Assignment 4, CS 251, Feb 19, 2019.

Exercise 1. LEAST COMMON ANCESTOR IN RED-BLACK TREES. Write an algorithm that, given pointers to nodes u and v in a red-black tree, finds the least common ancestor in $O(\log n)$ time.

SOLUTION. Assume that KEY$[u] <$ KEY$[v]$. Observe the following. Let u^* be the first node on the path p from u to the root with KEY$[u^*] >$ KEY$[v]$ if it exists at all. Similarly, let v^* be the first node on the path q from v to the root with KEY$[v^*] <$ KEY$[u]$ if it exists at all. If neither exists, then the least common ancestor is the root. Otherwise, it is either the child of u^* on p or the child of v^* on q. That child can be identified by the condition that its key must be in the range [KEY$[u]$,KEY$[v]$].

Exercise 2. COMPLETE BINARY TREES AND RED-BLACK TREES. We are given a sorted array A with elements denoted by $A[1], \ldots, A[n]$.

(i) Write a linear time algorithm for making a complete binary search tree with these elements, where the complete tree is stored in the standard fashion in another array, B, i.e., $B[1]$ is the root, $B[2]$ and $B[3]$ are the children of the root, and so forth.

(ii) Write a linear time algorithm for making a complete binary search tree with these elements, where the complete tree is stored with pointers only.

(iii) Given the tree output in part (ii), fill in the color fields in all nodes to make it a proper red-black tree. (And this serves also as a proof that a complete binary tree can always be made into a red-black tree).

SOLUTION. PART (I). An inorder traversal of the complete binary tree will do the job. The inorder traversal can be implemented recursively with a counter for counting the number of nodes visited.

```
$r \leftarrow 0$ (a counter for the number of visited nodes)
INORDER(1) (where INORDER is defined below)

INORDER($i$)
if $i \leq n$, then INORDER($2i$)
                    $r \leftarrow r + 1$
                    B[$i$] $\leftarrow A[r]$
                    INORDER($2i + 1$)
```

PART (II). Here we can do the inorder traversal as above, but we need to construct the complete binary tree first. We will make use of the procedure CREATENODE that creates an empty cell, which we can populate with left, right, parent and key fields. It can be done recursively: MAKETREE(i), given below, creates a cell for node i (where i refers to the complete binary tree numbering), fixes the entries in the cell, and returns a pointer to the cell for node i.

```
MAKETREE($i$)
if $i > n$ then return nil
        else CREATENODE($x$)
             if $i = 1$ then $p[x] \leftarrow$ nil
             $z \leftarrow$MAKETREE($2i$)
             $y \leftarrow$MAKETREE($2i + 1$)
             left $[x] \leftarrow z$
             right $[x] \leftarrow y$
             if $z \neq$ nil, then $p[z] \leftarrow x$
             if $y \neq$ nil, then $p[y] \leftarrow x$
             return $x$
```

PART (III). We can make all nodes in the complete binary tree black, except for the nodes in the last level, which can safely be colored red. The ranks of all nodes in the last two levels will be one, and thus, node ranks increase by one per level above these last two levels. In other words, in the 2-3-4 tree view, all black nodes command 2-pods except the ones in the second-to-last level, which command 2, 3 or 4-pods. We can use the construct of part (ii) and should only add the appropriate colors. Let the (complete binary tree) index of the first node on the last level be $f(n)$. Then in the code of part (ii), compute $\phi \leftarrow f(n)$ once, outside the procedure MAKETREE. Just below CREATENODE(x), add

```
if $i < \phi$, then color $[x] \leftarrow$ black
else           color $[x] \leftarrow$ red
```

Furthermore, a small modification may be needed if external nodes are explicitly implemented. Every "nil" pointer in the solution of (ii) should be replaced by an appropriate external node cell pointer. This is not shown here.

