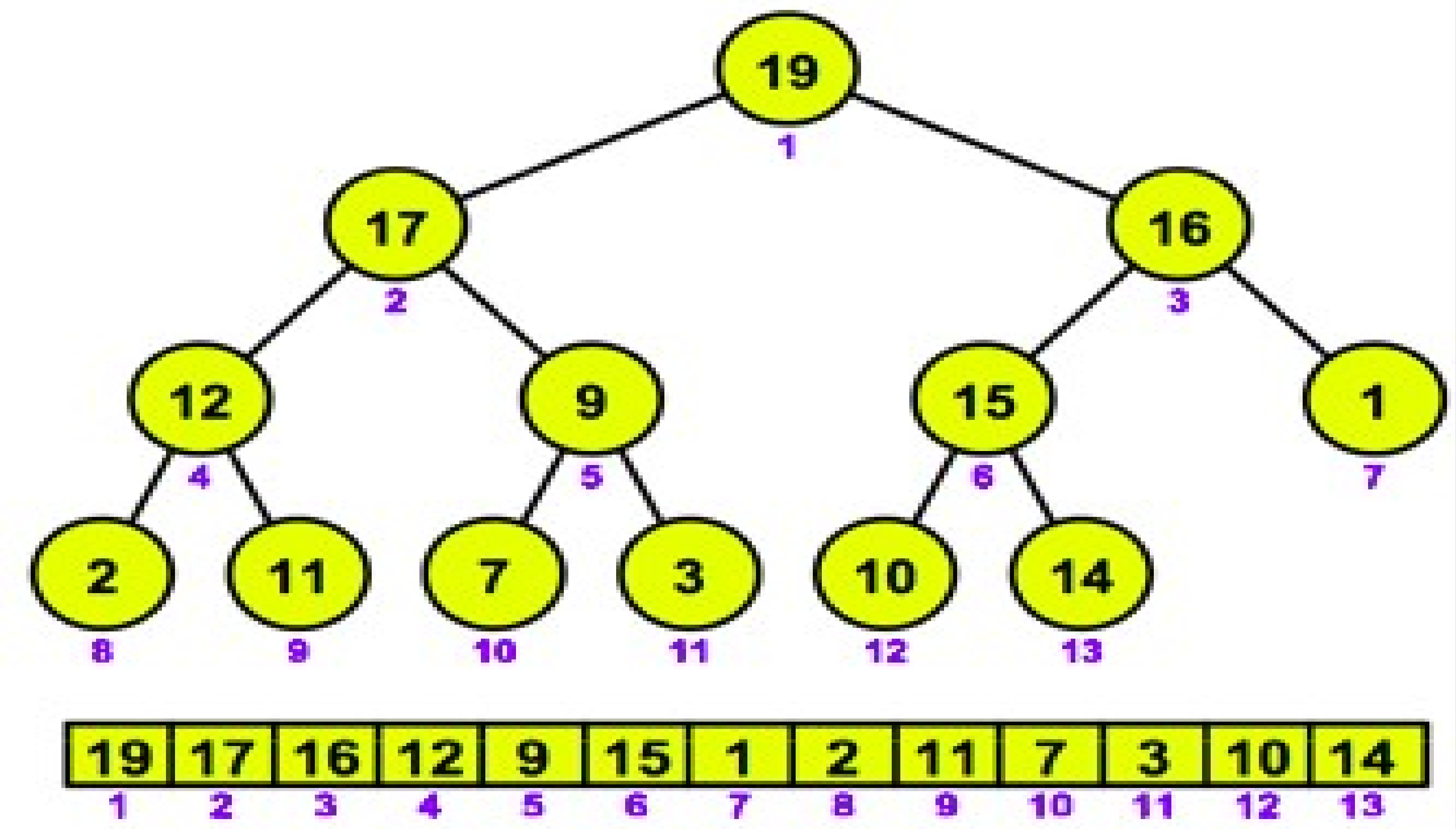
Heapsort


19
1
17
2
16
3
12
4
9
5
15
6
1
7
2
8
11
9
7
10
3
11
10
12
14
13
19 17 16 12 9 15 1 2 11 7 3 10 14
1 2 3 4 5 6 7 8 9 10 11 12 13

