CS 240 – Data Structures and Data Management

Module 2: Priority Queues

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2024

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

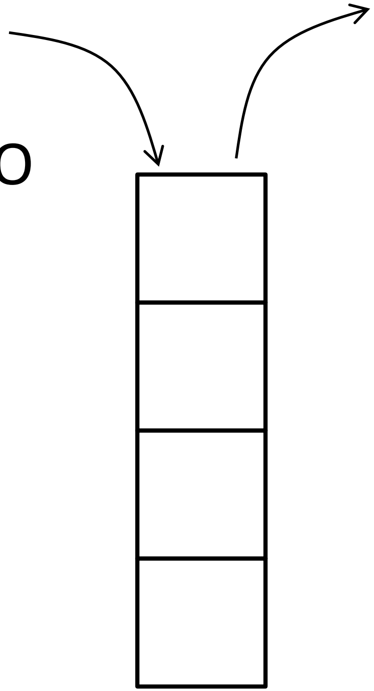
Outline

- 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)

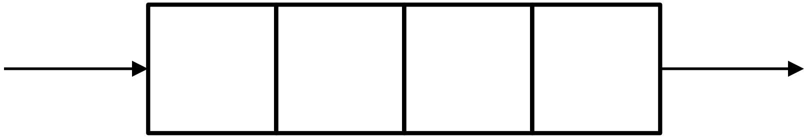
- A description of *information* and a collection of *operations* on that information
- The information accessed *only* through the operations
- ADT describes what is stored and what can be done with it, but not how it is implemented
- Can have various *realizations* of an ADT, which specify
  - how 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* insert an item
  - *pop* remove and return 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
    - both have constant time *push*/*pop*

Queue ADT



- ADT consisting of a collection of items removed in FIFO (first-in first-out) order
- Operations
  - *enqueue* insert an item
  - *dequeue* remove and return the least recently inserted
- Items enter queue at the rear and are removed from front
- Extra operations
  - *size*, *isEmpty*, and *peek*
- Realizations of Queue ADT
  - (circular) arrays
  - linked lists
    - both have constant time *enqueue* /*dequeue*

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*, *other info*) or (*priority*, *value*)
    - 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
    - sorting, etc.
- Question: How to simulate a stack/queue with a priority queue?

