CS-2001
DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

HEAP

Heap

- A Heap is a special Tree-based data structure in which the **tree is a complete binary tree**.
- Heap is a special case of **balanced binary tree** data structure where the *root-node key is compared with its children* and arranged accordingly.
- Generally, Heaps can be of two types:
  - **Max-Heap:**
  - **Min-Heap:**

Heap

Max-Heap:

- In a Max-Heap, the key present at the root node must be maximum among the keys present at all of its children.
- The same property must be recursively true for all sub-trees in that Binary Tree.
- If **α** has child node **β** then,

$$key(\alpha) \geq key(\beta)$$

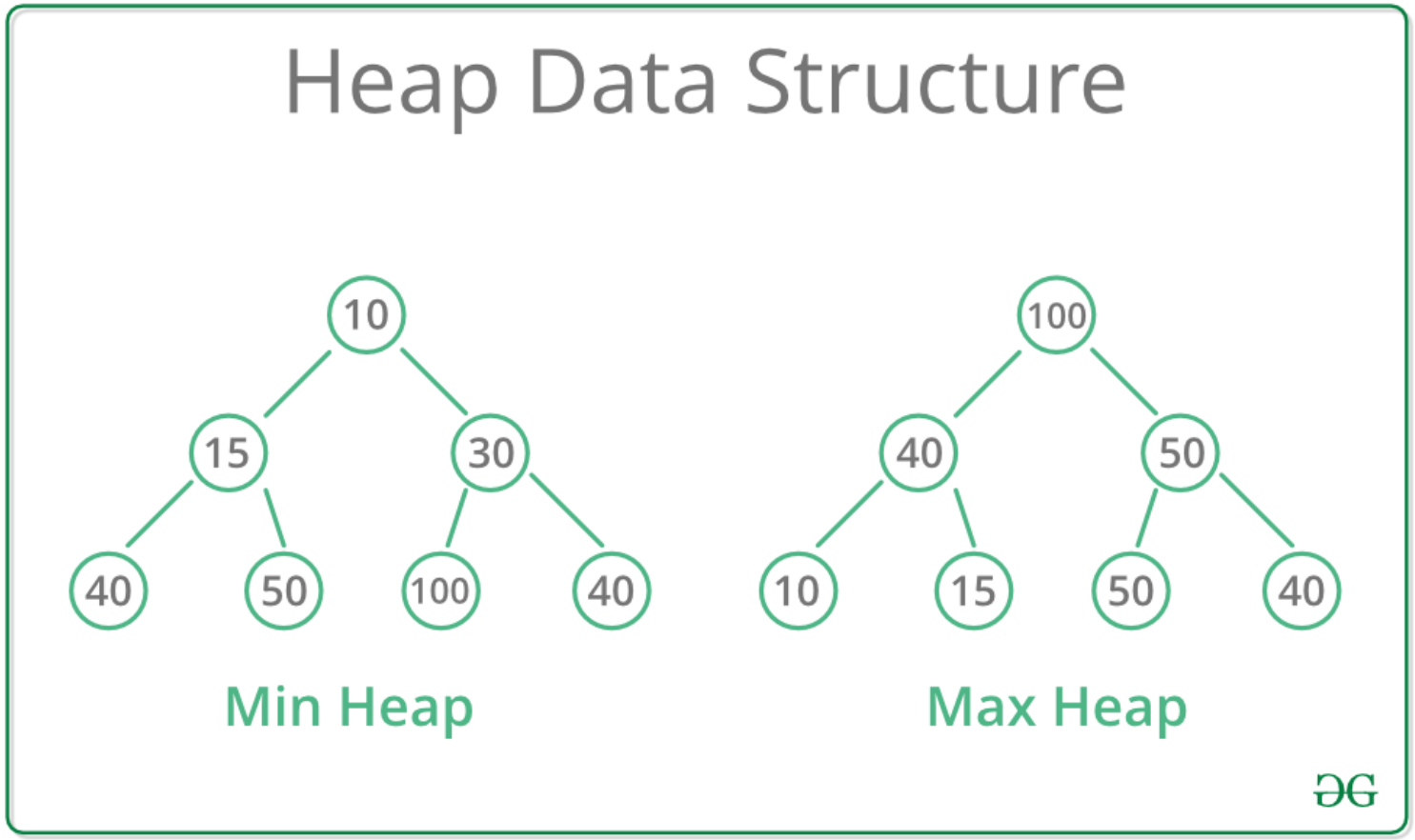
Heap

Min-Heap:

- In a Min-Heap, the key present at the root node must be minimum among the keys present at all of its children.
- The same property must be recursively true for all sub-trees in that Binary Tree.
- If **α** has child node **β** then,

$$key(\alpha) \leq key(\beta)$$

Examples


Heap Data Structure
10
15
30
40
50
100
40
Min Heap
100
40
50
10
15
50
40
Max Heap

