PRIORITY QUEUES (HEAPS)

by

David Toman

University of Waterloo

List of Slides

1
2 A Priority Queue ADT
3 Naive Implementation(s)
4 Binary Heap
5 Example
6 Complete Trees and Arrays
7 Example Declaration
8 Delete-max
9 Restructuring (sift-down)
10 Example
11 Insertion
12 Restructuring (sift-up)
13 Example
14 Example (cont.)
15 More Operations
16 d-Heaps
17 Applications
18 Summary

A Priority Queue ADT

- An ADT that stores a set of elements

  $\Rightarrow$ identified by, e.g., keys
  $\Rightarrow$ each with a **priority** (a number)

- Operations:

  $\Rightarrow$ **`PQcreate`**

  creates a new empty priority queue

  $\Rightarrow$ **`PQempty`**

  tests if a priority queue is empty

  $\Rightarrow$ **`PQinsert`**

  adds an element to the queue
  $\Rightarrow$ **priority** must be supplied here

  $\Rightarrow$ **`PQdeleteMax`**

  dequeue element with the highest priority

- we discuss "MAX-queue",
  "MIN-queue" is symmetric w.r.t. $<$.

Naive Implementation(s)

- list implementation(s):

  $\Rightarrow$ "insert to the beginning"

  — enqueue: $O(1)$

  — dequeue: $O(n)$

  $\Rightarrow$ "insert to a sorted list"

  — enqueue: $O(n)$

  — dequeue: $O(1)$

- search tree implementation

  — enqueue: $O(\log n)$

  — dequeue: $O(\log n)$

  lots of balancing to be done:
  we always remove the "right-most" node

Binary Heap

IDEA: simplified balanced tree

$\Rightarrow$ two properties:

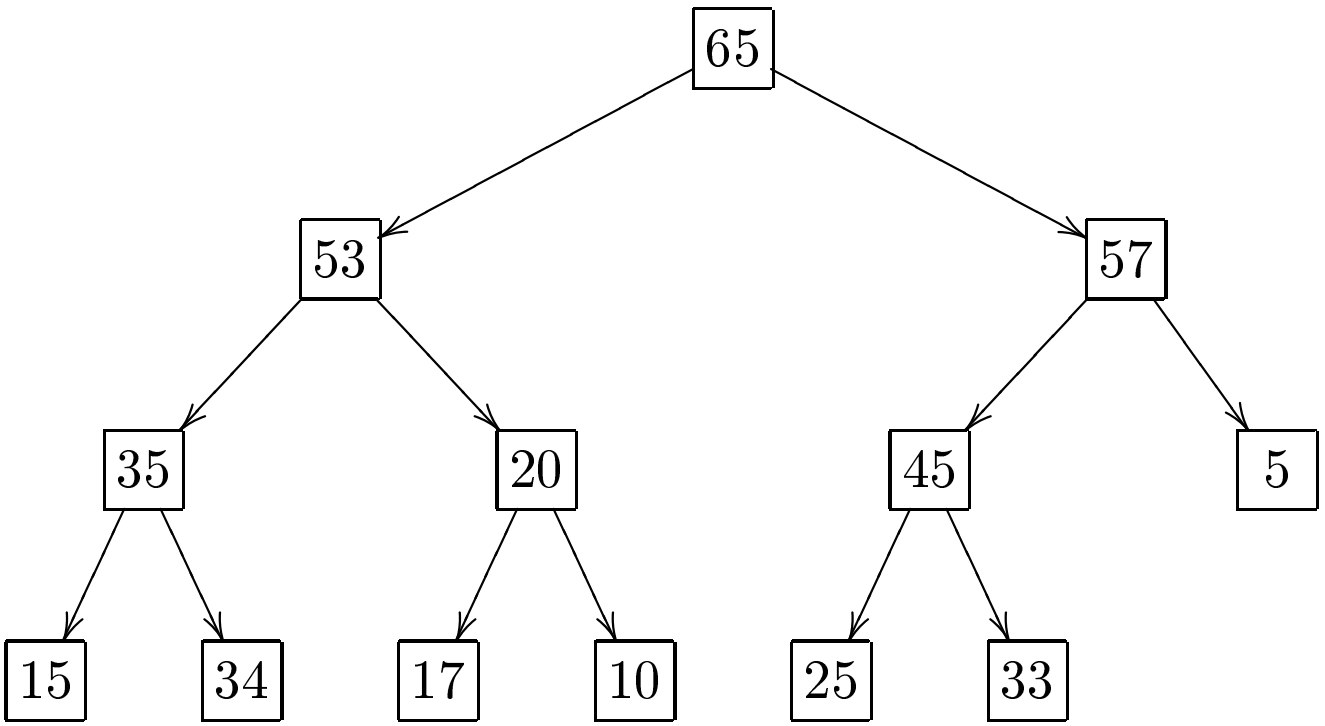
1. a *Heap Property*:

