Durham University

# Algorithms & Data Structures

Part – 3: Topic 4

Anish Jindal
anish.jindal@durham.ac.uk

Durham University

# Heaps

# Priority queues

Collection of prioritized elements that allows arbitrary element insertion and removal having the first priority (highest or lowest).

Key-value pair

Operations:

**P.add(k, v):** Insert an item with key k and value v into priority queue P.

**P.min( ):** Return a tuple, (k,v), representing the key and value of an item in priority queue P with minimum key (but do not remove the item); an error occurs if the priority queue is empty.

**P.remove_min( ):** Remove an item with minimum key from priority queue P, and return a tuple, (k,v), representing the key and value of the removed item; an error occurs if the priority queue is empty.

**P.is_empty( ):** Return True if priority queue P does not contain any items.

**len(P):** Return the number of items in priority queue P.

| Operation | Return Value | Priority Queue |
|---|---|---|
| P.add(5,A) | | {(5,A)} |
| P.add(9,C) | | {(5,A), (9,C)} |
| P.add(3,B) | | {(3,B), (5,A), (9,C)} |
| P.add(7,D) | | {(3,B), (5,A), (7,D), (9,C)} |
| P.min( ) | (3,B) | {(3,B), (5,A), (7,D), (9,C)} |
| P.remove_min( ) | (3,B) | {(5,A), (7,D), (9,C)} |
| P.remove_min( ) | (5,A) | {(7,D), (9,C)} |
| len(P) | 2 | {(7,D), (9,C)} |
| P.remove_min( ) | (7,D) | {(9,C)} |
| P.remove_min( ) | (9,C) | { } |
| P.is_empty( ) | True | { } |
| P.remove_min( ) | "error" | { } |

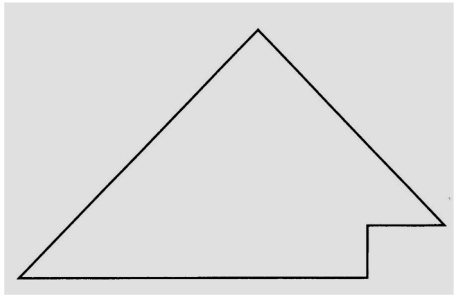
# (Binary) Heaps

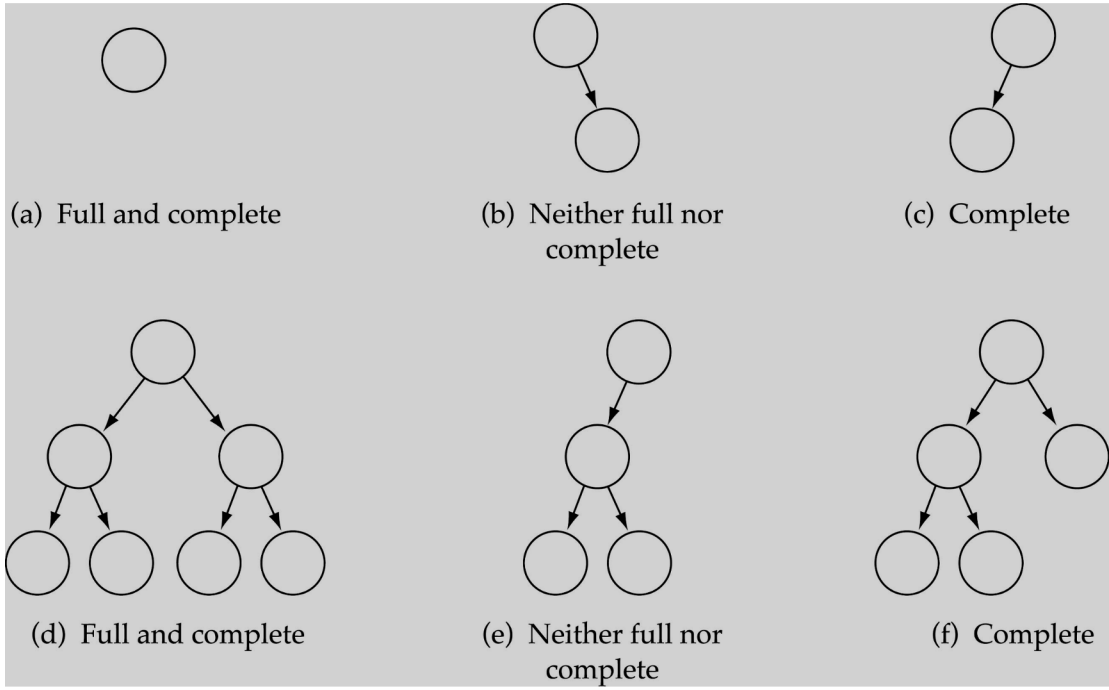
BSTs and family are “proper” tree structures

Heaps are trees as well, but typically assumed to be stored in a flat array

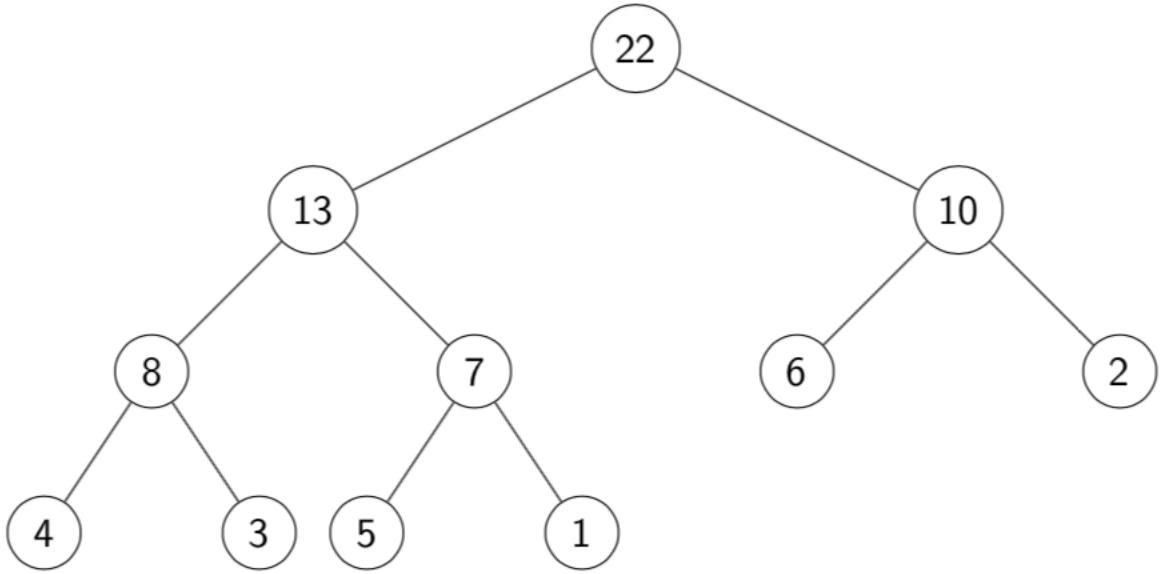
- Physically – linear array.
- Logically – binary tree, filled on all levels (except lowest.)

