

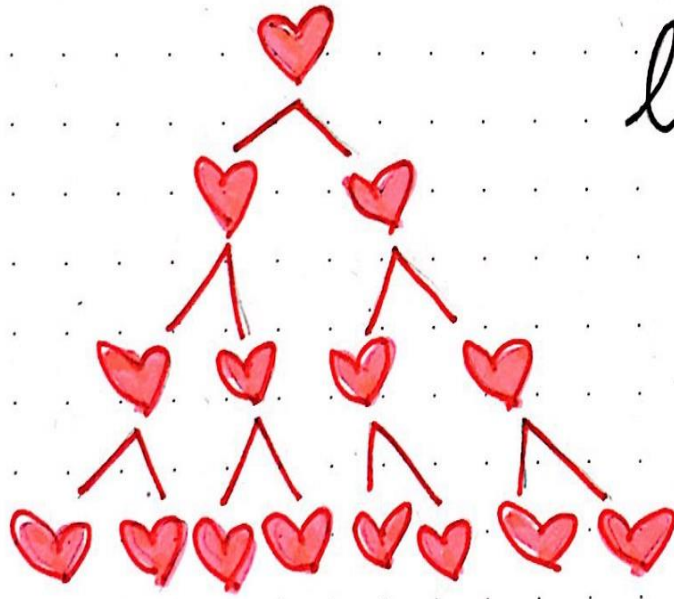


COMPUTER SCIENCE DEPARTMENT FACULTY OF  
ENGINEERING AND TECHNOLOGY

COMP2321

## Data Structures

# Chapter 6: Heap



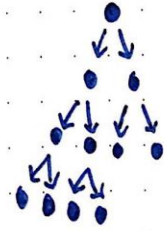
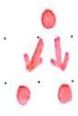
learning to love

HEARS

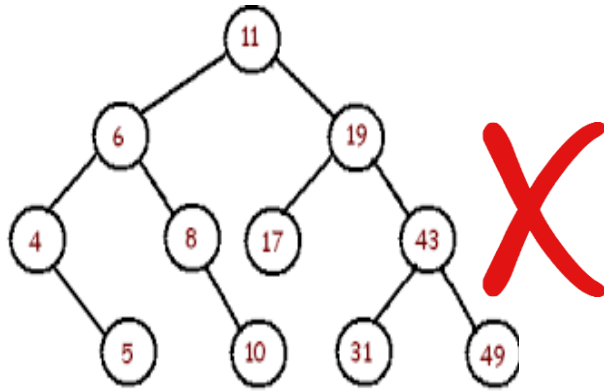
HEAPIFY

ALL THE THINGS!

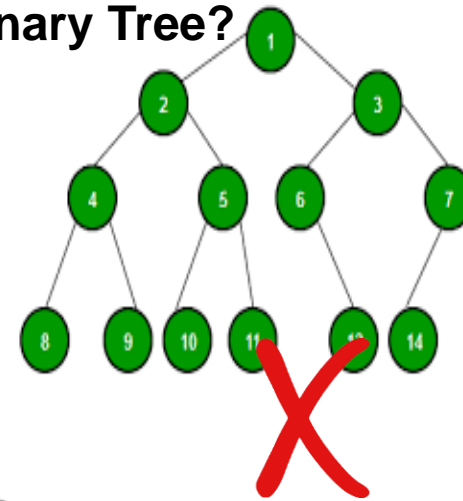
(with heap sort)



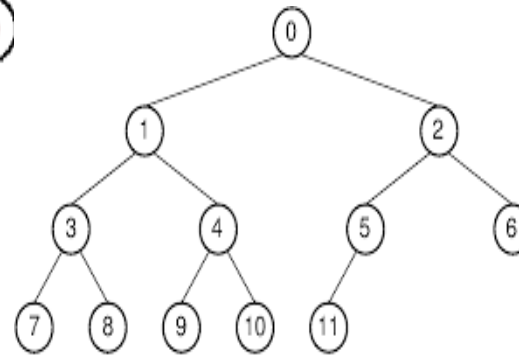
Which of the following is Complete Binary Tree?



