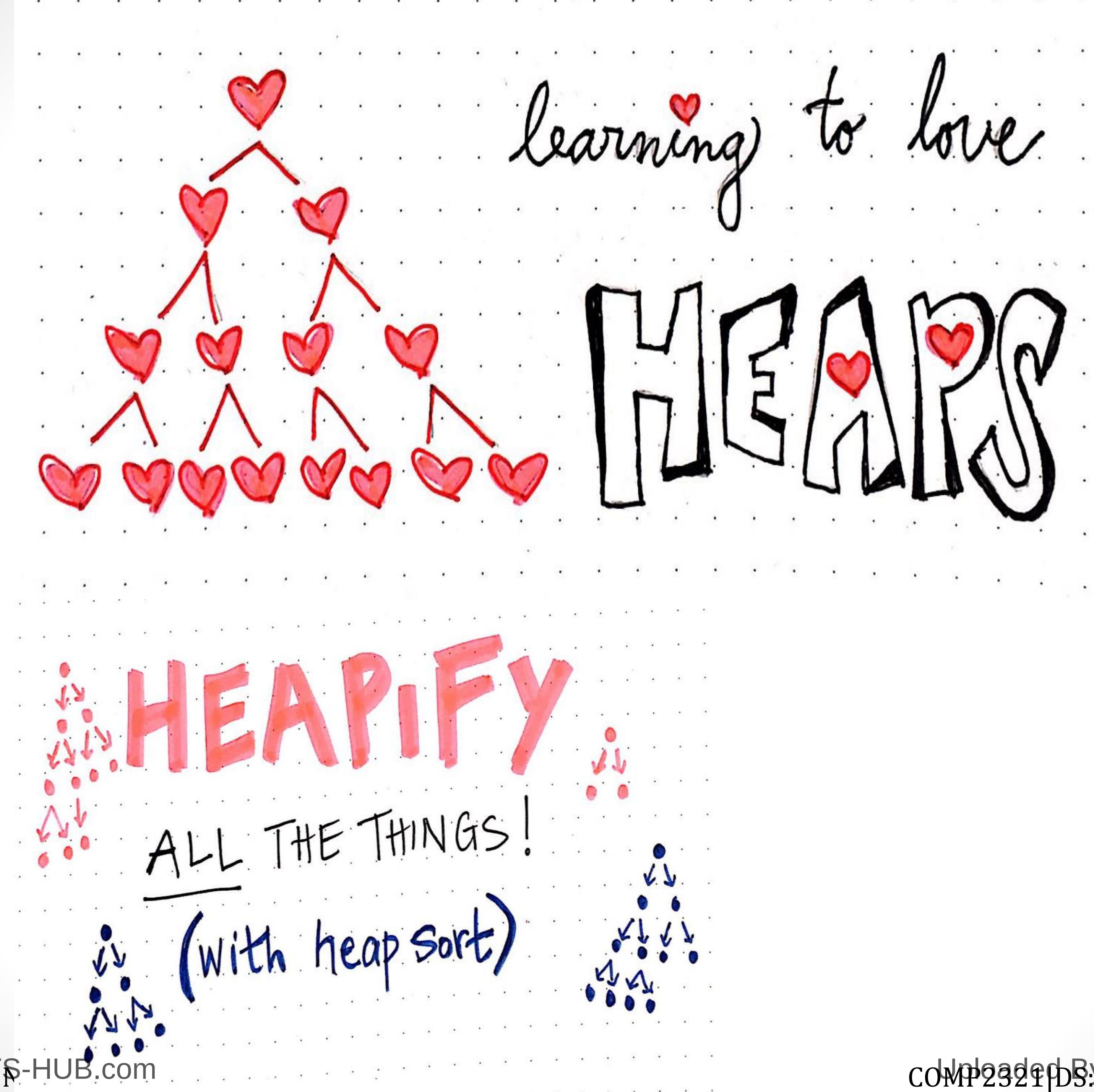
BIRZEIT UNIVERSITY

Computer Science Department Faculty of Engineering and Technology

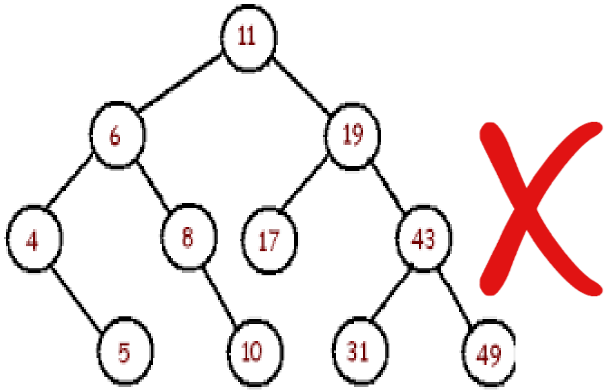
Comp2321

Data Structures

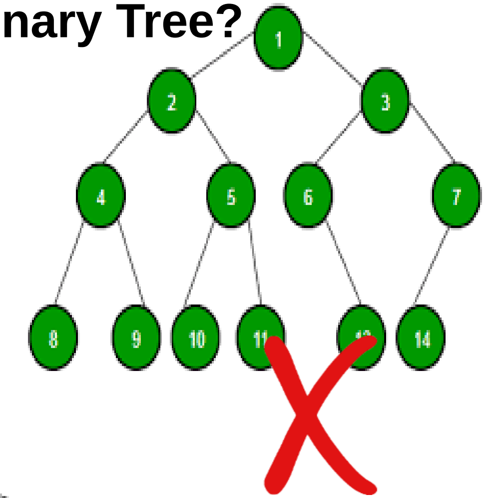
Chapter 6: Heap


learning to love
HEAPS
HEAPIFY
ALL THE THINGS!
(with heap sort)

