CS303 - Algorithms and Data Structures

Lecture 5

- HeapSort-

Professor : Mahmut Unan – UAB CS

Agenda

- Heap Sort
- Heap Sort Time Complexity
- Quick Sort

Trees

- **General tree** – hierarchical data structure
  - containing k ≥ 1 nodes N = {n1, n2,…, nk}
  - Connected by exactly (k – 1) links
- E= {e1, e2,…, ek-1}
  - **Root** node has no predecessor
  - **Leaves** have no successor
  - All other nodes are **internal nodes**
  - **Nodes** store information
  - Links often referred to as **edges**

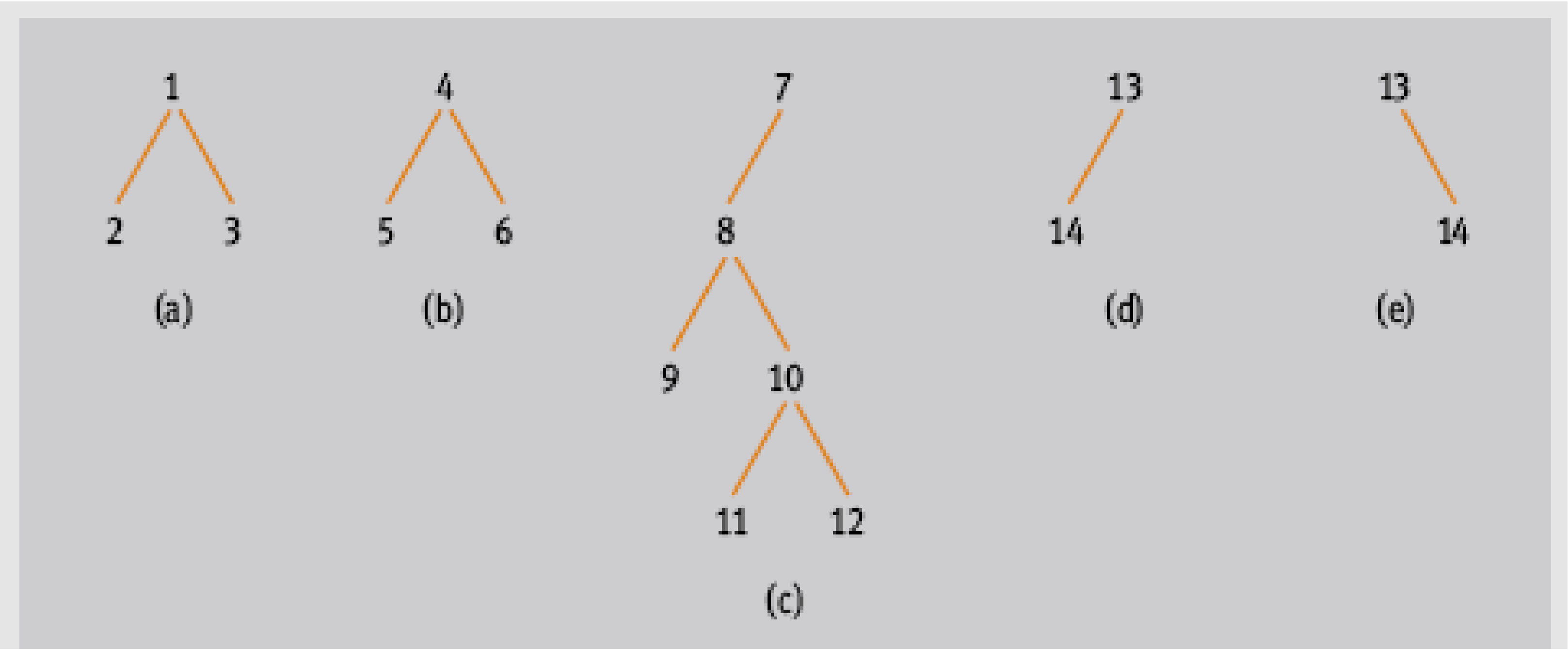
Trees (continued)

- Set of nodes, except the root, can be partitioned into zero or more disjoint subsets
  - Each partition is a **subtree**
  - Partitioning property holds for all subtrees
- Number of successors of a node is the **degree**
  - **Parent** – predecessor of a node
  - **Child** – successor of a node
  - Nodes with same parent are **siblings**

Binary Trees

- Binary trees are distinct from general trees
  - Binary trees can be empty
  - Degree no greater than two
- **Ordered trees:** every node is explicitly identifies as being either the left child or the right child of its parent
- **Binary tree**
  - Finite set of nodes, possibly empty
  - Consists of a root and two disjoint binary trees
  - Called left and right subtrees