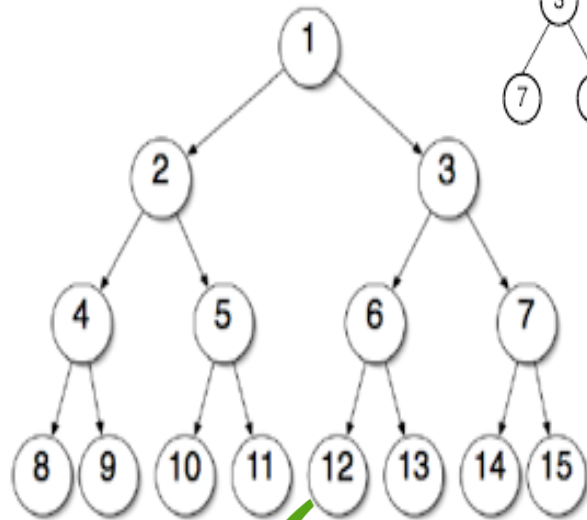
A



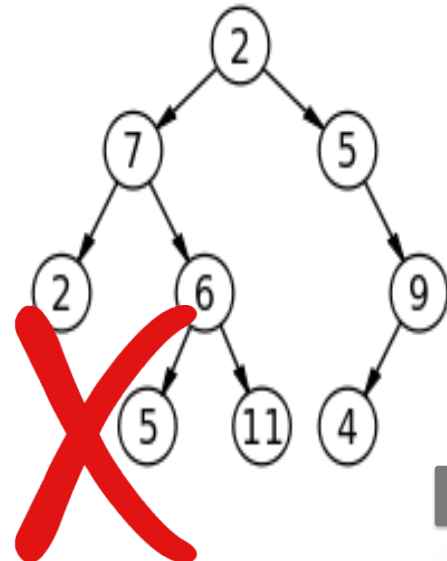
B



C



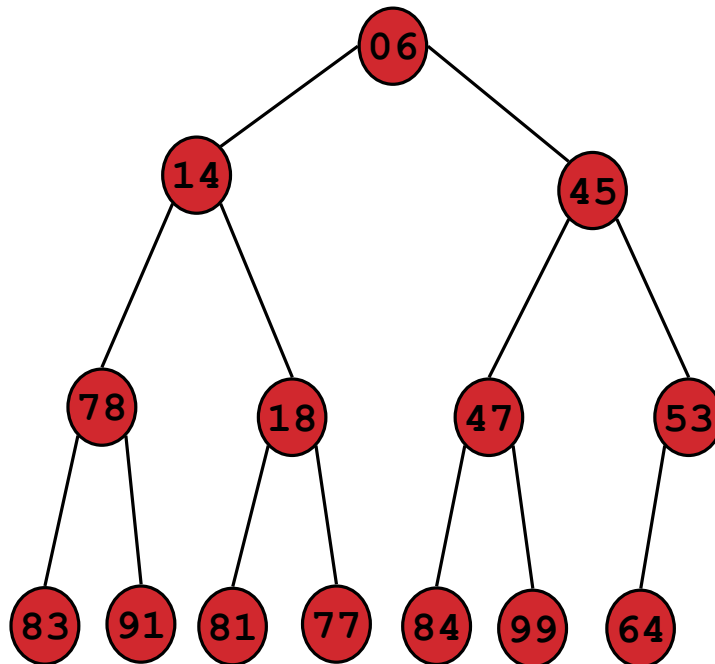
D



E

# Binary Heap: Definition

- Binary heap: it must follow the two properties:
  - Complete binary tree.
    - filled on all levels, except last, where filled from left to right
  - Order property, e.g. Min-heap or Max-heap
    - Min-heap: parent is less than children
    - Max-heap: parent is greater than children

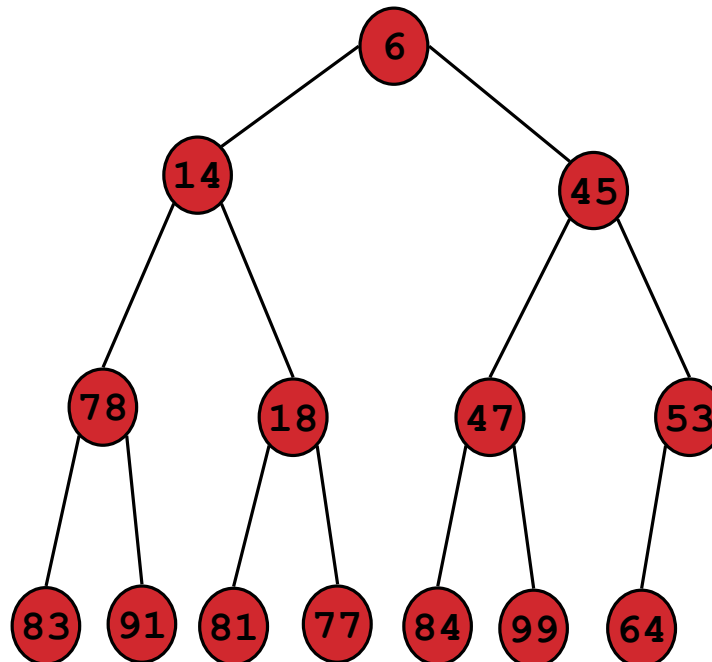


## Notes :

- 1) Node keys could be repeated
- 2) Left child may be greater than right child and vice versa

# Binary Heap: Properties

- Properties.
  - Minimum element is in the root.
  - Heap with  $N$  elements has height =  $\lfloor \log_2 N \rfloor$ .