- each tree node corresponds to an element of the array
- the tree is complete except perhaps the lowest level, filled left-to-right





# Example




Very much **not** a BST!

# Types

**Max-Heap**

For every node excluding the root,
value is at most that of its parent: $A[parent[i]] \geq A[i]$

Largest element is stored at the root.

In any subtree, no values are larger than the value stored at subtree root.

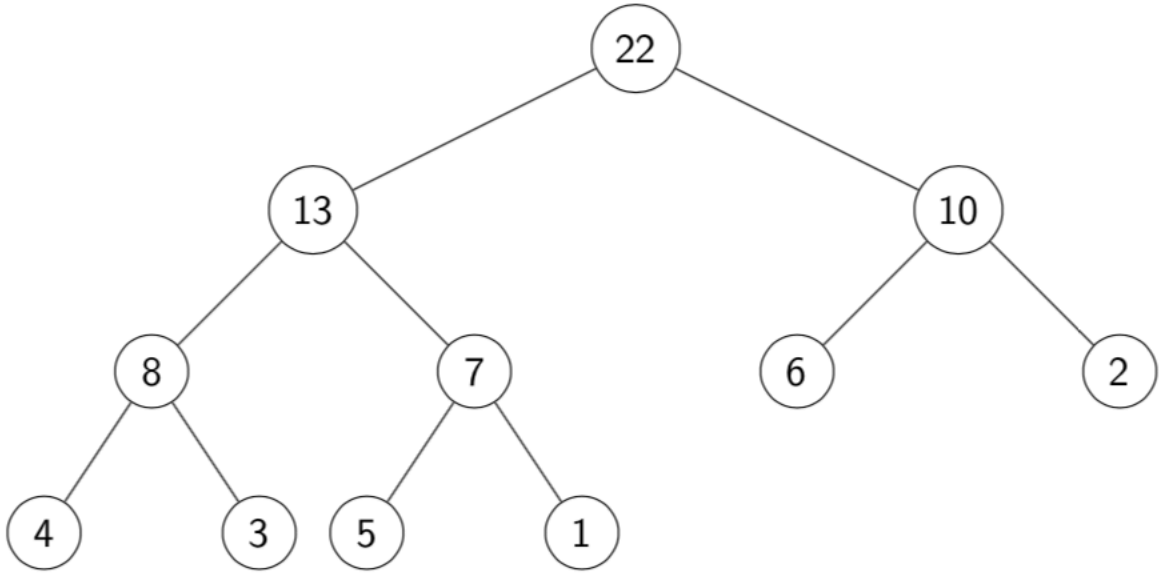
**Min-Heap**

For every node excluding the root,
value is at least that of its parent: $A[parent[i]] \leq A[i]$

Smallest element is stored at the root.

In any subtree, no values are smaller than the value stored at subtree root

# Example




Max-heap or Min-heap?

# Heap properties

It is a binary tree with the following properties:

- *Property 1:* it is a complete binary tree
- *Property 2:* the value stored at a node is greater or equal to the values stored at the children

(**heap property**)

- heap property: for all nodes n in the tree,
  v.parent.data >= v.data
  (assuming we have a node class like for BSTs)
- This is for max-heaps
  (for min-heaps, v.parent.data =< v.data)

Heap represented as an array A has two attributes:

1. Length(A) – number of elements in array A
2. HeapSize(A) – number of elements in heap stored in A

Clearly, need Length(A) ≥ HeapSize(A) at all times

Heap property now: A[v.parent.index] >= A[v.index]

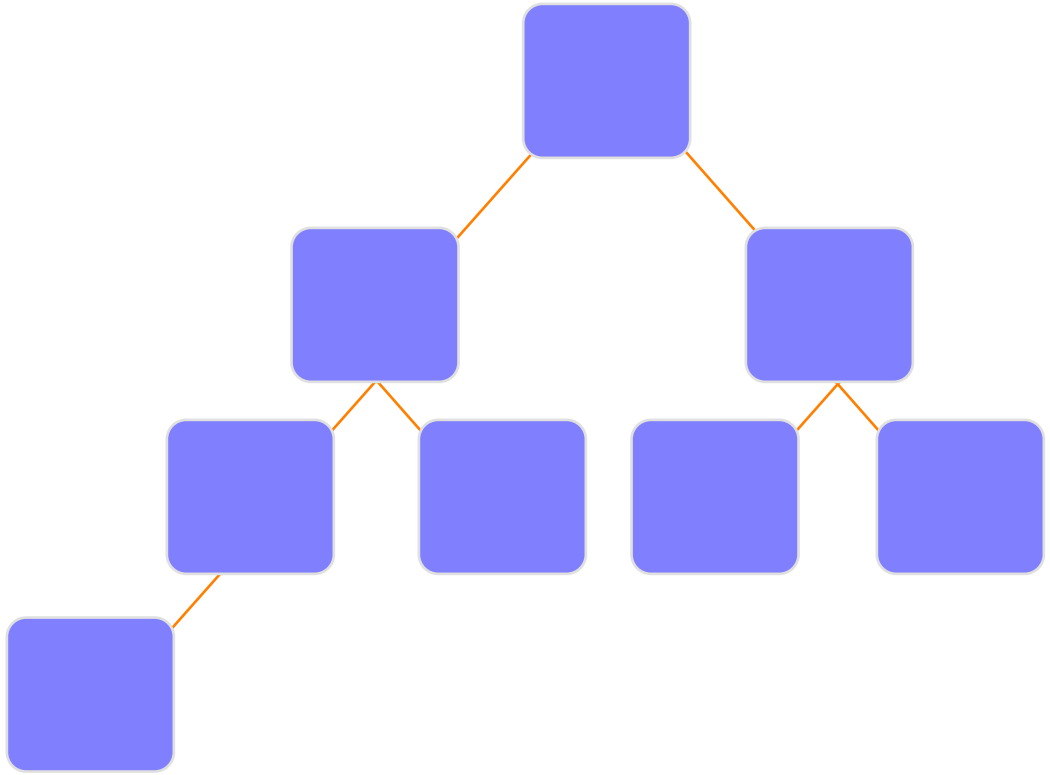
Height of a heap (h): $\lfloor \log n \rfloor$

