Exercises week 5 (partial solutions)

Complexity 2011-2012

A. Silva, H. Barendregt, B. Westerbaan & B. Westerbaan

Exercises marked with (†) are harder exercises. Exercises marked with a **(*)** can be handed in, we will correct them and give them back to you the week after. This week the answers should be handed in before **March 13 at 23h59 (CET time)**.

Handing in your answers: There are two options: e-mail to `alexandra@cs.ru.nl` or put your solutions in the post box of Alexandra (more detailed instructions are in the first week exercise sheet: `http://alexandrasilva.org/files/teaching/complexity2012/ex1.pdf`).

Exercise 1. (*) Consider the following propositions

$$(p \to q) \to (q \to p), \quad (p \to q) \vee (q \to p), \quad ((p \to q) \to p) \to p, \quad \neg((p \to q) \vee p) \vee (\neg(p \to q) \wedge q).$$

Give the truth tables for the three propositions and conclude whether they are valid, satisfiable or neither.

Answer.

p	q	$p \to q$	$q \to p$	$(p \to q) \to (q \to p)$
0	0	1	1	1
1	0	0	1	1
0	1	1	0	0
1	1	1	1	1

satisfiable

p	q	$p \to q$	$q \to p$	$(p \to q) \vee (q \to p)$
0	0	1	1	1
1	0	0	1	1
0	1	1	0	1
1	1	1	1	1

valid

p	q	$p \to q$	$(p \to q) \to p$	$((p \to q) \to p) \to p$
0	0	1	0	1
1	0	0	1	1
0	1	1	0	1
1	1	1	1	1

valid

p	q	$p \to q$	$(p \to q) \vee p$	$\neg((p \to q) \vee p)$	$\neg(p \to q)$	$\neg(p \to q) \wedge q$	$\neg((p \to q) \vee p) \vee (\neg(p \to q) \wedge q)$
0	0	1	1	0	0	0	0
1	0	0	1	0	1	0	0
0	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0

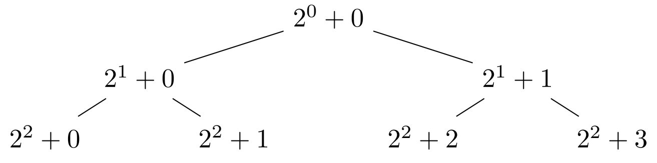
neither

Exercise 5. (*) Recall last week's exercise about heaps.

Given the index i of a node, it is easy to find the indices of the parent node $P(i)$, the left child node $L(i)$ and the right child node $R(i)$.

(i) Give L, R and P using multiplication, division, addition and/or floor.

Answer. Given an integer i larger than zero. There exist unique e and r such that $2^e + r = i$ and $r < 2^e$. Actually, $2^e + r$ is the index of the $(r+1)^{\text{th}}$ node in the $(e+1)^{\text{th}}$ level of the heap. In a picture:



Every predecessor of $2^e + r$ on the same level ($2^e + 0$, $2^e + 1$, …, $2^e + r - 1$) has two children. Note that there are r predecessors. These children are exactly the predecessors of the left child of $2^e + r$. Thus the left child of $2^e + r$ has index $2^{e+1} + 2r = 2(2^e + r)$. Hence

$$L(i) = 2i \qquad\qquad R(i) = 2i + 1.$$

(**Side remark:**

Note that here we start the indices at 1, *if we would start at* 0, *then*

$$L(i) = 2i + 1 \qquad\qquad R(i) = 2i + 2.$$

)

Conversely, if r is even, $2^e + r$ is the left-child of its parent. If r is odd, $2^e + r$ is the right-child of its parent. Thus

$$\begin{aligned} P(2^e + r) &= \begin{cases} 2^{e-1} + \frac{r}{2} & \text{if } r \text{ is even} \\ 2^{e-1} + \frac{r-1}{2} & \text{if } r \text{ is odd} \end{cases} \\ &= 2^{e-1} + \left\lfloor \frac{r}{2} \right\rfloor \\ &= \left\lfloor \frac{2^e + r}{2} \right\rfloor \\ P(i) &= \left\lfloor \frac{i}{2} \right\rfloor. \end{aligned}$$