Here is the code for $f(n)$, after noting that $f(n)$ is the largest power of two not exceeding n:

```
f(n)
$i \leftarrow 1$
repeat forever:  if $2i > n$ then return $i$ and halt
                           else $i \leftarrow 2i$
```

Exercise 3. SWEEPLINE ALGORITHMS. We are given n axis-aligned rectangles in the plane where each rectangle is of the form $[a,b] \times [c,d]$. Imagine that there is a light source in the northern direction, infinitely far away. Write an $O(n \log n)$ sweepline algorithm for finding all the rectangles that are partially illuminated by that light source.

SOLUTION. Here we first create $2n$ quadruples of the generic form (x,y,i,s), where x is an x-coordinate, y is a y-coordinate, i is the rectangle number (an integer between 1 and n) and s is a bit: 0 means "left endpoint", and 1 stands for "right endpoint". The i-th rectangle $[a,b] \times [c,d]$, for example, spawns the quadruples $(a,d,i,0)$ and $(b,d,i,1)$. Sort these quadruples by their first component, the x-coordinate, from small to large, and place them in a queue Q. We apply the sweepline algorithm and maintain a red-black tree of "live" rectangles. The keys are the second components, i.e., the y-coordinates. Live rectangles of maximal y-coordinate are illuminated. It is useful to maintain double pointers as well: each red-black tree node stores RECT $[\cdot]$, the rectangle it represents, and for the i-th rectangle (in an array, say), we keep a pointer to its node in the red-black tree, and call it REDBLACK$[i]$. We call the pointer to the root of the red-black tree t. Let MAXIMUM(t) return a pointer to the node in t of maximal key. The high level code looks like this:

```
makenull $(t)$
while $|Q| > 0$ do
   $(x,y,i,s) \leftarrow$ DEQUEUE$(Q)$
   if $s = 0$ then INSERT$((y,i),t)$
                   output RECT(MAXIMUM$(t)$)
              else DELETE(REDBLACK$[i]$, $t$)
                   output RECT(MAXIMUM$(t)$)
```

Exercise 4. THE SMALLEST LARGER INTEGER DATA STRUCTURE. Design an efficient data structure for the following abstract data type. The object is a subset S of $\{1,2,\ldots,n\}$, where n is fixed and known, and S is dynamic. The operations are as follows.

(i) ADD(i,S): adds i to S.

(ii) DELETE(i,S): deletes i from S.

(iii) MEMBER(i,S): returns true if $i \in S$.

(iv) NEAREST(i,S): returns the smallest $j \in S$ with $j > i$. It returns nil if such an integer does not exist.

SOLUTION. This is an exercise on augmented data structures. One solution would be to maintain a red-black tree with as keys the integers themselves. The NEAREST operation is nothing but the standard SUCCESSOR, which takes time $O(\text{height})$. It can be sped up to $O(1)$ time if we maintain a linked list of the sorted elements as seen in class.

A second solution not requiring a red-black tree is as follows. Assume that n is a power of two—if not, replace it by the next higher power of two, so $n = 2^k$ for an integer k. Then let the leaves of a complete binary tree, from left to right, represent the integers 1 to n. The path in the tree to leaf i is given by the length k binary expansion of $i - 1$. A leaf i has a key that is 1 if $i \in S$ and 0 else. Thus, ADD, DELETE, and MEMBER are clearly $O(\log n)$ time operations. For NEAREST, we augment the nodes of the tree with pointers to the smallest leaf with a "1" key in its subtree, and to nil if the subtree has no such leaves. Call this pointer MIN. Convince yourself that this parameter can be maintained in $O(\log n)$ time after ADD and DELETE. Then NEAREST(i, S) is easily found as well, after noting that in the standard complete binary tree implementation, the node number of i is $n - 1 + i$ (the tree has $n - 1$ internal nodes as it has n leaves and each internal node has degree two).

```
$(x, y) \leftarrow (n - 1 + i, \lfloor (n - 1 + i)/2 \rfloor)$ ($y$ is the parent of $x$)
while $y > 0$ do
  if $x = 2y$ (left child) then if MIN$(2y + 1) \neq$ nil
                                  then return MIN$[2y + 1]$ and halt
  $x \leftarrow y$
  $y \leftarrow \lfloor x/2 \rfloor$
return nil
```