# Building a heap

When a complete binary tree is built, its first node must be the root.

The second node is always the left child of the root.

The third node is always the right child of the root.



The next nodes always fill the next level from left-to-right.

# Insertion into a Heap

To insert element to a binary max-heap H

Step 1. add the new element node to the bottom left H

Step 2. If the new node value is bigger than its parent, swap their values. Set the parent node as new node, go to step 2.

Step 3. else exit.

Step 2 is called heapify

# What about them array indices, then?

Assume we start counting at position 1

- the root is in A[1]

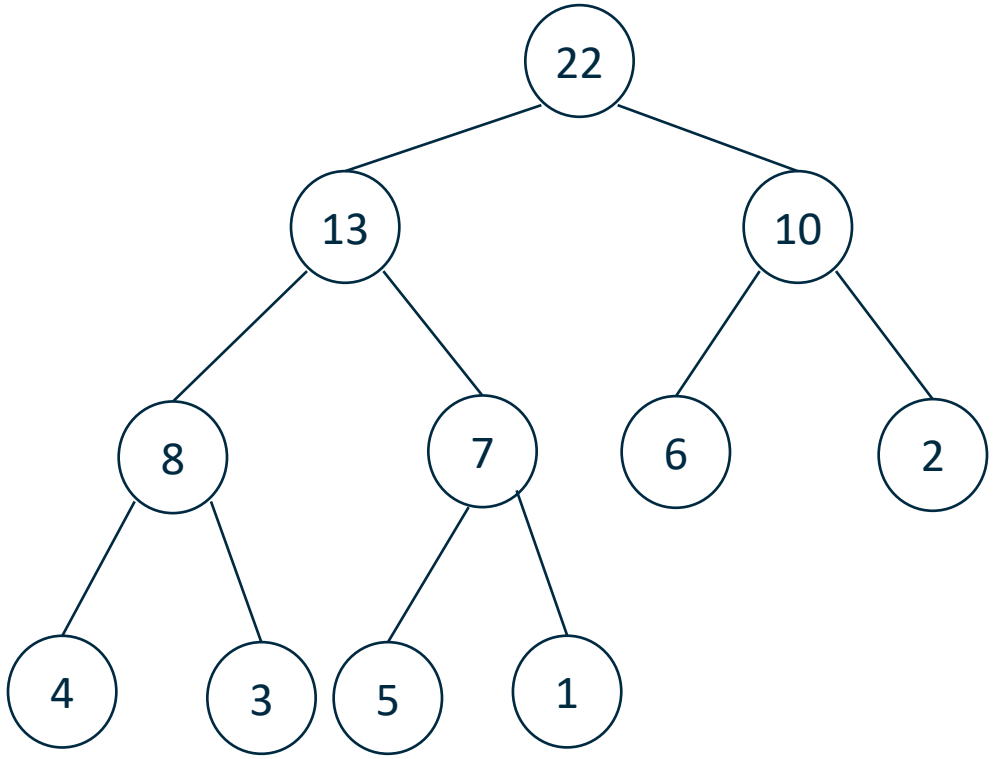
Parent-child relationships (for index I):

- left(i) = A[2*i]
- right(i) = A[2*i+1]
- parent(i) = A[i/2] – integer division, rounds down

So, v.left.index = 2*(v.index), and so forth

# Example: now with array indices

Max-heap as a binary tree.




Max-heap as an array.

| 22 | 13 | 10 | 8 | 7 | 6 | 2 | 4 | 3 | 5 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

The corresponding array: A=[22, 13,10, 8,7,6,2, 4,3,5,1]

# But. . . why???

Very good data structure for priority queues and for sorting

Given sequence of objects with different priorities, want to deal with highest priority items first

So as to support priority queues we need to extract maximum element from collection, quickly:

HeapExtractMax(A)

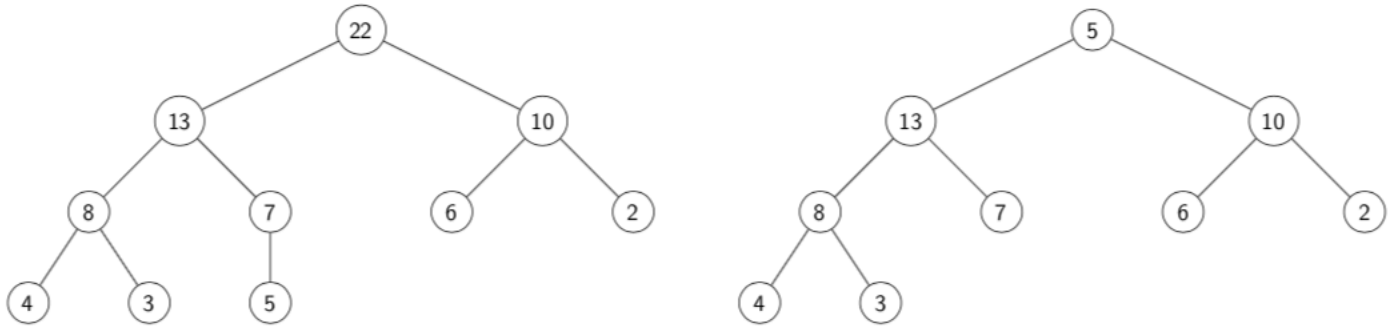
```
1  ret = A[1] // biggest element (highest priority)
2  A[1] = A[HeapSize(A)]
3  HeapSize(A) = HeapSize(A)-1
4  Heapify(A,1,HeapSize(A))
5  return ret
```