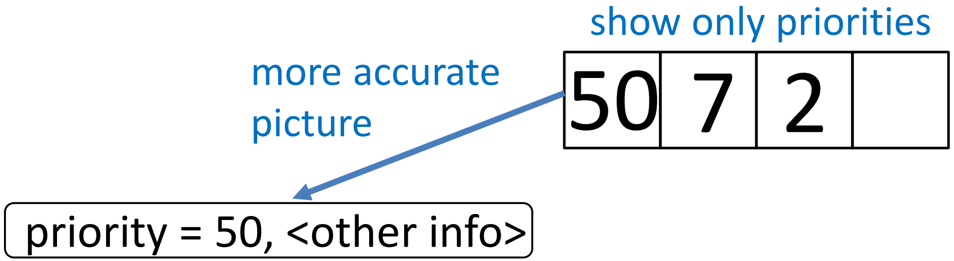
Using Priority Queue to Sort

```
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 $O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{deleteMax})$

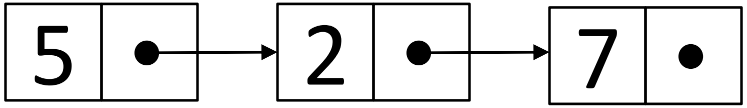
Realizations of Priority Queues

- **Attempt 1**: *unsorted arrays*



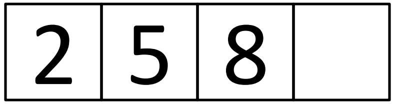
  - assume dynamic arrays
    - expand by doubling when needed
    - happens rarely, so amortized time over all insertions is $O(1)$
  - *insert*: $\Theta(1)$
  - *deleteMax*: $\Theta(n)$
  - PQ sort becomes $\Theta(n^2)$ in the worst and in the best cases
      - equivalent to *selection-sort*

- **Attempt 2**: *unsorted linked lists*
  - efficiency identical to Attempt 1



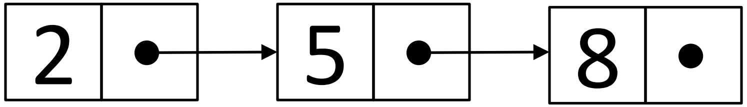
Realizations of Priority Queues

- **Attempt 3**: *sorted arrays*



  - store items in order of increasing priority
  - *deleteMax*: $\Theta(1)$
  - *insert*: $\Theta(n)$
    - in $O(1)$ in the best case (how?)
  - PQ-sort equivalent to insertion-sort
    - $\Theta(n^2)$ worst case
    - $\Theta(n)$ best case

- **Attempt 4**: *sorted linked-lists*

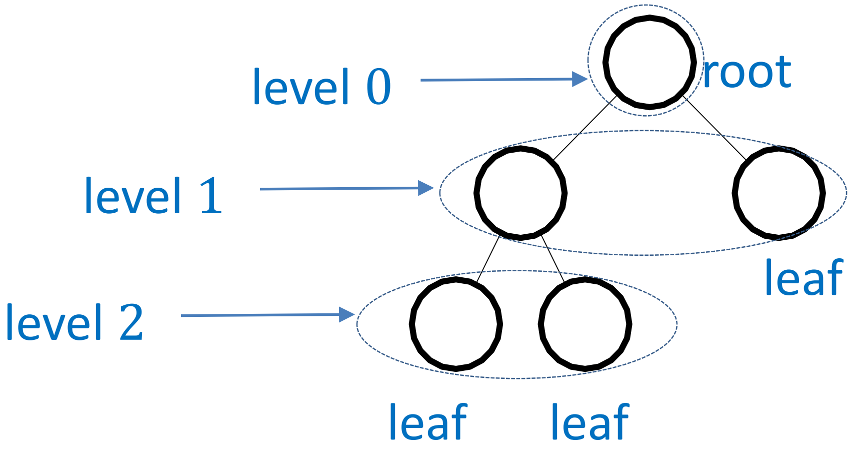


  - similar to Attempt 3

Outline

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

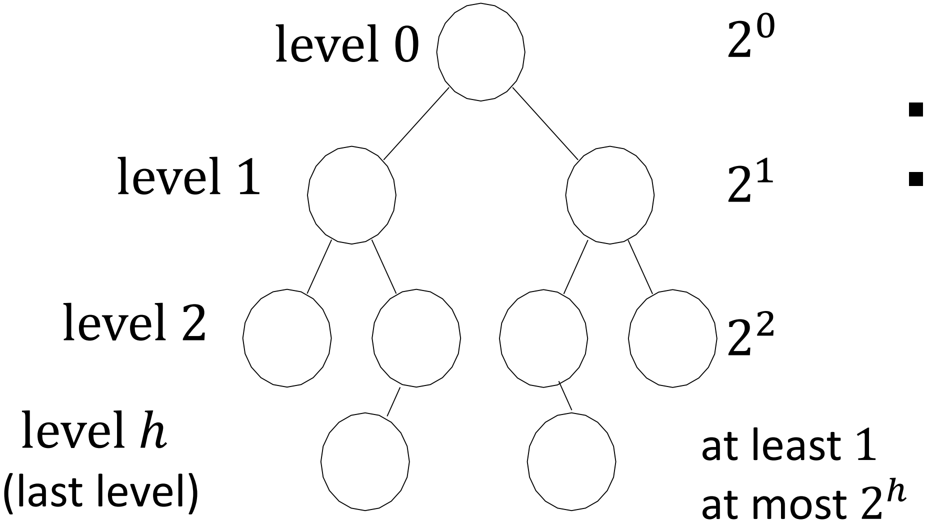
Binary Tree Review



- A *binary tree* is either
    - empty, or
    - consists of three parts
        - node
        - two binary trees
            - left subtree
            - right subtree
- Terminology
    - root, leaf, parent, child, level, sibling, ancestor, descendant
    - level l: all noes with distance l from the root (root is on level 0)
    - height of the tree is the longest path 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



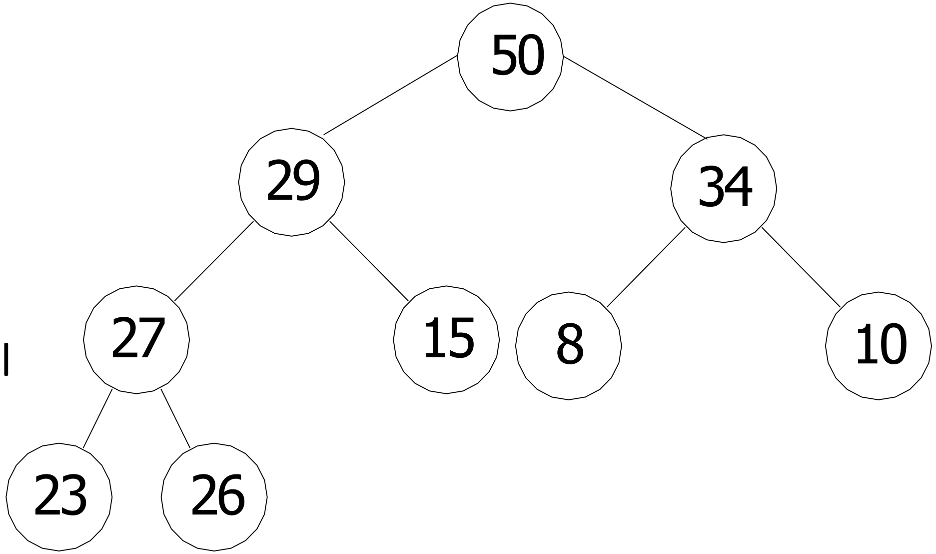
- level i has 2^i nodes
- level h has between 1 and 2^h nodes

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

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

- Therefore $n \leq 2^{h+1} - 1$
- Simplifying, $h \geq \log(n+1) - 1$
- 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

Heaps: Definition



- 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
     - last level is *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
- Heaps are ideal for implementing priority queues

Heap Height

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

- heap is a binary tree ⇒ height $h \in \Omega(\log n)$
- need to show $h \in O(\log n)$
- heap has all levels full except possibly level h
  - 2^i nodes at level $0 \le i \le h-1$
- Thus

at least last node at level h

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

$$n \ge 2^h - 1 \quad +1$$

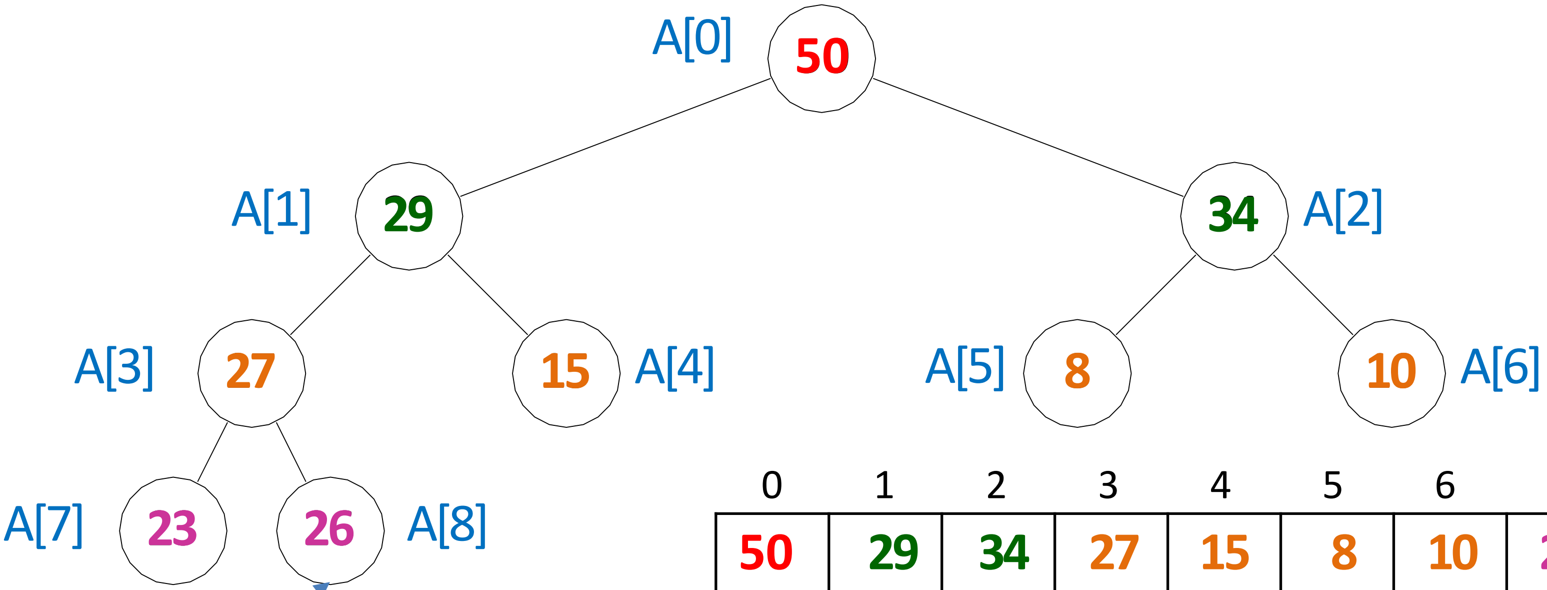
$$n \ge 2^h$$

$$h \le \log n$$

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

Storing Heaps in Arrays

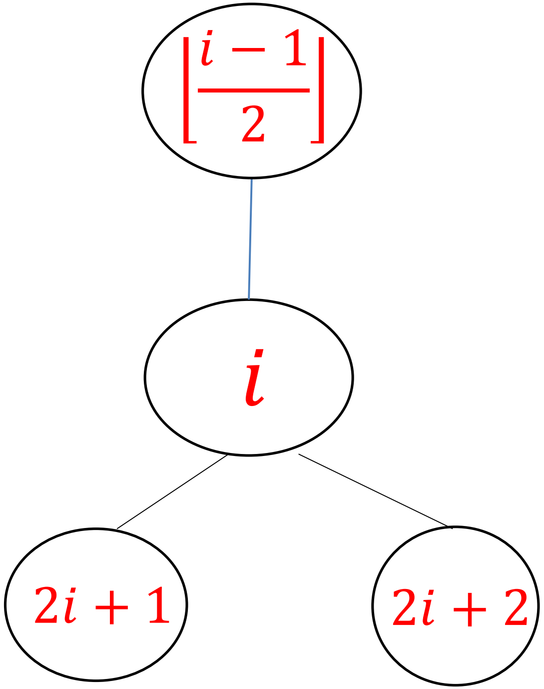
- Using linked structure for heaps wastes space
- Let H be a heap of n items and let A be an array of size n
  - store root in $A[0]$
  - continue storing *level-by-level* from top to bottom, in each level left-to-right



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

- Use dynamic arrays
  - A.size() = 9
- Last heap node is in $A[n-1]$

Heaps in Arrays: Navigation



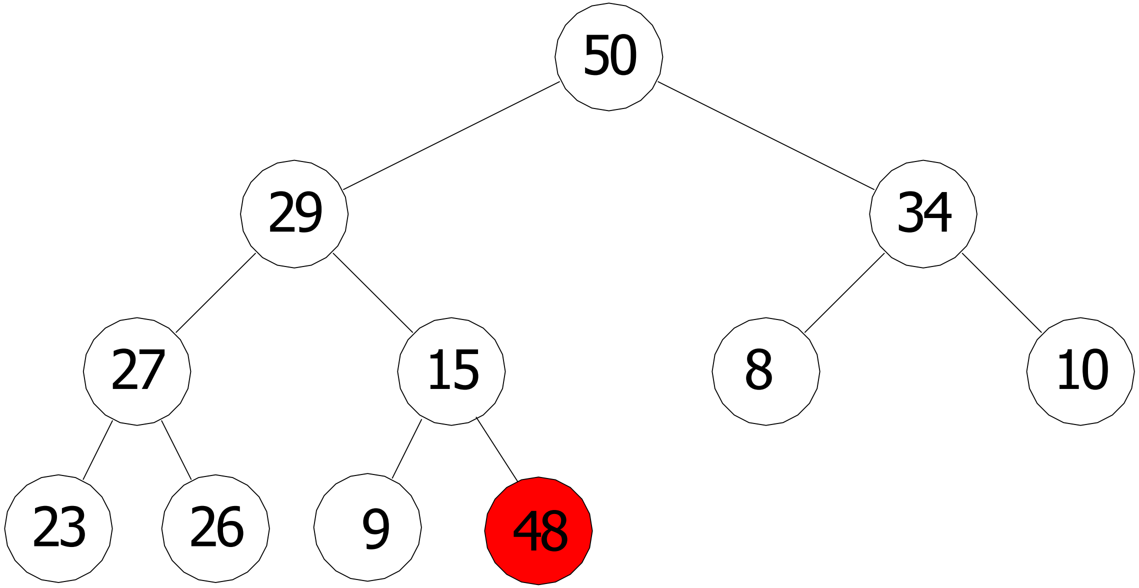
- Use node and index interchangeably
- Root is at index 0
- Last *node is* $n-1$
    - n is the size
- Left child of i, if exists, is $2i+1$
- Right child of i, if exists, is $2i+2$
- Parent of i, if exists, is $\left\lfloor \frac{i-1}{2} \right\rfloor$
- These nodes exist if index falls into range $\{0, \ldots n-1\}$
- Hide implementation details using helper-function
    - functions *root*( ), *parent*(i), *left*(i), right(i), *last*$()$
        - some helper functions need to know n
            - *left*(i), right(i), *last*$()$
        - assume data structure stores n explicitly

Outline

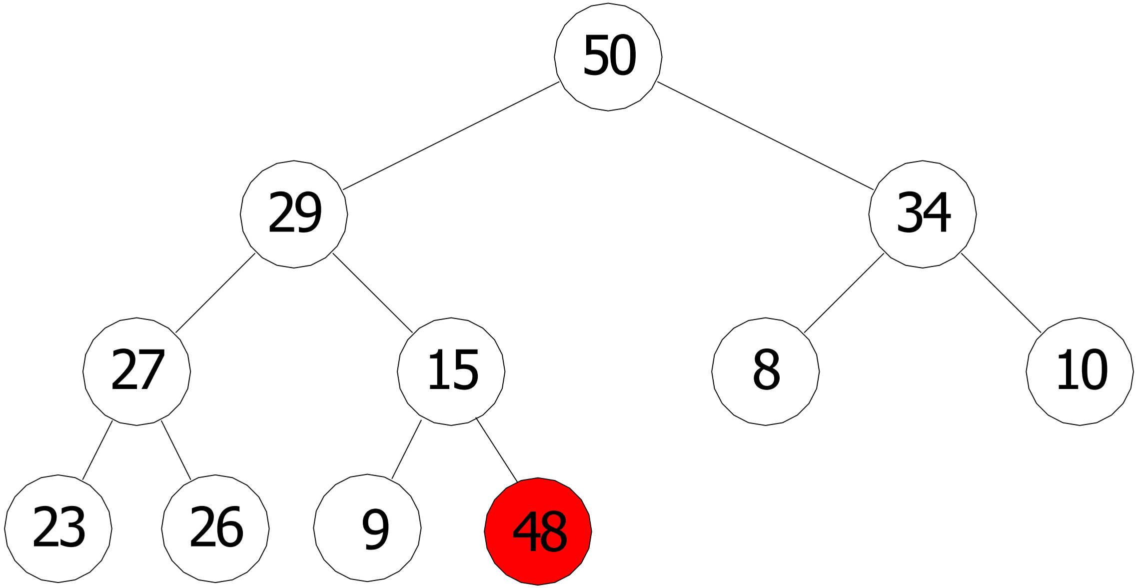
- Priority Queues
  - Abstract Data Types
  - ADT Priority Queue
  - Binary Heaps
  - Operations in Binary Heaps
  - PQ-Sort and Heapsort
  - Intro for the Selection Problem

Insertion in Heaps

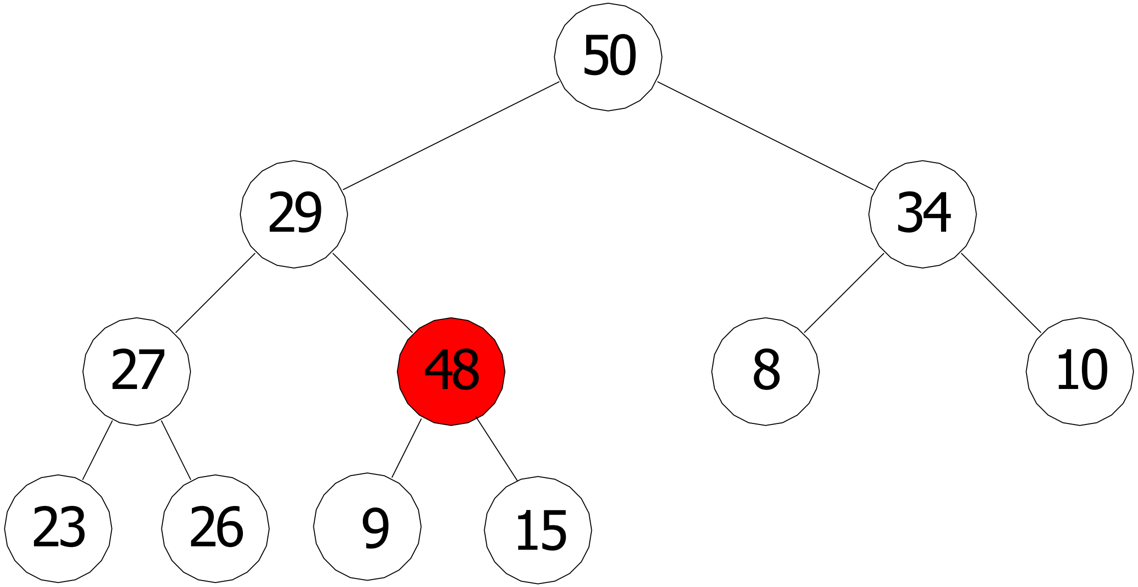
- Place new key at the first free leaf
- Heap-order property might be violated
- Perform a *fix-up*



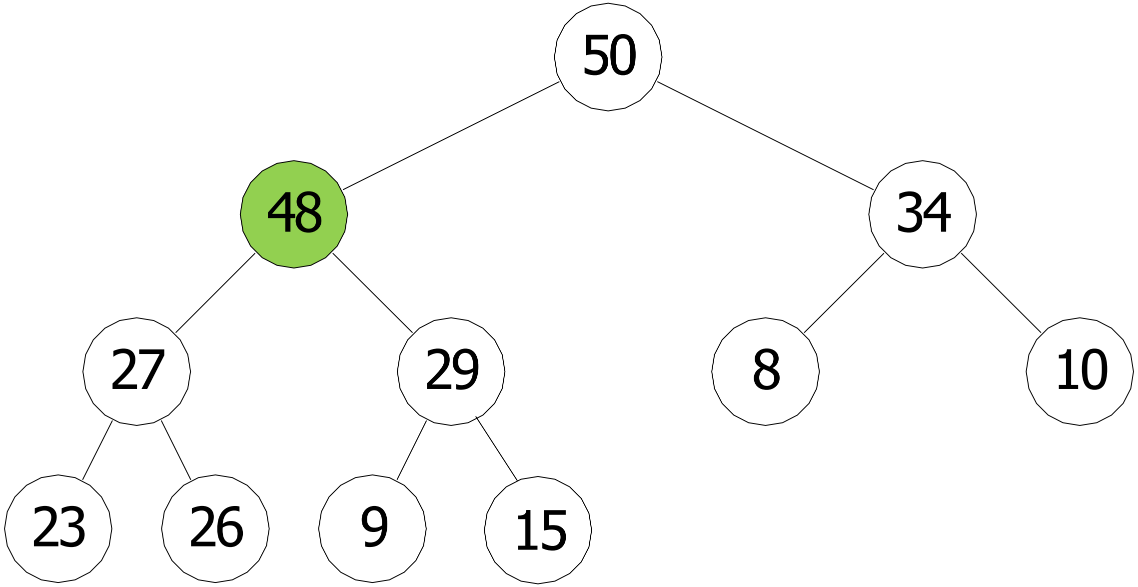
fix-up example



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
```

