Discover the properties of binary heaps

Running example

First property: level-completeness

In week 7 we have seen binary trees:

- We said they should be as “balanced” as possible.
- Perfect are the perfect binary trees.
- Now close to perfect come the level-complete binary trees:
  - We can partition the nodes of a (binary) tree $T$ into levels, according to their distance from the root.
  - We have levels $0, 1, \ldots, \text{ht}(T)$.
  - Level $k$ has from 1 to $2^k$ nodes.
  - If all levels $k$ except possibly of level $\text{ht}(T)$ are full (have precisely $2^k$ nodes in them), then we call the tree level-complete.

General remarks

- We return to sorting, considering HEAP-SORT and QUICK-SORT.

Reading from CLRS for week 7

- Chapter 6, Sections 6.1 - 6.5.
- Chapter 7, Sections 7.1, 7.2.
Examples

The binary tree

is level-complete (level-sizes are 1, 2, 4, 4), while

is not (level-sizes are 1, 2, 3, 6).

Second property: completeness

To have simple and efficient access to the nodes of the tree, the nodes of the last layer better are not placed in random order:

- Best is if they fill the positions from the left without gaps.
- A level-complete binary tree with such gap-less last layer is called a complete tree.
- So the level-complete binary tree on the examples-slide is not complete.
- While the running-example is complete.

Third property: the heap-property

The running-example is not a binary search tree:

- It would be too expensive to have this property together with the completeness property.
- However we have another property related to order (not just related to the structure of the tree): The value of every node is not less than the value of any of its successors (the nodes below it).
- This property is called the heap property.
- More precisely it is the max-heap property.

Definition 1

A binary heap is a binary tree which is complete and has the heap property.

More precisely we have binary max-heaps and binary min-heaps.
Fourth property: Efficient index computation

Consider the numbering (not the values) of the nodes of the running-example:

- This numbering follows the layers, beginning with the first layer and going from left to right.
- Due to the completeness property (no gaps!) these numbers yield easy relations between a parent and its children.
- If the node has number \( p \), then the left child has number \( 2p \), and the right child has number \( 2p + 1 \).
- And the parent has number \( \lfloor p/2 \rfloor \).

Efficient array implementation

For binary search trees we needed full-fledged trees (as discussed in week 7):

- That is, we needed nodes with three pointers: to the parent and to the two children.
- However now, for complete binary trees we can use a more efficient array implementation, using the numbering for the array-indices.

So a binary heap with \( m \) nodes is represented by an array with \( m \) elements:

- C-based languages use 0-based indices (while the book uses 1-based indices).
- For such an index \( 0 \leq i < m \) the index of the left child is \( 2i + 1 \), and the index of the right child is \( 2i + 2 \).
- While the index of the parent is \( \lfloor (i − 1)/2 \rfloor \).

The idea of heapification

- The input is an array \( A \) and index \( i \) into \( A \).
- It is assumed that the binary trees rooted at the left and right child of \( i \) are binary (max-)heaps, but we do not assume anything on \( A[i] \).
- After the "heapification", the values of the binary tree rooted at \( i \) have been rearranged, so that it is a binary (max-)heap now.

For that, the algorithm proceeds as follows:

- First the largest of \( A[i] \), \( A[l] \), \( A[r] \) is determined, where \( l = 2i \) and \( r = 2i + 1 \) (the two children).
- If \( A[i] \) is largest, then we are done.
- Otherwise \( A[i] \) is swapped with the largest element, and we call the procedure recursively on the changed subtree.
Analysing heapification

Obviously, we go down from the node to a leaf (in the worst case), and thus the running-time of heapification is linear in the height $h$ of the subtree.

This is $O(lg n)$, where $n$ is the number of nodes in the subtree (due to $h = \lceil lg n \rceil$).

