CS 240 – Data Structures and Data Management

Module 2: Priority Queues

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2023

Outline

- Priority Queues
  - Review: Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

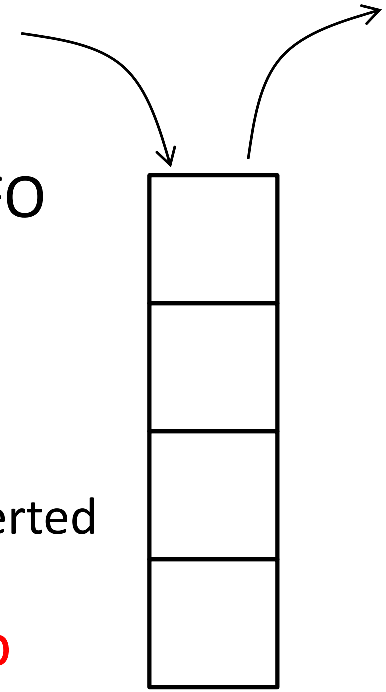
Outline

- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Abstract Data Type (ADT)

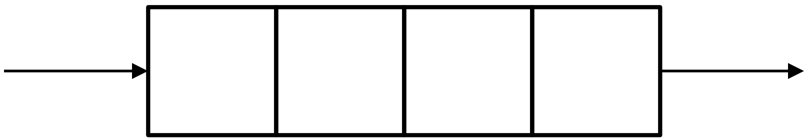
- A description of *information* and a collection of *operations* on that information
- The information accessed *only* through the operations
- ADT describes what is stored and what can be done with it, but not how it is implemented
- Can have various *realizations* of an ADT, which specify
  - how the information is stored (*data structure*)
  - how the operations are performed (*algorithms*)

Stack ADT



- ADT consisting of a collection of items removed in LIFO (last in first out order)
- Operations
  - push inserts an item
  - pop removes and typically returns the most recently inserted item
- Items enter at the top and are removed from the top
- Extra operations
  - size, isEmpty, and top
- Applications
  - addresses of recently visited sites in a Web browser, procedure calls
- Realizations of Stack ADT
  - arrays
  - linked lists

Queue ADT



- ADT consisting of a collection of items removed in FIFO (first-in first-out) order
- Operations
  - enqueue inserts an item
  - dequeue removes and typically returns the least recently inserted
- Items enter queue at the rear and are removed from front
- Extra operations
  - size, isEmpty, and front
- Realizations of Queue ADT
  - (circular) arrays
  - linked lists

Outline

- Priority Queues
  - Review: Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Priority Queue ADT

- Collection of items each having a *priority*
  - priority is also called *key*
- Operations
  - insert: insert an item tagged with a priority
  - deleteMax: remove and return the item of highest priority
    - also called extractMax
- Definition is for a maximum-oriented priority queue
- To define minimum-oriented priority queue, replace deleteMax by deleteMin
- Applications
  - typical "todo" list
  - simulation systems
  - sorting

Using Priority Queue to Sort

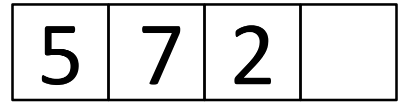
PQ-Sort$(A[0 \ldots n-1])$

initialize *PQ* to an empty priority queue
for $i \leftarrow 0$ **to** $n-1$ **do**
  PQ.insert$(A[i])$
for $i \leftarrow n-1$ **downto** 0 **do**
  $A[i] \leftarrow$ *PQ.deleteMax* ()

- $A[i]$ is item with priority $A[i]$
- Run-time depends on priority queue implementation
- Can write as $\mathrm{O}(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax})$

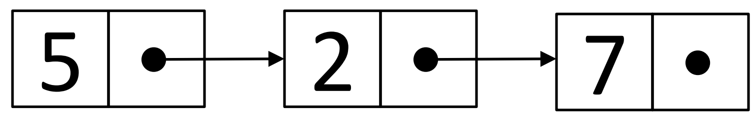
Realizations of Priority Queues

- Attempt 1: *unsorted arrays*



  - assume dynamic arrays
    - expand by doubling when needed
    - happens rarely, so amortized time over all insertions is $O(1)$
  - insert: $\Theta(1)$
  - deleteMax: $\Theta(n)$
  - PQ sort becomes $\Theta(n^2)$

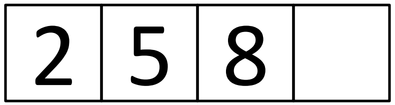
- Attempt 2: *unsorted linked lists*



  - efficiency identical to Attempt 1
  - this realization used for sorting yields *selection sort*

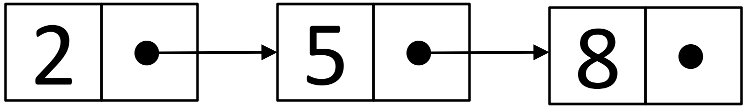
Realizations of Priority Queues

- Attempt 3: *sorted arrays*



  - Store items in order of increasing priority
  - deleteMax: $\Theta(1)$
  - insert: $\Theta(n)$
  - PQ-sort similar to InsertionSort and is $\Theta(n^2)$worst case

- Attempt 4: *sorted linked-lists*



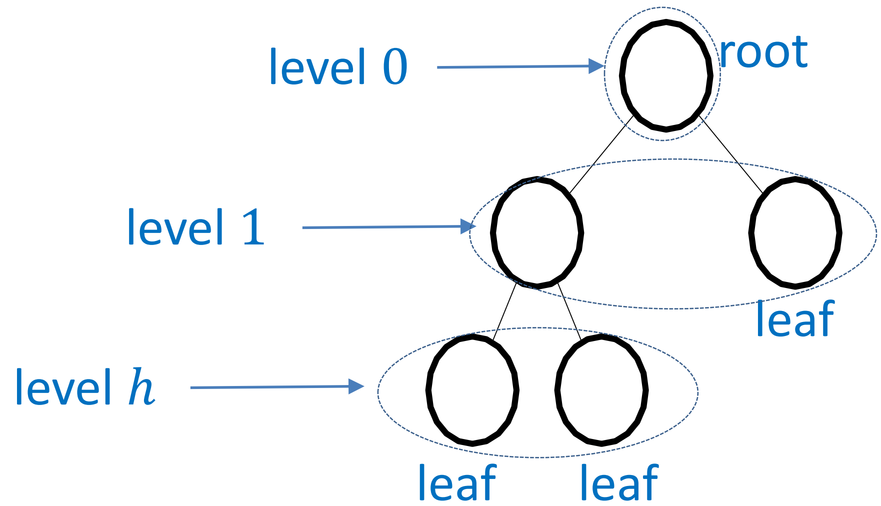
  - similar to Attempt 3

Outline

- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

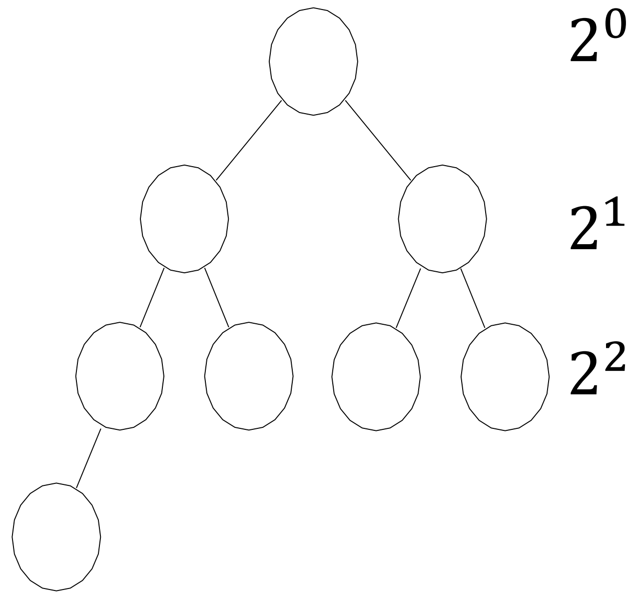
Binary Tree Review

- A *binary tree* is either
  - empty, or
  - consists of three parts
    - a node
    - left subtree
    - right subtree
- Terminology
  - root, leaf, parent, child, level, sibling, ancestor, descendant
  - height of the tree is the maximum level in the tree



Binary Tree Review

- Consider tree with n nodes of smallest possible height h
  - all levels must be as full as possible
    - except possibly the last level h
  - level 0 has 2^0 nodes
  - level 1 has 2^1 nodes
  - ...
  - level i has 2^i nodes
  - level h has between 1 and 2^h nodes



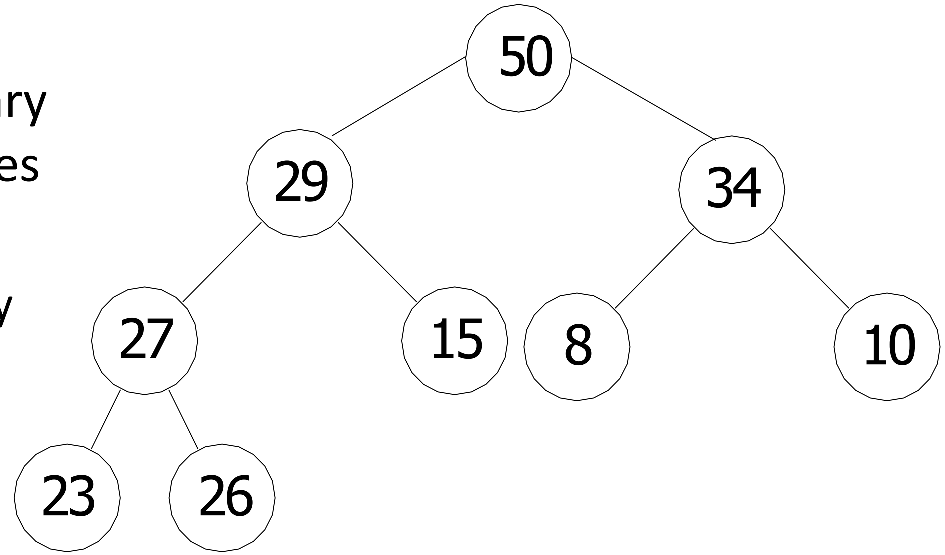
- Can bound

$$n \le 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$$

$$\begin{aligned} 2S &= \cancel{2^1} + \cancel{2^2} + \cdots + 2^{h+1} \\ -\quad S &= 2^0 + \cancel{2^1} + \cdots + \cancel{2^h} \\ \hline S &= 2^{h+1} - 1 \end{aligned}$$

