

Heaps. Heapsort.

[Reading: CLRS 6]

Laura Toma, csci2000, Bowdoin College

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*.

1 Priority Queue

- A priority queue maintains a set of elements which have priorities. For example: the elements can be numbers, and in this case the priority is the value; the elements can be people records containing first name, last name and age, where the priority of a person is the age. Generally speaking imagine an element as containing possibly several fields, and one of them is the designated priority. The goal is to store the set of elements in order to efficiently insert new elements, and find the elements of “best priority”.
- A priority queue supports the following operations on a set S of n elements:
 - Insert a new element e in S
 - Peek at the **best** element in S
 - Delete the **best** element in S
- The **best** element is the one having the **best** priority. Depending on the specific application and what priorities represent, sometimes smaller priorities are better than larger priorities (for e.g. when priority represents the error), other times larger priorities are better than smaller priorities (for e.g. when priority represents seniority). We can define **max-priority queues** and **min-priority queues**.
- A **max-priority queue** supports the following basic operations:
 - INSERT: Insert a new element e in S
 - FIND-MAX: Return the element with **max** priority in S
 - DELETE-MAX: Delete the elements with **max** priority in S
- Symmetrically, a **min-priority queue** supports the following basic operations:
 - INSERT: Insert a new element e in S
 - FIND-MIN: Return the element with **min** priority in S

- DELETE-MIN: Delete the elements with **min** priority in S
- Priority queues have many applications, e.g. in discrete event simulation, graph algorithms, process scheduling
- We can obviously sort using a priority queue:
 - Insert all elements using INSERT
 - Delete all elements in order using DELETE-MIN
- Sometimes we may require priority queues to support the following additional operations:
 - Change the priority of an element in S
 - Delete an element from S
- How to implement a priority queue? We'll focus on min-priority queues. Max-priority queues are symmetrical.

2 Implementing a priority queue with an array or list

- The first implementation that comes to mind is an ordered array:

1	3	5	6	7	8	9	11	12	15	17
---	---	---	---	---	---	---	----	----	----	----

- FIND-MIN can be performed in $O(1)$ time
- DELETE-MIN and INSERT both take $\Theta(n)$ time in the worst case since they would need to expand/compress the array after inserting or deleting element.
- If the array is unordered, then inserts are faster, because the new element can be added at the end (however in the worst case it's still $\Theta(n)$ because it needs to re-allocate the array and copy all elements over). Find-Min and Delete-Min would take $\Theta(n)$ time worst-case because they need to traverse the unordered array and search for the minimum.
- To avoid the limitations of arrays we could use an ordered double linked sorted list (to avoid the $O(n)$ expansion/compression cost)
 - FIND-MIN and DELETE-MIN are now constant time;
 - but INSERT can still take $O(n)$ time because it first needs to locate where in the list to insert the new element.

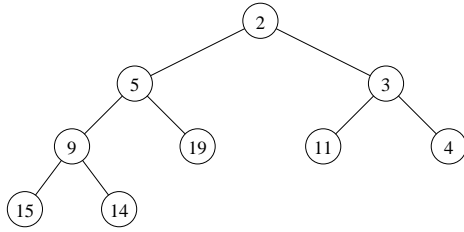
3 A Priority Queue with a Heap

The standard implementation of a priority queue is a simple and elegant structure called **heap**.

- Heap definition:

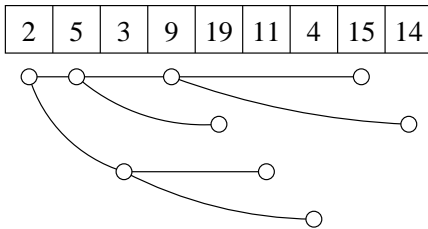
- Perfectly balanced binary tree; the lowest level can be incomplete, but filled from left-to-right.
- Heap property: For all nodes v we have $\text{priority}(v) \geq \text{priority}(\text{parent}(v))$

- Note: this is a *min-heap*; a symmetrical definition gives the *max-heap*.
- Min-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.

