Do not distribute course material

You may not and may not allow others to reproduce or distribute lecture notes and course materials publicly whether or not a fee is charged.

Merge sort is used in data bases!
It is a key component in Hadoop, a distributed file system.

Merge Sort continued

Merge Sort

43	37	20	25	4	72	15	19

DIVIDE IN HALF

43	37	20	25

4	72	15	19

CONQUER: SORT EACH HALF

20	25	37	43

left

4	15	19	72

right

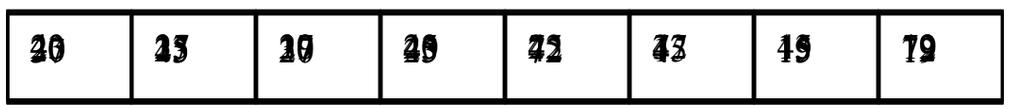
COMBINE: MERGE

4	15	19	20	25	37	43	72

Pseudocode

```
MERGE-SORT(A, p, r)
  if p < r                          ← check for base case
    q = ⌊(p + r)/2⌋                 ← divide
    MERGE-SORT(A, p, q)             ← conquer
    MERGE-SORT(A, q+1, r)           ← conquer
    MERGE(A, p, q, r)               ← combine
```

A

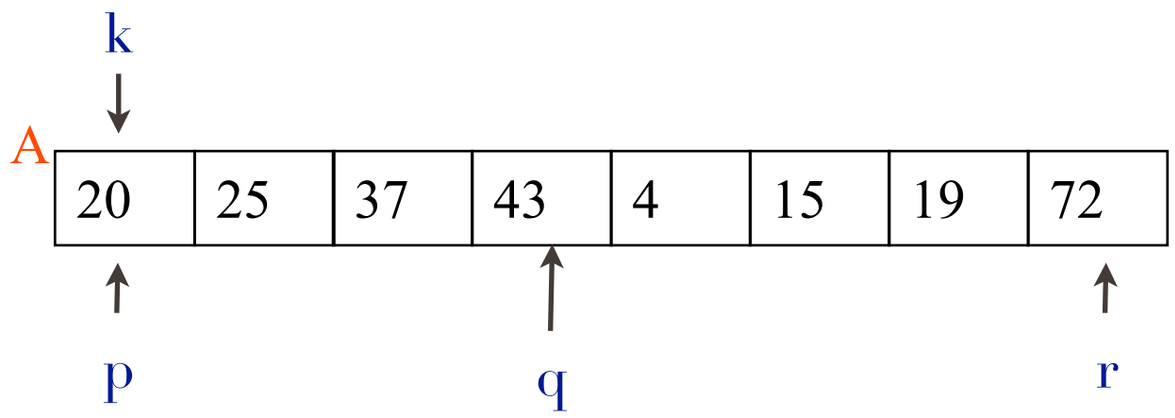


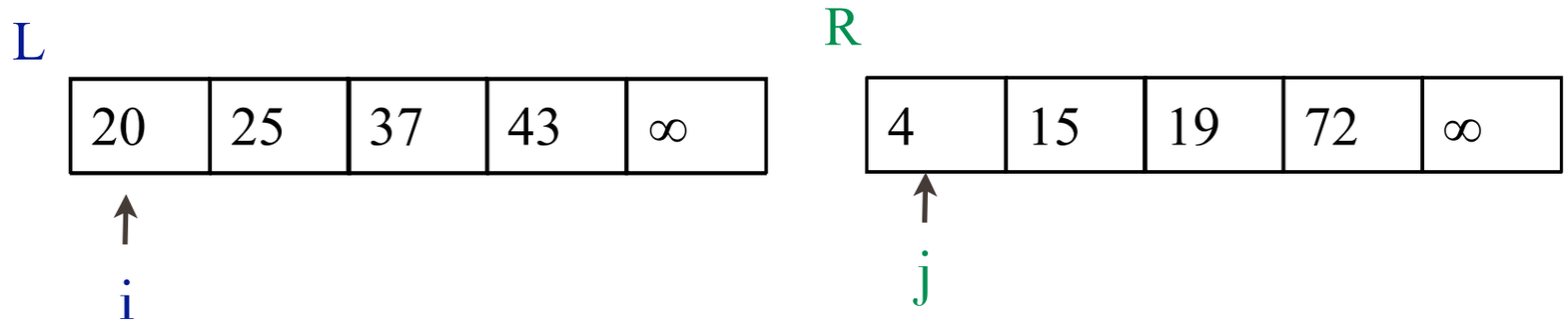
MERGE-SORT(A, 1, 1)

MERGE-SORT(A, 1, 2)

MERGE-SORT(A, 1, 4)

MERGE-SORT(A, 1, 8)





MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

Every time I go through the loop I perform a constant amount of work

See page 31 and 33 in the textbook for a proof that the merge function works correctly

Pseudocode

MERGE-SORT(A, p, r) $\qquad T(n) =$

if p < r
  q = ⌊(p + r)/2⌋ $\}$ $\Theta(1)$

  MERGE-SORT(A, p, q) $\}$ $2 \cdot T(n/2)$

  MERGE-SORT(A, q+1, r)

  MERGE(A, p, q, r) $\qquad \Theta(n)$

Running time: $T(n) = ?$

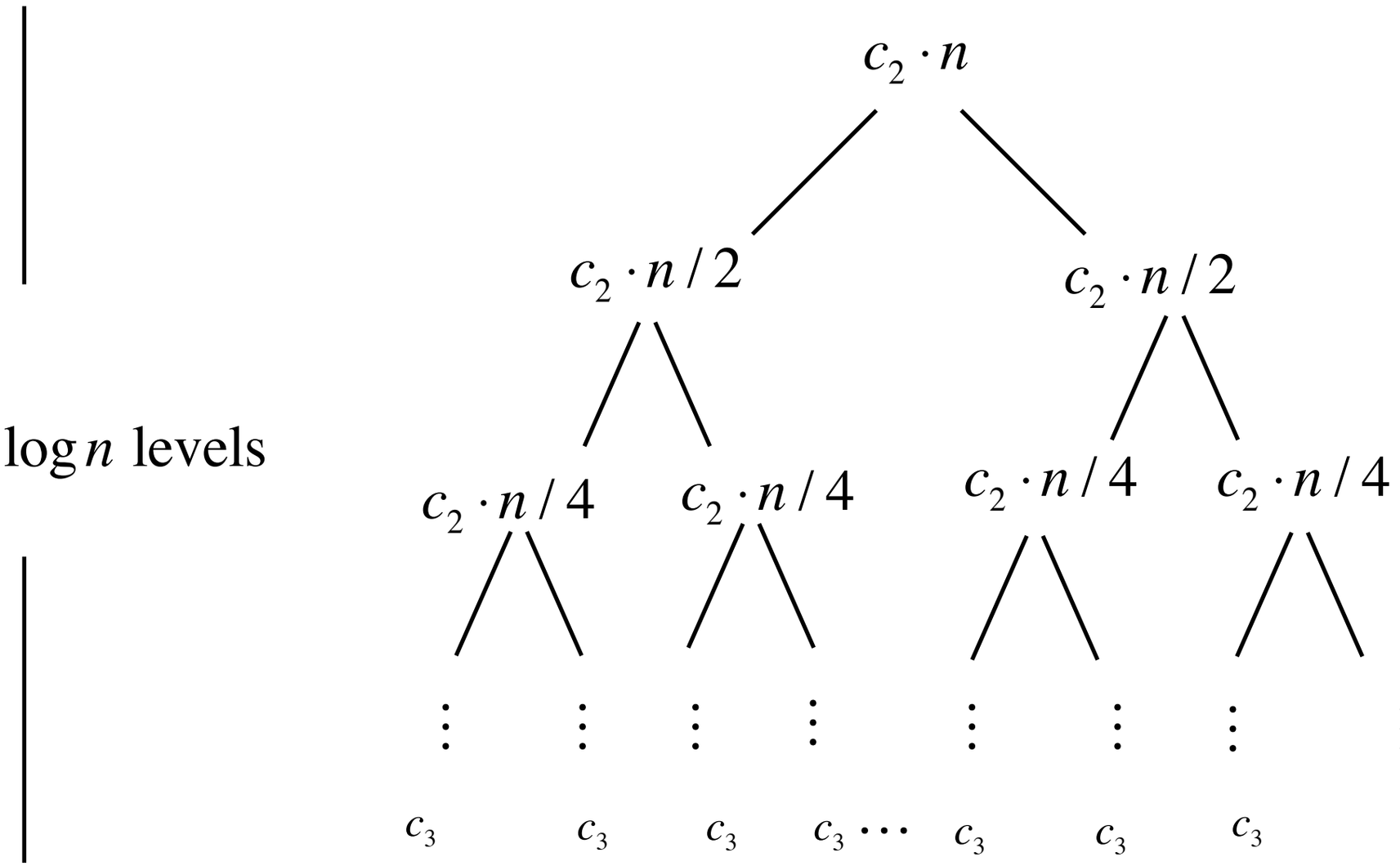
$$T(1) = \Theta(1)$$

$$T(n) = \Theta(1) + 2 \cdot T(n/2) + \Theta(n)$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

Here for simplicity, we assume n = 2^n

$$T(n) = \begin{cases} c_3 & \text{if } n = 1, \\ 2T(n/2) + c_2 \cdot n & \text{otherwise} \end{cases}$$



# per level	level cost
2^0	$1 \cdot c_2 \cdot n$
2^1	$2^1 c_2 \cdot \frac{n}{2}$
2^2	$2^2 c_2 \cdot \frac{n}{2^2}$
⋮	⋮
$2^{\log_2 n} = n$	$n \cdot c_3 \cdot \frac{n}{n}$

Total $c_3 \cdot n + c_2 \cdot n \cdot \log n$

therefore $T(n)$ is $\Theta(n \log n)$

The merge sort algorithm took $T(n) = 2T(n/2) + \theta(n)$ time, here we are simplifying the formula to make the explanation easier

Pseudocode

MERGE-SORT(A, p, r) $\quad T(n) =$

if p < r
  q = ⌊(p + r)/2⌋ $\}$ $\Theta(1)$

  MERGE-SORT(A, p, q)
  MERGE-SORT(A, q+1, r) $\}$ $2 \cdot T(n/2)$

  MERGE(A, p, q, r) $\quad \Theta(n)$

Running time: $T(n)$ is $\Theta(n \log n)$

Hot shot Programmer/amazing machine vs
a good algorithm/average machine/bad compiler

insertion sort

merge sort

Computer A

Best computer

10 billion instructions per second

Hot shot programmer

$2n^2$ instructions to sort n numbers

Computer B

ok computer

10 million instructions per second

Average programmer

(inefficient compiler)

50 n log(n) instructions to sort n numbers

n	Computer A	Computer B
10	$\frac{2*(10)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$	$\frac{50*10\lg(10) \text{ instructions}}{10^7 \text{ instructions/second}}$
10 million	$\frac{2*(10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$	$\frac{50*10^7\lg(10^7) \text{ instructions}}{10^7 \text{ instructions/second}}$
100 million	$\frac{2*(10^9)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$	$\frac{50*10^9\lg(10^9) \text{ instructions}}{10^7 \text{ instructions/second}}$

insertion sort merge sort

Computer A

Best computer

10 billion instructions per second

Hot shot programmer

$2n^2$ instructions to sort n numbers

Computer B

ok computer

10 million instructions per second

Average programmer

(inefficient compiler)

50 n log(n) instructions to sort n numbers

n		
10	20 nanoseconds	166.1 microseconds
10 million	5 hours 33 min. 20 seconds	19 min. 22.67 seconds
100 million	6 years 4 months 3.1 days	1 day 17 hours 31 min. 22.76 second

Running Times

n	insertion O(n^2)	merge O(nlog(n))
512	0.0018	0.000166
1024	0.00747	0.000356
2048	0.030801	0.000765
4096	0.120905	0.001644
8192	0.474183	0.003504
16384	1.87247	0.007584
32768	7.56883	0.015684
65536	29.9251	0.033017

Asymptotic Notation Examples

1. $\log_{200} n$ $\in \Theta(\log_2 n)$
2. $\log n!$ $\in \Theta(n \log n)$
3. $n! + C^n$ $\in \Theta(n!)$
4. $n! + n^n$ $\in \Theta(n^n)$
5. $\log^{200} n = (\log n)^{200}$ $\in \Theta(\log^{200} n)$

Useful Facts:

Change of basis

$$\log_a b = \log_a c^{\log_c b} = \log_c b \log_a c$$

Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+O\left(\frac{1}{n}\right)\right)$$

Some Important Big-Oh's

• O(1)	constant	assignment statement invoking a function
• O(log n)	logarithmic	binary search inserting into a red-black tree
• O(n)	linear	searching in an unordered list inserting a new item into a sorted vector
• O(n log n)	n log n	mergesort
• $O(n^2)$	quadratic	insertion sort
• $O(n^3)$	cubic	Brute force maximum subsequence sum algorithm
• $O(2^n)$	exponential	Finding all subsets of n items

Elementary Data Structures

It is all about the data!

- Basic Abstract Data Types (ADTs)
- Implementing a Priority Queue
  - Binary Heap
- Heap Sort

Data Structure

"A ***data structure*** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, …"

ADT (Abstract Data Type)

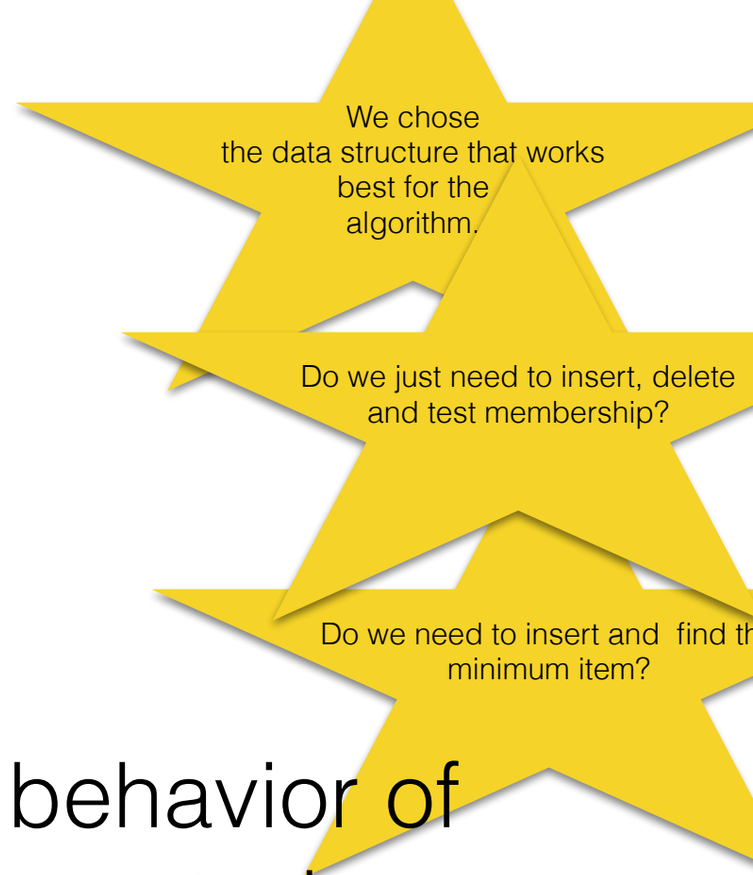
The Specification

A mathematical model of the data

methods used to modify and access the data

~~how the data organized in memory~~
~~what algorithms implement the methods~~

ADT



When designing an algorithm we use the behavior of the ADT *without thinking* of how it is implemented.