- Therefore $n \le 2^{h+1} - 1$
- Simplifying, $h \ge \log(n+1) - 1$
- Binary tree height is $\Omega(\log n)$
  - height is between $n-1$ and $\log(n+1) - 1$, which is $\Omega(\log n)$
  - note use of asymptotics for function other than time complexity

Third Realization of Priority Queue: Heaps



- A *max-oriented binary heap* is a binary tree with the following two properties
  1. Structural Property
     - all levels of a heap are completely filled, except (possibly) the last level
     - items in the last level are *left-justified*
  2. Heap-order Property
     - for any node i, key[parent of i] $\geq$ key[i]

- A *min-heap* is the same, but with opposite order property

- More accurate picture of nodes

Heap Height

Lemma: Height of a heap with n nodes is $\Theta(\log n)$

- Since heap is a binary tree, height h is $\Omega(\log n)$
- Need to show that height h is $O(\log n)$
- Heap has all levels full except possibly level h
  - 2^i nodes at level $0 \leq i \leq h-1$
- Thus

at least last node at level h

$$n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1$$

$$n \geq 2^h - 1 + 1$$

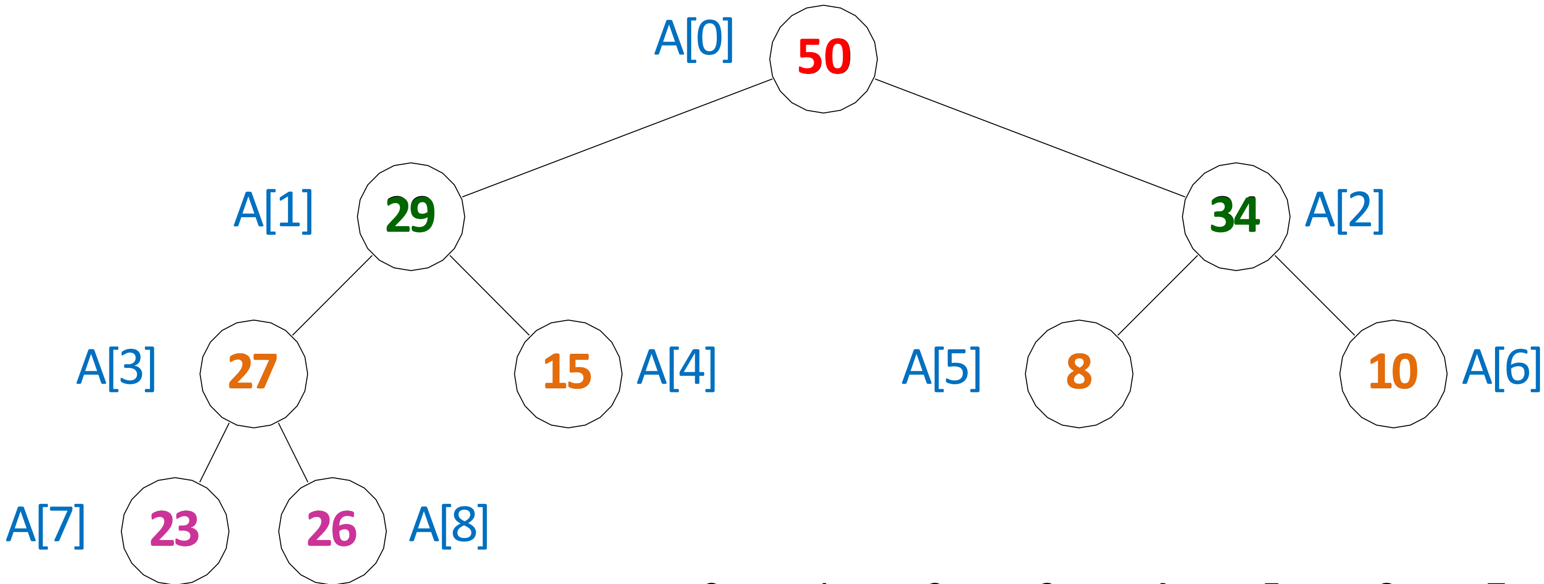
$$n \geq 2^h$$

$$h \leq \log n$$

- Thus $h \in O(\log n)$

Storing Heaps in Arrays

- Using linked structure for heaps wastes space
- Let H be a heap of n items and let A be an array of size n
  - store root in $A[0]$
  - continue storing *level-by-level* from top to bottom, in each level left-to-right



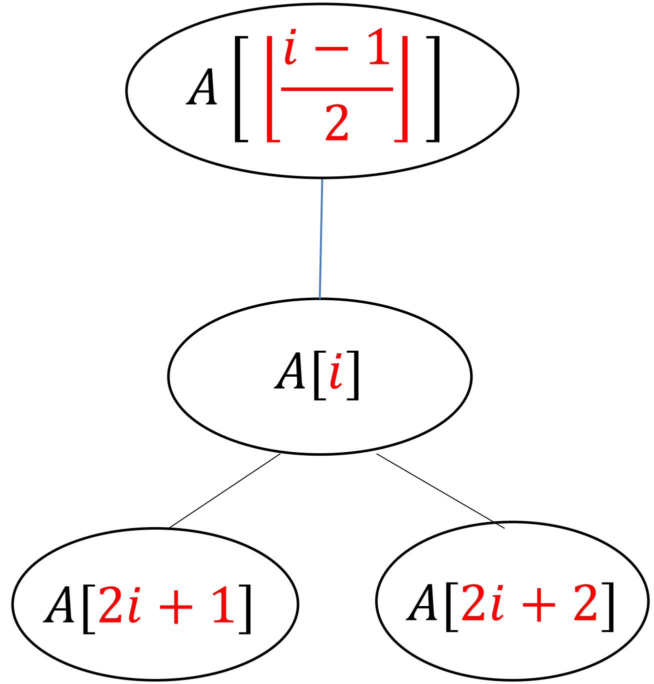
- Fits compactly into array

0	1	2	3	4	5	6	7	8
50	29	34	27	15	8	10	23	26

- Last heap node is in $A[n-1]$

Heaps in Arrays: Navigation

- *root* node is $A[0]$
- left child of $A[i]$, if exists, is $A[2i+1]$
- right child of $A[i]$, if exists, is $A[2i+2]$
- parent of $A[i]$, if exists, is $A\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right]$



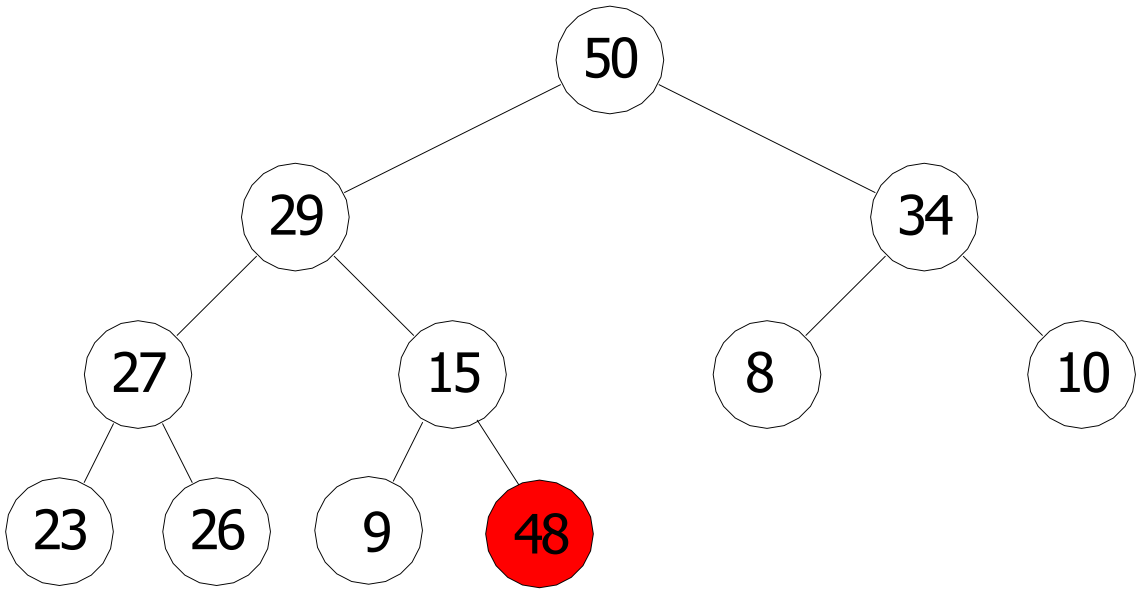
- Hide implementation details using helper-function
  - functions *root*( ), *parent*(i), *left*(i), right(i)
  - *last*() returns index of the last node in the heap
- Some of these helper functions need to know n,
  - omit it from pseudocode for simplicity

Outline

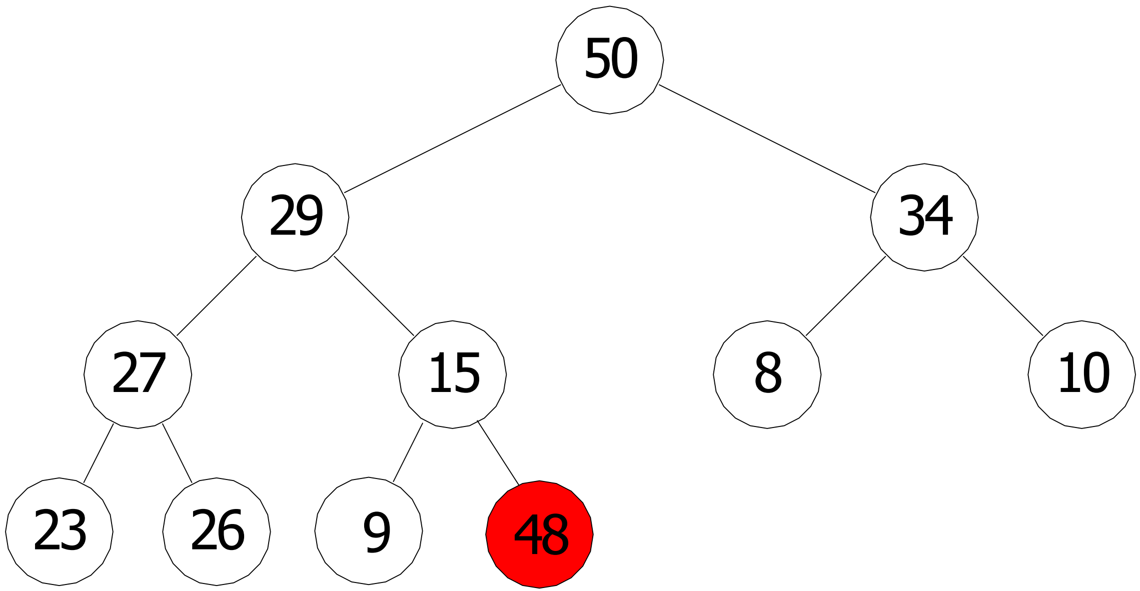
- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Insertion in Heaps

