Theory: Binary heap

27 minutes 8 / 8 problems solved

§1. Definition

A **binary heap** is a tree with the following properties:

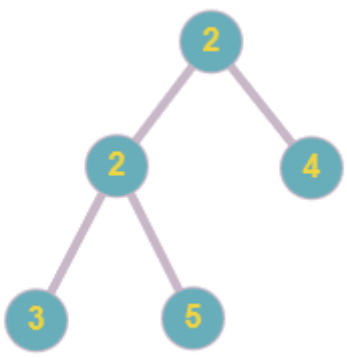
- each node's value is minimum among all values present in its subtrees;
- the difference between the maximum and minimum depth of the leaves is no greater than 1;
- it is always either a **complete** tree or an **almost complete** binary tree.

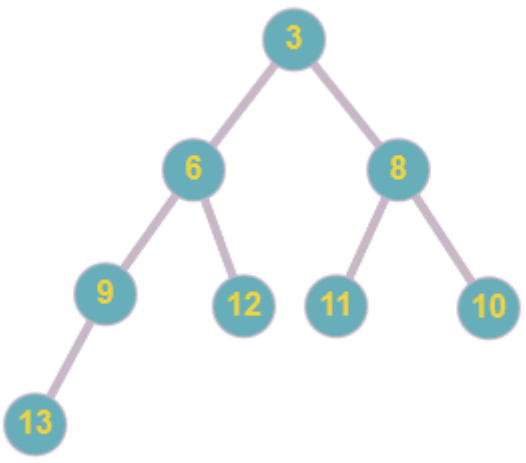
A **complete** binary tree of level n is a tree in which every node of level n is a leaf and every node of a level less than n has both right and left subtrees.

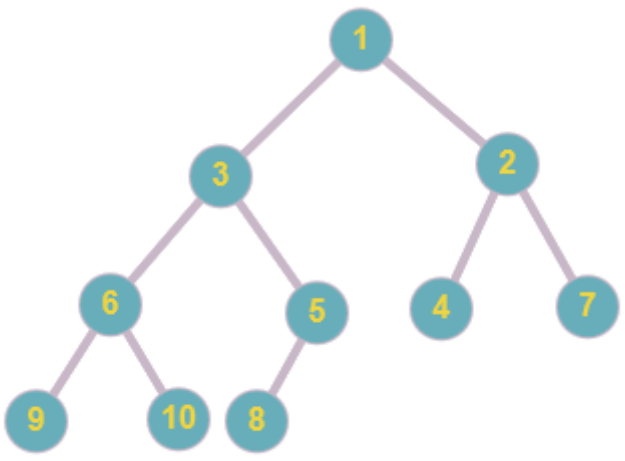
An **almost complete** binary tree is a binary tree for which there is a positive k, such that:

- every leaf in a tree has level k or $k+1$;
- if a node of the tree has a right child of level $k+1$, then all the left children of this node are leaves also have level $k+1$.

The images below show various examples of binary heaps:







Just like with binary search trees, the elements of a binary heap can be of any data type for which the comparison operator is defined (characters, strings, etc.)

Current topic:

✓ Binary heap ···

Topic depends on:

✓ Tree ···

Topic is required for:

Binary heap in Java ···

Table of contents:

In this topic, we will talk specifically about min-heaps which can also be referred to as **queues with the minimum**. Max heaps work absolutely the same. The only difference is the comparison characters that we will use during the construction of a heap and operations with it.

Two basic operations can be performed on binary heaps: **adding an item** and **removing (extracting)** the smallest one. If a binary heap allows extracting the smallest element, then it is called a **min-heap**, and in the case with the largest element, it is a **max-heap**.

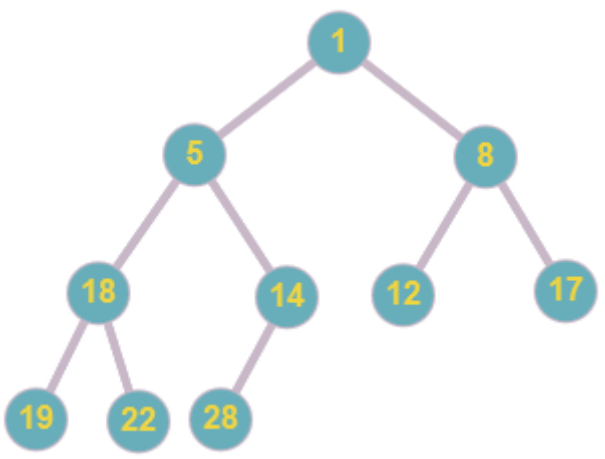
Let's analyze each of the two basic operations in more detail.