We have different implementations of the ADT, so we can *optimize* for performance

ADT - Abstract Data Types

- list, stack, queue, priority queue, set, dictionary

Common data structures

- array, linked list, hash table, tree (binary heap, binary search tree, red-black tree, 2-3 tree, b-tree)

“Most programers have seen them, and most good programmers realize they’ve written at least one. They are huge, messy, ugly program that should have been short, clean, beautiful programs.”

“Why do programmers write big programs when small ones will do? … arrays are typically used as fixed tables that are initialized at the beginning of a program and never altered.”

https://en.wikipedia.org/wiki/Toolbox#mediaviewer/File:Gereedschapskist.jpg

We often use these when we design an algorithm

Data Structure Needed For:

free list used to keep track of free memory in the heap (free store)

Determining if any of a company's customers is on a list of hacked SSN

Order fulfillment system for a company

Finding a name in a telephone book, and all the people with the same last name

Buffering video as it is streamed

Storing the address of recently visited web sites

Scheduling airplanes landing on Newark's runway

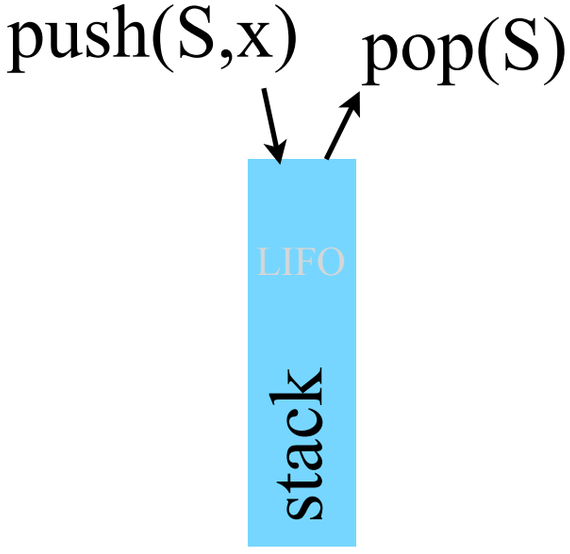
Creating a continuous limit order book (buyers post bids for stocks, and seller post offers to sell)

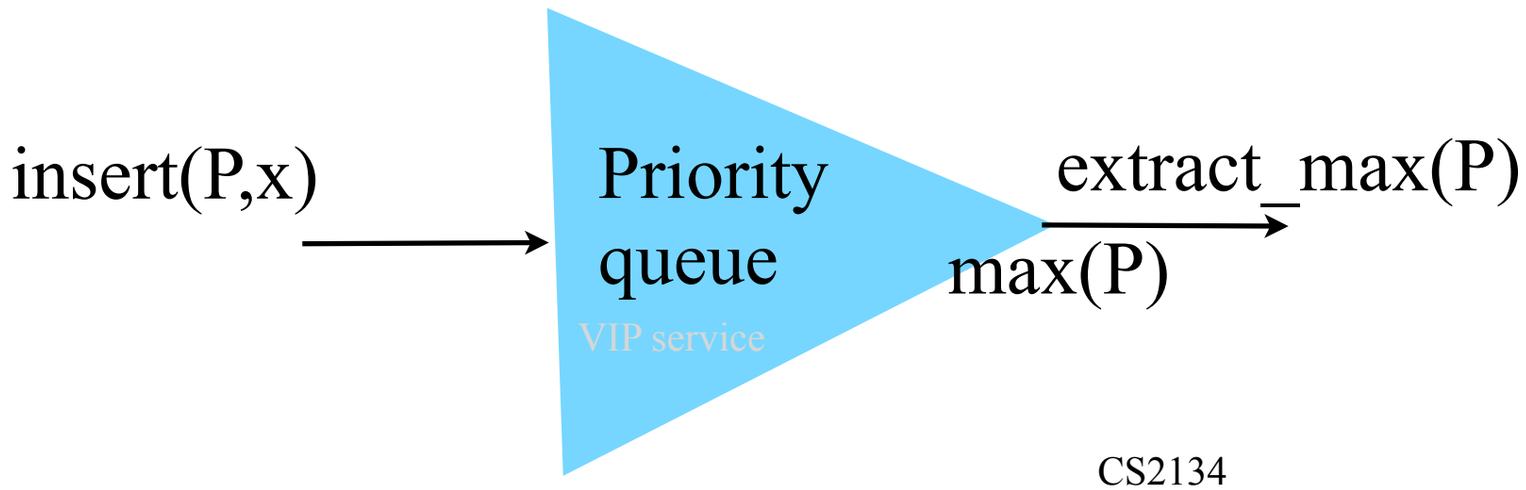
Online ticketing requests

ADT's

stack, queue, priority queue

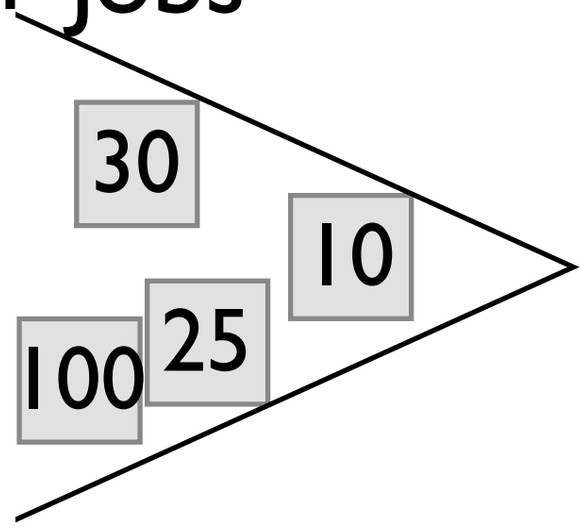
Retrieve items as a function of when it arrived, or its priority



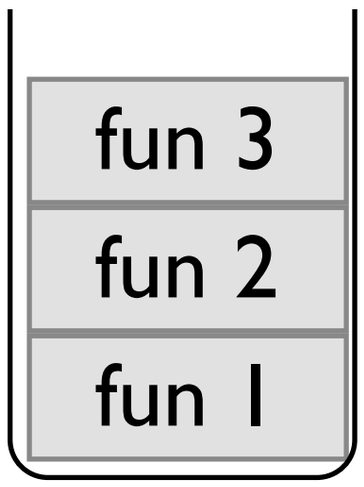


Which ADT data structure should you use?

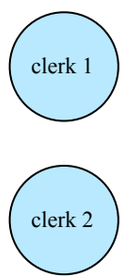
Printer Jobs



Function Calls

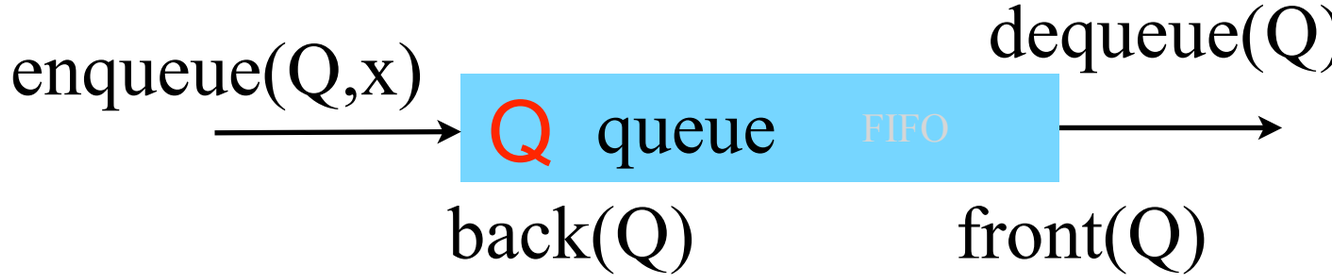


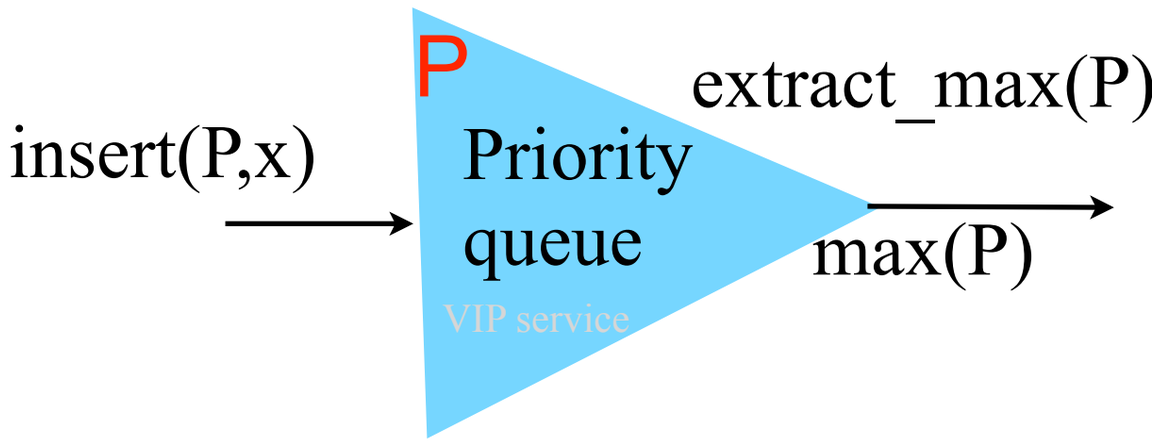
customersInLine

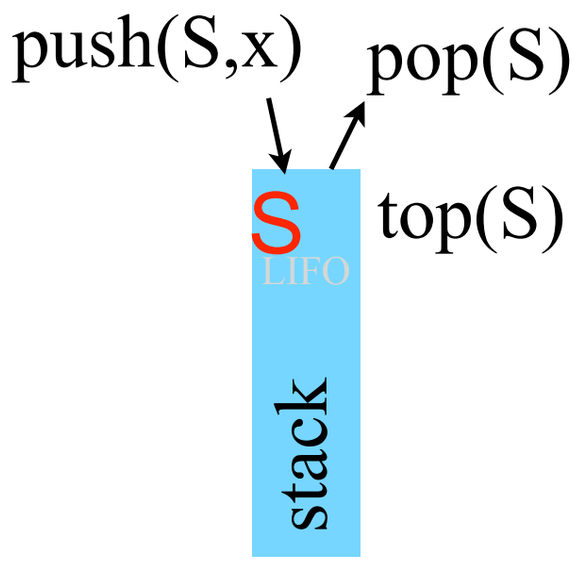


ADT's

stack, queue, priority queue







computer 1

computer 2

printer queue

printer

Sue's	Sue's	Sam's

stored video file

video buffering

p3	p2	p1

video player

Back Button for Web Pages

nyu.ed

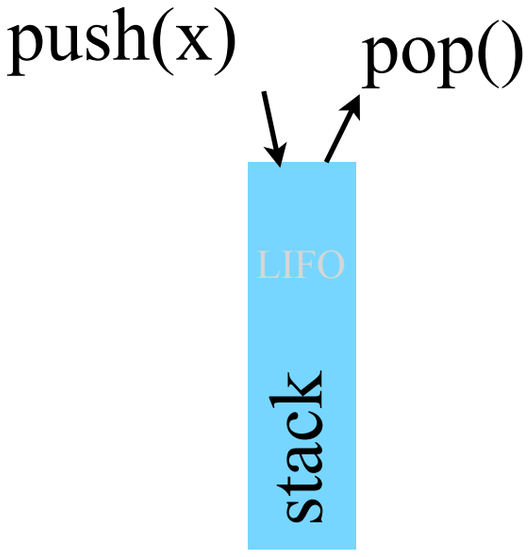
yahoo.com

google.com

Which ADT data structure should you use?

ADT's

stack, queue, priority queue



Running Time for this implementation?

Push(S,x)

Pop(S)

Stack-Empty(S)

How long does it take to enter a new item into a stack in the worst case (assuming there is room in the array implementation)?

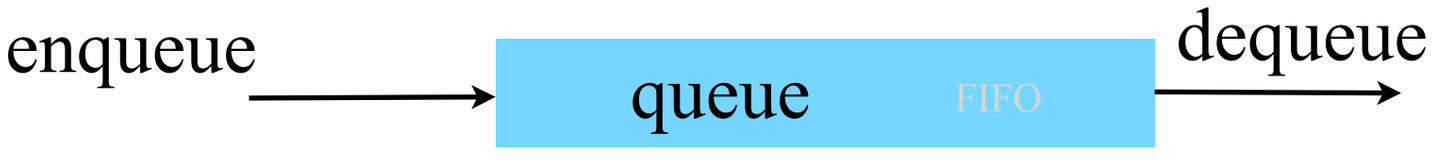
- O(1)
- O(log n)
- $O(n^{1/2})$
- O(n)
- None of the above

Total Results: 0

ADT's

stack, queue, priority queue

Implementation using an array



Instance attributes: Q.head, Q.tail, Q.length

Running Time for this implementation?

Enqueue(Q,x)

Dequeue(Q)

Queue-Empty(Q)

Q.head is the index of the most oldest item in the queue

Q.tail is the index just after the most recently inserted item

Q[Q.head .. Q.tail-1] is the queue where we "wrap around"

Q.length is the size of the array

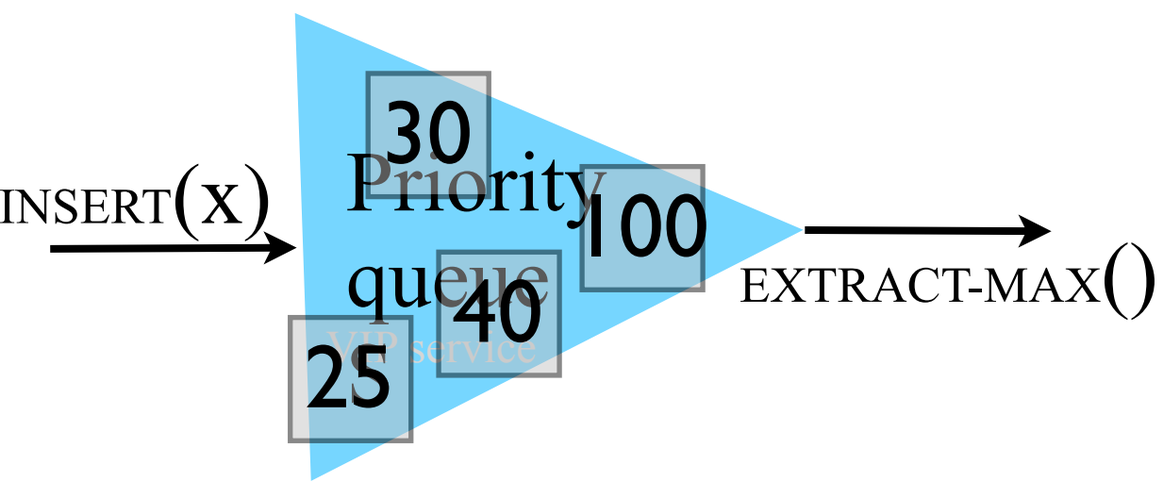
In the worst case, how long does it take to remove an item from a queue?