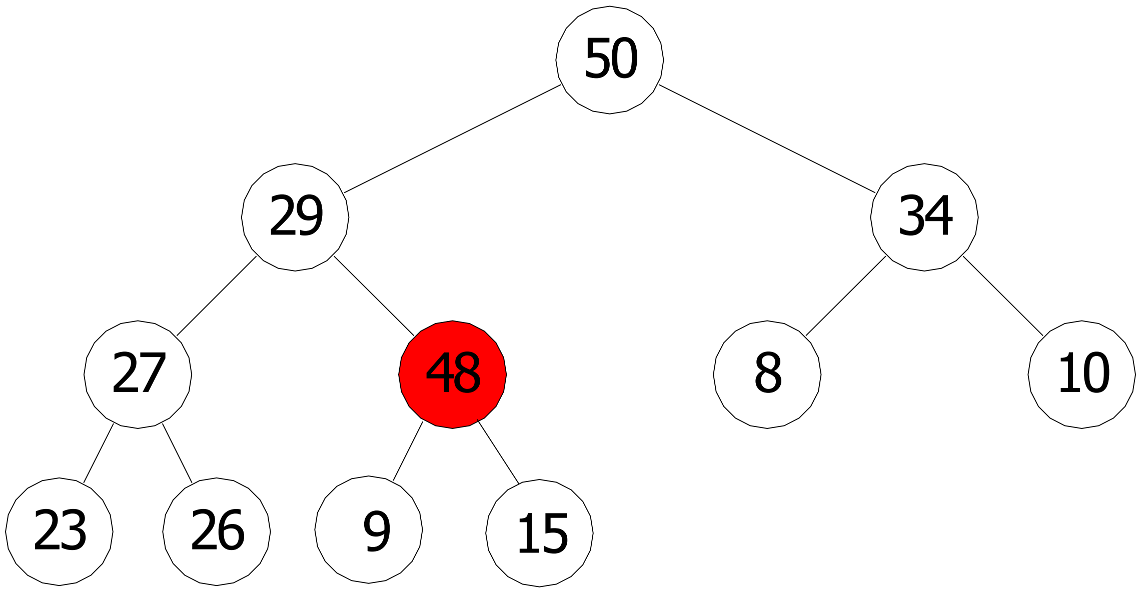
- Place new key at the first free leaf
- Heap-order property might be violated
- Perform a *fix-up*



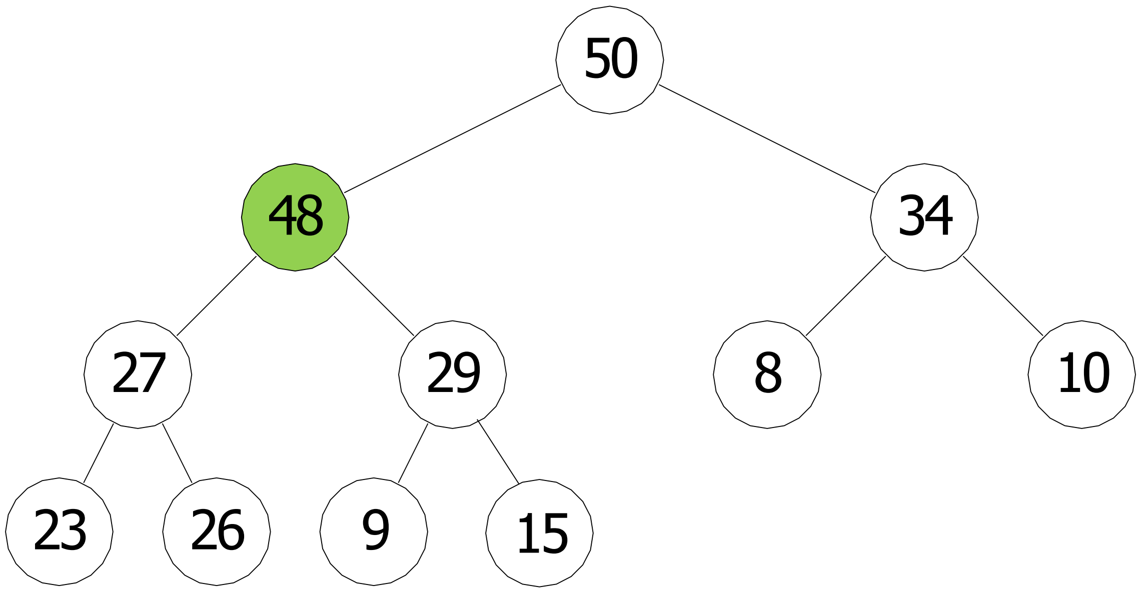
fix-up example



fix-up example



fix-up example



fix-up pseudocode

fix-up(A, i)

