Sample Solution for CS3323 Fall 2006 Assignment 4 **(61 marks)**

Due Monday, Nov. 13, by 5pm.

- Assignments should be handed in by placing them in the CS3323 bin on E level of Gillin Hall.

---

1. Design algorithms for performing the following operations on a binary tree T of size n, and analyze their worst-case running time. Your algorithms should avoid performing traversals of the entire tree.

   (a) **(8 marks)** preorderNext(v): return the node visited after node v in a preorder traversal of T.

   Solution: For the preorder traversal, nodes are visited before their left and right subtrees, so the following cases apply:

   i. if v is an internal node, the next node is its left child

   ii. if v is an external node, locate the first ancestor (call it z) of v such that v is in z's left subtree; the next node is z's right child if z exists, and *null* if z does not exist (another valid thing would be to return the root of the tree since it is the first vertex in the preorder traversal while v was the last).

   Algorithm preorderNext(v):
   Input: the current node v
   Output: the next node in the preorder traversal of T

```
if T.isInternal(v) then                          { case 1 }
  return T.leftChild(v)
else                                { case 2 }
  if T.isRoot(v) then                            { z not found }
    return null
  else
    current ← v
    z ← T.parent(current)
    while current = T.rightChild(z) do
          if T.isRoot(z) then                          { z not found }
            return null
```

```
        else
            current ← z
            z ← T.parent(current)
    return T.rightChild(z)
```

(b) **(8 marks)** inorderNext(v): return the node visited after node v in an inorder traversal of T.

Solution: In the inorder traversal, nodes are visited between the nodes of their left and right subtrees. The following cases apply:

i. if v is an internal node, the next node is the leftmost external node in v's right subtree

ii. if v is an external node, the next node is the first ancestor (call it z) of v such that v is in z's left subtree if z exists, and *null* (or the leftmost external node of the tree) if z does not exist (ie, v is the last node on the tour; the leftmost external node is the first).

Algorithm inorderNext(v):

```
Input: the current node v
Output: the next node in the inorder traversal of T

if T.isInternal(v) then                          { case 1 }
  current ← v
  while (T.isInternal(current)) do
        current ← T.leftChild(current)
  return current
else                          { case 2 }
    if T.isRoot(v) then                          { z not found }
      return null
    else
        current ← v
        z ← T.parent(current)
        while current = T.rightChild(z) do
              ifT.isRoot(z) then                          { z not found}
                return null
              else
                current ← z
                z ← T.parent(current)
        return z
```

Worst-case analysis for 2(b) and 2(c): the following picture has the examples for which worst case running times are realized. For part (c) it is when the tree grows left only. In this case, to find inorderNext(T.root()), the algorithm would have to traverse from the root down the tree to its leftmost node for $(n-1)/2$ hops (iterations of the while loop), so this gives us $O(n)$ running time. For part (b), the worst case is when the tree grows right only. Then to find preorderNext(v) where v is the rightmost node in the tree takes $(n-1)/2$ hops up the tree (at which point the algorithm realizes that v was the last vertex in the traversal), so it is also $O(n)$.

2. **(10 marks)** Let T be a binary tree with n nodes. It is realized with an implementation of the Binary Tree ADT that has $O(1)$ running time for all methods except *positions*() and *elements*(), which have $O(n)$ running time. Give an $O(n)$ time algorithm that uses the methods of the Binary Tree ADT to visit the nodes of T by ***level order traversal***. ***level order traversal*** visits the nodes in order of increasing depth, visiting the nodes at a given depth from left-to-right. Assume the existence of an $O(1)$ time visit(v) method (it should get called once on each vertex of T during the execution of your algorithm).

Solution:

The level-order traversal visits the nodes in order of increasing depth, visiting the nodes at a given depth from left-to-right. The key is to note that if $v_1, v_2, \ldots, v_m$ are the nodes of level k (from left-to-right), then leftChild(v_1), rightChild(v_1), leftChild(v_2), rightChild(v_2), ..., leftChild(v_m), rightChild(v_m) are the nodes of level $k+1$ (also from left-to-right). As a result, the left-to-right list of vertices to be visited in level $k+1$ can be built while visiting those in level k.

Algorithm levelOrderTraversal(T):
Input: a binary tree T
Output: none

let Q be an empty queue
Q.enqueue(T.root())
while Q is not empty **do**
    current $\leftarrow$ Q.dequeue()
    visit(*current*)
    if T.isInternal(*current*) **then**
        Q.enqueue(T.leftChild(*current*))
        Q.enqueue(T.rightChild(*current*))

At each point, Q contains all of the nodes to the right of the current node on the current level, and all of the nodes to the left of the current node's children on the next level. The running time is $O(n)$ because each node is put into the queue and taken out of the queue exactly once.

3. (a) **(5 marks)** Insert into an initially empty binary search tree items with the following keys (in this order): 30, 40, 23, 58, 48, 26, 11, 13. Draw the tree after each insertion.

   (b) **(5 marks)** Remove from the binary search tree built from (a) the following keys (in this order): 13, 40, 23. Draw the tree after each removal.