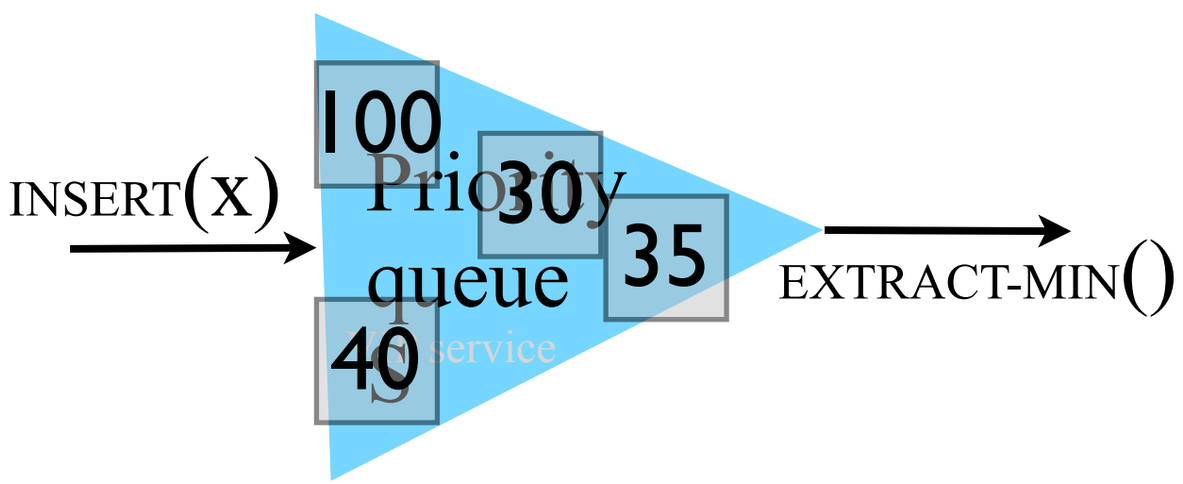
- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- None of the above

Total Results: 0

ADT Priority Queue

Stores data that is associated with a key
(key from a linearly ordered set)





max-priority queue

INSERT(S,x)

MAXIMUM(S)

EXTRACT-MAX(S)

INCREASE-KEY(S,x,k)

min-priority queue

INSERT(S,x)

MINIMUM(S)

EXTRACT-MIN(S)

DECREASE-KEY(S,x,k)

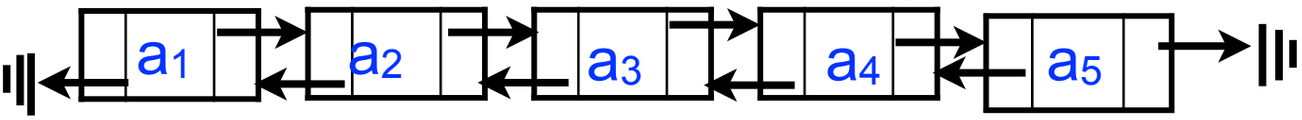
Applications

- Operating systems - process jobs by priority
- Graph algorithms - subroutine in a Dijkstra's shortest path algorithm
- Event simulation - find next event to happen
- Compression - Huffman encoding
- Heapsort

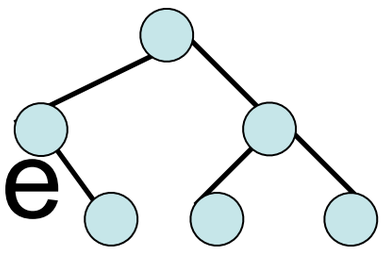
Pair share (write down the question)

How should we implement the priority queue?

- sorted list (array)?
- unsorted list(array)?



- sorted list (linked list)?
- unsorted list(linked list)?
- queue?
- stack?
- binary search tree



- balance binary search

INSERT(S,x)	EXTRACT-MIN(S)

Priority Queues are often implemented with a binary heap

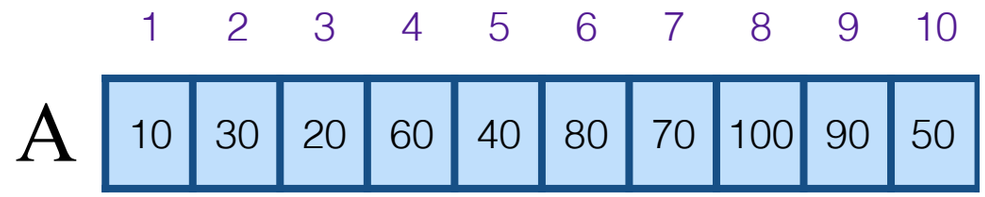
Each element in the array represents a tree node. How?

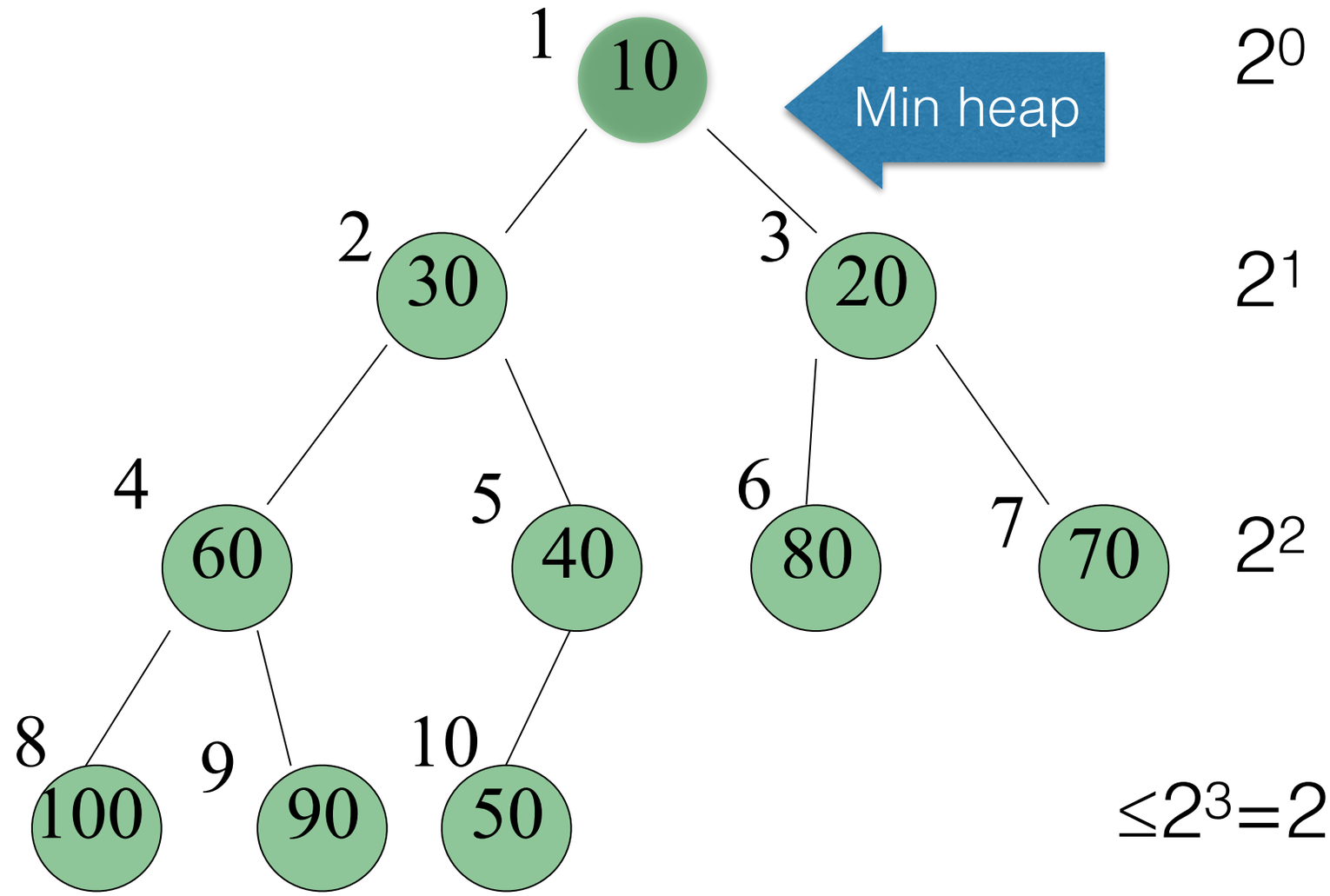
Elements stored in A[1..heap-size]

Binary Heap

Attributes:
length
heap-size

- array structure
- visualized as a *nearly complete tree*
- root of tree *is A[1]*
- PARENT(i) = i/2
- LEFT(i)= 2i
- RIGHT(i) = 2i + 1
- computing is fast with binary representation





What is the maximum number of nodes in a tree of height h?

- $2h$
- 2^h
- $2^h + 1$
- 2^{h-1}
- $2^{h-1} + 1$
- $2^{h+1} - 1$
- None of the above

Theorem: **Max heap**

A nearly complete binary tree of n nodes has height $\Theta(\log n)$.

Proof:

The number of nodes in a complete binary tree where all levels have the maximum number of nodes is $2^{h+1}-1$ where h is the height of the tree.

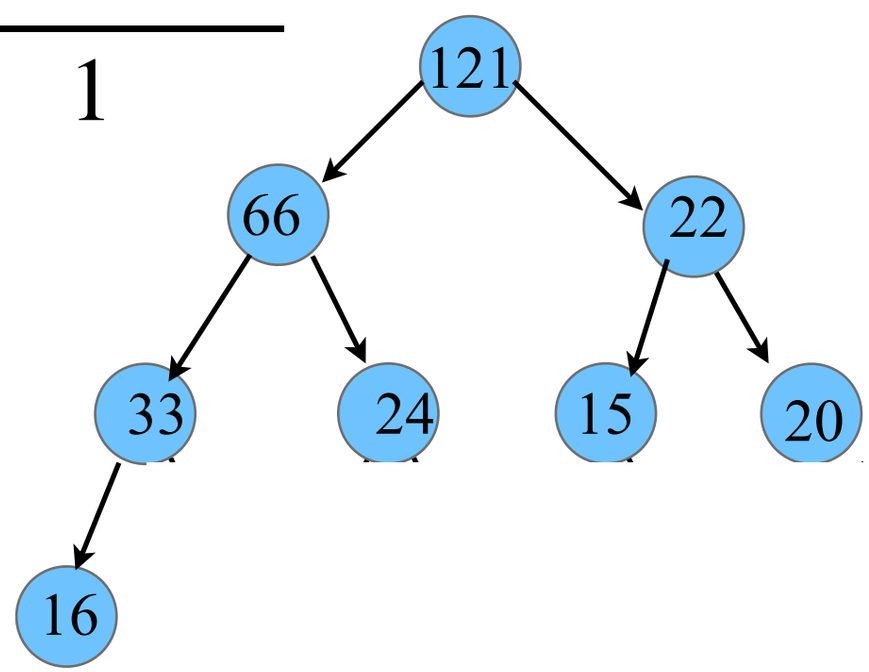
Therefore the number of nodes of a nearly complete tree of height h is $2^h \le n \le 2^{h+1}-1$

$h \le \log_2 n < h+1$

Max# nodes $$\sum_{i=0}^{h} 2^i = \frac{1-2^{h+1}}{1-2} = \frac{-1+2^{h+1}}{1}$$

Min# nodes $$1 + \sum_{i=0}^{h-1} 2^i = 1 + \frac{1-2^h}{1-2}$$

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



Level	#nodes
0	2^0
1	2^1
2	2^2
3	$\le 2^3=2^h$

Last node

Binary **Heap** Data Structure

- Nearly Complete binary tree: binary tree in which all levels are “full”, except for possibly bottom level which is filled in from the left
- Where h is height and n is size is $h = \Theta(\log n)$
- A **min binary heap** (min-heap for this lecture) is a complete binary tree in which every node satisfies the heap property:
  - PARENT(A[I]) <= A[i]
- A **max binary heap** (max-heap) is a complete binary tree in which every node satisfies the heap property:
  - PARENT(A[I]) >= A[I]
- Observe
  - By induction and transitivity of <=, in a min heap, the smallest node is at the root. Similarly for a max heap
  - heap is NOT Binary Search Tree

“**Heap** may refer to:

Computer science

- Heap (data structure), a data structure commonly used to implement a priority queue
- Heap (or *free store*), an area of memory used for dynamic memory allocation

Mathematics

- Heap (mathematics), a generalization of a group”