i: an index corresponding to heap node

while *parent*(i) exists **and** $\mathbf{A}[parent(i)].key < A[i].key$ **do**

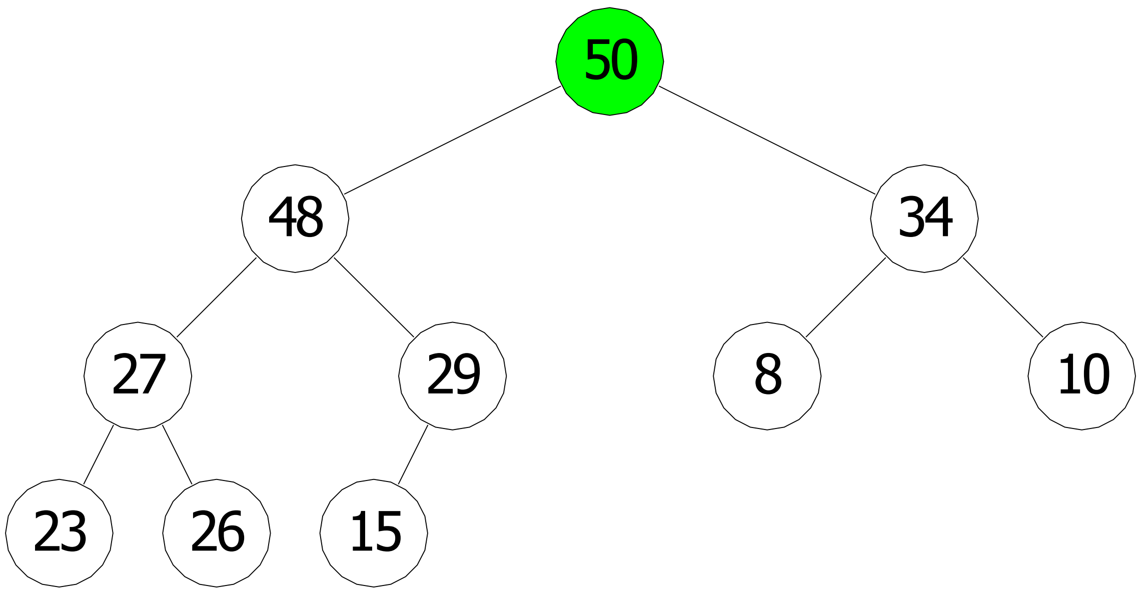
  swap $A[i]$ and $A[parent(i)]$

  $i \leftarrow parent(i)$ // move to one level up

- Worst case time complexity: $O(\text{heap height}) = O(\log n)$

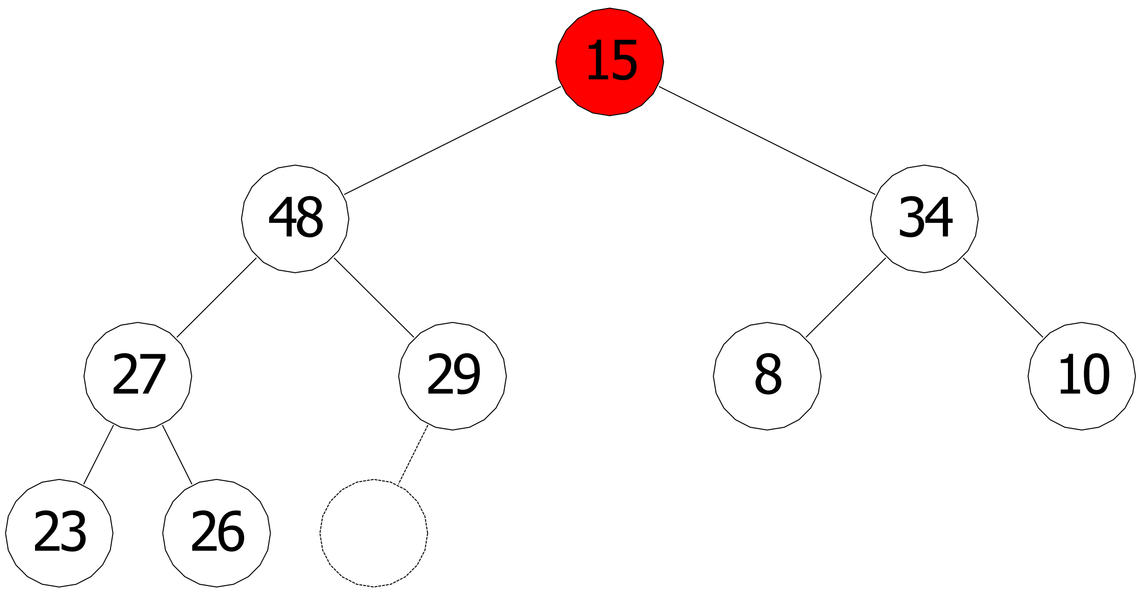
deleteMax in Heaps

- The root has the maximum item
- Replace root by the last leaf



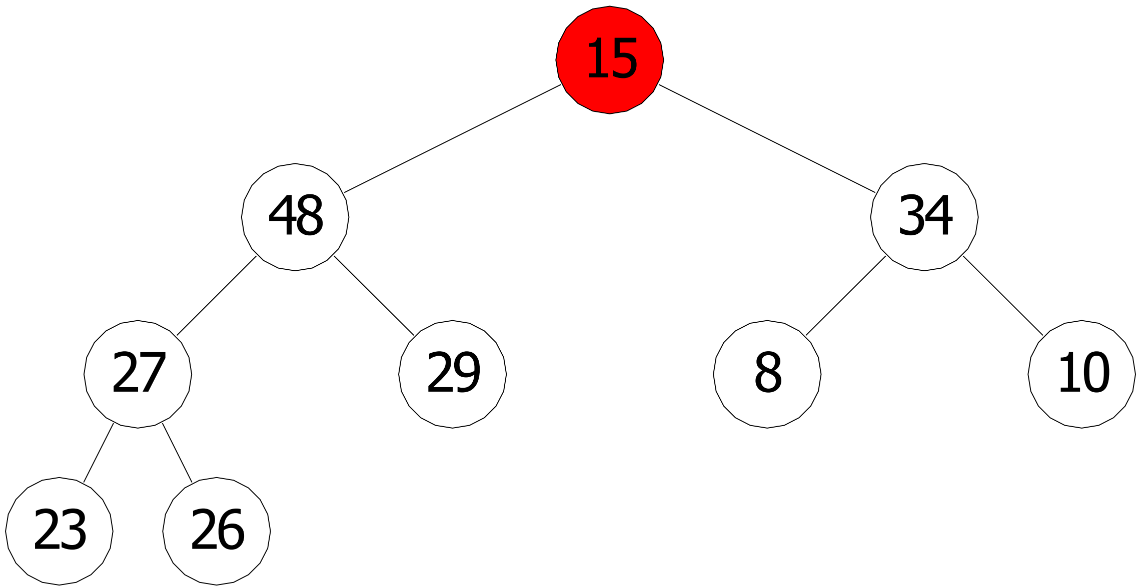
deleteMax in Heaps

- The root has the maximum item
- Replace root by the last leaf



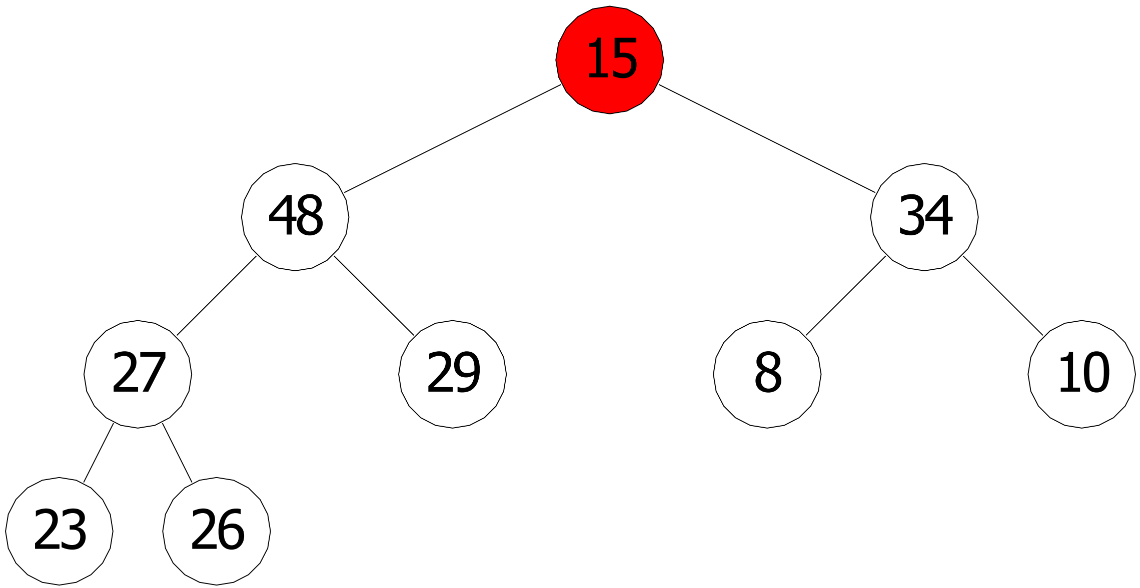
deleteMax in Heaps

- The root has the maximum item
- Replace root by the last leaf

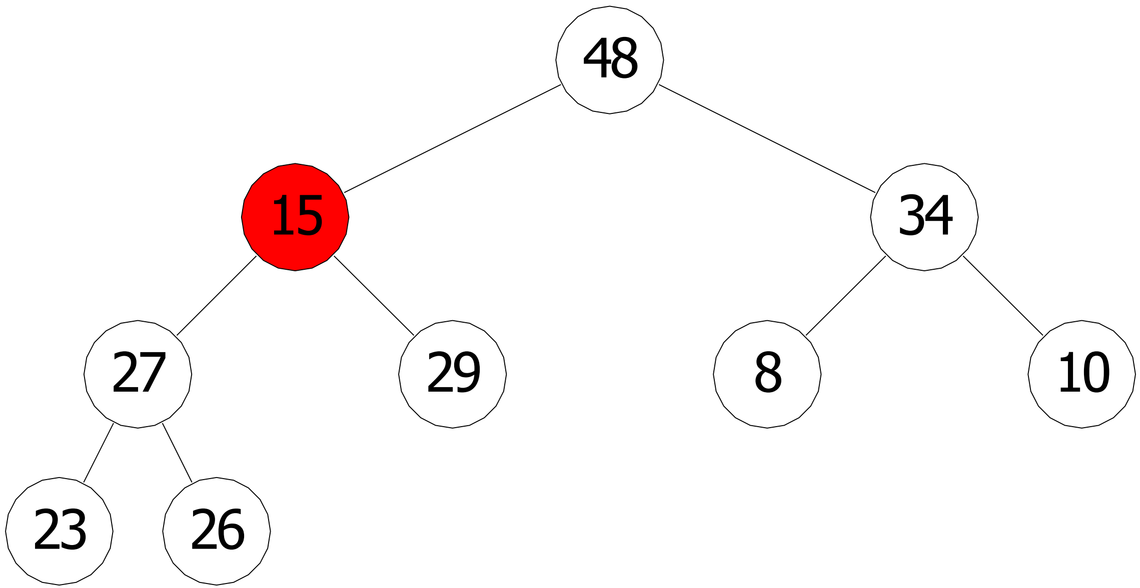


- The heap-order property might be violated
    - perform *fix-down*

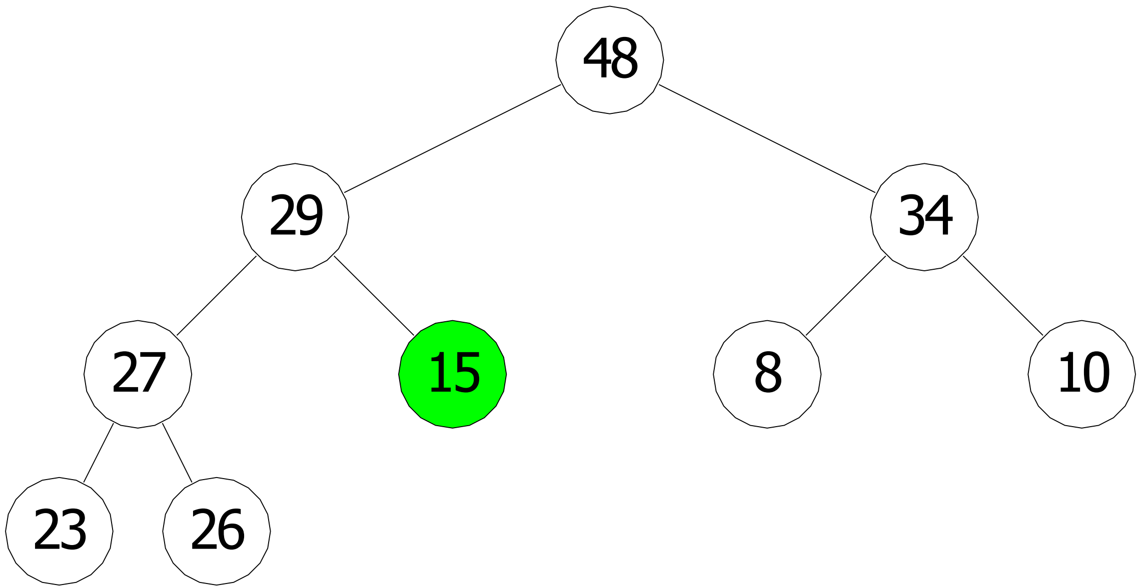
fix-down example



fix-down example

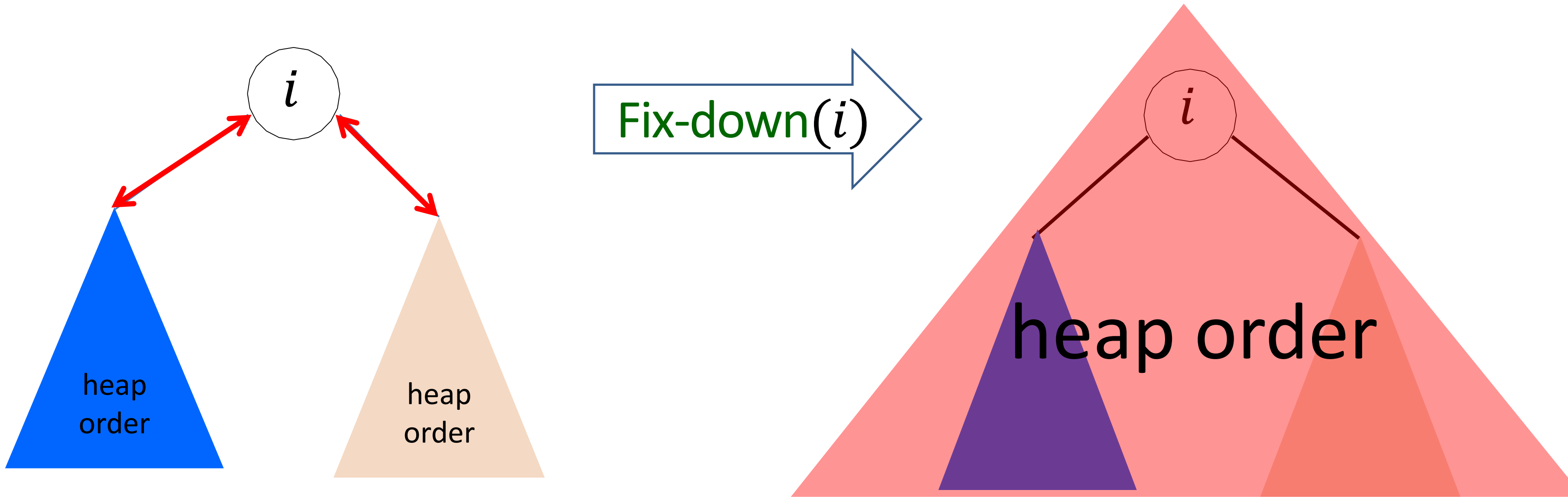


fix-down example



fix-down guarantees

- Let i be any node s.t. its left and right subtrees satisfy heap-order
- fix-down(i) restores the order in the subtree rooted at i
    - proof by induction on height