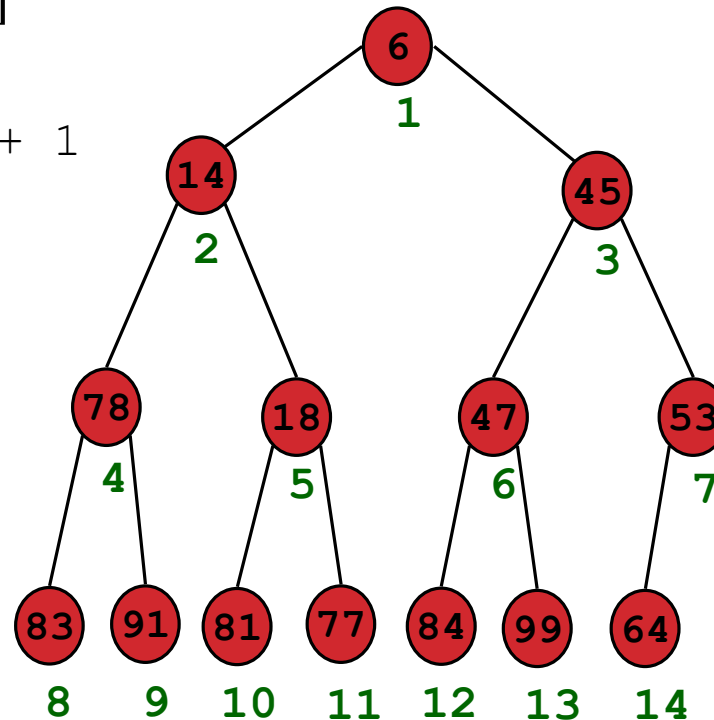


**N = 14**  
**Height = 3**

# Binary Heaps: Array Implementation

- Implementing binary heaps.
  - Use an array: no need for explicit parent or child pointers.

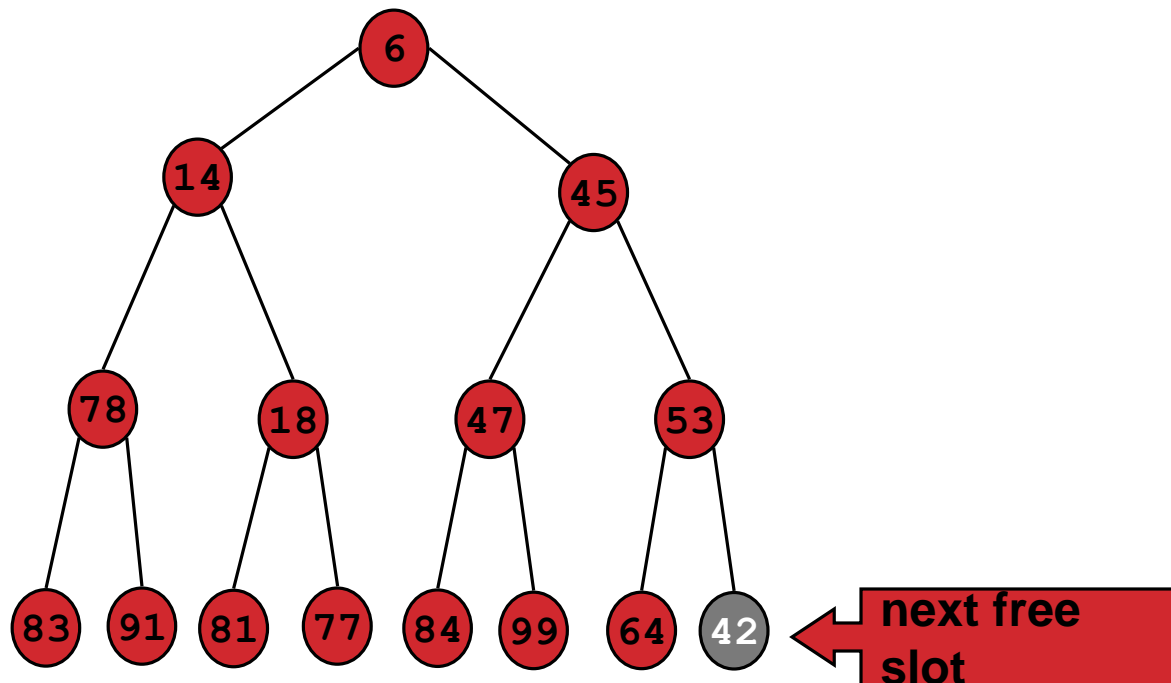
- $\text{Parent}(i) = \lfloor i/2 \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$



