S. Schwoon / D. Suwimonteerabuth Winter Term 2008/09

Fundamental Algorithms

Solutions to Example Problems

1. **Growth of functions**

   (a) False. Intuitively, the notation $\mathcal{O}(n^2)$ tells us that B runs "at most as fast as" n^2 time (modulo constant factors) on all inputs. This, however, does not prevent the existence of *some* (if not all) inputs, on which B runs faster than $\mathcal{O}(n)$. For instance, if the run-time (with input n) of A is determined by $f(n) = 3n$ and for B by $g(n) = n^2$, then $f \in \mathcal{O}(n)$ and $g \in \mathcal{O}(n^2)$ but $f(n) > g(n)$ for $n \leq 2$.

   (b) $f(n) = \begin{cases} 0 & \text{if } n \leq 0; \\ 2n + (n-1) + \sum_{i=1}^{n-1} i = \dfrac{n^2 + 5n - 2}{2} & \text{otherwise.} \end{cases}$

   (c) i. True. Choose $c = 3$ and $n_0 = 10$. Then for all $n \geq n_0$: $3n \leq 3n \log n$

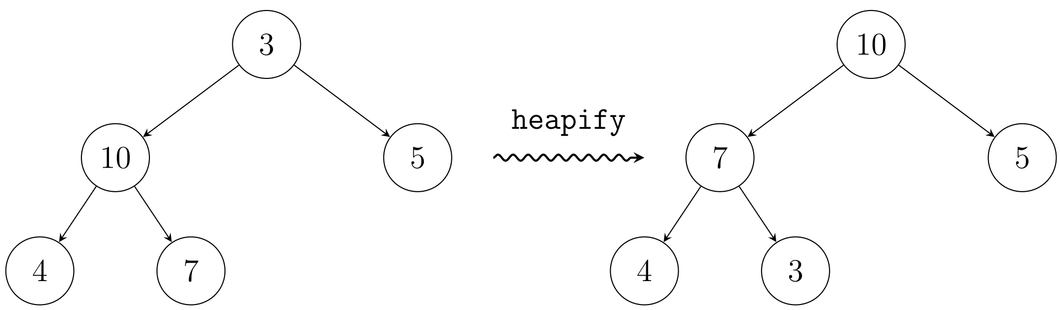
   ii. False. We need to prove that $\forall c > 0\ \forall n_0\ \exists n \geq n_0 : \frac{3}{2}n < c(n^{3/2} - n)$. Consider the inequality:

$$\begin{aligned} \frac{3}{2}n &< c(n^{3/2} - n) \\ \frac{3}{2}n &< c \cdot n(n^{1/2} - 1) \\ n &> \left(\frac{3}{2c} + 1\right)^2 . \end{aligned}$$

   Therefore, one can choose $n = \max\left(n_0, \left(\frac{3}{2c} + 1\right)^2\right)$ to prove that the statement above always holds.

2. **Sorting**

   (a) The array a is not a heap, because the number of the root is less that the numbers of its children. The array is a heap after a call to `heapify` with $i = 1$ and $k = 5$.



a `heapify`$(a, 5, 1)$

(b) There are two problems in the algorithm.

i. The first problem occurs when $\texttt{largest} + 1$ is greater than k. In that case, $\texttt{largest}+1$ must lie outside the slice $i \ldots k$, and therefore the element $a[\texttt{largest}+1]$ in the first if-statement can be undefined. The mistake occurs, for instance, for any even-sized array a (say, of size $2n$) and $i = n,\ \ k = 2n$. The problem can be fixed by adding an extra guard to the if-statement as follows:

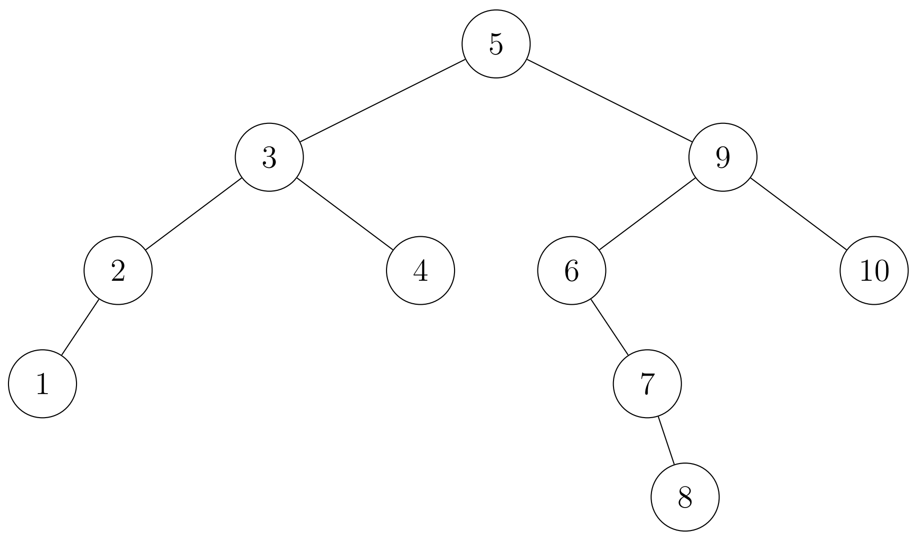
if $\texttt{largest} + 1 \leq k$ **and** $a[\texttt{largest} + 1] > a[\texttt{largest}]$ **then**