   $\Rightarrow$ the *priority* of a node is higher than
   the *priority* of its children

2. a *Structural Property*:

   complete binary tree on N nodes
   left-adjusted

Example



NOTE: there is no ordering required for children of nodes

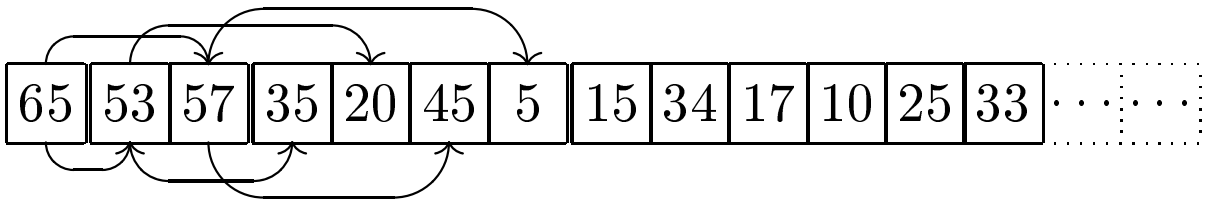
$\Rightarrow$ major difference from a search tree!

Complete Trees and Arrays

IDEA: in a compelte binary tree

$\Rightarrow$ all but the last "level" have exactly 2^i nodes

$\Rightarrow$ can be stored without use of pointers in an **array**:



For node i:

- left child: $2i$
- right child: $2i + 1$
- parent: $\lfloor i/2 \rfloor$

Example Declaration

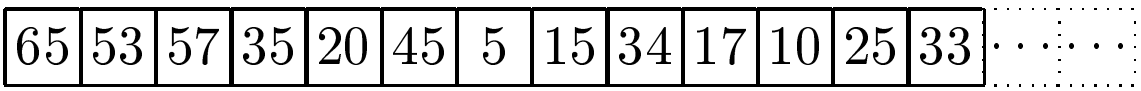
A Priority Queue with capacity N elements:

```
type PQ = record
            len: integer
            queue: array [1..N] of value;
          end;
```

Example (with 13 out of 16 elements):

len $= 13$

queue $=$



$\Rightarrow$ no pointers needed!

