INF102mandatory1 – CGJ008

Carl August Gjørsvik

[1] Quicksort

a)

	i	j	0	1	2	3	4	5	6	7	8	9
Initial values	0	10	11	12	4	13	2	5	11	5	16	14
Scan left, scan right	1	7	11	12	4	13	2	5	11	5	16	14
Exchange	1	7	11	5	4	13	2	5	11	12	16	14
Scan left, scan right	3	6	11	5	4	13	2	5	11	12	16	14
Exchange	3	6	11	5	4	11	2	5	13	12	16	14
Scan left, scan right	6	5	11	5	4	11	2	5	13	12	16	14
Final exchange	6	5	5	5	4	11	2	11	13	12	16	14
Result		5	5	5	4	11	2	11	13	12	16	14

b)

	lb	pivPos	ub	0	1	2	3	4	5	6	7	8	9
initial values →				11	12	4	13	2	5	11	5	16	14
	0	5	10	5	5	4	11	2	11	13	12	16	14
	0	2	5	4	2	5	11	5	11	13	12	16	14
	0	1	2	2	4	5	11	5	11	13	12	16	14
*	0		1	2	4	5	11	5	11	13	12	16	14
*	2		2	2	4	5	11	5	11	13	12	16	14
	3	4	5	2	4	5	5	11	11	13	12	16	14
*	3		4	2	4	5	5	11	11	13	12	16	14
*	5		5	2	4	5	5	11	11	13	12	16	14
	6	7	10	2	4	5	5	11	11	12	13	16	14
*	6		7	2	4	5	5	11	11	12	13	16	14
	8	9	10	2	4	5	5	11	11	12	13	14	16
*	8		9	2	4	5	5	11	11	12	13	14	16
*	10		10	2	4	5	5	11	11	12	13	14	16

* lb +1 >= ub → no partition for subarrays of size < 2

c)

The problem occurs when partition() is called with an array or sub-array comprised of n equal values. Using the normal implementation, incrementing i and j until they point at respectively greater or equal, and lesser or equal values, values will be swapped until i and j meet in the middle. The result is a lot of swapping, but only O$(\log n)$ recursive method-calls.

In the case where i and j "look" for elements strictly greater/lesser, i will scan to the end of the array and j will scan to the first element of the array. Quicksort will then make recursive calls with the sub-arrays created by a pivot at the left-most position, always with one sub-array of length 1 and another with length n-1, n-2, n-3… etc, effectively creating O(n) recursive calls.

[2] Priority Queues

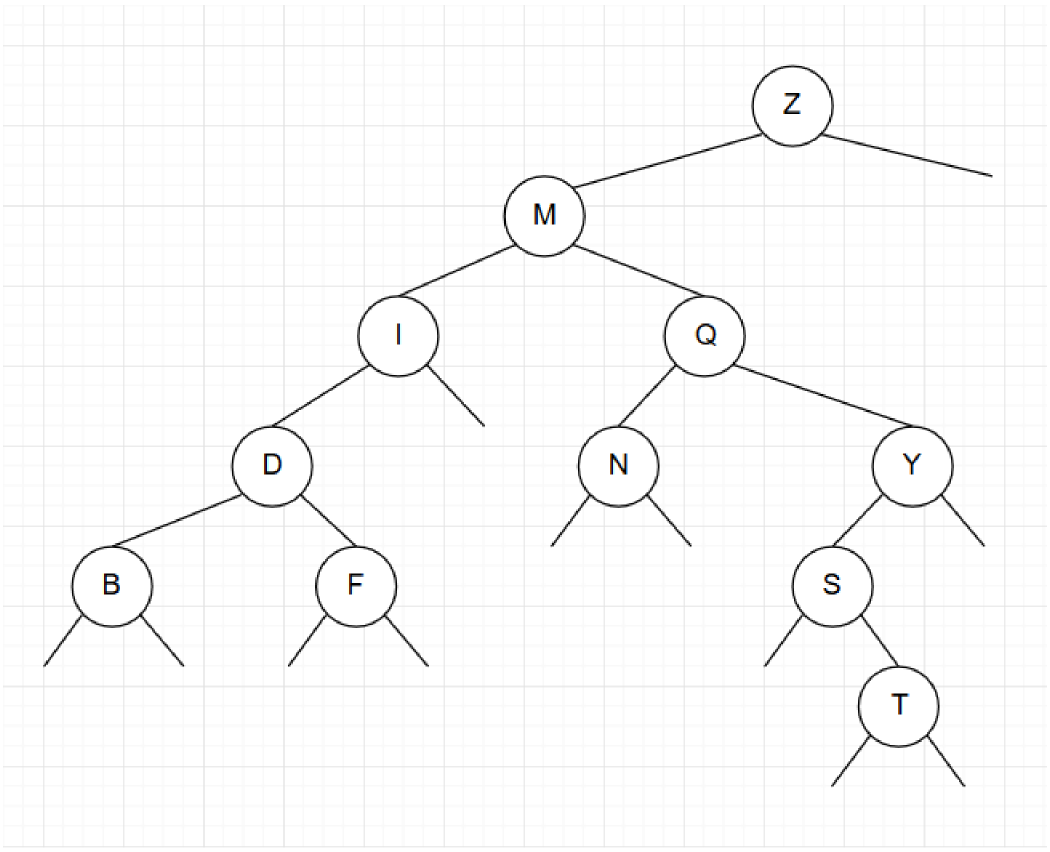
a) [null, T, S, R, N, P, O, A, E, I, G, H]

b)

Finding and deleting values in a stack/queue would be significantly slower. To remove a value, we would have to pop/poll until it is found, which could take O(n) time. Additionally, to update the variable keeping track of minimum/maximum, a search is needed.

