Theory: Binary heap in Java

35 minutes 0 / 5 problems solved

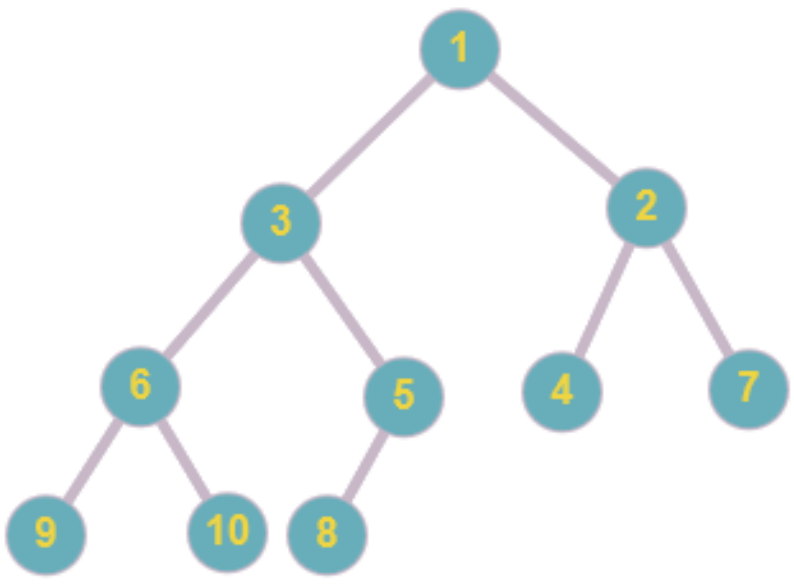
Skip this topic Start practicing

§1. Binary heap

Binary heap is a data structure which is a binary tree with certain required properties:

- values of children are not smaller than the parent's node value (**min-heap**);
- the difference between max and min depth of leaves is no greater than 1;
- the tree is complete.

The figure below is an example of a min-heap:



§2. Implementation in Java

As you remember, there is a convenient way of storing binary trees in **arrays**, where children of the node i will be nodes numbered $2i$ and $2i + 1$. Because the tree is complete, it is safe to say that the array will have no empty spaces and its size will be proportional to the size of the heap, which is very good in terms of memory management.

Take a look at the main class for min-heap implementation:

Current topic:

Binary heap in Java

Topic depends on

Trees in Java

Table of contents:

↑ Theory: Binary heap in Java

§1. Binary heap

§2. Implementation in Java

§3. Heapsort

§4. Conclusion

Feedback & Comments

```
public class MinHeap {
    private int[] heap;
    private int size;
    private int maxsize;

    public MinHeap(int maxsize) {
        this.maxsize = maxsize;
        this.size = 0;
        heap = new int[this.maxsize + 1];

        heap[0] = Integer.MIN_VALUE;

    }



    private void swap(int fpos, int spos) {

        int tmp;

        tmp = heap[fpos];

        heap[fpos] = heap[spos];

        heap[spos] = tmp;

    }



    private void minHeapify(int pos) {

        if (2 * pos == size) {

            if (heap[pos] > heap[2 * pos]) {

                swap(pos, 2 * pos);

                minHeapify(2 * pos);

            }

            return;

        }



        if (2 * pos <= size) {


    if (heap[pos] > heap[2 * pos] || heap[pos] > heap[2 * pos + 1]) {

                if (heap[2 * pos] < heap[2 * pos + 1]) {

                    swap(pos, 2 * pos);

                    minHeapify(2 * pos);

                }

                else {

                    swap(pos, 2 * pos + 1);

                    minHeapify(2 * pos + 1);

                }

            }
```

```
        }
    }


    public void insert(int element) {
        heap[++size] = element;
        int current = size;

        while (heap[current] < heap[current / 2]) {
            swap(current, current / 2);
            current = current / 2;
        }
    }


    public void minHeap() {
        for (int pos = (size / 2); pos >= 1; pos--) {
            minHeapify(pos);
        }
    }


    public int extractMin() {
        if (size == 0) {
            throw new NoSuchElementException("Heap is empty");
        }
        int popped = heap[1];

        heap[1] = heap[size--];
        minHeapify(1);
        return popped;
    }
}
```

Let's discuss its basic components:

- `int[] heap` – the main array containing the elements of the heap;

- `int size, maxsize` – current and max sizes of the heap;

- `void swap(a, b)` – a helper method that swaps two values at indexes `a` and `b`;

- `void insert(elem)` – a method for adding the new `elem` value to the heap;

- `int extractMin()` – extracts the minimum value which is located in the root from the min-heap and balances the tree;

- `void minHeapify(pos)` – pushes the value at the `pos` position to its proper place. This method is required when we remove the minimum element with *extractMin()*, replacing it with an element from the bottom then sifting it down to its required place. During the addition operation, we conversely lift the element from the bottom to where it should be, so this operation doesn't need a helper method, unlike *extractMin()*;

- `void minHeap()` – a method which transforms an arbitrary array into a min-heap in place. In other words, it is equivalent to several uses of `insert(elem)`, only more elegant. The usage of methods for creating a heap should be obvious: in the first case, we already have all the elements, and in the second case, we are told which nodes to add one by one. Please note that you should opt for `minheap()` method where you can: it has $O(n)$ asymptote for all elements altogether, while `insert(elem)` has $O(\log n)$ for each element, which is slower.

You have probably noticed that the zeroth element of the `heap` array is initialized with the smallest integer. This is done in order to ensure that surfacing of the number during addition will eventually finish, since `while (heap[current] <heap[current / 2])` will stop when `heap[current / 2]` becomes the smallest integer.

Now it's time to learn how to use the class:

```
MinHeap minHeap = new MinHeap(15);
minHeap.insert(5);
minHeap.insert(45);
minHeap.insert(83);
minHeap.insert(23);
minHeap.insert(34);
minHeap.insert(71);
minHeap.insert(36);
minHeap.insert(10);

minHeap.insert(13);

out.println(minHeap.extractMin()); // 5

out.println(minHeap.extractMin()); // 10

minHeap.insert(4);

out.println(minHeap.extractMin()); // 4

out.println(minHeap.extractMin()); // 13
```

At first, we initialize the heap, knowing in advance that its size will not be greater than 15. Then we add elements 5, 45, 83, 23, 34, 71, 36, 10 and 13. After that we extract two minimum values, which are clearly 5 and 10. Next, we add 4 to the heap and it becomes the minimum element. Finally, we call the "extract Min" method twice, and it returns 4 and 13 respectively. Both addition and minimum extraction operations work in $O(\log n)$.

§3. Heapsort

We will try to code this sort (which we discussed in theory) during this lesson. In fact, we already have minHeap() method that transforms an array into a min-heap, and we simply need to call extractMin() *n* times to get a sorted array. We won't describe the full code of heapsort here: you will receive a task for that later, and theoretical description given earlier should suffice. To remind you, the resulting algorithm will have $O(n) + O(n \log n) = O(n \log n)$ time complexity and $O(n)$ memory complexity. The latter is achieved by algorithm working in place: no extra memory is required.

§4. Conclusion

In this lesson, we examined how binary heaps work and considered the main operations, time complexity and heapsort. But there is a catch: Java has `PriorityQueue`, an innate binary heap that can work with arbitrary data types.
In most cases, this binary heap implementation is enough, but you should know its structure anyway: you might need to modify the algorithm which you cannot do with PriorityQueue. During the next exercises you will get an opportunity to write your own PriorityQueue.