Delete-max

- the maximal element (with highest priority)
  is **at the root** of the heap

  $\Rightarrow$ follows from the *Heap Property*

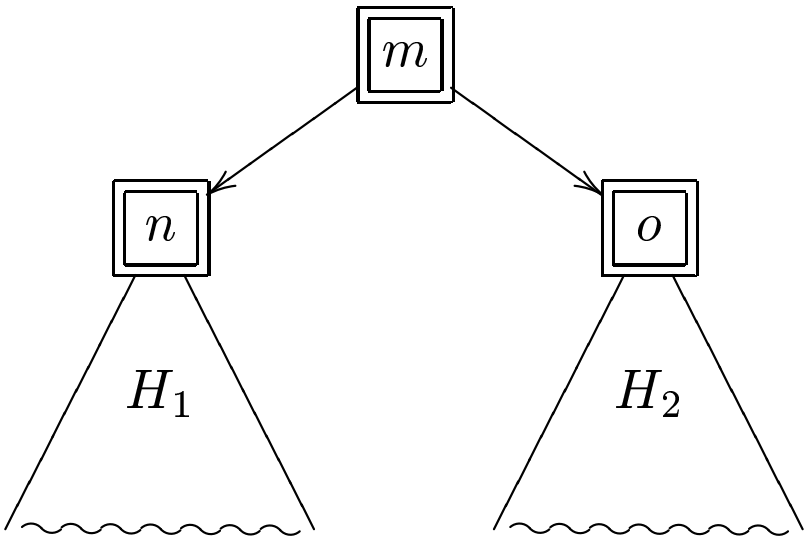
- replace by the “last” element

  $\Rightarrow$ this isn’t heap any more

- reestablish the *Heap Property*

  $\Rightarrow$ move the new root to its “rightful place”

Restructuring (sift-down)



Situations (m new, n and o roots of "valid heaps"):

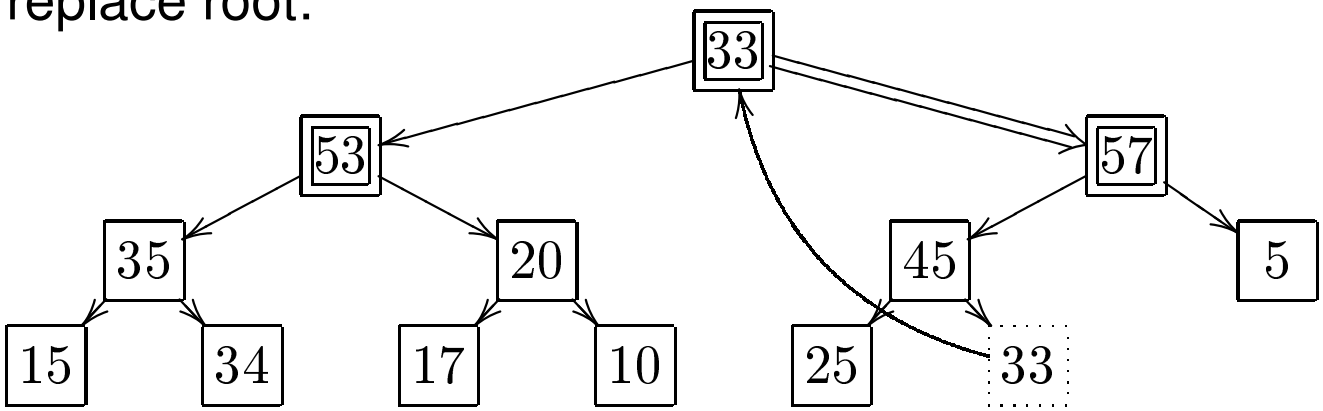
- $m > n, o$ – no problem
- $n > m, o$ – problem

  $\Rightarrow$ swap m and n and (recursively) restructure H_1
- $o > m, n$ – problem

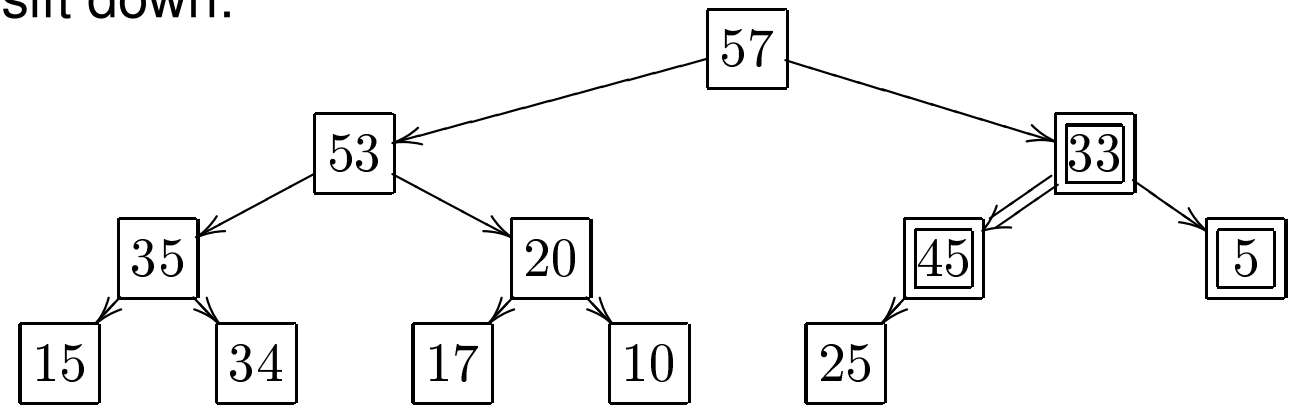
  $\Rightarrow$ swap m and o and (recursively) restructure H_2

