operation takes time proportional to a constant plus the number of deep edges converted to root edges. Since there aren't very many deep edges (just O(n) of them), the total time spent on Find operations is at most O((number of Find operations)+(number of edges converted from deep edges to root edges)) or O(m+n).