# ساختمان داده ها و الگوریتم ها(CE203)

# جلسه سیزدهم:
# هرم و مرتب سازی هرمی

**سجاد شیرعلی شهرضا**

**پاییز 1400**

***دوشنبه، 17 آبان 1400***

Based on Washington University CSE 373 Slides from Autumn 2020 by Hunter Schafer

# جلسه دوازدهم:
# هرم و مرتب سازی هرمی

**شنبه، 4 اردیبهشت 1400**

# اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 6
- یادآوری تمرین دوم:
  - شنبه هفته آینده، 22 آبان
  - ساعت 8 صبح
  - از طریق سایت کورسز (بخش تشریحی) و کوئرا (بخش برنامه نویسی)

# Interface vs. Implementation

**Interface:** the operations of an ADT

**Implementation:** the code for a data structure

# Interface vs. Implementation

**Interface:** the operations of an ADT

- What you see on documentation web pages

**Implementation:** the code for a data structure

- What you see in source files

# Interface vs. Implementation

**Interface:** the operations of an ADT

- What you see on documentation web pages
- Method names and specifications

**Implementation:** the code for a data structure

- What you see in source files
- Fields and method bodies

# Interface vs. Implementation

**Interface:** the operations of an ADT

- What you see on documentation web pages
- Method names and specifications
- Abstract from details: what to do, not how to do it

**Implementation:** the code for a data structure

- What you see in source files
- Fields and method bodies
- Provide the details: how to do operation

# Interface vs. Implementation

**Interface:** the operations of an ADT

- What you see on documentation web pages
- Method names and specifications
- Abstract from details: what to do, not how to do it
- Java syntax: **interface**

**Implementation:** the code for a data structure

- What you see in source files
- Fields and method bodies
- Provide the details: how to do operation
- Java syntax: **class**

# Interface vs. Implementation

**Interface:** the operations of an ADT

- What you see on documentation web pages
- Method names and specifications
- Abstract from details: what to do, not how to do it
- Java syntax: **interface**

**Implementation:** the code for a data structure

- What you see in source files
- Fields and method bodies
- Provide the details: how to do operation
- Java syntax: **class**

Could be many implementations of an interface
e.g. List: ArrayList, LinkedList

# ADTs (interfaces)

| ADT | Description |
|---|---|
| List | Ordered collection (aka sequence) |
| Set | Unordered collection with no duplicates |
| Map | Collection of keys and values, like a dictionary |
| Stack | Last-in-first-out (LIFO) collection |
| Queue | First-in-first-out (FIFO) collection |
| Priority Queue | *Later this lecture!* |

# Implementations of ADTs

| Interface | Implementation (data structure) |
|---|---|
| List | ArrayList, LinkedList |
| Set | HashSet, TreeSet |
| Map | HashMap, TreeMap |
| Stack | *Can be done with a LinkedList* |
| Queue | *Can be done with a LinkedList* |
| Priority Queue | *Can be done with a heap — later this lecture!* |


سوال؟

# صف با اولویت

***مجموعه ای از اشیاء دارای اولویت مختلف***

# Priority Queue

- Primary operation:
  - Stack:  remove newest element
  - Queue:  remove oldest element
  - Priority queue:  remove highest priority element
- Priority:
  - Additional information for each element
  - Needs to be **Comparable**

# Priority Queue

| Priority | Task |
|---|---|
| 0 | CE 203 Assignment 2 |
| 1 | CE 203 Midterm |
| 2 | Football News |
| 2 | Opening universities news |

# java.util.PriorityQueue<E>

```
class PriorityQueue<E> {
 boolean add(E e); //insert e.
 E poll(); //remove&return min elem.
 E peek(); //return min elem.
 boolean contains(E e);
 boolean remove(E e);
 int size();
 ...
}
```

# Implementations

**LinkedList**

| | |
|---|---|
| `add()` | **put new element at front – O(1)** |
| `poll()` | **must search the list – O(n)** |
| `peek()` | **must search the list – O(n)** |

# Implementations

**LinkedList**

| | |
|---|---|
| `add()` | **put new element at front – O(1)** |
| `poll()` | **must search the list – O(n)** |
| `peek()` | **must search the list – O(n)** |

**LinkedList that is always sorted**

| | |
|---|---|
| `add()` | **must search the list – O(n)** |
| `poll()` | **highest priority element at front – O(1)** |
| `peek()` | **same – O(1)** |

# Implementations

**LinkedList**

| | |
|---|---|
| `add()` | **put new element at front – O(1)** |
| `poll()` | **must search the list – O(n)** |
| `peek()` | **must search the list – O(n)** |

**LinkedList that is always sorted**

| | |
|---|---|
| `add()` | **must search the list – O(n)** |
| `poll()` | **highest priority element at front – O(1)** |
| `peek()` | **same – O(1)** |

**Can we do better?**


سوال؟

هرم

# A Heap..

**Is a binary tree satisfying 2 properties:**

Do not confuse with heap memory – different use of the word heap.

# A Heap..

**Is a binary tree satisfying 2 properties:**

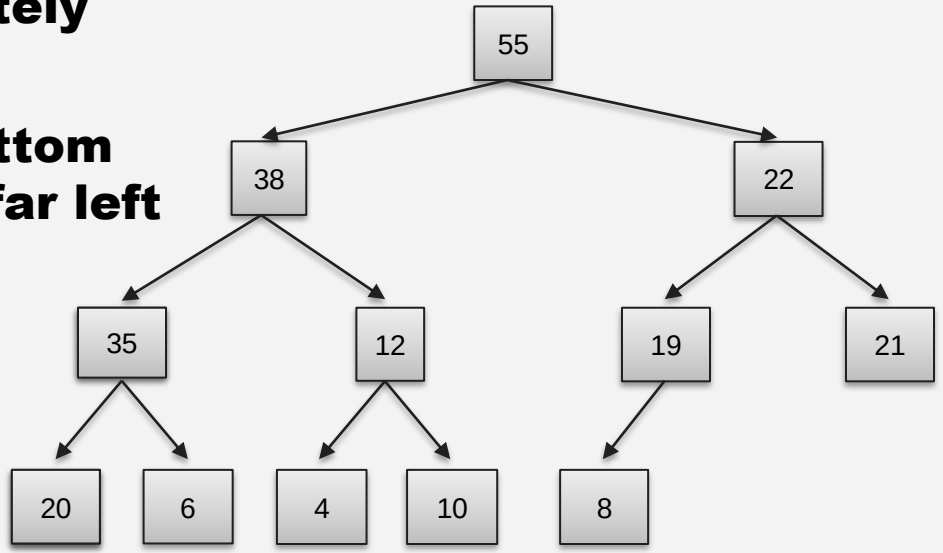
1) **Completeness. Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.**

Do not confuse with heap memory – different use of the word heap.

# Completeness

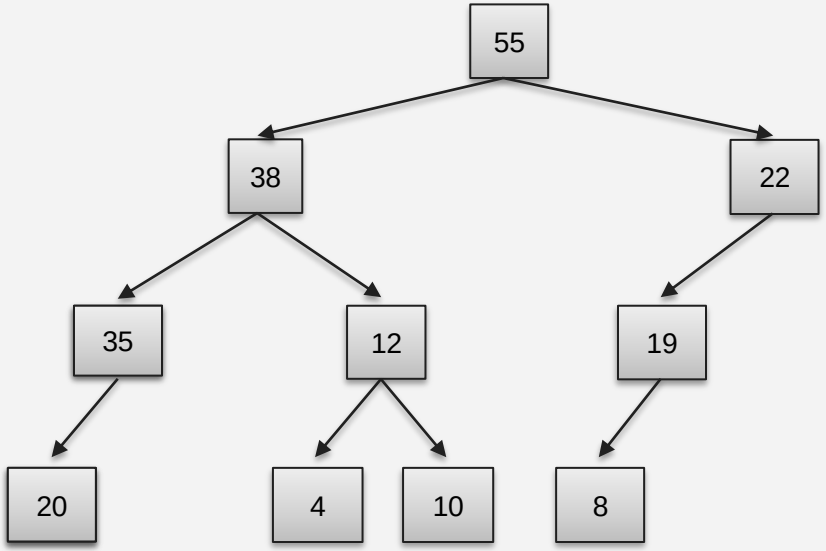
**Every level (except last) completely filled.**

