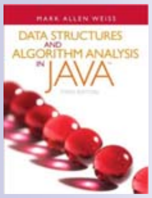
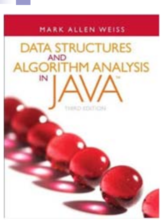


FLORIDA INTERNATIONAL UNIVERSITY

Priority Queues (Heaps)

Dr. Antonio L. Bajuelos

School of Computing & Information Sciences



COP-3530 - Data Structures

Module #4: Priority Queues

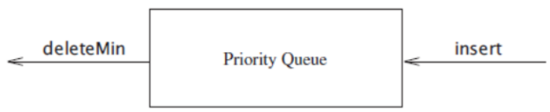
Outline:

- **Priority Queues ADT.**
- **Simple implementations of Priority Queues.**
- **Efficient implementation - Heaps.**
- **Heaps operations:**
  - **insert / add**
  - **deleteMin**
  - **Complexity analysis.**
- **Java code of heaps.**

2

Priority Queues. The Model

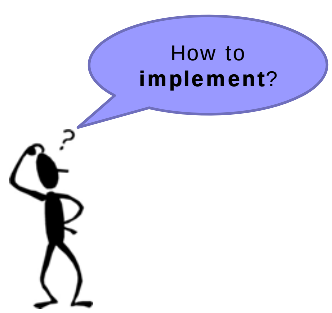
- A **priority queue** is a data structure that allows at least the following two operations:
  - **insert / add**;
  - **deleteMin**, which finds, returns, and removes the minimum element in the **priority queue**.
- Note that:
  - The **insert** operation is the equivalent of **enqueue**, and **deleteMin** is the **priority queue** equivalent of the **dequeue** operation.



Priority Queues. The Model (cont…)

- **Important points:**
  - Each item has a "**priority**"
    - For example: the minimum element is the one with the greater priority (i.e. priority "1" is more important than priority "4")
  - **Main operations: insert** and **deleteMin**
- **Example:**

  insert *x1* with priority 7
  insert *x2* with priority 5
  insert *x3* with priority 6
  a = **deleteMin** **// *x*2**
  b = **deleteMin** **// *x*3**
  insert *x4* with priority 4
  insert *x5* with priority 8
  c = **deleteMin** **// *x*4**
  d = **deleteMin** **// *x*1**



Priority Queues. Simple implementation

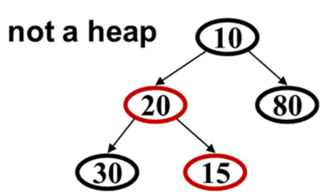
- **Priority Queues in Simple Linked-List**
  - **insertion** at the front is O(1)
  - **deleteMin** is O(N)
- **Priority Queues in Sorted Linked-List**
  - **insertion** is O(N)
  - **deleteMin** is O(1)
- **Priority Queues in Binary Search Tree**
  - **insertion** is (in average) O(logN))
  - **deleteMin** is (in average) O(logN))

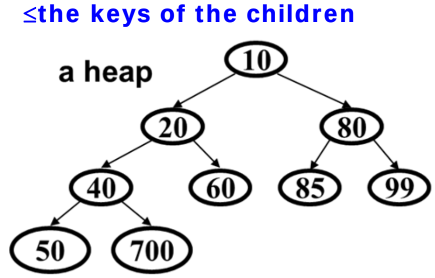
Priority Queues. Efficient implementation

A **binary min-heap** (or **binary heap** or **heap**) has:

- **Structure property:** A **complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
- **(Min) Heap property:** The priority of every (non-root) node is less important than the priority of its parent.

the key of a node **≤the keys of the children**

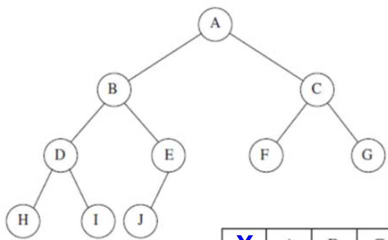




***Heap is Not** a Binary Search Tree!!!*

Priority Queues (heap). Simple representation

- **Important points:**
  - **Heap -** a **complete binary tree** is so regular then it can be represented in an **array** and no links are necessary.