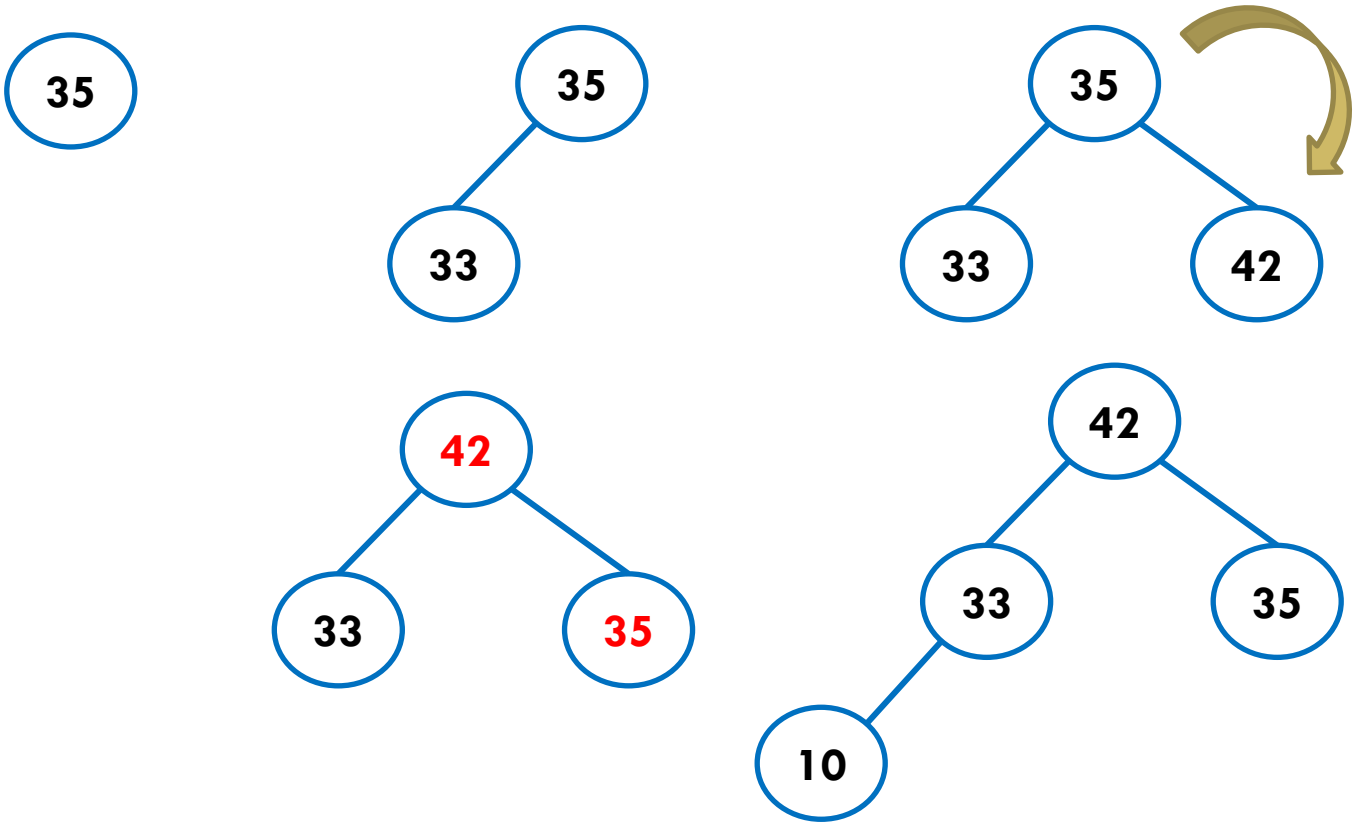
MAX HEAP

Max Heap Construction Algorithm

Step 1 – Create a new node at the end of the heap.
Step 2 – Assign a new value to the node.
Step 3 – Compare the value of this child node with its parent.
Step 4 – If the value of the parent is less than a child, then swap them.
Step 5 – Repeat steps 3 & 4 until Heap property holds.

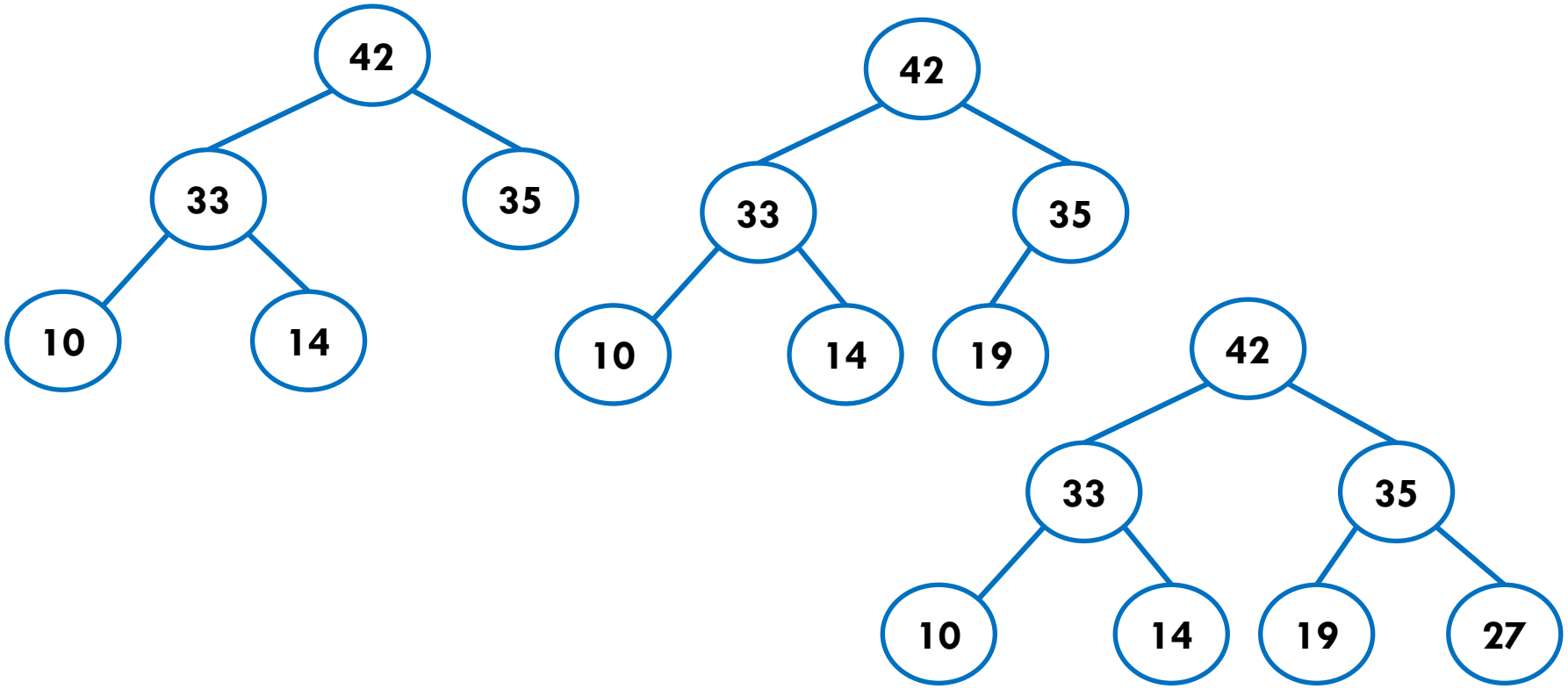
Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31