From http://en.wikipedia.org/wiki/Heap

Min-heap operations

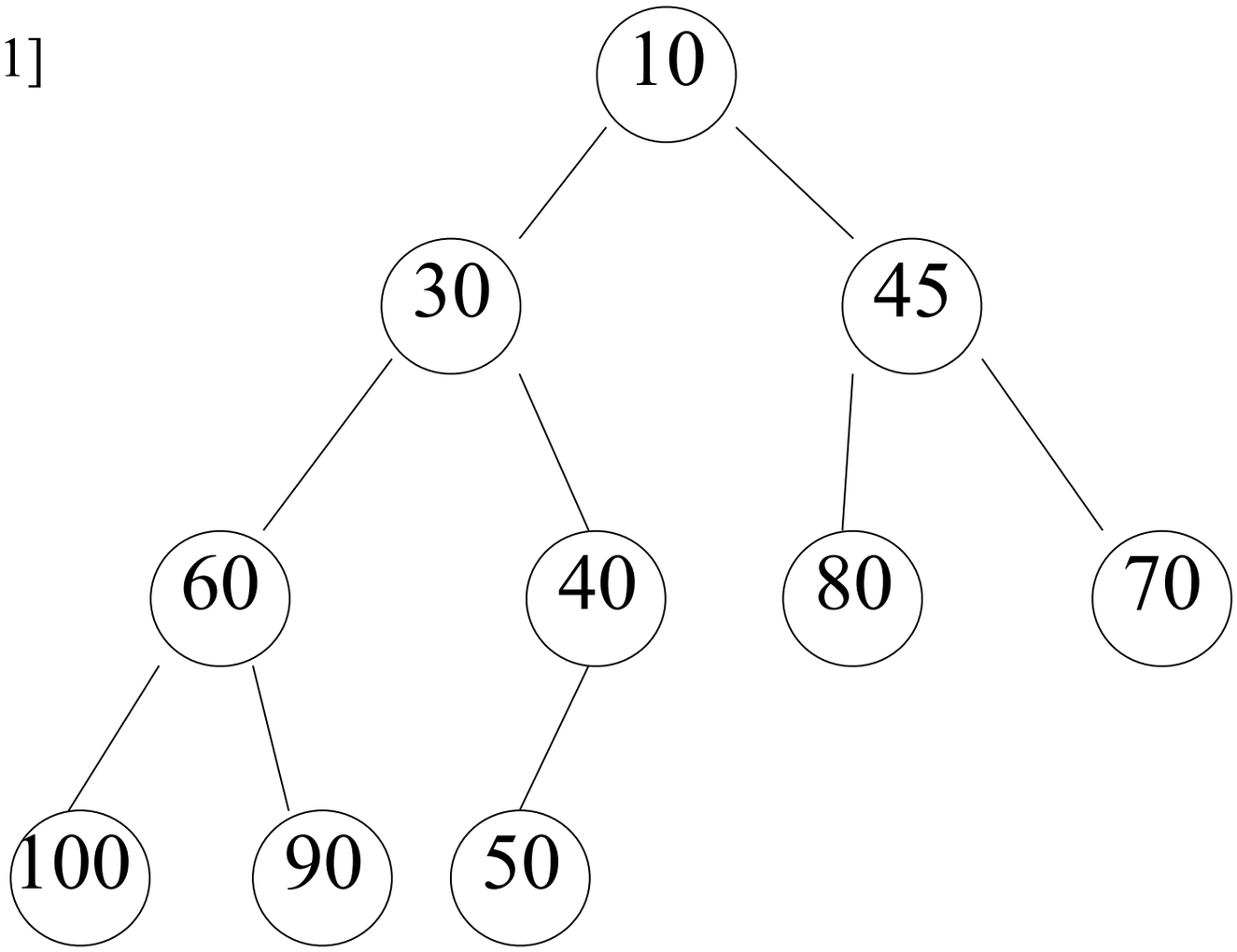
- HEAP-MINIMUM(A)
- HEAP-EXTRACT-MIN(A)
- MIN-HEAP-INSERT(A,key)
- HEAP-DECREASE-KEY(A, i, key)
- BUILD-MIN-HEAP(A) // build a heap from an unsorted array
- MIN-HEAPIFY(A,i)

A max heap works the same as a min heap:
exchange min for max
and when comparing keys
exchange < for >
and ≤ for ≥

Finding the Minimum item in a Min Heap

HEAP-MINIMUM(A)

return A[1]



10	30	45	60	40	80	70	100	90	50
1	2	3	4	5	6	7	8	9	10

Running time?

When just one item is out of place…

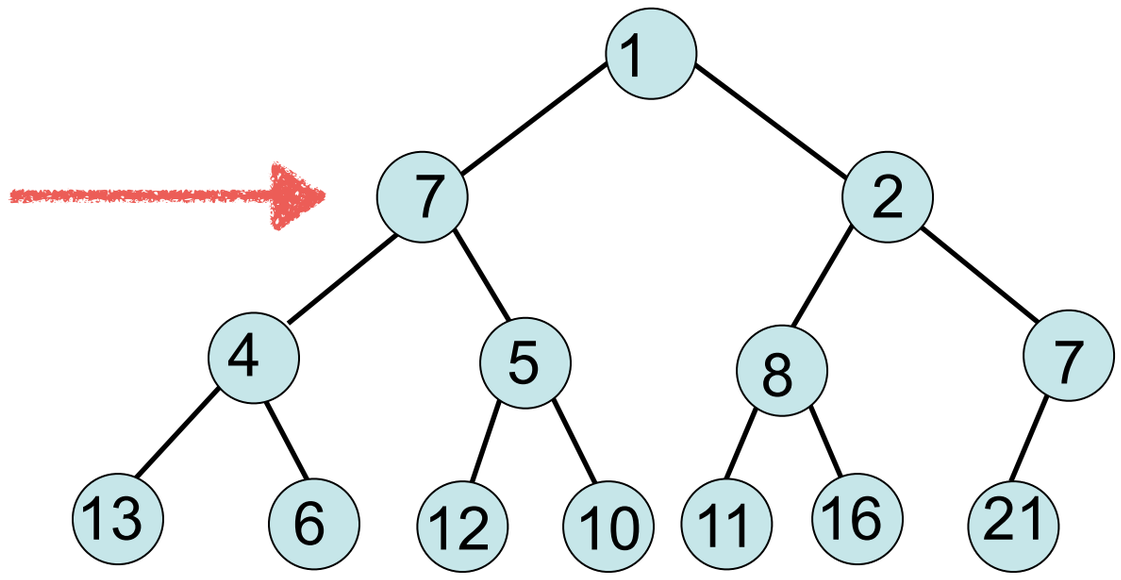
MIN-HEAPIFY(A, 2)

exchange A[2] with A[4]

MIN-HEAPIFY(A, 4)

exchange A[4] with A[9]

Assume trees rooted at LEFT(i) and RIGHT(i) are min-heaps

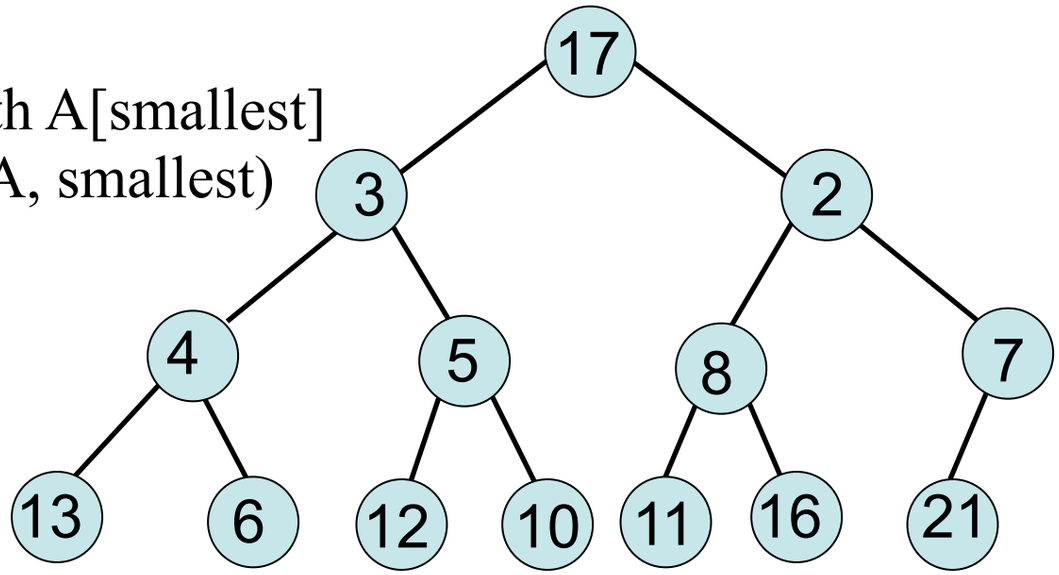


Maintaining the heap order property

```
MIN-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap-size and A[l] < A[i]
            smallest = l
    else smallest = i
    if r <= A.heap-size and A[r] < A[smallest]
            smallest = r
    if  smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

MIN-HEAPIFY(A,7)

percolateDown



1	2	3	4	5	6	7	8	9	10	11	12	13	14
17	3	2	4	5	8	7	13	6	12	10	11	16	21
2	3	17	4	5	8	7	13	6	12	10	11	16	21
2	3	7	4	5	8	17	13	6	12	10	11	16	21

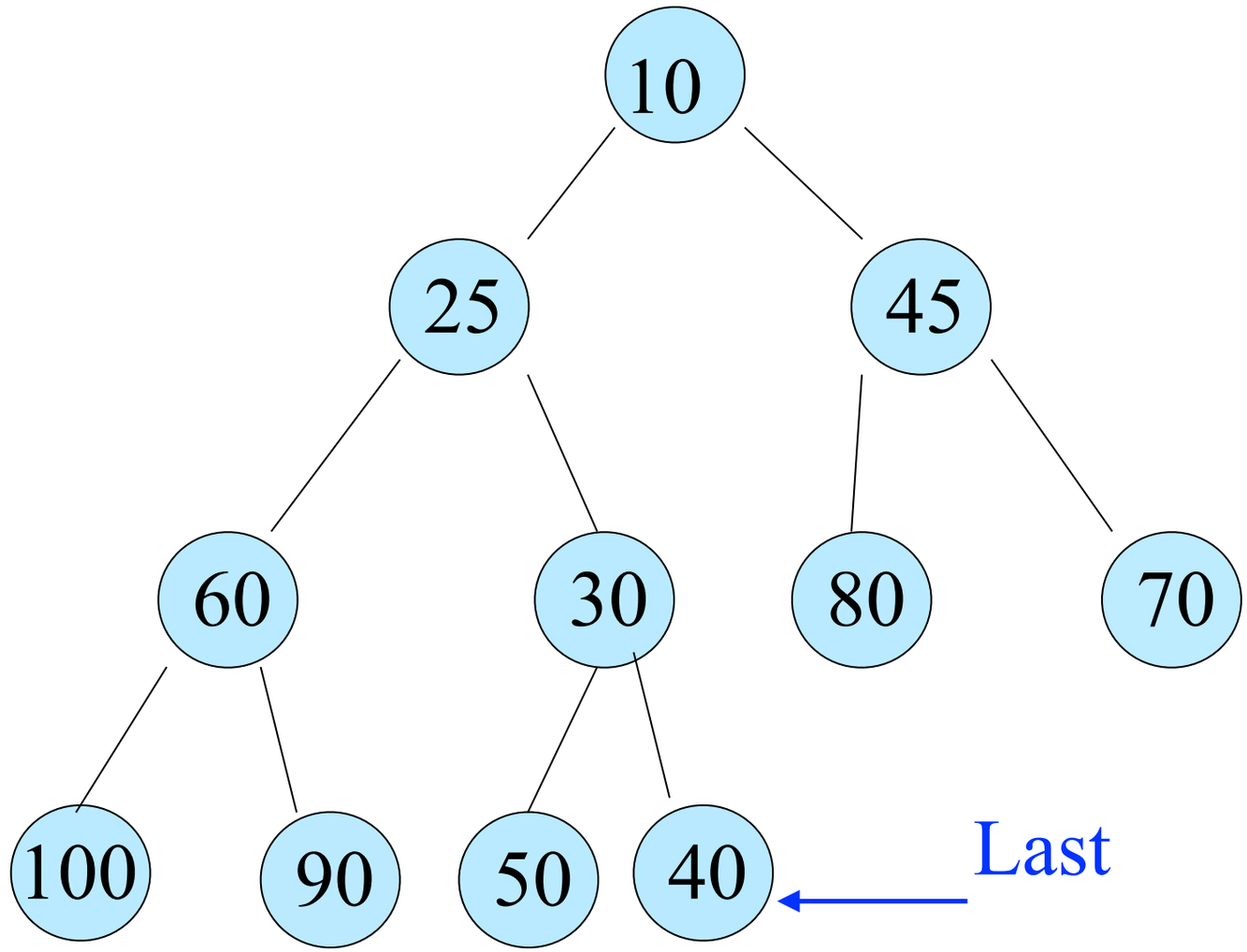
Back to being a min binary heap!

Heap-Extract-Min

```
HEAP-EXTRACT-MIN(A)
    if A.heap-size < 1
        error “heap underflow”
    min = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size -1
    MIN-HEAPIFY(A, 1)
    return min
```

Place last item in hole and percolate down

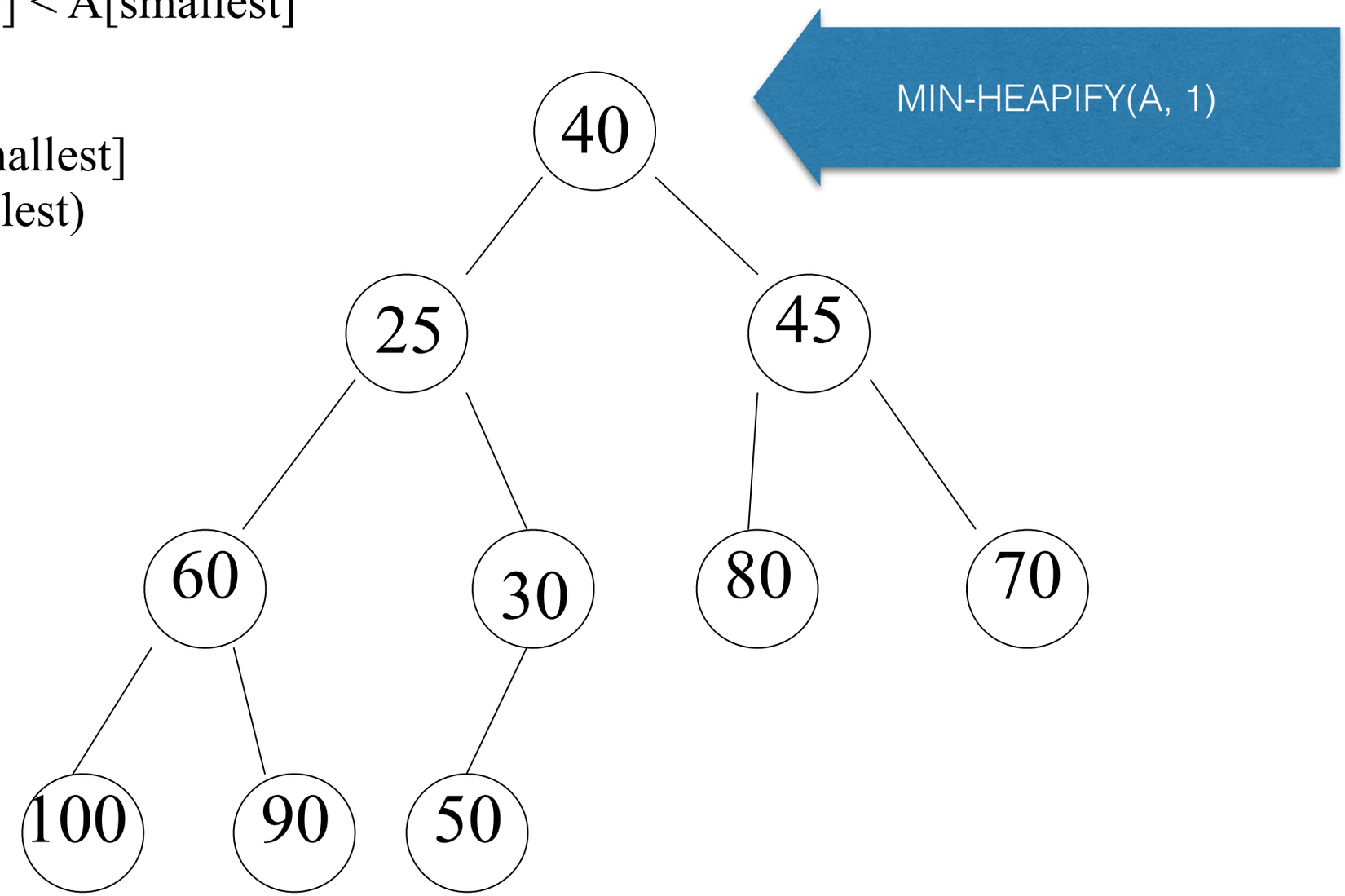
Delete Min



10	25	45	60	30	80	70	100	90	50	40
1	2	3	4	5	6	7	8	9	10	11

Percolate down by swapping with smaller child.