**Nodes on bottom level are as far left as possible.**



# Completeness

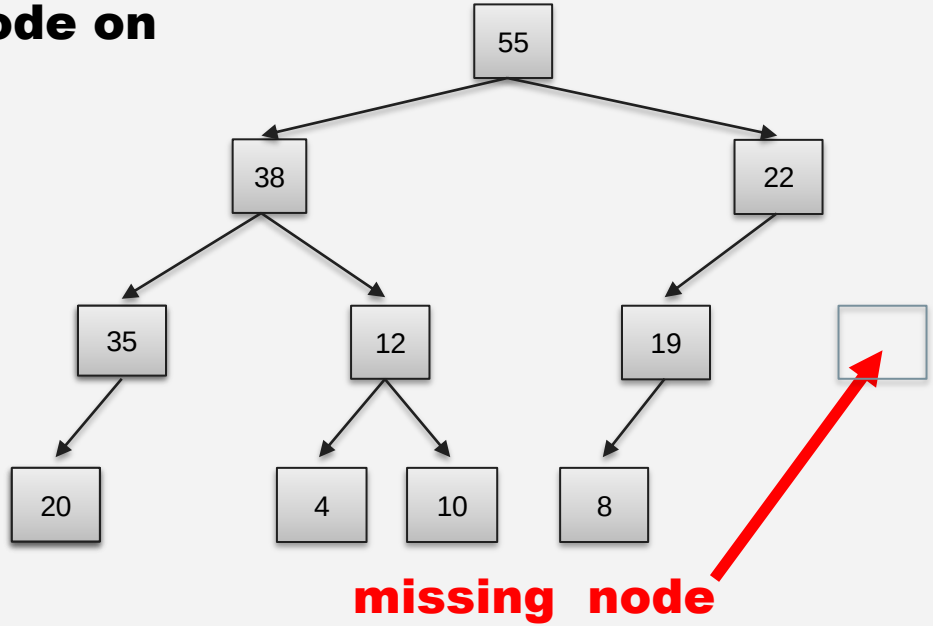
**Not a heap because:**



# Completeness

**Not a heap because:**

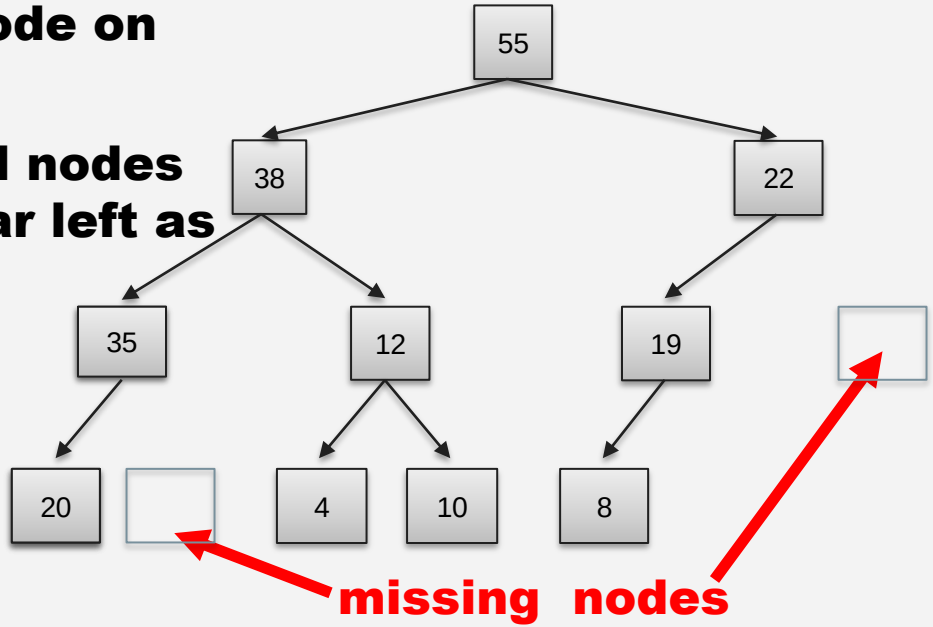
- **missing a node on level 2**



# Completeness

**Not a heap because:**

- **missing a node on level 2**
- **bottom level nodes are not as far left as possible**



# A Heap..

**Is a binary tree satisfying 2 properties:**

1) **Completeness. Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.**
2) **Heap-order**:

# A Heap..

**Is a binary tree satisfying 2 properties:**

1) **Completeness. Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.**
2) **Heap-order**:

**Max-Heap: every element in tree is <= its parent**

"max on top"

# A Heap..

**Is a binary tree satisfying 2 properties:**

1) **Completeness. Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.**
2) **Heap-order**:

**Max-Heap: every element in tree is <= its parent**

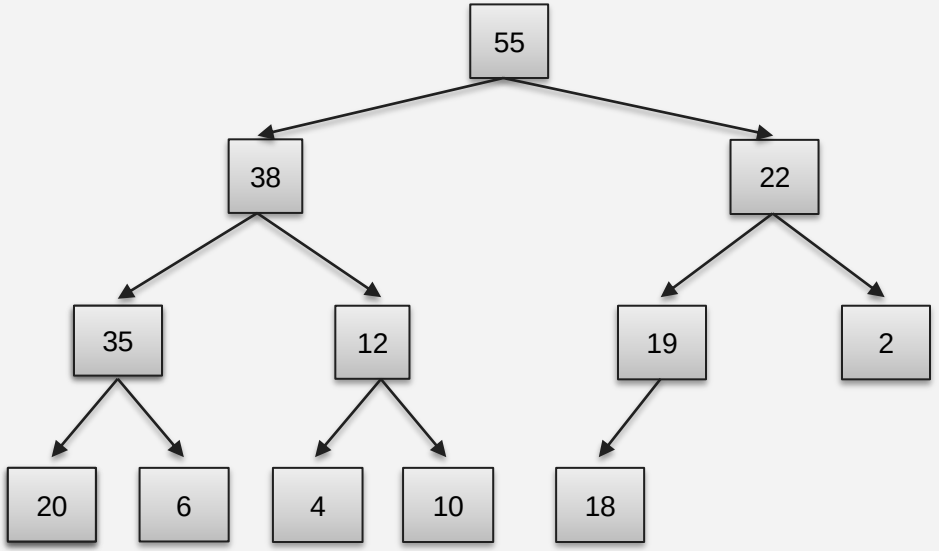
"max on top"

**Min-Heap: every element in tree is >= its parent**

"min on top"

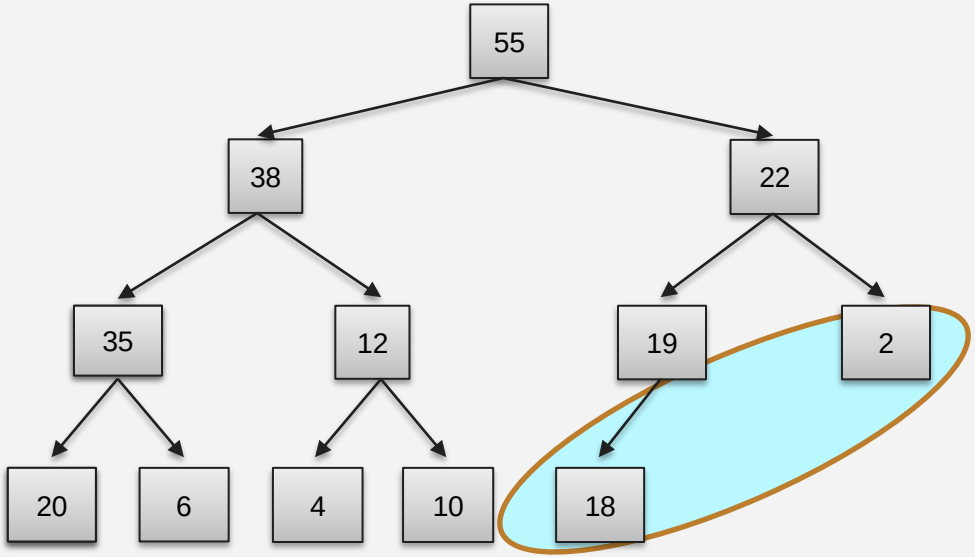
# Heap-order (max-heap)

**Every element is <= its parent**



# Heap-order (max-heap)

**Every element is <= its parent**




**Note: Bigger elements can be deeper in the tree!**

# A Heap..

**Is a binary tree satisfying 2 properties**

1) **Completeness. Every level of the tree (except last) is completely filled. All holes in last level are all the way to the right.**
2) **Heap-order.**

   **Max-Heap: every element in tree is <= its parent**

**Primary operations:**

1) **add(e): add a new element to the heap**
2) **poll(): delete the max element and return it**
3) **peek(): return the max element**