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

Insert Pseudocode

54	32	15	17	**44**

size = 5

fix up

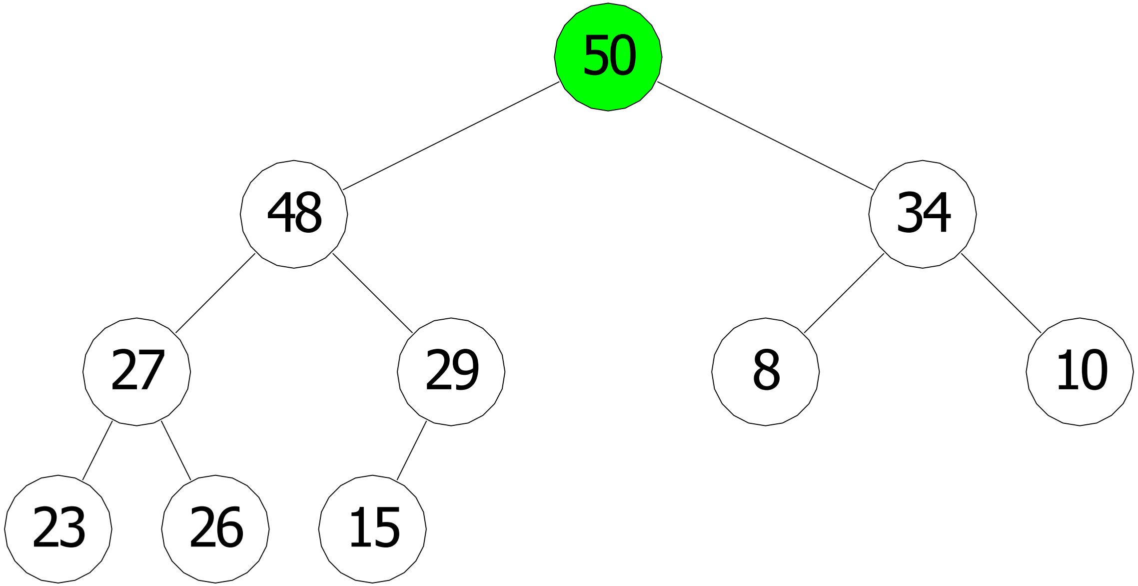
54	44	15	17	32

- Class for heap
    - variable *size* is a class variable to keep track of the number of items
- Store items in array A
- *insert* is $O(\log n)$

