Heapsort

Richard Hua

Semester 2, 2021

Abstract data types

An Abstract Data Type (ADT) is a mathematically specified collection of objects together with operations that can be performed on them, subject to certain rules.
Purely theoretical concept, not to be confused with language-specific features (e.g. Java abstract class).
Commonly used in algorithm design and analysis.
ADT is independent of any implementation.

Example

Examples of ADT includes queues, stacks, graphs or even integers and float point number.

Priority queue

Definition

A **priority queue** is a container ADT where each element has a key (from a totally ordered set, as with comparison-based sorting) called its priority. There are operations allowing us to insert an element, and to find and delete the element of the highest priority.

Priority queues are very common in algorithms.

- Discrete event simulation.
- Graph algorithms (later in the course).
- Sorting.

Can be implement in many ways.

Exercise

Show how a queue and a stack can be interpreted as special cases of a priority queue.

Continued

Exercise

Show how a queue and a stack can be interpreted as special cases of a priority queue.

If we use the push time stamp value of items into a stack, how should we set the priority rule to ensure the priority queue behave like a stack?

Priority queue sort

The following algorithm is obviously correct:
Suppose we are given a list of n items. Start with an empty priority queue Q and

- Successively insert all elements into Q, using the sorting keys as the priority.
- Successively remove the highest priority element until Q is empty.

Performance depends on the time complexity of push/pop operations.

Continued

- Successively insert all elements into Q, using the sorting keys as the priority.
- Successively remove the highest priority element until Q is empty.

Performance depends on the time complexity of push/pop operations.

Example

- Using an unordered list to implement Q.
- Using an ordered list to implement Q.

Can we do better?

Binary heap

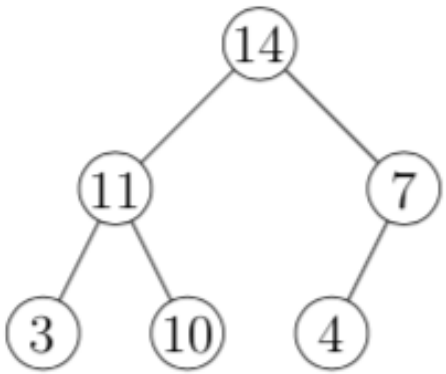
Definition

A **binary heap** is a binary tree that

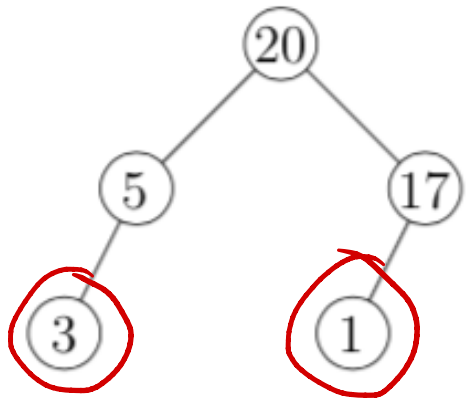
- is **left-complete**. Every level except perhaps the last is full and the last level is left-filled);
- has the **partial order property**. On every path from the root, the keys decrease (or increase).

Example

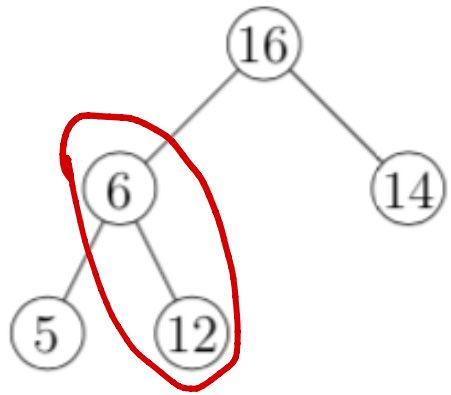
Are the following trees valid binary heap?



