INF102 FALL 2018

MANDATORY ASSIGNMENT 1

EXAMPLE SOLUTION

[1] QUICKSORT

(a)

0	1	2	3	4	5	6	7	8	9	i	j	Comment
[11,	12,	4,	13,	2,	5,	11,	5,	16,	14]	0	10	Initial array
[11,	12,	4,	13,	2,	5,	11,	5,	16,	14]	1	7	Scan left/right
[11,	5,	4,	13,	2,	5,	11,	12,	16,	14]	1	7	Exchange
[11,	5,	4,	13,	2,	5,	11,	12,	16,	14]	3	6	Scan left/right
[11,	5,	4,	11,	2,	5,	13,	12,	16,	14]	3	6	Exchange
[11,	5,	4,	11,	2,	5,	13,	12,	16,	14]	6	5	Scan left/right
[5,	5,	4,	11,	2,	11,	13,	12,	16,	14]	-	5	Final exchange
[5,	5,	4,	11,	2,	11,	13,	12,	16,	14]			Result

(b)

0	1	2	3	4	5	6	7	8	9	lb	pivPos	ub	Comment
[11,	12,	4,	13,	2,	5,	11,	5,	16,	14]				Initial array
[5,	5,	4,	11,	2,	11,	13,	12,	16,	14]	0	5	10	
[4,	2,	5,	11,	5,	11,	13,	12,	16,	14]	0	2	5	
[2,	4,	5,	11,	5,	11,	13,	12,	16,	14]	0	1	2	
[2,	4,	5,	11,	5,	11,	13,	12,	16,	14]	0	-	1	Size 1 subarray
[2,	4,	5,	11,	5,	11,	13,	12,	16,	14]	2	-	2	Size 0 subarray
[2,	4,	5,	5,	11,	11,	13,	12,	16,	14]	3	4	5	
[2,	4,	5,	5,	11,	11,	13,	12,	16,	14]	3	-	4	Size 1 subarray
[2,	4,	5,	5,	11,	11,	13,	12,	16,	14]	5	-	5	Size 0 subarray
[2,	4,	5,	5,	11,	11,	12,	13,	16,	14]	6	7	10	
[2,	4,	5,	5,	11,	11,	12,	13,	16,	14]	6	-	7	Size 1 subarray
[2,	4,	5,	5,	11,	11,	12,	13,	14,	16]	8	9	10	
[2,	4,	5,	5,	11,	11,	12,	13,	14,	16]	8	-	9	Size 1 subarray
[2,	4,	5,	5,	11,	11,	12,	13,	14,	16]	10	-	10	Size 0 subarray
[2,	4,	5,	5,	11,	11,	12,	13,	14,	16]				Sorted array

(c) Consider an array where all items have the same value. Then the scan-right step will move `j` all the way to left, until it reaches `lb`; hence the pivot position will always be the leftmost position, and we lose the balance between our two recursive calls, which is crucial for obtaining a linearithmic runtime. In the two recursive calls of quicksort, we end up with one empty list in one call, and one list of size n-1 in the other; so in total we end up with one call to quicksort for every subarray length 1, 2, … n, each call spending time proportional to that length. The total runtime hence becomes $\mathcal{O}(n^2)$ for such arrays.

(d) See IterativeQuick.java

[2] PRIORITY QUEUES

(a) Inserting S. Here given with trace:

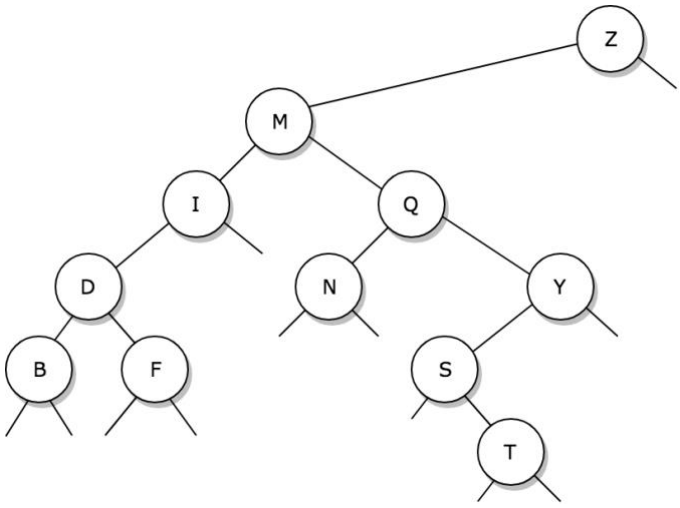
```
      0  1  2  3  4  5  6  7  8  9 10 11
[ null, T, P, R, N, H, O, A, E, I, G ]       // Original array
[ null, T, P, R, N, H, O, A, E, I, G, S ]    // Add at end
[ null, T, P, R, N, S, O, A, E, I, G, H ]    // Swap with parent: YES
[ null, T, S, R, N, P, O, A, E, I, G, H ]    // Swap with parent: YES
[ null, T, S, R, N, P, O, A, E, I, G, H ]    // Swap with parent: NO
[ null, T, S, R, N, P, O, A, E, I, G, H ]    // Final array
```

(b) We could certainly remember in a variable what the minimum/maximum value is for the peek() function, however, this is conditioned on that we never remove an element from our collection. Once we poll (and hence remove) we need to update the variable holding the value for the minimum and maximum. If a stack or queue is our underlying data structure, we would need to check every element in order to update our minimum/maximum, making the poll function much slower.

(c) See IndexMinPQ.java

[3] BINARY SEARCH TREE

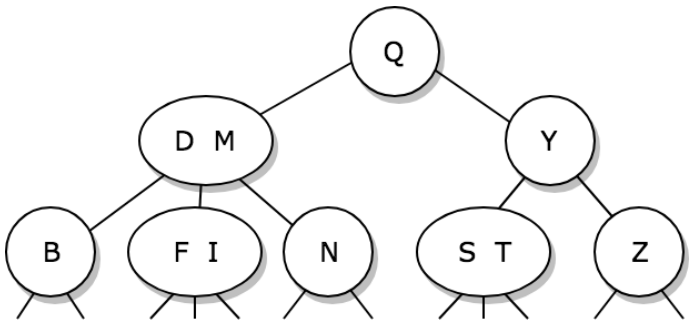
(a) Inserting `Z M Q N Y I D S B F T` in that order into an initially empty BST, yields this tree:



(b) See BinarySearchTree.java
(c) See BinarySearchTree.java
(d) See BSTDebugging.java

[4] BALANCED BINARY SEARCH TREES

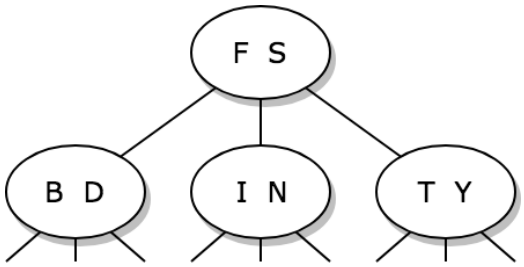
(a) Inserting `Z M Q N Y I D S B F T` in that order into an initially empty 2-3 tree, yields:



(b) We make an insertion order of `T F B S D I Y N` which yields a 2-3 tree of height 1:

`B F I S T D N Y`

This yields the following 2-3 tree:



