The Sorting Problem

Input: A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle b_1, b_2, \ldots, b_n \rangle$ of the input sequence such that $b_1 \le b_2 \ldots \le b_n$

Example:

- Input: ⟨ **31,41,59,26,41,58** ⟩
- Output: ⟨ **26,31,41,41,58,59** ⟩

Sorting - The Data

- In practice, we usually sort **records** with **keys** and **satellite data** (non-key data)

- Sometimes, if the records are large, we sort pointers to the records

- For now, we ignore satellite data
  assume that we are dealing only with keys only
  i.e. focus on sorting algorithms

Recursion

```
Insertion-Sort(A,n)
  if  n > 1
    Insertion-Sort(A,n-1)
    Put-In-Place(A[n],A,n)
```

```
T(n) = T(n-1) + n
```

Divide and Conquer

Divide

the problem into several (disjoint) sub-problems

Conquer

the sub-problems by solving them recursively

Combine

the solutions to the sub-problems
into a solution for the original problem

Divide and Conquer

Running Time:

$$n = n_1 + n_2 + \dots + n_k$$

$$T(n) = T_{divide}(n) + T(n_1) + T(n_2) + \dots + T(n_k) + T_{combine}(n)$$

If all sub-problems are of same size:

$$n = n_{sub} * (n/n_{sub})$$

$$T(n) = T_{divide}(n) + n_{sub} * T(n/n_{sub}) + T_{combine}(n)$$

Merge-Sort

Divide: Split the list into 2 equal sized sub-lists

Conquer: Recursively sort each of these sub-lists
(using Merge-Sort)

Combine: Merge the two sorted sub-lists
to make a single sorted list

$$T(n) = T_{split}(n) + 2T(n/2) + T_{merge}(n)$$

Merge

Merge(A,p,q,r) $p \le q < r$

Point to the beginning of each sub-array

choose the smallest of the two elements

move it to merged array

and advance the appropriate pointer

Running Time: **cn** for some constant **c** > 0

and **n = r−p+1**

Merge-Sort

```
Merge-Sort(A,p,r)
    if  p < r
        q ← ⌊(p + r)/2⌋
        Merge-Sort(A,p,q)
        Merge-Sort(A,q+1,r)
        Merge(A,p,q,r)
```

$$q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$$

To sort A = ⟨A[1],A[2],…,A[n]⟩:

```
    Merge-Sort(A,1,n)
```

Merge-Sort

Running Time:

$$T(n) = T_D(n) + 2T(n/2) + T_M(n)$$

$T(1) = c_1$ for some $c_1 > 0$

$T_D(n) = c_2$ for some $c_2 > 0$

$T_M(n) = c_3 n$ for some $c_3 > 0$

We can show that

$T(n) = dn\log n$ for some $d > 0$

Properties of Sorting Algorithms

- **In place**

  only a constant number of elements of the input array are ever stored outside the array