✓
✓



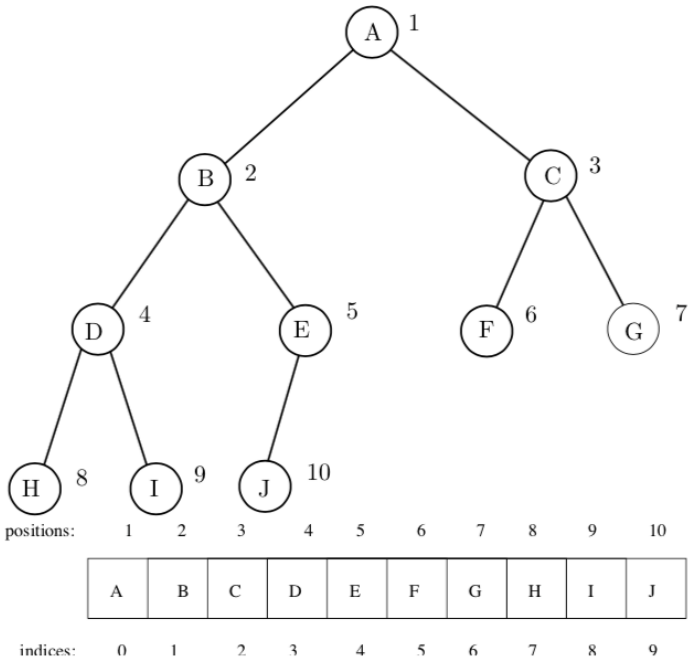
X



X

Array-based representation of heap

Binary trees in general are not easily represented using arrays.



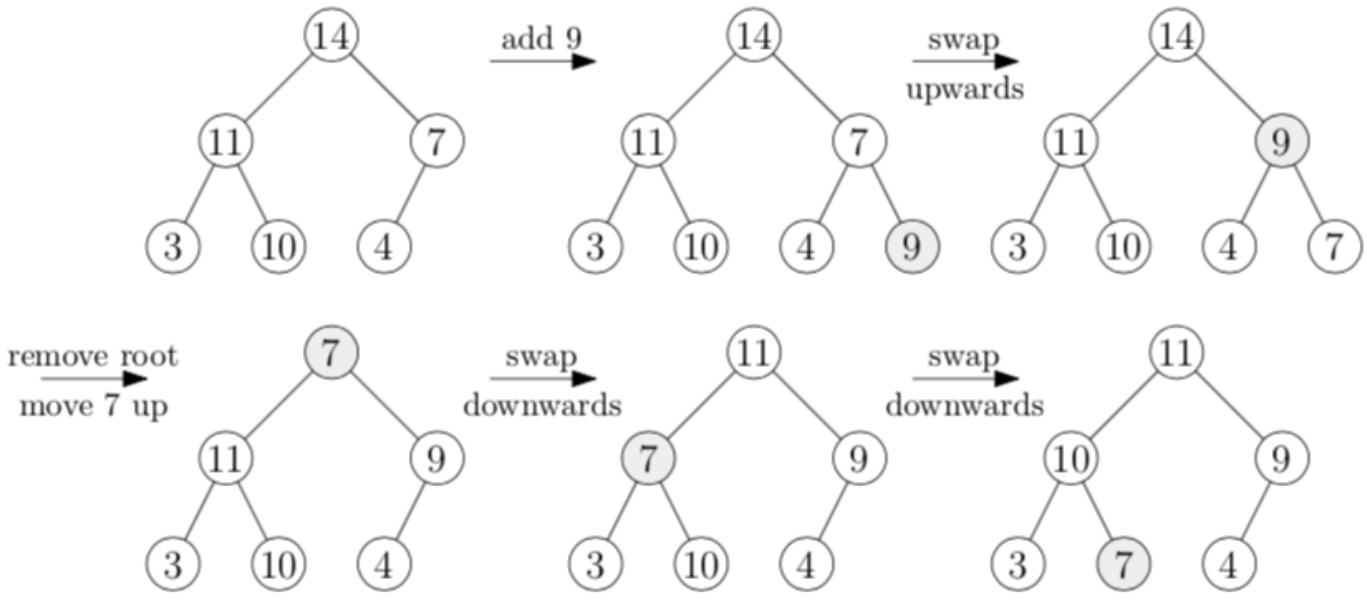
Level-order traversal of binary heap can be easily stored in an array. Node in position p has its parent, left and right child in position $\lfloor p/2 \rfloor$, $2p$ and $2p+1$ respectively (note that the position index starts at 1 instead of 0).

Binary heap

A (maximum) heap can implement a priority queue (for maximum item value) as follows:

- Keys are stored in nodes.
- Maximum item is at the root.
- To insert a node, create a new leaf at the bottom level as far left as possible. Swap it upwards (sometimes called bubble up) until no swap is required. is required.
- To delete the maximum, remove the root. Put the rightmost leaf in the root position. Swap it downward (bubble down) until no swap is required.