```
MIN-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap-size and A[l] < A[i]
        smallest = l
    else smallest = i
    if r <= A.heap-size and A[r] < A[smallest]
        smallest = r
    if  smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```



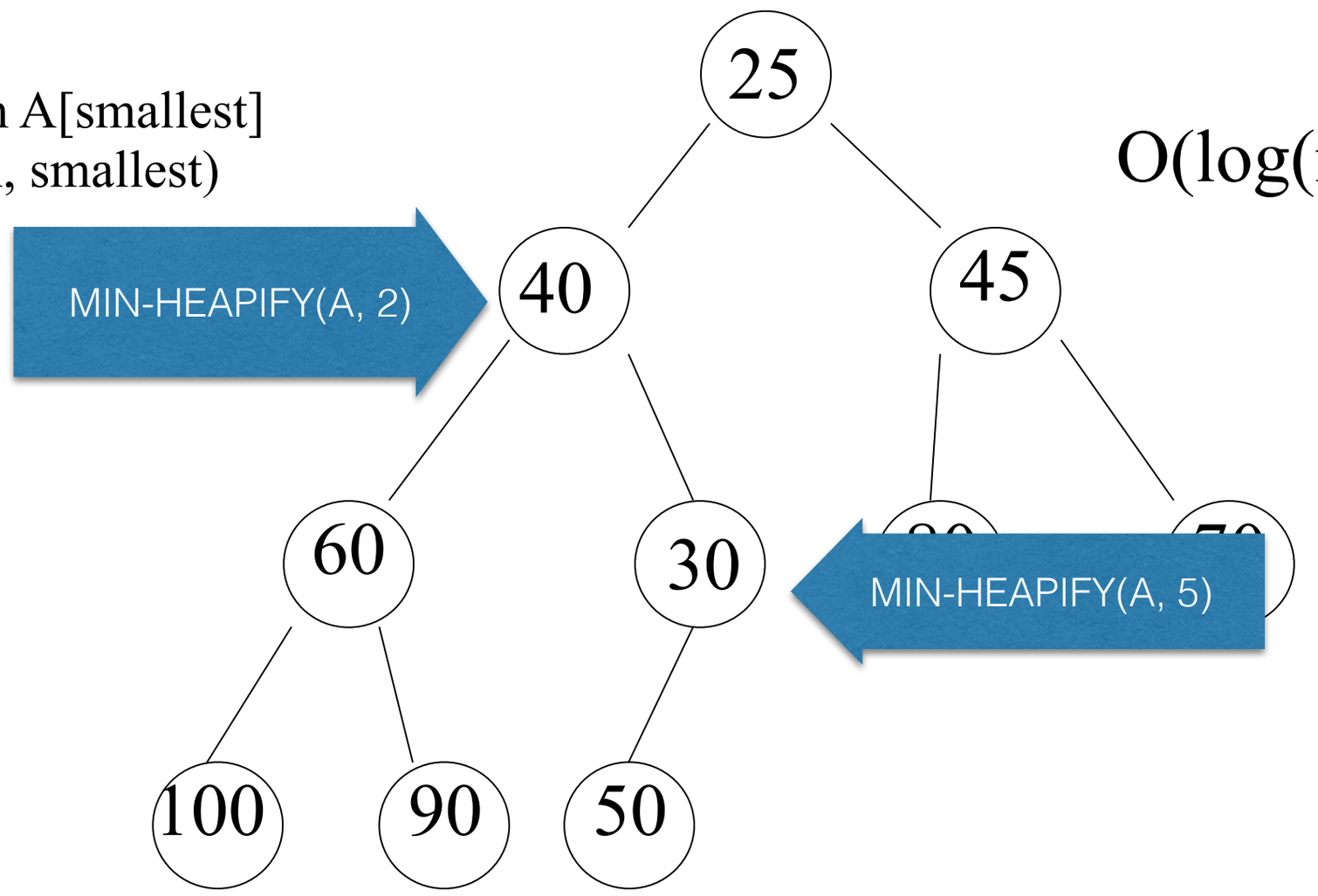
40	25	45	60	30	80	70	100	90	50
1	2	3	4	5	6	7	8	9	10

Percolate down by swapping with smaller child.

```
MIN-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap-size and A[l] < A[i]
        smallest = l
    else smallest = i
    if r <= A.heap-size and A[r] < A[smallest]
        smallest = r
    if  smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

Time?

O(log(n))

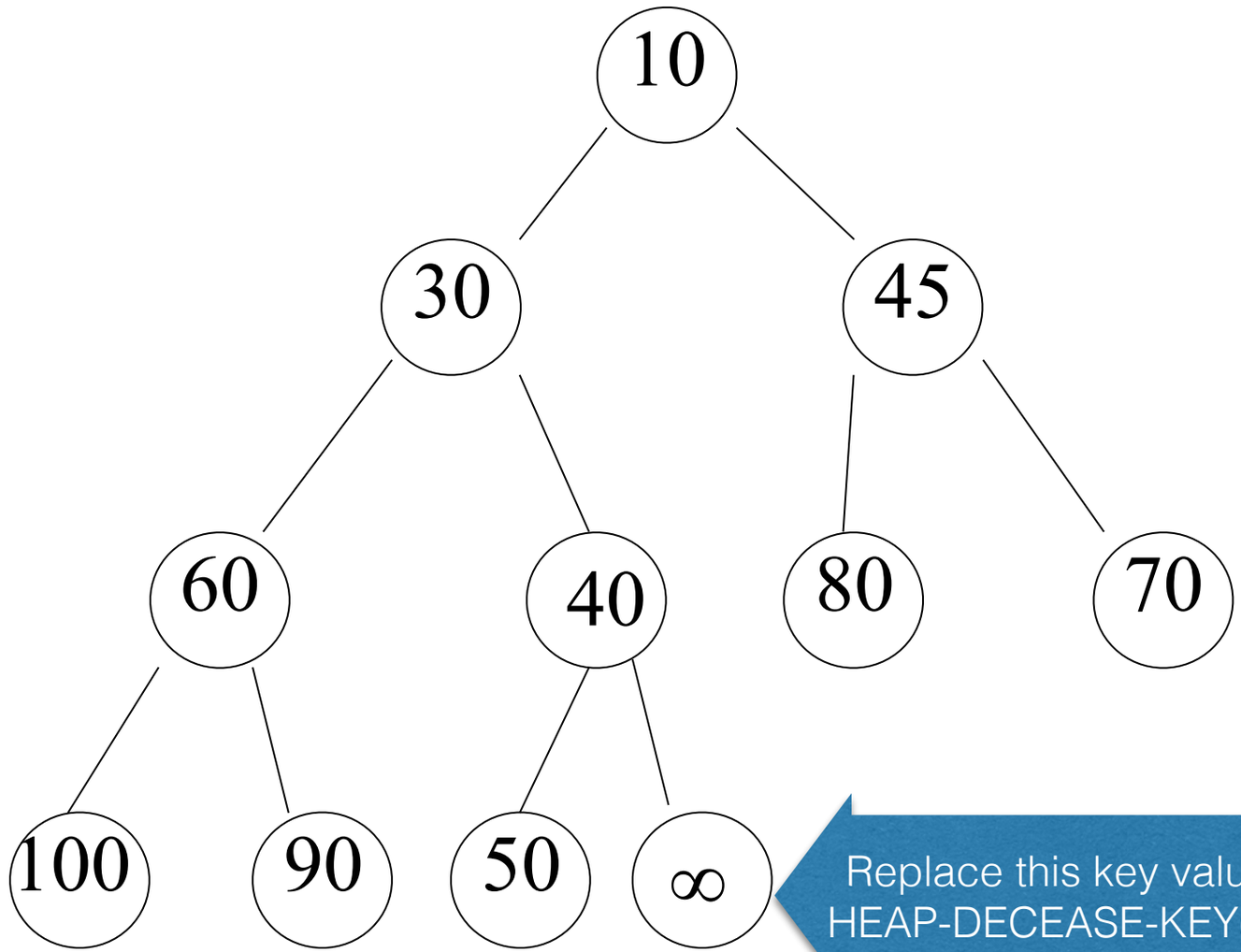


25	40	45	60	30	80	70	100	90	50
1	2	3	4	5	6	7	8	9	10

Heap-Decrease-Key

```
HEAP-DECREASE-KEY(A, i, key)
    if key > A[i]
        error "new key is larger"
    A[i] = key
    while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

Percolate up



10	30	45	60	40	80	70	100	90	50	∞
1	2	3	4	5	6	7	8	9	10	11

Min-Heap-Insert

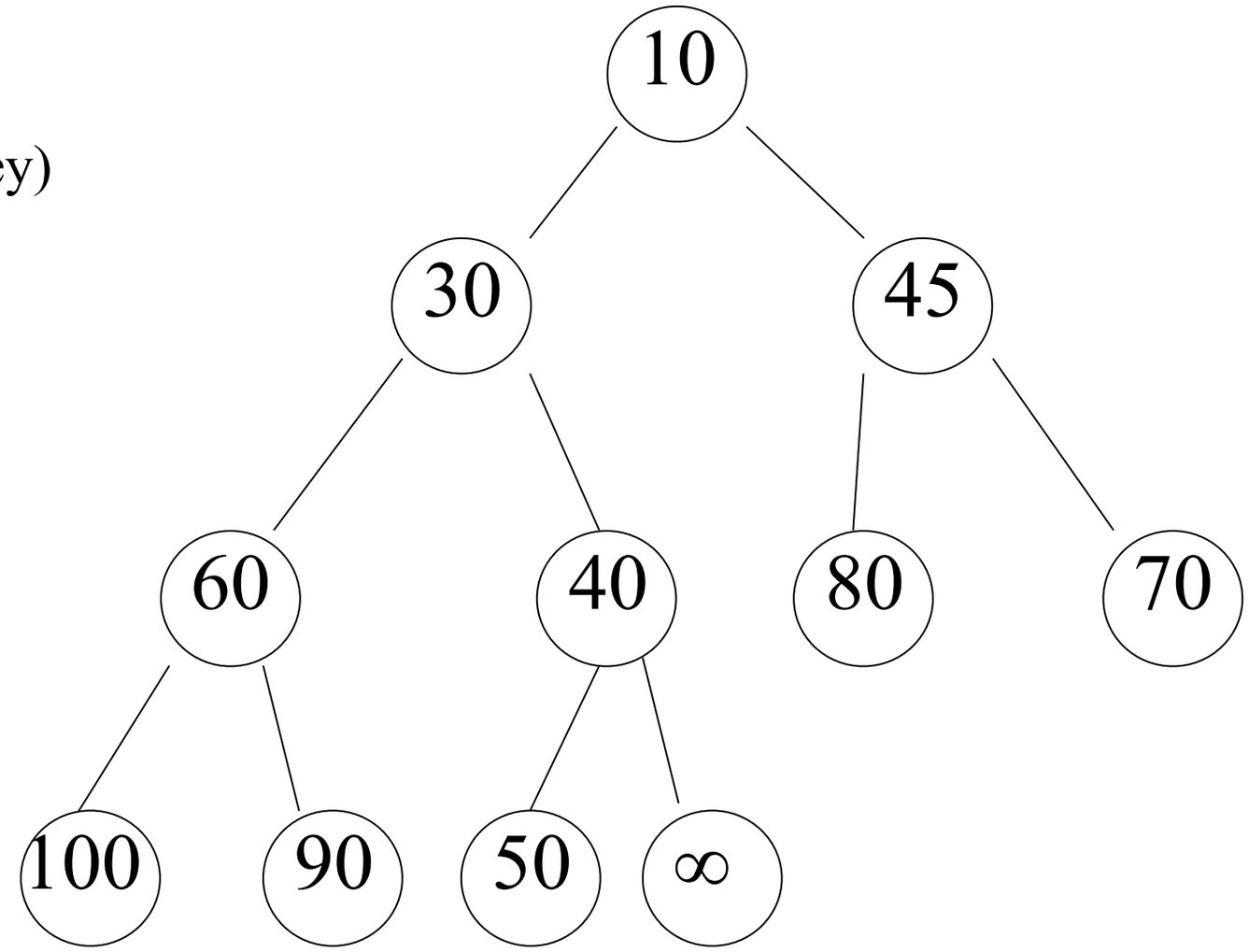
HEAP-DECREASE-KEY(A,11,25)

MIN-HEAP-INSERT(A, key)

A.heap-size = A.heap-size + 1

A[A.heap-size] = ∞

HEAP-DECREASE-KEY(A, A.heap-size, key)



10	30	45	60	40	80	70	100	90	50	∞
1	2	3	4	5	6	7	8	9	10	11

HEAP-DECREASE-KEY(A, i, key)

if key > A[i]

error “new key is larger”