- **Comparison based**

  the only operation we can perform on keys

  is to **compare two keys**

  A **non-comparison based** sorting algorithm

  - looks at values of individual elements
  - requires some prior knowledge

- **Stable**

  elements with the same key keep their order

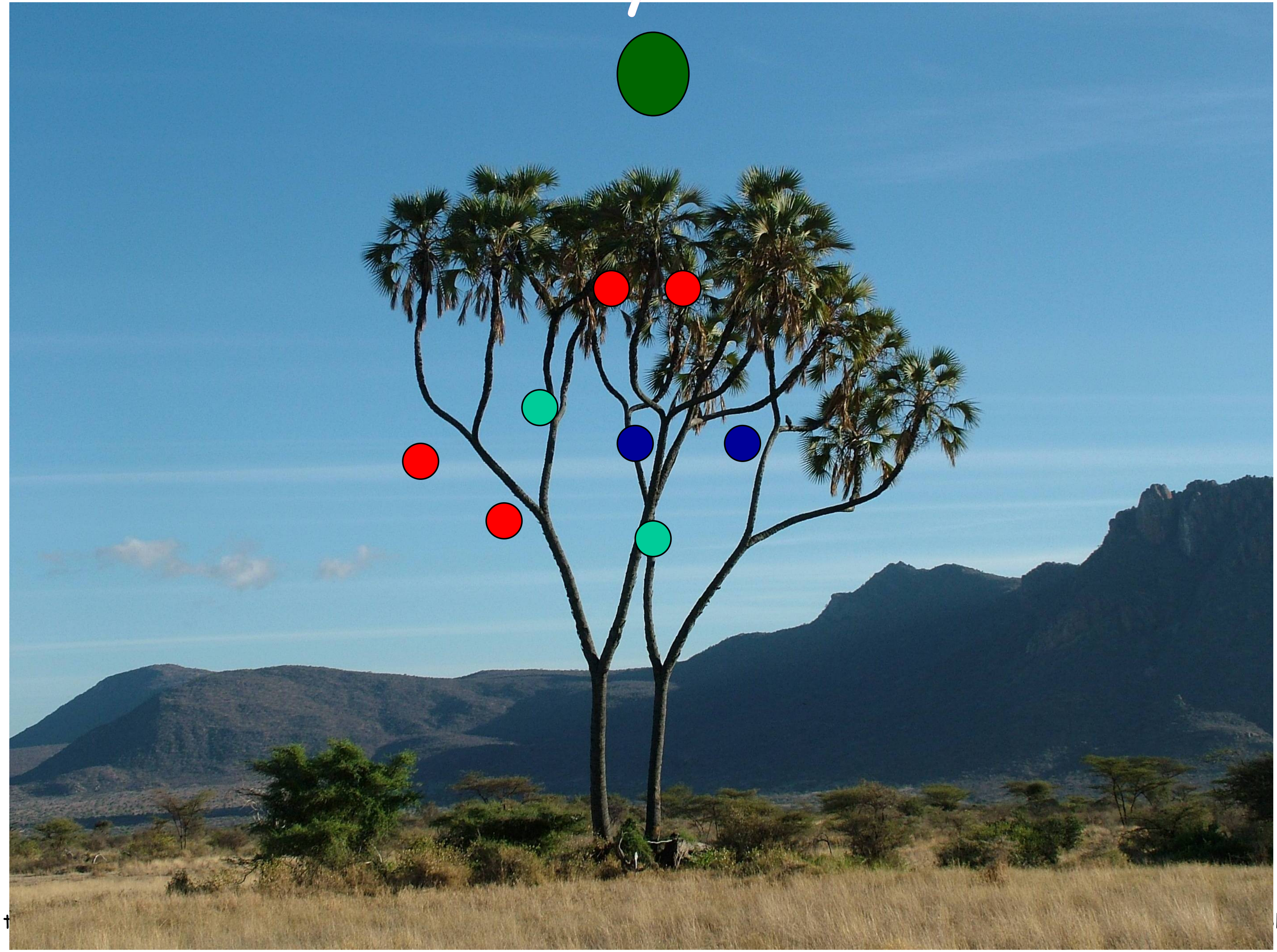
Heap Sort

- Running time – roughly nlog(n)
  like Merge Sort
  unlike Insertion Sort

- In place
  like Insertion Sort
  unlike Merge Sort

- Uses a **heap**

Binary Trees



Binary Trees

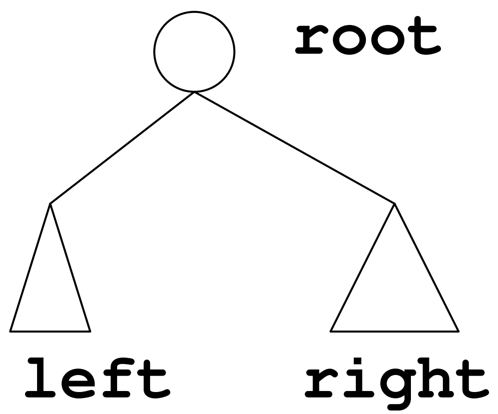
Recursive Definition:

A binary tree

- contains no nodes (Λ),

  or

- has 3 disjoint components:
  - a **root** node, with
  - one binary subtree called its left subtree, and
  - one binary subtree called its right subtree

