Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
Johannes Lengler, Gleb Novikov, Chris Wendler

28. October 2019

Algorithms & Data Structures — Exercise sheet 6 — HS 19

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

Points: ____

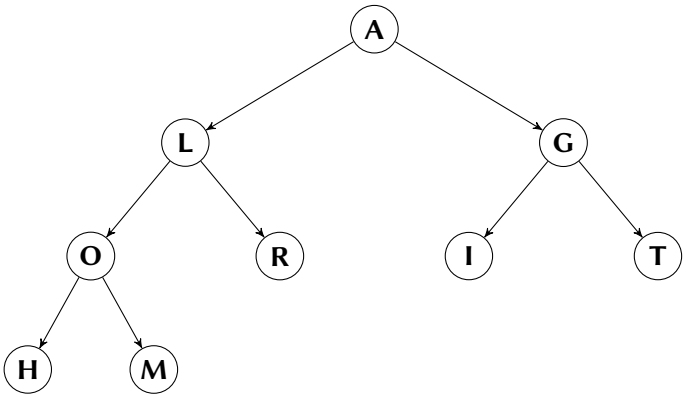
Submission: On Monday, 04 November 2019, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 6.1 *Heapsort* **(1 point)**.

Given the array [A, L, G, O, R, I, T, H, M], we want to sort it in ascending alphabetical order using Heapsort.

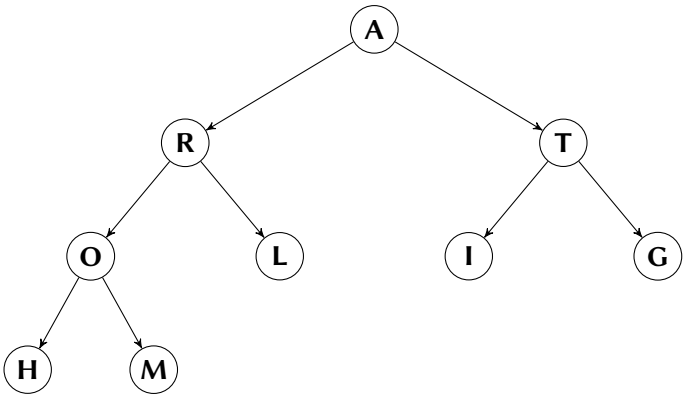
a) From the lecture and the script you know a method to construct a heap in linear time. Draw the resulting max binary heap if this method is applied to the above array.

Solution: The array [A, L, G, O, R, I, T, H, M] can be interpreted as the following binary tree:

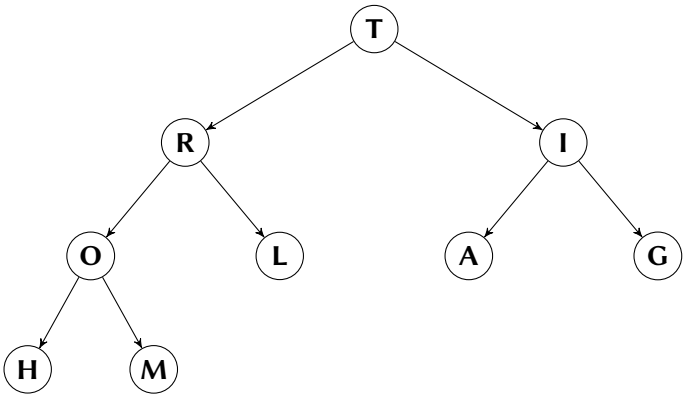


The root of the heap is at level 0. Heapyfing the subtree with root at level 2 does not change anything as it already satisfies the max heap condition.