Announcements

Homework 1 posted,
due Sunday Oct. 1

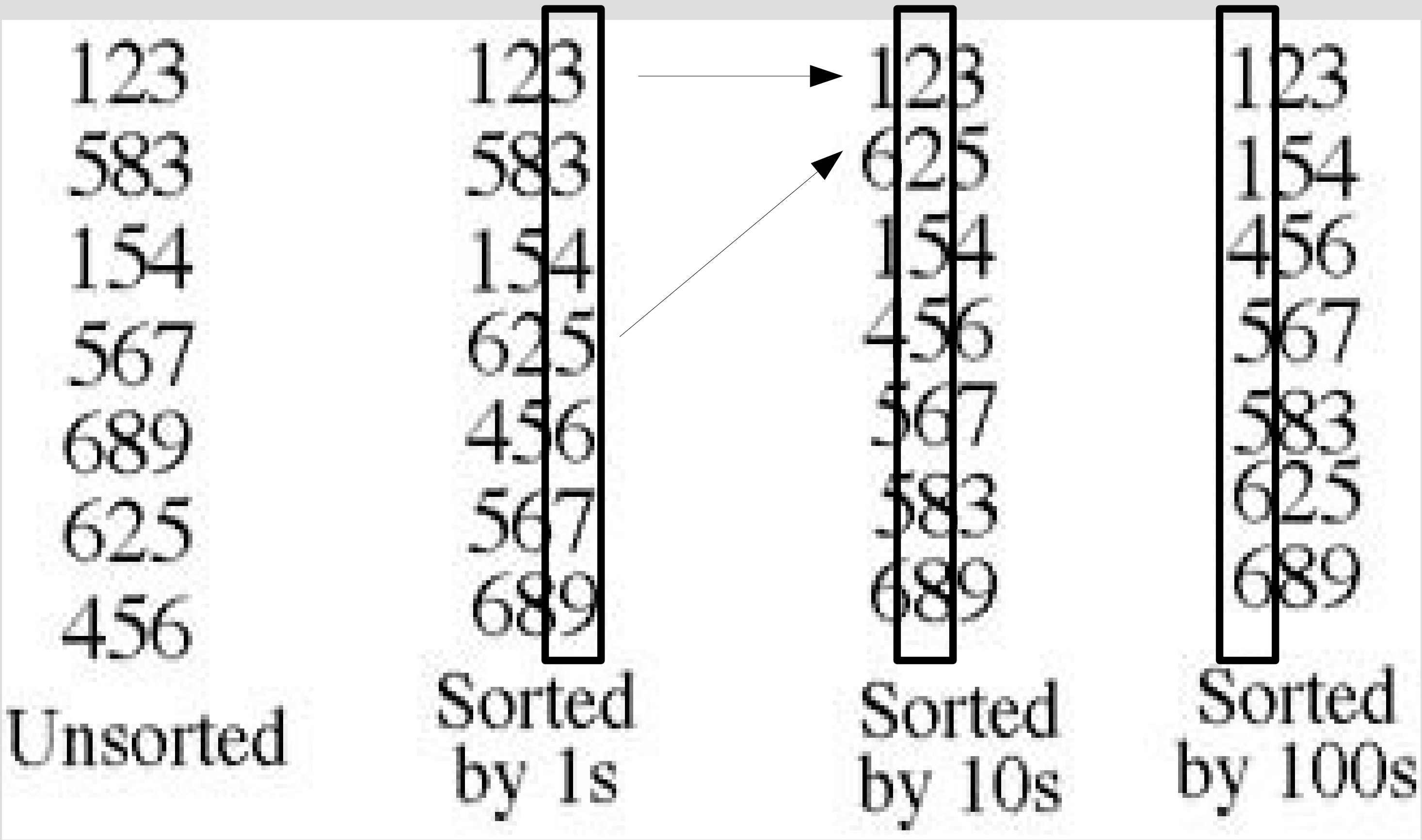
Radix sort

Use a **stable** sort to sort from the least significant digit to most

Psuedo code: (A=input)

```
for i = 1 to d
   stable sort of A on digit i
   // i.e. use counting sort
```

Radix sort



Stable means you can draw lines without crossing for a single digit

Radix sort

Run time?

Radix sort

Run time?

$O((b/r) (n+2^r))$

b-bits total, r bits per 'digit'

$d = b/r$ digits

Each count sort takes $O(n + 2^r)$

runs count sort d times...