X	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

  - For any element in array position ***i***, the **left child** is in position **2*i***, the **right child** is in the cell after the left child **(2*i* + 1)**, and the parent is in position ***i* / 2**.
  - Links are not required and the operations required to traverse the tree are extremely simple

Priority Queues (heap). Operations

- **findMin**: return **root.data**
- **deleteMin**:
  - **answer = root.data**
  - Move **right-most leaf** in last row to **root** to restore **structure property**
  - If necessary, **percolate down** to restore **heap-order property**
- **insert:**
  - Put new node in next position on bottom row to restore structure property
  - **Percolate up** to restore **heap-order property**

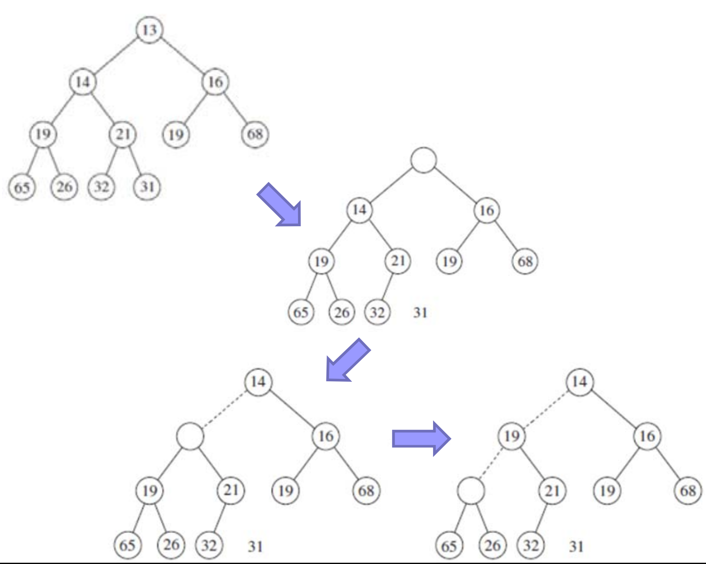
Priority Queues (heap). deleteMin

- **deleteMin. Percolate down strategy.**
  - **answer = root.data** (key at the root)
  - Replace the key at the root by the key of the last (right-most) leaf node.
  - Delete the last leaf node.
  - As long as the heap order property is violated, **percolate down**.

9

Priority Queues (heap). deleteMin

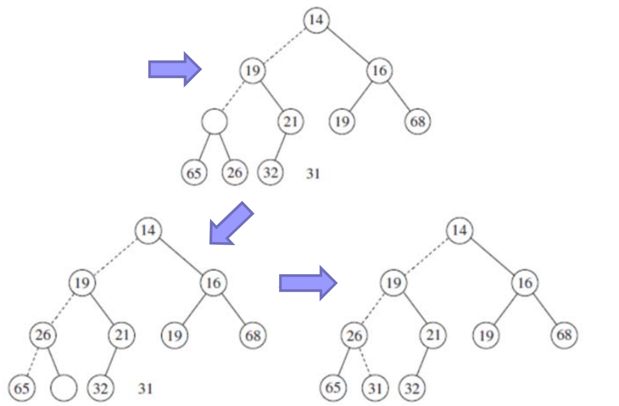
- **deleteMin. Example:**



10

Priority Queues (heap). deleteMin

- **deleteMin**:
  - **Percolate down:**
    - Keep comparing priority of item with both children.
      - If priority is less important, swap with the most important child and go down one level.
      - Done if both children are less important than the item or we've reached a leaf node.



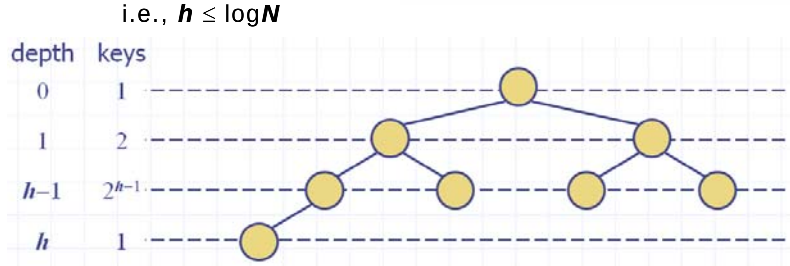
Priority Queues (heap). deleteMin

- **Theorem:** A **heap** storing ***N*** nodes has height **O(log*N*)**.