Running time: O(1) plus time for heapification

# Example

```
HeapExtractMax(A)
1  ret = A[1]
2  A[1] = A[HeapSize(A)]
3  HeapSize(A) = HeapSize(A)-1
4  Heapify(A,1,HeapSize(A))
5  return ret
```

After extraction (deletion) of maximum, but before heapification

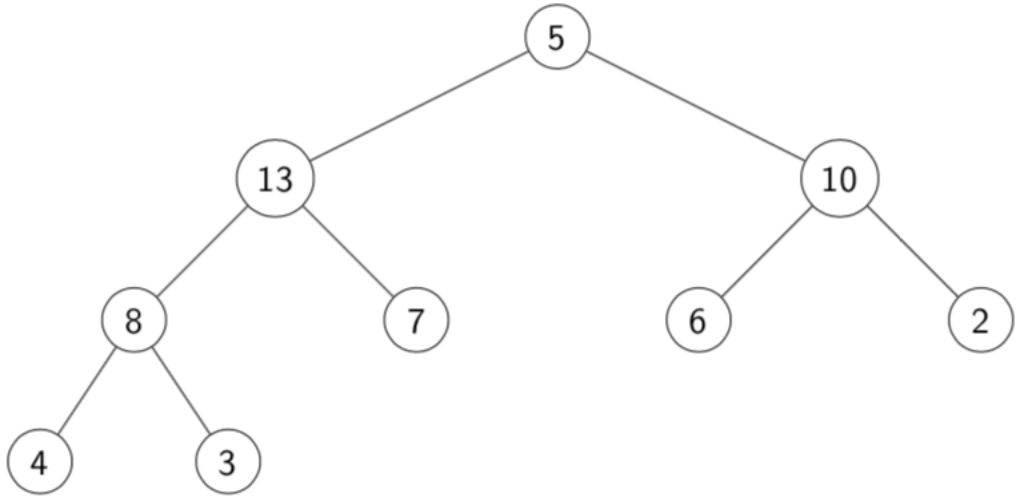


# What's to come?

1. Heapify: maintaining heap property
2. BuildHeap: initially building a heap
3. HeapSort: sorting using a heap

# Heapify

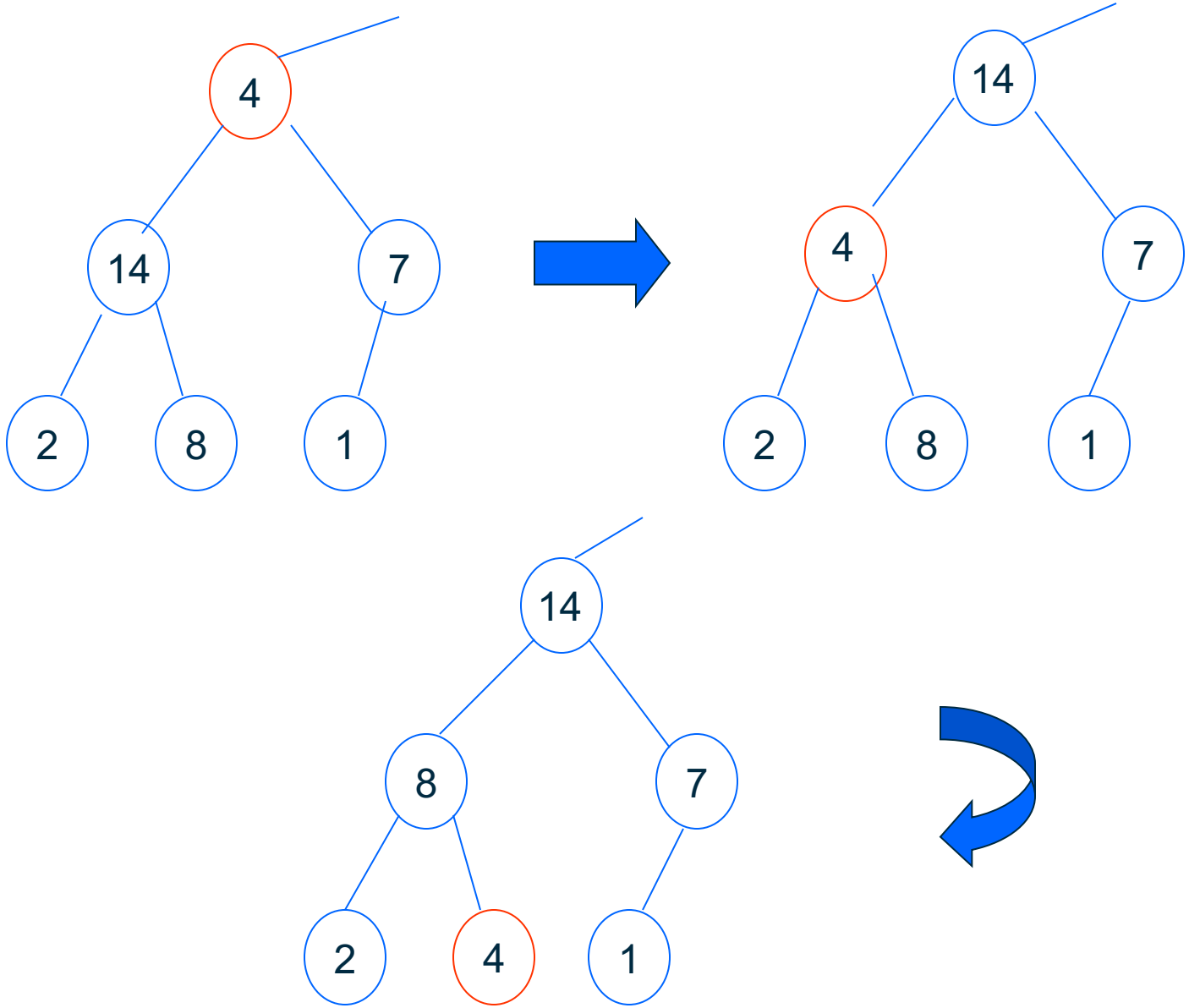
Given something like this, how to turn it into a proper heap again?




