University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Schneider

Algorithms and Data Structures
Winter Term 2019/2020
Sample Solution Exercise Sheet 6

Remark: For this exercise, watch the tenth video lecture.

Exercise 1: Mergesort

(a) Give the pseudo code of an algorithm `mergesort` to sort a given array A of size n that has runtime $\mathcal{O}(n \log n)$. It should implement the following rough algorithm description.

Divide the array in the "middle" and sort the left and right part recursively. Then merge the resulting two sorted subarrays into a sorted array.

You may utilize the procedure `merge` defined for an earlier exercise as subprocedure in `mergesort`.

(b) Prove the running time of your implementation of `mergesort`.

Sample Solution

(a)

Algorithm 1 `mergesort`$(A[\ell, r])$

if $\ell = r$ **then return**	▷ *base case, subarray* $A[\ell, r]$ *has just one element, i.e., is sorted*
$m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$	▷ *the "middle" index*
`mergesort`$(A[m+1, r])$	▷ *sort right half of* $A[\ell, r]$
`mergesort`$(A[\ell, m])$	▷ *sort left half of* $A[\ell, r]$
`merge`$(A[\ell, m], A[m+1, r])$	▷ *merge sorted subarrays* $A[\ell, m], A[m+1, r]$ *into* $A[\ell, r]$

A call of `mergesort`$(A[0, n-1])$ sorts A.

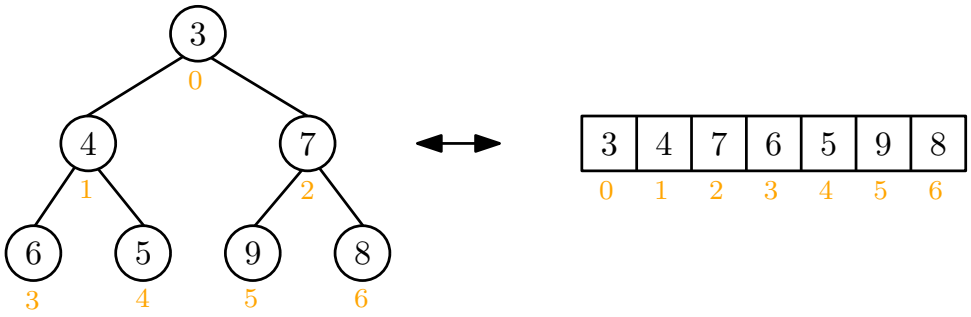
(b) The running time of `mergesort` on a (sub-)array of size n is the time for merging (which is $\mathcal{O}$("size of subarray") $= \mathcal{O}(n)$, c.f., Exercise Sheet 4), plus the running time of the two recursive calls of `mergesort` on arrays of size at most $\lceil \frac{n}{2} \rceil$. We end up with the following recursion:

$$T(n) = 2 \cdot T(\lceil \tfrac{n}{2} \rceil) + \mathcal{O}(n)$$

This solves to $\mathcal{O}(n \log n)$ running time using the Master Theorem.

Exercise 2: Heaps

A (min-)heap is a binary tree (a rooted tree where each node has at most two children) that stores keys in its nodes. The heap condition is defined as follows. *For each node, all keys in the left and right subtree are at least as large as the key of the node they are attached to.* We can store a heap in an array layer by layer, as exemplified in the following figure:



(a) **Insert Operation.** To insert an element, add it to the first free leaf position of the tree, which is the first free position in a layer of the tree that is not yet complete or the first free entry of the array if the heap is represented as such. Then swap the position of the current key with its parent, as long as said parent has a bigger key or there is no parent any more ("sift up").

Start with an empty heap and draw the current status (as a tree) after inserting the following keys (one tree for each inserted key): 7,8,4,9,3,5,2,6. Finally, give the heap in array representation.

(b) **Delete Min Operation.** First return the key in the root, then replace it with the key of the last element in the heap (the rightmost key in the lowest layer). Then repair the heap condition from the root down ("sift down") as follows. While the current node has children with smaller keys, swap positions with the smaller child.