Fix-Down

fix-down(A, i)

i: index corresponding to a heap node, A: heap array

while i is not a leaf **do**

$j \leftarrow$ left child of i

if $j \neq last()$ **and** $A[j+1].key > A[\,j\,].key$ **then**

$j \leftarrow j+1$ // right child is larger

// at this point, j indexes the child with the larger key

if $A[\,i\,]$.key $\geq A[\,j\,].key$ // order is fixed, done

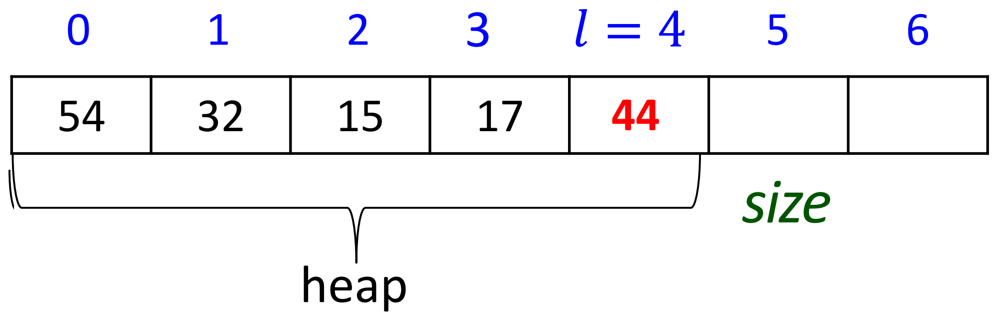
break

swap $A[\,i\,]$ **and** $A[\,j\,]$

$i \leftarrow j$ // move to one level down

- Time: O(heap height) $= O(\log n)$

Priority Queue Realization Using Heaps



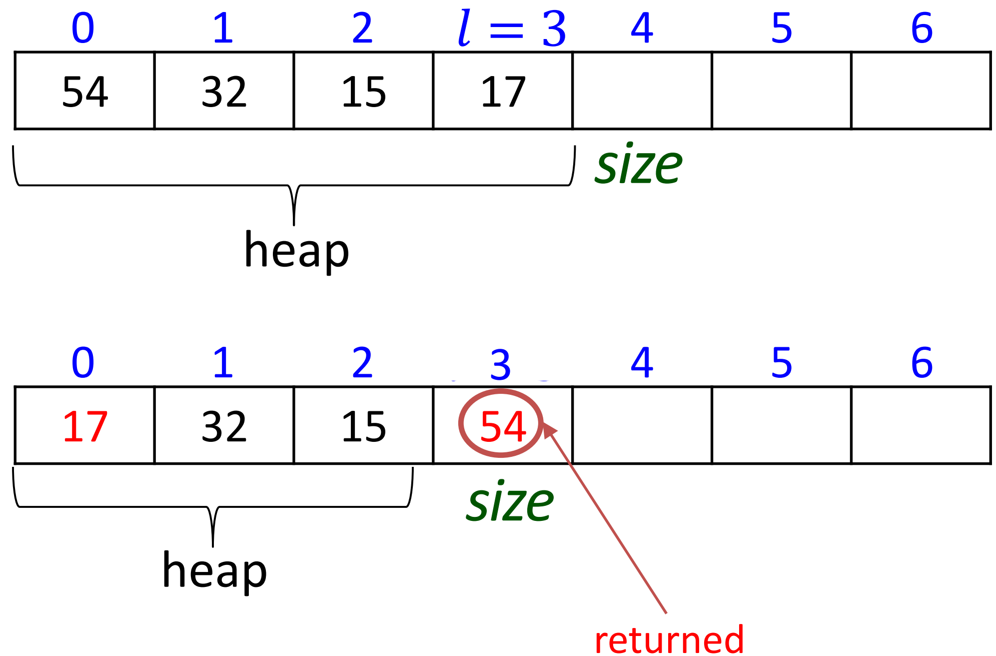
- Store items in priority queue in array *A* and keep track of *size*

insert(x)
  increase $size$
  $l \leftarrow last()$
  $A[l] \leftarrow x$
  fix-up (A, l)

- *insert* is $O(\log n)$

Priority Queue Realization Using Heaps

deleteMax ( )
  $l \leftarrow last()$
  swap $A[root()]$ and $A[l]$
  decrease $size$
  fix-down$\big(A, root(\,)\big)$
  return $(A[l])$



- *deleteMax* is $O(\log n)$

Outline

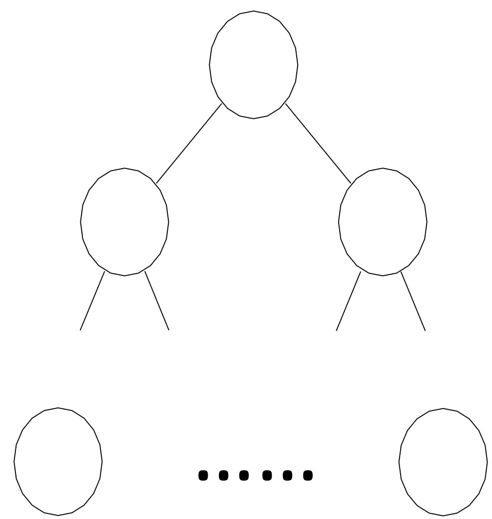
- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Sorting using Heaps

- Can sort with priority queue in $O(init + n \cdot insert + n \cdot deleteMax)$

PQSortWithHeaps(A)
  $H \leftarrow$ empty heap
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $H.\text{insert}(A[i])$
  for $k \leftarrow n-1$ **downto** 0 **do**
    $A[i] \leftarrow H.\text{deleteMax}(\,)$

- simple heap building
- uses additional array of size n for storing heap H
- insert uses *fix-up*
- worst-case time is $\Theta(n \log n)$



2^0

2^1

$2^h \approx \dfrac{n}{2}$

$$n = \underbrace{2^0 + 2^1 + \cdots + 2^{h-1}}_{\text{all levels except last}} + 2^h$$

$$n = 2^h - 1 + 2^h$$

$$\frac{n+1}{2} = 2^h$$

- In the worst case, for $n/2$ nodes do $\log n$ work, total work $\frac{n}{2} \log n$

Sorting using Heaps

- Can sort with priority queue in $O(init + n \cdot insert + n \cdot deleteMax)$

PQ-SortWithHeaps(A)

$H \leftarrow$ empty heap
for $k \leftarrow 0$ **to** $n-1$ **do**
  $H.$insert$(A[k])$
for $k \leftarrow n-1$ **downto** 0 **do**
  $A[k] \leftarrow H.\text{deleteMax}(\,)$

- simple heap building
- uses additional array of size n for storing heap H
- insert uses *fix-up*
- worst-case time is $\Theta(n \log n)$

- *PQ-Sort* with heap is $O(n \log n)$ and not in place
  - need $O(n)$ additional space for heap array H
- Heapsort: improvement to *PQ-Sort* with two added tricks
  1. use the input array A to store the heap!
  2. heap can be built in linear time if know all items in advance
  - heapsort is in-place, needs $O(1)$ additional (or *auxiliary*) space

Building Heap Directly In Input Array

A						
17	32	15	54	2	25	3

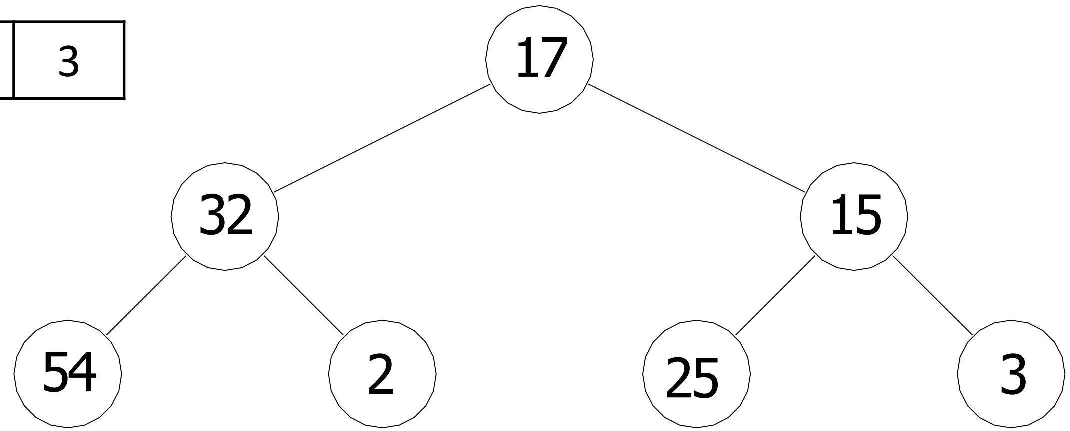
A						
54	25	32	17	2	15	3

Problem statement: build a heap from n items in $A[0, \ldots, n-1]$ without using additional space

