More sorting algorithms: Heap sort & Radix sort

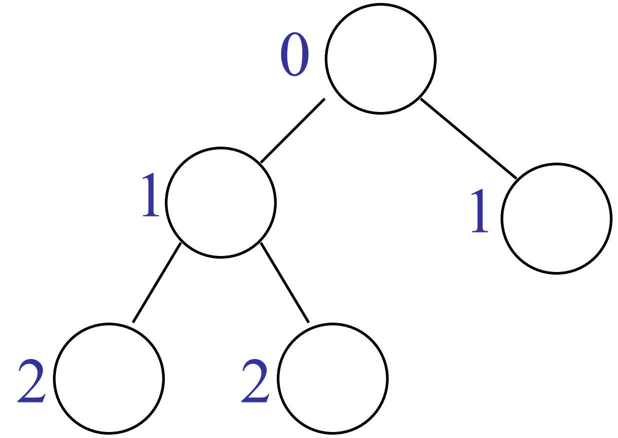
Heap Data Structure and Heap Sort

Basic Definition

- Depth of a tree
  - The depth of a node in a tree is the number of edges in the unique path from the root to that node
  - The depth of a tree is the maximum depth of all nodes in the tree
  - A leaf in a tree is any node with no children
  - Internal node is any node that has at least one child

Depth of tree nodes

- Depth of a node:
  - If node is the root, then depth = 0
  - Otherwise, depth of its parent + 1
- Depth of a tree is the maximum depth of its leaves



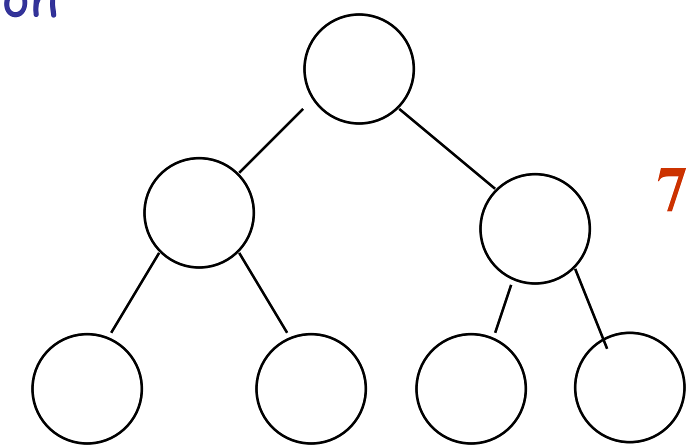
A tree of depth 2

Terminologies

- Complete binary tree
  - Every internal node has two children
  - All leaves have depth d
- Essentially complete binary tree
  - It is a complete binary tree down to a depth of d-1
  - The nodes with depth d are as far to the left as possible

A complete binary tree

- A complete binary tree is a binary tree such that:
  - All internal nodes have 2 children
  - All leaves have the same depth d
- Number of nodes at level k = 2^k
- Total number of nodes in a complete binary tree with depth d is $n = 2^{d+1} - 1$
  - Exercise: Proof by induction