Heap operation example



Heap operation time complexity

Question

What is the time complexity of bubble up/down?

We make 1 comparison and 1 swap per level, the overall all time is bounded by the height of the heap i.e. $\Theta(h)$.

Binary heap height

Lemma

The height of a complete binary tree with n nodes is at most $\lfloor \lg n \rfloor$.

Proof.

Depending on the number of nodes at the bottom level, a complete tree of height h contains at least $2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^h$ nodes and at most $2^0 + \cdots + 2^h = 2^{h+1} - 1$ nodes. So $2^h \leqslant n < 2^{h+1}$, $h \leqslant \lg n < h + 1$. □

Heapsort analysis

Given a list of n items, we need to build a heap first.
Suppose we do it iteratively (i.e. we insert the nodes one by one into an initially empty heap).
After each insertion, we need to bubble up the new node.
Overall time is $\lfloor \lg 1 \rfloor + \lfloor \lg 2 \rfloor + \cdots + \lfloor \lg n \rfloor \leqslant \sum_{i=\phi}^{n} \lg n = \lg n! \approx \lg(\sqrt{2\pi n} n^n e^{-n}) = \lg \sqrt{2\pi n} + \lg n^n + \lg e^{-n} \in \Theta(n \lg n)$.
The repeated action of removing the root n times is $O(n \lg n)$.
Heapsort is $\Theta(n \lg n)$ in all cases.

Stable

Question

Is Heapsort stable?

No.

Exercise

Find an example that shows Heapsort is not stable.

In-place

Question

Is Heapsort in-place?

In-place

Question

Is Heapsort in-place?

What we have seen is not, we need extra space for the heap.
Can we make it in-place?

Continued

Observation: For Heapsort to work, we need a heap with n elements.
The iterative heap-building algorithm maintains the heap property after each insertion.
This is unnecessary (discovered in 1964 by Robert W. Floyd 1936 - 2001, Turing Award 1978).

Continued

Lemma

A complete binary tree satisfies the (max) heap property if and only if the maximum key is at the root, and the left and right subtrees of the root also satisfy the heap property with respect to the same total order.

Proof is very straightforward, we leave it as an exercise.

Recursive heap-building algorithm

Lemma

We can build a heap recursively in $\Theta(n)$ time from a list of size n.

This doesn't seem helpful for what we want....

Continued

Lemma

We can build a heap recursively in $\Theta(n)$ time from a list of size n.

Proof.

Let $T(h)$ denote the worst-case time to build a heap of height at most h. To construct the heap, each of the two subtrees attached to the root are first transformed into heaps of height at most $h-1$ (the right subtree could be of height $h-2$). Then in the worst case, the root need to bubble down the tree for a distance of at most h steps that takes time $O(h)$.
Recurrence formula is $T(h) = 2T(h-1) + ch$, $T(0) = 0$.

Continued

Proof.

$T(h) = 2T(h-1) + h$. Substitute $T(h-1) = 2T(h-2) + (h-1)$ back into the RHS we get $T(h) = 2^2 T(h-2) + h + 2(h-1)$. Do it again and we get $T(h) = 2^3 T(h-3) + h + 2(h-1) + 2^2(h-2)$. At the end we get $T(h) = \sum_{i=1}^{h} i2^{h-i} = 2^{h+1} - h - 2 \in \Theta(2^h)$. □

Continued

And it turns out we don't need the recursion at all.
The key at each position p bubbles down only after all of its descendants have been processed by the same bubble down procedure.
We apply this procedure in reverse level order so that when p is being processed, all of its descendants have already been bubbled down.
Obviously, leaves do not need to bubble down so we start at the non-leaf node with the highest position index.

Pseudo-code

function MAKEHEAP(list $a[0 \cdots n-1]$)
    $k \leftarrow n-1$
    $i \leftarrow \lfloor \frac{n-1}{2} \rfloor$
    while $i \geqslant 0$ **do**
        BUBBLEDOWN(a, i, k)
        $i \leftarrow i-1$
    return a

Pseudo-code

function HEAPSORT(list $a[0 \cdots n-1]$)
    $a \leftarrow$ MAKEHEAP(a)
    $k \leftarrow n-1$

    while $k > 0$ **do**
        SWAP($a[k], a[0]$)
        $k \leftarrow k-1$
        BUBBLEDOWN($a, 0, k$)
    return a

Example