Any ideas?

# Idea

1. starting at the root, identify largest of current node v and its children
2. suppose largest element is in w
3. If w ≠ v
    1. swap A[w] and A[v]
    2. recurse into w (contains now what root contained previously)

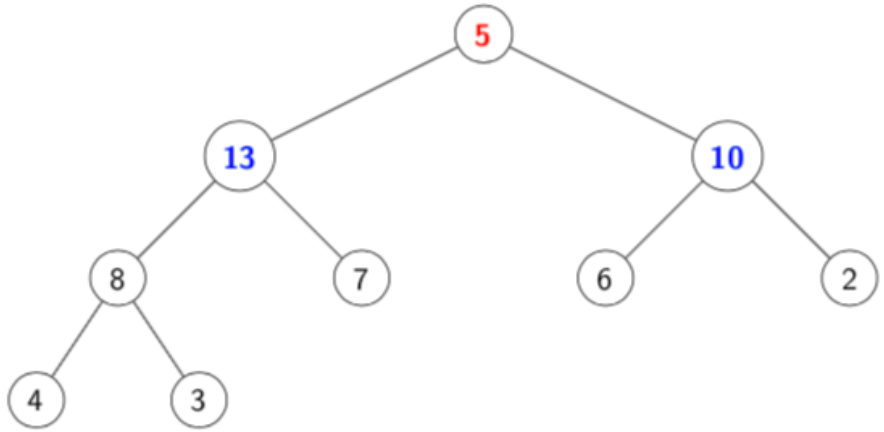

4
14
7
2
8
1
14
4
7
2
8
1
14
8
7
2
4
1

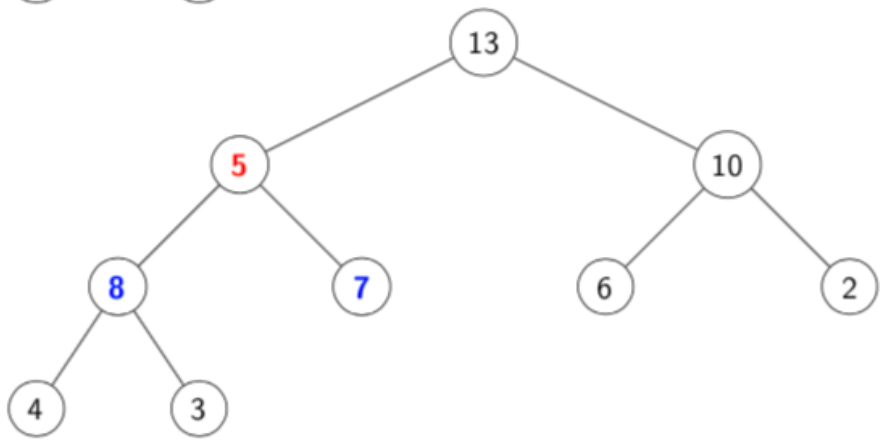
## Heapify (A, v, n)

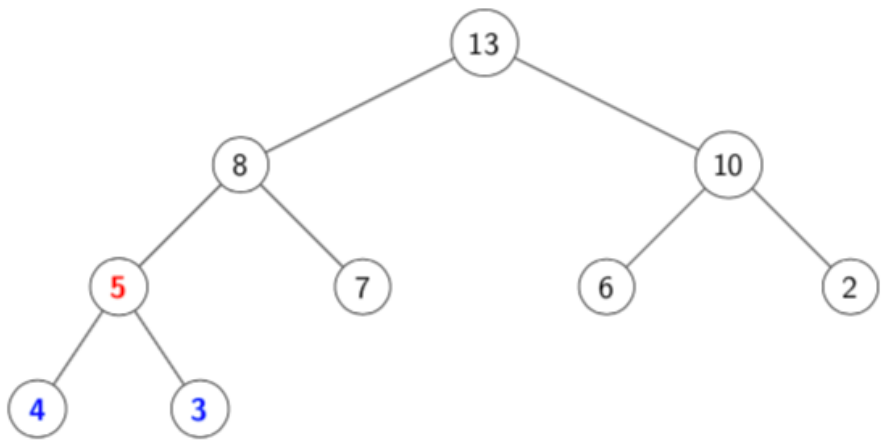
```
// n is heap size
// find largest among v and 2v (left child)
largest = v
if 2v <= n and A[2v]>A[v] then largest = 2v

// find largest among winner above and
// 2v+1 (right child)
if 2v+1 <= n and A[2v+1]>A[largest] then
  largest = 2v+1

if largest != v then
  swap A[v], A[largest]
  Heapify (A, largest, n)
```




Running time: linear in height of tree, O(logn)

# Running time

Heapify (A, v, n)

```
// n is heap size
// find largest among v and 2v (left child)
largest = v
if 2v <= n and A[2v]>A[v] then largest = 2v

// find largest among winner above and
// 2v+1 (right child)
if 2v+1 <= n and A[2v+1]>A[largest] then
  largest = 2v+1

if largest != v then
  swap A[v], A[largest]
  Heapify (A, largest, n)
```

Time to fix node *i* and its children = **Θ(1)**

*T*(size of subree at *largest*)

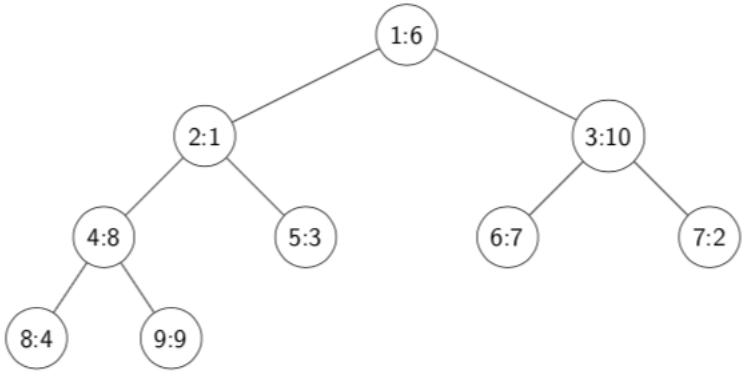
*O*(log *n*)