A[i] = key

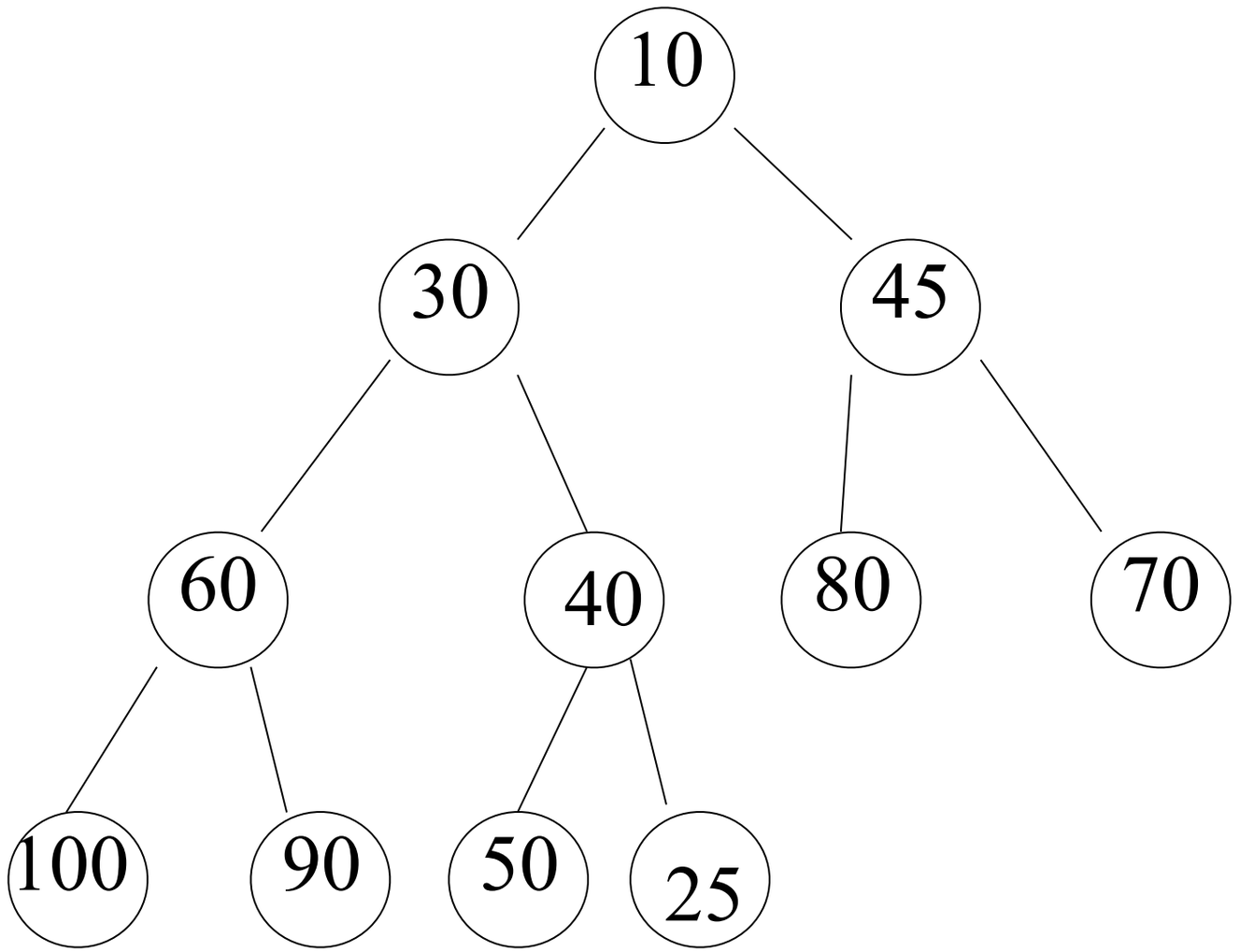
while i > 1 **and** A[PARENT(i)] > A[i]

exchange A[i] with A[PARENT(i)]

i = PARENT(i)

Percolate up

HEAP-DECREASE-KEY(A,11,25)



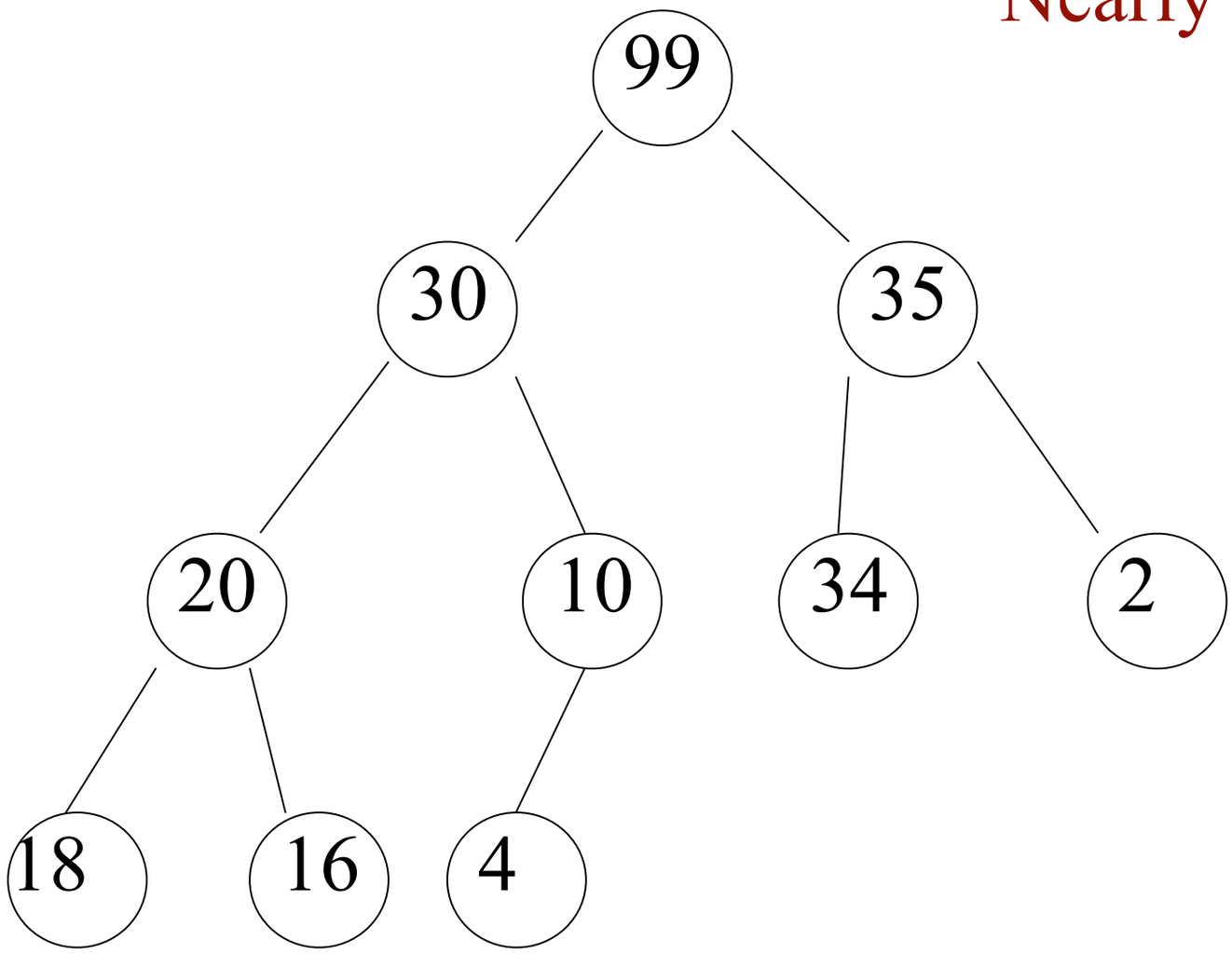
10	30	45	60	40	80	70	100	90	50	25
1	2	3	4	5	6	7	8	9	10	11

Time for insertion? O(log(n)) Random input average 2.607 comparisons!

A Max Heap Example

Max Heap

Nearly Complete Tree

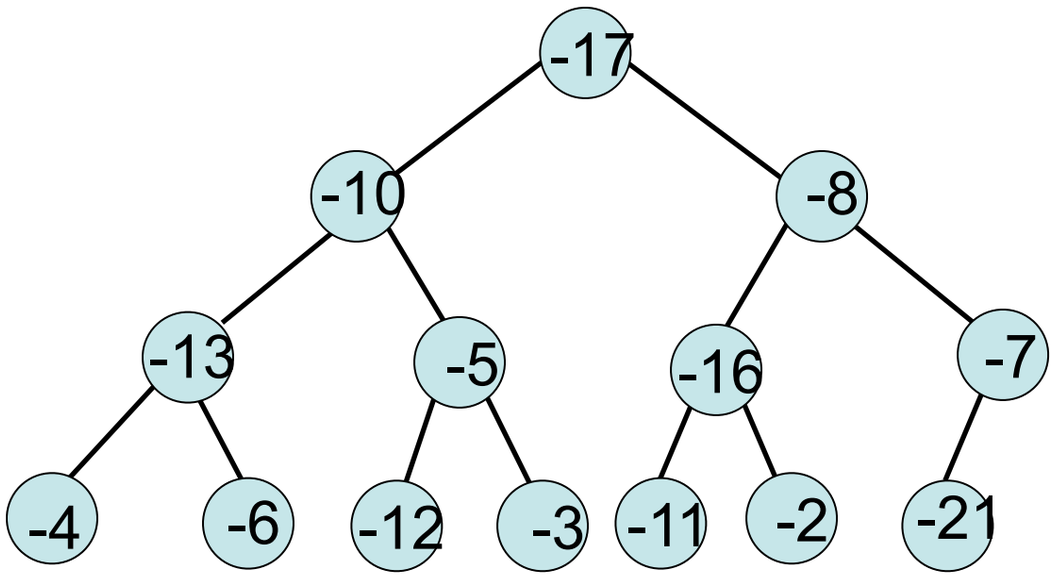


How can we turn A into a max heap?

A

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-17	-10	-8	-13	-5	-16	-7	-4	-6	-12	-3	-11	-2	-21

How can we turn A into a max heap?



Where to start?

How long does it take?

A

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-17	-10	-8	-13	-5	-16	-7	-4	-6	-12	-3	-11	-2	-21

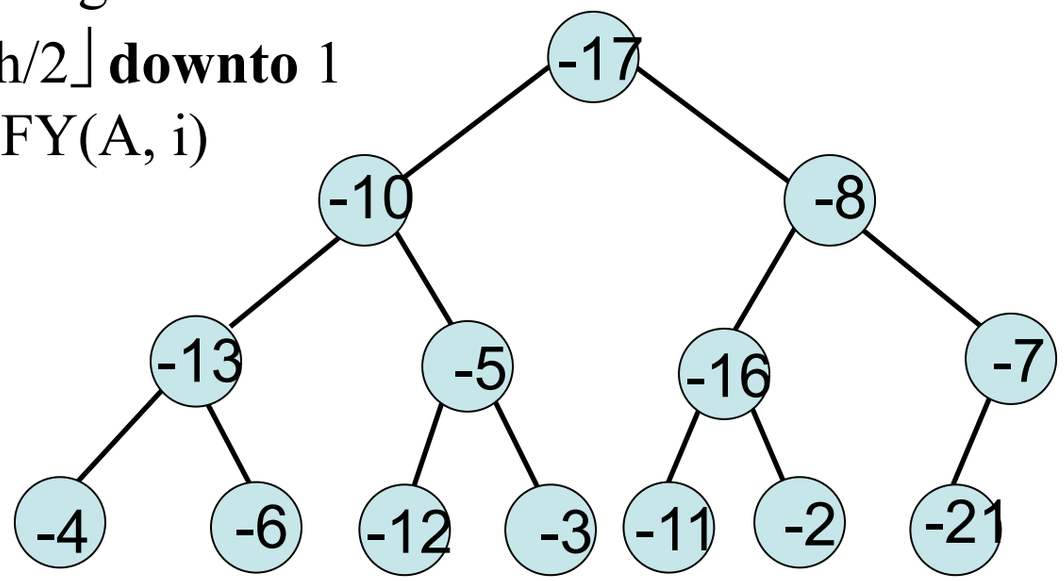
Building a max-heap

BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



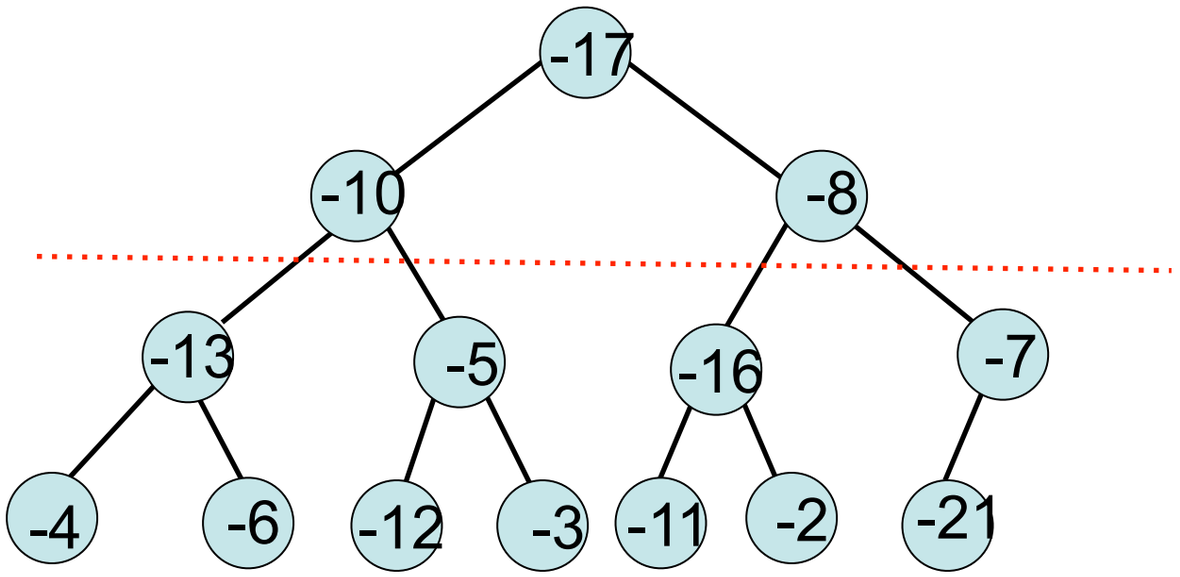
1	2	3	4	5	6	7	8	9	10	11	12	13	14
-17	-10	-8	-13	-5	-16	-7	-4	-6	-12	-3	-11	-2	-21

BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



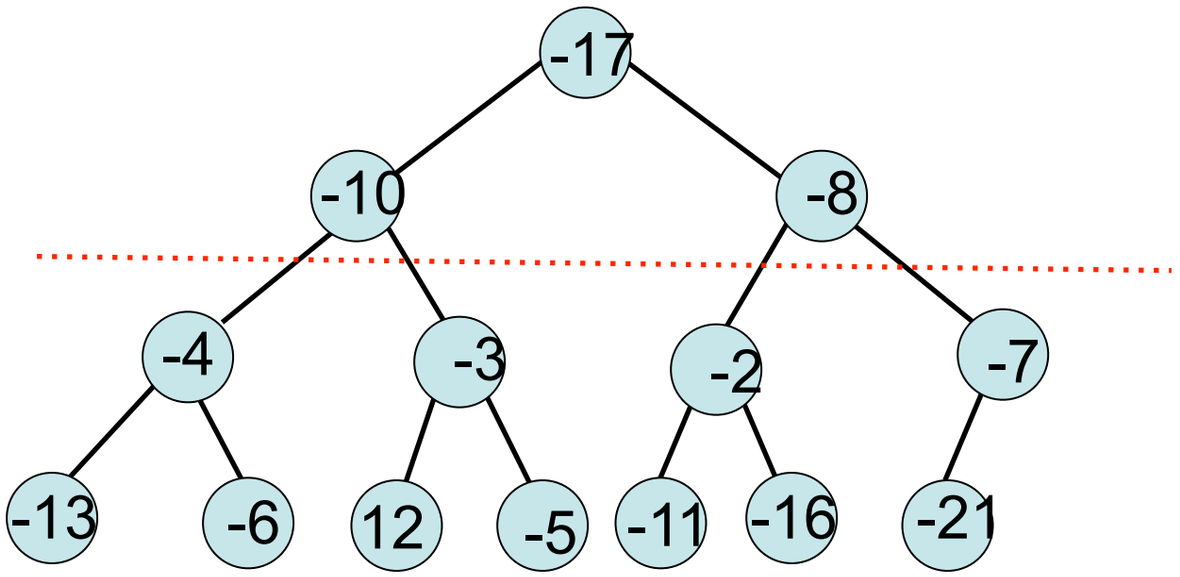
`percolateDown` all trees below the dotted line

BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



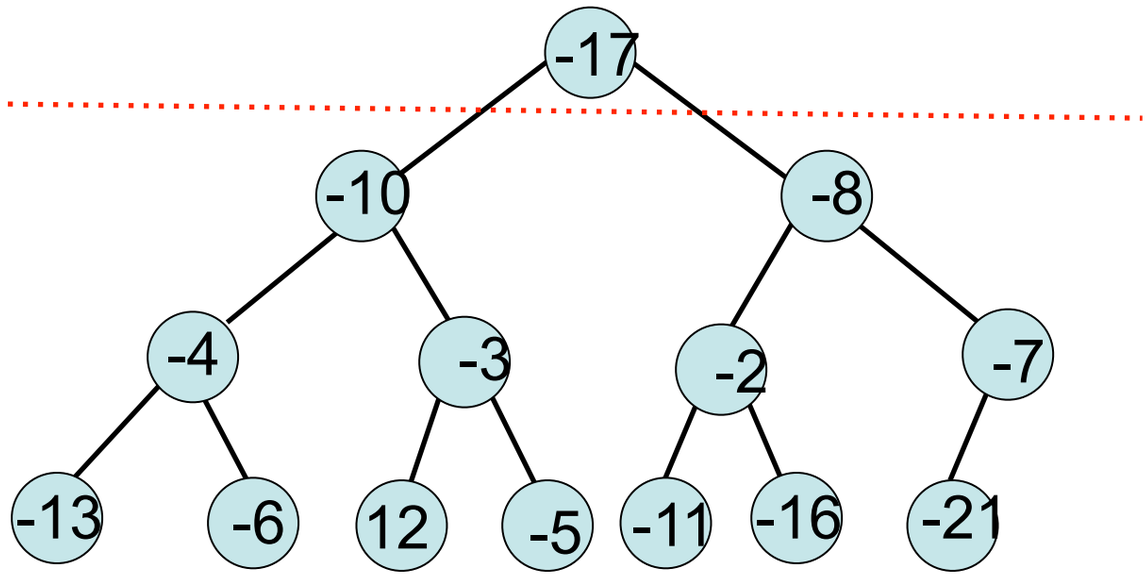
Done percolating down all trees below the dotted line

BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



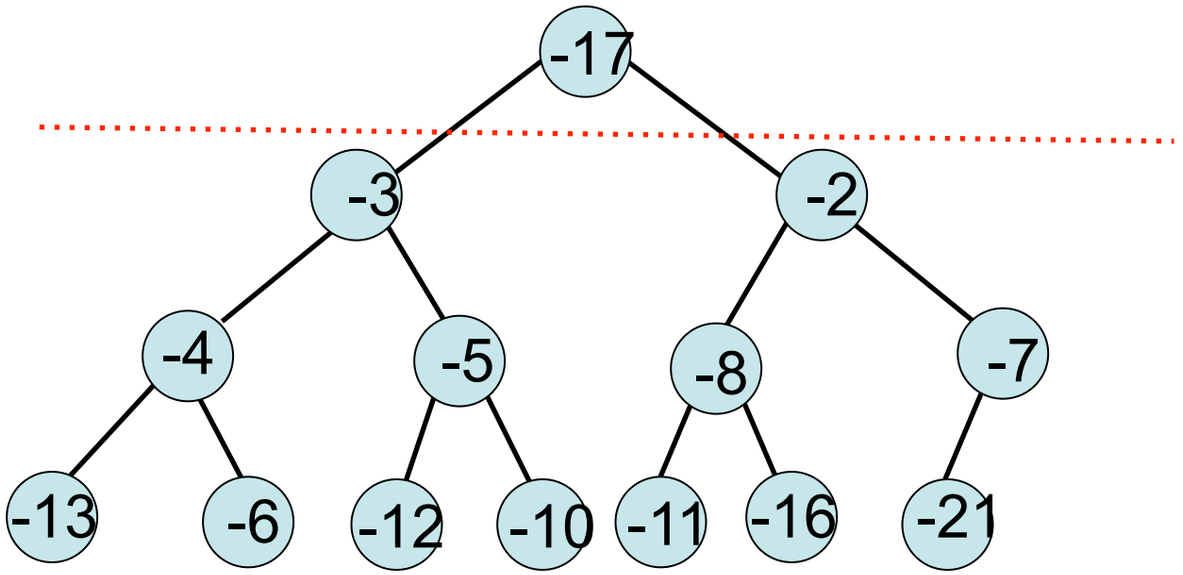
percolating down all trees below the dotted line

BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



Done percolating down all trees below the dotted line

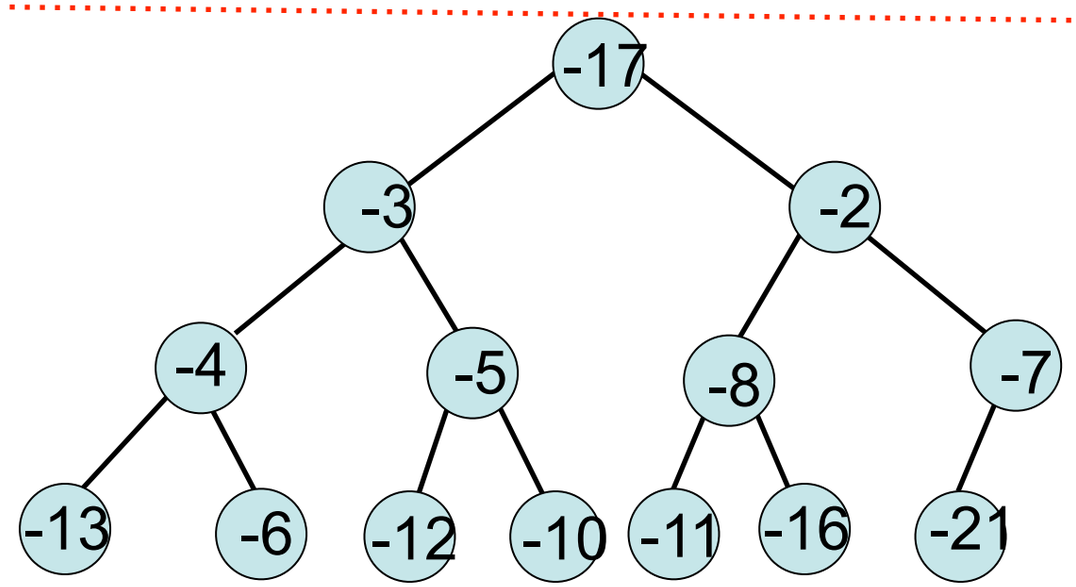
BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⌊A.length/2⌋ **downto** 1

MAX-HEAPIFY(A, i)

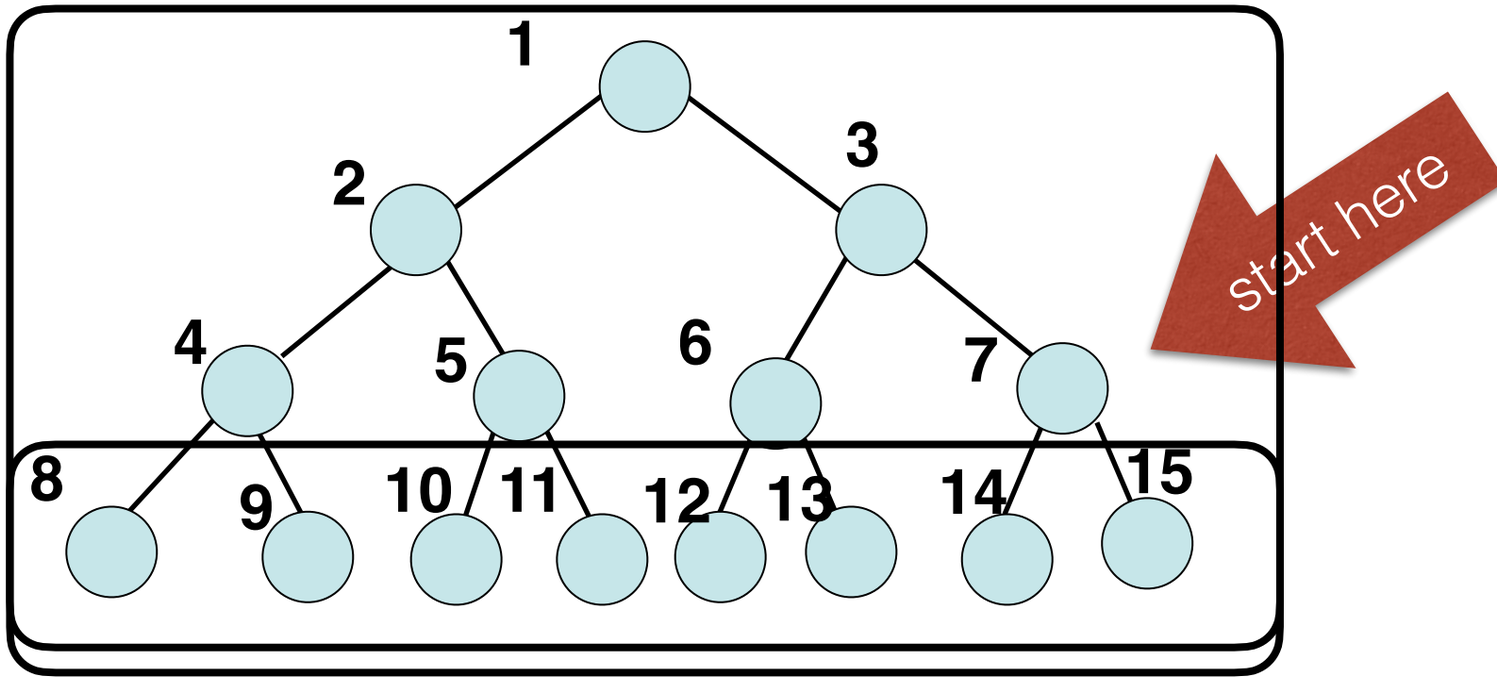
`percolateDown` all trees below the dotted line



Can we prove BUILD-MAX-HEAP works?

BUILD-MAX-HEAP(A)

A.heap-size = A.length
for i = ⌊A.length/2⌋ **downto** 1
MAX-HEAPIFY(A, i)



Loop Invariant: At the start of each iteration of the for loop, each node i + 1, i +2, …, n is the root of a max-heap.

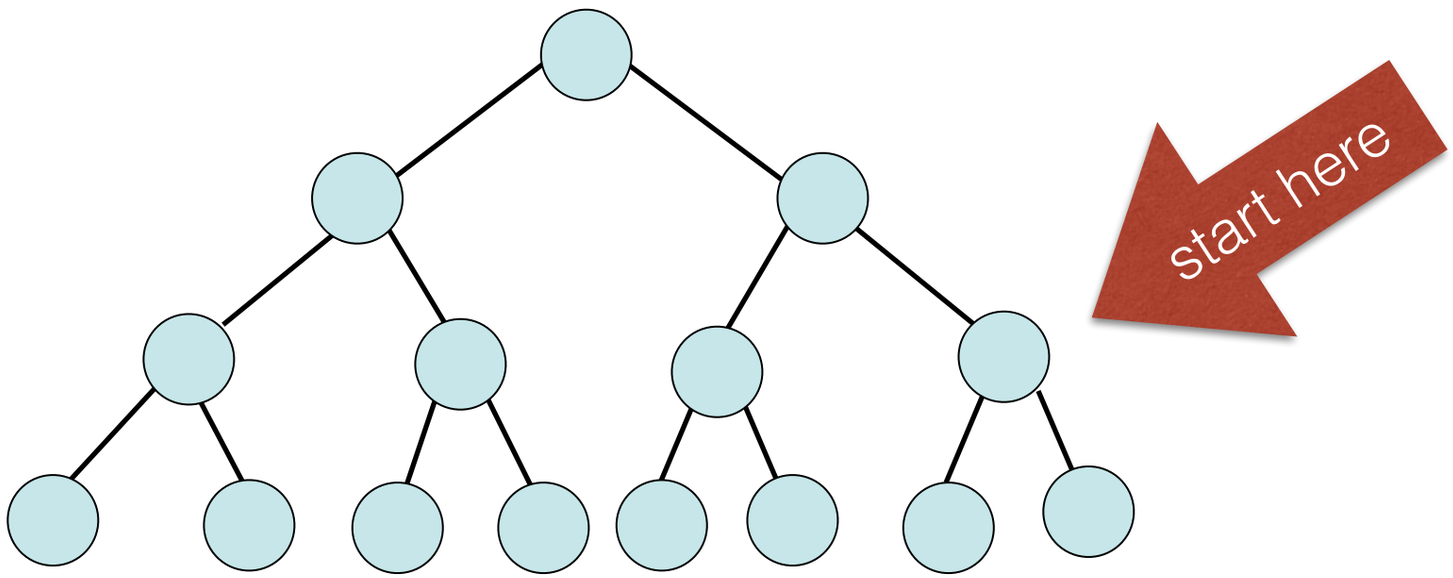
Initialization: Each node ⌊n/2⌋ +1, ⌊n/2⌋ +2, …,n is a leaf, which is the root of a max-heap. Since i = ⌊n/2⌋ before the first iteration of the for loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both max-heaps. Correctly assuming that i+1, i+2, …, n are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root.

Decrementing i re-establishes the loop invariant at each iteration.

Termination: When the loop terminates, i = 0. By the loop invariant, each node, notably node 1, is the root of a max-heap.