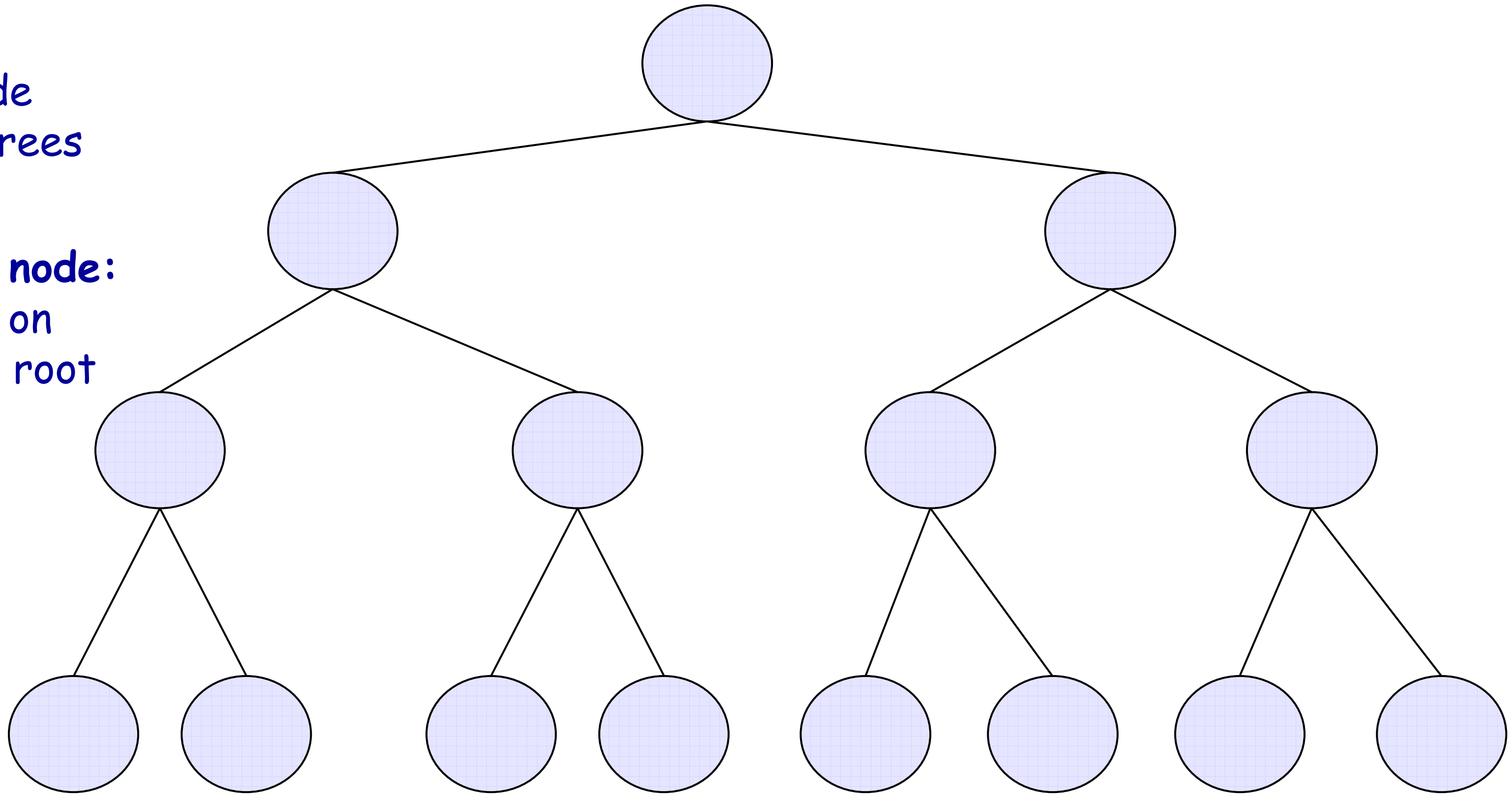


Complete Binary Trees

A Binary Tree is complete if every internal node has exactly two children and all leaves are at the same depth:

Leaf: a node whose subtrees are empty

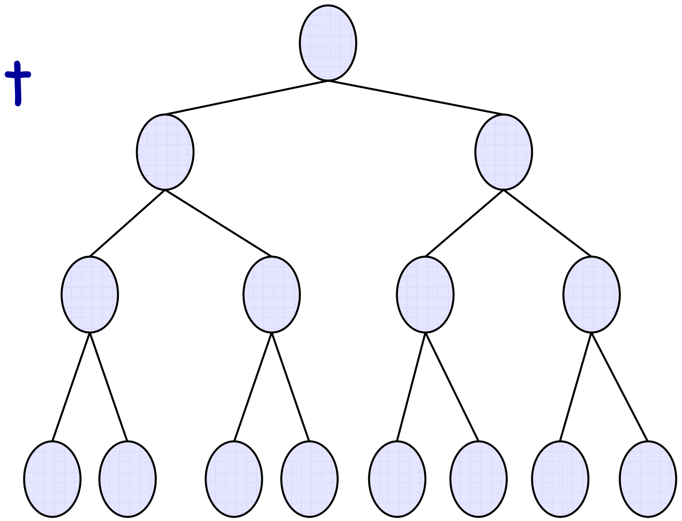
depth of a node: # of edges on path to the root



