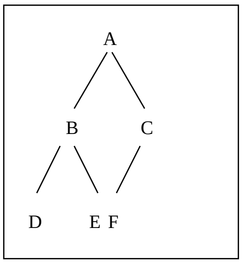
Heaps

- Definition of a heap
- What are they for: priority queues
- Insertion and deletion into heaps
- Implementation of heaps
- Heap sort

Not to be confused with: heap as the portion of computer memory available to a programmer with **new** in Java

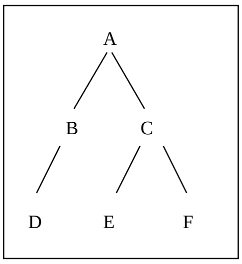
Complete Binary Trees

- Perfectly balanced, except possibly at the lowest level, and
- All the leaves at the lowest level are as far to the left as possible (it is filled from left to right level by level)



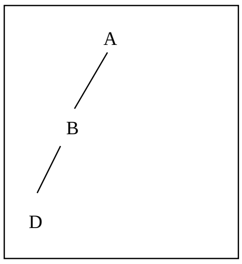
Complete Binary Trees

- NOT A COMPLETE BINARY TREE



Complete Binary Trees

- NOT A COMPLETE BINARY TREE



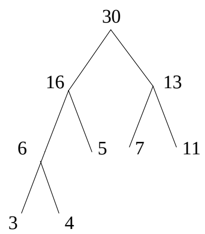
Definition of a Heap

A heap is a *complete binary tree* with the following ordering property:

The value of every parent node is greater than or equal to the values of either of its daughter nodes.

The largest node is always the root.

Example



Priority Queues

- A priority queue is a queue where the items are ordered with respect to their priority.
- If two items have the same priority, than the item which arrived first is removed first.
- A priority queue can be implemented as an ordered vector, but it is an overkill: total order is not necessary, it is enough to always have the largest item at the head of the queue.

Implementation using a heap

- Implement priority queue as a heap.
- Remove root node for dequeueing.
- Insert new node for enqueuing.

Heap ADT

- *Logical domain* : complete binary trees satisfying the heap property
- Methods:
  - **`insert(item)`**
    Pre: tree is a heap
    Post: item inserted into tree, preserving heap property

Heap methods continued

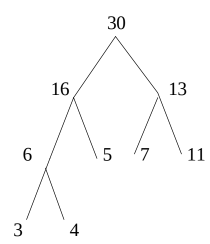
- **`remove()`**
  Pre: tree is a non empty heap
  Post: root node is removed and value returned, while preserving heap property of tree
- **`heapify(array)`**
  Post: unordered array converted to heap.

We need the latter method for heapsort.

Heap Implementation

- Since a heap is a complete binary tree, can use an array or Vector implementation which is suitable for all complete binary trees
- Root at position 0.
- Daughters of node at position i in positions 2i+1 and 2i+2.
- Parent of node at position i occupies (i-1)/2 (round towards 0).

Example



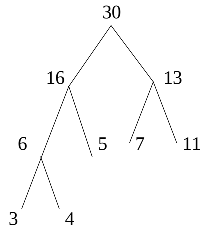
30	16	13	6	5	7	11	3	4

Heap Insertion

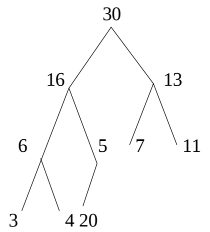
To add a new item to a heap:

- Add it as a leaf node in the next available position.
- This is likely to destroy heap property.
- If new leaf has greater value than its parent, swap values round.
- Continue swapping new value upwards until it is at a position where it is smaller than its parent.

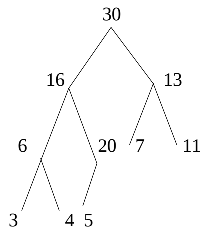
Example: insert 20



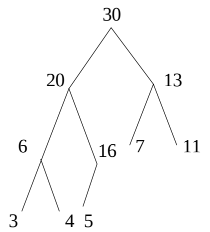
Insert in the next available position:



Bubble 20 up to correct position:



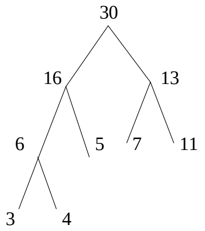
Bubble 20 up to correct position:



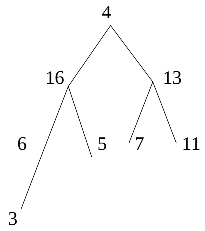
Deleting the Root from a Heap

- Replace root node by last available leaf. This may destroy the heap property
- Find which of (new) root's daughters has largest value.
- If new root smaller than largest daughter, swap values.
- Continue swapping new root value downwards until it is bigger than both its daughters.