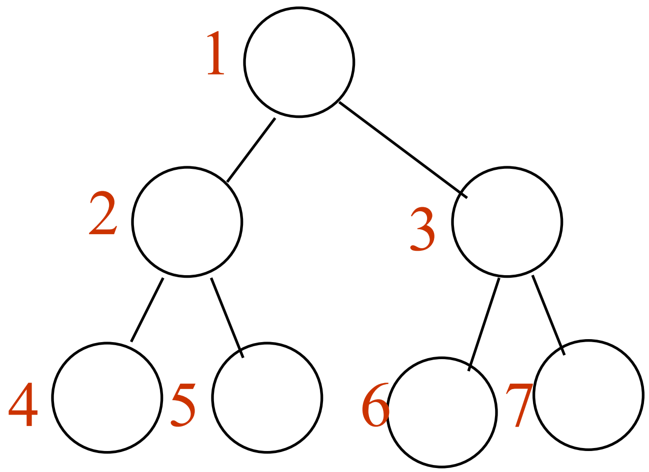


$\mathbf{7} = 2^{2+1} - 1$

A full binary tree of depth = height = 2

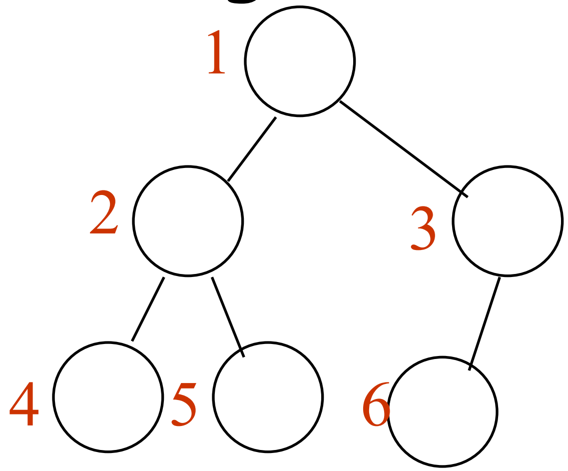
A complete binary tree (cont.)

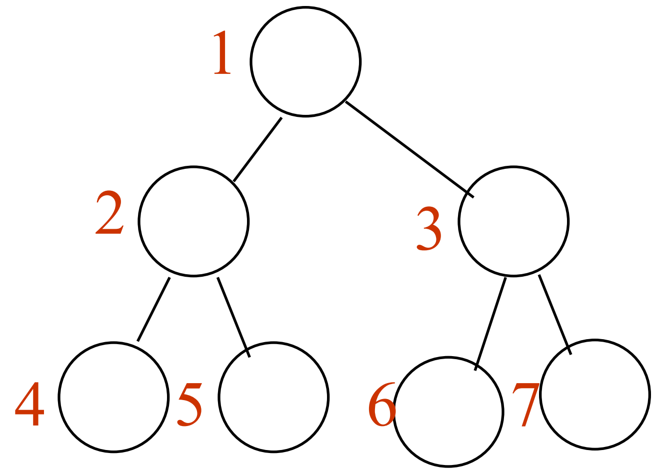
- Number of the nodes of a full (complete) binary tree of depth d:
  - root at depth 0 is numbered 1
  - The nodes at depth 1, …, d are numbered consecutively from left to right, in increasing depth
  - You can store the nodes in a 1D array in increasing order of node number



Essential complete binary tree

- An ***essential*** *complete binary tree* of depth *d* and *n* nodes is a binary tree such that its nodes would have the numbers 1, …, *n* in a binary tree of depth *d*.
- The number of nodes $2^d \le n \le 2^{d+1} - 1$
- $d = \lfloor \lg n \rfloor$ (See the next slide for proof)





Depth of an essential complete binary tree

- Number of nodes n satisfy:

$$2^d \leq n \leq 2^{d+1} - 1 \quad (1)$$

- By taking the log base 2, we get:

$$d \leq \lg n \leq d + 1 \quad (2)$$

- Since d is integer but $lg\ n$ may not be an integer,

$$d = \lfloor \lg n \rfloor$$

- For a complete binary tree, $d = \lfloor \lg n \rfloor$ because (1) & (2) are satisfied for a complete binary tree too

Heap Property

- Heap

  A heap is an essentially complete binary tree such that

  - The values stored at the nodes come from an ordered set
  - The value stored at each node is less than or equal to the values stored at its children → min-heap

- Usage of heap
  - Heap sorting
  - Priority queue

Priority Queue

- A priority queue is a *collection* of zero or more items,
  - Each item is associated with a priority

- Operations:
  - Insert a new item
  - Find the item with the highest priority
  - Delete the highest priority item