Complete Binary Trees

Height of a node: Number of edges on longest path to a leaf

Height of a tree = height of its root



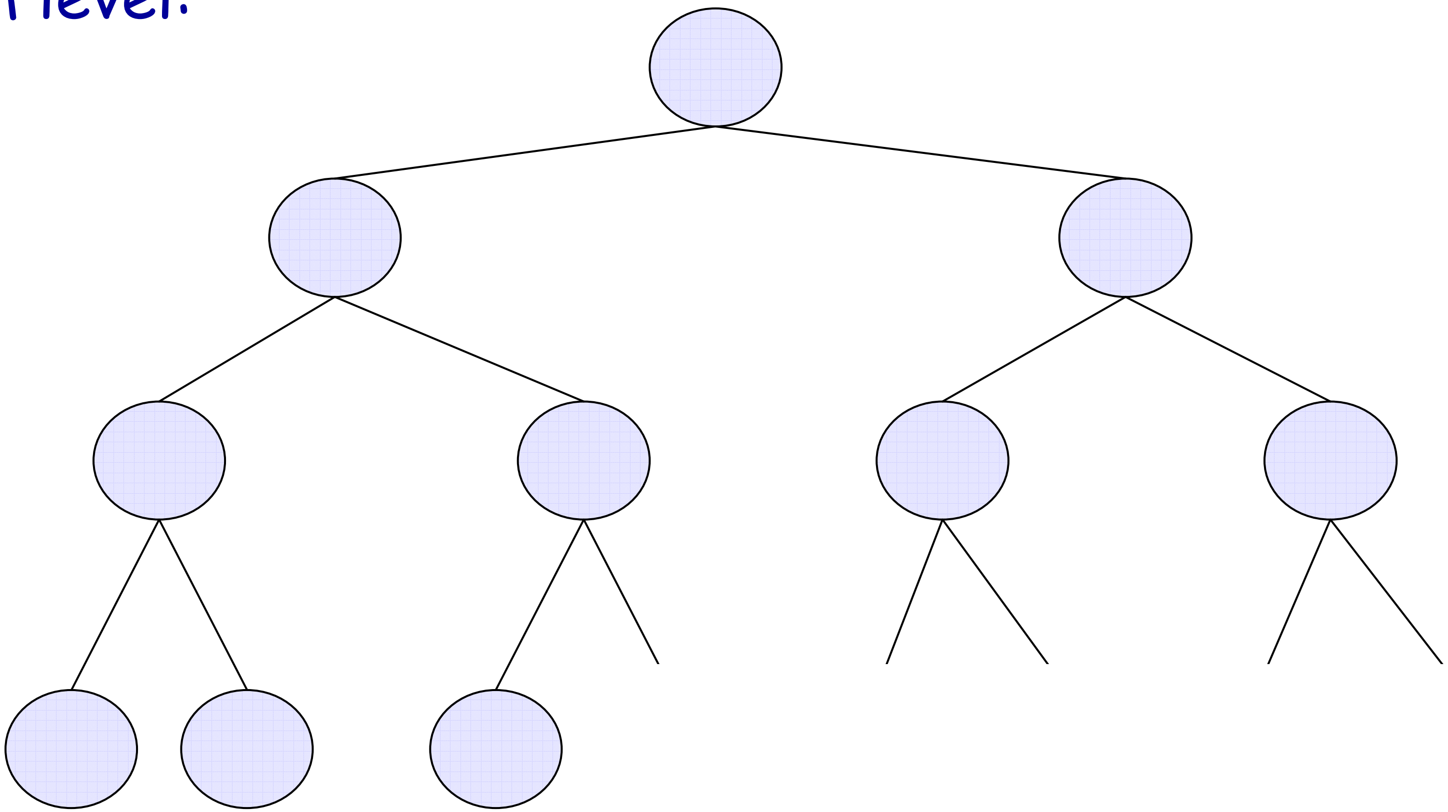
Lemma: A complete binary tree of height h has $2^{h+1}-1$ nodes

Proof: By induction on h
h=0: leaf, $2^1-1=1$ node

h>0: Tree consists of two complete trees of height h-1 plus the root. Total: $(2^h-1) + (2^h-1) +1 = 2^{h+1}-1$

Almost Complete Binary Trees

An almost complete binary tree is a complete tree possibly missing some nodes on the right side of the bottom level:



(Binary) Heaps - ADT

- An almost complete binary tree
- each node contains a key
- Keys satisfy the **heap property**:

  each node's key $\geq$ its children's keys

Binary Tree

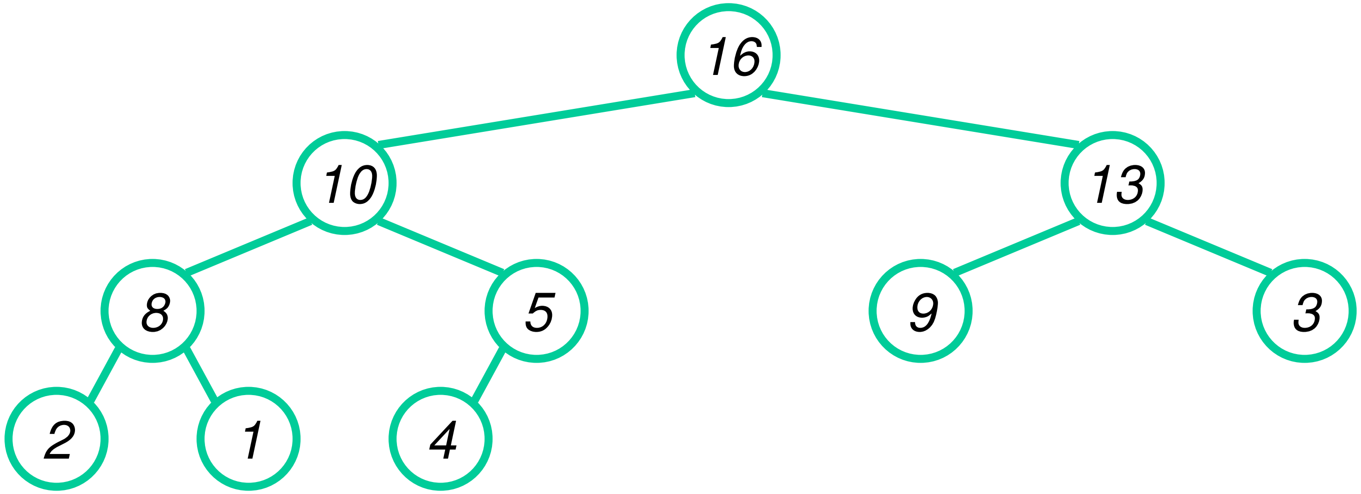
An array implementation:

- **`root`** - at **`A[1]`**
- **`parent(i)`** is in **`A[i/2]`**
  - **`Left(i)`** is in **`A[2i]`**
  - **`Right(i)`** is in **`A[2i+1]`**

height of a node - longest path down to a leaf

height of the tree - height of the root

Implementing Heaps by Arrays

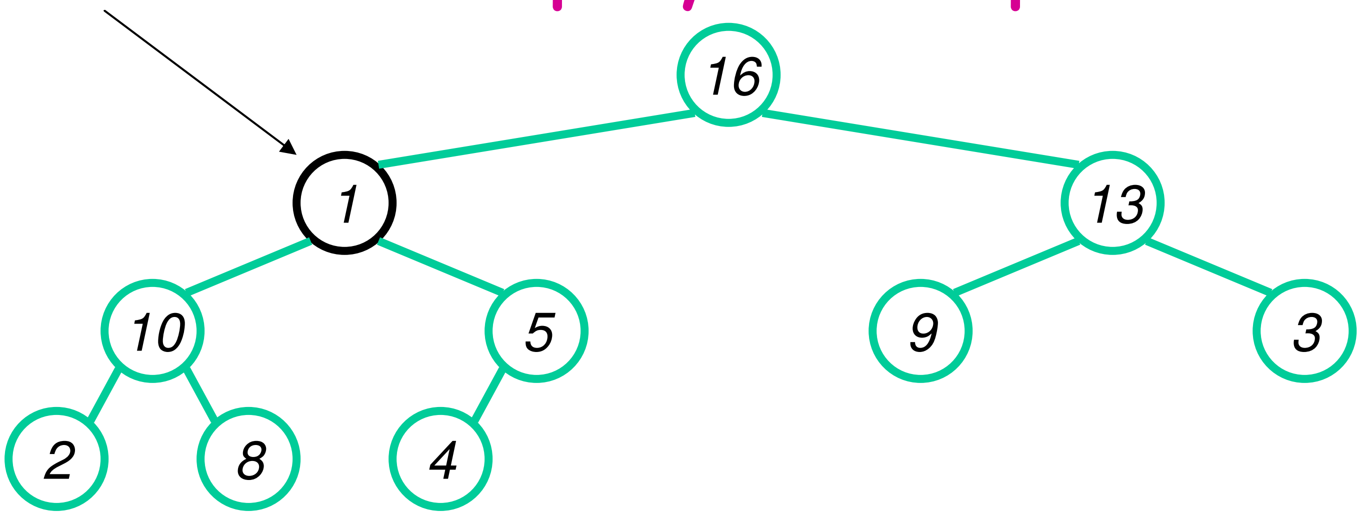



```
Parent(A,i)
    return [i/2]

Left(A,i)
    return 2i

Right(A,i)
    return (2i+1)
```