Durham University

# BuildHeap

Task: given array A with n arbitrary numbers stored in it, convert A into a heap

Example: A=[6,1,10,8,3,7,2,4,9] gives



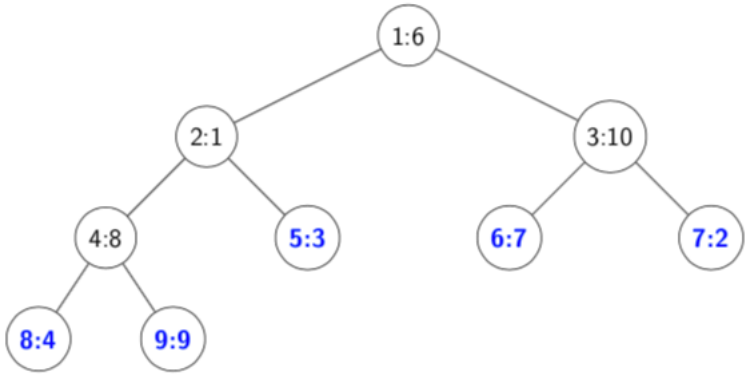

Clearly a lot of work to be done here!

**Note:** leaves are fine!

**Idea:** start from leaves and build up from there

BuildHeap(A,n)

```
for i=n downto 1 do
  Heapify(A,i,n)
endfor
```



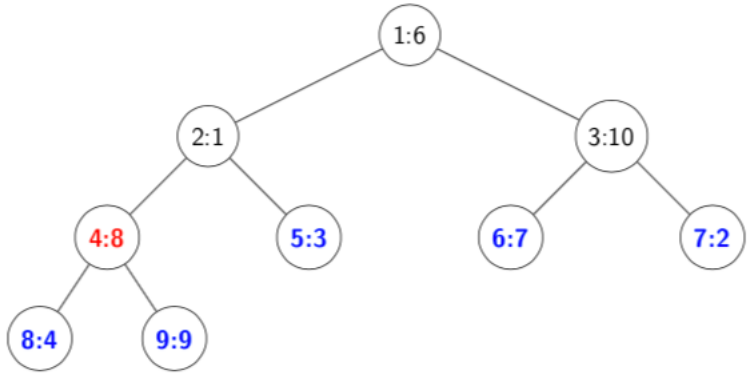

Heapify(A,9,9) exits – leaf

Heapify(A,8,9) exits – leaf

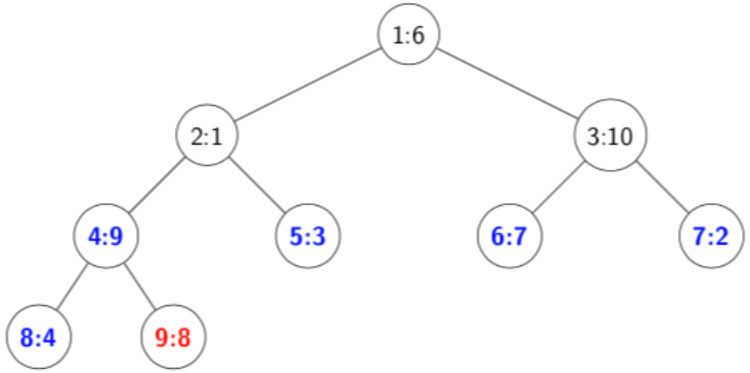
Heapify(A,7,9) exits – leaf

Heapify(A,6,9) exits – leaf

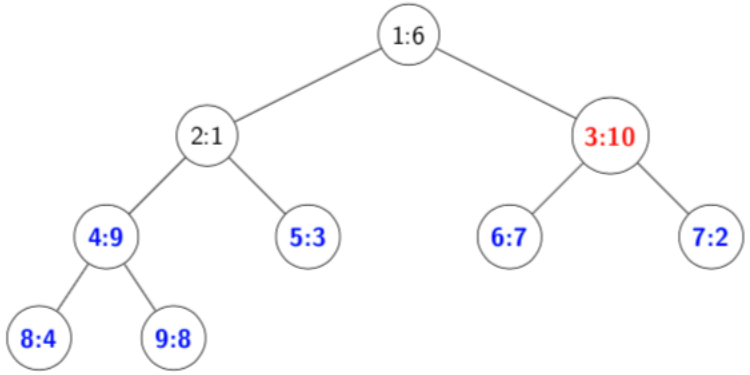
Heapify(A,5,9) exits – leaf



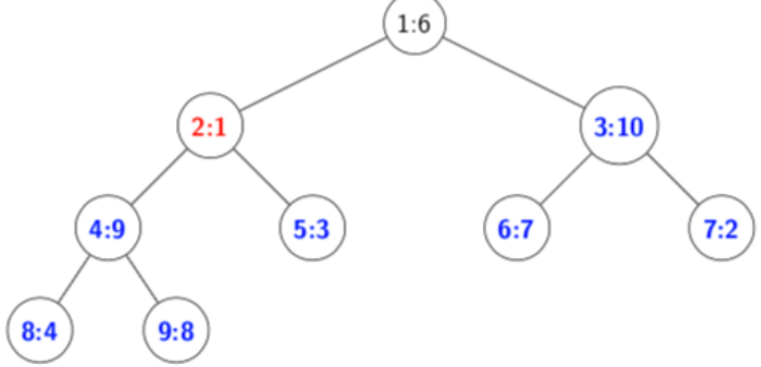

Heapify(A,4,9) puts largest of 8,4,9 into A[4]