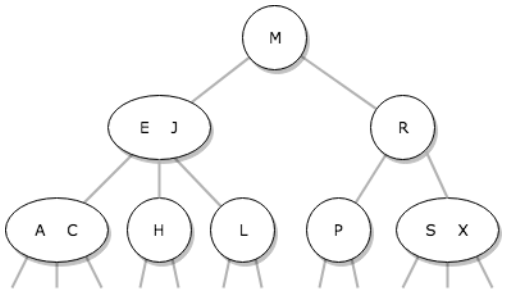
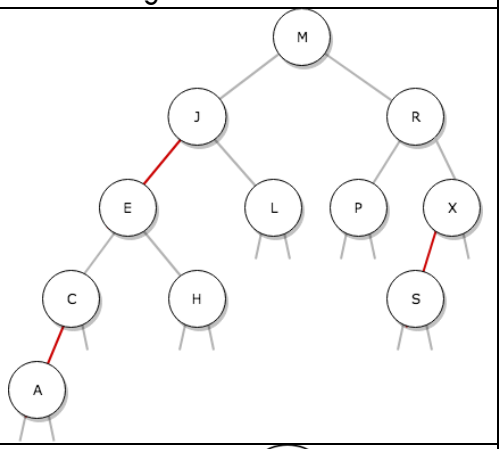
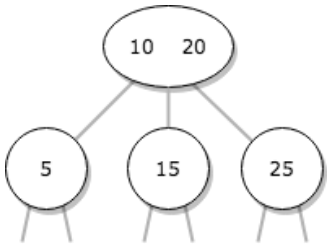
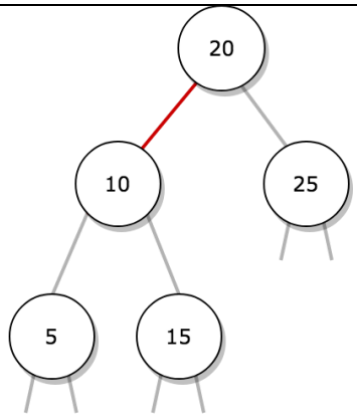
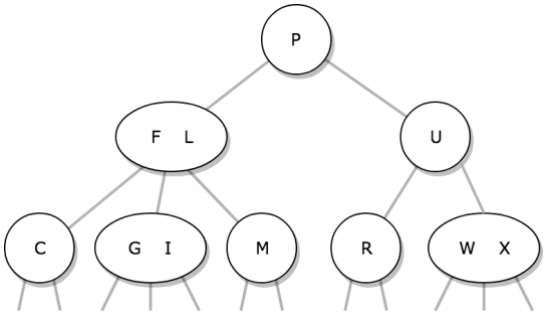
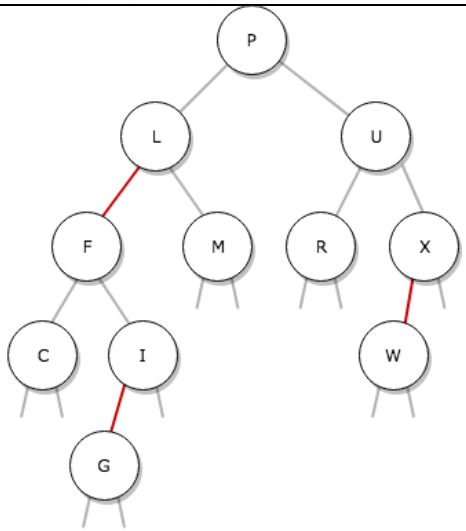
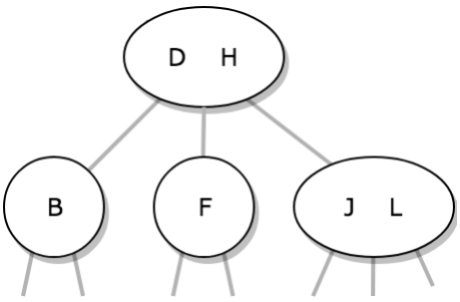
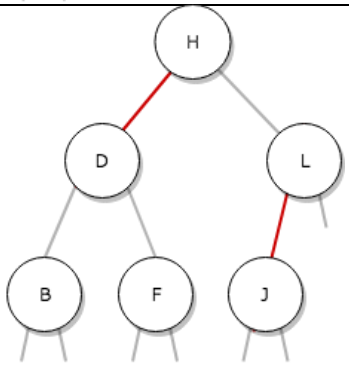
In fact, this is the only valid 2-3 tree of height 1 on the provided key set. In order for a permutation of the key set to form this 2-3 tree, the following must hold:

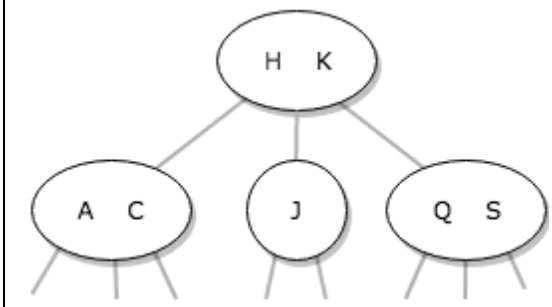
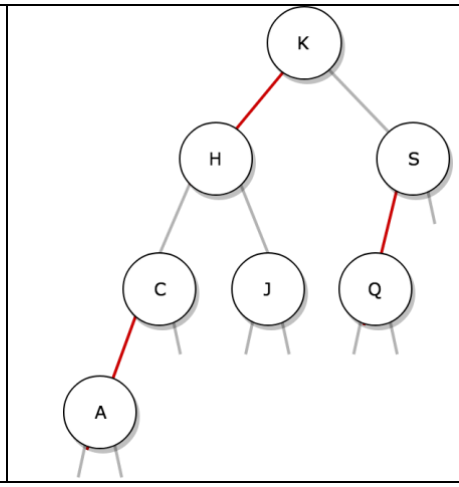
- The first three insertions must contain
  - An element from the root node; let this be element x
    - x naturally partitions the other elements; those before x in the sorted order, and those after x.
  - Two elements which are on opposite sides of x in the sorted order
- Let y be the other (non-x) element in the root node. Without loss of generality, we can assume y comes after x (otherwise, just reverse the forthcoming arguments).
- Among the elements after x in the sorted order, the first three to appear in the insertion order must
  - Contain y
  - Contain two elements on opposite sides of y

As long as the constraints above are met, any ordering will yield the depicted 2-3 tree.

(c) (iii) and (iv) are red-black BST's.
(i) is not balanced, (ii) is not ordered (F can't be to the left of E)

(d) See table on next page

Task	2-3 tree	Left-leaning red-black BST
i		
ii		
iii		
iv		

v		

(e) We give the order of right rotations (R) left rotations (L) and flips (F) when n is inserted

LRFFRFRFRFL

Breaking it down:

L	on subtree first rooted at m – becomes rooted at n
R	on subtree first rooted at o – becomes rooted at n
F	on subtree now rooted at n (after previous rotation)
F	on subtree rooted at l
R	on subtree first rooted at q – becomes rooted at p
F	on subtree now rooted at p
R	on subtree first rooted at s – becomes rooted at r
F	on subtree now rooted at r
R	on subtree first rooted at u – becomes rooted at t
F	on subtree now rooted at t
L	on subtree first rooted at j – becomes rooted at t

(f) The maximum depth of a left-leaning red-black tree with n nodes is $2 \cdot log_2(n)$. We first claim that it is not possible to exceed this depth, due to the correspondence with 2-3 trees: In a left-leaning red-black BST, the number of black edges from a leaf to the root is exactly the same as the distance from the leaf to the root in the corresponding 2-3 -tree. Since these have depth at most $\log_2(n)$ (in the worst-case of only two-nodes), we can bound the number of black edges between a leaf and the root to $\log_2(n)$. Because red edges are not allowed to be incident to each other, we know that at least half the edges on the path from a leaf to the root are black. Thus the claim holds.

We next show that it is actually possible to obtain a maximum depth arbitrarily close to $2 \cdot log_2(n)$. This can happen in a 2-3 tree where all nodes are two-nodes except the nodes following the very left-most path. As n grows towards infinity, the *proportion* of nodes which are three-nodes in such a tree tends towards 0 – hence, in the limit, the depth of such a special 2-3 tree approaches $\log_2(n)$ despite not all nodes being two-nodes. Now, let us go back to that leftmost path of three-nodes – represented as a left-leaning red-black tree, that path will be twice as long as the depth of the 2-3 tree (since each of the three-nodes is represented by two nodes in the left-leaning red-black tree).