Heapify(A,i) – fix Heap properties given a violation at position i

	1	2	3	4	5	6	7	8	9	10
A =	16	10	13	8	5	9	3	2	1	4

Heapify Example

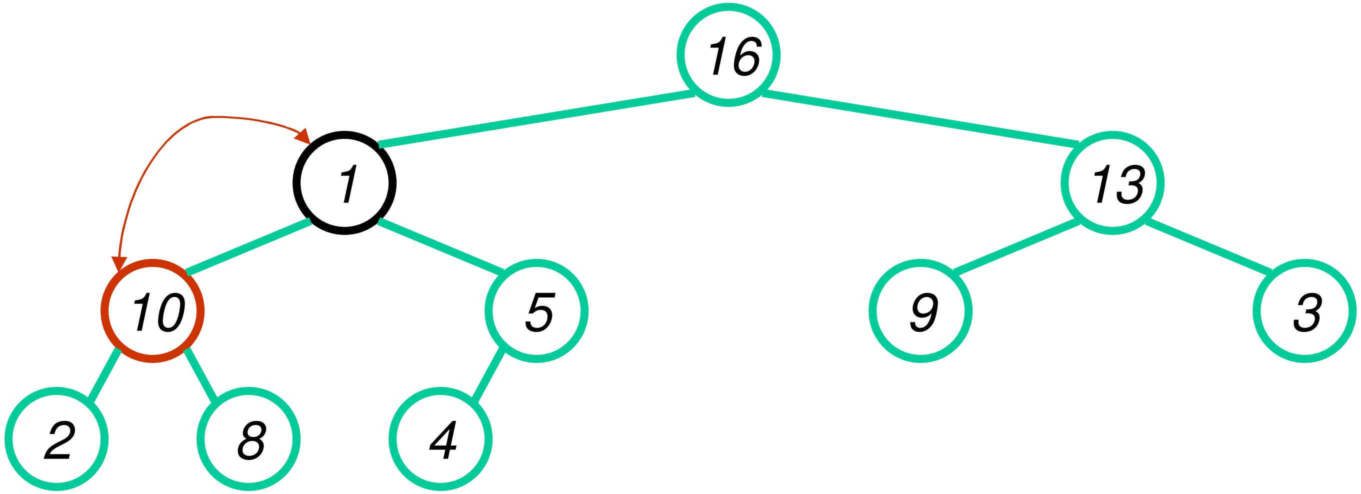


A =

16	1	13	10	5	9	3	2	8	4

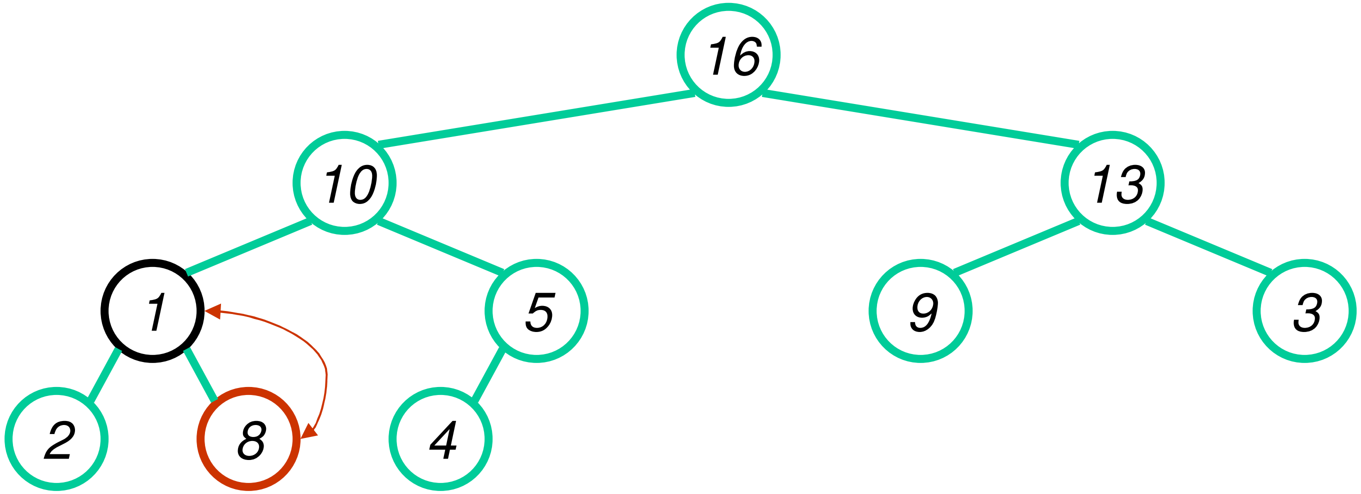
Heapify(A,i) – fix Heap properties given a violation at position i