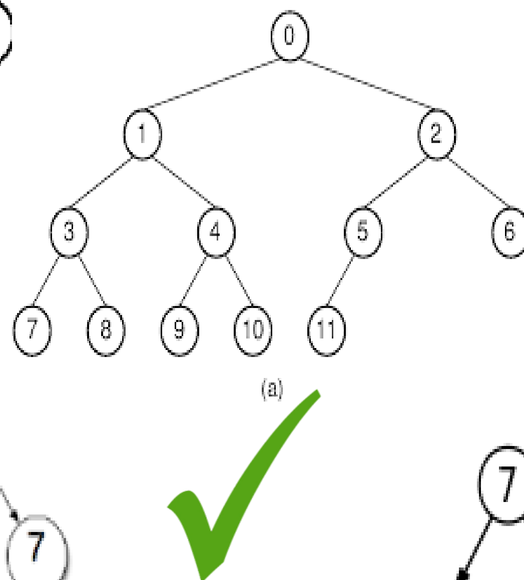
Which of the following is Complete Binary Tree?



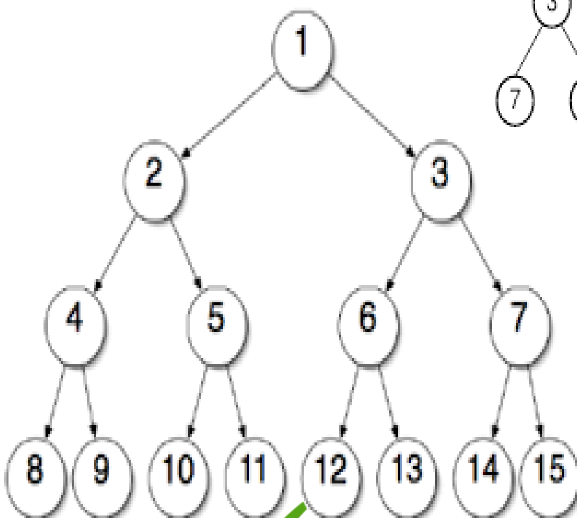
A



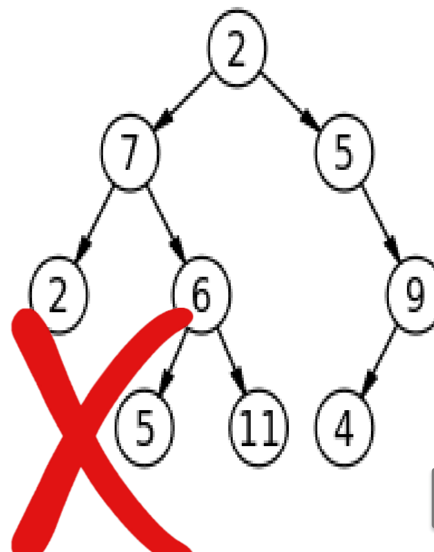
B



C



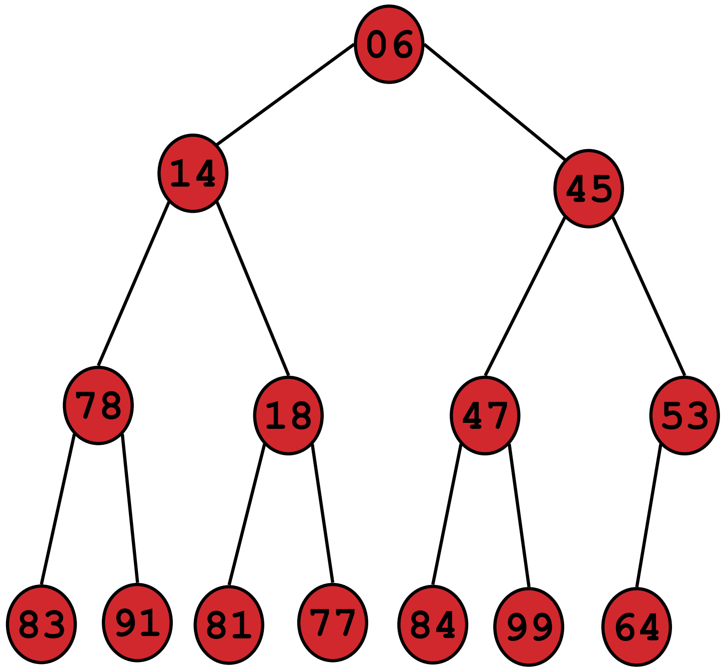
D



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

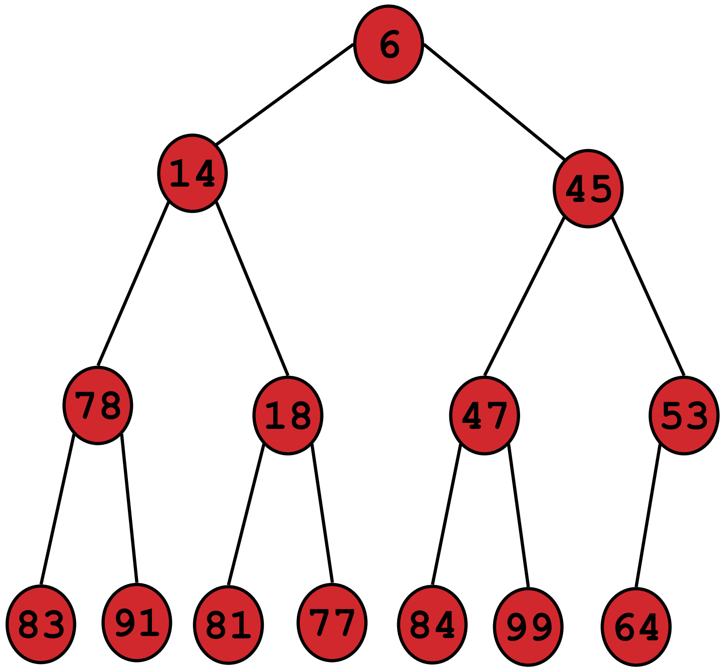


Notes :

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

Binary Heap: Properties