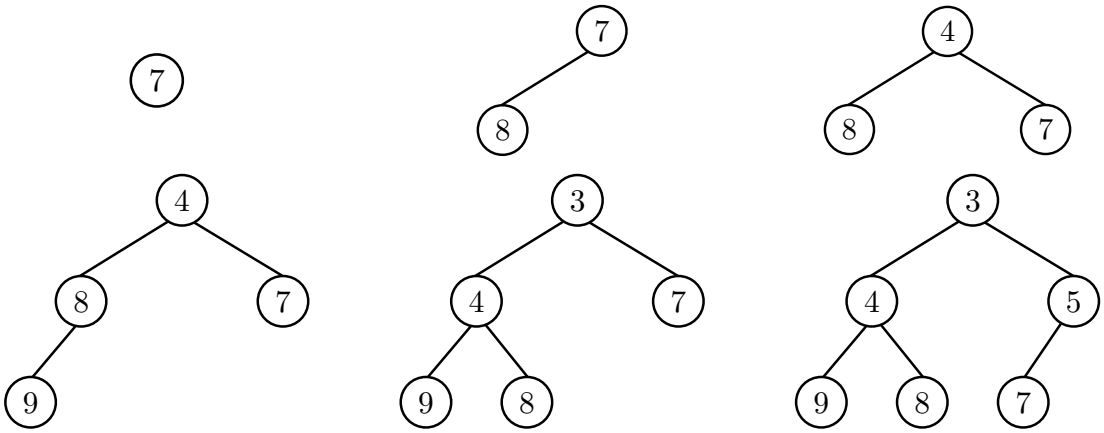
Delete the minimum three times in the tree resulting from part (a), and draw the state of the heap after each operation (as a tree or as array).

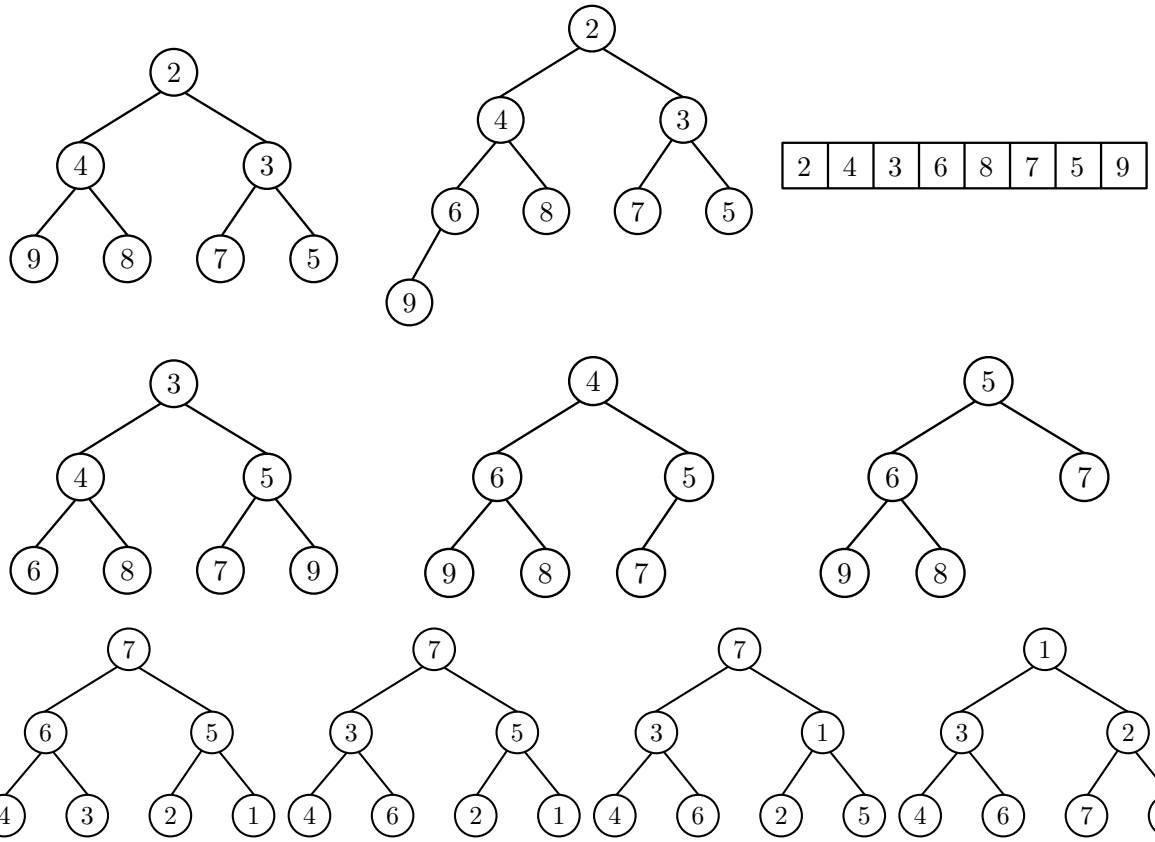
(c) **Heapify Operation.** The heapify operation takes a binary tree (which can also be represented as an array), and rearranges the tree to establish the heap condition. The procedure can be defined recursively as follows. Consider the current node v (initially the root of the tree). Call the heapify procedure recursively on the left and then the right child to make valid heaps out of the left and right subtrees of v. Then conduct a sift down operation on v (as described in part (b)).