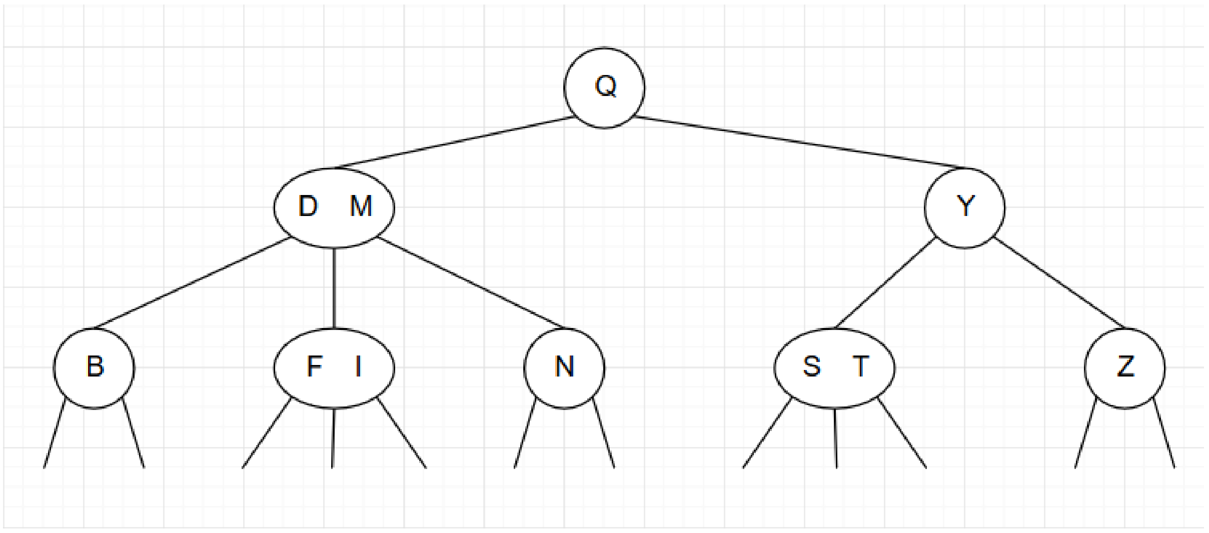
[3] Binary Search Tree

a)

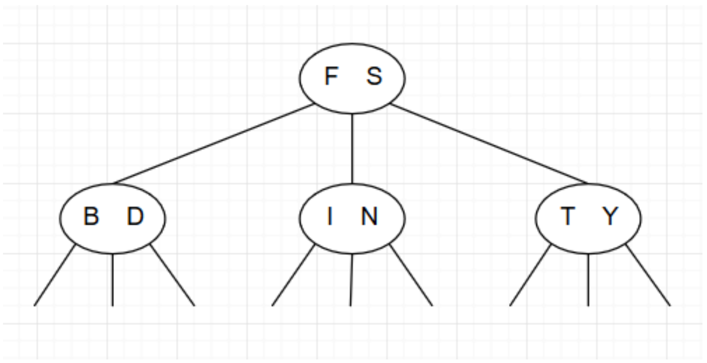


[4] Balanced Binary Search Trees

a)