$O(d(n+2^r)) = O(b/r (n + 2^r))$

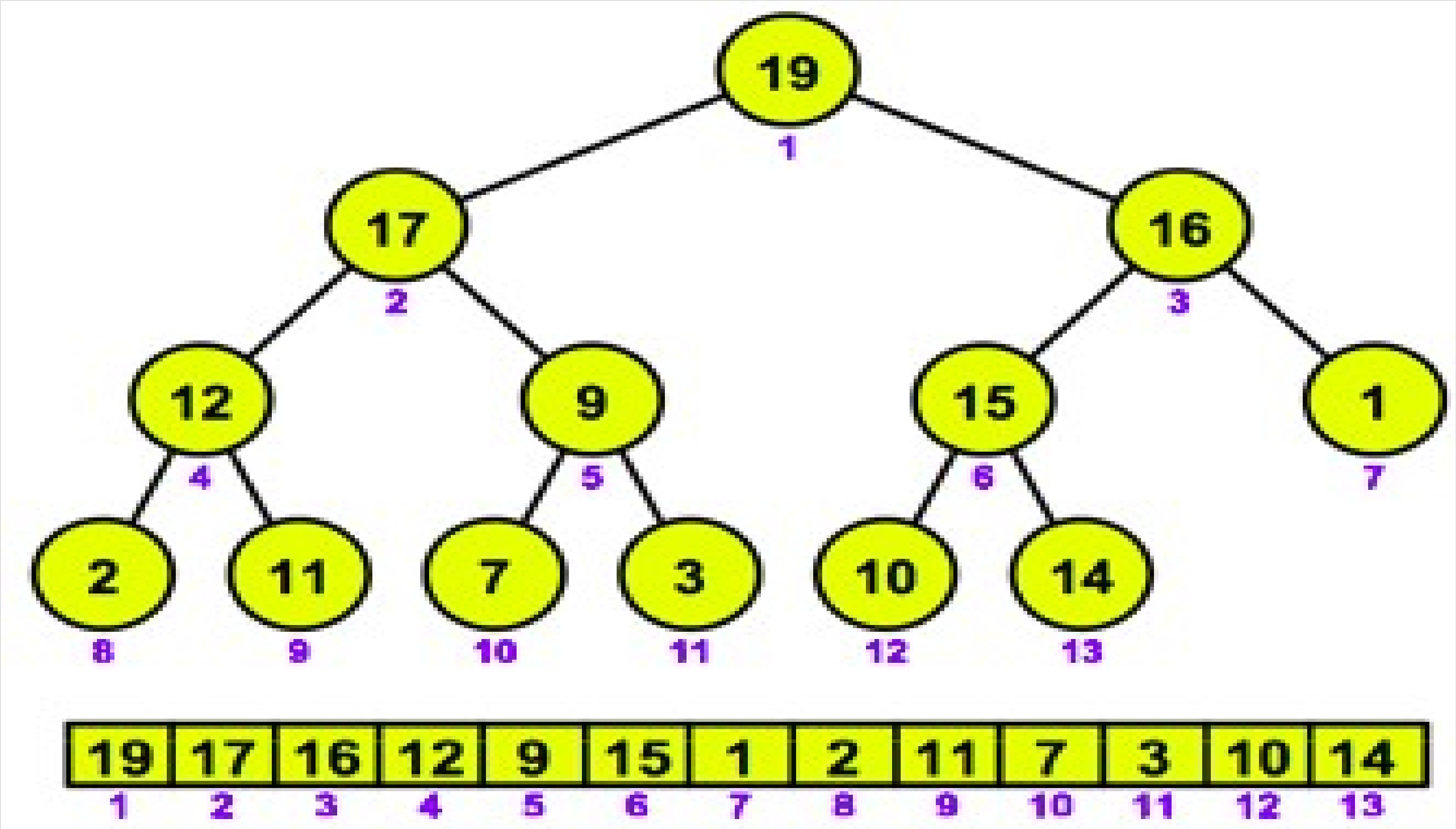
Radix sort

Run time?