ii. There is another problem when the comparison in the second if-statement is false, i.e., the array a is already a heap. In that case, i is incorrectly set to `largest`, and therefore $a[i]$ is wrongly set to `tmp` after the loop exits. For instance, if a is $8, 10, 5, 4, 7$, i is 1, and k is 5, then the algorithm outputs $10, 10, 5, 4, 8$. The bug can be fixed by immediately exiting the loop when the comparison fails, i.e., the second if-statement becomes:

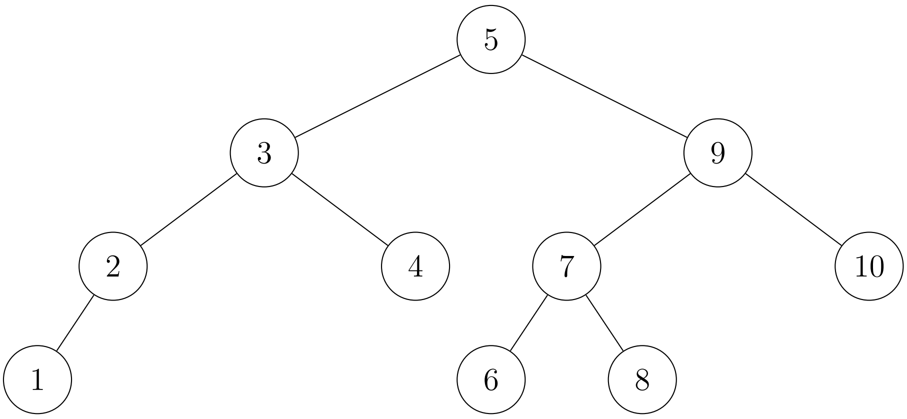
if $a[\texttt{largest}] > \texttt{tmp}$ **then**
    $a[i] := a[\texttt{largest}]$;
else
    break;
fi

3. **Searching**

(a)



(b) The tree in (a) is not an AVL tree, since the node with value 6 violates the balancing property. This can be fixed by the a single "right-right" rotation, which results in the following tree.



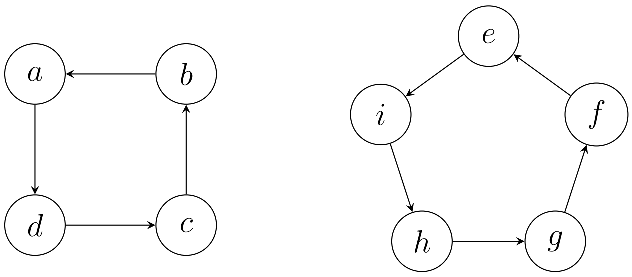
(c) Let t be a pointer to the ordered binary tree. Call the following procedure $\texttt{fill}(t, a, 1)$ to fill the array, where the last two arguments are in-out parameters.

Procedure `fill`
Input: pointer t to an ordered binary tree, array a, index i
Output: fill the array a, starting from index i, with elements from the tree

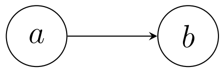
if $t = \texttt{NIL}$ **then return** ;
$\texttt{fill}(t \rightarrow \texttt{.right}, a, i)$;
$a[i] := t \rightarrow \texttt{.value}$; $i := i + 1$;
$\texttt{fill}(t \rightarrow \texttt{.left}, a, i)$;

4. **Graphs**

(a) Consider the graph shown below. The desired numbering results from starting first at d and then i.



(b) The proposal is wrong. Consider the following graph G, which (trivially) contains two SCCs. In pre-order numbering if node b is visited first, then b is labeled with 1 and a with 2. Then, when performing another search from the node with the lowest number, i.e. b, in G_R we wrongly conclude that a and b are in an SCC.



(c) Recall that a heuristic function h is *monotone* if for all two adjacent nodes w and z, where $d(w, z)$ denotes the length of the actual shortest path between them, then

$$h(w) \leq d(w, z) + h(z) \ .$$

In both graphs, one can readily check that the property holds for each pair of nodes.