Chapter 14

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 14.1-1

The call sequence is:
$OS-SELECT(T.root, 10)$
$OS-SELECT(T.root.left, 10)$
$OS-SELECT(T.root.left.right, 2)$
$OS-SELECT(T.root.left.right.left, 2)$
$OS-SELECT(T.root.left.right.left.right, 1)$
Then, we have that the node (with key 20) that is returned is $T.root.left.right.left.right$

Exercise 14.1-2

OS-RANK(T,x) operates as follows. r is set to 0 and y is set to x. On the first iteration of the while loop, y is set to the node with key 38. On the second iteration, r is increased to 2 and y is set to the node with key 30. On the third iteration, y is set to the node with key 41. On the fourth iteration r is increased to 15 and y is set to the node with key 26, the root. This breaks the while loop, and rank 15 is returned.

Exercise 14.1-3

See the algorithm OS-SELECT'

Exercise 14.1-4

See the algorithm RECURSIVE-OS-KEY-RANK.

Exercise 14.1-5

The desired result is OS-SELECT(T,OS-RANK(T,x)+i). This has runtime O(h), which by the properties of red black trees, is $O(\lg(n))$.

Exercise 14.1-6

First perform the usual BST insertion procedure on z, the node to be inserted. Then add 1 to the rank of every node on the path from the root to z

Algorithm 1 O

```
S-SELECT'(x,i)
  r = x.left.size
  while $i \neq r$ do
      if $i > r$ then
          x = x.right
          i = i-r
      else
          x=x.left
      end if
      r = x.left.size
  end while
  return x
```

Algorithm 2 RECURSIVE-OS-KEY-RANK(T,k)

```
1: if $T.root.key == k$ then
2:     Return $T.root.left.size + 1$
3: else if $T.root.key > k$ then
4:     Return $RECURSIVE - OS - KEY - RANK(T.root.left, k)$
5: else
6:     Return $RECURSIVE - OS - KEY - RANK(T.root.right, k) + T.root.left.size + 1$
7: end if
```

such that z is in the left subtree of that node. Since the added node is a leaf, it will have no subtrees so its rank will always be 1. When a left rotation is performed on x, its rank within its subtree will remain the same. The rank of $x.right$ will be increased by the rank of x, plus one. If we perform a right rotation on a node y, its rank will decrement by $y.left.rank + 1$. The rank of $y.left$ will remain unchanged. For deletion of z, decrement the rank of every node on the path from z to the root such that z is in the left subtree of that node. For any rotations, use the same rules as before.

Exercise 14.1-7

See the algorithm INV-COUNT(L). It does assume that all the elements of the list are distinct. To adapt it to the not neccesarily distinct case, each time that the we do the search, we should be selecting the element with that key that comes first in an inorder traversal.

Algorithm 3 INV-COUNT(L)

```
Construct an order statistic tree $T$ for all the elements in $L$
$t = -|L|$
for $i$ from 0 to $|L| - 1$ do
    $t = t + OS - RANK(T, SEARCH(T, L[i]))$
    remove the node corresponding to $L[i]$ from $T$
end for
return $t$
```

Exercise 14.1-8

Choose a point on the circle and assign it key value 1. The rank of any point will be its position when read in the sequence starting with point 1, and reading clockwise around the circle. Next, we label each point with a key. Reading clockwise around the circle, if a point's chordal companion has a key, assign it the same key. Otherwise, assign it the next lowest unused integer key value. Then use the algorithm given in the solution to problem 2-4 d to count the number of inversions (based on key value), with the caveat that an inversion from key i to key j (with $i > j$)doesn't count if the rank of the companion of i is smaller than the rank of j.

Exercise 14.2-1

Along all the nodes in an inorder traversal, add a prev and succ pointer. If we keep the head to be the first node, and, make it circularly linked, then this clearly allows for the four operations to work in constant time. We still need to be sure to maintain this linked list structure throughout all the tree modifications. Suppose we insert a node into the BST to be a left child, then we can insert it into this doubly linked list immediately before it's parent, which can be done in constant time. Similarly, if it is a right child, then we would insert it immediately after its parent. Deletion of the element is just the usual deletion in a linked list.

Exercise 14.2-2

Since the black height of a node depends only on the black height and color of its children, Theorem 14.1 implies that we can maintain the attribute without affecting the asymptotic performance of the other red-black tree operations. The same is not true for maintaining the depths of nodes. If we delete the root of a tree we could potentially have to update the depths of $O(n)$ nodes, making the DELETE operation asymptotically slower than before.

Exercise 14.2-3

After performing the rotate operation, starting at the deeper of the two nodes that were moved by the rotate, say x, set $x.f = x.left.f \otimes x.a \otimes x.right.f$. Then, do the same thing for the higher up node in the rotation. For size, instead set $x.size = x.left.size + x.right.size + 1$ and then do the same for the higher up node after the rotation.

Exercise 14.2-4

The following algorithm runs in $\Theta(m + \lg n)$ time. There could be as many as $O(\lg n)$ recursive calls to locate the smallest and largest elements necessary to print, and every other constant time call will print one of the m keys between a and b. We can't do better than this because there are m keys to output, and to find a key requires $\lg n$ time to search.