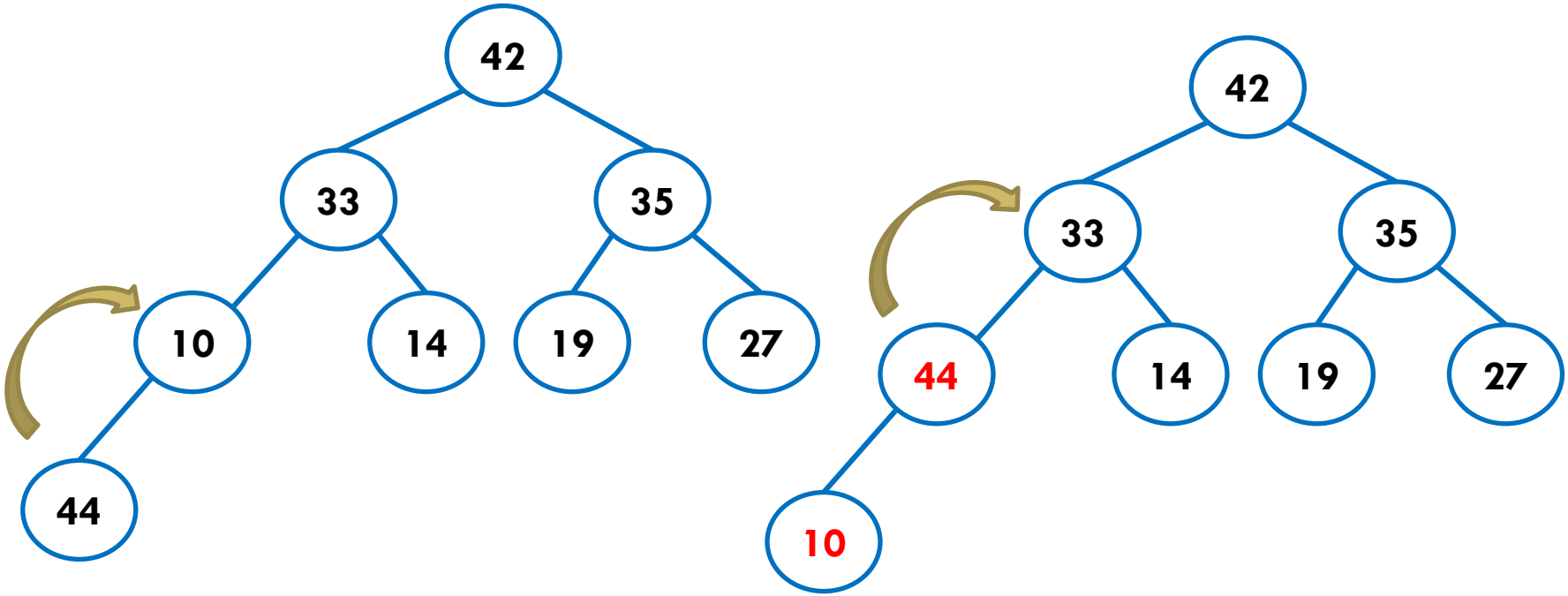
Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31



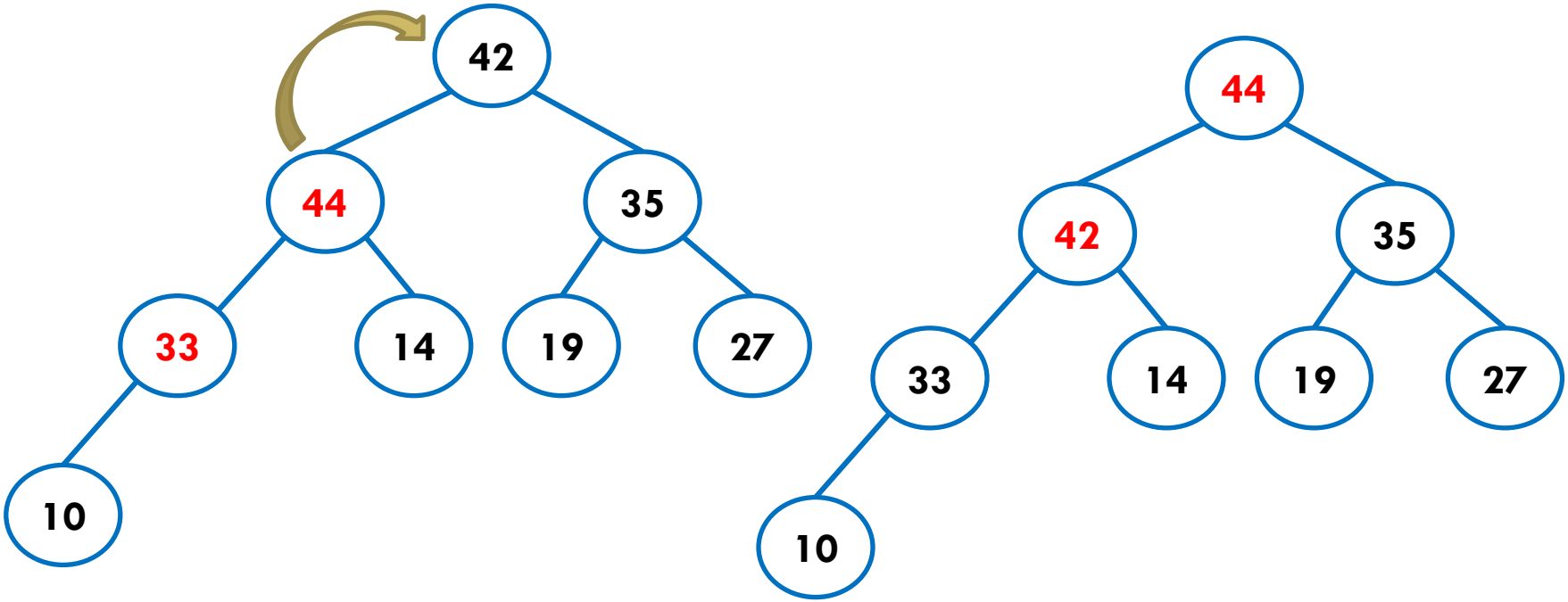
Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31



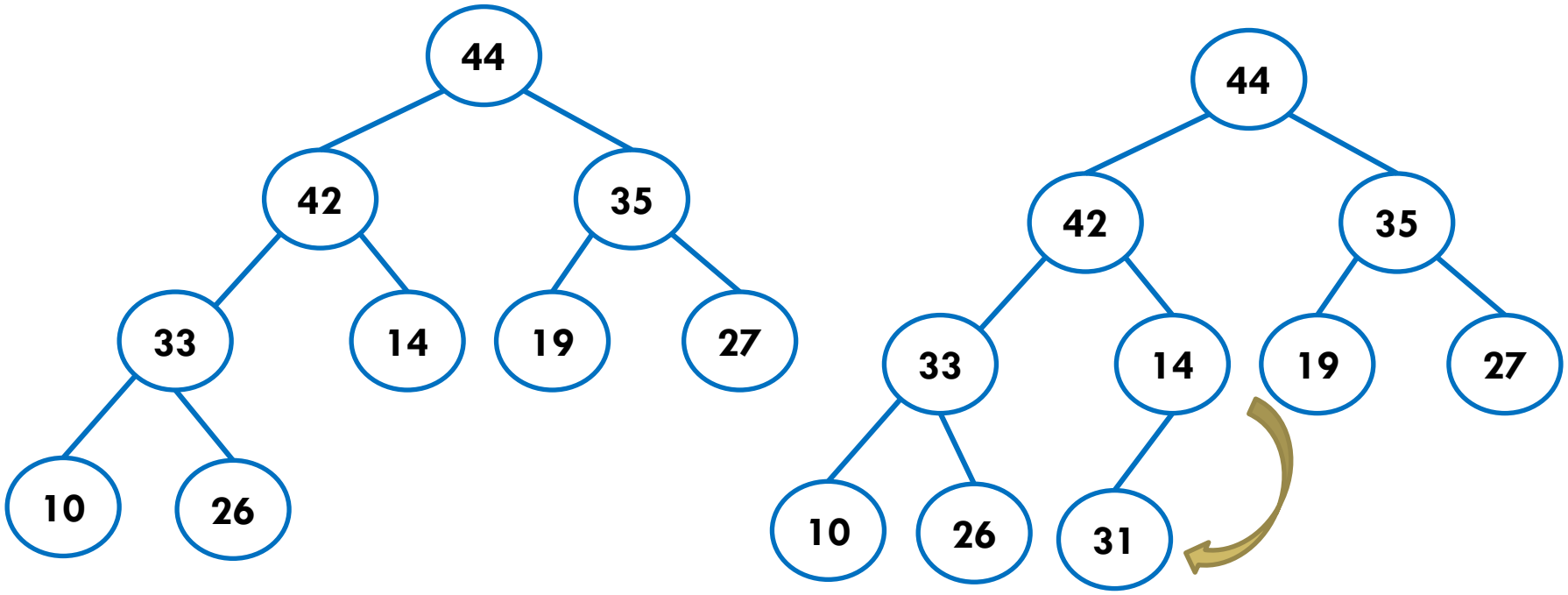
Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31