Binary Trees (continued)



Heaps

- **Heap** is a binary tree satisfying two conditions:

  – **Order property** – data value in a node is no greater than data values stored in descendants

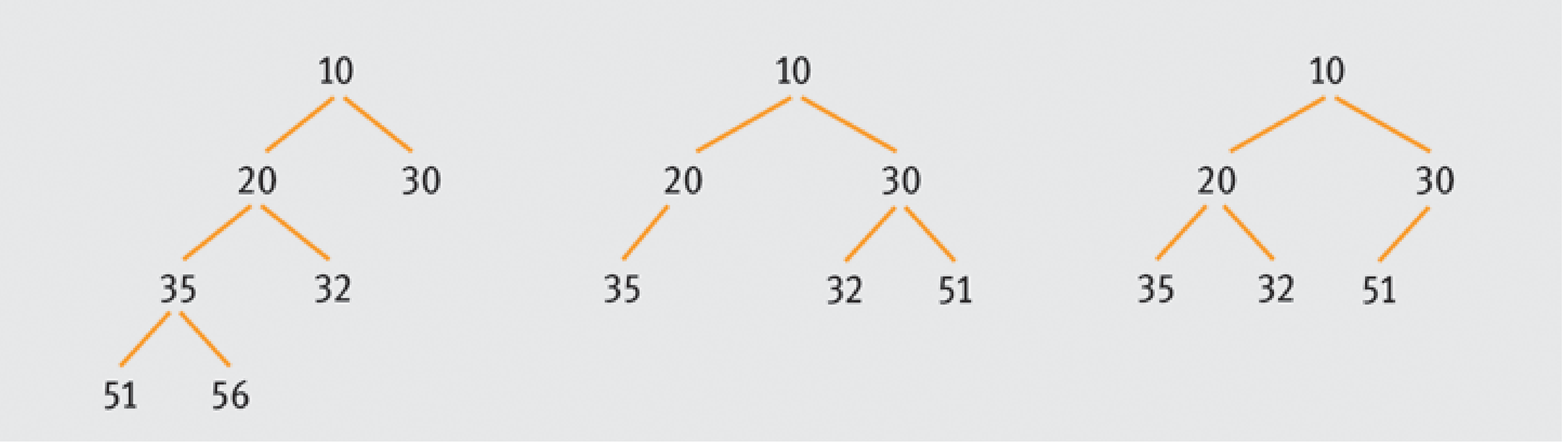
- – **Structure property** – **nearly complete binary tree (complete except last level)**
- Two important mutator methods

  – Insert a new value

  – Remove, return smallest/largest value (remove the root)

- Insertions must maintain order and structure properties

Heaps (continued)

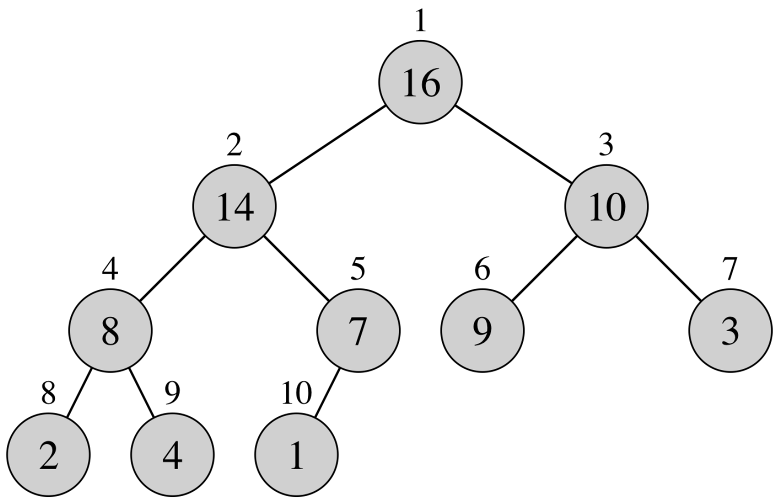


Which of the above binary trees is a complete tree or nearly complete binary tree?