Algorithm 4 RB-ENUMERATE(x,a,b)

```
if $a \leq x.key \leq b$ then
    print $x$
end if
if $a \leq x.key$ and $x.left \neq NIL$ then
    $RB-ENUMERATE(x.left, a, b)$
end if
if $x.key \leq b$ and $x.right \neq NIL$ then
    $RB-ENUMERATE(x.right, a, b)$
end if
Return
```

Exercise 14.3-1

after rearranging the nodes, starting with the lower of the two nodes moved, set it's max attribute to be the maximum of its right endpoint and the the two max attributes of its children. Do the same for the higher up of the two moved nodes.

Exercise 14.3-2

Change the weak inequality on line 3 to be strict.

Exercise 14.3-3

Consider the usual interval search given, but, instead of breaking out of the loop asa soon as we have an overlap, we just keep track of the most recently seen overlap, and keep going in the loop until we hit $T.nil$. We then return the most recently seen overlap. We have that this is the overlapping interval with minimum left end point because the search always goes to the left it contains an overlapping interval, and the left children are the ones with smaller left endpoint.

Exercise 14.3-4

Algorithm 5 INTERVAL(T,i)

```
if $T.root$ overlaps $i$ then
    Print $T.root$
end if
if $T.root.left \neq T.nil$ and $T.root.left.max \geq i.low$ then
    INTERVAL(T.root.left,i)
end if
if $T.root.right \neq T.nil$ and $T.root.right.max \geq i.low$ and $T.root.right.key \leq i.high$ then
    INTERVAL(T.root.right,i)
end if
```

The algorithm examines each node at most twice and performs constant time checks so the runtime cannot exceed $O(n)$. If a recursive call is made on a branch of the tree, then that branch must contain an overlapping interval, so the runtime also cannot exceed $O(k \lg n)$ since the height is at most n and there are k intervals in the output list.

Exercise 14.3-5

We could modify the interval tree insertion procedure to, once you find a place to put the given interval, then we perform an insertion procedure based on the right hand endpoints. Then, to perform INTERVAL-SEARCH-EXACTLY(T,i) first perform a search for the left hand endpoint, then, perform a search for the right hand endpoint based on the BST rooted at this node for the right hand endpoint, stopping the search if we ever come across a element with a different left hand endpoint.

Exercise 14.3-6

Store the elements in a red-black tree, where the key value is the value of each number itself. The auxiliary attribute stored at a node x will be the min gap between elements in the subtree rooted at x, the maximum value contained in subtree rooted at x, and the minimum value contained in the subtree rooted at x. The min gap at a leaf will be ∞. Since we can determine the attributes of a node x using only the information about the key at x, and the attributes in $x.left$ and $x.right$, Theorem 14.1 implies that we can maintain the values in all nodes of the tree during insertion and deletion without asymptotically affecting their $O(\lg n)$ performance. For MIN-GAP, just check the min gap at the root, in constant time.

Exercise 14.3-7

Let L be the set of left coordinates of rectangles. Let R be the set of right coordinates of rectangles. Sort both of these sets in $O(n\lg(n))$ time. Then, we will have a pointer to L and a pointer to R. If the pointer to L is smaller, call interval search on T for the up-down interval corresponding to this left hand side. If it contains something that intersects the up-down bounds of this rectangle, there is an intersection, so stop. Otherwise add this interval to T and increment the pointer to L. If R is the smaller one, remove the up-down interval that that right hand side corresponds to and increment the pointer to R. Since all the interval tree operations used run in time $O(\lg(n))$ and we only call them at most $3n$ times, we have that the runtime is $O(n\lg(n))$.

Problem 14-1

a. Suppose we have a point of maximum overlap p. Then, as long as we imagine moving the point p but don't pass any of the endpoints of any of the intervals, then we won't be changing the number of intervals containing p. So, we just move it to the right until we hit the endpoint of some interval, then, we have a point of maximum overlap that is the endpoint of an interval.

b. We will present a simple solution to this problem that runs in time $O(n\lg(n))$ which doesn't augment a red-black tree even though that is what is suggested by the hint. Consider a list of elements so that each element has a integer $x.pos$ and a field that says whether it is a left endpoint or a right endpoint $x.dir = L$ or R. Then, sort this list on the pos attribute of each element. Then, run through the list with a running total of how many intervals you are currently in, subtracting one for each right endpoint and adding one for each left endpoint. Also keep track of the running max of these values, and the endpoint that has that value. Then, this point that attains the running max is what should be returned.

Problem 14-2

a. Create a circular doubly linked list. Continue to advance m places in the list (not counting the sentinel), followed by printing and deleting the current node, until the list is empty. Since m is a constant and we advance at most mn places, the runtime is $O(n)$.

b. Begin by creating an order statistic tree for the given elements, where rank starts at 0 and ends at $n-1$, the rank of a node is its position in the original order minus 1, and we store each element's value in an attribute $x.value$. Print the element with rank $m(n)-1$, then delete it from the tree. Then proceed as follows: If you've just printed the k^{th} value which had rank r, delete that node from the tree, and the $(k+1)^{st}$ value to be printed will have rank $r-1+m(n) \mod (n-k)$. Since deletion and lookup take $O(\lg n)$ and there are n nodes, the runtime is $O(n \lg n)$.