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

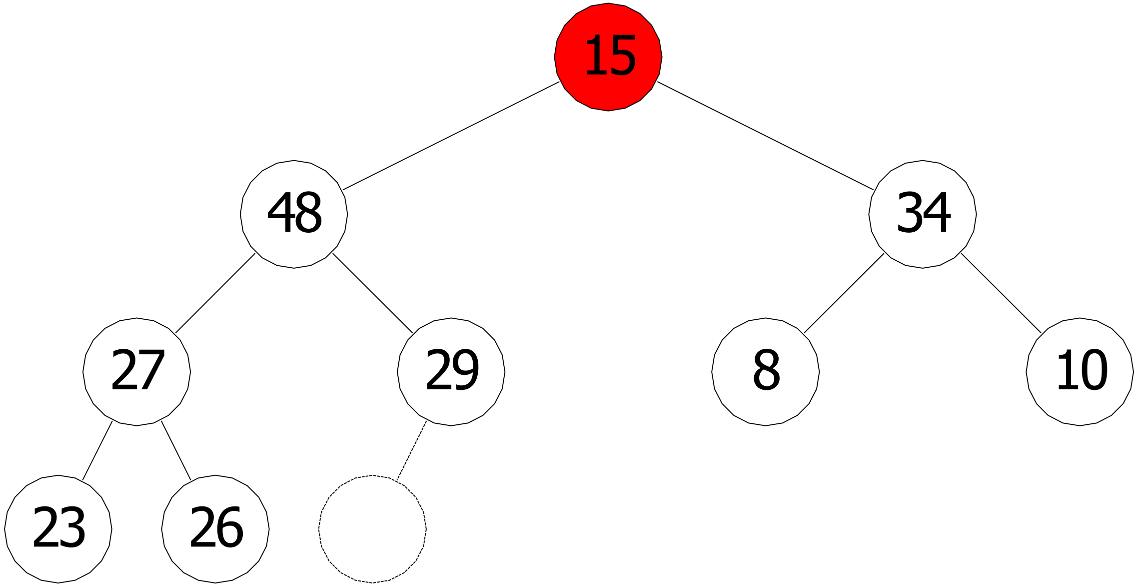
deleteMax in Heaps

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



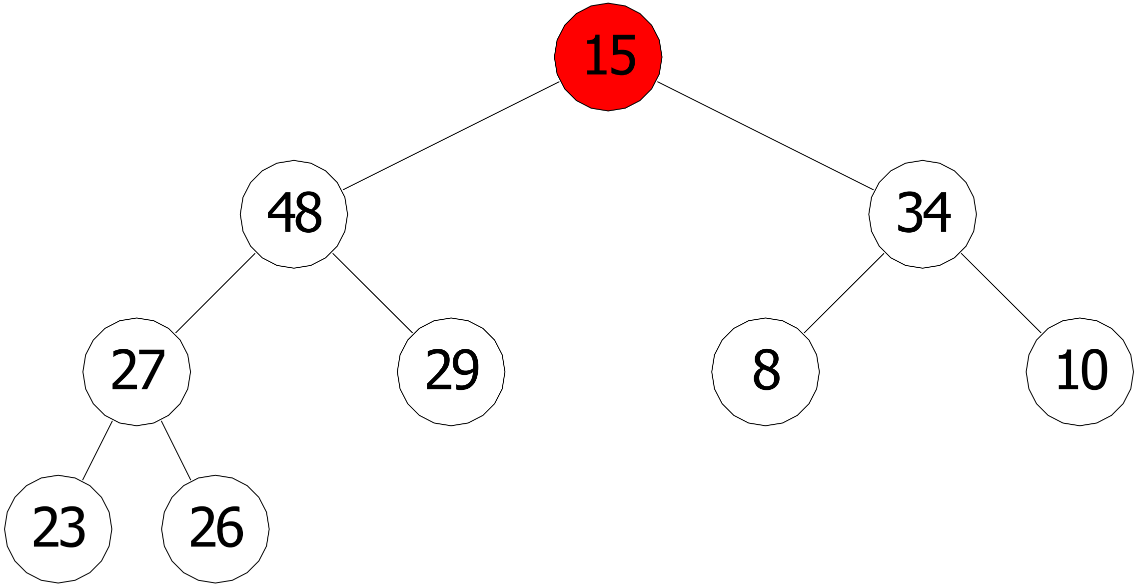
deleteMax in Heaps

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



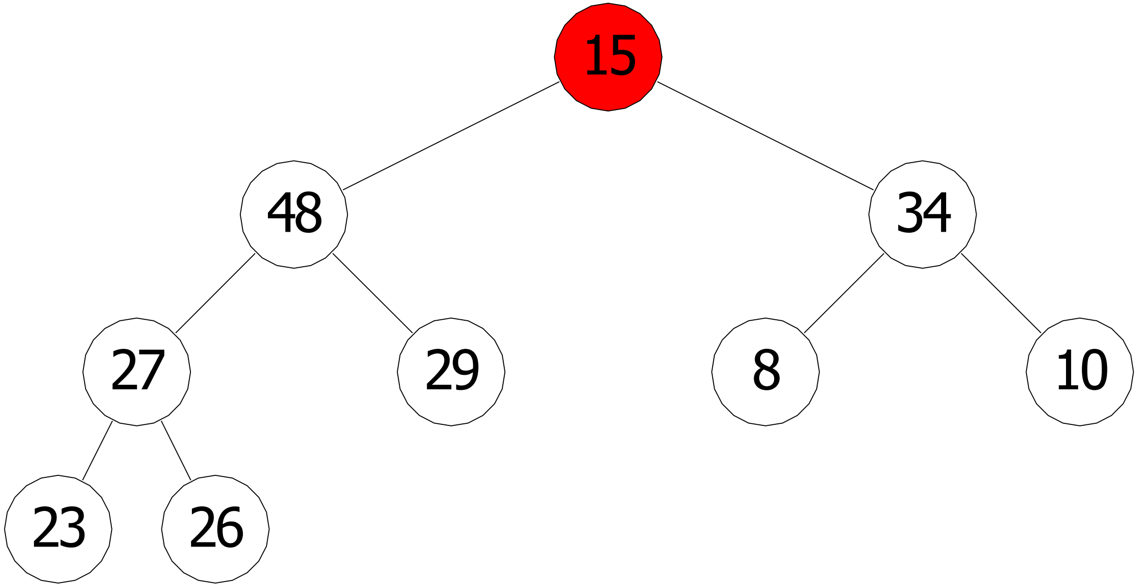
deleteMax in Heaps

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

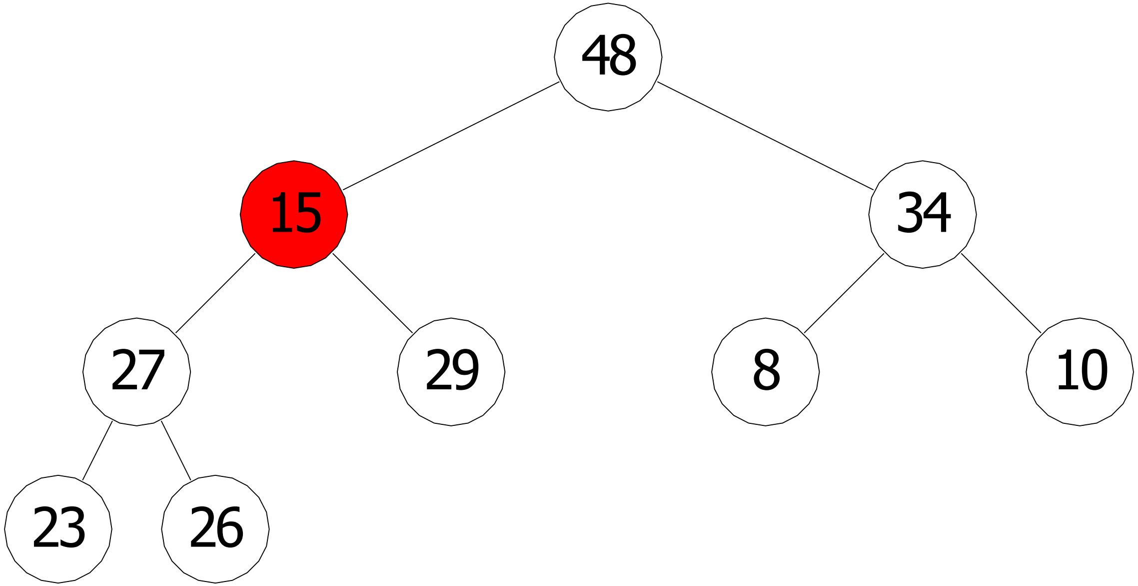


- The heap-order property might be violated
    - perform *fix-down*

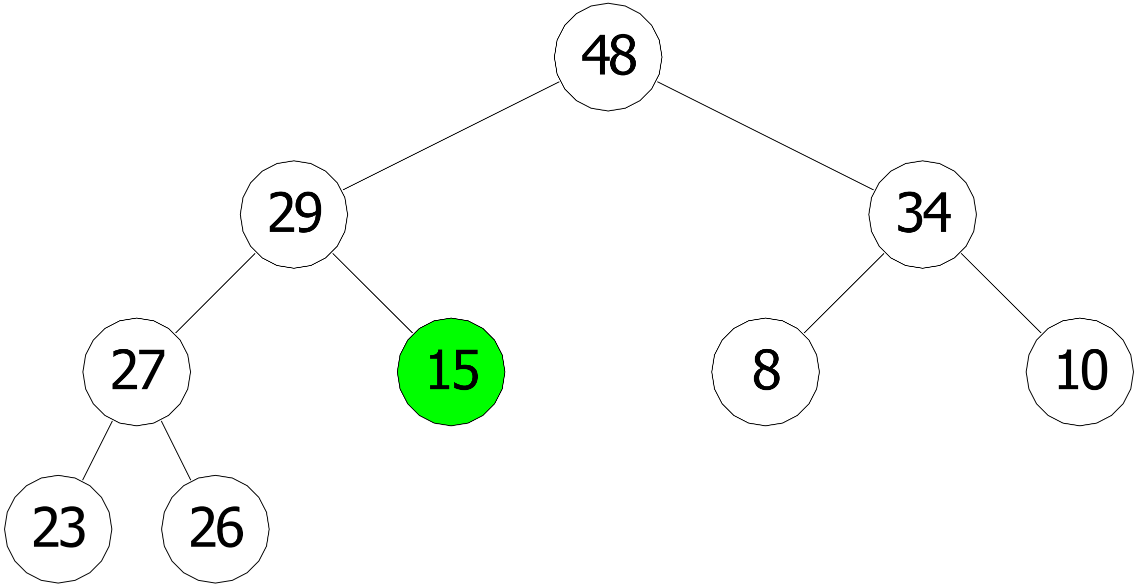
fix-down example



fix-down example



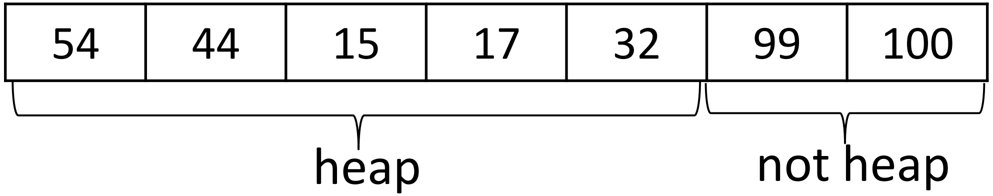
fix-down example



Fix-Down

```
fix-down$(A, i, n)$
$A$: array that stores a heap of size $n$ in locations $0 \dots n-1$
$i$: index corresponding to a heap node,
while $i$ is not a leaf do
        $j \leftarrow$ left child of $i$
        if $i$ has right child and $A[\text{right child of } i].key > A[\,j\,].key$ then
                $j \leftarrow$ right child of $i$
        if $A[i]$.key $\geq A[\,j\,].key$                    // right child has  larger key
                break                                        // order is fixed, done
        swap $A[i]$ and $A[j]$
        $i \leftarrow j$                                     // move to one level down
```