Heapsort Algorithm

- Build a heap
- For i = 1 to n - 1

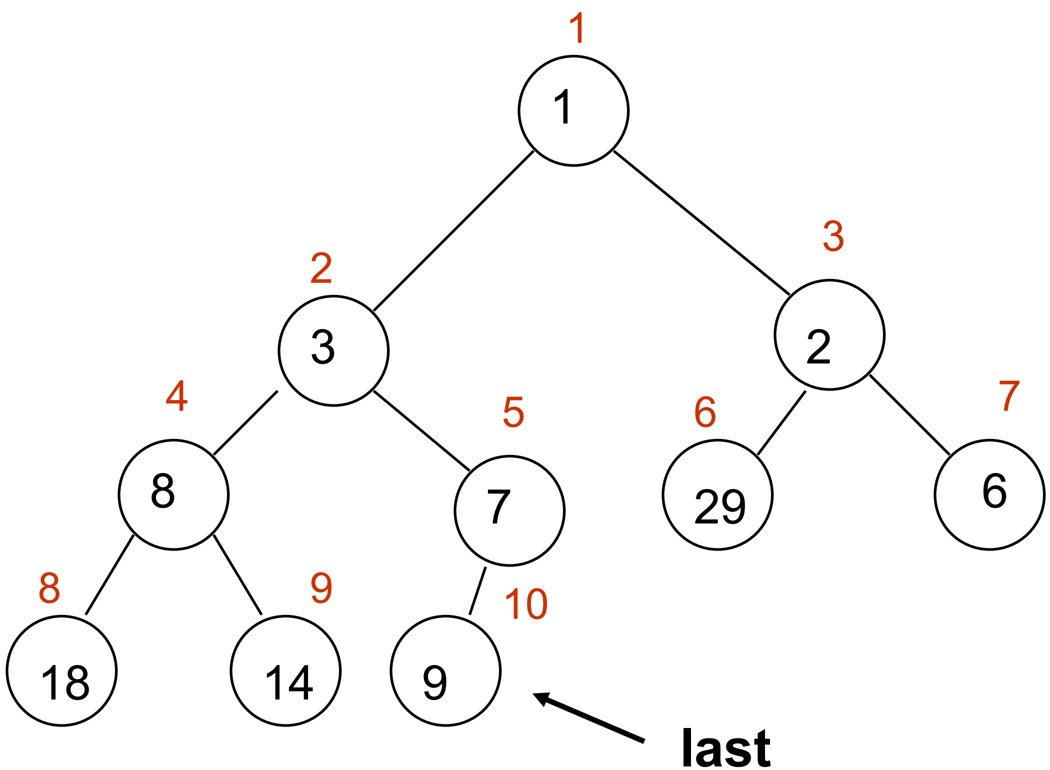
  {
  - Remove the root from the heap and insert it into answer[i]
  - Move the last node to the root
  - Heapify
    - Rearrange the new tree to support the heap property

  }

  return answer[1..n]