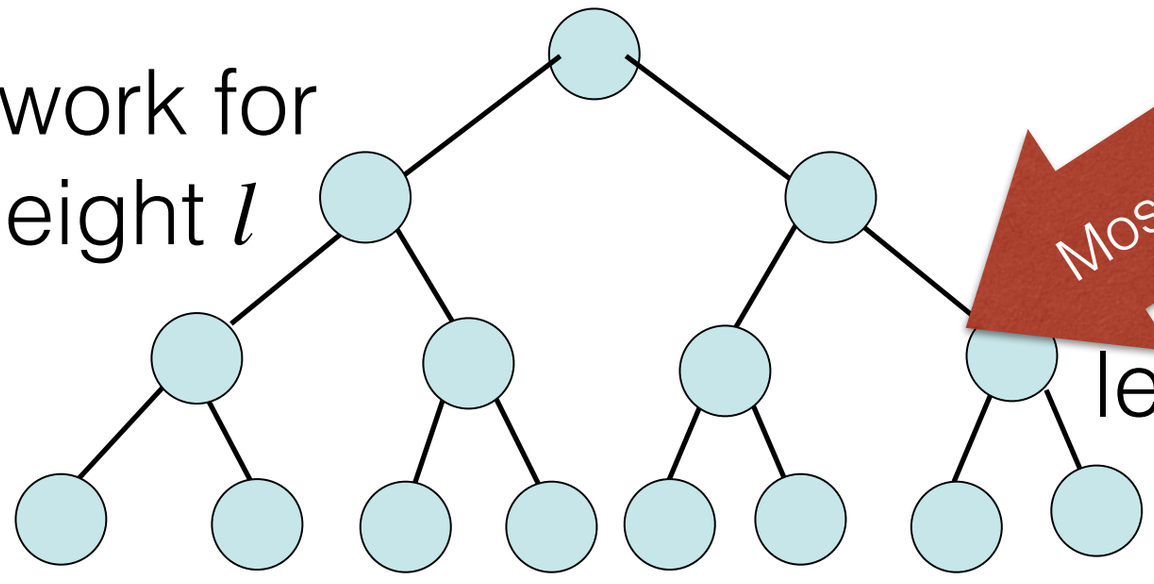
How long does BUILD-MAX-HEAP take?



Does it really take that long?

- Each percolate down takes O(log n) time. We call percolate down ~n/2 times. Therefore we know BUILD-MAX-HEAP(A) takes O(n log n) time.
- Our typical analysis is that if we do x operations that take time at most y, then the running time is O(x·y)
- Did we overestimate too much?

O(l) amount of work for any node at height l



each node on this level takes a constant amount of work

"**Amortized analysis:** Determine worst-case running time of a sequence of data structure operations as a function of the input size."

"**Aggregate method:** Sum up sequence of operations, weighted by their cost."

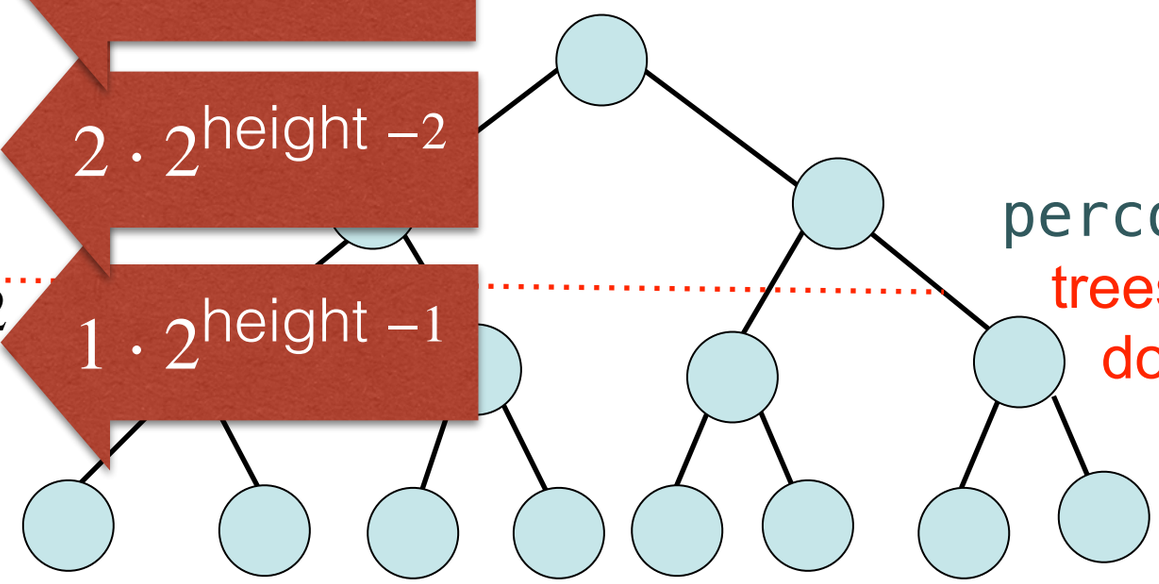
How many operations?

height	depth	max # nodes
3	0	$1 = 2^0$
2	1	$2 = 2^1$
1	2	$4 = 2^2$
0	3	$7 \leq 2^3$

$$c \cdot 3 \cdot 2^0 \quad \text{3} \cdot 2^{\text{height} - 3}$$

$$+c \cdot 2 \cdot 2^1 \quad \text{2} \cdot 2^{\text{height} - 2}$$

$$+c \cdot 1 \cdot 2^2 \quad \text{1} \cdot 2^{\text{height} - 1}$$



`percolateDown` all trees below the dotted line

$$T(n) \leq c \cdot 1 \cdot 2^2 + c \cdot 2 \cdot 2^1 + c \cdot 3 \cdot 2^0$$

$$T(n) \leq c \sum_{i=1}^{h} 2^{h-i} \cdot i$$

$i =$ node height

$h =$ height of tree

$h =$ height of tree

How many operations?

node height	*depth*	max *# nodes*	
h	0	$1 = 2^0$	$c \cdot h \cdot 2^0$
h-1	1	$2 = 2^1$	$c \cdot (h-1) \cdot 2^1$
h-2	2	$4 = 2^2$	$c \cdot (h-2) \cdot 2^2$
h-3	3	$8 = 2^3$	$c \cdot (h-3) \cdot 2^3$
h-i	i	2^i	$c \cdot (h-i) \cdot 2^i$
1	h-1	2^{h-1}	$c \cdot (1) \cdot 2^{h-1}$

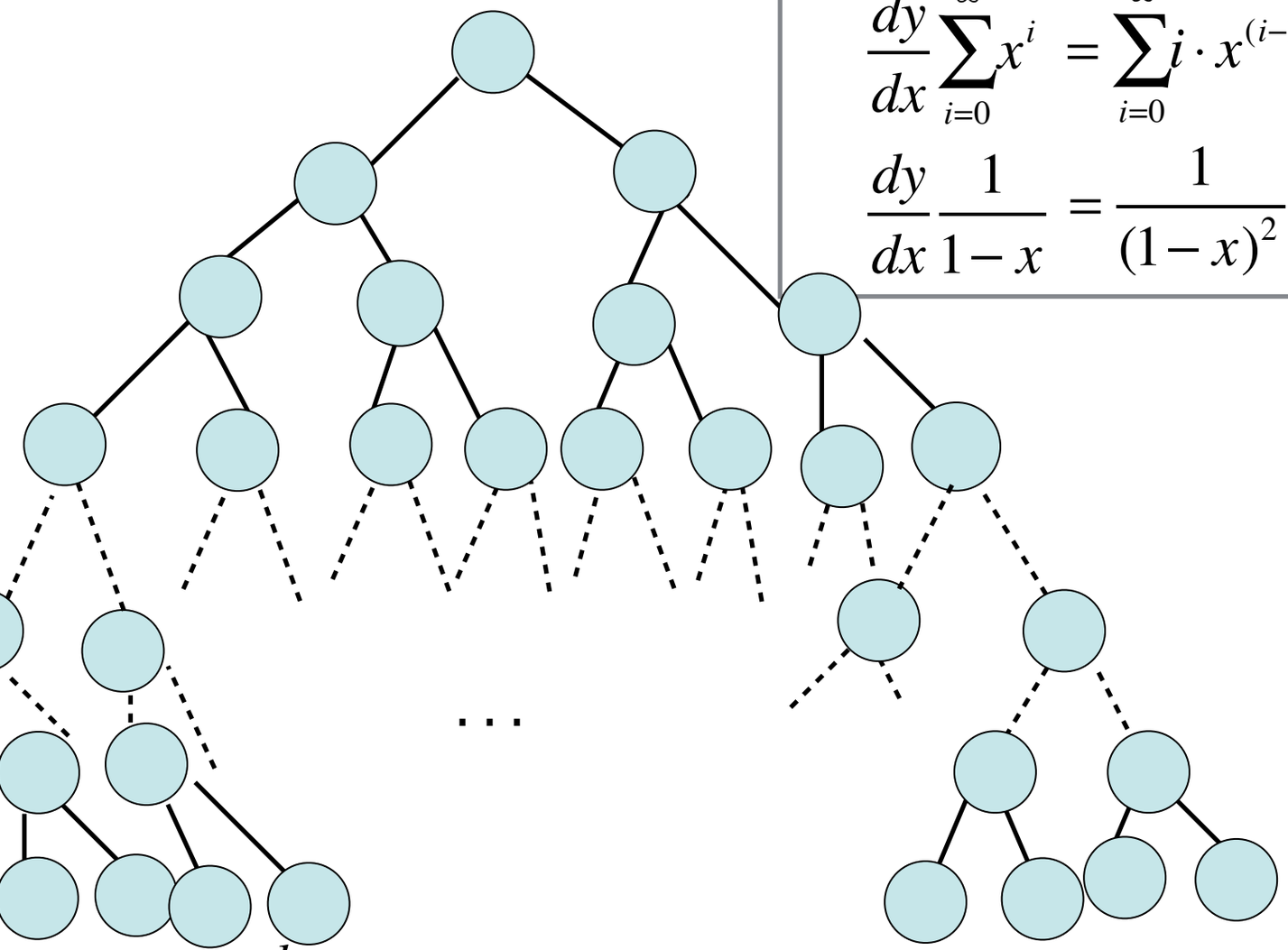
$i = node\ height$

$|x| < 1 \rightarrow$ workspace

let $x = \frac{1}{2}$ $\quad \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

$$\frac{dy}{dx} \sum_{i=0}^{\infty} x^i = \sum_{i=0}^{\infty} i \cdot x^{(i-1)}$$

$$\frac{dy}{dx} \frac{1}{1-x} = \frac{1}{(1-x)^2}$$



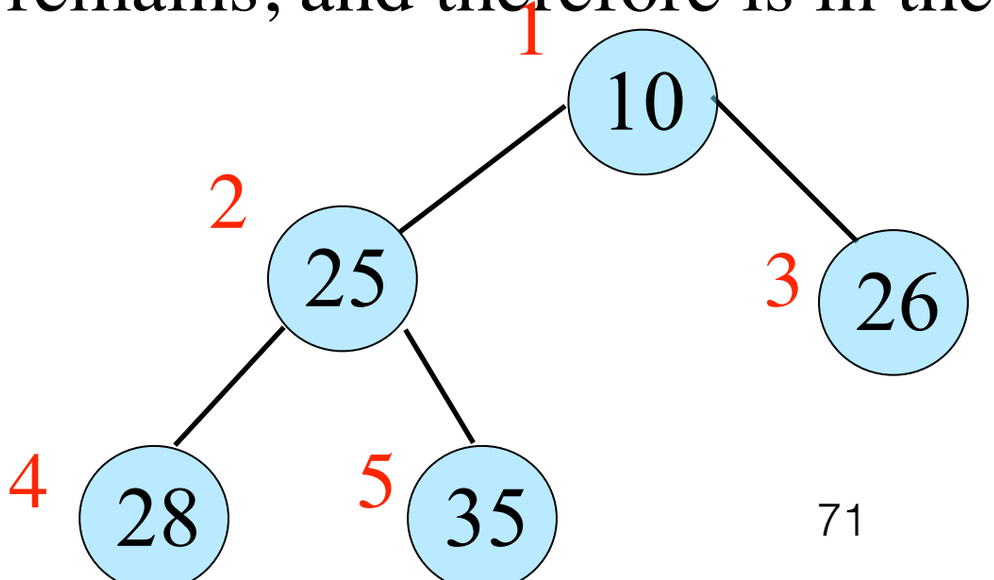
$$T(n) \le c \sum_{i=1}^{h} 2^{h-i} \cdot i \quad \le c 2^{\log n} \sum_{i=1}^{h} 2^{-i} \cdot i \quad \le cn \sum_{i=1}^{\infty} 2^{-i} \cdot i$$

$$= cn \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i \cdot i = \frac{cn}{2} \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^{i-1} = \frac{cn}{2} \frac{1}{(1-1/2)^2} = O(n)$$

Heap Sort

Given an input array, the heapsort algorithm acts as follows:

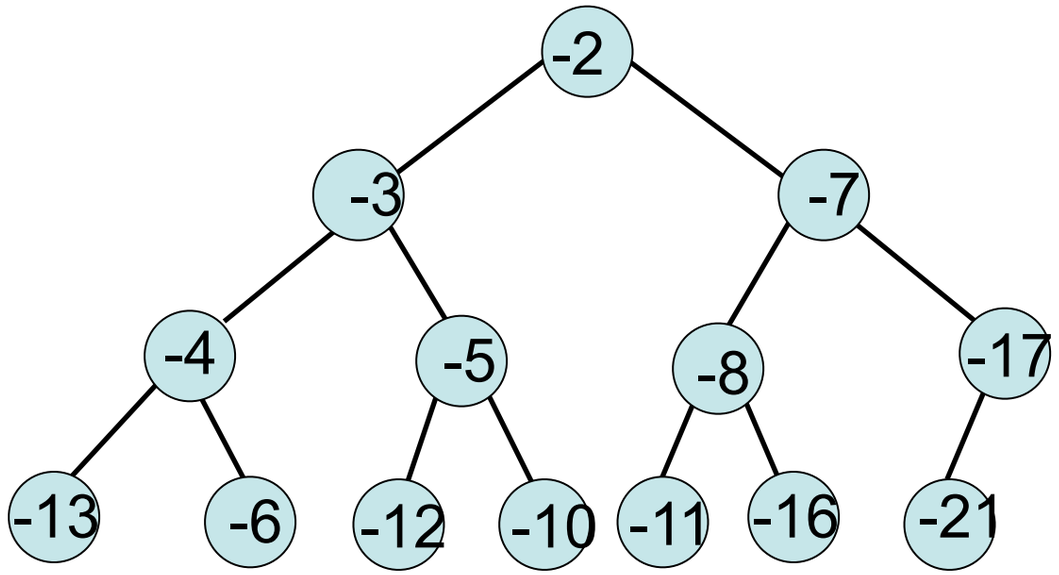
- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.



HeapSort

HEAPSORT(A)

BUILD-MAX-HEAP(A)
for i = A.length **downto** 2
exchange A[l] with A[i]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)



And so on… till all the items are sorted

Heap-Sort

O(n log n) worst case time!

sort in place!