(ii) The merit of heaps is that some useful operations related to them are easy and fast. This week, we will explicitly study the functions `heapify` and `popmin`, which we included in last week's assignment without showing the algorithms.

```
int popmin(int A[], int N) {
 /* N is the size of the array */
 int r = A[0];
 A[0] = A[N-1];
 fix(A,N-1,0);
 return r;
}

void heapify(int A[], int N) {
 /* N is the size of the array */
 for(i=N-1, i>=0,i--)
```

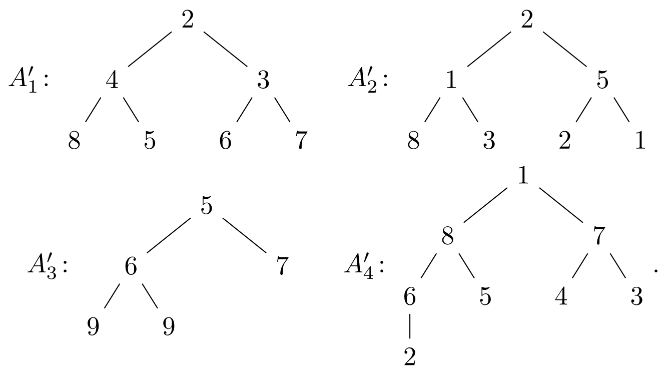
```
     fix(A,N,i);
}

void fix(int A[], int N, int i)  {
 /* N is the size of the array */
 int j = i;
 if (R(i) < N) {
    if (A[i] >= A[L(i)] ||  A[i] >= A[R(i)])
       if (A[L(i)] >= A[R(i)])
              j = R(i);
       else   j = L(i);
 }
 else {
  if (L(i) < N && A[i] >= A[L(i)])
     j = L(i);
 }

 if (j != i){
  swap(A, i,j);
  fix(A,N, j);
 }
}
```

`heapify` reorders an array such that it becomes a heap; `popmin` returns and removes the least element from the heap. Consider the following arrays

$$A_1 = [1,2,3,4,5,6,7,8] \quad A_2 = [1,2,5,1,3,2,1,8] \quad A_3 = [3,5,7,9,6,9] \quad A_4 = [9,8,7,6,5,4,3,2,1]$$

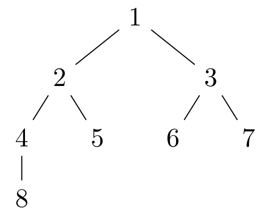
(a) Apply `popmin` to A_1, A_2, A_3 and A_4. Give the return values and the changed arrays.

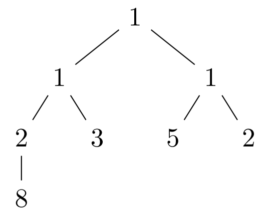
Answer. `popmin` returns for A_1, A_2, A_3 and A_4 respectively 1, 1, 3 and 9 and the arrays are changed to

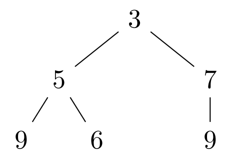


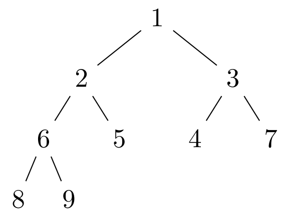
(b) Apply `heapify` to A_1, A_2, A_3 and A_4. Give the changed arrays.

Answer.

A''_1:


A''_2:


A''_3:


A''_4:


(c) Describe what `fix` does.