Consider the tree represented by the array $A = [7, 6, 5, 4, 3, 2, 1]$. Conduct heapify on A and draw the state of A (as array or tree) after each recursive heapify.

Sample Solution

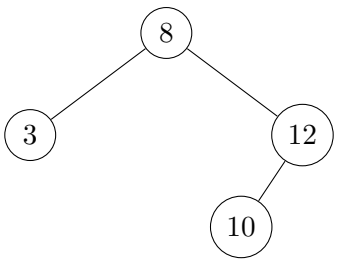
(a)





Exercise 3: Binary Search Trees

Consider the following binary search tree



(a) Give all sequences of `insert`(key) operations that generate the tree.

(b) Draw the tree after each of operation of the following sequence: `insert`(6), `insert`(5), `insert`(11), `insert`(13), `remove`(3), `remove`(8).

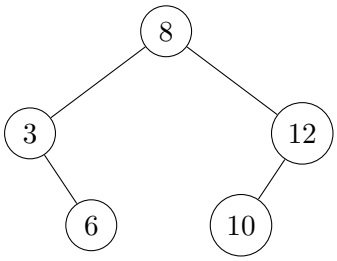
Sample Solution

(a) (i) `insert`(8),`insert`(3), `insert`(12), `insert`(10)

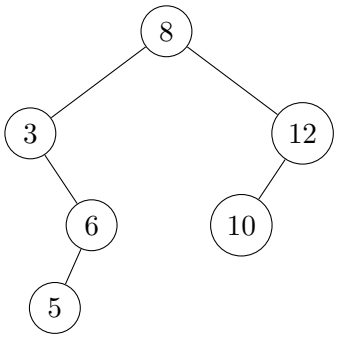
(ii) `insert`(8),`insert`(12), `insert`(3), `insert`(10)

(iii) `insert`(8),`insert`(12), `insert`(10), `insert`(3)

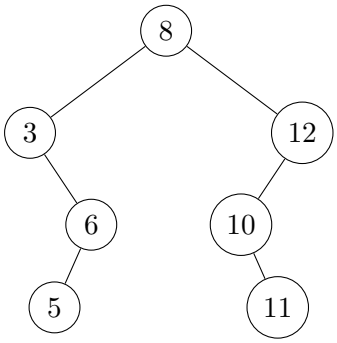
(b) After `insert`(6):



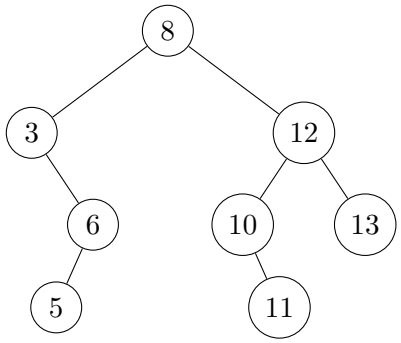
After `insert`(5):



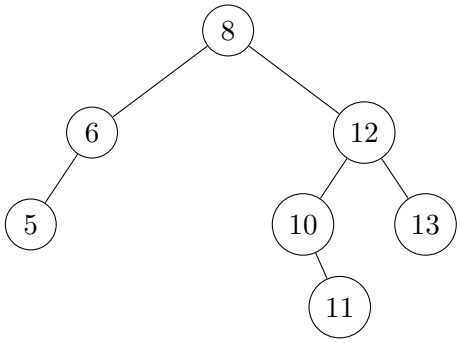
After `insert`(11):



After `insert`(13):



After `remove`(3):



After `remove`(8):