Example

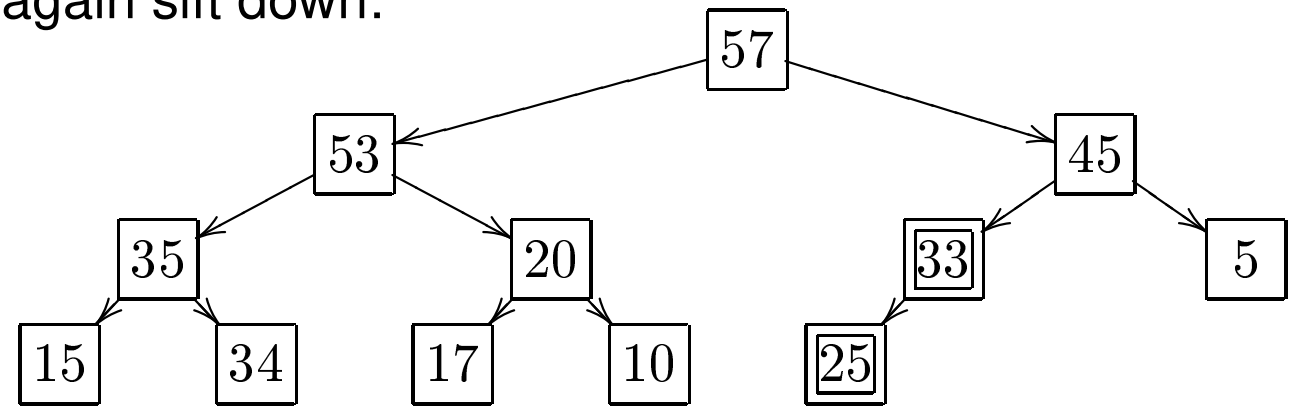
- replace root:



- sift down:



- again sift down:



- done!

Insertion

IDEA: reverse the *deleteMax* process:

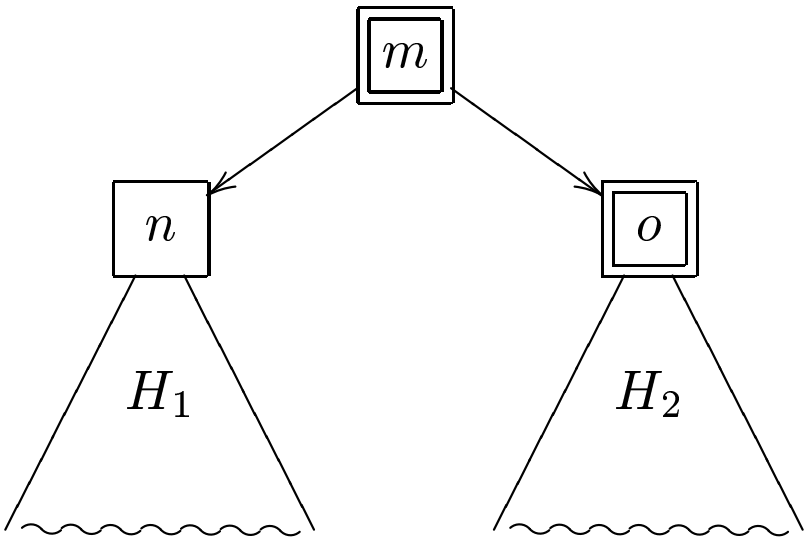
- add the new element “to the end”

  $\Rightarrow$ this violates the heap property

- restructure the heap

  by moving the new element up the tree (sift-up)