Proof: We have $N \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$

Thus, $N \geq 2^h$, by using $\sum_{k=0}^{n-1} 2^k = 1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1$

i.e., $h \leq \log N$



- **Running time of deleteMin is O(log*N*)**

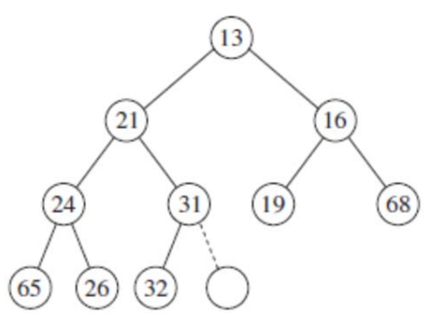
Priority Queues (heap). Insert/add

- **General Strategy**:
  - Add a value to the tree (<u>create a hole in the next available location</u>, since otherwise the tree will not be complete).
  - **Focus on restoring the heap-order property**.
- **What is the running time?**
  - Like **deleteMin**, the insert/add process (worst-case time) is proportional to tree height: **O(log*N*).**
  - But... On **average**, the percolation terminates early; it has been shown that 2.607 comparisons are required on average to perform an insert. So **insert** is, on **average**, **O(1)**.

13

Priority Queues (heap). Insert/add

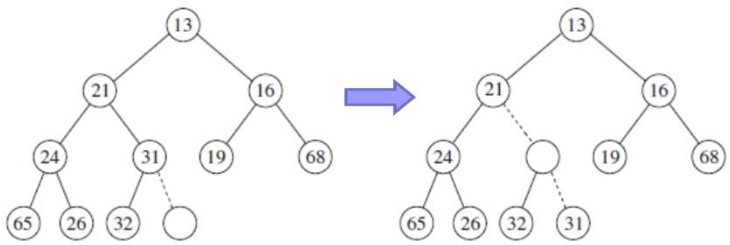
- **Insert. Percolate up strategy**:
  -
    ```
    insert (key)
    {
       if (the heap is full) throw an exception;
       insert key at the end of the heap;
       while(key is not in the root node and key < parent(key))
           swap(key,parent(key));
    }
    ```
  - **Example:** insert(14)



14

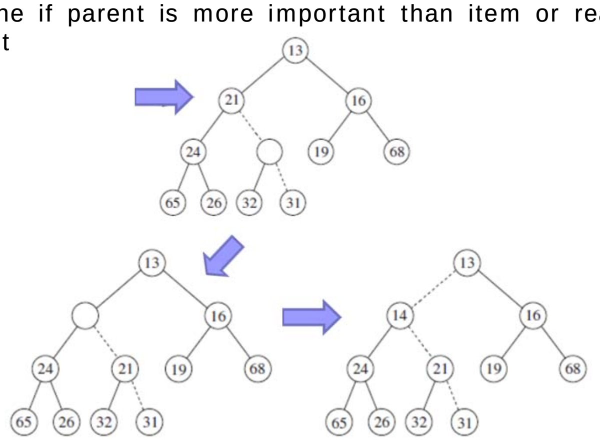
Priority Queues (heap). Insert/add

- **Example: insert(14)**:
  - **Percolate up:**
    - Put new data in new location.
    - If parent is less important, swap with parent, and continue
    - Done if parent is more important than item or reached root



Priority Queues (heap). Insert/add

- **insert(14)**:
  - **Percolate up:**
    - Put new data in new location.
    - If parent is less important, swap with parent, and continue
    - Done if parent is more important than item or reached root



Priority Queues (heap). Java Code

- **See Java code for (binary) Heap in:**

http://users.cis.fiu.edu/~weiss/cop3530_sum08/July16.java

Author: Prof. Mark Weiss

17