# Priority queues

**Heaps can implement priority queues**

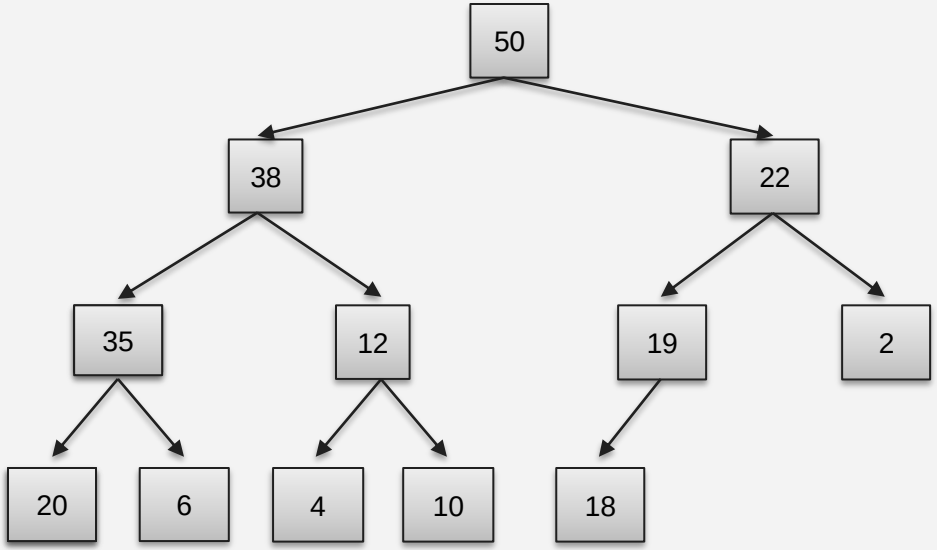
- Each heap node contains priority of a queue item
- Efficiency we will achieve:
  - add(): O(log n)
  - poll(): O(log n)
  - peek(): O(1)
- No linear time operations:  better than lists


سوال؟

# الگوریتم های هرم

**چگونگی تغییر هرم**

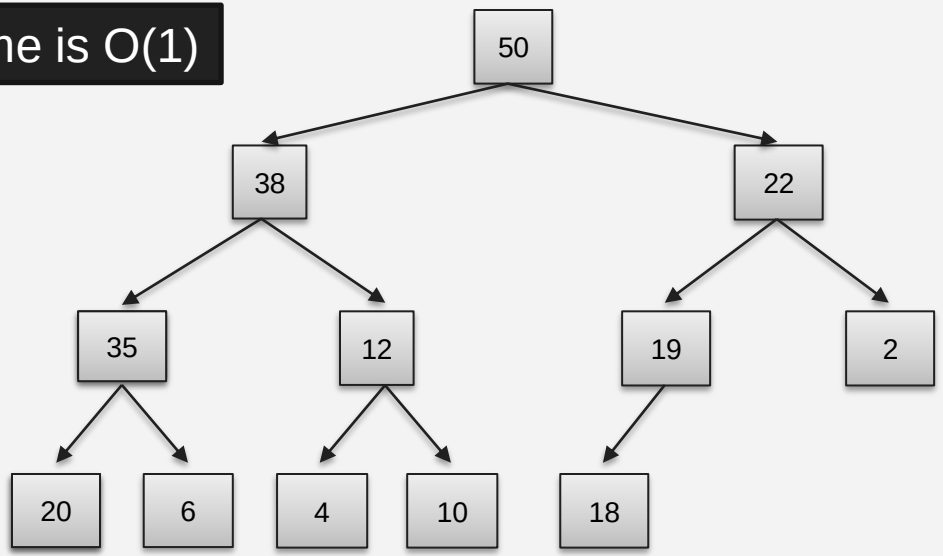
# Heap: peek()



# Heap: peek()



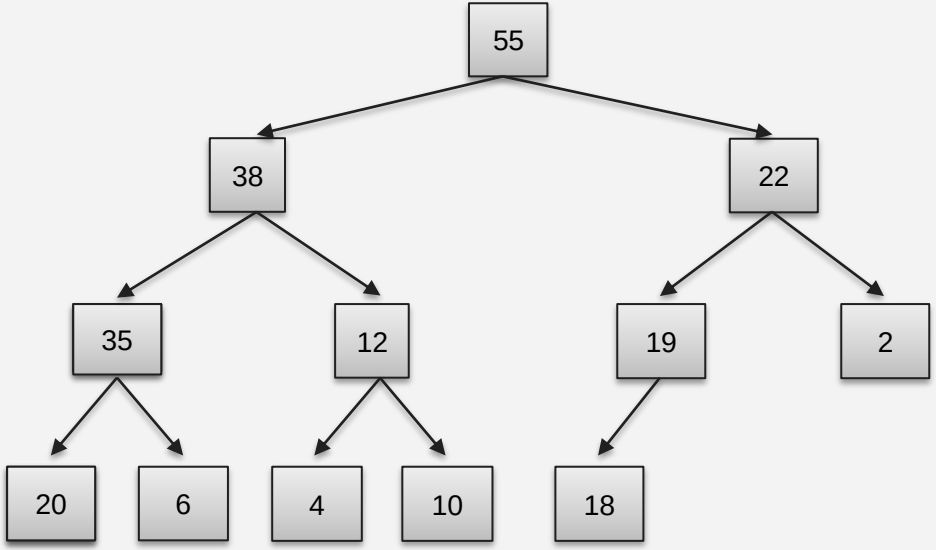
Time is O(1)