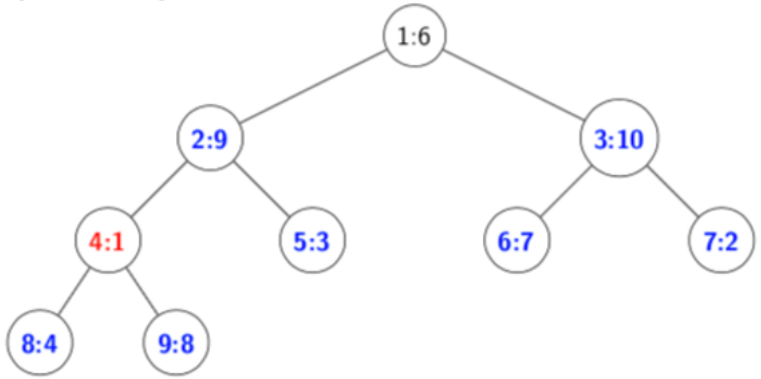

and recursively calls Heapify on A[9] (here: exits)



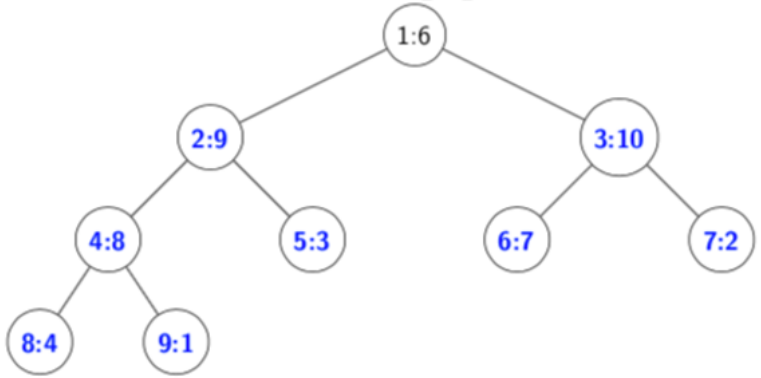

Heapify(A,3,9) puts largest of 10,7,2 into A[3] (here: exits)

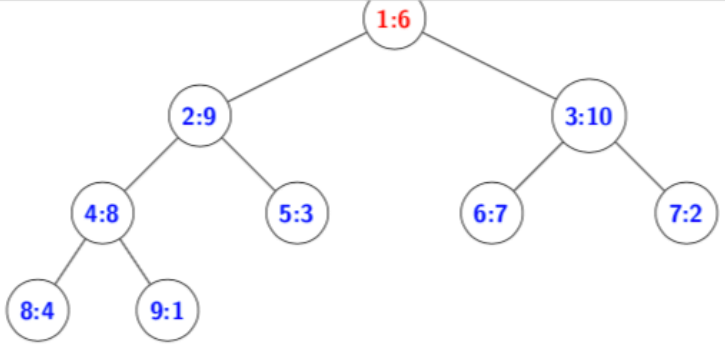



Heapify(A,2,9) puts largest of 1,9,3 into A[2]

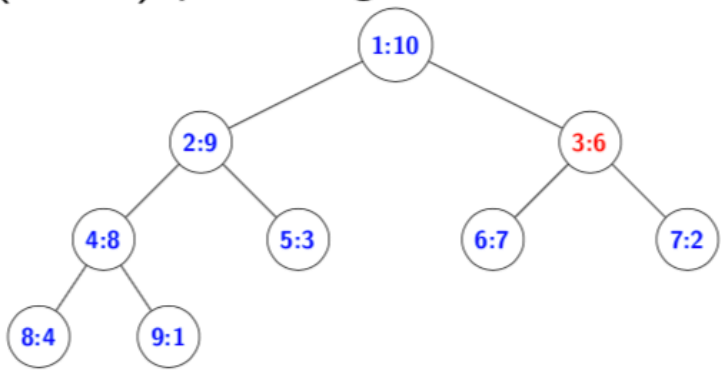



and recurses into A[4]

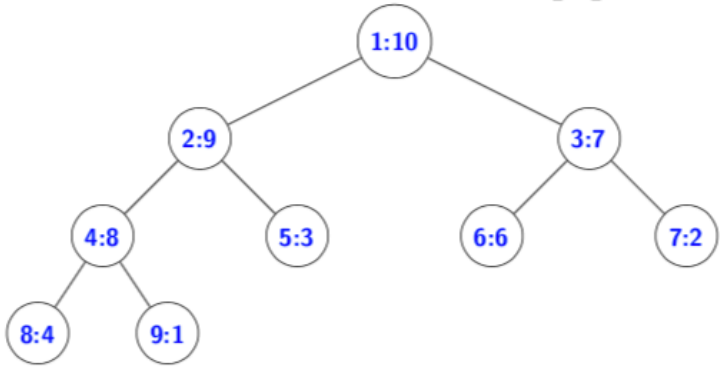





Heapify(A,1,9) puts largest of 6,9,10 into A[1]




and recurses into A[3]



# Running time BuildHeap

## Loose upper bound:

- Cost of a *Heapify* call × No. of calls to *MaxHeapify*
- $O(\log n) \times O(n) = O(n \log n)$

## Tighter bound:

- Cost of a call to *Heapify* at a node depends on the height, $h$, of the node – $O(h)$.
- Height of most nodes smaller than $n$.
- Height of nodes $h$ ranges from 1 to $\lfloor \lg n \rfloor$.
- No. of nodes of height $h$ is $\lceil n/2^h \rceil$

# Running time BuildHeap

$$T(n) = \sum_{h=1}^{\log n} (\#\text{ nodes at height } h) \cdot O(h)$$

$$\leq \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot O(h) \;=\; O\left(\sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h\right) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right)$$

Well known: for $x \in (0,1)$,

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

$$\sum_{h=1}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h$$

Use this as (x = 1/2)

$$= \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

and hence

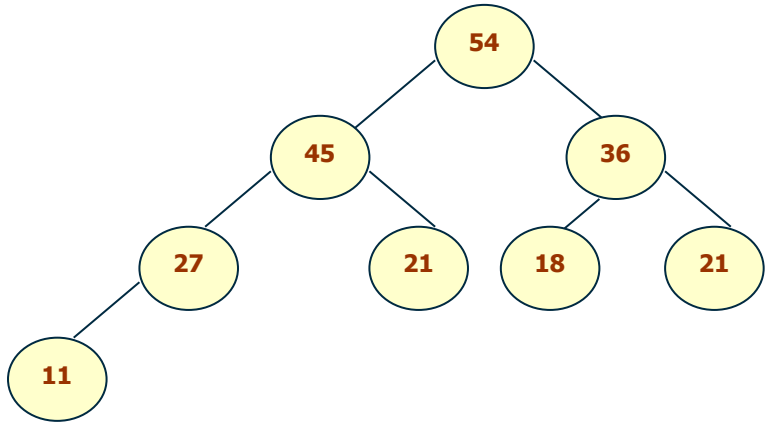
$$T(n) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right) = O(n)$$

Durham University

Can build a heap from an unordered array in linear time

# Example (simpler)

Insert 99 into the following heap



# Extraction (Deletion)

An element is always deleted from the root of the heap. So, deleting an element from the heap is done in these major steps.