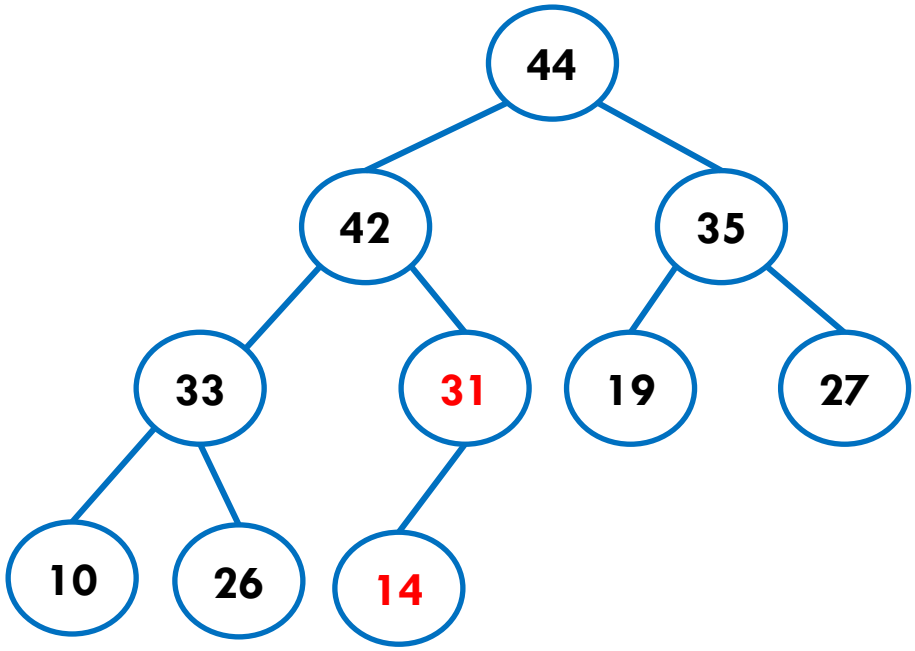
Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31



Example … Max Heap

For Input → 35 33 42 10 14 19 27 44 26 31

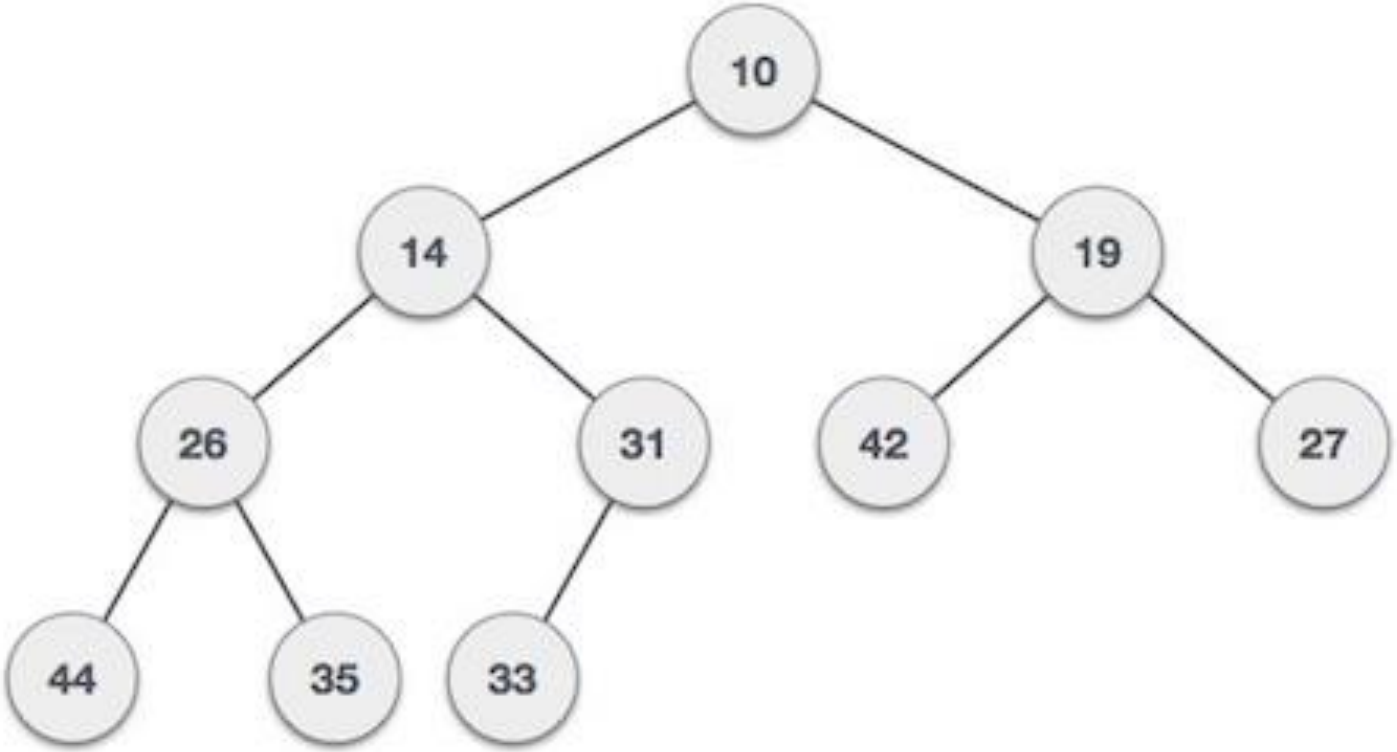


Example … Max Heap


```
For Input → 35 33 42 10 14 19 27 44 26 31
```

Example … Min Heap

For Input → 35 33 42 10 14 19 27 44 26 31



Max Heap … Deletion

Step 1 – Remove the root node.