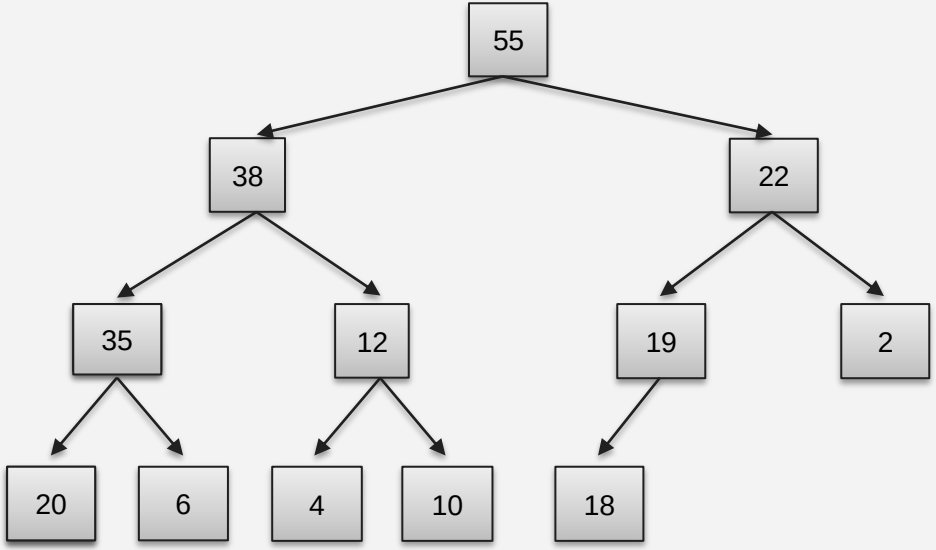

50

1. **Return root value**

# Heap: add(e)

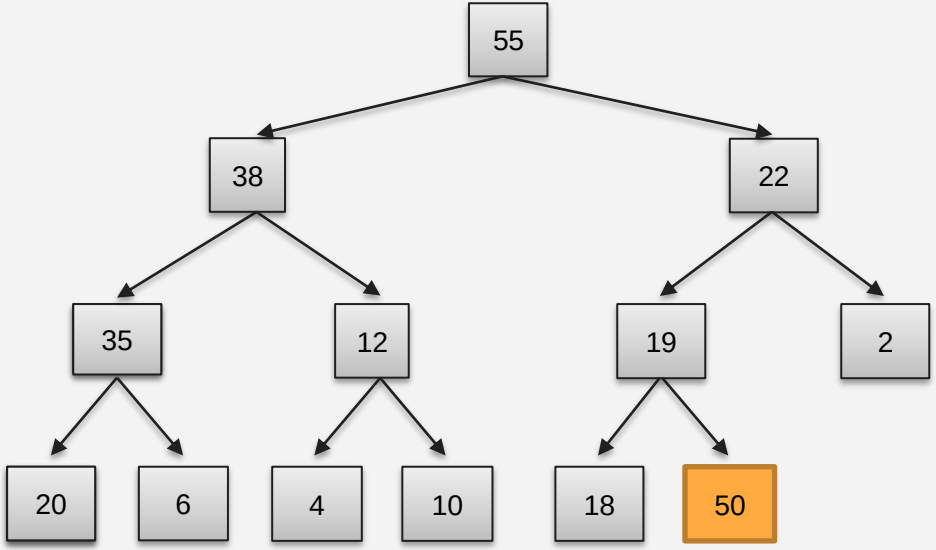


# Heap: add(e)



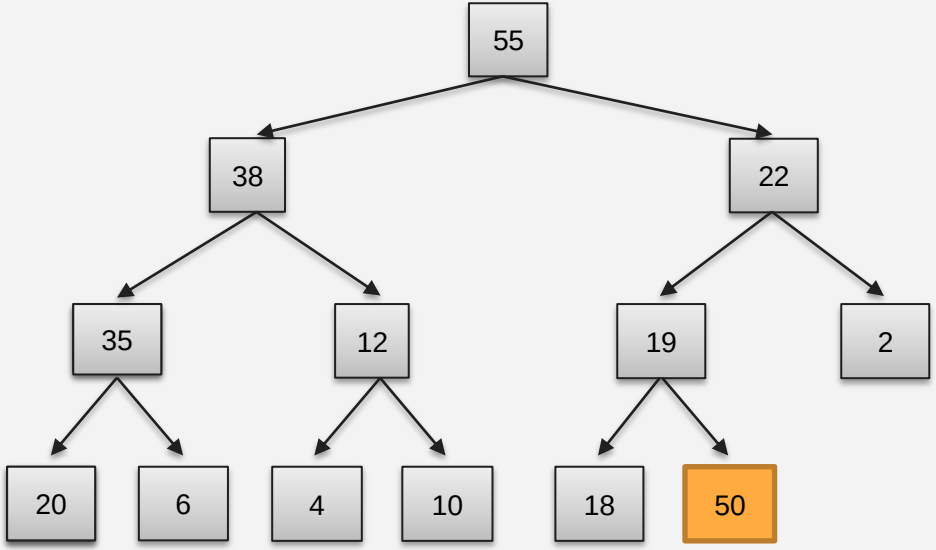

1. **Put in the new element in a new node (leftmost empty leaf)**

# Heap: add(e)



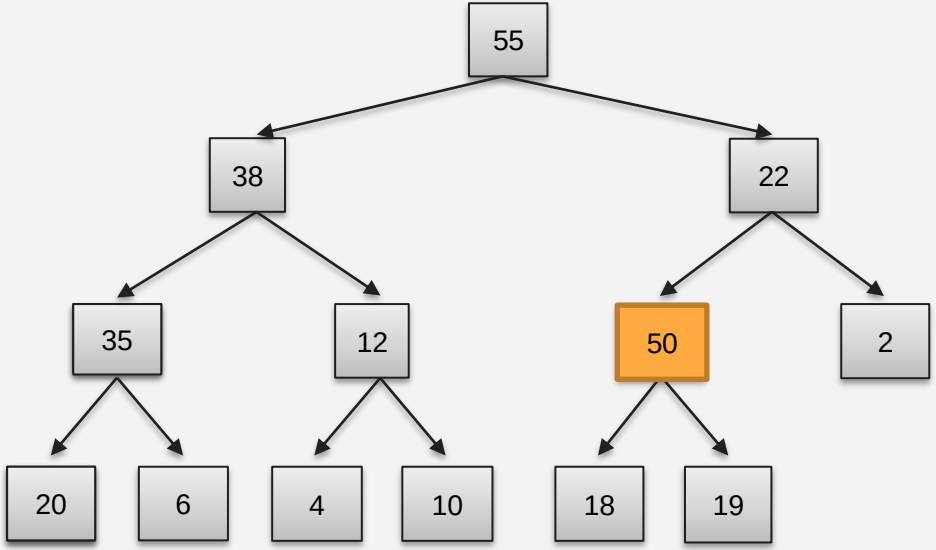

1. **Put in the new element in a new node (leftmost empty leaf)**

# Heap: add(e)



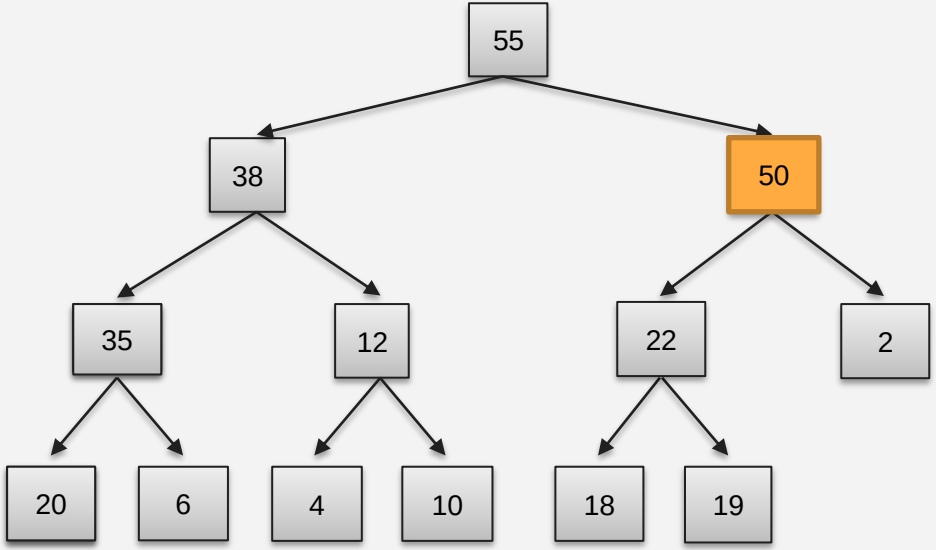

1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

# Heap: add(e)




1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

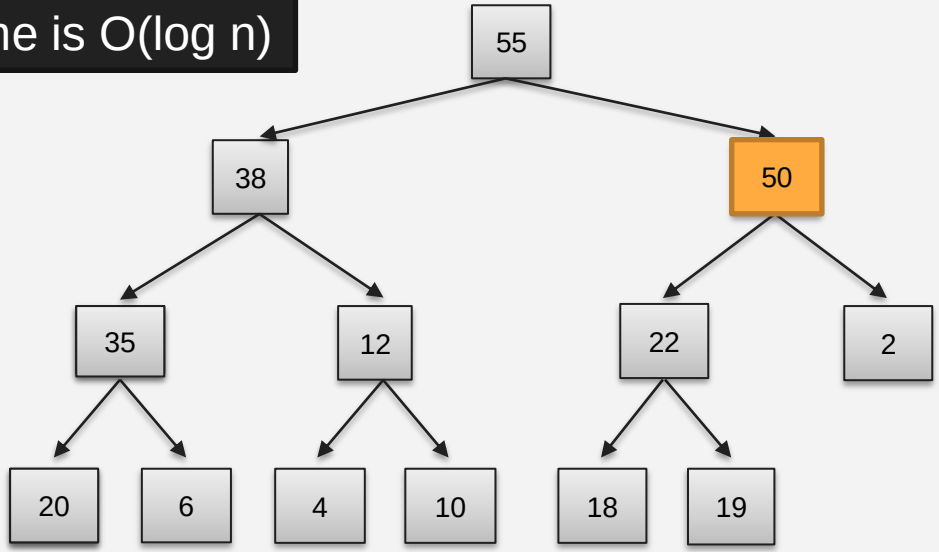
# Heap: add(e)




1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

# Heap: add(e)

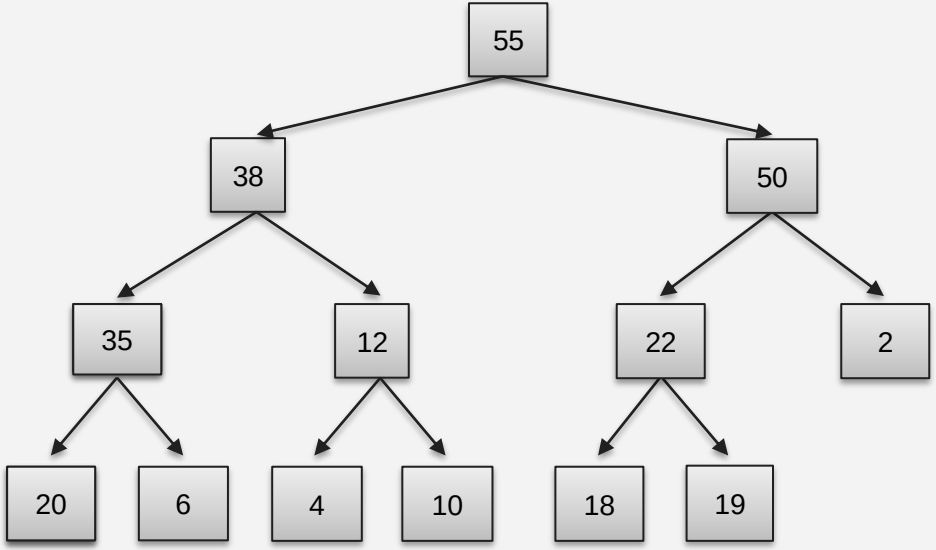
Time is O(log n)