- i.e. put items in $A[0, \ldots, n-1]$ in heap-order

Building Heap Directly In Input Array

A	17	32	15	54	2	25	3

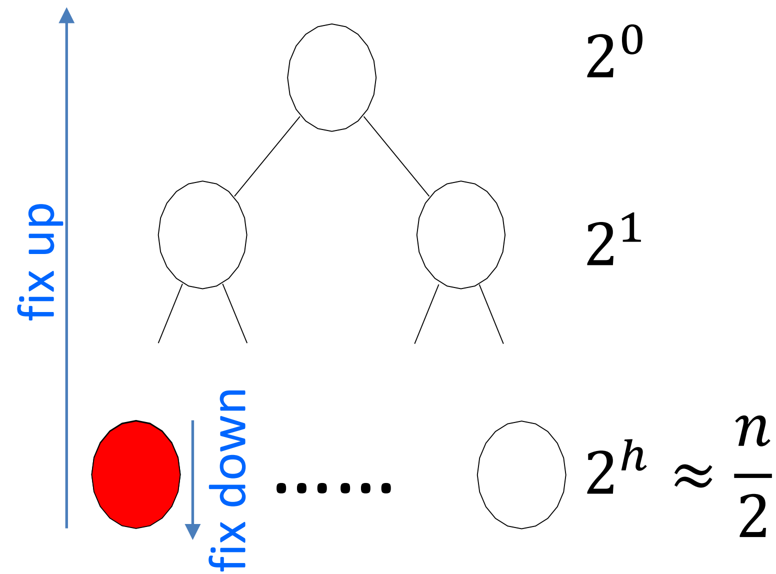


Problem statement: build a heap from n items in $A[0, \ldots, n-1]$ without using additional space

- i.e. put items in $A[0, \ldots, n-1]$ in heap-order
- Treat array as a binary tree
- Heap order does not hold
  - can use either *fix-down* or *fix-up* for each node
  - both work, but *fix-down* is more efficient

Building Heap Directly In Input Array: Fix-Up vs. Fix-Down

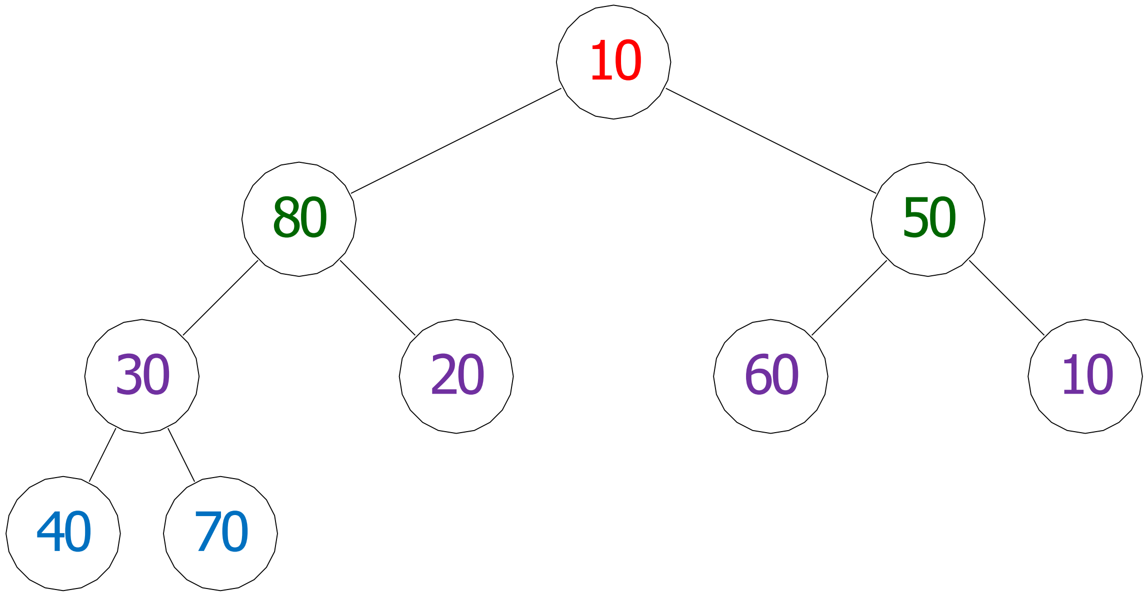
- Worst case scenario
  - deepest nodes are most numerous, there are $\frac{n}{2}$ of them
- For each deep node
  - fix-up takes $O(\log n)$ time
  - fix-down takes $O(1)$ time



- Fix-up called for all $\frac{n}{2}$ deepest nodes takes $O(n \log n)$ time
- Fix-down for all $\frac{n}{2}$ deepest nodes takes $O(n)$ time

Heapify Example

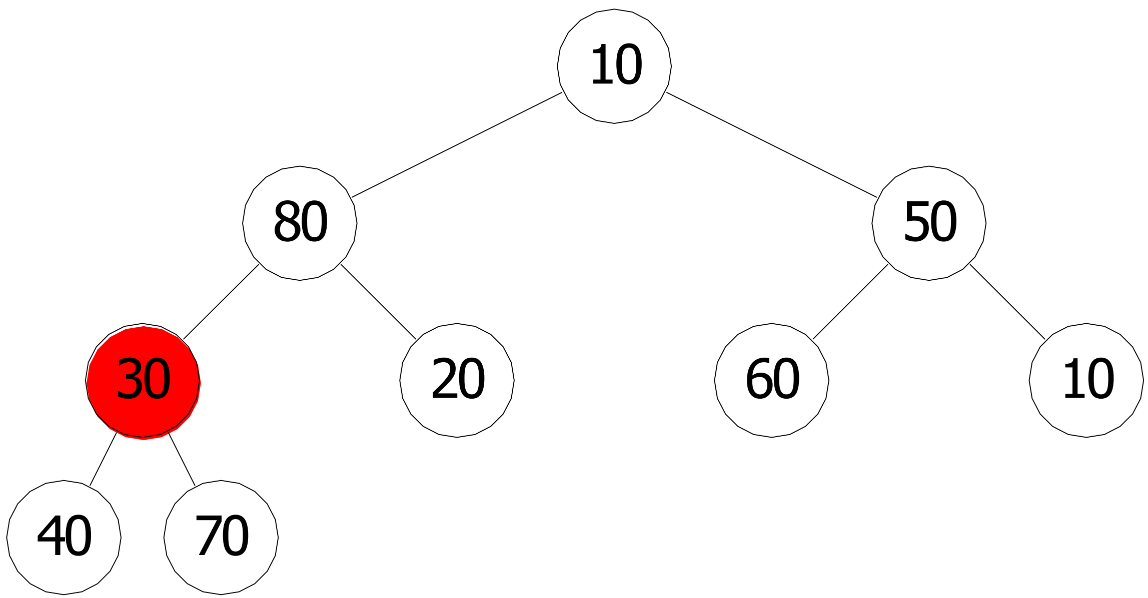
- Arbitrary array $A = \{10, 80, 50, 30, 20, 60, 10, 40, 70\}$
- View it as binary tree using our normal indexing for heaps



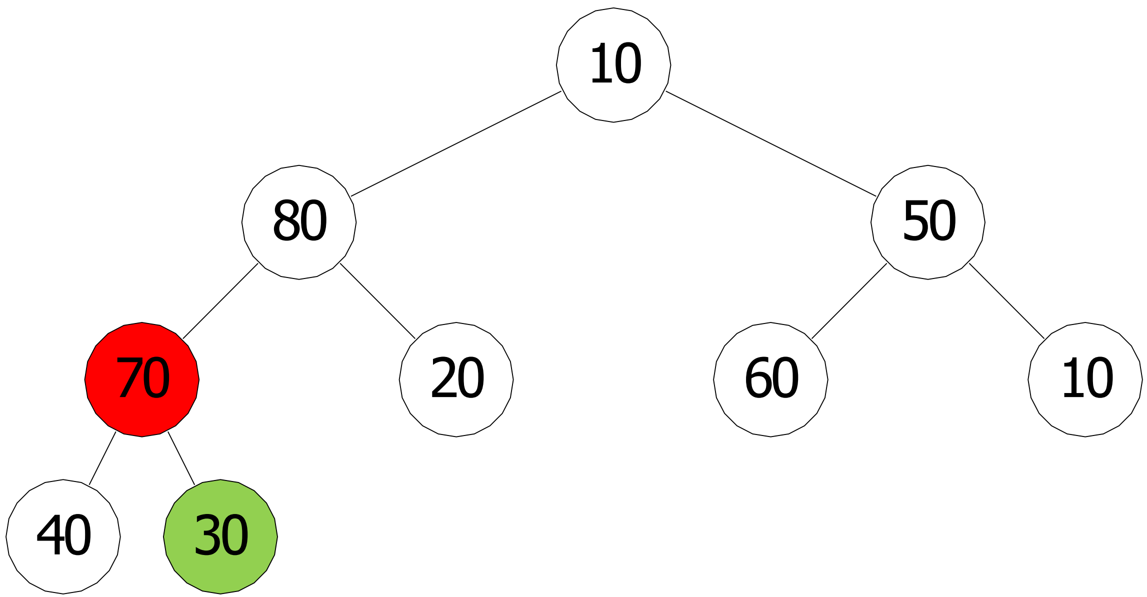
- In general, do not get a heap
- Put it in heap order by repeatedly calling *fix-down*
  - resulting algorithm is called *heapify*

Heapify Example

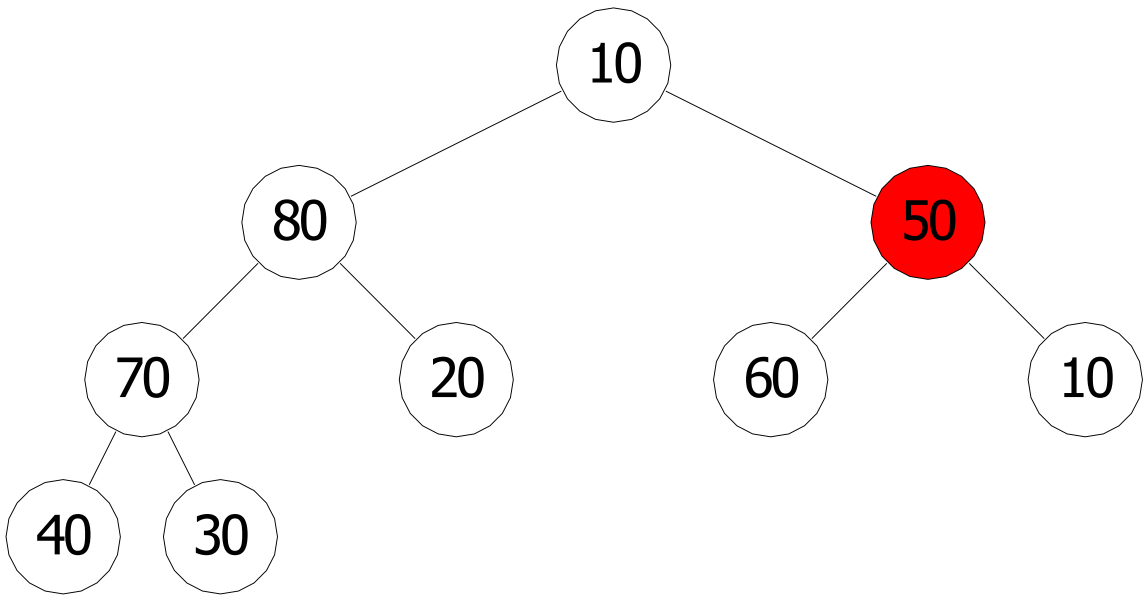
- No need to call *fix-down* on the leaves
  - No harm, but *fix-down* will do nothing for the leaves
- Start calling *fix-down* with the parent of last node
  - this is the deepest and leftmost non-leaf node