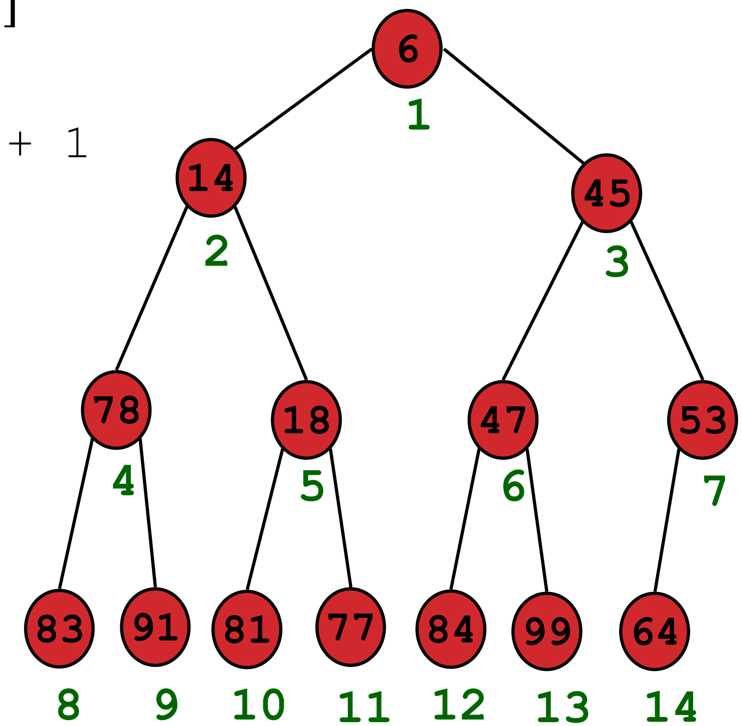
- Properties.
  - Minimum element is in the root.
  - Heap with N elements has height = $\lfloor log_2 N \rfloor$.



N = 14
Height = 3

Binary Heaps: Array Implementation

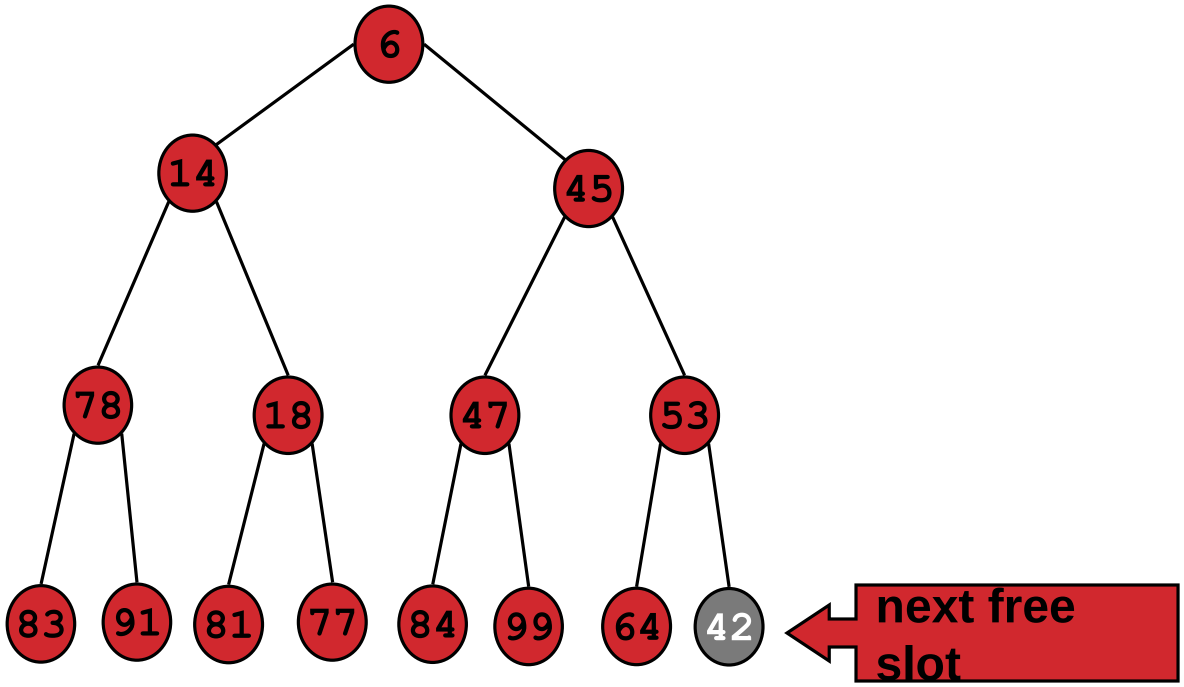
- Implementing binary heaps.
  - Use an array: no need for explicit parent or child pointers.
    - `Parent(i) = ` $\lfloor i/2 \rfloor$
    - `Left(i)   = 2i`
    - `Right(i)  = 2i + 1`