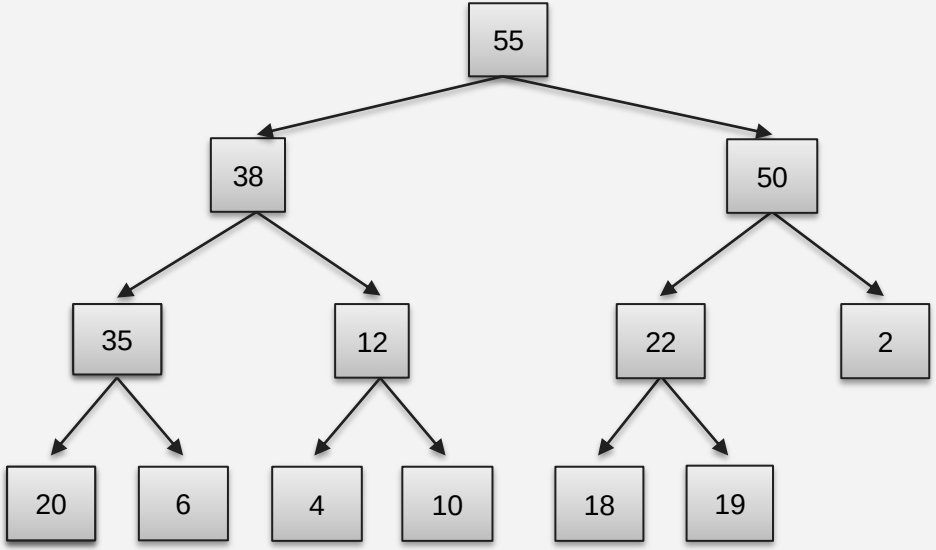

1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**


سوال؟

# Heap: poll()

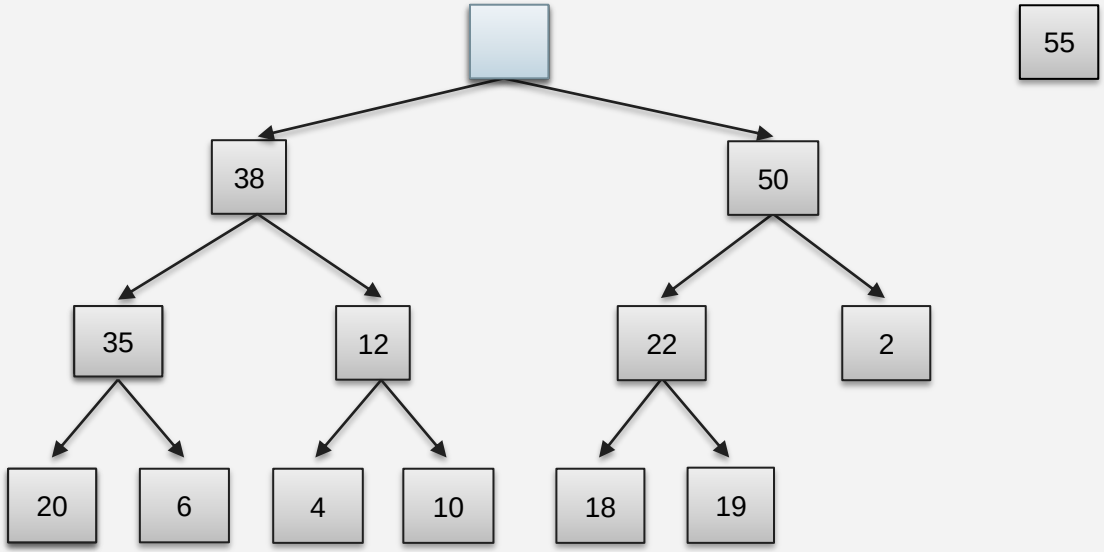


# Heap: poll()



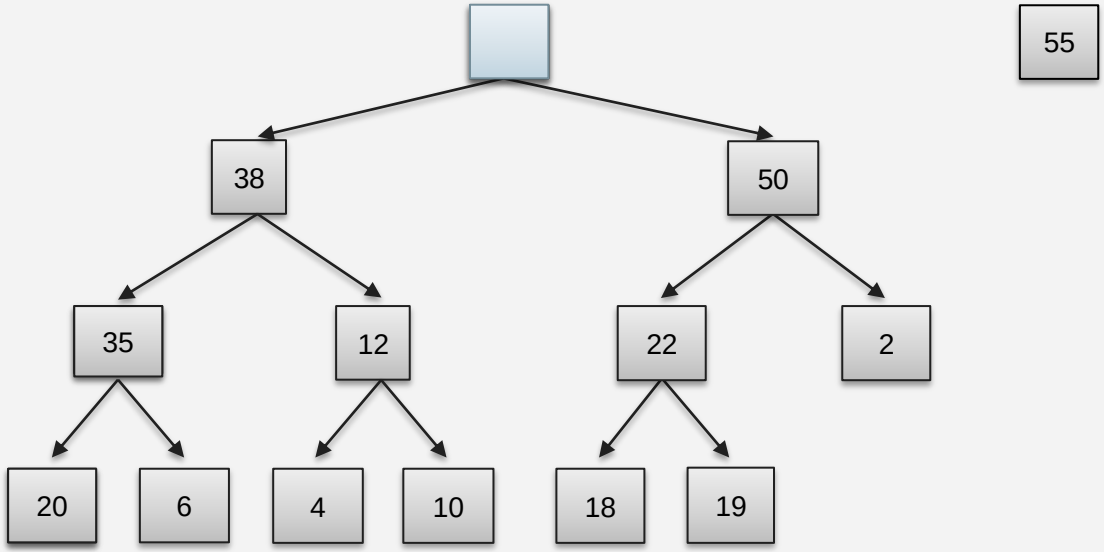

1. **Save root element in a local variable**

# Heap: poll()



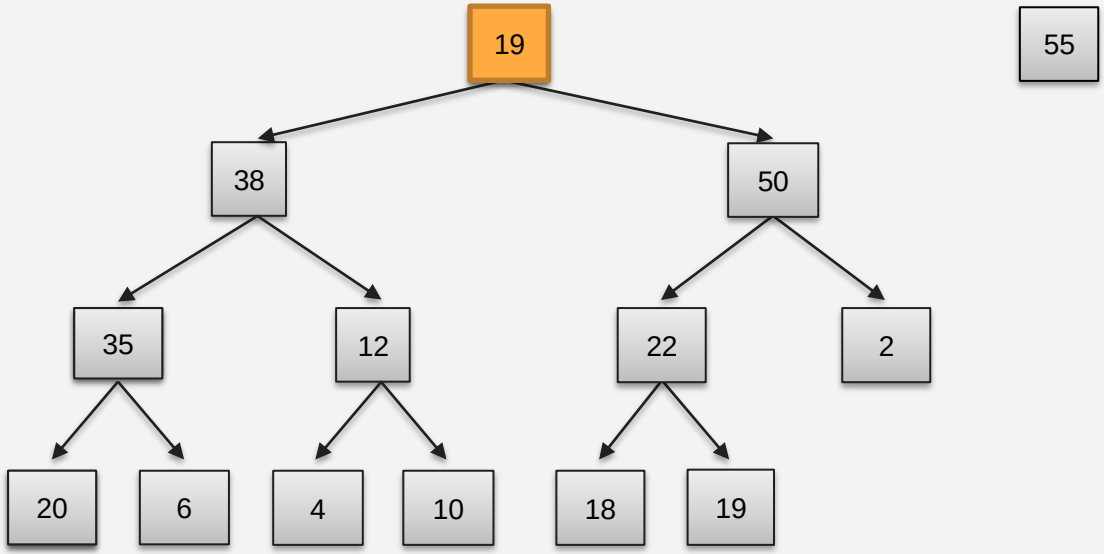

1. **Save root element in a local variable**

# Heap: poll()



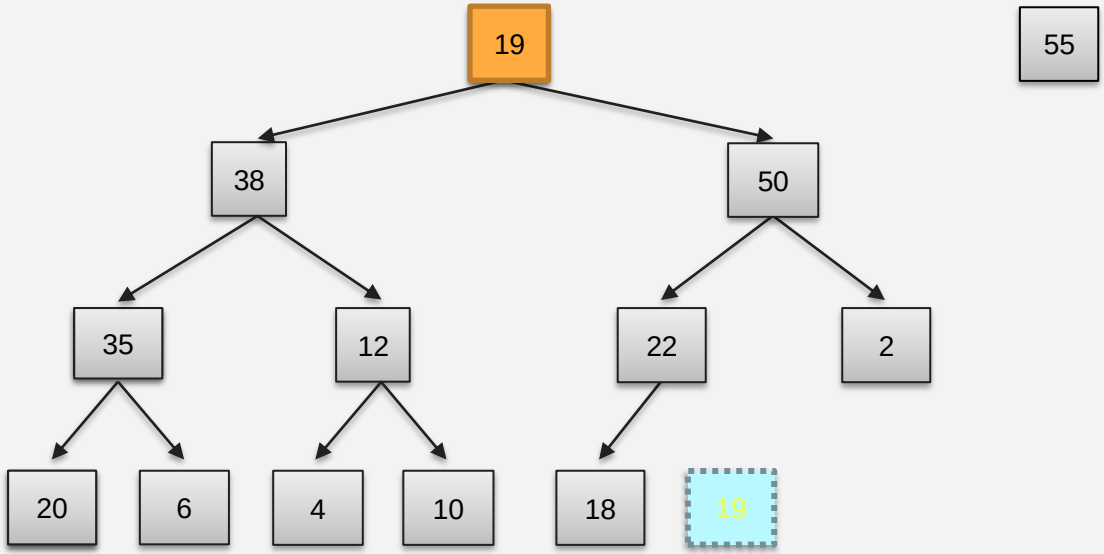

1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

# Heap: poll()



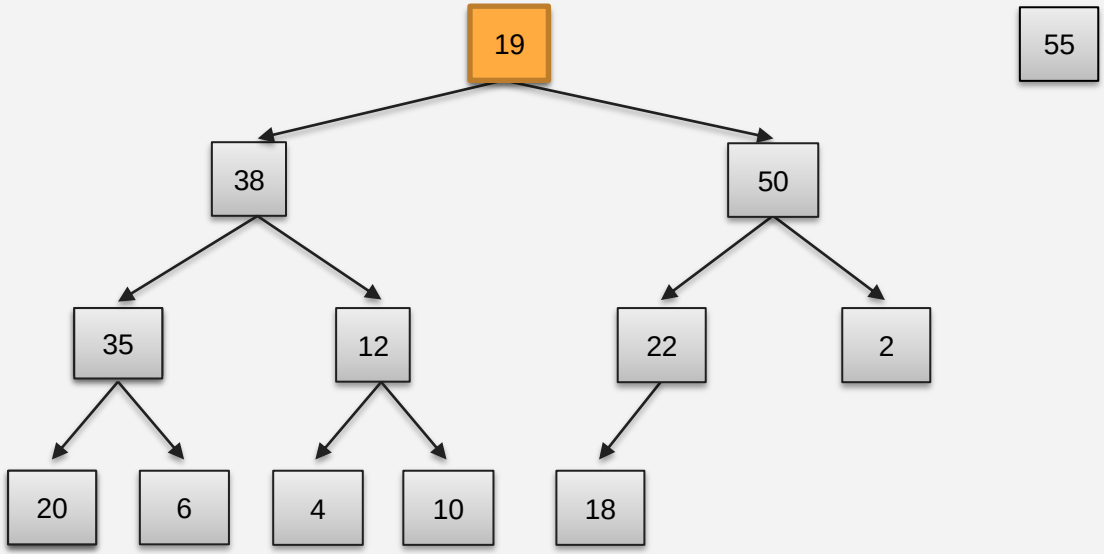

1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

# Heap: poll()



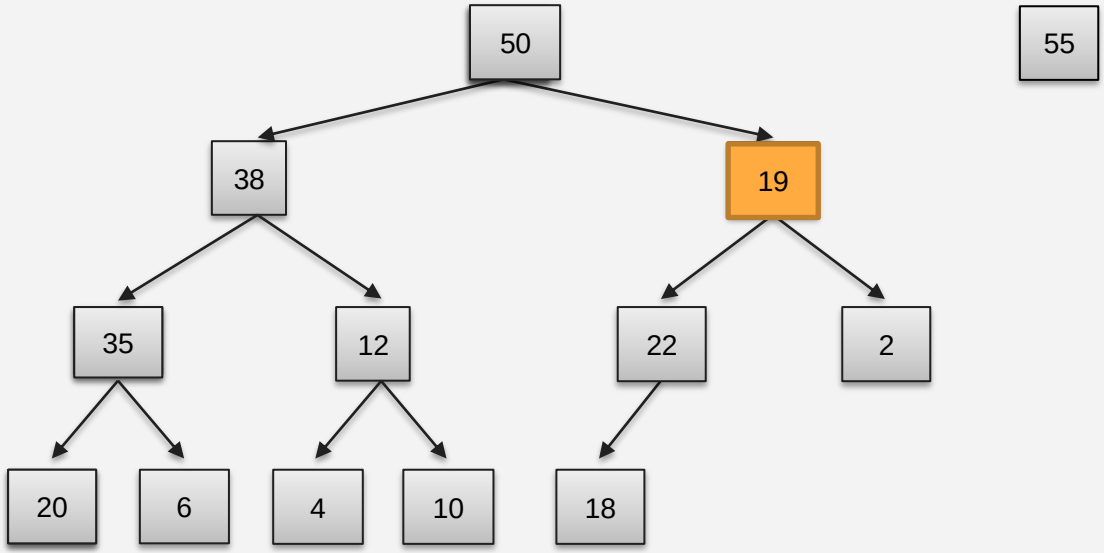

1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

# Heap: poll()



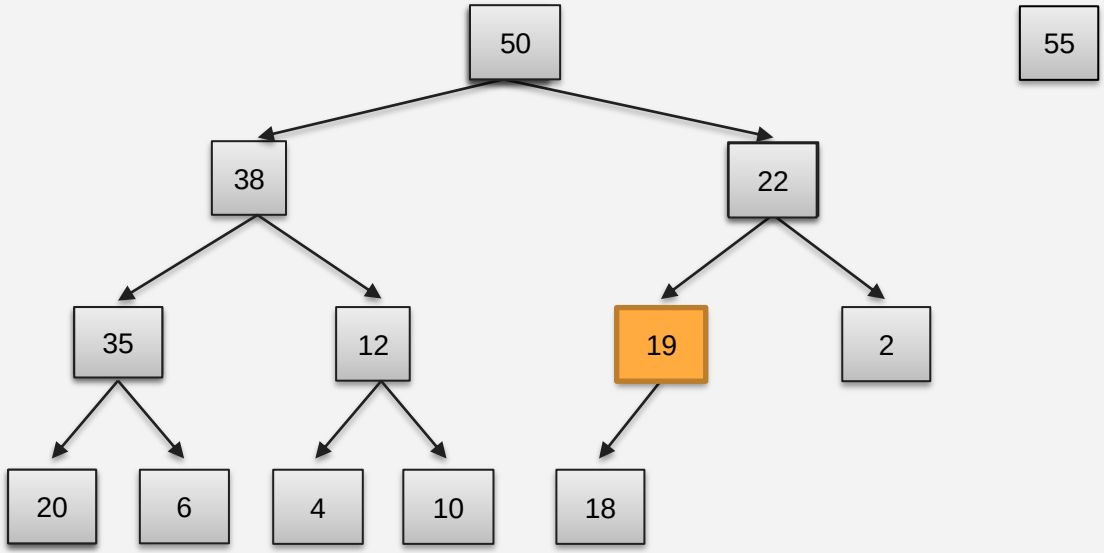