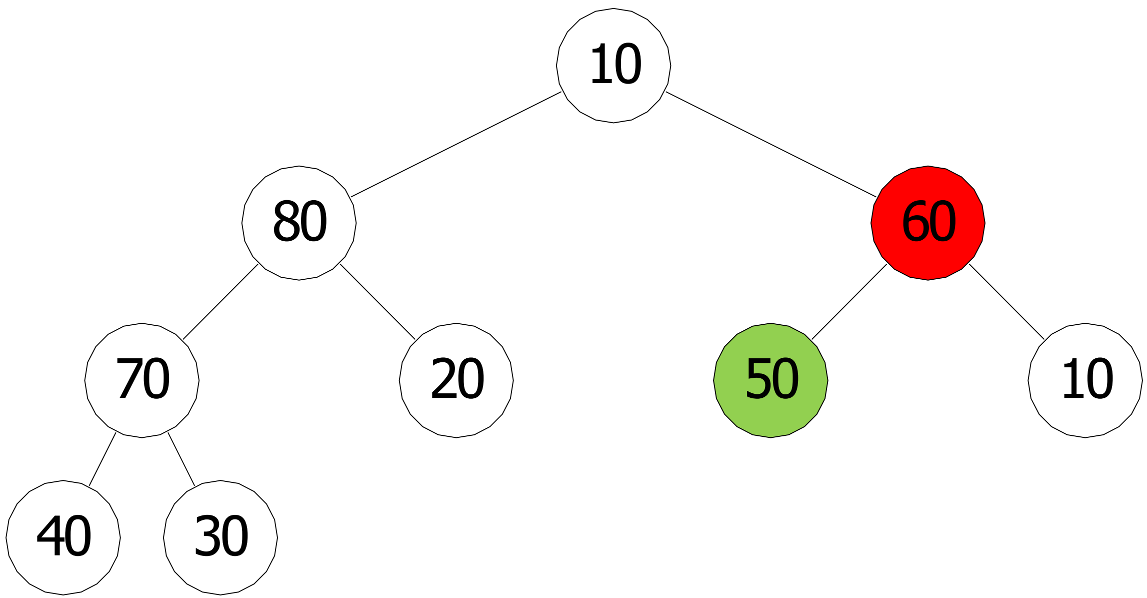
Heapify Example



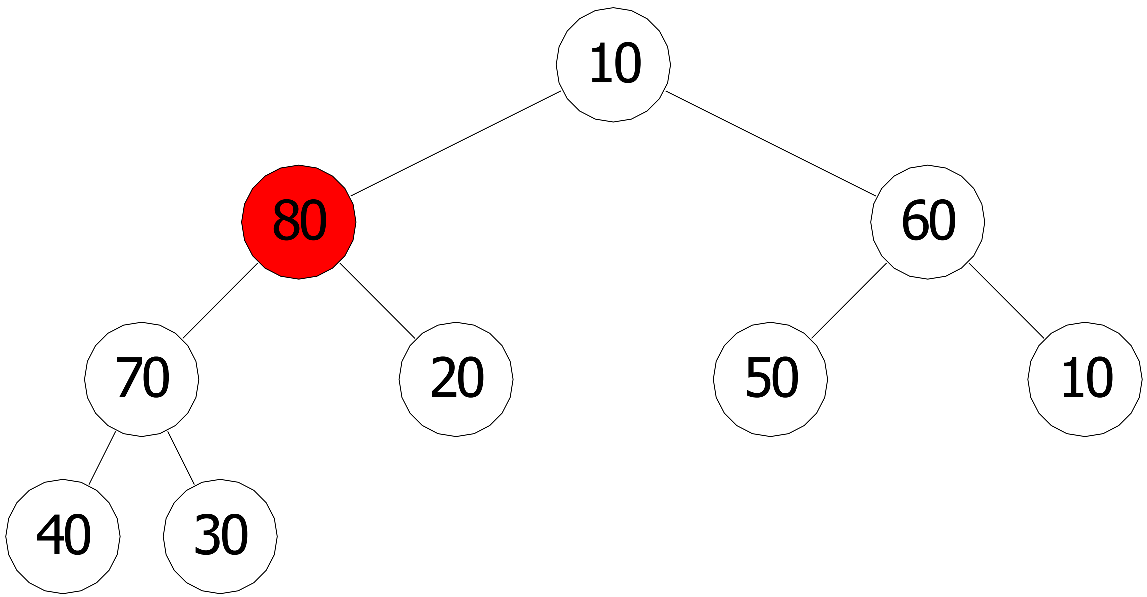
Heapify Example



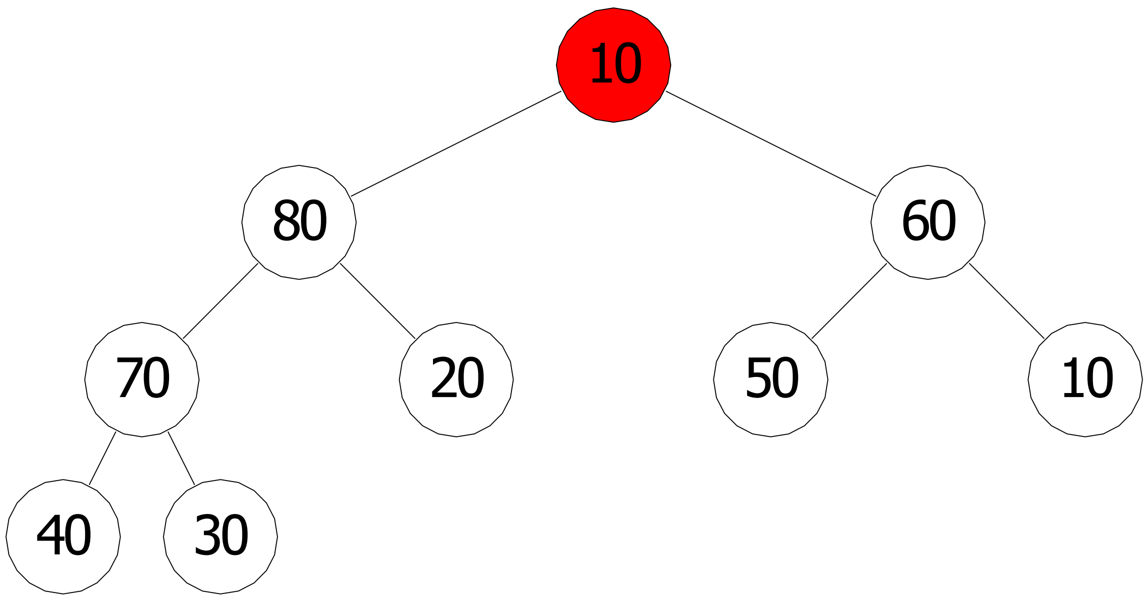
Heapify Example



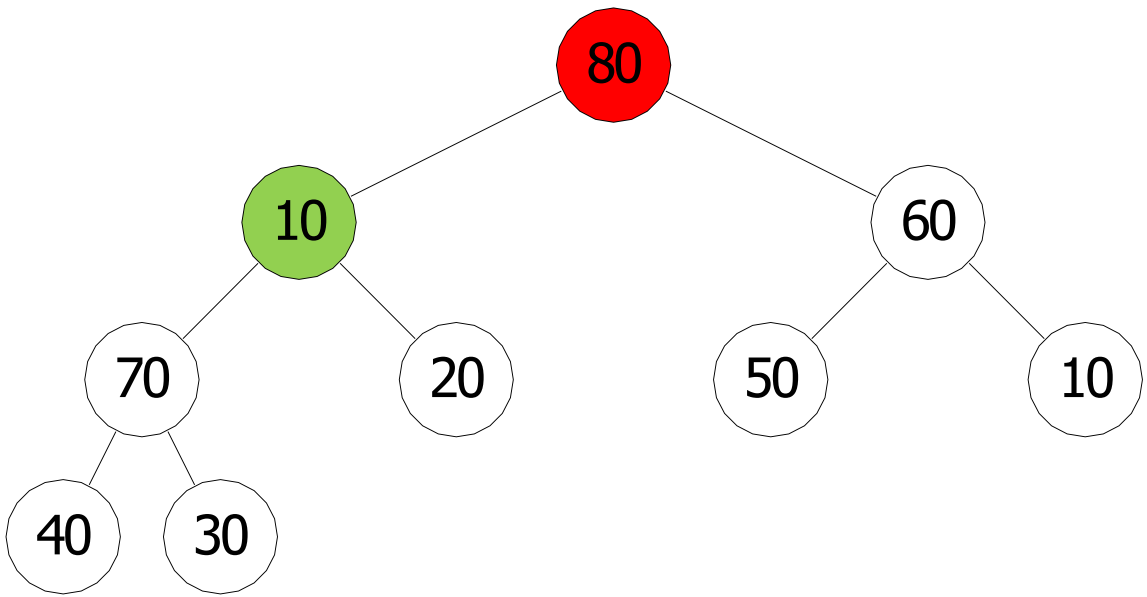
Heapify Example



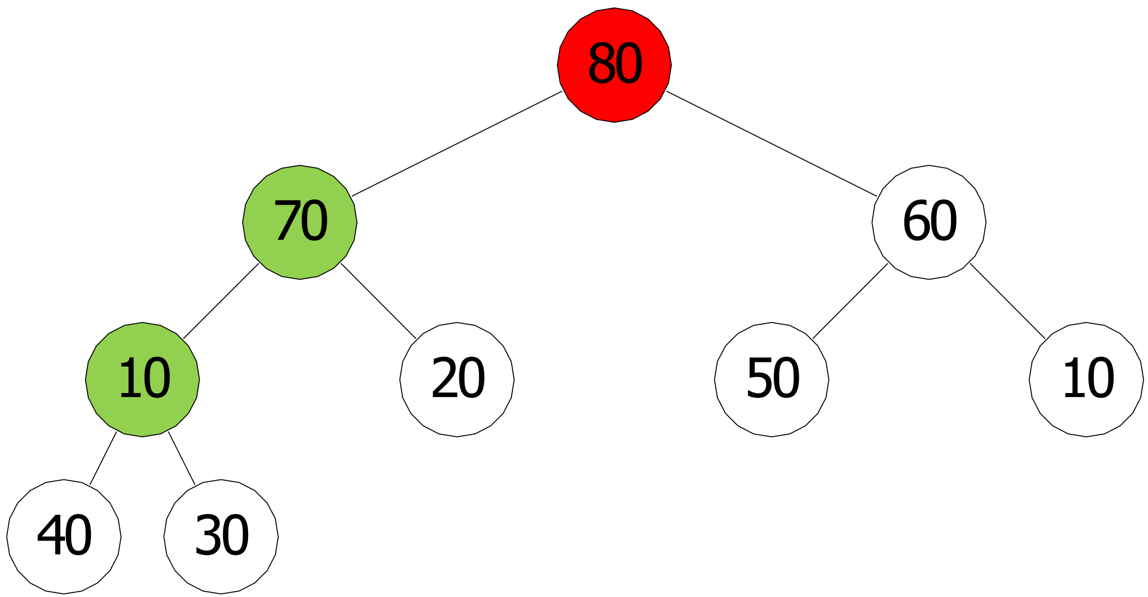
Heapify Example



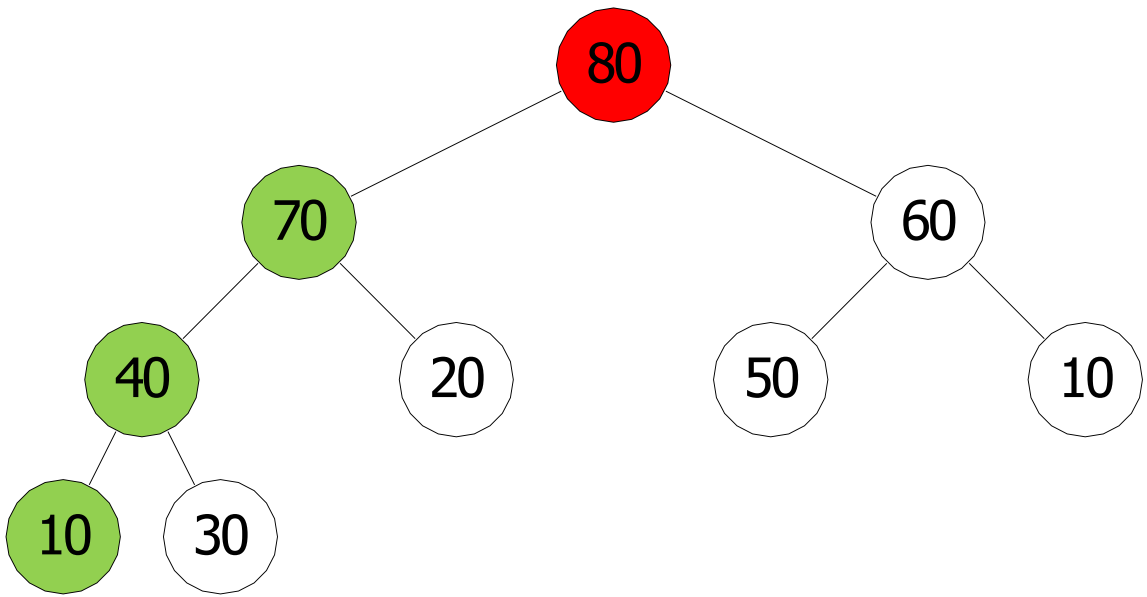
Heapify Example



Heapify Example



Heapify Example



Heapify Pseudocode

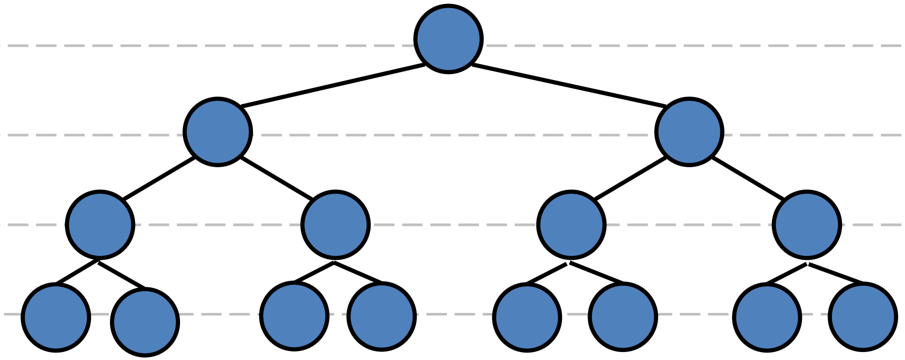
heapify (A)
A : an array
    for $i \leftarrow$ *parent* $($*last*$())$ **downto** 0 **do**
        fix-down (A, i)

- Straightforward analysis yields complexity $O(n \log n)$
- Careful analysis yields complexity $\Theta(n)$
- A heap can be built in linear time if we know all items in advance

Heapify Analysis

$h \leq \log n$

depth	nodes	work per node
0	2^0	h
1	2^1	$h-1$
i	2^i	$h-i$
$h-1$	2^{h-1}	1



$$\sum_{i=0}^{h-1} 2^i(h-i) = 2^h \sum_{i=0}^{h-1} \frac{2^i(h-i)}{2^h} = 2^h \sum_{i=0}^{h-1} \frac{(h-i)}{2^{h-i}}$$

$$= 2^h \left(\frac{h}{2^h} + \frac{h-1}{2^{h-1}} + \cdots + \frac{1}{2^1} \right)$$

$$= 2^h \underbrace{\sum_{i=1}^{h} \frac{i}{2^i}}_{\text{convergent series}} \leq 2^h c \leq 2^{\log n} c = cn$$

$$\lim_{i \to \infty} \frac{2^i(i+1)}{i 2^{i+1}} = \frac{1}{2}$$

HeapSort

0	1	2	3	4	5	6
30	54	15	17	5	32	6

heapify

n=7

0	1	2	3	4	5	6
54	30	32	17	5	15	6

deleteMax

n=6

0	1	2	3	4	5	6
32	30	15	17	5	6	54

deleteMax

n=5

0	1	2	3	4	5	6
30	17	15	6	5	32	54

deleteMax

n=4

0	1	2	3	4	5	6
17	6	15	5	30	32	54

deleteMax

n=3

0	1	2	3	4	5	6
15	6	5	17	30	32	54

HeapSort

HeapSort(A)

for $i \leftarrow$ *parent* (*last*()) **downto** 0 **do**

fix-down (A, i)

while $n > 1$

swap items $A[root()]$ and $A[last()]$

decrease n

fix-down$(A, root())$

heapify

$\Theta(n)$

deleteMax, n times

$\Theta(n \log n)$

- Similar to *PQ-Sort* with heaps, but uses input array A for storing heap
- In-place, i.e. only $O(1)$ extra space

Heap Summary