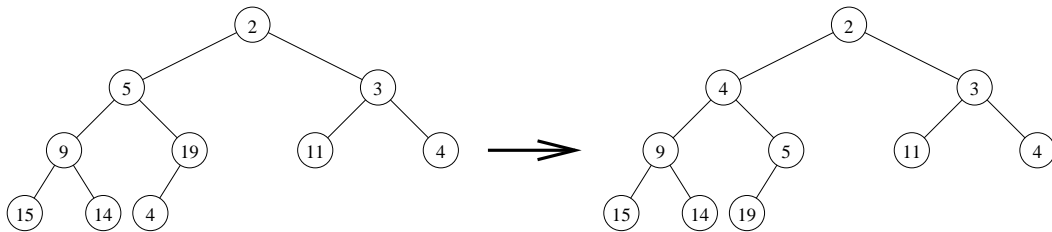
Example:



- 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 heaps:
 - 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).
- INSERT in a heap:
 - 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 referred to as UP-HEAPIFY).

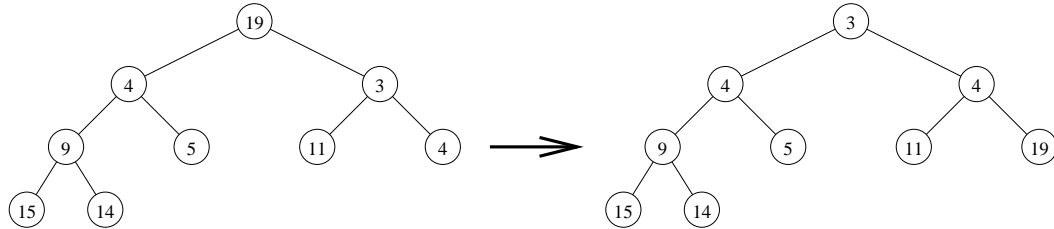
Example: Insertion of 4



- FIND-MIN in a heap:

- Return root element
- DELETE-MIN in a heap:
 - 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 referred to as DOWN-HEAPIFY or sometimes just HEAPIFY).

Example:



- HEAPIFY
 - Used by DELETE-MIN
 - Generally speaking HEAPIFY can be called on a node i : HEAPIFY(i) means it is called on node i in the heap. When calling Heapify(i) 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.
- **Running time:** All operations traverse at most one root-leaf path $\Rightarrow O(\log n)$ time.
- Other operations: CHANGE and DELETE
 - Changing the priority of a given node or deleting a given node can be handled similarly in $O(\log n)$ time. However, it's important to realize that 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”.

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

Say you got n elements in an array A , and you want to build a heap out of them.

- The straightforward way is to insert them one at a time.

BUILDHEAP-straightforward (A)

- For $i = 1$ to n : INSERT ($A[i]$)

This works but takes $\Theta(n \lg n)$ worst case.

- It is actually possible to build a heap faster, in linear time. This is a neat and simple algorithm. The idea is to call Heapify() on all nodes, going bottom-up in the tree.

BUILDHEAP-smart (A)

– For $i = n/2$ down to 1: HEAPIFY(i)

- Recall how HEAPIFY works: HEAPIFY(i) means it's called on node i in the heap. When calling HEAPIFY(i) the left and right children of node i must be heaps, and after HEAPIFY(i) is complete, the tree rooted at node i is a heap.
- Correctness: Why does this work? Note that we call Heapify bottom-up, which means that when doing level i , all trees rooted at level $i - 1$ have been already made into heaps, so the invariants of Heapify are fulfilled (Note: Formally this would require induction on height of tree).
- Analysis:
 - The leaves are at height 0, the level above is at height 1, and so on, the root is at height $\log n$
 - Cost of HEAPIFY(i) on a node i at height h is $O(h)$
 - In a heap of n elements there are $\lceil \frac{n}{2^h} \rceil$ elements at height h
 - Total cost is $\sum_{i=1}^{\log n} h \cdot \lceil \frac{n}{2^h} \rceil = \Theta(n) \cdot \sum_{i=1}^{\log n} \frac{h}{2^h} = O(1) \implies$ the total buildheap cost is $\Theta(n)$

4 Heapsort

- Sorting using a heap takes $\Theta(n \log n)$ time.
 - $O(n)$ time to build a heap.
 - $n \cdot O(\log n)$ time to call Delete-Min to get the next element in order
- This is not in place. An in-place sorting algorithm with a heap is possible, and it is referred to as *heapsort*.
 - Build a max-heap
 - Repeatedly, delete the largest element, and put it at the end of the array.