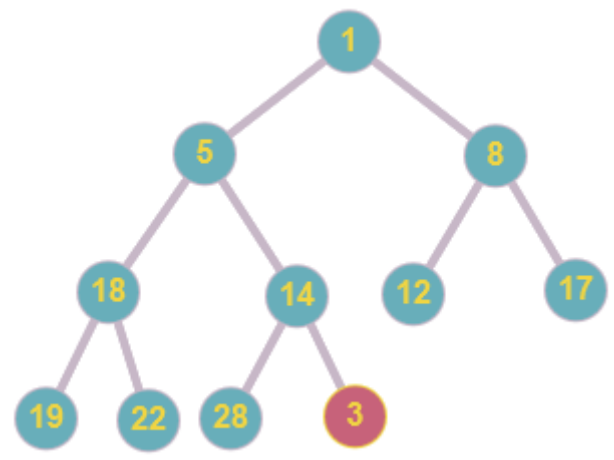
§2. Adding: Insert(key)

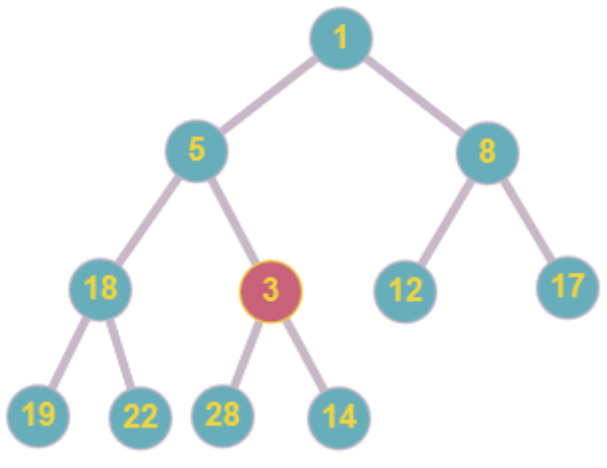
The features of this operation are dictated by the three basic properties of a binary heap that we have listed in the beginning. Because of the second and the third property, the only possible place to insert a new node is after the last node at the last level. If it is filled, then we put the node as the first one at the next level.

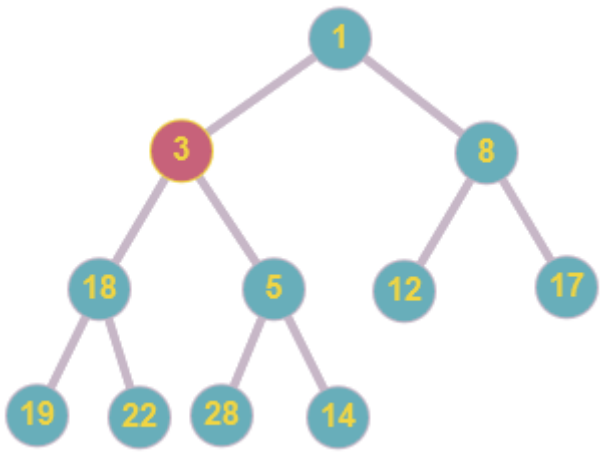
However, this simple operation may still break the first property. If it happens, here is how you fix it: first, swap the current node with its parent, and then repeat the action until the requirement is met. Note that the violation of the first property can only occur when we insert a node and have to move it up from the lowest level to its correct position. There will be no more than $log(n)$ nodes in that subtree that will be "out of place": moving the node up, we make sure that the first property is satisfied. Other subtrees will not be affected.

The following images show the sequence of rotations when adding item "3":









The algorithm is completed when the tree satisfies the first property: 1 <= 3, so a correct binary heap is achieved.

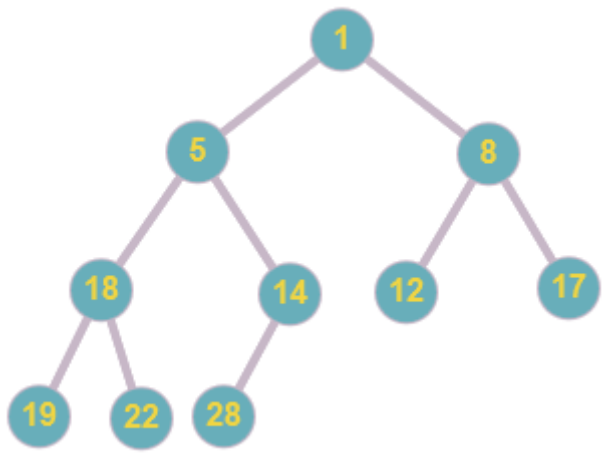
§3. Extracting a minimum: ExtractMin()