- Binary heap: binary tree that satisfies structural property and heap order property
- Heaps are one possible realization of ADT PriorityQueue
  - *insert* takes $O(\log n)$ time
  - *deleteMax* takes $O(\log n)$ time
  - also supports *findMax* in $O(1)$ time
- A binary heap can be built in linear time, if all elements are known beforehand
- With binary heaps leads to a sorting algorithm with $O(n \log n)$ worst case time
- We have seen max-oriented version of heaps
- There exists a symmetric min-oriented version supporting *insert* and *deleteMin* with same run times

Outline

- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Selection

	0	1	2	3	4	5	6
	3	6	10	0	5	4	9
sorted	0	3	4	5	6	9	10

- **Select(k) problem** find *kth item* in array A of n numbers
  - item that would be in $A[k]$ if A was sorted in nondecreasing order
    - this is $(k+1)$ smallest item in the array
  - example: select(3) = 5
  - nondecreasing = increasing if keys do not repeat
- **Solution 1**
  - make $k+1$ passes through A, deleting minimum number each time
  - $\Theta(kn)$
  - $k = n/2$, time complexity is $\Theta(n^2)$
    - efficient solution is harder to obtain if k is a median
- **Solution 2**
  - sort array A and return number at index k
  - $\Theta(n \log n)$
  - complexity does not depend on k

Selection

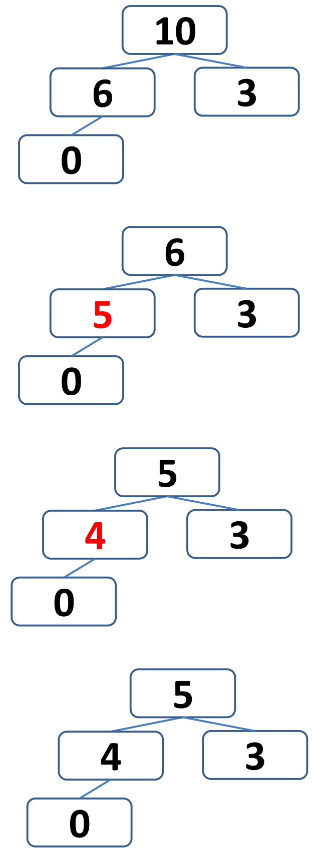
- **Solution 3**
  - scan A and maintain $k+1$ smallest numbers seen so far in max-heap
  - example: select(3)

3	6	10	0	5	4	9

3	6	10	0	5	4	9

3	6	10	0	5	4	9

3	6	10	0	5	4	9



  - at the end, kth item is on the heap top (5 in our example)
  - $\Theta(n \log k)$ time complexity
  - for $k = n/2$, this solution is $\Theta(n \log n)$

Selection

0	1	2	3	4	5	6	7	8	9
3	6	10	0	5	4	9	2	1	7

- **Solution 4**
  - make A into a min-heap by calling *heapify*(A)
  - call *deleteMin*(A) $k+1$ times
  - $\Theta(n + k \log n)$
    - better than $\Theta(n \log k)$ time complexity of **solution 3**
  - if $k = n/2$, this solution is $\Theta(n \log n)$