- Pass n because for some usages of *fix-down*, A stores heap only in the front part



54	44	15	17	32	99	100

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

Pseudocode for deleteMax

```
deleteMax ( )
    $l \leftarrow last()$
    $toReturn = A[root()]$
    $A[root()] = A[l]$
    decrease $size$
    $fix\text{-}down(A, root(\,), size)$
    return toReturn
```

$l = 3$

54	32	15	17

size = 4

toReturn = 54

Pseudocode for deleteMax

```
$deleteMax\ (\ )$
    $l \leftarrow last()$
    $toReturn = A[root()]$
    $A[root()] = A[l]$
    decrease $size$
    $fix\text{-}down(A, root(\ ), size)$
    **return** $toReturn$
```

$l = 3$

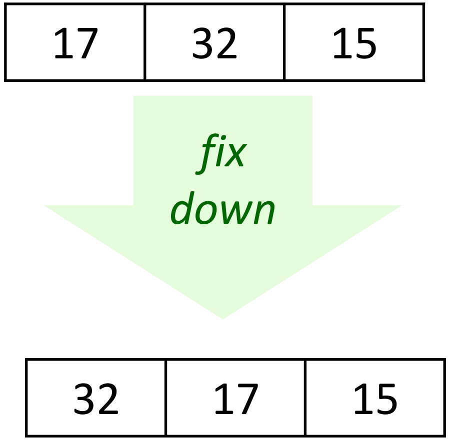
17	32	15	17

size = 4

toReturn = 54

Pseudocode for deleteMax

```
$deleteMax\,()$
    $l \leftarrow last()$
    $toReturn = A[root()]$
    $A[root()] = A[l]$
    decrease $size$
    $fix\text{-}down(A, root(\,), size)$
    **return** $toReturn$
```



17	32	15

32	17	15

$size = 3$

$toReturn = 54$

- *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

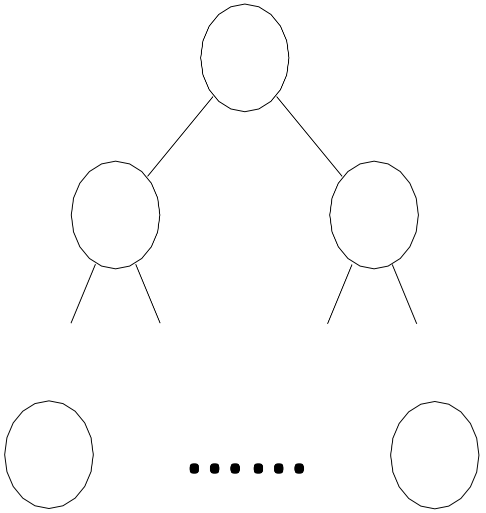
Sorting using Heaps

- Time to sort with priority queue is $O(init + n \cdot insert + n \cdot deleteMax)$