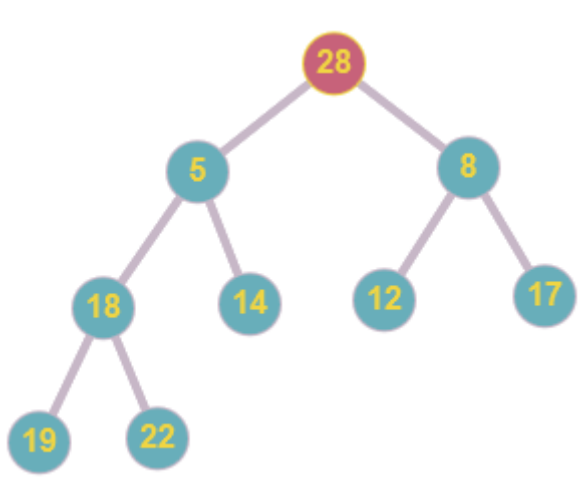
As you may have noticed, the minimum element in the min-heap is always the root of the tree, so basically we must learn to remove the **root** element.

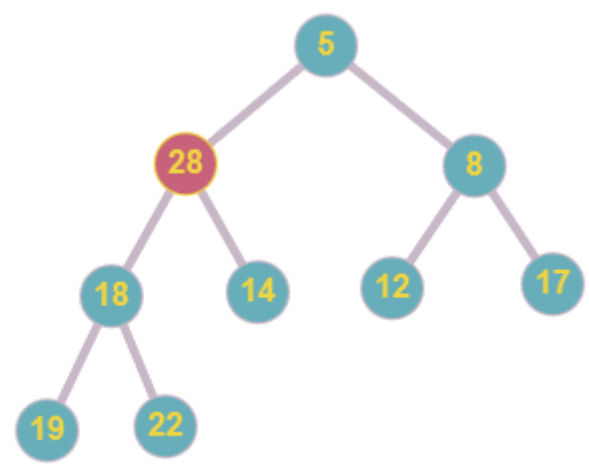
The first step is to save the value in the root for the subsequent return of the function. Next, move the last item in the heap to the root of the tree. The structure you get will not satisfy the first condition, so now our task is to move this element down to a position where all requirements are met.

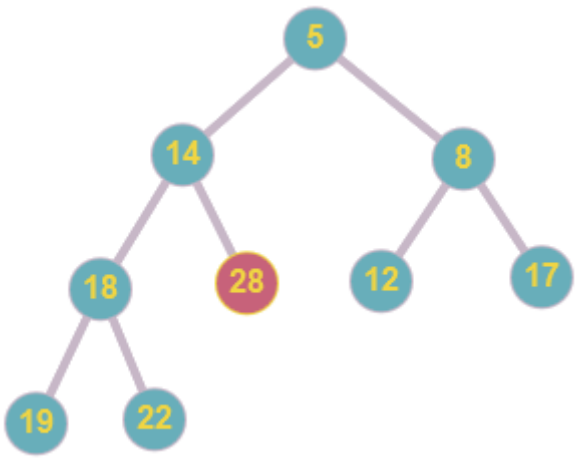
In contrast to the rotation of elements during the insertion operation, we have two options here: either the right descendant or the left. For a min-heap, you should swap the current node with its smallest child; for a max-heap, go for the greatest one.

The images below show a chain of rotations after the extraction of the minimum.









The result is a tree that satisfies all three conditions.

There are other operations that are based on those two, such as removal and alteration of any value in the binary heap. Using these operations will require to change the data structure a bit. We will talk about it during practice.

Building a binary heap over an existing array of data takes $O(n)$ time. All mentioned operations on a binary heap take O(log n) at worst. When adding an element or extracting a minimum, the number of rotations will be no more than the tree depth. Since the tree is almost complete, its depth does not exceed $O(\log n)$.

§4. Heapsort

Based on the proposed data structure, one can take a very interesting approach to its elements' sorting. This approach is called **heapsort**. The idea is to keep extracting the minimum as long as the size of the heap doesn't drop to zero. The construction of a binary heap will take O(n) time, and there will also be n extract operations for $O(\log n)$. Thus, the total asymptote is $O(n) + n * O(\log n) = O(n * \log n)$.

The advantage of the heapsort is that it is "on the spot" and has good complexity. Also, if you need to find the k-th minimum, you only need $O(k * \log n)$ time, and not $O(n * \log n)$ as it is with the quicksort.

Report a typo

31 users liked this theory. **6** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (1) Show discussion