Answer. Given an index i, consider the full subtree with i as root. If this subtree satisfies the heap property, except possibly on the root (that is, the value of the root is larger than one of the values of its children), then $\texttt{fix}(A, i)$ corrects this: it reordens the subtree such that it becomes a heap. It does this by swapping the root of the subtree with the child with the smallest value and then recursively calling `fix` on the old index of the swapped child.

That swap makes sure that the heap property is satisfied at the root. However, the heap property may not be satisfied anymore at child which was swapped. Hence the recursion.

(d) Prove that `popmin` runs worst-case in time of order $O(\log n)$.

Answer. We first analyse the worst-case running time of `fix`. Given an array H and index i. Let h be the height of the subtree of H as complete binary tree, with the i^{th} node as root. Let $T_{\texttt{fix}}(h)$ be the maximum number of steps that `fix` uses on any input array H and index i where the height of the subtree headed by i is h. Note that if $h = 1$, then `fix` won't recurse. Thus we have

$$T_{\texttt{fix}}(h) \leq \begin{cases} T_{\texttt{fix}}(h-1) + C_1 & h > 1 \\ C_1 & h = 1. \end{cases}$$

And thus $T_{\texttt{fix}}(h) \leq C_1 \cdot h$. In other words, $T_{\texttt{fix}} \in O(h)$. If $i = 1$ then h is the actual height of H seen as complete binary tree. That is $h = \lceil \log(n+1) \rceil$. One can prove that $\lceil \log(n+1) \rceil \in O(\log n)$.

Concerning `popmin`, we can immediately see that it has the same complexity as `fix`, that is $T_{\texttt{popmin}} \in O(\log n)$ too.

(e) Prove that `heapify` runs worst-case in time of order $O(n)$. (Hint: If $S = \sum_{i=0}^{n} i2^i$, look at $2S - S$ to derive a closed formula for S.)

Answer. `heapify` runs `fix` on every node. Recall $T_{\texttt{fix}}(h) \leq Ch$, where h is the height of the subtree headed by i. `heapify` calls `fix` for every node. Note that there are 2^{i-1} nodes at height i. Let h be the height of A seen as tree. Then certainly

$$\begin{aligned} T_{\texttt{heapify}}(n) &\leq \sum_{i=1}^{h} T_{\texttt{fix}}(i) \cdot 2^{h-i} \\ &= C \sum_{i=1}^{h} i \cdot 2^{h-i}. \end{aligned}$$

Let $S := \sum_{i=1}^{h} i \cdot 2^{h-i}$. Then

$$\begin{aligned} S &= h \cdot 2^0 + (h-1)2^1 + \cdots + 2 \cdot 2^{h-2} + 12^{h-1} \\ 2S &= h \cdot 2^1 + (h-1)2^2 + \cdots + 2 \cdot 2^{h-1} + 12^{h} \\ 2S - S = S &= -h \cdot 2^0 + 2^1 + 2^2 + \cdots + 2^{h-2} + 2^{h-1} + 2^h. \end{aligned}$$

Furthermore let $T := \sum_{i=1}^{h} 2^i$, then

$$\begin{aligned} T &= 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h \\ 2T &= 2^2 + 2^3 + \cdots + 2^h + 2^{h+1} \\ 2T - T = T &= 2^{h+1} - 2. \end{aligned}$$

Note that $S = T - h$ and thus

$$S = 2^{h+1} - 2 - h.$$

The previous combined with $h \leq \log n + 1$, yields

$$\begin{aligned} T_{\texttt{heapify}} &\leq C2^{h+1} - 2 - h \\ &\leq C2^{h+1} \\ &\leq C2^{\log n + 2} \\ &= C2^{\log n}2^2 \\ &= 4Cn \in O(n). \end{aligned}$$

(f) Prove that `heapify` runs worst-case in time of order $\Omega(n)$.

Answer. The main loop of `heapify` contains n iterations. Thus $T_{\texttt{heapify}} \in \Omega(n)$.