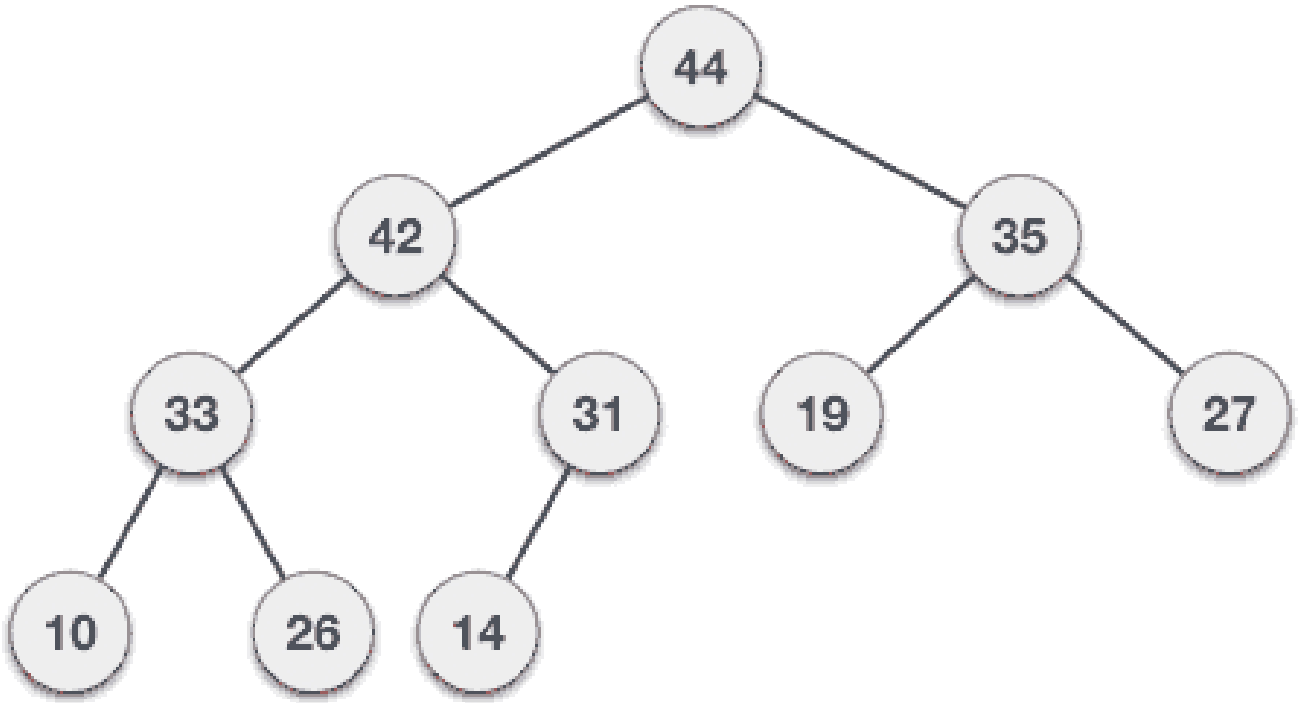
Step 2 – Move the last element of the last level to the root.

Step 3 – Compare the value of this child node with its parent.

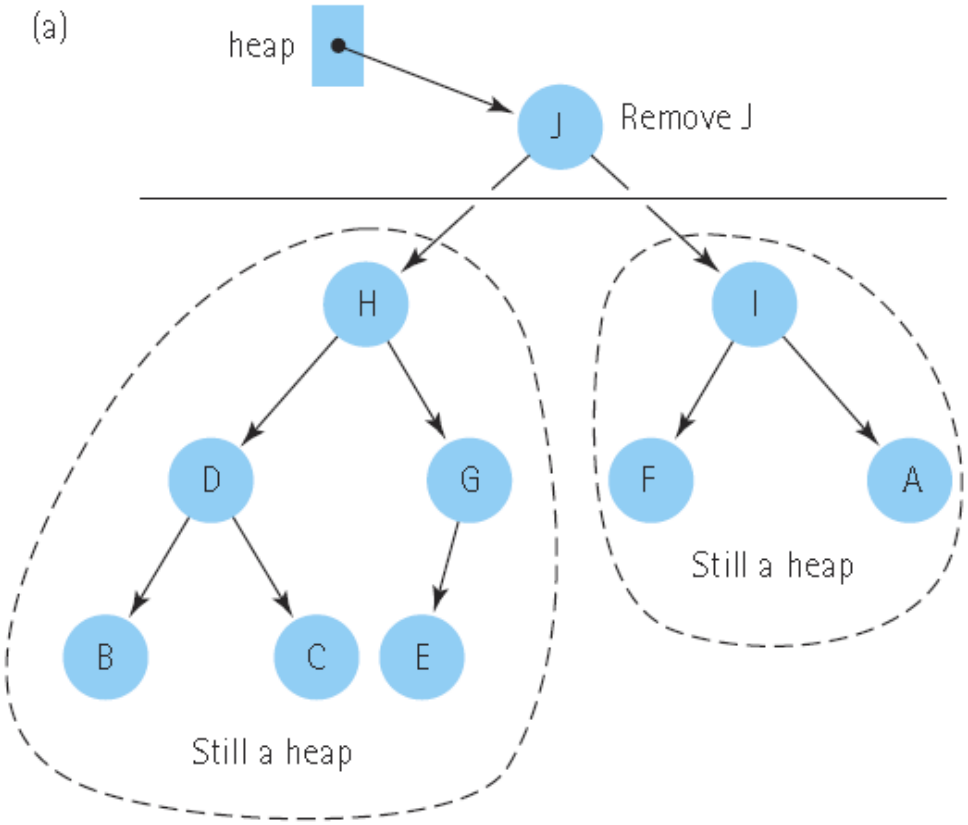
Step 4 – If the value of the parent is less than the child, then swap them. Swap the replacement node with the **largest child** node.

Step 5 – Repeat steps 3 & 4 until the Heap property holds.