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

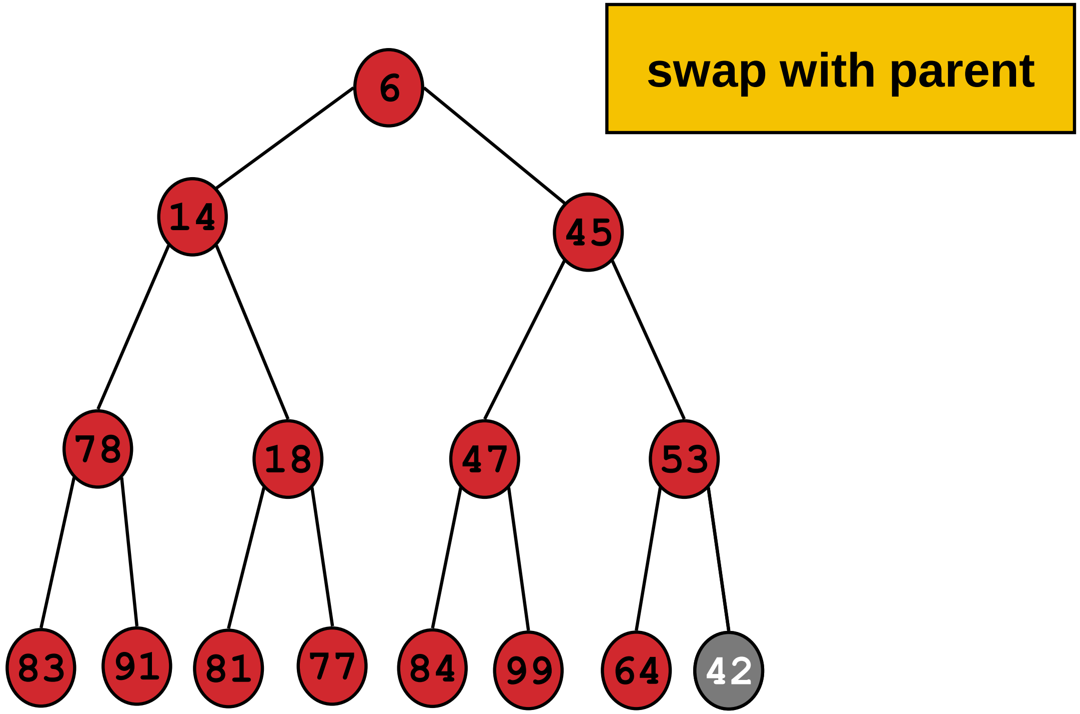
Binary Heap: Insertion

- 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



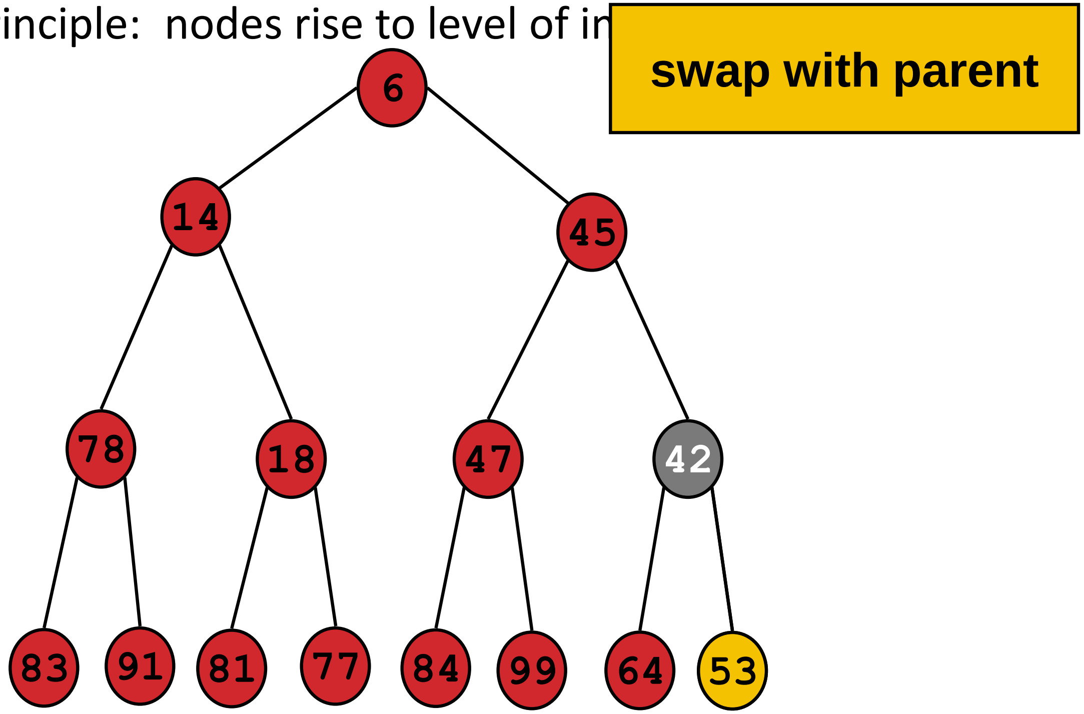
Binary Heap: Insertion

- 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