if $b < \lg(n)$, $\Theta(n)$

if $b \geq \lg(n)$, $\Theta(n \lg n)$

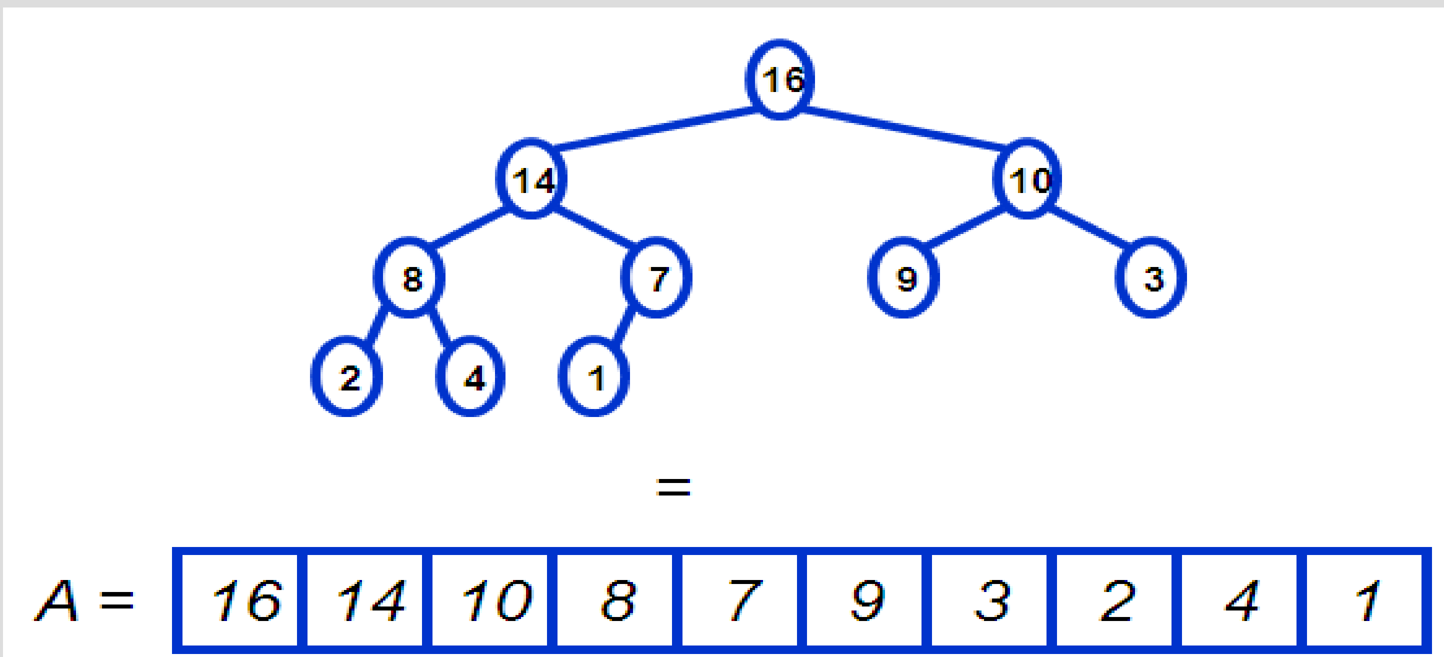
Heapsort



19
1
17
2
16
3
12
4
9
5
15
6
1
7
2
8
11
9
7
10
3
11
10
12
14
13
19 17 16 12 9 15 1 2 11 7 3 10 14
1 2 3 4 5 6 7 8 9 10 11 12 13

Binary tree as array

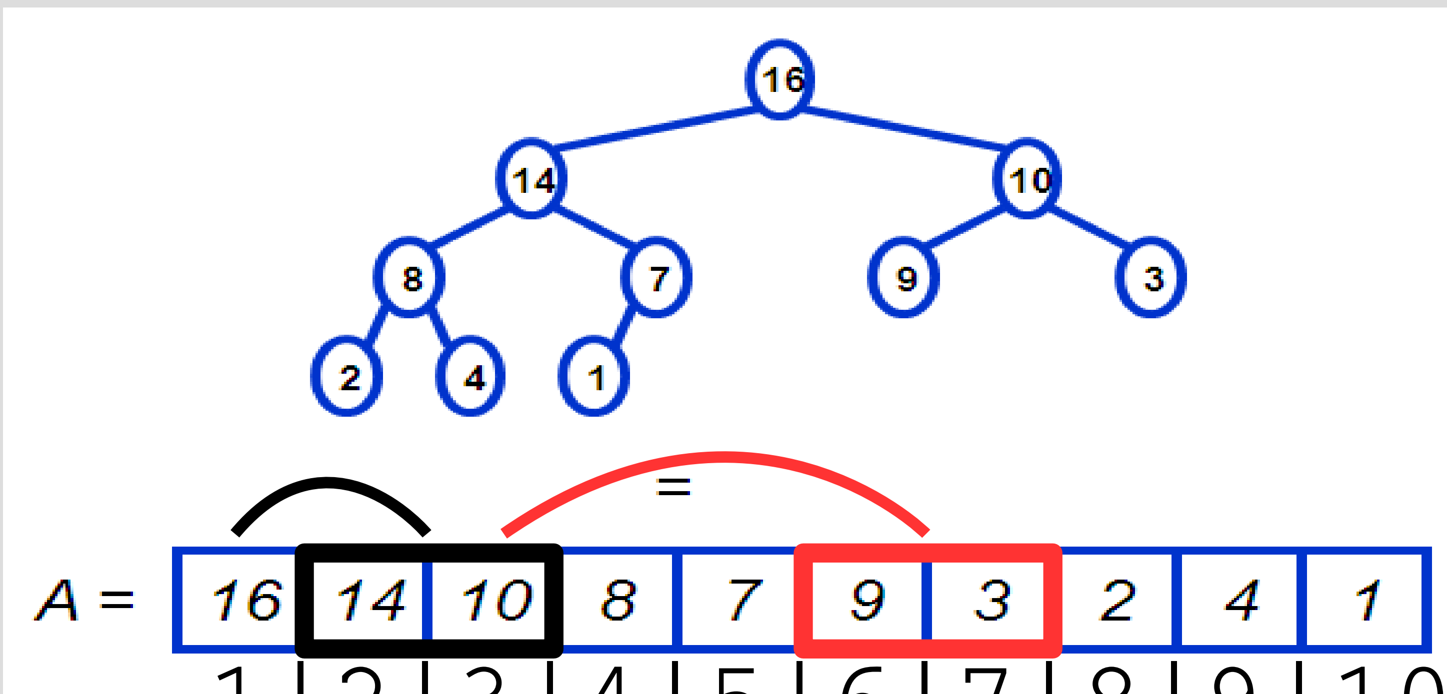
It is possible to represent binary trees as an array