Heap data structure

- Exercise: Do heapsort using this heap



$root = 1$

$Parent(i) = \lfloor i/2 \rfloor$

$Left(i) = 2i$

$Right(i) = 2i+1$

array	1	3	2	8	7	29	6	18	14	9
	1	2	3	4	5	6	7	8	9	10

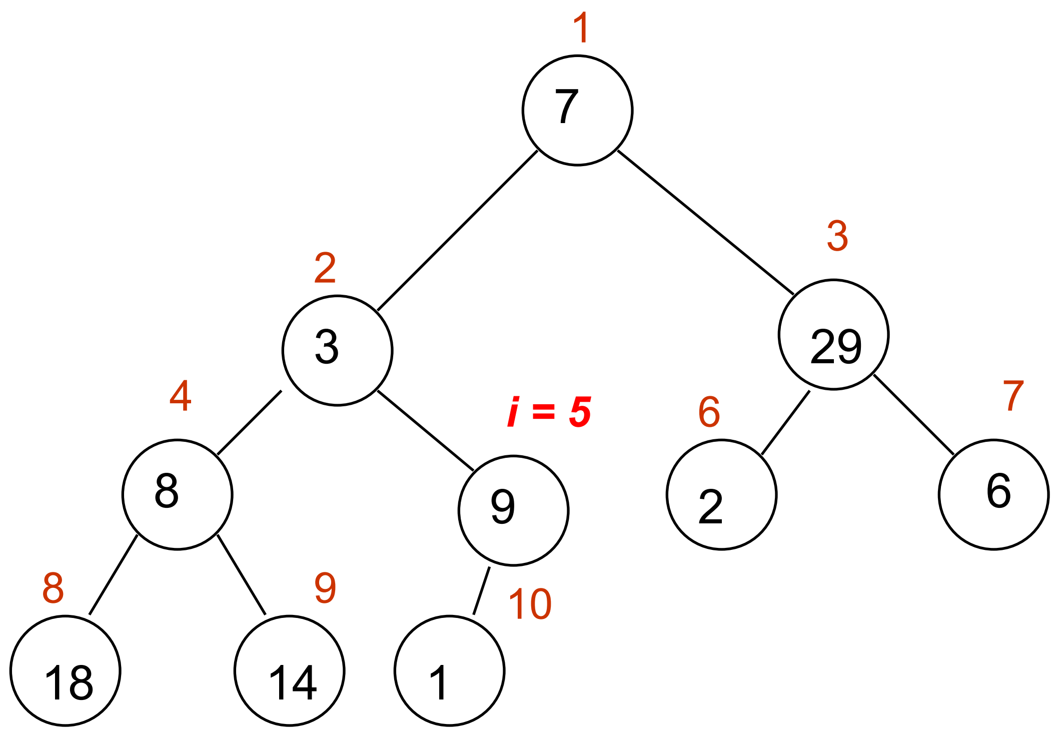
How to build a heap in the first place?

for i = ⌊n/2⌋ downto 1
do heapify

/* n is the last node and ⌊n/2⌋ is its parent in an essentially complete binary tree */

Exercise: Build a min-heap
- Take a bottom-up approach starting from node 5

O(n) for building a heap



Worst case time complexity for heaps

- Build heap with *n* items
  - $\Theta(n)$
- *findMin*()
  - $\Theta(1)$
- *deleteMin*() from a heap with n items
  - $\Theta(\lg n)$
- *insert*() into a heap with n items
  - $\Theta(\lg n)$
- Total $O(n\lg n)$

Lower Bound for Sorting by Comparisons: Recap

- When a list of n integers is given as input, there are n! permutations
- Build a decision tree that has n! leaf nodes where each leaf could be a sorted permutation
- In each node of a decision tree, compare two specific numbers
- Based on the result of the comparisons, take left or right branch
- Depth of the decision tree indicates #total comparisons to reach a sorted permutation
  - Depth: lg n!
  - n! is approximately $(n/e)^n$
  - lg n! = nlg(n/e)
  - Hence, sorting by comparisons is $\Omega(nlgn)$

Linear sorts

Radix sort

Radix sort

- When we know nothing about data to be sorted, we have no choice but sorting them by comparisons. So, sorting is $\Omega(nlgn)$.
- However, if we know something about data, we can take advantage of the knowledge to do sorting faster.
  - Example: Suppose we know that the keys are all nonnegative integers represented in base 10. Also, each key is at most d digits where d is a positive constant.