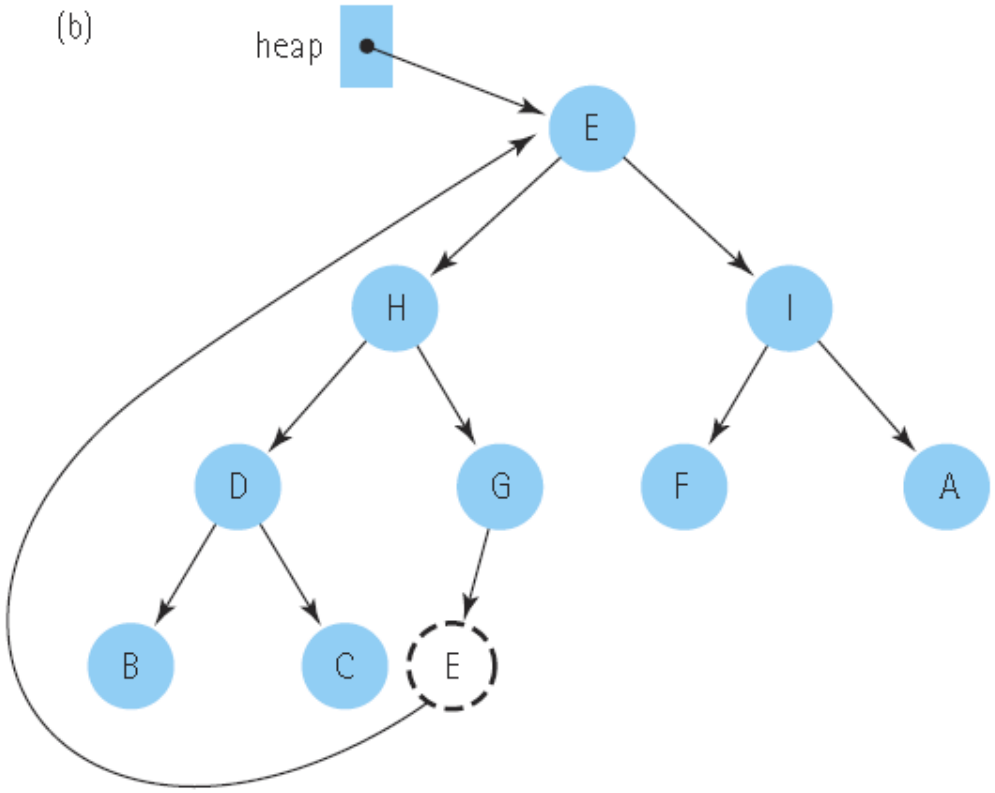
Max Heap … Deletion



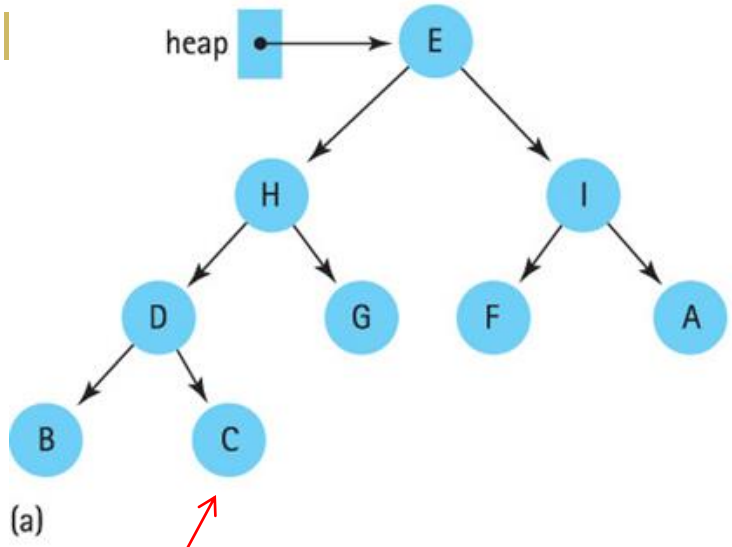
Max Heap … Deletion



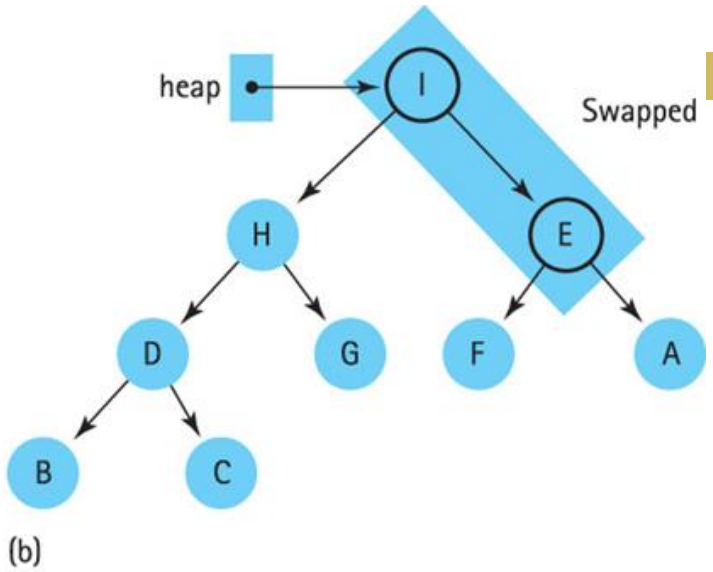
Max Heap … Deletion

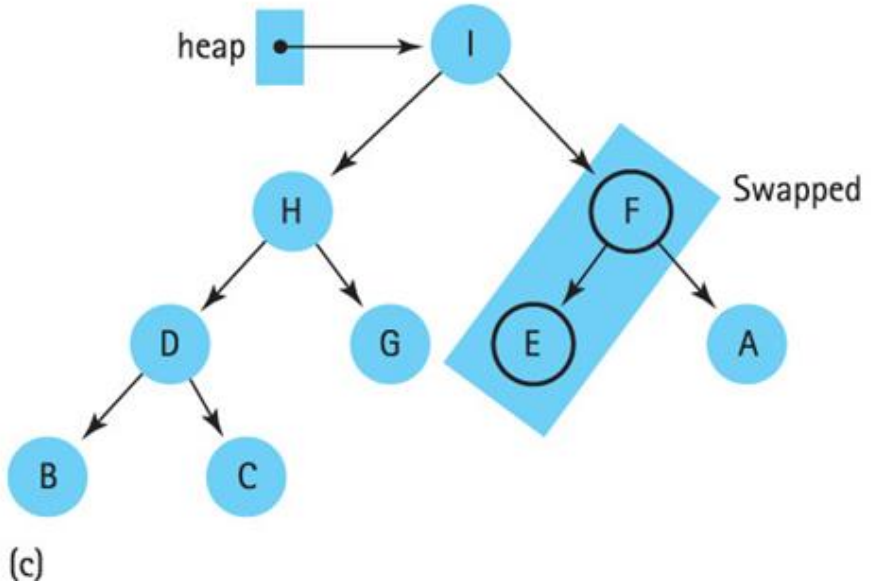


Max Heap … Deletion

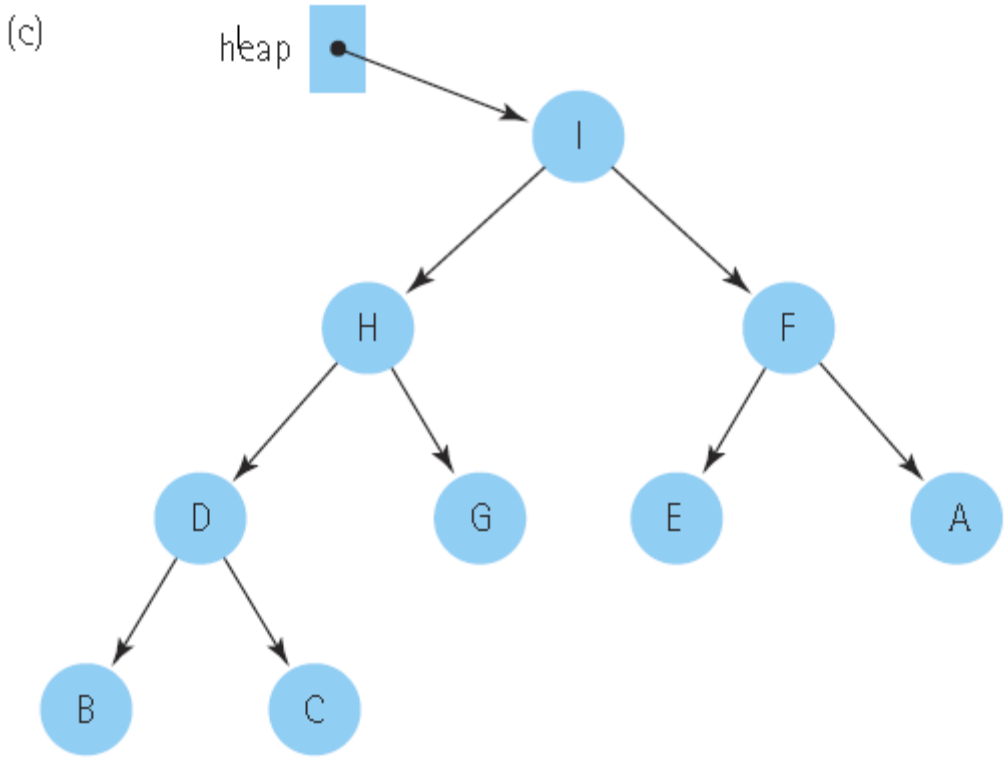


bottom



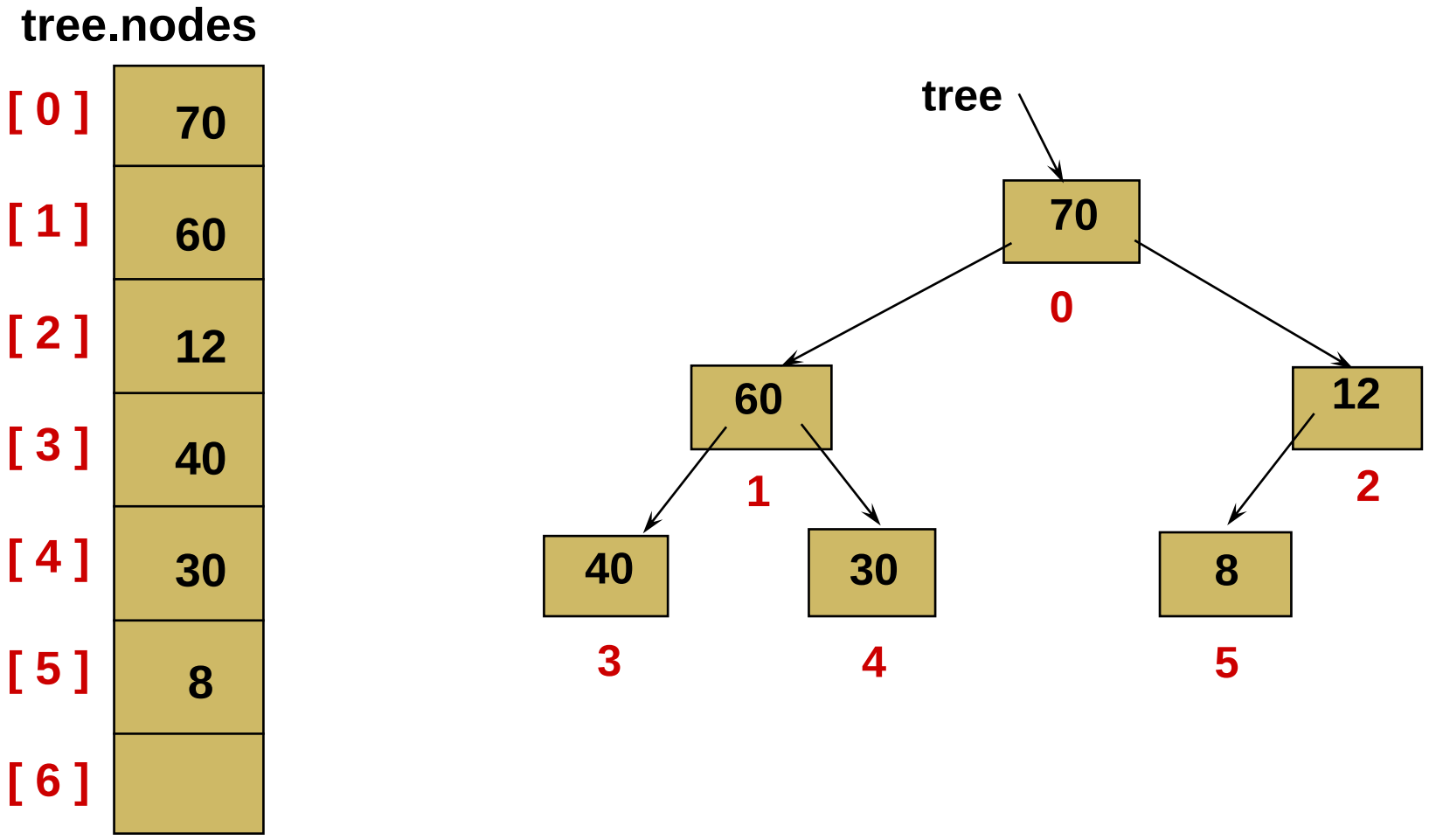


Max Heap … Deletion

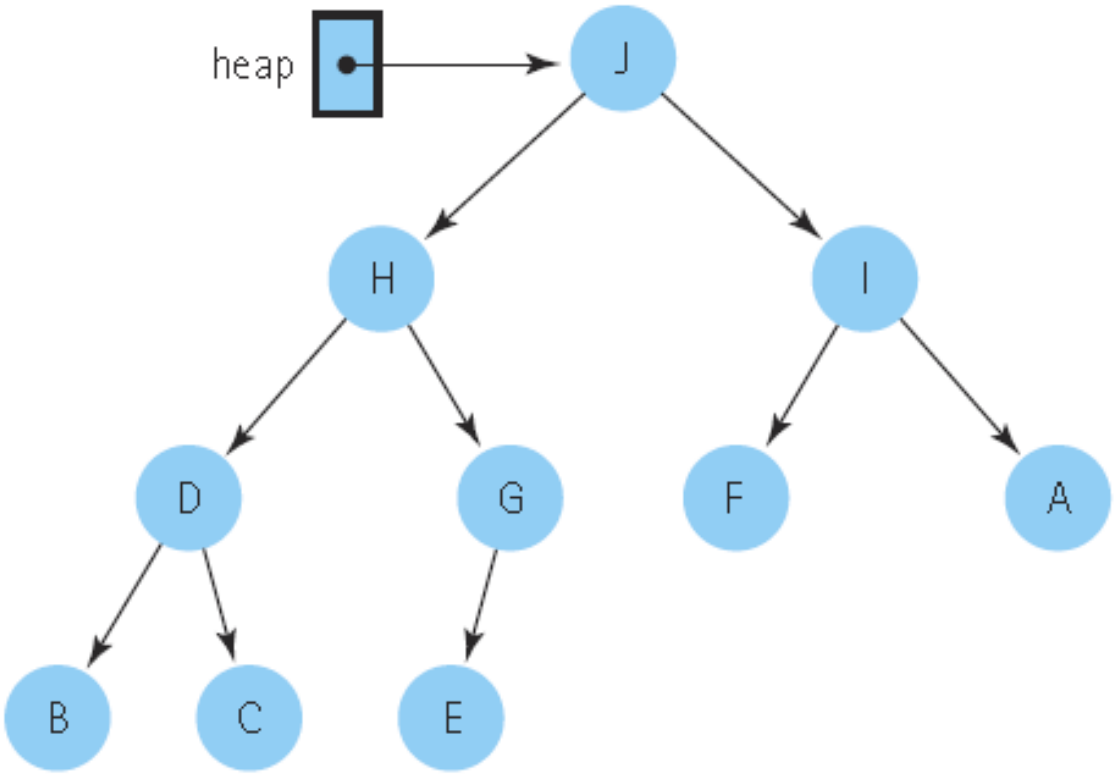


HEAP IMPLEMENTATION

Heap … Array Implementation



Heap … Array Implementation

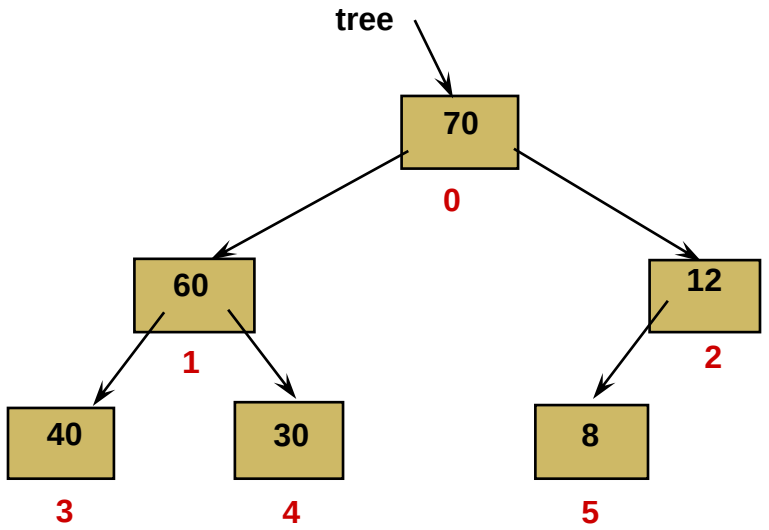


heap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

Heap … Array Implementation

- For any node **`tree.nodes[index]`**
  - its left child is in
    - **`tree.nodes[index*2 + 1]`**
  - right child is in
    - **`tree.nodes[index*2 + 2]`**
  - its parent is in
    - **`tree.nodes[(index - 1)/2]`**



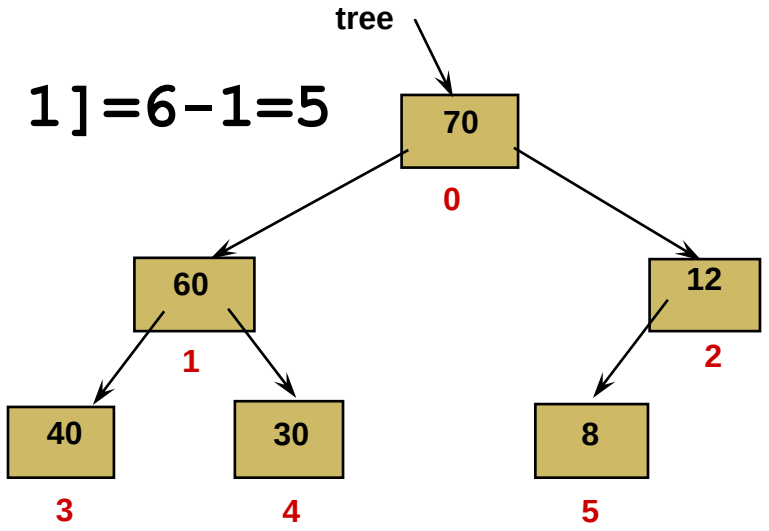
Heap … Array Implementation

Leaf nodes:

`tree.nodes[numElements/2]=6/2=3`

to

`tree.nodes[numElements - 1]=6-1=5`



Heap … Array Implementation


```
#include <iostream>
using namespace std;
// To heapify a subtree rooted with node i which is an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i){
        int largest = i;        // Initialize largest as root
        int l = 2 * i + 1;      // left = 2*i + 1
        int r = 2 * i + 2;      // right = 2*i + 2