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

Binary tree as array

index 'i' is the parent of '2i' and '2i+1'



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

Binary tree as array

Is it possible to represent any tree with a constant branching factor as an array?

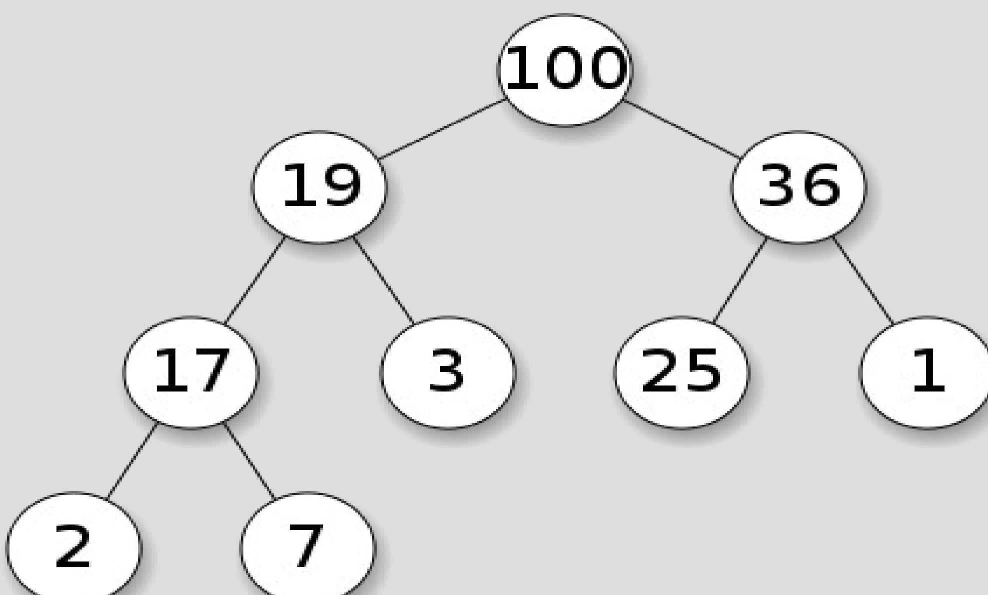
Binary tree as array

Is it possible to represent any tree with a constant branching factor as an array?

Yes, but the notation is awkward

Heaps

A <u>max heap</u> is a tree where the parent is larger than its children (A <u>min heap</u> is the opposite)



Heapsort

The idea behind heapsort is to:

1. Build a heap
2. Pull out the largest (root) and re-compile the heap
3. (repeat)

Heapsort

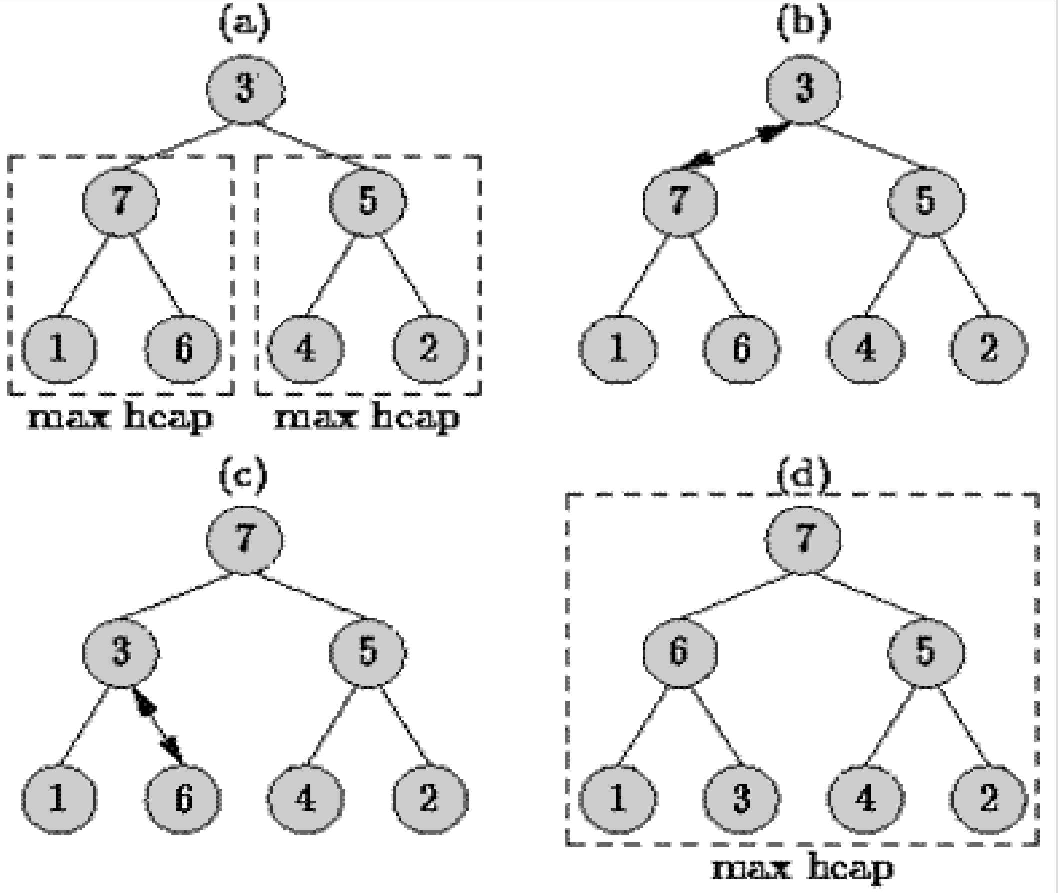
To do this, we will define subroutines:

1. Max-Heapify = maintains heap property

2. Build-Max-Heap = make sequence into a max-heap

Max-Heapify

Input: a root of two max-heaps
Output: a max-heap



Max-Heapify

```
Pseudocode Max-Heapify(A,i):
left = left(i)      // 2*i
right = right(i)  // 2*i+1
L = arg_max( A[left], A[right], A[ i ])
if (L not i)
   exchange A[ i ] with A[ L ]
   Max-Heapify(A, L)
// now make me do it!
```

Max-Heapify

Runtime?

Max-Heapify

Runtime?

Obviously (is it?): lg n

$T(n) = T(2/3\ n) + O(1)$ // why?

Or...

$T(n) = T(1/2\ n) + O(1)$

Master's theorem

Master's theorem: (proof 4.6)

For $a \geq 1$, $b \geq 1$, $T(n) = a\,T(n/b) + f(n)$

If f(n) is... (3 cases)

$O(n^c)$ for $c < \log_b a$, $T(n)$ is $\Theta(n^{\log_b a})$

$\Theta(n^{\log_b a})$, then $T(n)$ is $\Theta(n^{\log_b a} \lg n)$

$\Omega(n^c)$ for $c > \log_b a$, $T(n)$ is $\Theta(f(n))$

Max-Heapify

Runtime?

Obviously (is it?): lg n

$T(n) = T(2/3\ n) + O(1)$ // why?

Or...