1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**

# Heap: poll()




1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**

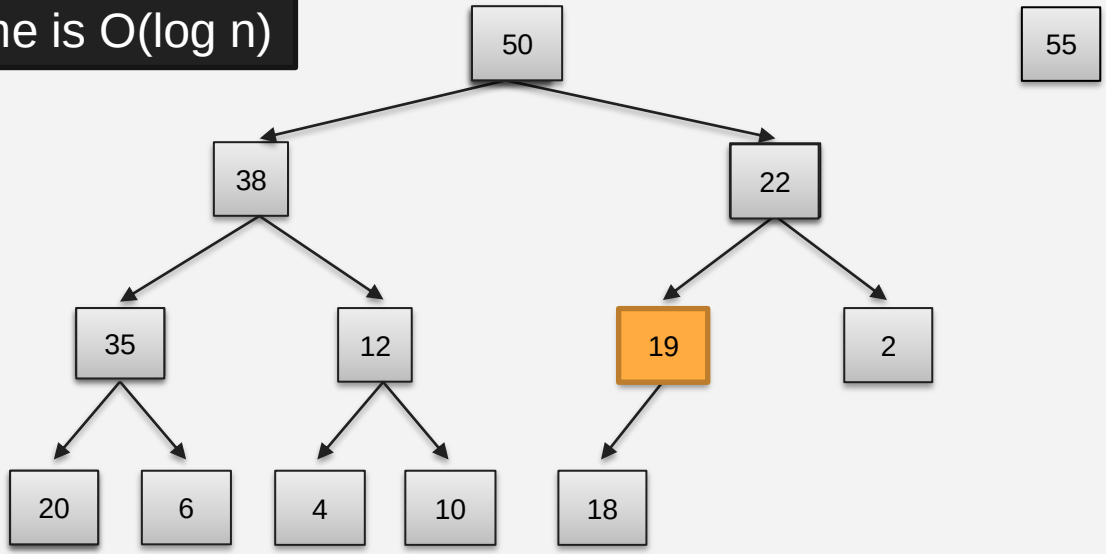
# Heap: poll()




1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**

# Heap: poll()

Time is O(log n)




1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**


سوال؟

# پیاده سازی هرم

**چگونگی پیاده سازی هرم**

# Tree implementation

```
public class HeapNode<E> {
  private E value;
  private HeapNode left;
  private HeapNode right;
  ...
}
```

But since tree is complete, even more space-efficient implementation is possible…

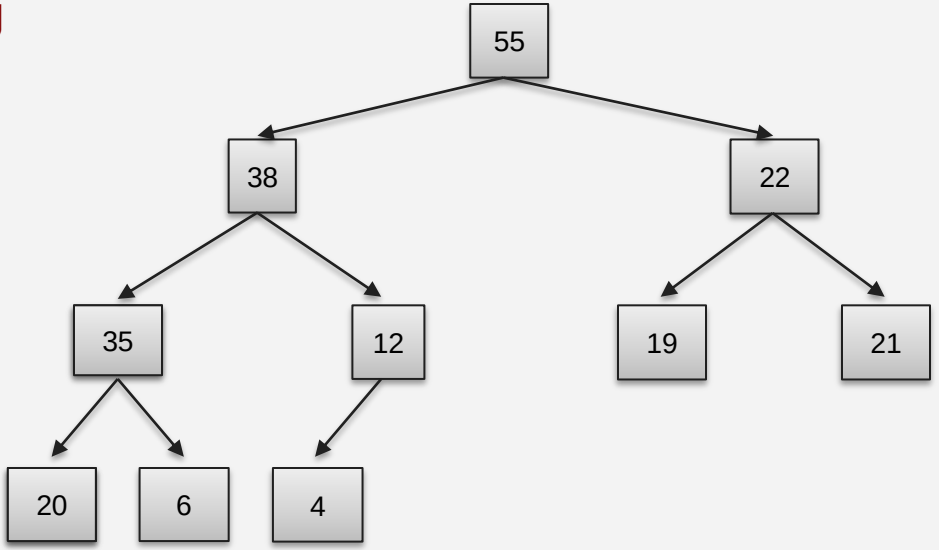
# Array implementation

```
public class Heap<E> {
  (* represent tree as array *)
  private E[] heap;
  ...
}
```