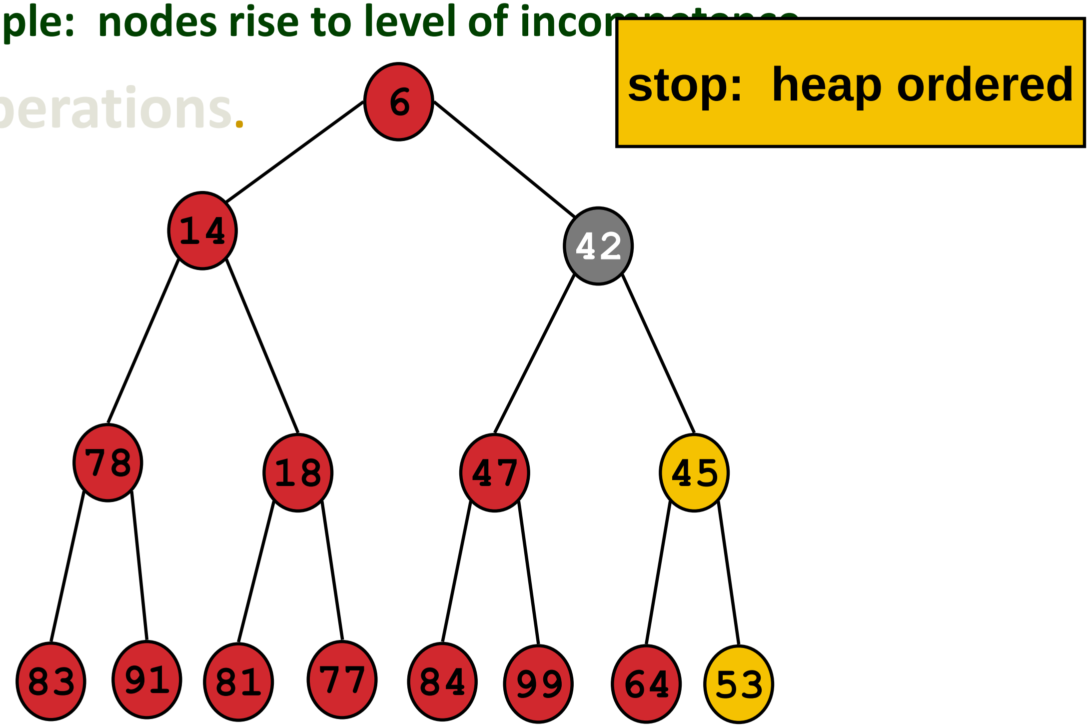
- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of in



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.



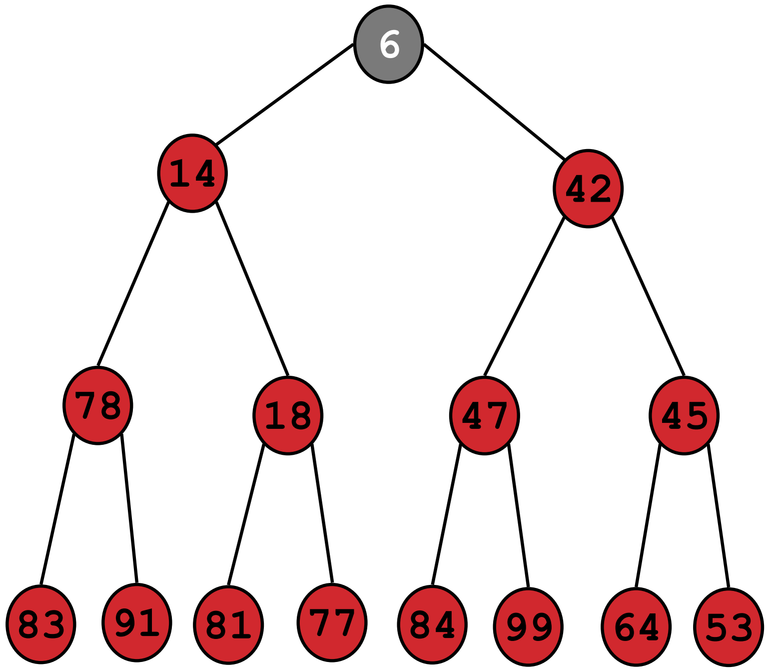
Given Data:
44, 33, 77 ,11, 55, 88, 66, 22
Build max heap tree ? (show all works).

