Heaps

Abstract data type to store values ordered by a **total order** relationship $\preceq$. Operations provided should be able to provide efficient handling to the data type, provide the following tasks:

- Building an heap from a set of data;
- find minimum wrt $\preceq$ in a fast way;
- extract minimum wrt $\preceq$;
- decrease by one values in the heap wrt $\preceq$;
- insert a new value.

If the order $\preceq$ is $\leq$ is called **min-heap**, if the order $\preceq$ is $\geq$ is called **max-heap**.

Heaps can be used to implement *priority queue*, where the element to be extracted minimizes a priority criterium.

Binary heaps

A **Binary heap** is a nearly complete binary tree (i.e. complete up to the second last level and all leaves are on the left, namely the last level must be filled from the left). *Heap-property*: each parent should be $\preceq$ than any child wrt our chosen order relation, namely $parent(p) \preceq p \ \forall$ nodes.

We can represent a binary heap like a tree, or with **array representation**. In this last representation we place the value labelling the root node as first element of the array (index 1); the i-th position of the array represent a node (every node is represented by an index) in such a way that:

- the left child of the node in position i has index $2 \cdot i$;
- the right child of the node in position i has index $2 \cdot i + 1$;
- the parent of the node in position i $\left\lfloor \frac{i}{2} \right\rfloor$.

Functions of a binary heap

Finding the minimum of a binary heap corresponds to finding the root, complexity is $\Theta(1)$.

Remove the minimum: once we remove the minimum we must preserve the fact that we end up in an almost complete tree (topological structure) and the heap property. A possible way to do it is to replace the root by using the rightmost leaf of the last level (hence deleting this leaf). In this way we preserve the topology of the heap. The heap property in the root however may be lost. In order to restore it, we can consider the root node and its children and we take the minimum wrt $\preceq$. At this point we swap the value labelling the root and the smallest value we found (if the smallest value is in the root, heap property hadn't been violeted). However it could be the case that we pushed down the

problem, i.e. between the child of the root and its children (if any). So we need to repeat the procedure on the subtree rooted on the new node:

- if it is the minimum in the subtree, we are done;
- however we find the minimum and swap.

This procedure is called **HEAPIFY**.

Complexity of removing the minimum:

- replacing the root costs:
  - $\Theta(1)$ identify the root;
  - $\Theta(1)$ find the rightmost leaf;
  - $\Theta(1)$ replace the value;
- each iteration of heapify consists of
- 2 comparisons to find the minimum and
- at most one swap.

We may have pushed down the product, so the complexity is $O(log_2(n))$, where n is the number of elements in the heap. Note that we are using the big-O notation, i.e. we are bounding from only above, because we can be lucky and bound from below with a constant complexity ($\Omega(1)$).

Building a binary heap: we get the heap for free just building an array, however we have to fix the heap property. Idea is to fix it bottom-up using **Heapify**. On the leaves heapify doesn't do anything. Then we call heapify on the second last level, on the third last and so on until we get to the root.

The cost of performing these operation:

- heapify costs O(h) (i.e. it is $\leq c \cdot h$ for some constant c), with h being the height of the tree (if the tree has n nodes, h is $\lfloor \log_2(n) \rfloor$); if a tree has n nodes, at level h nodes are at most $\lceil \frac{n}{2^{h+1}} \rceil$).
- using the fact that we can replace O(n) with $c \cdot n$ we obtain the complexity of building a binary heap: $T_{BH} \leq \sum_{h=0}^{\lfloor \log_2(n) \rfloor} (\lceil \frac{n}{2^{h+1}} \rceil) =$
  $= \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{n}{2^h} \cdot (c \cdot h) \leq$
  $\leq c \cdot n \cdot \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \leq$
  $\leq c \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} =$
  $= c \cdot n \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \cdot c \cdot n \in O(n).$

Decreasing a node's key (wrt $\preceq$): it preserves the heap property on the subtree rooted on the node, but it may broke the property wrt its parent. In order to fix it we swap the new value with its parent (this solve the problem on the subtree rooted on the parent) until we end up in a situation in which the heap property is restored on all the tree (here again we are moving bottom-up). The

complexity of decreasing a key is $O(log_2(n))$, since the heap height is $\lfloor \log_2(n) \rfloor$ since the computation takes $\Theta(1)$ at each step.

Inserting a new value (preserving the heap topology). Note that we cannot find the right place to insert a node just by browsing the tree. Idea is to insert an element in the last level of the heap, in the leftmost available position and insert as a key the maximum possible value for the order relation, and then decrease it to the desired value by means of the **decreasing a key** function (actually it is enough to insert the maximum value ever been in the heap). The complexity of inserting a value is $O(log_2(n))$ (the same as decreasing a node's key).