Topic 15: Heaps and Heapsort

(CLRS 6)

CPS 230, Fall 2001

1 Introduction

- Data structures play an important role in algorithms design.
  - we will now discuss priority queues and structures for maintaining ordered sets.

2 Priority Queue

- A priority queue supports the following operations on a set S of n elements:
  - INSERT: Insert a new element e in S.
  - FINDMIN: Return the minimal element in S.
  - DELETEMIN: Delete the minimal element in S.
- Sometimes we are also interested in supporting the following operations:
  - CHANGE: Change the key (priority) of an element in S.
  - DELETE: Delete an element from S.
- We can obviously sort using a priority queue:
  - Insert all elements using INSERT.
  - Delete all elements in order using FINDMIN and DELETEMIN.
- Priority queues have many other applications, e.g. in scheduling, discrete event simulation, and graph algorithms.

2.1 Array or List implementations

- The first implementation that comes to mind is sorted array:

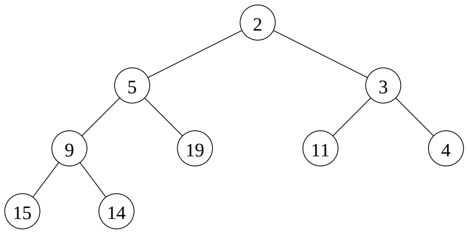
1	3	5	6	7	8	9	11	12	15	17

  - FINDMIN can be performed in $O(1)$ time.
  - DELETEMIN and INSERT takes $O(n)$ time since we need to expand/compress the array after inserting or deleting element.

- If the array is unordered (i.e., not sorted), then all operations take $O(n)$ time.
- We could use a doubly-linked sorted list instead of an array to avoid the $O(n)$ cost of expansion or compression.
  - But INSERT will still take $O(n)$ time.

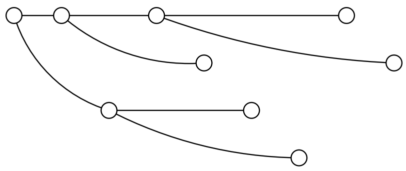
2.2 Heap implementation

- One way of implementing a priority queue is using a heap.
  - Note: We use a min-heap; the book describes a max-heap.
- Heap definition:
  - Perfectly balanced binary tree.
    * Lowest level can be incomplete (but filled from left-to-right).
  - For all non-root nodes v, we have $key(v) \geq key\big(parent(v)\big)$.
- Example:



- Heap can be implemented (stored) in two ways (at least):
  - Using pointers.
  - In an array level-by-level, left-to-right. The array representation of previous example is pictured below. The left son is pictured on the same vertical level as the parent, and the right son is pictured lower.

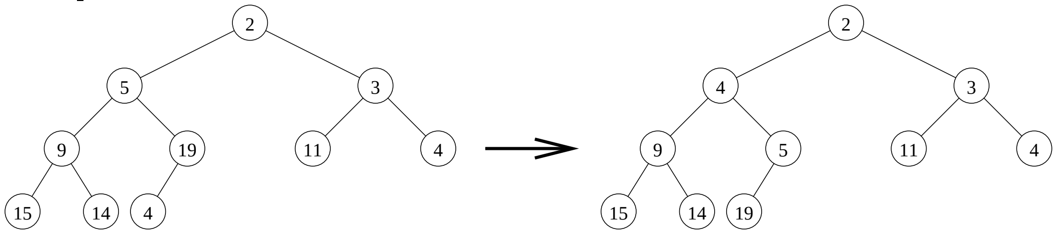
2	5	3	9	19	11	4	15	14



    * Note the nice property that the left and right children of the node stored in entry i are stored in entries $2i$ and $2i+1$, respectively.
- Properties of heap:
  - Height $\Theta(\log n)$.
  - Minimum of S is stored in root.

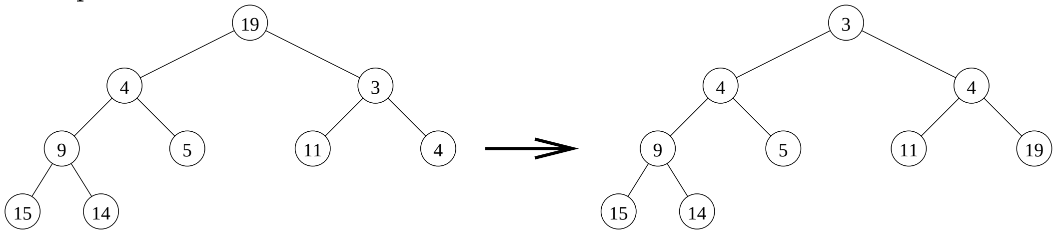
- Operations:
  - INSERT
    - ∗ Insert the new element in a new leaf at the leftmost possible position on lowest level.
    - ∗ Repeatedly swap the newly inserted element with the element in its parent node until the heap order is reestablished (UP-HEAPIFY).

    Example: Insertion of 4



  - FINDMIN
    - ∗ Return root element.
  - DELETEMIN
    - ∗ Delete element in root.
    - ∗ Move element from rightmost leaf on lowest level to the root (and delete leaf).
    - ∗ Repeatedly swap the element with the smaller of its two children until the heap order is reestablished (DOWN-HEAPIFY).

    Example:



- All operations traverse at most one root-leaf path $\implies$ $O(\log n)$ running time..
- CHANGE and DELETE can be handled similarly in $O(\log n)$ time.
  - Assuming that we know the element to be changed/deleted.

2.3 Heapsort

- Sorting using a heap, which we call *Heapsort*, takes $\Theta(n \log n)$ time.
  - $n \cdot O(\log n)$ time to insert all elements into the heap.
  - $n \cdot O(\log n)$ time to output the sorted elements by a sequence of DELETEMIN operations.
- Sometimes we would like to build a heap in $O(n)$ time rather than in $O(n \log n)$ time. For example, we can get a constant-factor improvement in sorting if we come up with a very efficient way of initially constructing the heap of n elements. The DELETEMIN operations still take $O(n \log n)$ time.

- Place elements in any order in a perfectly balanced tree.
- DOWN-HEAPIFY all nodes level-by-level, bottom-up.

Correctness:

- Induction on height of tree: When doing level i, all trees rooted at level $i-1$ are heaps.

Analysis to show $O(n)$ time for DOWN-HEAPIFY:

- Define leaves to be on level 1 (root on level $\log n$).
- n elements $\implies$ $\leq \lceil \frac{n}{2} \rceil$ leaves $\implies$ $\lceil \frac{n}{2^h} \rceil$ elements on level h.
- Cost of DOWN-HEAPIFY on a node on level h is h.
- Total cost: $\Theta\big(\sum_{i=1}^{\log n} h \cdot \lceil \frac{n}{2^h} \rceil\big) = \Theta\big(n \sum_{i=1}^{\log n} \frac{h}{2^h}\big)$.
- $\sum_{i=1}^{\log n} \frac{h}{2^h} < 2$, and thus cost of DOWN-HEAPIFY is $\Theta(n)$. Proof:
  * Assume $|x| < 1$ and differentiate $\sum_{h=0}^{\infty} x^h = \frac{1}{x-1}$.
  * Derivative of LHS is $\sum_{h=0}^{\infty} hx^{h-1}$, and derivative of RHS is $\frac{1}{(x-1)^2}$.

  $\implies$ $\sum_{h=0}^{\infty} hx^h = \frac{x}{(x-1)^2}$ $\implies$ $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1/2-1)^2} = 2$.