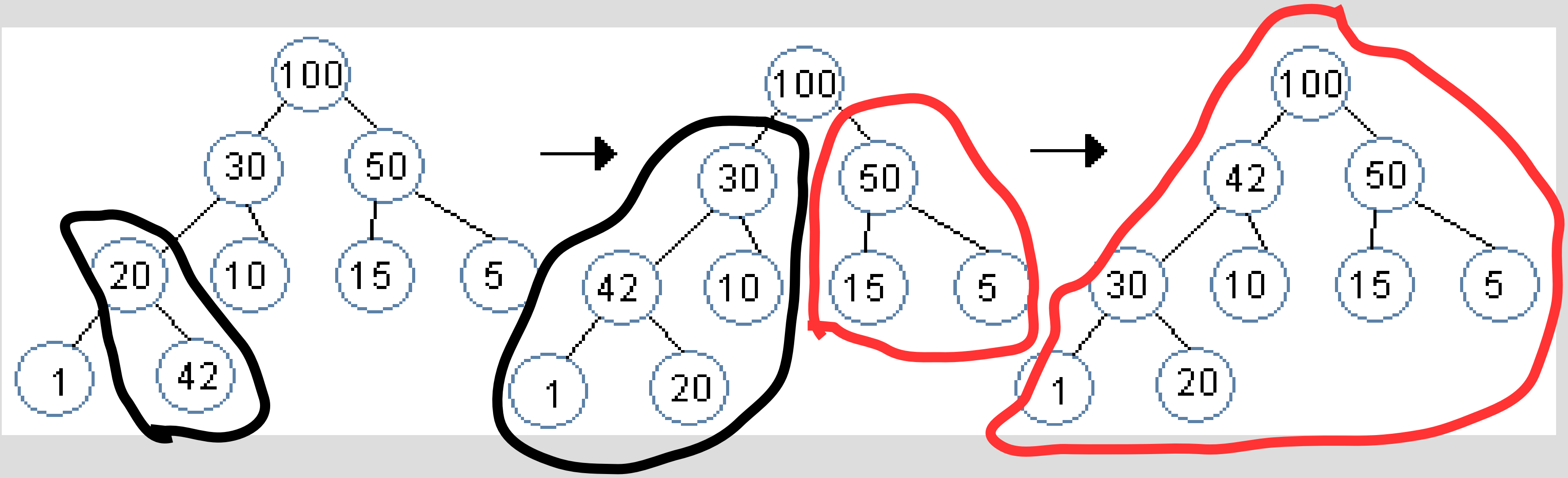
$T(n) = T(1/2\ n) + O(1) = O(\lg n)$

Build-Max-Heap

Next we build a full heap from an unsorted sequence

```
Build-Max-Heap(A)
for i = floor( A.length/2 ) to 1
   Heapify(A, i)
```

Build-Max-Heap



Red part is already Heapified

Build-Max-Heap

Correctness:

Base: Each alone leaf is a max-heap

Step: if A[i] to A[n] are in a heap, then Heapify(A, i-1) will make i-1 a heap as well

Termination: loop ends at i=1, which is the root (so all heap)

Build-Max-Heap

Runtime?

Build-Max-Heap

Runtime?

O(n lg n) is obvious, but we can get a better bound…

Show ceiling($n/2^{h+1}$) nodes at any height 'h'

Build-Max-Heap

Heapify from height 'h' takes O(h)

$\text{sum}_{h=0}^{\lg n}$ ceiling($n/2^{h+1}$) O(h)