Restructuring (sift-up)



Situations (similarly for m and n):

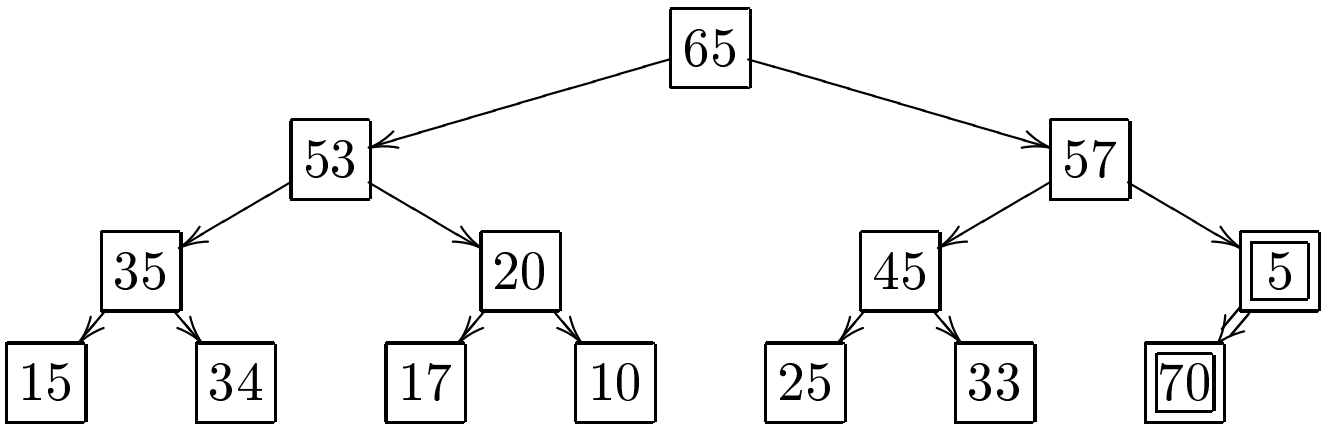
- $m > o$ – no problem
- $m < o$ – problem

  $\Rightarrow$ swap m and o and restructure w.r.t. parent of m

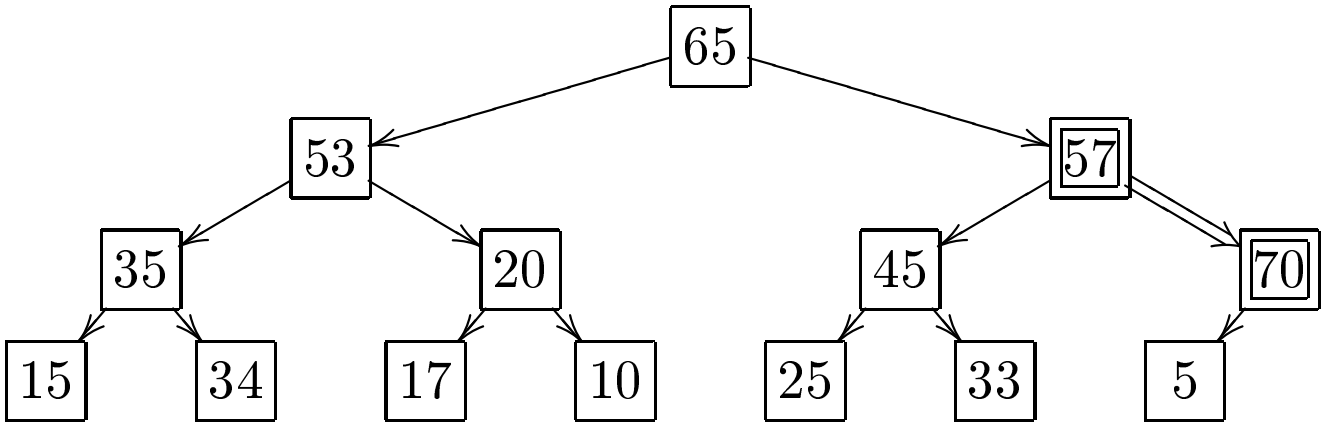
$\Rightarrow$ why don't we have to look at n?
$\Rightarrow$ why don't we have to compare m to rest of H_2?