Heapify Example



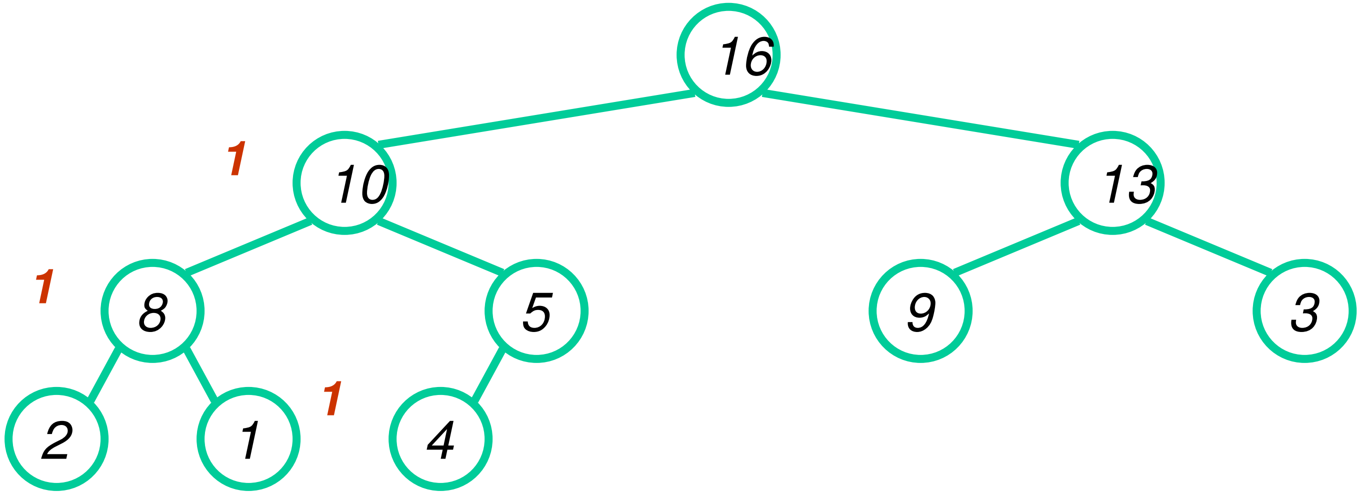
A =	16	1	13	10	5	9	3	2	8	4

Heapify Example



A =	16	10	13	1	5	9	3	2	8	4

Heapify Example



A =									
16	10	13	8	5	9	3	2	1	4

Heapify

```
Heapify(A,i)
   left ← Left(i)                          /* 2i */
   right ← Right(i)                      /* 2i+1 */
   if left ≤ heap-size and  A[left] > A[i]
     largest ← left
     else  largest ← i
   if right ≤ heap-size and A[right] > A[largest]
     largest ← right
   if  largest ≠  i
     swap(A[i],A[largest])
     Heapify(A,largest)
```

Heapify - Running Time

- $c_1 > 0$ - to fix relationships among **A[i], A[Left(i)], A[Right(i)]**
- Height of the tree is **logn**, so

$$\mathbf{T(n) \leq dlogn}$$

$\Rightarrow$ **Heapify** on a node of height **h** takes roughly **dh** steps

Build-Heap

```
BuildHeap(A)
    heapsize[A] ← length[A]
    for  i ← length[A]/2 downto 1
         Heapify(A,i)
```

Running Time: at most **cn** for some **c>0**

(After **BuildHeap** – **A[1]** stores max element)

Build-Heap - Running Time

- We have about **n/2** calls to **Heapify**
- **Cost of** $\leq$ **dlogn** - for each call to **Heapify**

=> TOTAL: $\leq$ **d(n/2)logn**

But we can do better and show a cost of **cn** to achieve a total running time <u>linear</u> in n.

Build-Heap - Running Time

- Assume $N = 2^K - 1$ (a full binary tree of height k)
  - Level 1: $k - 1$ steps for 1 item
  - Level 2: $k - 2$ steps for 2 items
  - Level 3: $k - 3$ steps for 4 items
  - In general: Level i : $k - i$ steps for 2^{i-1} items
  - Until Level k-1: 1 step for 2^{k-2} items

$$\text{Total Steps} = c\sum_{i=1}^{k-1}(k-i)2^{i-1} = c(2^k - k - 1)$$
$$= c' N$$

By induction on k

Heap-Sort

```
Heap-Sort(A)
  Build-Heap(A)
  for i ← heap-size[A] downto 2
     swap A[1] ↔ A[i]         /* extract-max */
     heap-size[A] ← heap-size[A]-1
     Heapify(A,1)              /* fix heap */
```

<u>Running Time:</u> at most **dnlgn** for some d>0

Priority Queue ADT

Priority Queue – a set of elements **S**, each with a key

Operations:

- **insert(S,x)** - insert element **x** into **S**

  $$S \leftarrow S \cup \{x\}$$

- **max(S)** - return element of **S** with largest key

- **extract-max(S)** - remove and return element of **S** with largest key

Heap-Maximum

Heap-Maximum(A)

```
if heap-size[A] ≥ 1
     return( A[1] )
```

=> <u>Running Time:</u> constant

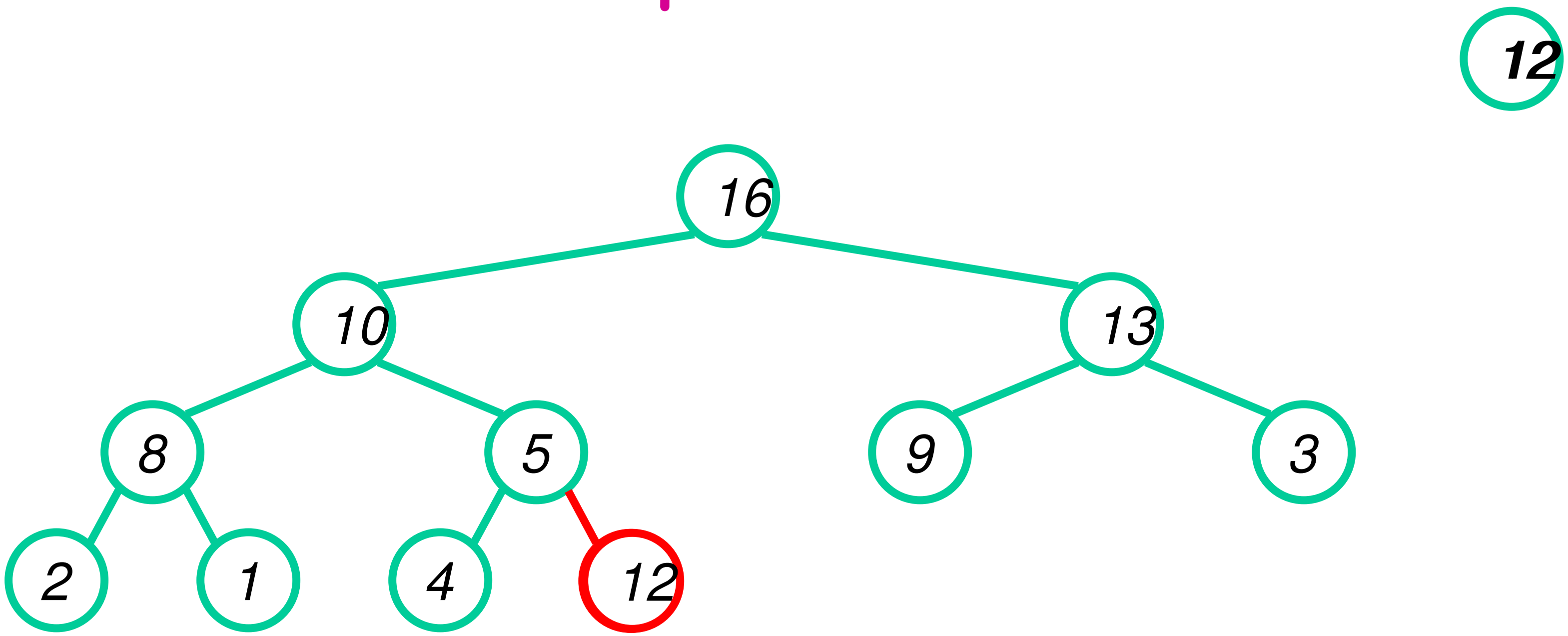
Heap Extract-Max

```
Heap-Extract-Max(A)
    if heap-size[A] < 1
        error "heap underflow"
    max ← A[1]
    A[1] ← A[heap-size[A]]
    heap-size[A] ← heap-size[A]-1
    Heapify(A,1)
    return  max
```