Solved At board

Binary Heap: Delete Min

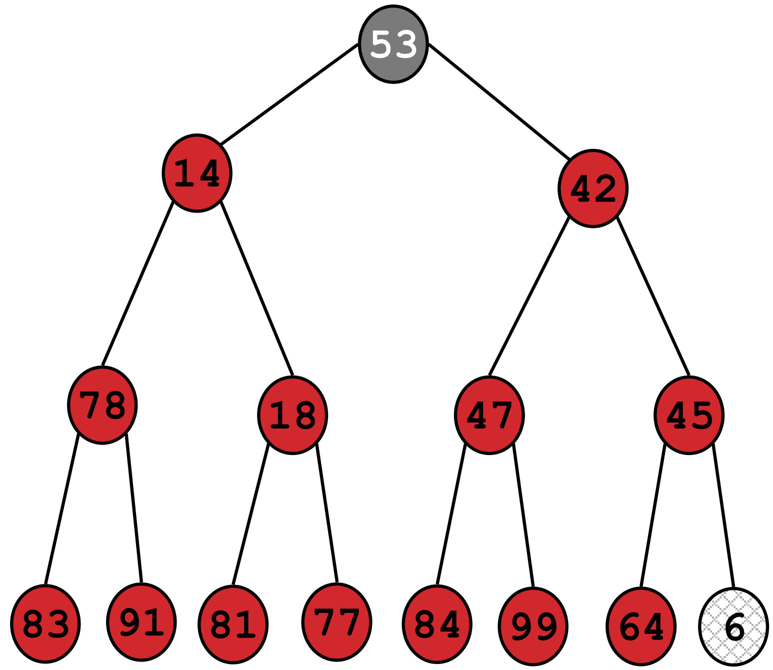
Delete minimum element from heap.