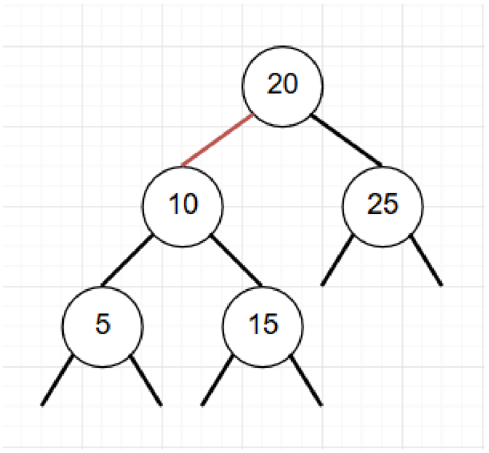
b) Insertion order: B F Y I S D N T



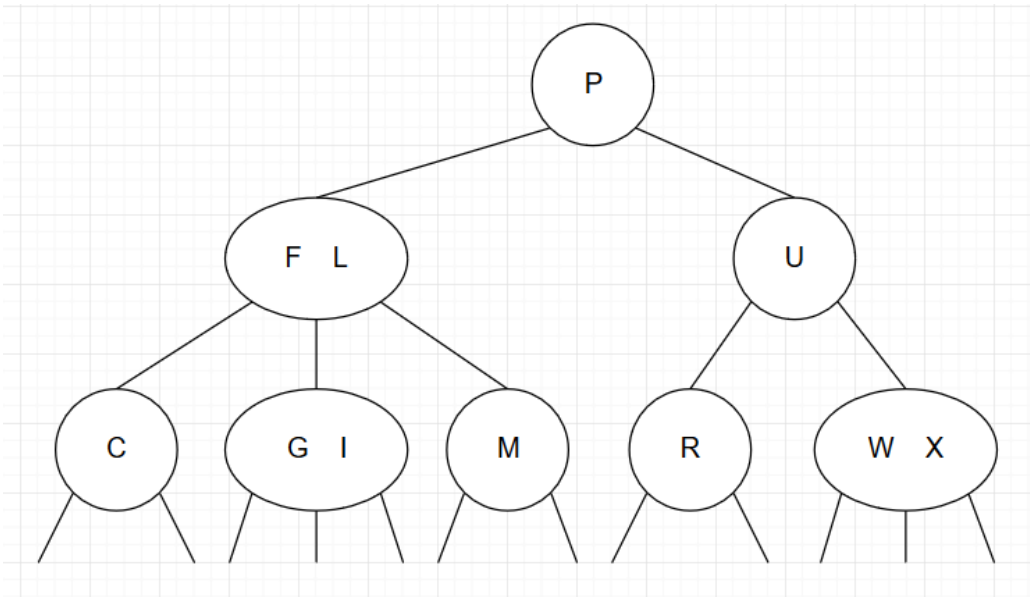
c) (iii) and (iv) are Red-Black BSTs

d)

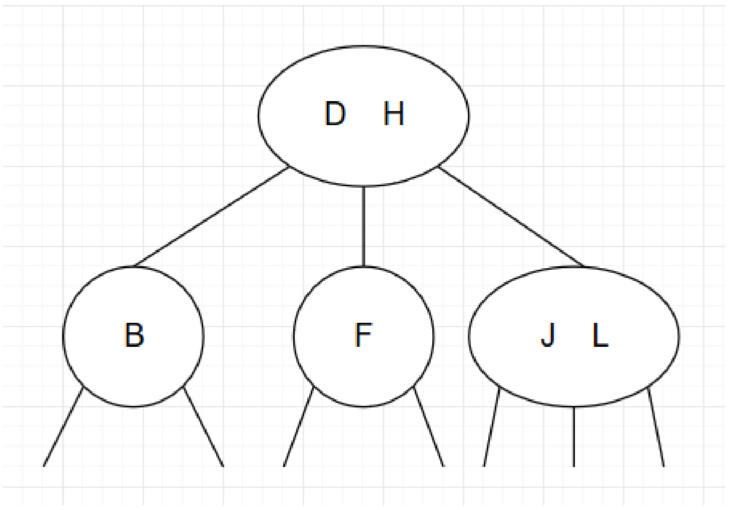
$ii)$



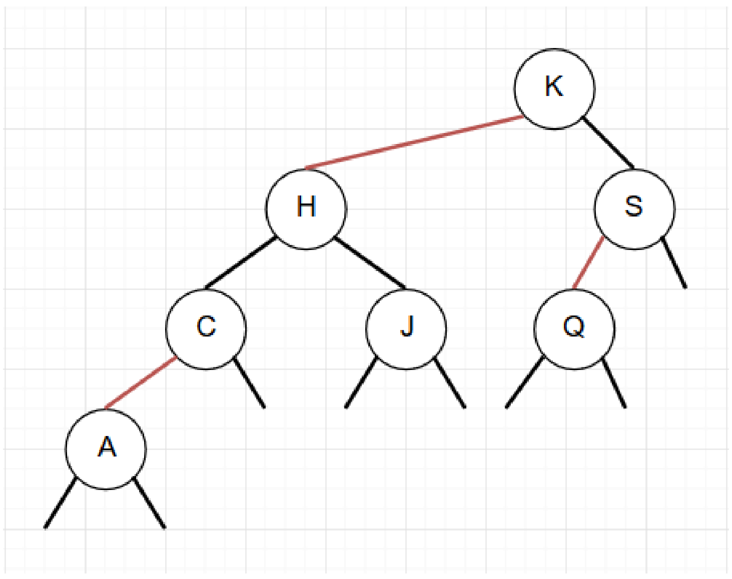
$iii)$



$iv)$



$v)$



e)

L R F F R F R F R F L

f)

The maximum height relative to n can be achieved when the tree is ordered such that the link from root to the leftmost null-node is made up of alternating red and black links, and except from the red links in this path, the entirety of the tree contains only black links.

Except for the left subtree from root, the red-black tree is a “full binary tree” where every path from root to null-node is $\sim \log_2 n$. Because no two consecutive links in the leftmost path can be red and it must contain the same amount of black links as the shortest root-null-node path in the tree, the maximum height becomes twice the shortest height: $\sim 2\log_2 n$.