Heaps. Heapsort.

(CLRS 6)

1 Introduction

So far we have discussed tools necessary for analysis of algorithms (growth, summations and recurrences) and we have seen a couple of sorting algorithms as case-studies.

Today we discuss a data structure called *priority queue*, and its implementation with a heap. The heap will lead us to a different algorithm for sorting, called *heapsort*.

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
- Priority queues have many applications, e.g. in discrete event simulation, graph algorithms
- We can obviously sort using a priority queue:
  - Insert all elements using Insert
  - Delete all elements in order using FindMin and DeleteMin

3 Priority Queue implementations

3.1 A Priority Queue with an Array or List

- The first implementation that comes to mind is ordered 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.

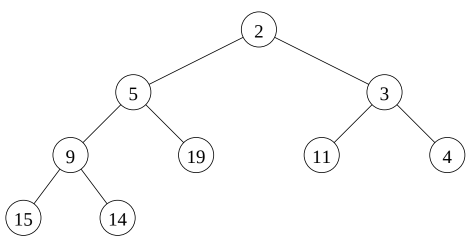
- If the array is unordered all operations take $O(n)$ time.
- We could use double linked sorted list instead of array to avoid the $O(n)$ expansion/compression cost
  - but Insert can still take $O(n)$ time.

3.2 A Priority Queue with a Heap

- The common way of implementing a priority queue is using a heap
- Heap definition:

  - Perfectly balanced binary tree
    - lowest level can be incomplete (but filled from left-to-right)
  - For all nodes v we have key$(v) \geq$key(parent(v))

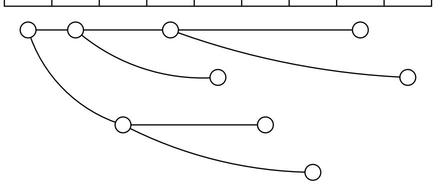
- Note: this is a *min-heap*; a symmetrical definition is possible, giving a *max-heap*.
- Example:



- The beauty of heaps is that although they are trees, they can be implemented as arrays. The elements in the heap are stored level-by-level, left-to-right in the array.

  Example:

2	5	3	9	19	11	4	15	14



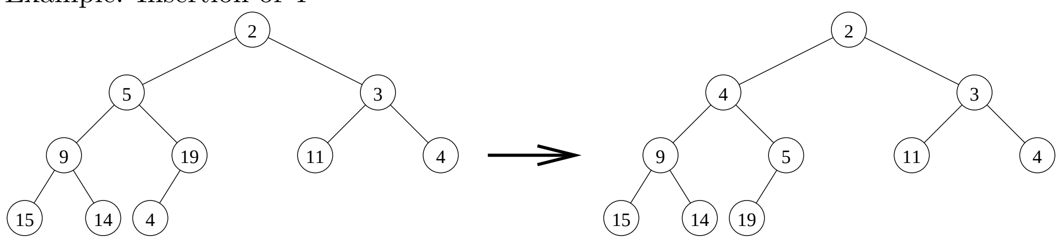
  - the left and right children of node in entry i are in entry $2i$ and $2i+1$, respectively
  - the parent of node in entry i is in entry $\lfloor \frac{i}{2} \rfloor$

- Properties of heap:

  - Height $\Theta(\log n)$
  - For a min-heap: Minimum of S is stored in root (for a max-heap, the maximum element is stored in the root).

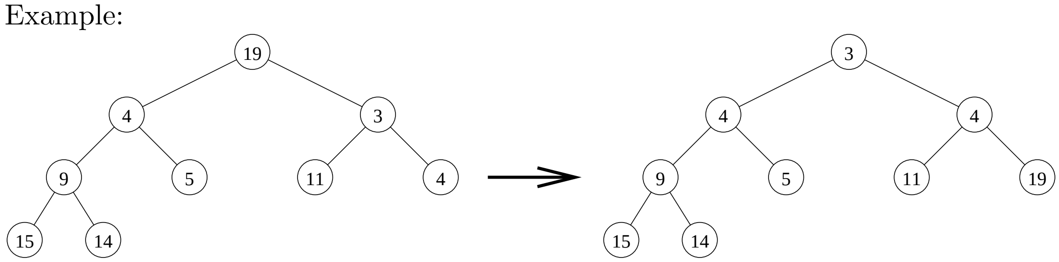
- Operations:
  - Insert
    * Insert element in new leaf in leftmost possible position on lowest level
    * Repeatedly swap element with element in parent node until heap order is reestablished (this is refered to as up-heapify.

      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 element with the smaller of the children elements until heap order is reestablished (this is refered to as down-heapify or sometimes just Heapify).

      Example:



  - By default Heapify works on the root node ($i = 1$). Heapify(i) means it's called on node i in the heap. Prior to this call, the left and right children of node i must be heaps. After Heapify (i) is complete, the tree rooted at node i is a heap.
  - Changing the priority of a given node or deleting a given node can be handled similarly in $O(\log n)$ time.
    * Note: We can delete or update nodes in a heap if we are given their index in the array. For e.g. we cannot say "delete the node with priority 37" because we cannot search (efficiently) in a heap! But we can say "delete the node at index 5".

- **Running time:** All operations traverse at most one root-leaf path $\Rightarrow O(\log n)$ time.

3.3 Heapsort

- Sorting using heap takes $\Theta(n \log n)$ time.
  - $n \cdot O(\log n)$ time to insert all elements (build the heap)
  - $n \cdot O(\log n)$ time to output sorted elements
- This is not in place. An in-place sorting algorithm with a heap is possible, and is reffered to as *heapsort*.
  - Build a max-heap
  - Repeatedly, delete the largest element, and put it at the end of the array.

3.4 Building a heap in $O(n)$ time

- Sometimes we would like to build a heap faster than $O(n \log n)$
- By default HEAPIFY works on the root node ($i = 1$). HEAPIFY(i) means it's called on node i in the heap. Prior to this call, the left and right children of node i must be heaps. After HEAPIFY (i) is complete, the tree rooted at node i is a heap.
  - BUILDHEAP (A)
    - DOWN-HEAPIFY all nodes level-by-level, bottom-up (starting at node $n/2$)
  - Correctness:
    - Induction on height of tree: When doing level i, all trees rooted at level $i-1$ are heaps.
  - Analysis:
    - The leaves are at height 0, the root is at height $\log n$
    - Cost of DOWN-HEAPIFY on a node at height h is h
    - n elements $\Rightarrow \leq \lceil \frac{n}{2} \rceil$ leaves, ..., $\lceil \frac{n}{2^h} \rceil$ elements at height h
    - Total cost: $\sum_{i=1}^{\log n} h \cdot \lceil \frac{n}{2^h} \rceil = \Theta(n) \cdot \sum_{i=1}^{\log n} \frac{h}{2^h}$
    - It can be shown that $\sum_{i=1}^{\log n} \frac{h}{2^h} = O(1) \Longrightarrow$ the total buildheap cost is $\Theta(n)$