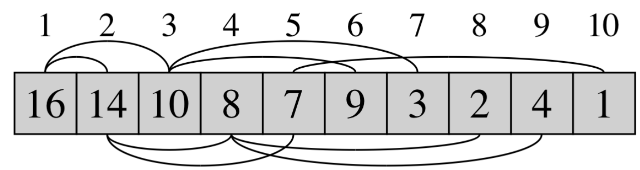
Implementation of Heaps Using One-Dimensional Arrays

- Heaps stored in one-dimensional array in strict left-to-right **level order**
- One-dimensional array representation of a heap is called a **heapform**
  - Do not need pointers
  - If node in position *i* then left child is in
position $2i+1$
- Insert a new value – place value in a unique location that maintains structure property
  - Corresponds to h[size] in heapform array

Implementation of Heaps Using One-Dimensional Arrays (continued)



(a)



(b)

PARENT(i) return |_i / 2_|
LEFT_CHILD (i) return 2i
RIGHT_CHILD(i) return 2i + 1
(remember, in pseudo code indexing starts from 1)

Implementation of Heaps Using One-Dimensional Arrays (continued)

- Heap is balanced tree by definition
  - Height is O(log *n*)
- Remove smallest/largest element in the heap
  - By definition, the root of the tree
- Replace the far-right node on lowest level *i*
  - Determine the correct location for the value moved to the root
- Maximum number of times to exchange a node with its smaller child is equal to the height

Heap sort

- Heap sort algorithm
  - Build a heap structure that contains *n* elements to be sorted
  - Remove the root, print it, and rebuild the heap

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

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

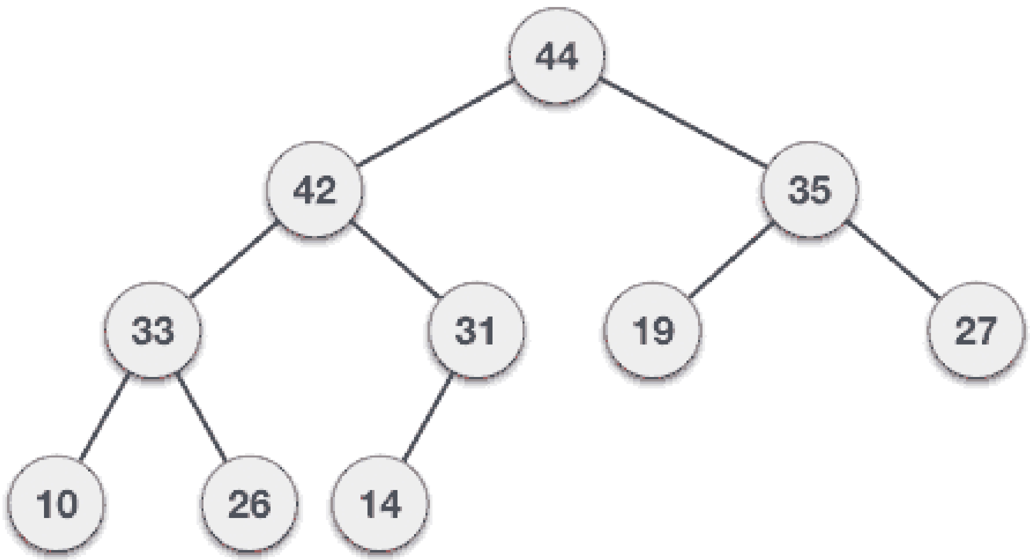
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Example / 2 (Max Heap Construction Algorithm)

Input 35 33 42 10 14 19 27 44 26 31

https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm

Example / 2 (Max Heap Deletion Algorithm)



https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm

Example / 5

- Letters = [B, D, F , X, Z, A, T, K]

Application of Heaps

- Heap sort
  - Both phases of heap sort are O(log *n*) executed *n* times; thus, heap sort is O(*n* log *n*)

Running time: After **n** iterations the Heap is empty every iteration involves a **swap** and a **max_heapify** operation; hence it takes **O(log n)** time

Overall **O(n log n)**

Application of Heaps

- Behavior same for average and worst case
- Unlike merge sort, sorts in place
- Unlike insertion sort, running time is O(*n log n*)
- Priority queue – Heap implementation uses priority as key field
- Heaps – *garbage collected storage* used in OS and programming languages

Time Complexity Calculation

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

- Build Heap → O(n)

Time Complexity Calculation / 2

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

- Heapify → O(log(n))

Time Complexity Calculation / 2

- Heap Sort → O(n*log(n))

Heap Sort has **O(nlog(n))** time complexities for all the cases (best case, average case and worst case).

- The **MAX - HEAPIFY** procedure, which runs in O(lg n) time, is the key to maintaining the max -heap property.
- The BUI**LD - MAX - HEAP** procedure, which runs in O(n) time, produces a max -heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in O(nlg n) time, sorts an array in place.