<u>Running Time:</u> dlgn + c = d'lgn

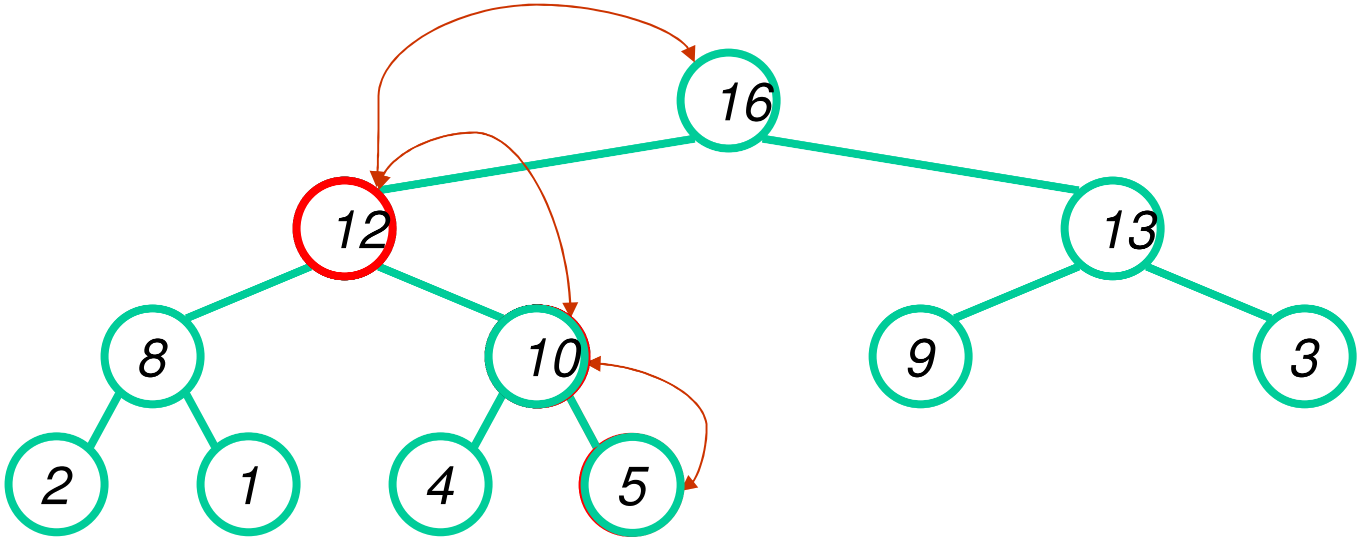
when heap-size[A] = n

Heap Insert



A =	16	10	13	8	5	9	3	2	1	4	

Heap Insert



A =

16	12	13	8	10	9	3	2	1	4	5

Heap-Insert

```
Heap-Insert(A,key)
    heap-size[A] ← heap-size[A]+1
    i ← heap-size[A]
    while  i>0  and  A[parent(i)]<key
        A[i] ← A[parent(i)]
        i ← parent(i)
    A[i] ← key
```

<u>**Running Time:**</u> dlgn

when heap-size[A] = n

PQ Sorting

```
PQ-Sort(A)
    S ← φ
    for  i ← 1 to n
        Heap-Insert(S,A[i])
    for  i ← n downto 1
        SortedA[i] ← Extract-Max(S)
```

PQ here stands for **P**riority **Q**ueue