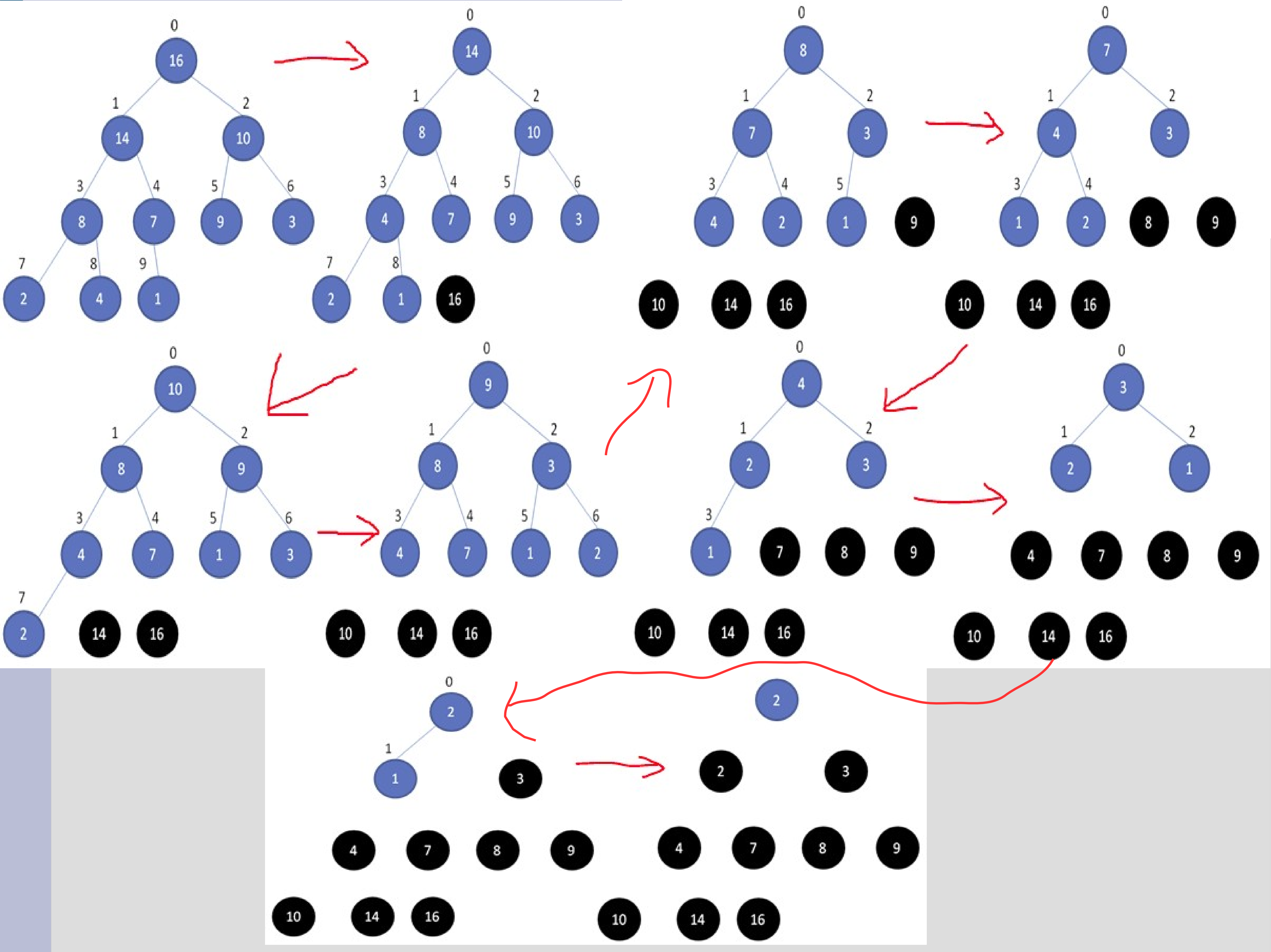
=O(n $\text{sum}_{h=0}^{\lg n}$ ceiling($h/2^{h+1}$))

($\text{sum}_{x=0}^{\infty}$ k x^k = $x/(1-x)^2$, x=1/2)

=O(n 4/2) = O(n)

Heapsort

```
Heapsort(A):
Build-Max-Heap(A)
for i = A.length to 2
   Swap A[ 1 ], A[ i ]
   A.heapsize = A.heapsize – 1
   Max-Heapify(A, 1)
```



Heapsort

Runtime?

Heapsort

Runtime?

Run Max-Heapify O(n) times

So... O(n lg n)

Priority queues

Heaps can also be used to implement priority queues (i.e. airplane boarding lines)

Operations supported are: Insert, Maximum, Exctract-Max and Increase-key

Priority queues

Maximum(A):

  return A[1]

Extract-Max(A):

max = A[1]

A[1] = A.heapsize

A.heapsize = A.heapsize – 1

Max-Heapify(A, 1),   return max

Priority queues

```
Increase-key(A, i, key):
A[ i ] = key
while ( i>1 and A [floor(i/2)] < A[i])
   swap A[ i ], A [floor(i/2)]
   i = floor(i/2)
```

Opposite of Max-Heapify... move high keys up instead of low down

Priority queues

Insert(A, key):

A.heapsize = A.heapsize + 1

A [A.heapsize] = -∞

Increase-key(A, A.heapsize, key)

Priority queues

Runtime?

Maximum =

Extract-Max =

Increase-Key =

Insert =

Priority queues

Runtime?

Maximum = O(1)
Extract-Max = O(lg n)
Increase-Key = O(lg n)
Insert = O(lg n)

Sorting comparisons:

Name	Average	Worst-case
Insertion[s,i]	$O(n^2)$	$O(n^2)$
Merge[s,p]	$O(n \lg n)$	$O(n \lg n)$
Heap[i]	$O(n \lg n)$	$O(n \lg n)$
Quick[p]	$O(n \lg n)$	$O(n^2)$
Counting[s]	$O(n + k)$	$O(n + k)$
Radix[s]	$O(d(n+k))$	$O(d(n+k))$
Bucket[s,p]	$O(n)$	$O(n^2)$

Sorting comparisons:

https://www.youtube.com/watch?v=kPRA0W1kECg