The idea of building a binary heap

One starts with an arbitrary array $A$ of length $n$, which shall be re-arranged into a binary heap. Our example is

$$A = (4, 1, 3, 2, 16, 9, 10, 14, 8, 7).$$

We repair (heapify) the binary trees bottom-up:

- The leaves (the final part, from $\lceil n/2 \rceil + 1$ to $n$) are already binary heaps on their own.
- For the other nodes, from right to left, we just call the heapify-procedure.

Roughly we have $O(n \cdot lg n)$ many operations:

- Here however it pays off to take into account that most of the subtrees are small.
- Then we get run-time $O(n)$.

So building a heap is linear in the number of elements.
Heapify and remove from last to first

The idea of HEAP-SORT

Now the task is to sort an array $A$ of length $n$:

1. First make a heap out of $A$ (in linear time).
2. Repeat the following until $n = 1$:
   - The maximum element is now $A[1]$, swap that with the last element $A[n]$, and remove that last element, i.e., set $n := n - 1$.
   - Now perform heapification for the root, i.e., $i = 1$. We have a binary (max-)heap again (of length one less).

The run-time is $O(n \cdot \log n)$.

All basic operations are (nearly) there

Recall that a (basic) (max-)priority queue has the operations:

- MAXIMUM
- DELETE-MAX
- INSERTION.

We use an array $A$ containing a binary (max-)heap (the task is just to maintain the heap-property!):

- For deleting the maximum element, we put the last element $A[n]$ into $A[1]$, decrease the length by one (i.e., $n := n - 1$), and heapify the root (i.e., $i = 1$).
- And we add a new element by adding it to the end of the current array, and heapifying all its predecessors up on the way to the root.

Examples

Using our running-example, a few slides ago for HEAP-SORT:

Considering it from (a) to (j), we can see what happens when we perform a sequence of DELETE-MAX operations, until the heap only contains one element (we ignore here the shaded elements — they are visible only for the HEAP-SORT).

And considering the sequence in reverse order, we can see what happens when we call INSERTION on the respective first shaded elements (these are special insertions, always inserting a new max-element).
Analysis

- **MAXIMUM** is a constant-time operation.
- **DELETE-MAX** is one application of heapification, and so need time $O(\lg n)$ (where $n$ is the current number of elements in the heap).
- **INSERTION** seems to up to the current height many applications of heapification, and thus would look like $O((\lg n)^2)$, but it’s easy to see that it is $O(\lg n)$ as well (see the tutorial).

Remark on ranges

In the book arrays are 1-based:

- So the indices for an array $A$ of length $n$ are $1, \ldots, n$.
- Accordingly, a sub-array is given by indices $p, \ldots, r$.

For Java-code we use 0-based arrays:

- So the indices are $0, \ldots, n - 1$.
- Accordingly, a sub-array is given by indices $p < r$, meaning the range $p, \ldots, r - 1$.

Range-bounds for a sub-array are here now always left-closed and right-open!

So the whole array is given by the range-parameters $0, n$.

The idea of QUICK-SORT

Remember MERGE-SORT:

- A divide-and-conquer algorithm for sorting an array in time $O(n \cdot \lg n)$.
- The array is split in half, the two parts are sorted recursively (via MERGE-SORT), and then the two sorted half-arrays are merged to the sorted (full-)array.

Now we split along an element $x$ of the array:

- We partition into elements $\leq x$ (first array) and $> x$ (second array).
- Then we sort the two sub-arrays recursively.
- Done!

The main procedure

```java
public static void sort(final int[] A, final int p, final int r) {
    assert(A != null);
    assert(p >= 0);
    assert(p <= r);
    assert(r <= A.length);
    final int length = r - p;
    if (length <= 1) return;
    place_partition_element_last(A, p, r);
    final int q = partition(A, p, r);
    assert(p <= q);
    assert(q < r);
    sort(A, p, q);
    sort(A, q+1, r);
}
```
The idea of partitioning in-place

