Topic 4, Basic Set ADT: Solutions

Copyright (c) 2016 Daniel D. Suthers. All rights reserved. These solution notes may only be used by students in ICS 311 Spring 2016 at the University of Hawaii.

1. (3 pts) For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic set operation listed given a list of length *n*?

- ***k* is the key (set element) and *p* a position in the data structure**
- **Assume that keys are unique.**
- **Search returns a position.** We don't have to search for the item if we have a position p for it.
- Sorted lists are sorted in **ascending order** of the relation "<"
- Predecessor, successor, minimum and maximum are **with respect to ordering of keys in the set under "<", *not* necessarily the ordering of the list data structure.**

Grading: the shaded cells were already filled in. 0.1 point will be deducted for each unshaded boldface cell that is wrong. There are 32 such cells, but we stop at 3 points.

	Unsorted, Singly Linked (no tail pointer)	**Sorted, Singly Linked** (no tail pointer)	**Unsorted, Doubly Linked, Sentinel and Tail pointer**	**Sorted, Doubly Linked, Sentinel and Tail pointer**
minimum()	**Θ(n)**	**Θ(1)**	**Θ(n)**	**Θ(1)**
maximum()	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(1)**
Using key (set element) k:				
insert(k)	Θ(1)	Θ(n)	Θ(1)	Θ(n)
search(k)	Θ(n)	Θ(n)	Θ(n)	Θ(n)
delete(k)	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(n)**
successor(k)	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(n)**
predecessor(k)	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(n)**
Using position p (which references a linked list cell containing a key):				
delete(p)	**Θ(n)**	**Θ(n)**	**Θ(1)**	**Θ(1)**
successor(p)	**Θ(n)**	**Θ(1)**	**Θ(n)**	**Θ(1)**
predecessor(p)	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(1)**

Justifications (not required; just to explain the above):

Search for a key k under uniform distribution requires n/2 comparisons on average, but this is Θ(n).

minimum(): Must search, except when sorted and you have a pointer to the beginning of the sorted list.

maximum(): Must search, except when sorted and you have a pointer to the end of the sorted list: only DLL is Θ(1).

insert(k): If unsorted can put at beginning of list; otherwise must find position.

search(k): In all cases you have to search the list to find the list element containing the key. Being sorted does not speed this up with a linked list.

delete(k): In all cases you have to search the list to find the list element containing the key to delete.

successor(k): In all cases you have to search the list to find the list element containing the key of interest. Then you can take one more step to find the successor.

predecessor(k): In all cases you have to search the list to find the list element containing the key of interest. Along the way you can keep track of the prior element, or use the prev pointer of DLL when found.

delete(p): Even if you have the list cell to delete you still have to search for its predecessor in a SLL as there is no prev pointer. DLL already has access to both neighbors in the list so they can be redirected around the deleted cell.

successor(p): Both SLL and DLL have next links: only when sorted is this the successor. Otherwise have to search the entire list to find the next larger element than whatever is stored at p.

predecessor(p): Only the DLL has prev links: only when sorted is this the predecessor. Otherwise have to search the entire list to find the next smaller element than whatever is stored at p.

2. Printing Binary Trees (2 pts)
(a) Write a Θ(n)-time recursive procedure that, given an n-node binary tree, prints out the key of each node of the tree in **preorder.** Assume that trees consist of vertices of class **`TreeNode`** with instance variables **`parent`**, **`left`**, **`right`**, and **`key`.** Your recursive procedure takes a **`TreeNode`** as its argument (the root of the tree or subtree being considered by the recursive

call).

```
printTreeNodes(TreeNode root)
1  if (root != null)
2       print (root.key)
3       printTreeNodes(root.left)
4       printTreeNodes(root.right)
```

(b) Prove that your algorithm is Θ(n)-time. If you run into difficulties ask for a hint.

A single call to printTreeNodes processes one node of the tree. Each of the lines 1-4 in this procedure takes constant time, so it is Θ(1) to process one node. Assuming that the algorithm is correct (would require a separate proof), each node is printed and hence visited exactly once. There are n nodes. So Θ(1*n) = Θ(n).

If you tried to write a recurrence relation you would have encountered the difficulty that you don't know how to divide the problem, as a binary tree is not necessarily balanced. For example, you cannot assume that it is 2T((n-1)/2) + Θ(1). It might be T(1) + T(n-2), or T(2) + T(n-3) or … etc. But all nodes are visited so we can reason in aggregate as done above.

Extra Credit (2 pts)

3. Which of the Θ results in the ADT runtime table must be changed if we do not assume unique keys, and why? (Describe the worst case scenario. 1 pt)

In the Position section, successor for sorted SLL, successor for sorted DLL, and predecessor for sorted DLL must be changed from Θ(1) to Θ(n). Suppose we give these a position p and ask for successor, and the key at p is some k. Without uniqueness, the keys at the next O(n) positions could also be k, and we'd have to skip over them before we finally find the first key x > k.

Students also pointed out that delete(k) becomes problematic. Do we delete the first k we find, or all of them?

Note that the insert code we are working with does not guarantee uniqueness. That would have to be done by the next layer of code, which would check for presence with search before inserting.

4. Modify `printTreeNodes` to print out the nodes in in-order (0.5pts)**:**

Move line 2 between 3 and 4.

5 Modify `printTreeNodes` to print out the nodes in postorder (0.5pts):

Move line 2 to the end.

If you finish early, you may discuss these potential homework problems:

- How would you rewrite your printTreeNodes procedure to use iteration (for or while loop) and a stack rather than recursion?
- How would you modify your printTreeNodes procedure to print out the nodes of an N-ary tree using the left-child right-sibling representation in arbitrary order?