        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
                largest = l;
        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
                largest = r;
        // If largest is not root
        if (largest != i) {
                swap(arr[i], arr[largest]);
                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
        }
}
```

Heap … Array Implementation


```
// Function to build a Max-Heap from the given array
void buildHeap(int arr[], int n) {
        // Index of last non-leaf node
        int startIdx = (n / 2) - 1;
        // Perform reverse level order traversal from last
        //non-leaf node and heapify each node
        for (int i = startIdx; i >= 0; i--) {
                heapify(arr, n, i);
        }
}
```

Heap … Array Implementation


```
// A utility function to print the array representation
// of Heap
void printHeap(int arr[], int n) {
        cout << "Array representation of Heap is:\n";
        for (int i = 0; i < n; ++i)
                cout << arr[i] << " ";
        cout << "\n";
}
```

Heap … Array Implementation


```
int main() {
        int arr[] = { 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17 };
        int n = sizeof(arr) / sizeof(arr[0]);
        buildHeap(arr, n);
        printHeap(arr, n);
        return 0;
}
```

Heap … Array Implementation


```
#include <iostream>
using namespace std;
// To heapify a subtree rooted with node i which is an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i){
        int largest = i;        // Initialize largest as root
        int l = 2 * i + 1;      // left = 2*i + 1
        int r = 2 * i + 2;      // right = 2*i + 2
        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
                largest = l;
        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
                largest = r;
        // If largest is not root
        if (largest != i) {
                swap(arr[i], arr[largest]);
                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
        }
}
```

```
// Function to build a Max-Heap from the given array
void buildHeap(int arr[], int n) {
        // Index of last non-leaf node
        int startIdx = (n / 2) - 1;
        // Perform reverse level order traversal from last
        //non-leaf node and heapify each node
        for (int i = startIdx; i >= 0; i--) {
                heapify(arr, n, i);
        }
}
```

```
int main() {
        int arr[] = { 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17 };
        int n = sizeof(arr) / sizeof(arr[0]);
        buildHeap(arr, n);
        printHeap(arr, n);
        return 0;
}
```

Reading Materials

- Schaum's Outlines: Chapter # 7
- D. S. Malik: Chapter # 11
- Nell Dale: Chapter # 8