Example

- add new element:

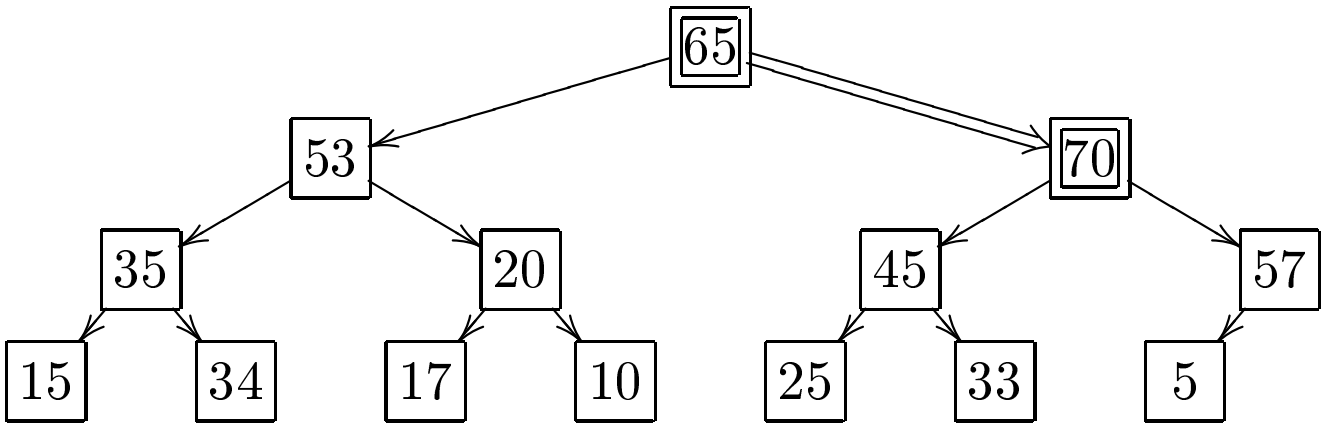


- sift up:

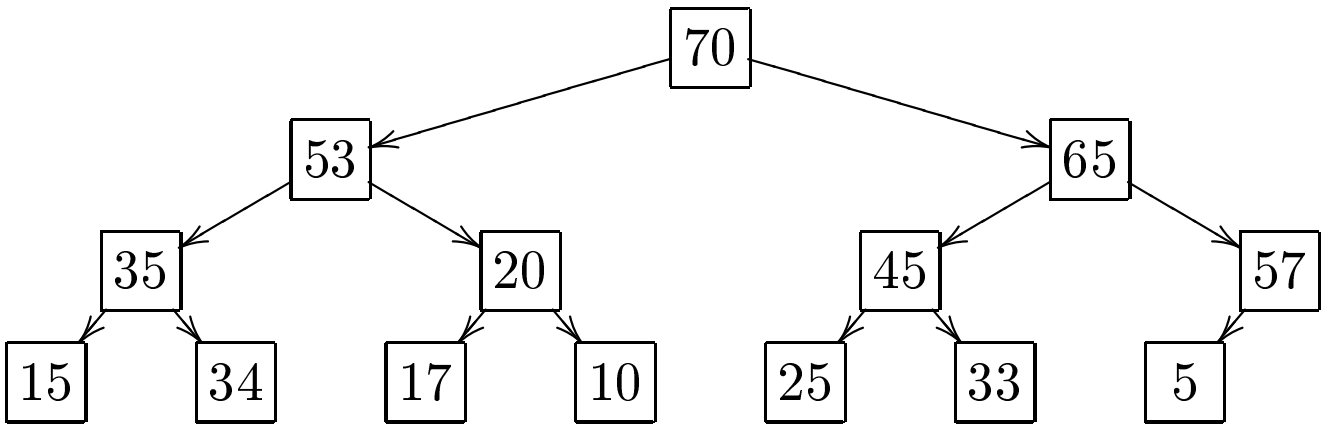


Example (cont.)