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

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

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



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

$$\frac{n+1}{2} \le 2^h \quad \Rightarrow \frac{n+1}{4} \le 2^{h-1}$$

$$\frac{n}{2} > 2^{h-1} > \frac{n}{4}$$

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

Sorting using Heaps

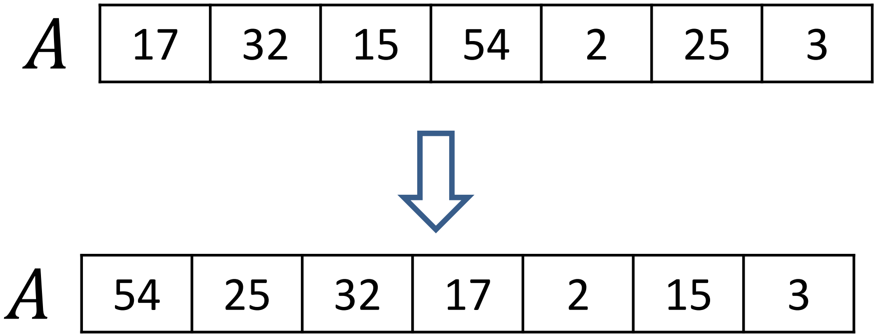
- Can sort with priority queue in $O(init + n \cdot insert + n \cdot deleteMax)$

```
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

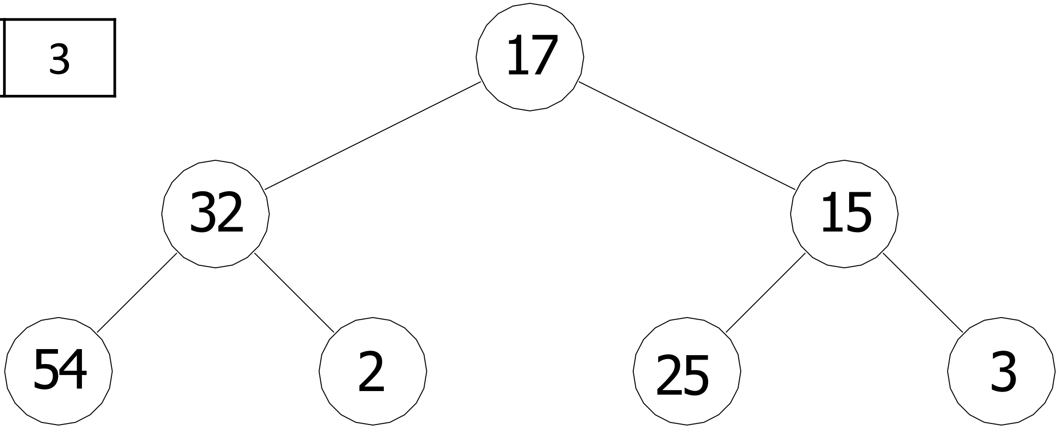
Problem statement: build a heap from n items in $A[0, \dots, n-1]$ without using additional space

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

Building Heap Directly In Input Array

A

17	32	15	54	2	25	3

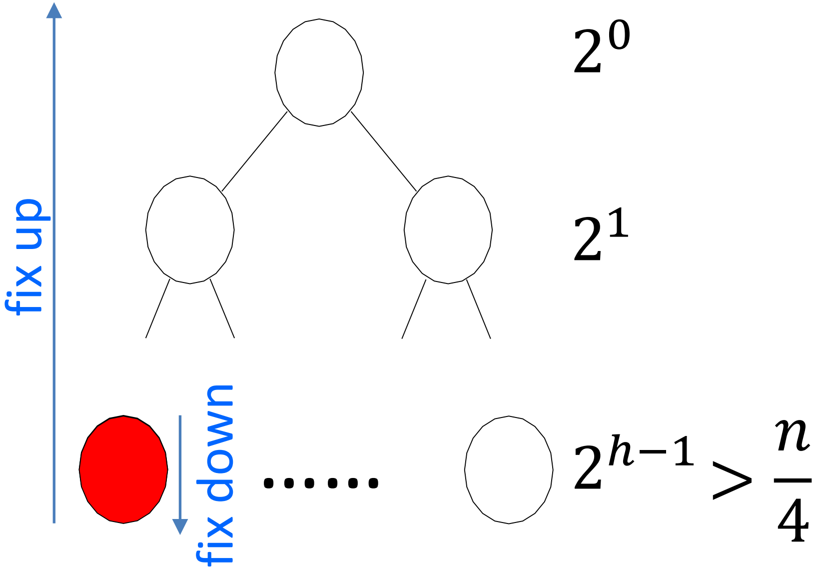


Problem statement: build a heap from n items in $A[0, \dots, n-1]$ without using additional space

- i.e. put items in $A[0, \dots, n-1]$ in heap-order
- Look at array A as a binary tree
- Heap-order (most likely) does not hold
- But we can change the order to obey heap-order
    - 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

- At least $\frac{n}{4}$ nodes at level $h-1$
- For each such node
  - fix-up takes $O(\log n)$ time
  - fix-down takes $O(1)$ time

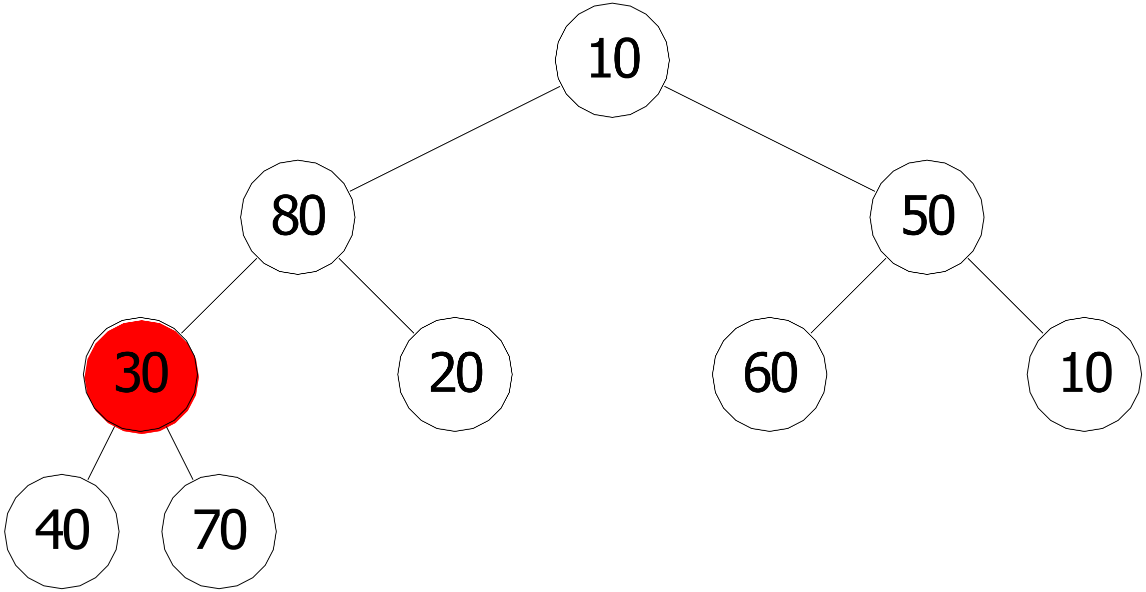