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



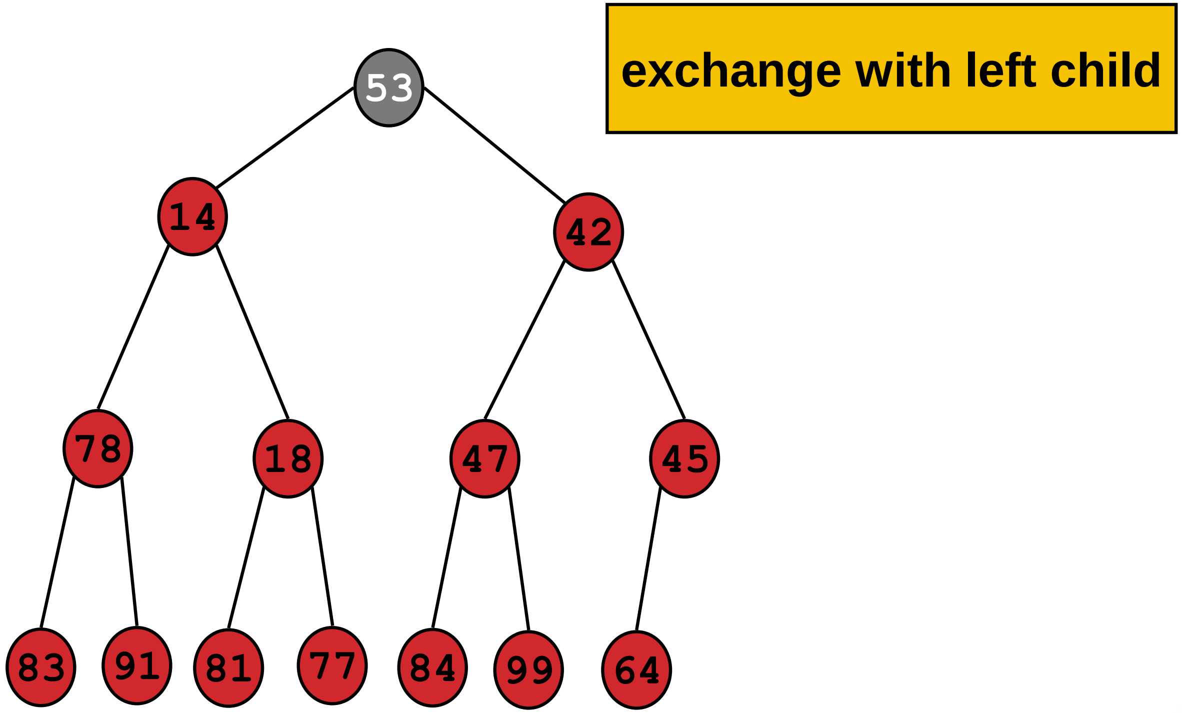
Binary Heap: Delete Min

- 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



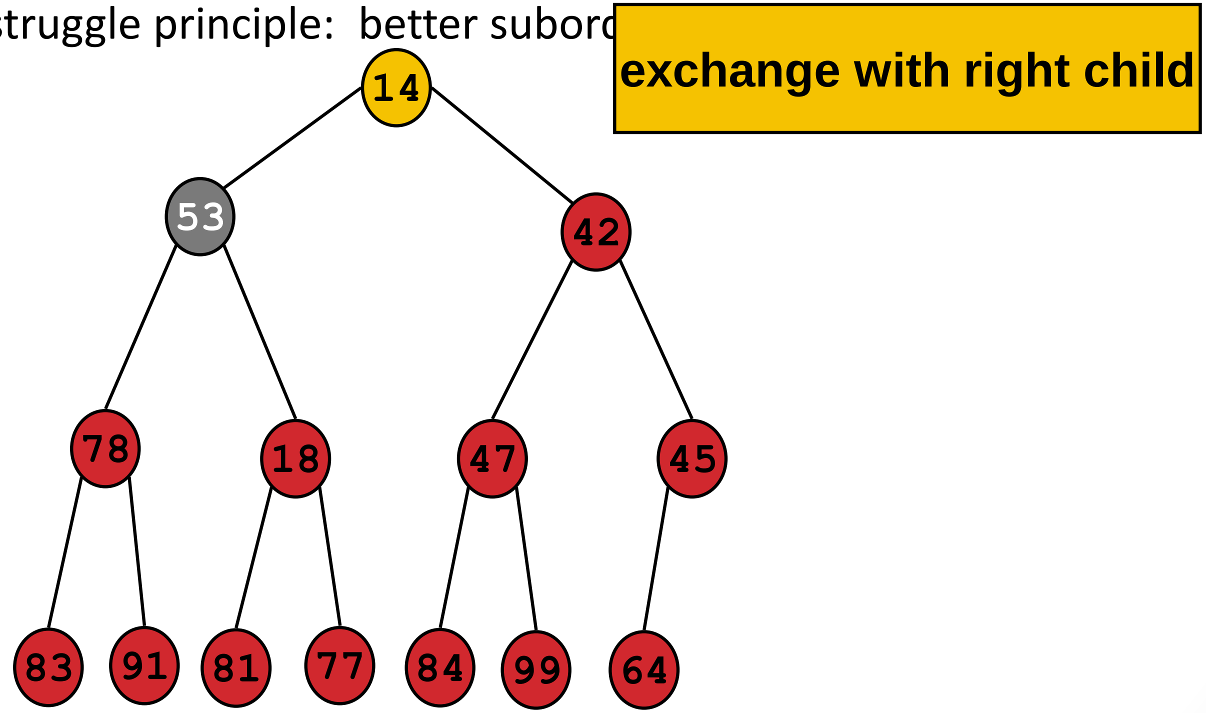
Binary Heap: Delete Min

- 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



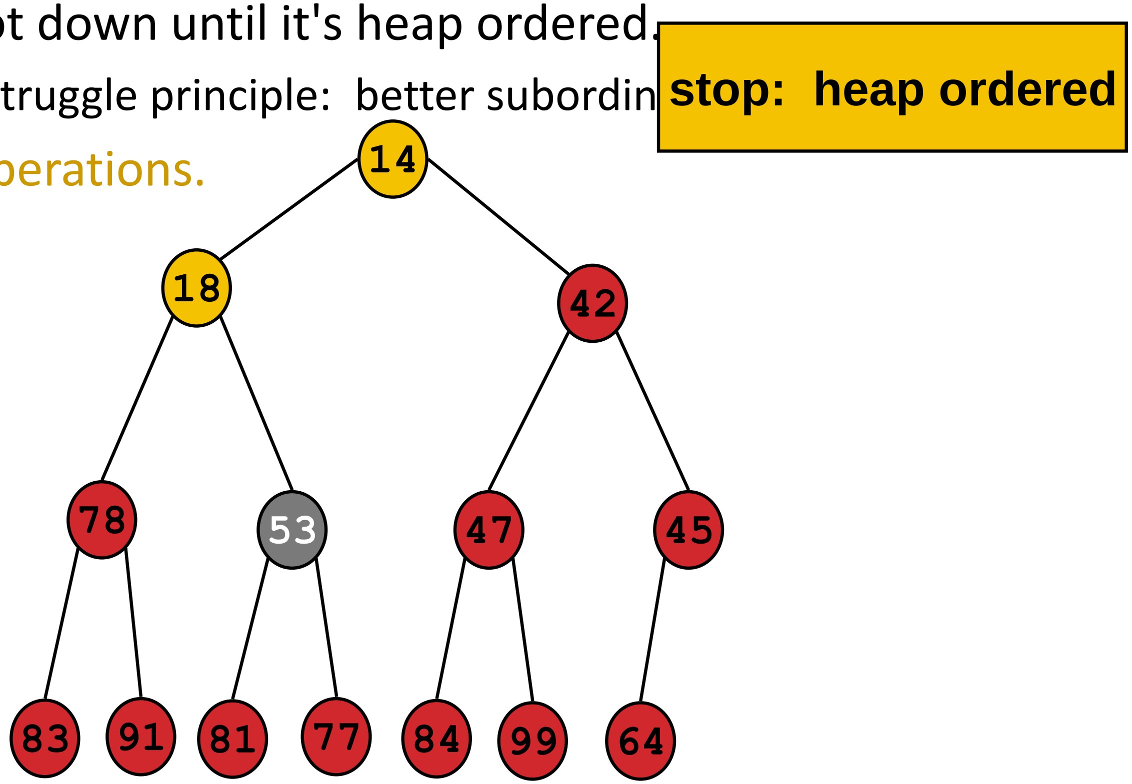
Binary Heap: Delete Min

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

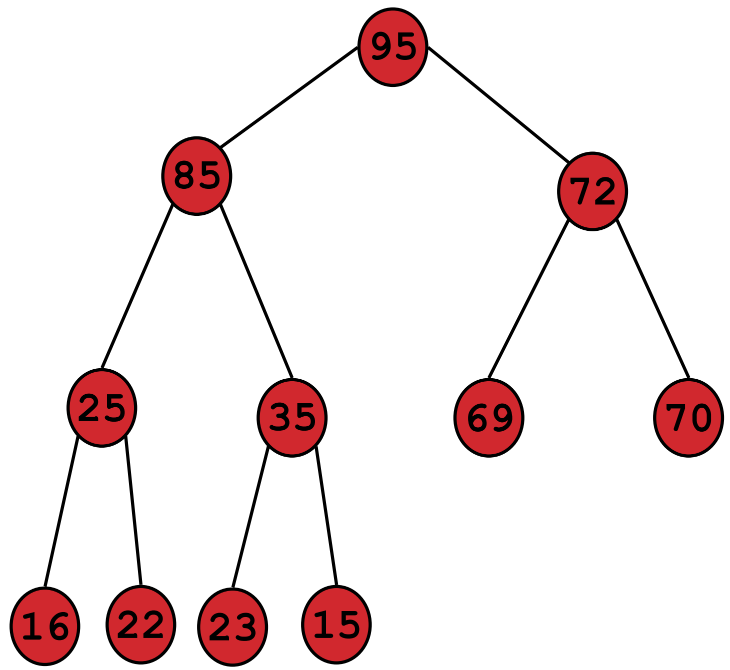


Binary Heap: Delete Min

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



Example: Delete root (95)

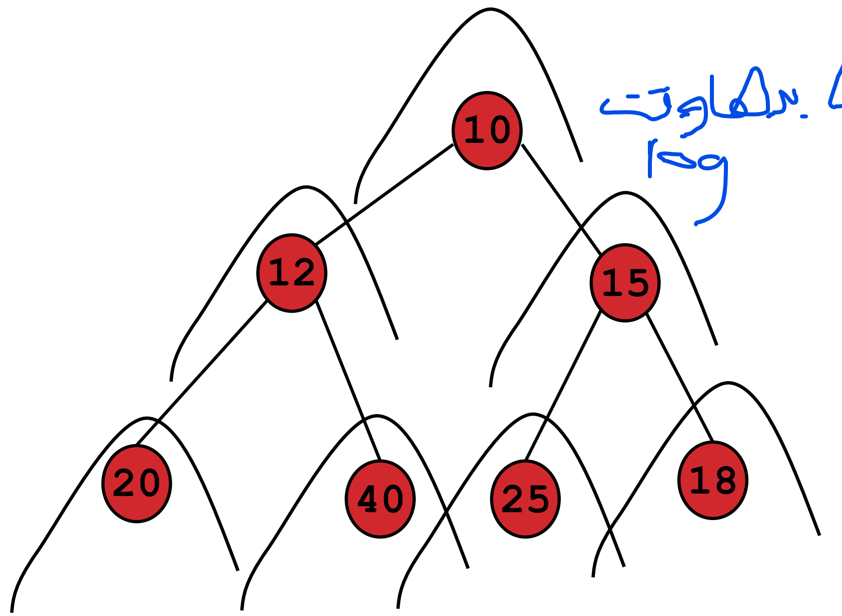


Solved At board

Heapify

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)



```c
#include <stdio.h>
#include <limits.h>


int FRONT=1;
int size=0;
//C implementation of Min Heap
struct MinHeap {
    int maxsize;
    int Heap[];
} heap;

struct MinHeap heap;
void MinHeap(int maxsize)
{

    heap.maxsize = maxsize;
    int Heap[maxsize + 1];
    Heap[0] =INT_MIN;
}

```

```c
// 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
```

```c
// Function that returns true if the passed
// node is a leaf node
int isLeaf(int pos)
{
    if (pos >= (size / 2) && pos <=size) {
        return 1;
    }
    return 0;
}

// Function to swap two nodes of the heap
void swap(int fpos, int spos)
{
    int tmp;
    tmp = heap.Heap[fpos];
    heap.Heap[fpos] = heap.Heap[spos];
