# *Algorithms and Data Structures*

**CMPSC 465**

Priority Queues and
Binary Heaps

**Paul Medvedev**

*based on slides by S. Raskhodnikova , A. Smith, K. Wayne, C. Leiserson and E. Demaine.*

1

## Priority Queue Abstract Data Type

- Dynamic set of pairs (key, data), called elements
- Supports operations:
  - **MakeNewPQ**()
  - **Insert**($S$,$x$) where $S$ is a PQ and $x$ is a (key,data) pair
  - **Extract-Max**($S$) removes and returns the element with the highest key value (or one of them if several have the same value)
- Example: managing jobs on a processor, want to execute job in queue with the highest priority
- Can also get a "min" version, that supports **Extract-Min** instead of **Extract-Max**
- Sometimes support additional operations like **Delete**, **Increase-Key**, **Decrease-Key**, etc.

2

## Rooted trees

- Rooted Tree: collection of nodes and edges
  - Edges go down from root (from parents to children)
  - No two paths to the same node
  - Sometimes, children have "names" (e.g. left, right, middle, etc)

## Heaps: Tree Structure

- Data Structure that implements Priority Queue
  - Conceptually: binary tree
  - In memory: (often) stored in an array
- Max-Heap property:
  - For every node other than root, $key[Parent(i)] \geq key[i]$
- Recall: *complete* binary tree
  - all leaves at the same level, and
  - every internal node has exactly 2 children
- Heaps are nearly complete binary trees
  - Every level except possibly the bottom one is full
  - Bottom layer is filled left to right

4

## Height

- Heap Height =
  - length of longest simple path from root to some leaf
  - e.g. a one-node heap has height 0,
    a two- or three-node heap has height 1, ...
- Exercises:
  - What is are max. and min. number of elements in a heap of height h?
    - ans: min = $2^h$,
      max = $2^{h+1}$ -1
  - What is height as a function of number of nodes n?
    - ans: floor( log(n) )

5

## Array representation

- Instead of dynamically allocated tree, heaps often represented in a fixed-size array
  - smaller constants, works well in hardware.
- Idea: store elements of tree layer by layer, top to bottom, left to right
- Navigate tree by calculating positions of neighboring nodes:
  - Left(i) := 2i
  - Right(i) := 2i+1
  - Parent(i) := floor(i/2)
- Example: [20 15 8 10 7 5 6]

6

## Review Questions

- Is the following a valid max-heap?
  - 20 10 4 9 6 3 2 8 7 5 12 1
  - (If not, repair it to make it valid)
- Is an array sorted in decreasing order a valid max-heap?

7

## Local Repair: Heapify "Down"

- Two important "local repair operations":
  - **Heapify "Down"**
  - Heapify "Up"
- Suppose we start from a valid heap and **decrease** key of node ***i***, so
  - it is still smaller than parent
  - the two subtrees rooted at *i* remain valid
- How do we rearrange nodes to make the heap valid again?
  - Idea: let new, smaller key "sink" to correct position
    - Find index with largest key among {*i*, Left(*i*), Right(*i*)}
    - If i != largest, EXCHANGE i with largest and recurse on largest

8

## Local Repair: Heapify "Down"

- Pseudocode in CLRS Chap 6.2: "MAX-HEAPIFY"
- Exercise: what does Max-Heapify do on the following input?
  - 2.5 10 4 9 6 3 2 8 7 5 0 1
- Running time: O(height) = O(log n)
  - Tight in worst case
- Correctness: by induction on height
  - IH: Let T be a binary tree with the left/right subtrees having the max-heap property. Then, after Max-Heapify, T has the max-heap property.
  - Ind. Step: Suppose the IH holds for all trees of height less than *h*. Then it holds for trees of height *h*.

9

## Priority Queue: Extract-Max

- This gives us our first PQ operation:
- Extract-Max(A)
  1. tmp := A[1]
  2. A[1] := A[heap-size]
  3. heap-size := heap-size-1
  4. MAX-HEAPIFY-DOWN(A,1)
  5. **return** tmp

10

## Local Repair: Heapify "Up"

- Two important "local repair operations"
  - Heapify "Down"
  - **Heapify "Up"**
- Suppose we start from a valid heap and **increase** key of node ***i***, so
  - it is still larger than both children, but
  - might be larger than parent
- How to rearrange nodes to make heap valid again?
  - Idea: let new, larger key "float" **up** to right position
    - If A[i] > A[Parent(i)], EXCHANGE i with parent and recurse on parent
    - Pseudocode in CLRS Chap 6.5: "Heap-Increase-Key"

11

## Local Repair: Heapify "Up"

- Exercise: what does Max-Heapify-Up do on the following input?
  - 20 10 4 9 6 3 2 8 7 21 0 1
- Running time: O(log n)
  - tight in worst case
- Correctness: by induction on height

12

## Priority Queue: Insert operation

- This gives us our second PQ operation:
- Insert(A,x)
  1. A[heap-size+1] := x
  2. heap-size := heap-size+1
  3. MAX-HEAPIFY-UP(A,heap-size)

13

## Other PQ operations

- Max(S): return max element without extracting it
- Increase-Key(S,i,new-key)
- Decrease-Key(S,i,new-key)
- Delete(S,i)
  - Delete the element in position *i*
  - Move the last element to position *i*. If the new key at position *i* is smaller than the old key, then call Decrease-Key at position *i*. If its larger, then call Increase-Key. Otherwise, do nothing.
- A PQ contains (key, data) pairs. The data is stored separate from the heap. But, we keep an auxiliary array of pointers together with the heap array. These pointers point from each key to its corresponding data.

14

## Heap Sort

- Heaps give an easy O(n log n) sorting algorithm:
  - For *i* =2 to n
    - Insert(Q,A[i])
  - For *i* = n downto 2
    - A[i] := Extract-Max(Q)
- There is a faster way (O(n) time) to build a heap from an unsorted array.

15