- Fix-up called for all $\frac{n}{4}$ nodes at level $h-1$ takes $O(n \log n)$ time
- Fix-down for all $\frac{n}{4}$ nodes at level $h-1$ takes $O(n)$ time

Heapify Example

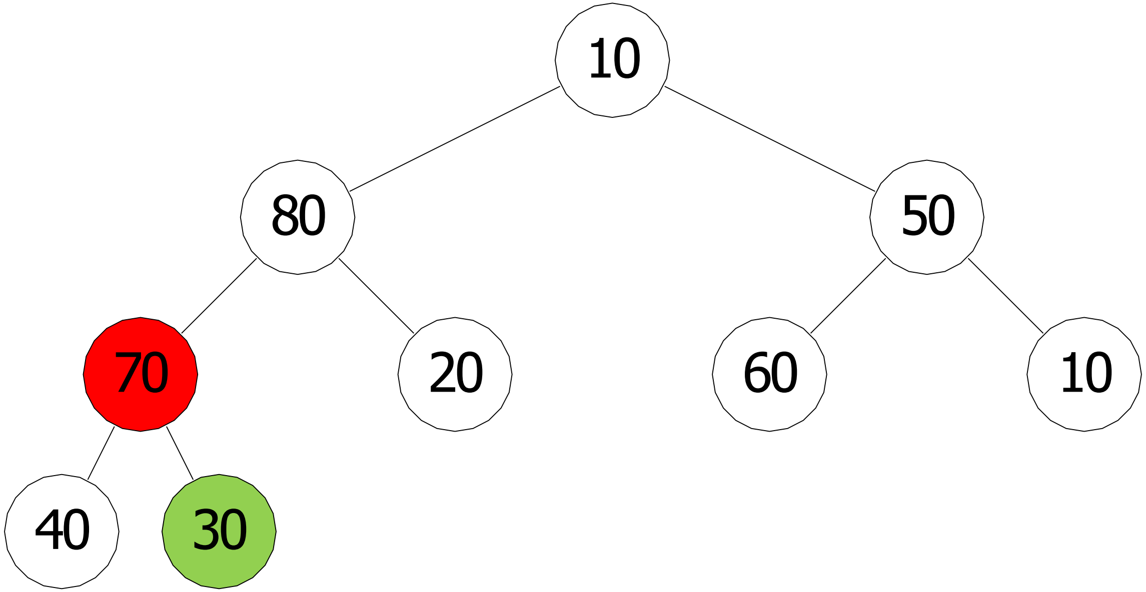
A

10	80	50	30	20	60	10	40	70

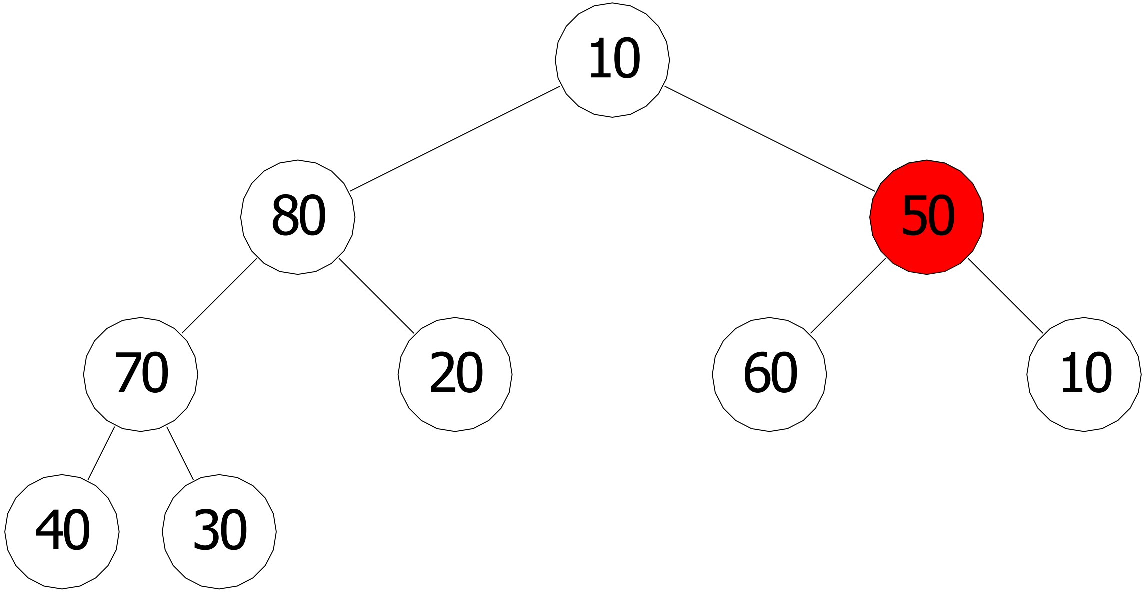


- No need to call *fix-down* on the leaves
    - no harm, but *fix-down* will do nothing for the leaves
- Start calling *fix-down* with the parent of last node
    - this is the deepest and rightmost non-leaf node

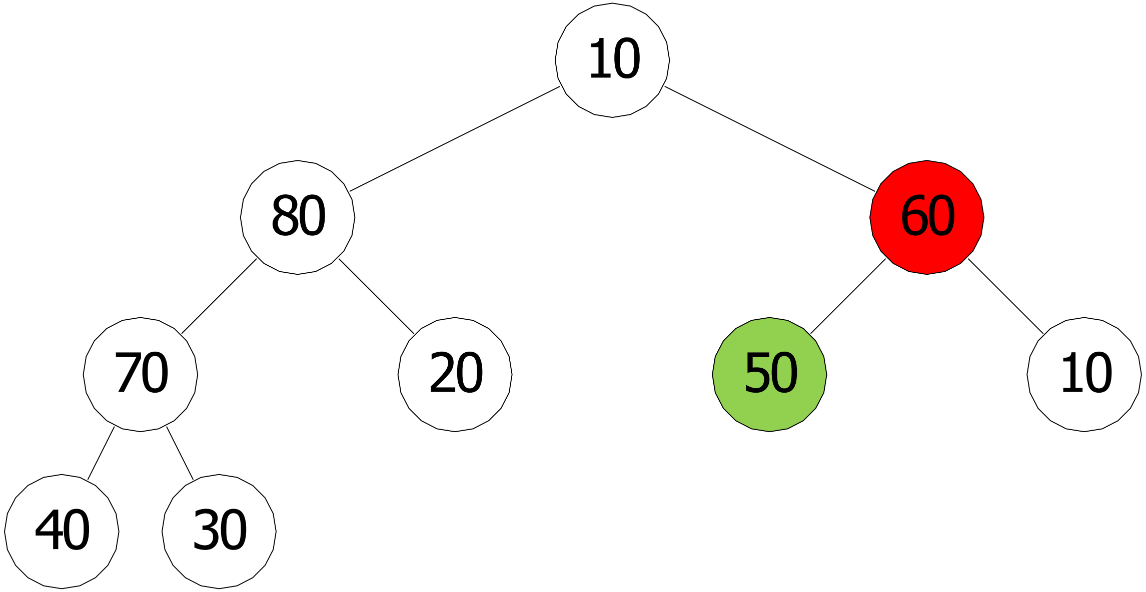
Heapify Example



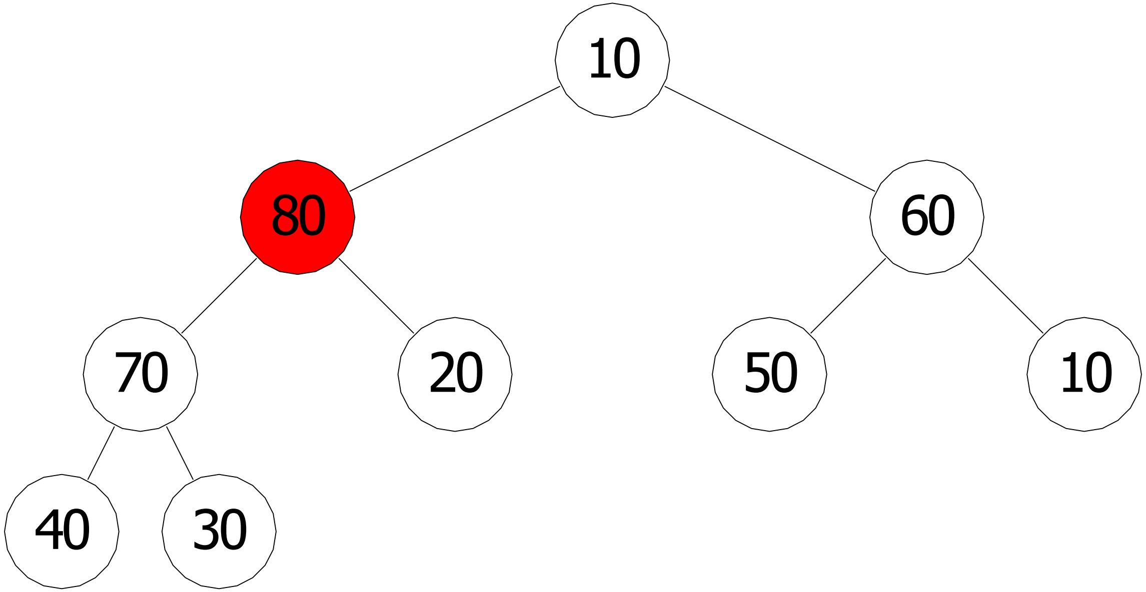
Heapify Example



Heapify Example

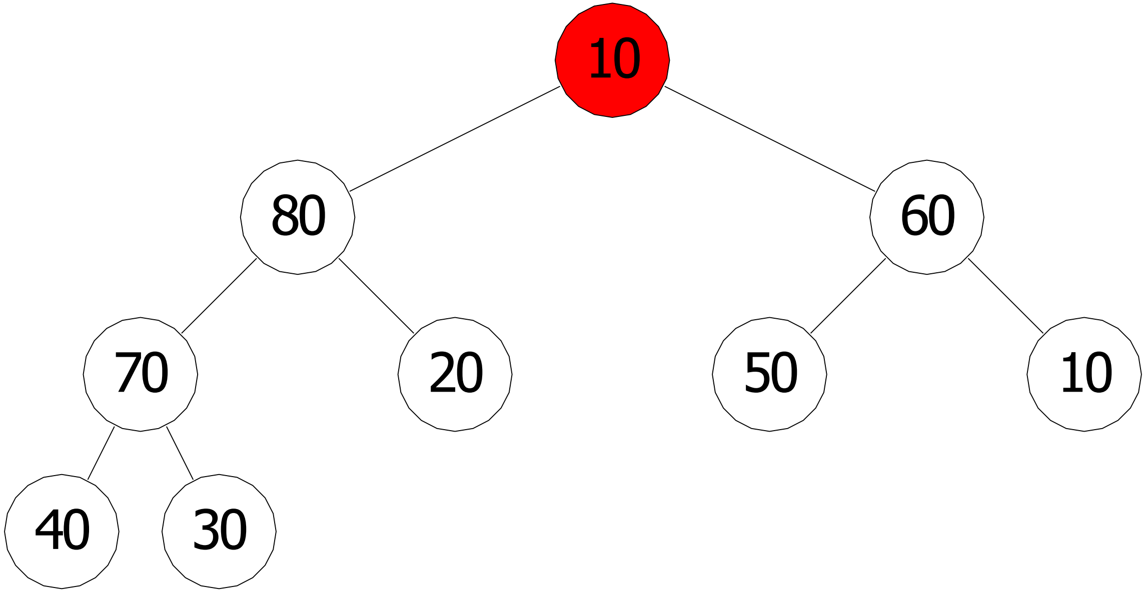


Heapify Example

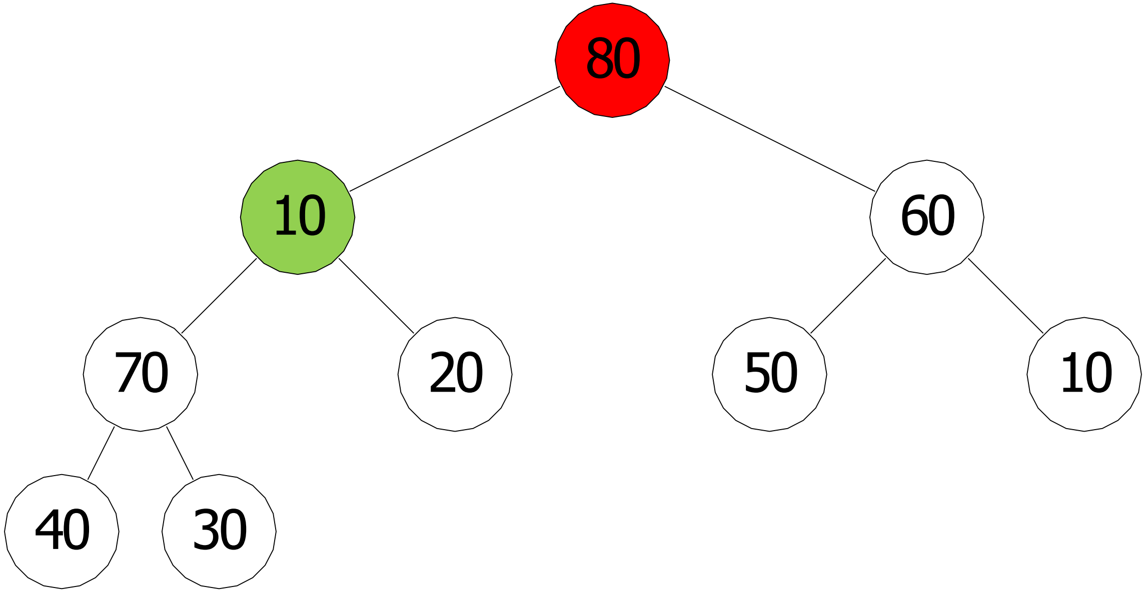


no need to do anything

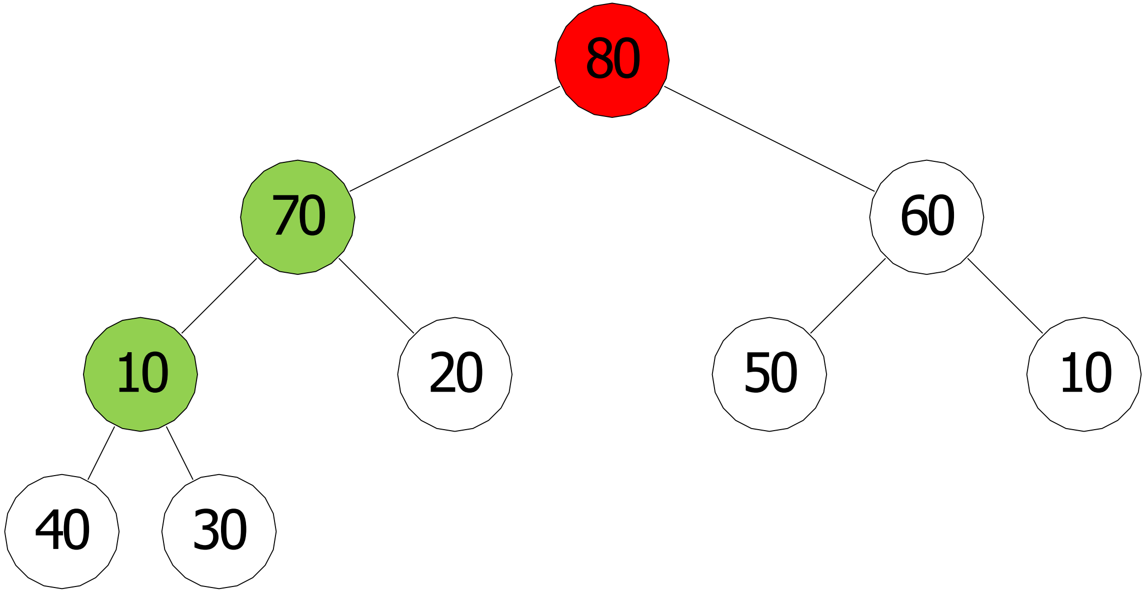
Heapify Example



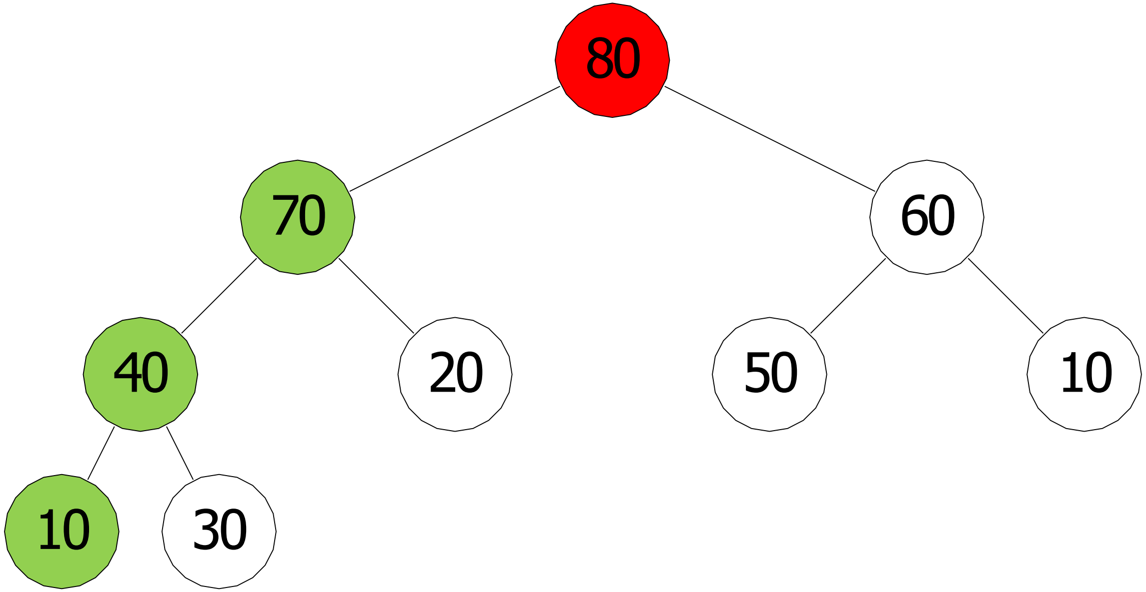
Heapify Example



Heapify Example



Heapify Example



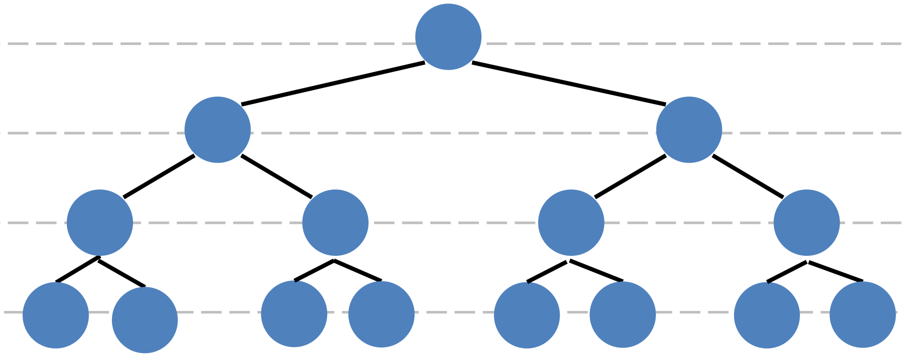
done!

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



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

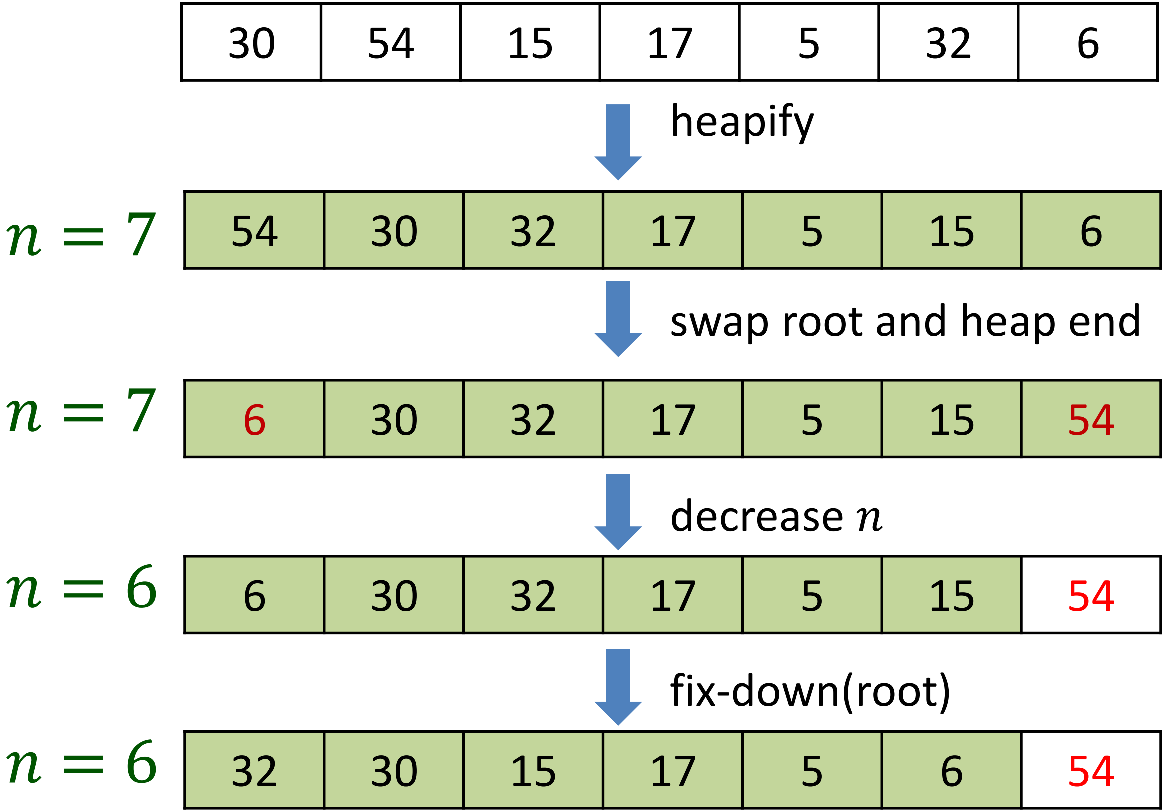
$$\sum_{i=0}^{h-1} 2^i(h-i) = 2^h \sum_{i=0}^{h-1} \frac{2^i(h-i)}{2^h} = 2^h \sum_{i=0}^{h-1} \frac{(h-i)}{2^{h-i}}$$

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

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

$$\lim_{i \to \infty} \frac{2^i(i+1)}{i2^{i+1}} = \frac{1}{2}$$