    heap.Heap[spos] = tmp;
}

// Function to heapify the node at pos
```

```
// Function to insert a node into the heap
void insert(int element)
{

    if (size >= heap.maxsize) {
        return;
    }
    heap.Heap[++size] = element;
    int current = size;

    while (heap.Heap[current] < heap.Heap[parent(current)]) {
        swap(current, parent(current));
        current = parent(current);
    }
}

```

```
// Function to heapify the node at pos
void minHeapify(int pos)
{

    // If the node is a non-leaf node and greater
    // than any of its child
    if (!isLeaf(pos)) {
        if (heap.Heap[pos] > heap.Heap[leftChild(pos)]
            || heap.Heap[pos] > heap.Heap[rightChild(pos)]) {

            // Swap with the left child and heapify
            // the left child
            if (heap.Heap[leftChild(pos)] < heap.Heap[rightChild(pos)]) {
                swap(pos, leftChild(pos));
                minHeapify(leftChild(pos));
            }

                // Swap with the right child and heapify
                // the right child
            else {
                swap(pos, rightChild(pos));
                minHeapify(rightChild(pos));
            }
        }
    }
}
```

```
// Function to build the min heap using
// the minHeapify
void minHeap()
{
    for (int pos = (size / 2); pos >= 1; pos--) {
        minHeapify(pos);
    }
}
```

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