4. **(10 marks)** Let T be a binary search tree, and let x be a key. Give an efficient algorithm for finding the smallest key y in T such that $y > x$. Note that x may or may not be in T. Explain why your algorithm has the running time it does.

   Solution: For each node v of a binary search tree, the keys in v's left subtree are less than v and the keys in v's right subtree are greater than v. Thus, at each node a choice can be made:

   - if $v \leq x$ then y is the smallest thing larger than x in v's right subtree (if $v = x$ then y is the smallest thing in the right subtree, but this is just a special case of y being in the right subtree and does not need to be treated separately)
   - if $v > x$ then y is the smaller of v and the smallest thing greater than x in v's left subtree (it is not necessary to consider v's right subtree because while everything there is larger than x, it is also larger than v)

   This leads to the following algorithm:

```
Algorithm findFloor(T,x):
        Input: a binary search tree T and a key x
        Output: the smallest y in T such that y > x if such a y exists, and ∞ otherwise
        current ← T.root()
        best ← null
        while (T.isInternal(current)) do
            if (current.key() ≤ x) then
                current ← T.rightChild(current)          { y is in right subtree }
            else
                best ← current.key()                     { current node is a candidate for y }
                current ← T.leftChild(current)           { y is in left subtree }
        return best
```

Note that when *best* is updated we can use *best* $\leftarrow$ *current*.key() instead of *best* $\leftarrow$ min(*best*, *current*.key()) because as soon as *best* is updated with the current node's value, we go to the left subtree and all of the keys in that subtree are less than the current node (and the new value of *best*).

The running time is $O(h)$, where h is the height of the tree, because in each iteration of the **while** loop, *current* advances one level down the tree. Note that in the worst case this is $O(n)$ and in the best case it is $O(\log n)$.

5. **(10 marks)** Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). Note that the keys do not need to be reported in sorted order. Your algorithm should run in $O(k)$ time, where k is the number of keys reported.

Solution: Note that if v is a node of T that is larger then x, then all the keys stored in its subtrees are larger then x. Therefore it is enough for us to scan from the root the nodes of T that are smaller then x. There are two basic ways to do this: a recursive one and an iterative one that uses an external data structure (either queue or stack).

Recursive algorithm has to be run as RecurFindSmaller(T, $T.root()$,x) to find all keys in T that are smaller then or equal to x.

Algorithm RecurFindSmaller(T, v, x)
Input: a heap T; a node of T, v; a query key x
Output: keys of in the subtree of T rooted in v that are smaller or equal to x
if (T.isInternal(v)) **then**
  if (v.key() $\leq x$) **then**
    report key v.key()
    RecurFindSmaller(T, T.leftChild(v), x)
    RecurFindSmaller(T, T.rightChild(v), x)

Iterative algorithm using a queue (or alternatively a stack if we replace enqueue and dequeue methods by push and pop methods correspondingly.

Algorithm IterFindSmaller(T, x)
Input: a heap T; a query key x
Output: keys of in the subtree of T rooted in v that are smaller or equal to x
Q is an empty queue
Q.enqueue(T.root())
while (! Q.isEmpty()) **do**

```
current ← Q.dequeue()
if (T.isInternal(current) and current.key() ≤ x) then
    report key current.key()
    Q.enqueue(T.leftChild(current))
    Q.enqueue(T.rightChild(current))
```

This algorithm runs in $O(k)$ for the following reason. As we traverse down the tree, each node with a key smaller or equal to x gets scanned exactly once. The only other nodes that get scanned are their children. Given k returned keys, their k corresponding nodes in the heap have $2k$ children. Despite the fact that some of their children are the nodes whose key is $\leq x$ also, this still gives us the desired upper bound: we end up scanning $3k$ nodes (k nodes corresponding to the reported keys, and $2k$ of their children), which is $O(k)$.

6. **(5 marks)** Illustrate the execution of the heap-sort algorithm on the following input sequence: $(2, 5, 16, 4, 10, 23, 39, 18, 26, 15)$. Show the contents of both the heap and the sequence at each step of the algorithm.

   solution:

   Sequence contents at each of the above steps:

   (a) $(5, 16, 4, 10, 23, 39, 18, 26, 15)$
   (b) $(16, 4, 10, 23, 39, 18, 26, 15)$
   (c) $(4, 10, 23, 39, 18, 26, 15)$
   (d) $(10, 23, 39, 18, 26, 15)$
   (e) $(23, 39, 18, 26, 15)$
   (f) $(39, 18, 26, 15)$
   (g) $(18, 26, 15)$
   (h) $(26, 15)$
   (i) (15)
   (j) $()$

   Now come the steps that refill the sequence in sorted order.

   Sequence contents during the last ten steps are:

   (a) (2)

(b) $(2, 4)$

(c) $(2, 4, 5)$

(d) $(2, 4, 5, 10)$

(e) $(2, 4, 5, 10, 15)$

(f) $(2, 4, 5, 10, 15, 16)$

(g) $(2, 4, 5, 10, 15, 16, 18)$

(h) $(2, 4, 5, 10, 15, 16, 18, 23)$

(i) $(2, 4, 5, 10, 15, 16, 18, 23, 26)$

(j) $(2, 4, 5, 10, 15, 16, 18, 23, 26, 35)$