HeapSort



30	54	15	17	5	32	6

heapify

$n = 7$

54	30	32	17	5	15	6

swap root and heap end

$n = 7$

6	30	32	17	5	15	54

decrease n

$n = 6$

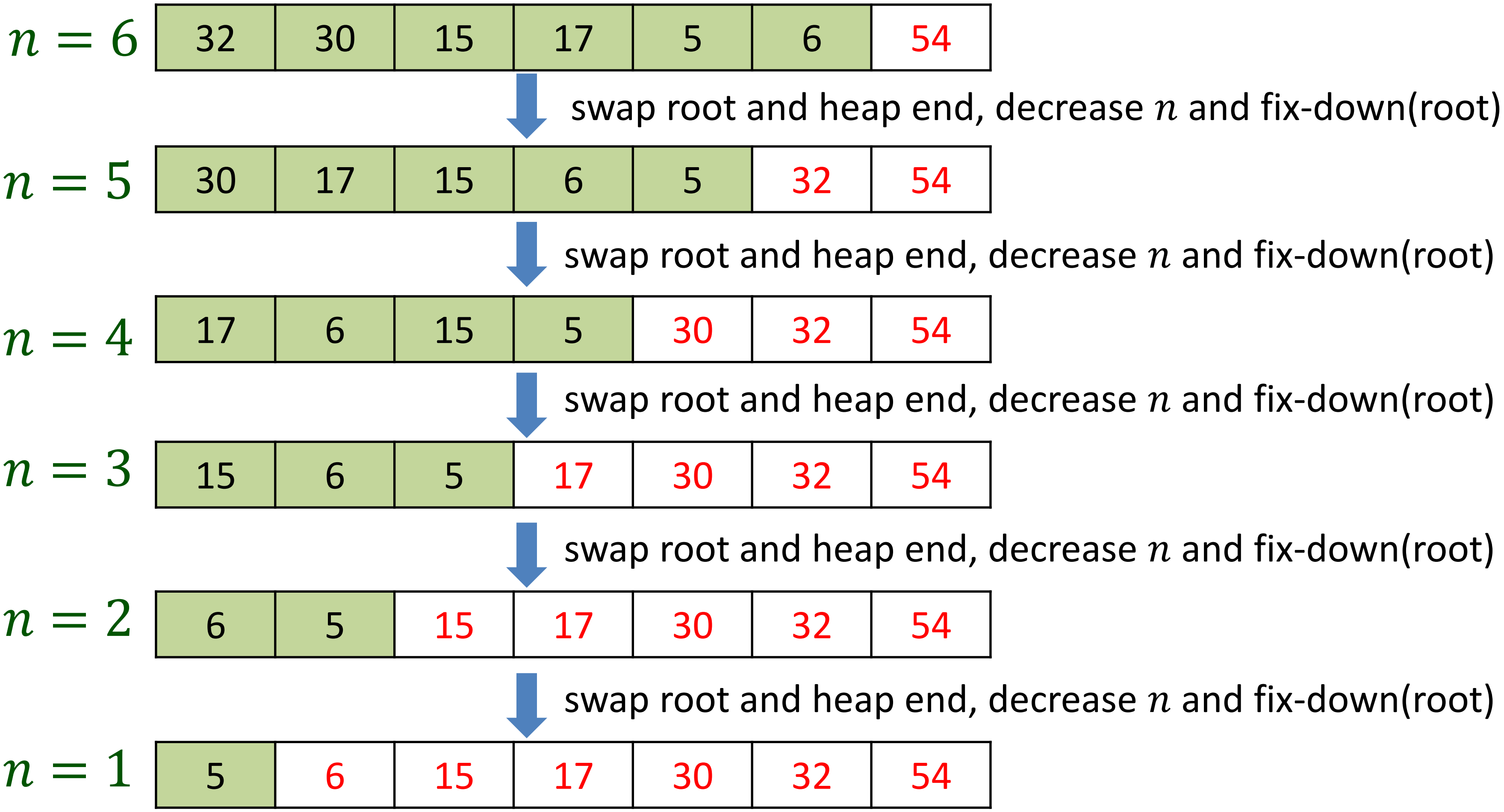
6	30	32	17	5	15	54

fix-down(root)

$n = 6$

32	30	15	17	5	6	54

HeapSort



$n = 6$	32	30	15	17	5	6	54

swap root and heap end, decrease n and fix-down(root)

$n = 5$	30	17	15	6	5	32	54

swap root and heap end, decrease n and fix-down(root)

$n = 4$	17	6	15	5	30	32	54

swap root and heap end, decrease n and fix-down(root)

$n = 3$	15	6	5	17	30	32	54

swap root and heap end, decrease n and fix-down(root)

$n = 2$	6	5	15	17	30	32	54

swap root and heap end, decrease n and fix-down(root)

$n = 1$	5	6	15	17	30	32	54

Sorted!

HeapSort

```
HeapSort(A)
 $n \leftarrow A$.size()
   for $i \leftarrow$ parent (last()) downto 0 do
       fix-down $(A, i, n)$
   while $n > 1$
       swap items $A[root()]$ and $A[last()]$
       decrease $n$
       fix-down$(A, root(), n)$
```

heapify $\Theta(n)$

$\Theta(n \log n)$

- 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 an in-place 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

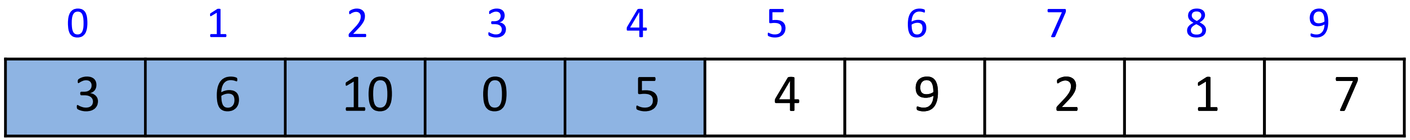
Selection

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

- Define

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

Selection



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

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