# Numbering tree nodes

**Number node starting at root row by row, left to right**

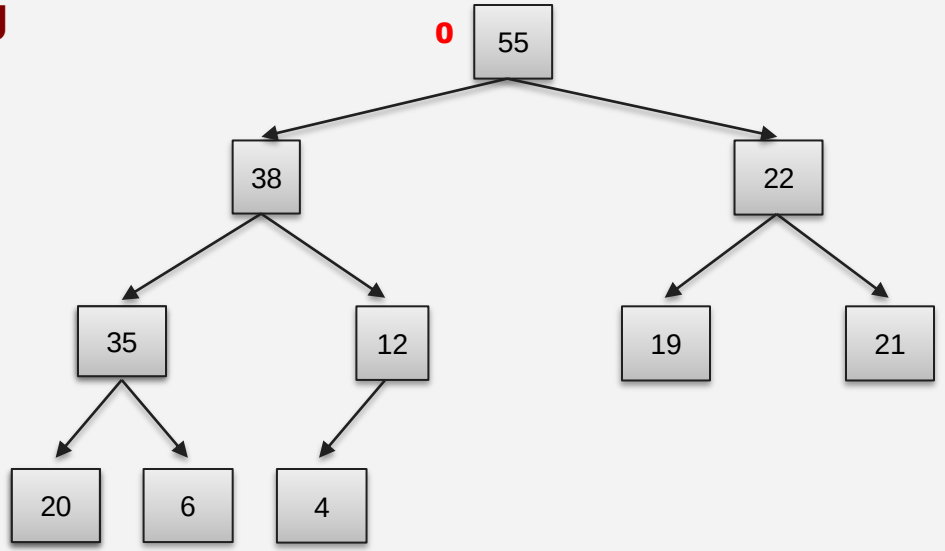
**Same order as level-order traversal**



# Numbering tree nodes

**Number node starting at root row by row, left to right**

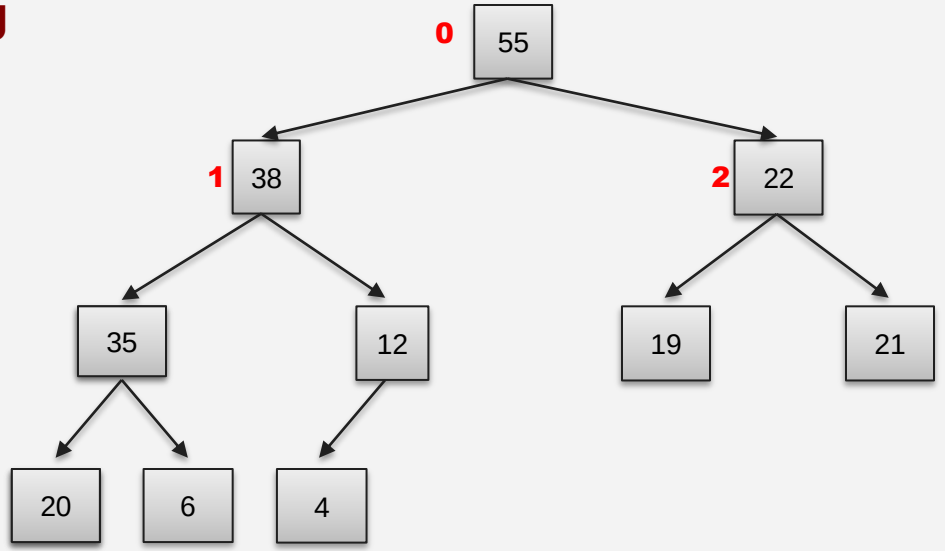
**Same order as level-order traversal**



# Numbering tree nodes

**Number node starting at root row by row, left to right**

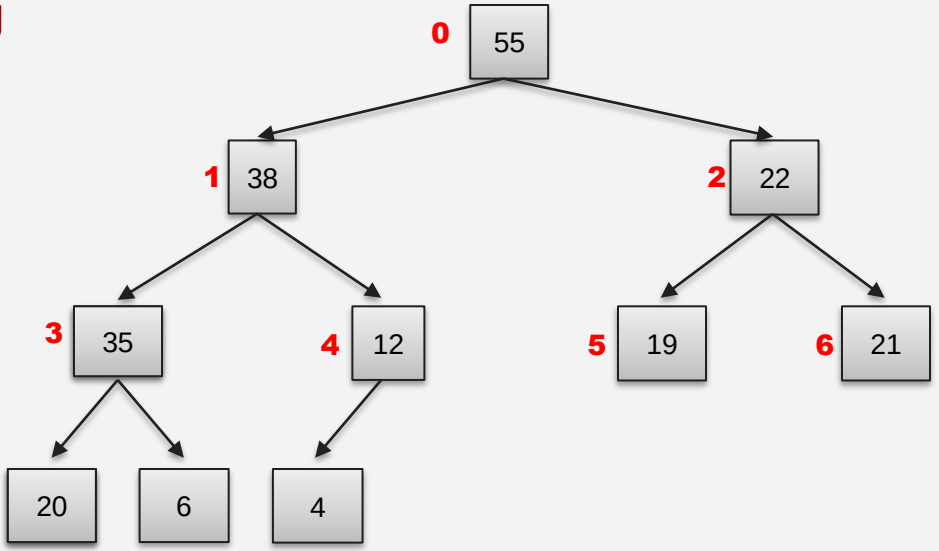
**Same order as level-order traversal**



# Numbering tree nodes

**Number node starting at root row by row, left to right**

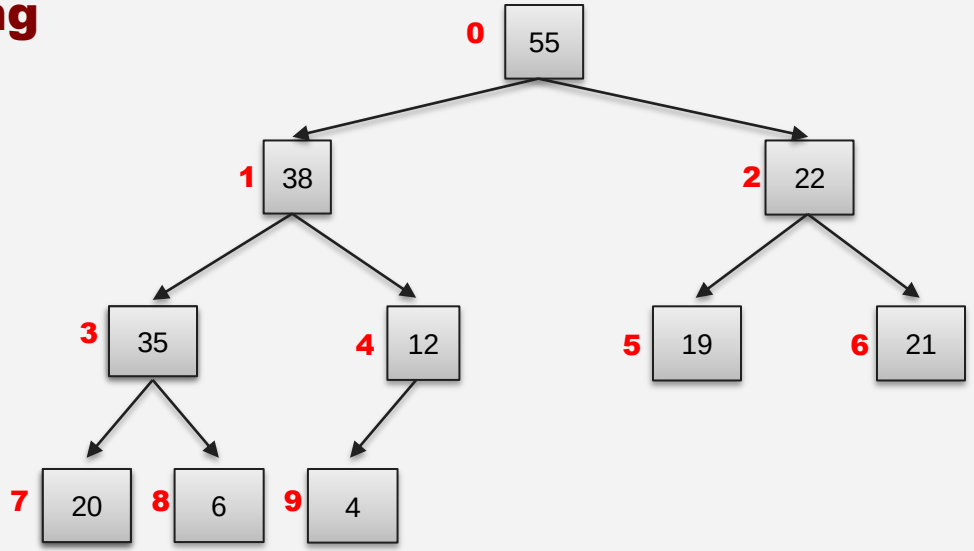
**Same order as level-order traversal**



# Numbering tree nodes

**Number node starting at root row by row, left to right**

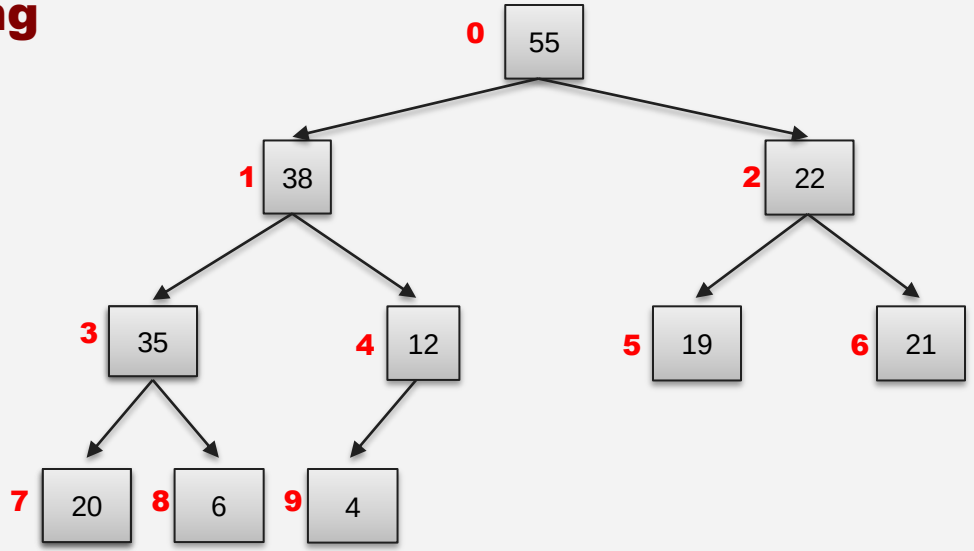
**Same order as level-order traversal**



# Numbering tree nodes

**Number node starting at root row by row, left to right**

**Same order as level-order traversal**

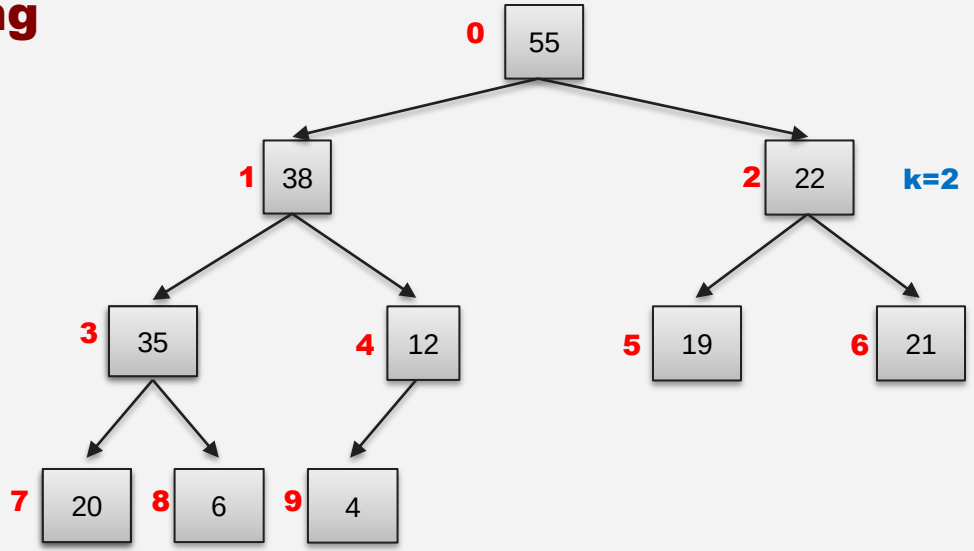



Children of node **k** are nodes **2k+1** and **2k+2**