The code

Instead of \( i \) we use \( q = i + 1 \):

```java
private static int partition(final int[] A, final int p, final int r) {
    assert (p+1 < r);
    final int x = A[r-1];
    int q = p;
    for (int j = p; j < r-1; ++j) {
        final int v = A[j];
        if (v <= x) {
        }
    }
    return q;
}
```

Selecting the pivot

The partitioning-procedure expects the partitioning-element to be the last array-element. So for selecting the pivot, we can just choose the last element:

```java
private static void place_partition_element_last(final int[] A, final int p, final int r) {}
```

However this makes it vulnerable to “malicious” choices, so we better randomise:

```java
private static void place_partition_element_last(final int[] A, final int p, final int r) {
    final int i = p+(int)Math.random()*(r-p);
}
```
A not unreasonable tree

Worst-case

However, as the tutorial shows:

The worst-case run-time of QUICK-SORT is $\Theta(n^2)$
(for both versions)!

- This can be repaired, making also the worst-case run-time $\Theta(n \cdot \log n)$.
- For example by using median-computation in linear time for the choice of the pivot.
- However, in practice this is typically not worth the effort!

Average-case

If we actually achieve that both sub-arrays are at least a constant fraction $\alpha$ of the whole array (in the previous picture, that’s $\alpha = 0.1$), then we get

$$T(n) = T(\alpha \cdot n) + T((1 - \alpha) \cdot n) + \Theta(n).$$

That’s basically the second case of the Master Theorem (the picture says it’s similar to $\alpha = \frac{1}{2}$), and so we would get

$$T(n) = \Theta(n \cdot \log n).$$

And we actually get that:

- for the non-randomised version (choosing always the last element as pivot), when averaging over all possible input sequences (without repetitions);
- for the randomised version (choosing a random pivot), when averaging over all (internal!) random choices; here we do not have to assume something on the inputs, except that all values are different.

HEAP-SORT on sorted sequence

What does HEAP-SORT on an already sorted sequence? And what’s the complexity? Consider the input sequence

$$1, 2, \ldots, 10.$$
**Simplifying insertion**

When discussing insertion into a (max-)priority-queue, implemented via a binary (max-)heap, we just used a general addition of one element into an existing heap:

- The insertion-procedure used heapification up on the path to the root.
- Now actually we have always special cases of heapification — namely which?

**QUICK-SORT on constant sequences**

What is QUICK-SORT doing on a constant sequence, in its three incarnations:

- pivot is last element
- pivot is random element
- pivot is median element?

One of the two sub-arrays will have size 1, and QUICK-SORT degenerates to an \( O(n^2) \) algorithm (which does nothing).

What can we do about it?

We can refine the partition-procedure by

- not just splitting into two parts,
- but into three parts: all elements < \( x \), all elements = \( x \), and all elements > \( x \).

Then we choose the pivot-index as the middle index of the part of all elements = \( x \). We get \( O(n \log n) \) for constant sequences.

**Change to the partitioning procedure**

What happens if we change the line

```c
```

of function partition to

```c
```

- Can we do it?
- Would it have advantages?

**Worst-case for QUICK-SORT**

Consider sequences without repetitions, and assume the pivot is always the last element:

- What is a worst-case input?
- And what is QUICK-SORT doing on it?

Every already sorted sequence is a worst-case example! QUICK-SORT behaves as with constant sequences.

Note that this is avoided with randomised pivot-choice (and, of course, with median pivot-choice).
Worst-case $O(n \log n)$ for QUICK-SORT

How can we achieve $O(n \log n)$ in the worst-case for QUICK-SORT?

- The point is that just choosing, within our current framework, the median-\textit{element} is not enough, but we need the change the framework, allowing to compute the median-\textit{index}.
- Best is to remove the function \texttt{place\_partition\_element\_last}, and leave the partitioning fully to function \texttt{partition}.

Then the main procedure becomes (without the asserts):

```java
public static void sort(final int[] A, final int p, final int r) {
    if (r-p <= 1) return;
    final int q = partition(A, p, r);
    sort(A, p, q); sort(A, q+1, r);
}
```