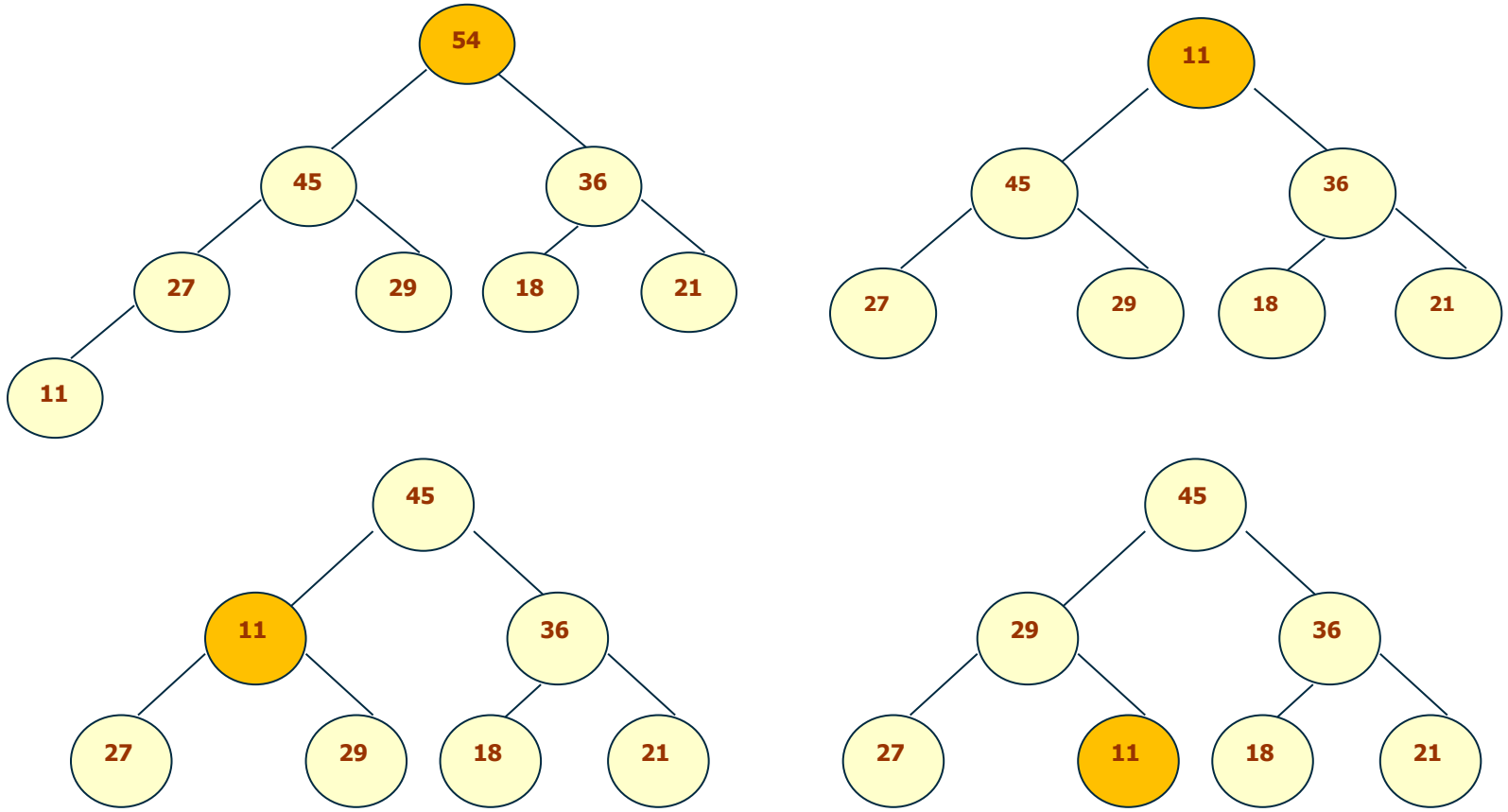
Step 1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.

Step 2. Delete the last node.

Step 3. Move down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

# Example

Consider the heap H given below and delete the root node's value.




Durham University

# Finally: HeapSort

Trivial:

- call BuildHeap on unsorted data, and
- repeatedly call HeapExtractMin until empty

Running time: $O(n) + n \cdot O(\log n) = O(n \log n)$

This version is incorrect.

Why?

– Can’t find min in a max-heap quickly enough

# Proper HeapSort

HeapSort(A)

```
BuildHeap(A, Length(A))
for i = Length(A) downto 2 do
  swap A[i] and A[1]
  HeapSize(A) = HeapSize(A)-1
  Heapify(A, 1, HeapSize(A))
endfor
```

Running time: $O(n) + n \cdot O(\log n) = O(n \log n)$

# Example

4, 1, 3, 2, 16, 9, 10, 14, 8, 7

# Sorting Algorithms

We've seen

| Algorithm | Running Time |
|---|---|
| SelectionSort | $O(n^2)$ |
| InsertionSort | $O(n^2)$ |
| MergeSort | $O(n \log n)$ |
| QuickSort | $O(n^2)$ |
| BucketSort | $O(n + K)$ |
| RadixSort | $O(n \log K)$ |
| HeapSort | $O(n \log n)$ |

(worst case)

Trade-offs involving, e.g., efficiency, memory usage, stability

Most programming languages have a built-in sorting function:
highly optimized combination of various algorithms

Durham University

# Thank you