- again sift up:



- sift up last time:



- done!

More Operations

- **`PQIncreaseKey`**, **`PQdecreaseKey`**

  $\Rightarrow$ update *priority* for a record id'ed by a given key

  $\Rightarrow$ fix-up the heap:

  increase $\rightarrow$ sift-up

  decrease $\rightarrow$ sift-down

- **`PQdeleteKey`**

  $\Rightarrow$ use **`PQincreaseKey`** to make it the "largest"

  $\Rightarrow$ then execute **`PQdequeue`**

- **`PQmakeHeap`**:

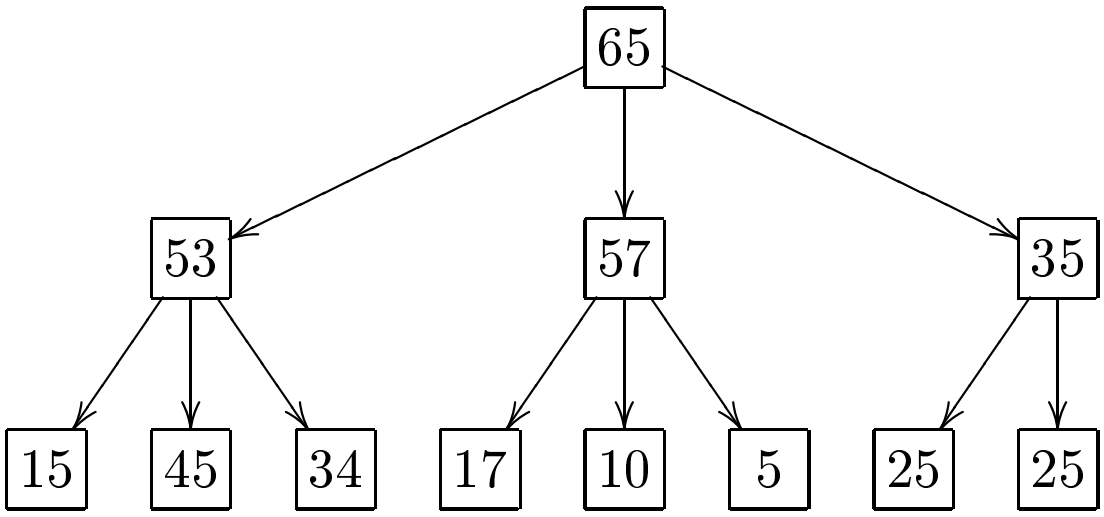
```
for i:=(N div 2) downto 1 do PQsiftDown(i);
```

  $\Rightarrow$ runs in $O(N)$ (why?)

d-Heaps

IDEA: instead of 2 children we require *exactly* d children

$\Rightarrow$ complete, left-adjusted d-ary trees



$\Rightarrow$ can be implemented in arrays

$\Rightarrow$ sift-down requires more comparisons!

Applications

- The Selection Problem (find k-th largest element)

  $\Rightarrow$ simple solution:

  (1) make-heap with N elements $O(N)$
  (2) dequeue k elements $O(k \log N)$

  $\Rightarrow$ needs a heap of size $O(N)$

  $\Rightarrow$ alternative solution:

  (1) build a MIN-heap of first k elements
  (2) for remaining elements:
  if the next one is larger than
  the one on the top of the heap then
  dequeue the minimal element
  insert the new element

  $\Rightarrow$ Total $O(N \log k)$ but uses only $O(k)$-sized Heap!

- Heap Sort

  $\Rightarrow$ make-heap out of N elements: $O(N)$
  $\Rightarrow$ dequeue them in order: $O(N \log N)$

Summary

- Heaps are extremely simple to implement

  $\Rightarrow$ way easier than trees (with balancing!)
  $\Rightarrow$ no "extra space" for pointers
  implemented using an *array*

- efficiently implement PQ's

- often sufficient to implement 'pseudo-ordering'

  $\Rightarrow$ find maximal/minimal element (k-th one)
  $\Rightarrow$ efficiently sort an array
  $\Rightarrow$ ...