Heapifying the subtrees with roots at level 1 yields:



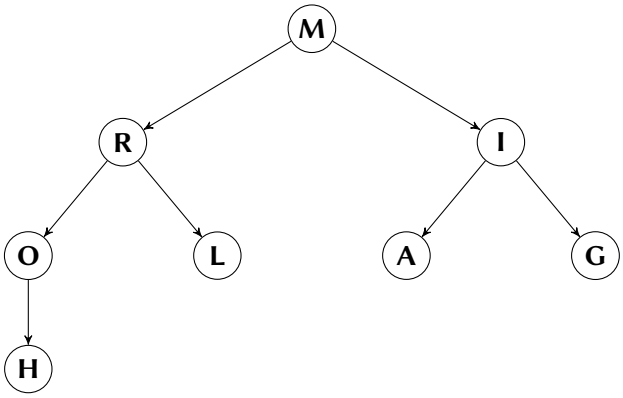
Now, heapifying the subtree at the root node yields



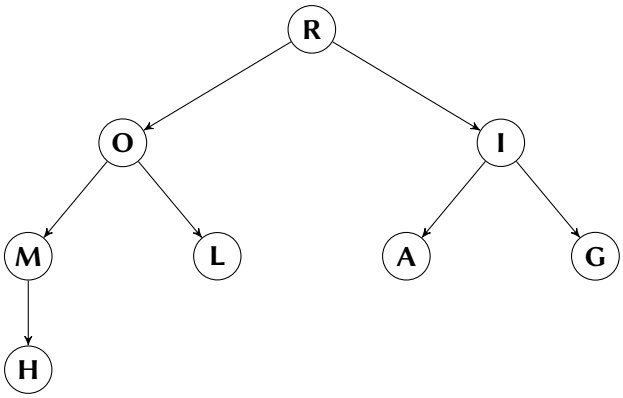
which corresponds to the array [T, R, I, O, L, A, G, H, M].

b) Sort the above array in ascending alphabetical order with heapsort, beginning with the heap that you obtained in a). Draw the array after each intermediate step in which a key is moved to its final position.

Solution: We begin with the max binary heap [T, R, I, O, L, A, G, H, M]. We extract the root T and put it into the last position in the array, i.e., we swap T with the last element M, removing T from the heap



and sifting M downwards until the heap condition is restored



Now, the array is [R, O, I, M, L, A, G, H, T] and contains the one-smaller heap in the front and a sorted entries in the end.

The array after the subsequent steps are as follows. Blue letters are at their final positions.

```
(a) Swap R and H:  [H, O, I, M, L, A, G, R, T]
    Sift H down:   [O, M, I, H, L, A, G, R, T]

(b) Swap O and G:  [G, M, I, H, L, A, O, R, T]
    Sift G down:   [M, L, I, H, G, A, O, R, T]

(c) Swap M and A:  [A, L, I, H, G, M, O, R, T]
    Sift A down:   [L, H, I, A, G, M, O, R, T]

(d) Swap L and G:  [G, H, I, A, L, M, O, R, T]
    Sift G down:   [I, H, G, A, L, M, O, R, T]

(e) Swap I and A:  [A, H, G, I, L, M, O, R, T]
    Sift A down:   [H, A, G, I, L, M, O, R, T]

(f) Swap H and G:  [G, A, H, I, L, M, O, R, T]
    Sift G down:   [G, A, H, I, L, M, O, R, T]

(g) Swap G and A:  [A, G, H, I, L, M, O, R, T]
    done:          [A, G, H, I, L, M, O, R, T].
```

We are done.

Exercise 6.2 *Sorting algorithms* ***(This exercise is from January 2019 exam)*** .

Below you see four sequences of snapshots, each obtained during the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

3	8	5	4	1	2	7	6
3	5	4	1	2	7	6	8
3	4	1	2	5	6	7	8

Bubblesort

3	8	5	4	1	2	7	6
3	5	8	4	1	2	7	6
3	4	5	8	1	2	7	6

Insertionsort

3	8	5	4	1	2	7	6
3	8	4	5	1	2	6	7
3	4	5	8	1	2	6	7

Mergesort

3	8	5	4	1	2	7	6
3	6	5	4	1	2	7	8
3	2	5	4	1	6	7	8

Selectionsort

Exercise 6.3 *Finding nouns* **(2 points)**.