1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	14	45	78	18	47	53	83	91	81	77	84	99	64

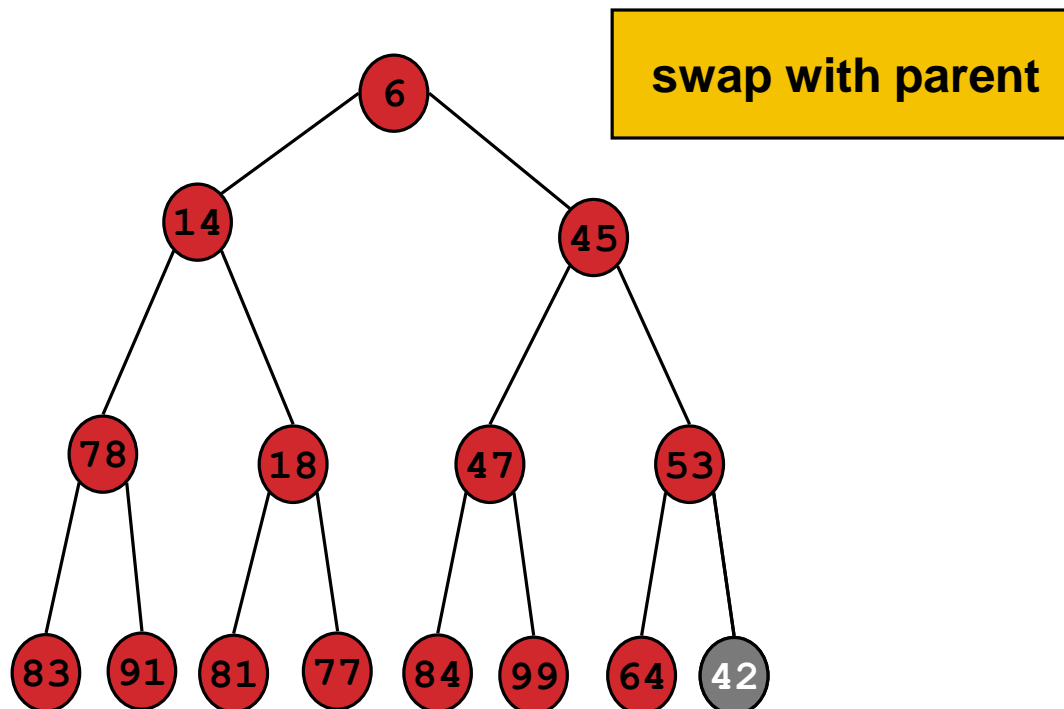
# Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence



# Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence

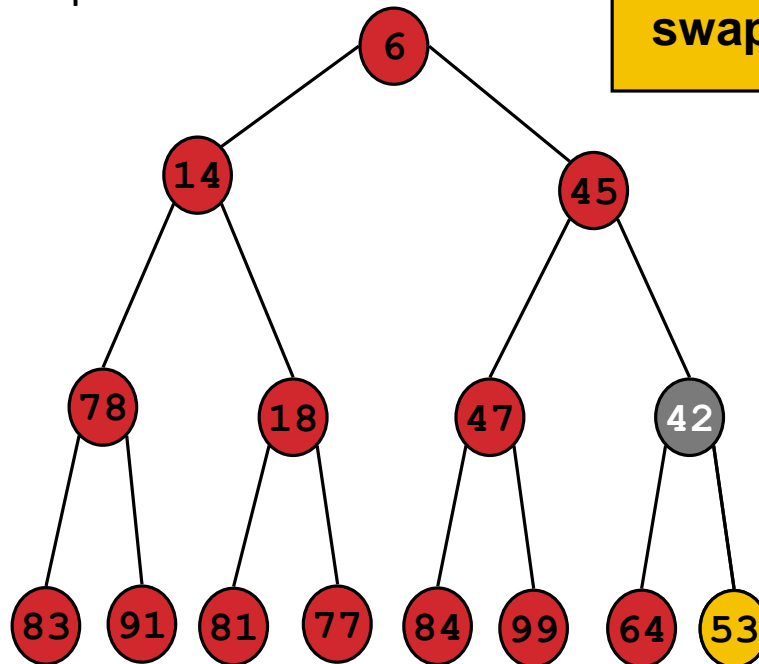




# Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of in

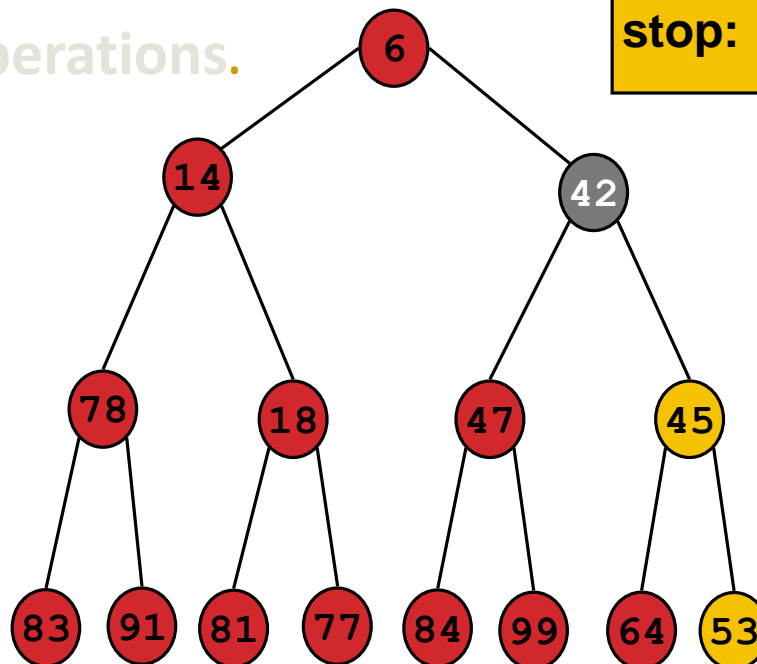
**swap with parent**



# Binary Heap: Insertion

Insert element  $x$  into heap.

- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence
- $O(\log N)$  operations.



stop: heap ordered

Given Data:

44, 33, 77 ,11, 55, 88, 66, 22

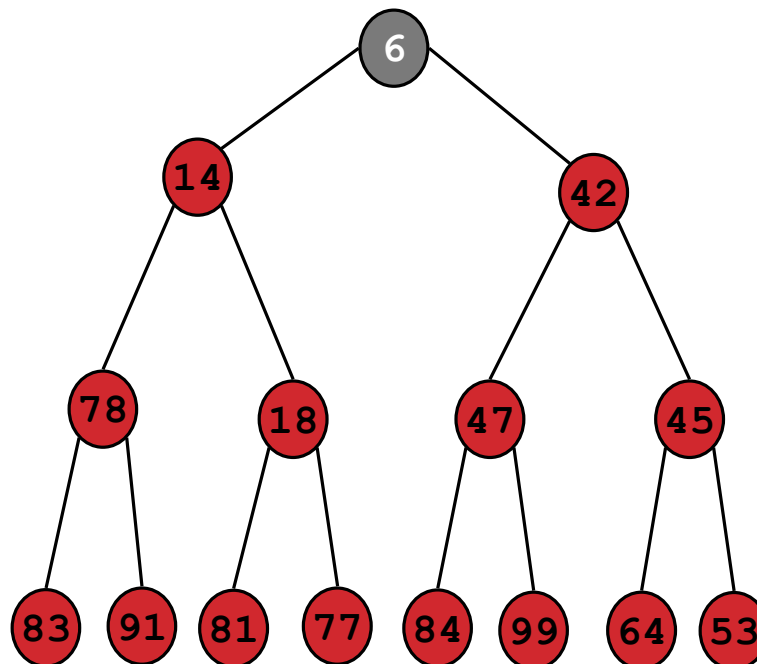
Build max heap tree ? (show all works).

Solved At board

# Binary Heap: Delete Min

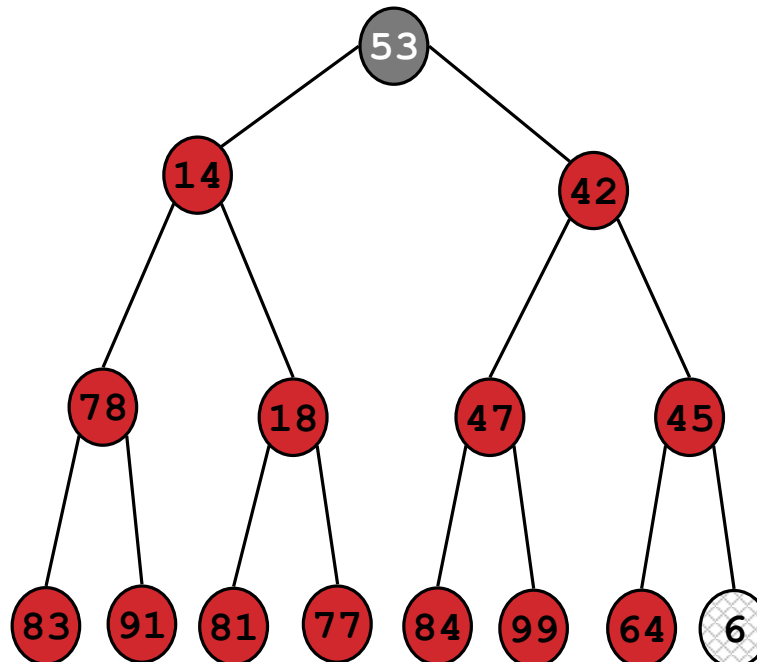
Delete minimum element from heap.

- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



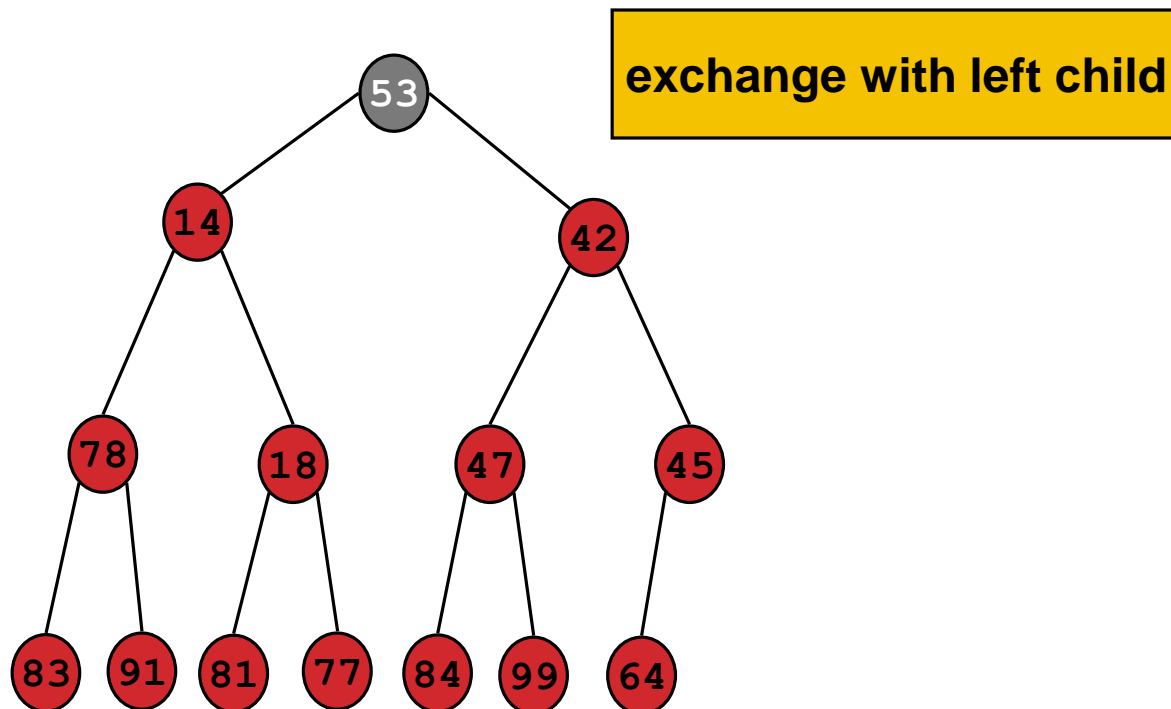
# Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



# Binary Heap: Delete Min

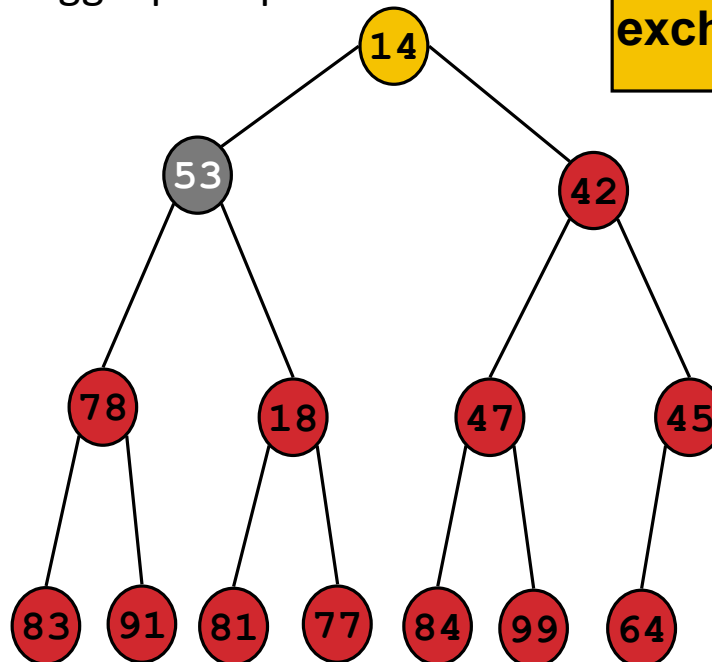
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



# Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subord

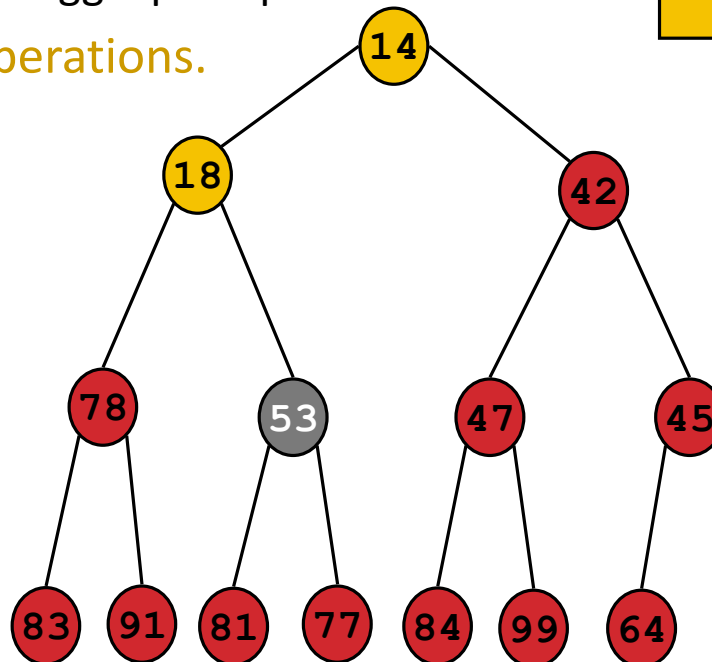
**exchange with right child**



# Binary Heap: Delete Min

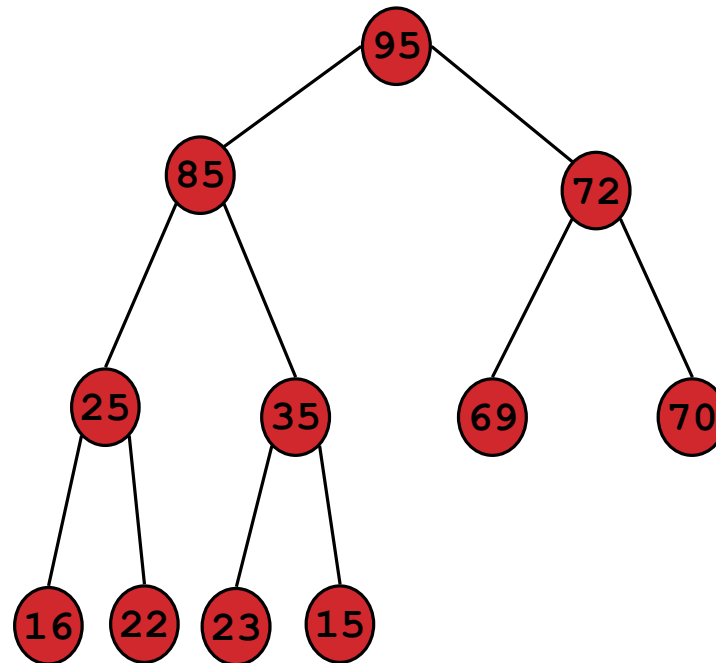
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinates
  - $O(\log N)$  operations.

**stop: heap ordered**





## Example: Delete root (95)

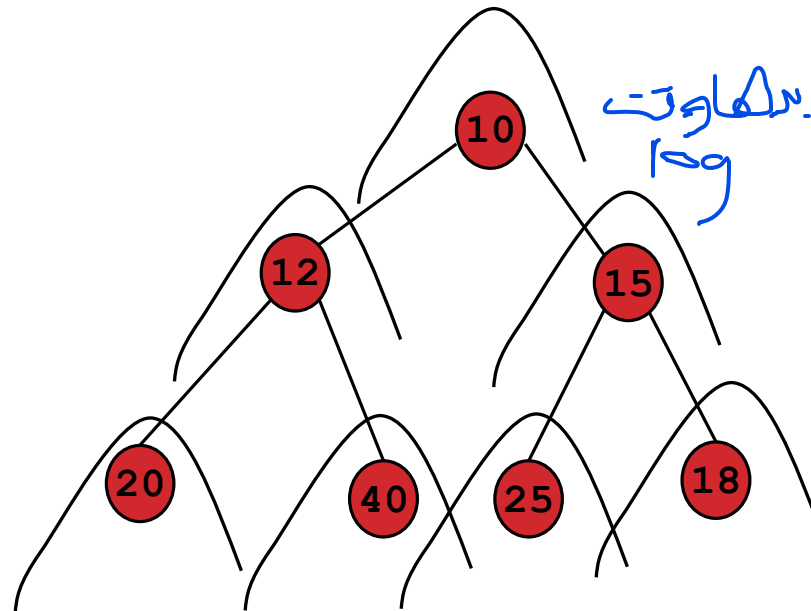


Solved At board

# Heapify *new node* *عملية Insert لكل*

Looking at every node as it's heap from bottom to up

This take less time  $O(\log n)$ , since creation of heap is costing  $O(n \log n)$