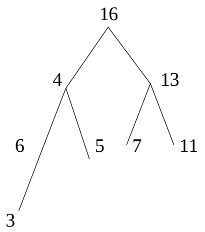
Example: delete 30



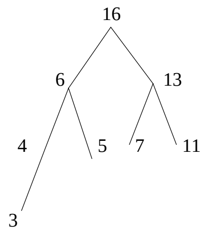
Replace 30 by 4:



Bubble 4 down:



Bubble 4 down:

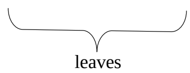


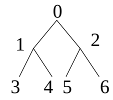
Heapifying an Array

- Start off with unordered array containing N elements, and convert it to an array representing a heap.
- All leaf nodes (positions >= N/2) trivially satisfy heap property. Start with parents of leaf nodes.
- Look at each parent in turn.
- If necessary, bubble parent value down.
- Then move to next level up, and look at each parent there.
- Bubble parent value down if necessary.
- Continue until you reach root node.

Example

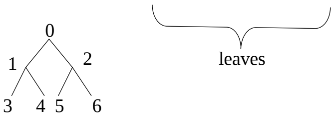
0	1	2	3	4	5	6
20	30	40	62	16	7	77





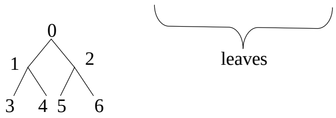
Look at parent of 7 and 77:

0	1	2	3	4	5	6
20	30	40	62	16	7	77



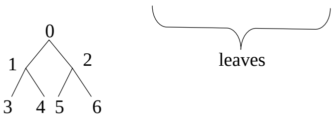
Needs bubbled down!

0	1	2	3	4	5	6
20	30	40	62	16	7	77



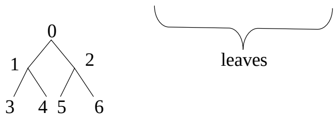
Needs bubbled down!

0	1	2	3	4	5	6
20	30	77	62	16	7	40



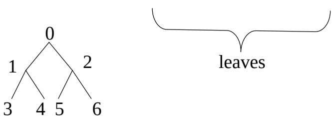
Look at parent of 62 and 16:

0	1	2	3	4	5	6
20	30	77	62	16	7	40



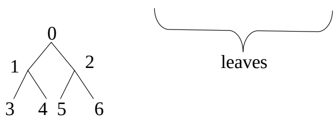
Needs bubbled down!

0	1	2	3	4	5	6
20	30	77	62	16	7	40



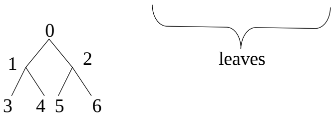
Needs bubbled down!

0	1	2	3	4	5	6
20	62	77	30	16	7	40



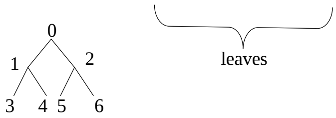
Look at parent of 62 and 77:

0	1	2	3	4	5	6
20	62	77	30	16	7	40



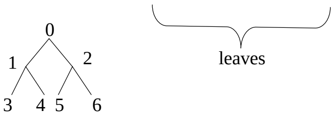
Needs bubbled down!

0	1	2	3	4	5	6
20	62	77	30	16	7	40



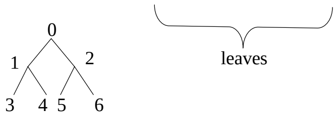
Needs bubbled down!

0	1	2	3	4	5	6
77	62	20	30	16	7	40



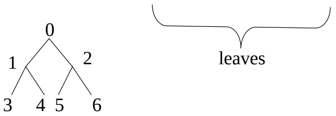
Needs bubbled down!

0	1	2	3	4	5	6
77	62	20	30	16	7	40



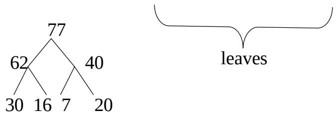
Needs bubbled down!

0	1	2	3	4	5	6
77	62	40	30	16	7	20



Needs bubbled down!

0	1	2	3	4	5	6
77	62	40	30	16	7	20



Complexity Results

- Insertion and deletion performance dominated by number of swaps necessary to bubble items up or down.
- Worst case number of swaps = depth of tree = $\log_2 N$, where N is number of nodes in tree.
- Hence, insertion and deletion is O(log N).
- To heapify, we make N/2 calls to bubbleDown method.
- Hence O(N log N).

Heap Sort

- Convert unordered array of size N to heap of size N.
- Successively remove root node from heap, and place at position N - 1 in array.
- This reduces size of heap by 1, and so decrement N by 1.
- Continue removing root node from heap of size N' and placing at position N' - 1.
- Until heap is emptied.

Complexity of Heap Sort

- Heapifying array is O(N log N).
- Removing root from heap is O(log N).
- We have to remove root N times.
- Hence removal of all roots is O(N log N).
- Hence heap sort is O(N log N).

Summary

- Heaps: Complete binary trees where every parent is larger than its daughters.
- Implemented as vectors.
- Bubbling elements up/down to insert/remove.
- Heapifying arrays.
- Heaps as a way of implementing priority queues (e.g. printer queues).
- Heap sort.