Assume that you are working at the department of linguistics. You have several files on your computer that contain nouns of a very rare language (words in this language are written using standard English alphabet). Initially, the nouns were separated by spaces, but due to some technical error in your text editor, all the spaces disappeared. For example, if the original file contained

kuzdra bokr gostak doshes

then the corrupted file contains

kuzdrabokrgostakdoshes

Fortunately, you can call an expert in this language and ask him whether a string is a noun of this language or not. Your goal is to recover original sequences of nouns from corrupted files.

a) Provide a *dynamic programming* algorithm that takes as input a content of some file S (which is a string of English letters) and determines whether one can split S by spaces to get a sequence of nouns of the target language. Assume that you call an expert using some function $f(x)$ that returns true if a string x is a noun of a target language and false otherwise. Number of calls of f should be $\mathcal{O}(n^2)$ where n is a length of S (i.e. n is the number of letters in S). Your DP table should be *one-dimentional* and number of entries should be *linear*, i.e. $\mathcal{O}(n)$.

   For example, if all the nouns of the language are {bokr, bokrgos, doshes, drabok, gostak, kuz, kuzdra, takdos}, (that is, function f outputs true only at such strings), then your algorithm should output true on input "kuzdrabokrgostakdoshes" and false on input "kuztakdoshes".

   Address the following aspects in your solution:

   1) *Definition of the DP table*: What is the meaning of each entry?
   2) *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
   3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
   4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?

   Let $S = S[1], S[2], \ldots, S[n]$ be a string of length n and, for $i \leq j$, let $S[i:j] = S[i], \ldots, S[j]$.

   Meaning of a table entry (in words):

   $DP[i]$: a boolean value indicating whether $S_{1:i}$ can be splitted in the way described in the task.

Computation of an entry (initialization and recursion):

We initialize the table by setting $DP[0] = \texttt{true}$.

Then, the entries can be computed using the recursion

$$DP[i] = \begin{cases} \texttt{true} & \text{if there is an entry } 0 \leq k < i \text{ s.t. } DP[k] = \texttt{true} \text{ and } f(S[k+1:i]) = \texttt{true}, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Order of computation: From left to right.

Computing the output: The entry $DP[n]$ contains the solution.

b) State the running time of your algorithm in part a) in Θ-notation in terms of n (where n is the length of S). You can assume that calls of f take unit time.

In the worst case we require $\Theta(i)$ elementary operations (calls of f and bookkeeping operations) in order to determine $DP[i]$, namely,

$$f(S[i:i]), f(S[i-1:i]), \ldots, f(S[1:i]).$$

Thus, the running time is $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$.

c) Use DP table from part a) to efficiently find a sequence of nouns in the case when such a sequence exists. More precisely, provide an algorithm that takes as input a string S and a DP table from part a) and either splits S by spaces so that each string surrounded by spaces is a noun in a target language, or outputs false if it's not possible to split S in this way. If there are more than one possibility to split S in such a way, your algorithm can output any of these possibilities.

For example, if the input is

kuzdrabokrgostakdoshes

and the nouns of the language are {bokr, bokrgos, doshes, drabok, gostak, kuz, kuzdra, takdos}, then your algorithm should output

kuzdra bokr gostak doshes

Hint: *Follow your answer back through your DP table (Rückverfolgen).*

Solution: We start at the index $i = n$ and the empty sequence of nouns, and look for the largest index $0 \leq k < i$ such that $DP[k] = \texttt{true}$ and $f(S[k+1:i]) = \texttt{true}$. Then, we add $S[k+1:i]$ to the beginning of current sequence of nouns, update i to $i = k$ and continue finding the next noun. We do this until $i = 0$.

d) State the running time of your algorithm in part c) in Θ-notation in terms of n (where n is the length of S). You can assume that calls of f take unit time.

Solution:

Filling the output sequence requires $\Theta(n)$ operations. We traverse our DP table of size $n+1$ backwards and for every entry containing $\texttt{true}$ we call f at most once. Thus, the running time is $\Theta(n)$ (since the DP table contains $\texttt{true}$ at most n times).