*كل*  
*وهدة 4 5 15 12 به اوقات*  
*log*

```

1  #include <stdio.h>
2  #include <limits.h>
3
4
5  int FRONT=1;
6  int size=0;
7  //C implementation of Min Heap
8  struct MinHeap {
9      int maxsize;
10     int Heap[];
11 } heap;
12
13 struct MinHeap heap;
14 void MinHeap(int maxsize)
15 {
16
17     heap.maxsize = maxsize;
18     int Heap[maxsize + 1];
19     Heap[0] =INT_MIN;
20 }
21

```

```

// Function to return the position of
// the parent for the node currently
// at pos
int parent(int pos)
{
    return pos / 2;
}

// Function to return the position of the
// left child for the node currently at pos
int leftChild(int pos)
{
    return (2 * pos);
}

// Function to return the position of
// the right child for the node currently
// at pos
int rightChild(int pos)
{
    return (2 * pos) + 1;
}

// Function that returns true if the passed
// node is a leaf node

```

```

46 // Function that returns true if the passed
47 // node is a leaf node
48 int isLeaf(int pos)
49 {
50     if (pos >= (size / 2) && pos <=size) {
51         return 1;
52     }
53     return 0;
54 }
55
56 // Function to swap two nodes of the heap
57 void swap(int fpos, int spos)
58 {
59     int tmp;
60     tmp = heap.Heap[fpos];
61     heap.Heap[fpos] = heap.Heap[spos];
62     heap.Heap[spos] = tmp;
63 }
64
65 // Function to heapify the node at pos

```

```
95 // Function to insert a node into the heap
96 void insert(int element)
97 {
98
99     if (size >= heap.maxsize) {
100         return;
101     }
102     heap.Heap[++size] = element;
103     int current = size;
104
105     while (heap.Heap[current] < heap.Heap[parent(current)]) {
106         swap(current, parent(current));
107         current = parent(current);
108     }
109 }
110
```

```

65 // Function to heapify the node at pos
66 void minHeapify(int pos)
67 {
68
69     // If the node is a non-leaf node and greater
70     // than any of its child
71     if (!isLeaf(pos)) {
72         if (heap.Heap[pos] > heap.Heap[leftChild(pos)]
73             || heap.Heap[pos] > heap.Heap[rightChild(pos)]) {
74
75             // Swap with the left child and heapify
76             // the left child
77             if (heap.Heap[leftChild(pos)] < heap.Heap[rightChild(pos)]) {
78                 swap(pos, leftChild(pos));
79                 minHeapify(leftChild(pos));
80             }
81
82             // Swap with the right child and heapify
83             // the right child
84             else {
85                 swap(pos, rightChild(pos));
86                 minHeapify(rightChild(pos));
87             }
88         }
89     }
90 }
91

```

```
120 // Function to build the min heap using
121 // the minHeapify
122 void minHeap()
123 {
124     for (int pos = (size / 2); pos >= 1; pos--) {
125         minHeapify(pos);
126     }
127 }
128
129
```

# Heapsort

## Heap Sort Algorithm for sorting :

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1.

Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.



# Heapsort

## Time Complexity:

- Time complexity of heapify is  $O(\log N)$ .
- Time complexity of create And BuildHeap() is  $O(N)$
- overall time complexity of Heap Sort is  $O(N \log N)$ .
- Heapsort.
  - Insert  $N$  items into binary heap.
  - Perform  $N$  delete-min operations.
  - $O(N \log N)$  sort.
  - No extra storage.

# Heapsort

```
void sort(int arr[], int n)
```

```
{
```

```
    // Build heap (rearrange array)
```

```
    //for (int i = n / 2 - 1; i >= 0; i--)
```

```
        //heapify(arr, n, i);
```

```
    // One by one extract an element from heap
```

```
    for (int i=n-1; i>0; i--)
```

```
    {
```

```
        // Move current root to end
```

```
        int temp = arr[0];
```

```
        arr[0] = arr[i];
```

```
        arr[i] = temp;
```

```
        // call max heapify on the reduced heap
```

```
        heapify(arr, i, 0);
```

```
    }
```

```
}
```

**THE  
END**