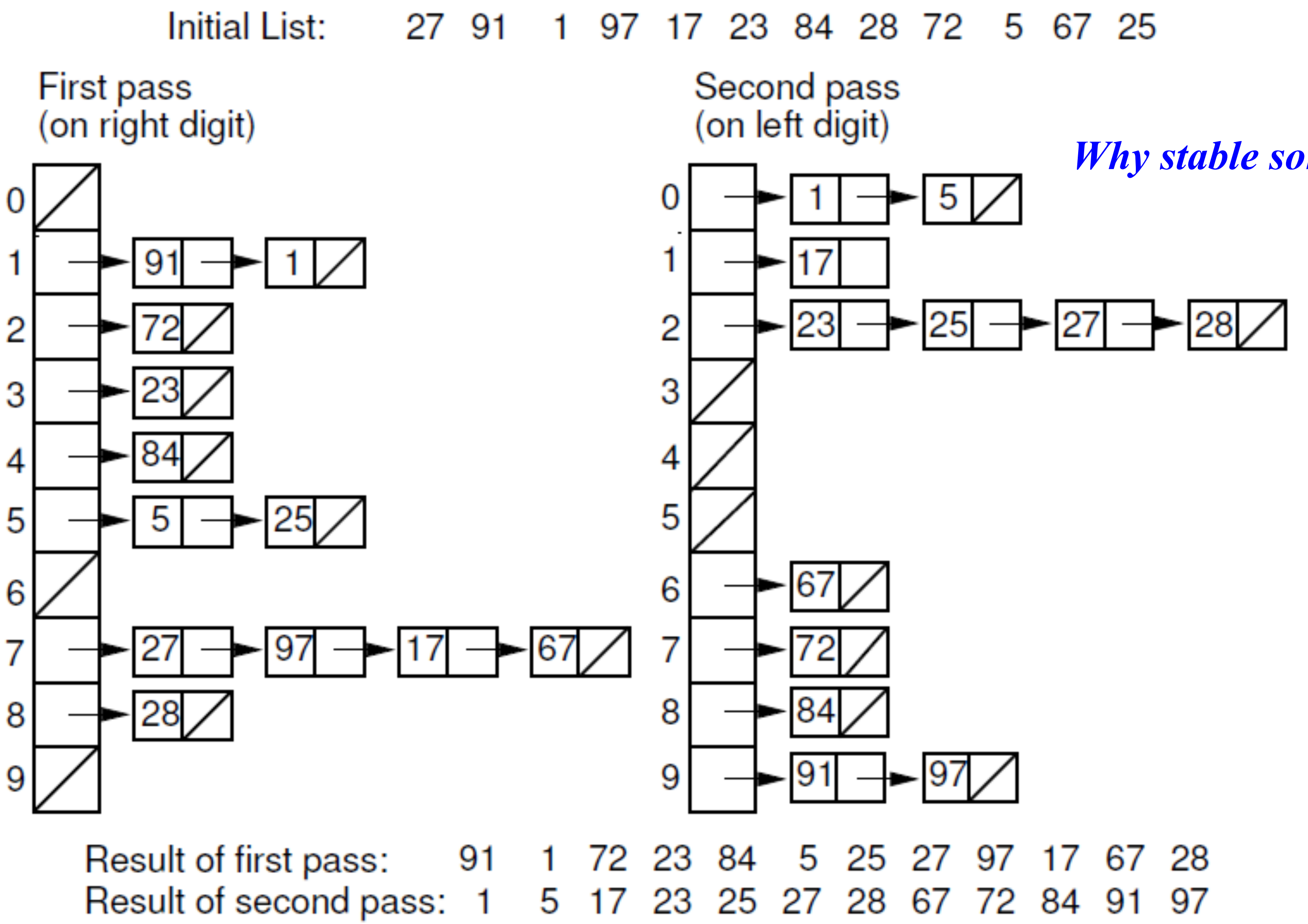
Radix sort

- Main idea
  - Break key into "digit" representation

    key = $i_d, i_{d-1}, \ldots, i_2, i_1$
  - "digit" can be a number in any base, a character, etc.
- Radix sort:

  for i= 1 **to** d

  sort "digit" i using a stable sort
- Analysis : Θ(d $*$ (stable sort time)) where d is the number of "digits"
- Counting sort can be used for stable sorting



Why stable sort?

Source: C.A. Shaffer, A Practical Introduction to Data Structures and Algorithm Analysis (freely available online)