Parent of node **k** is node **(k-1)/2**

# Numbering tree nodes

**Number node starting at root row by row, left to right**

**Same order as level-order traversal**

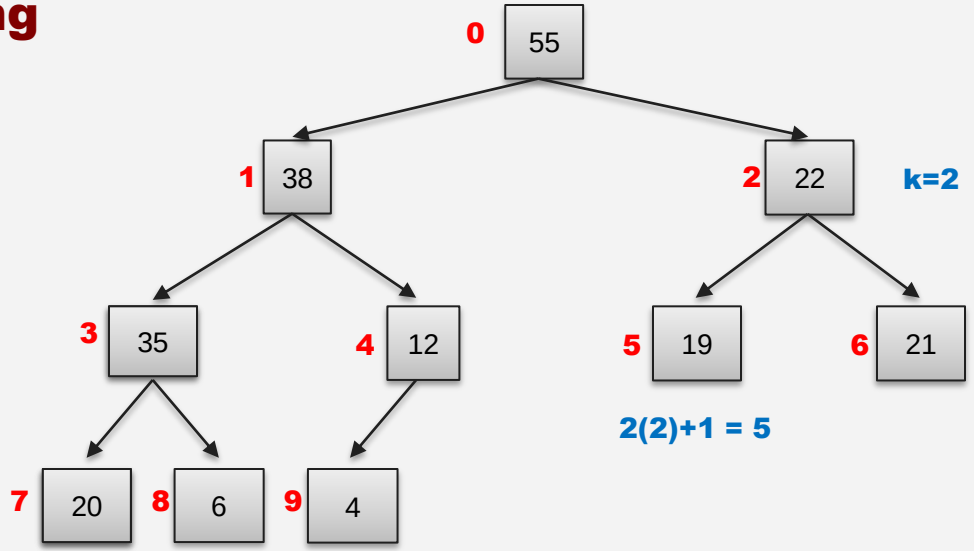



Children of node **k** are nodes **2k+1** and **2k+2**
Parent of node **k** is node **(k-1)/2**

# Numbering tree nodes

**Number node starting at root row by row, left to right**

**Same order as level-order traversal**

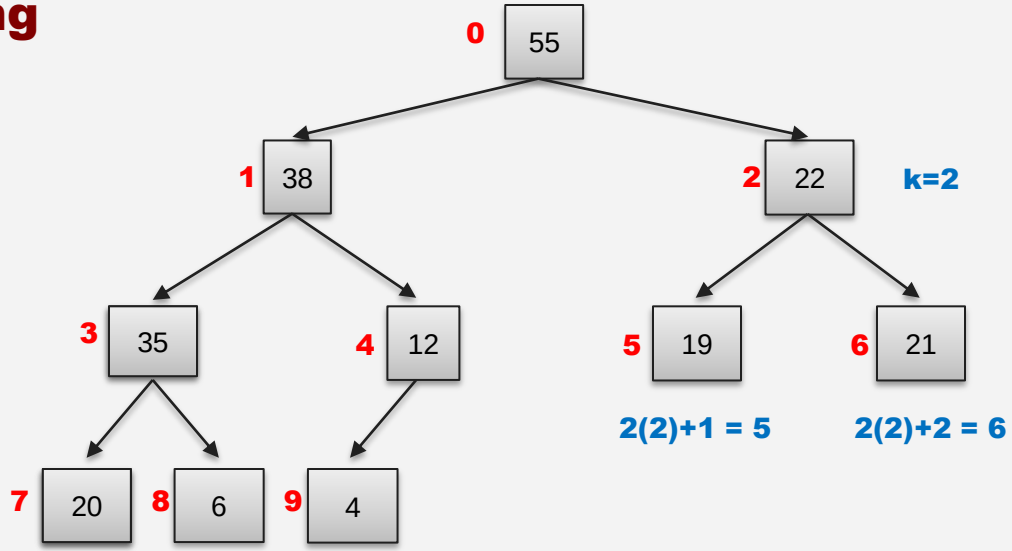



Children of node **k** are nodes **2k+1** and **2k+2**
Parent of node **k** is node **(k-1)/2**

# Numbering tree nodes

**Number node starting at root row by row, left to right**

**Same order as level-order traversal**



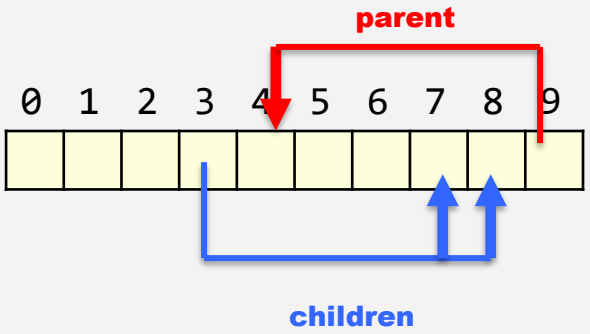

Children of node **k** are nodes **2k+1** and **2k+2**
Parent of node **k** is node **(k-1)/2**

# Represent tree with array

- Store node number i in:

  **b[i]**
- Children of **b[k]** are:

  **b[2k +1]** and **b[2k +2]**
- Parent of **b[k]** is **b[**$\lfloor (k-1)/2 \rfloor$**]**

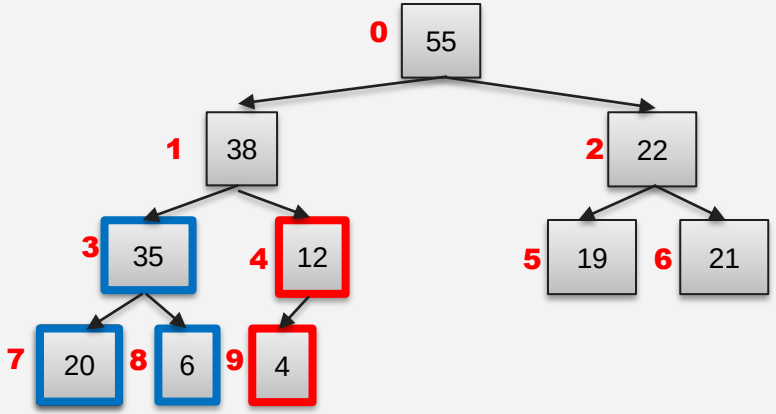


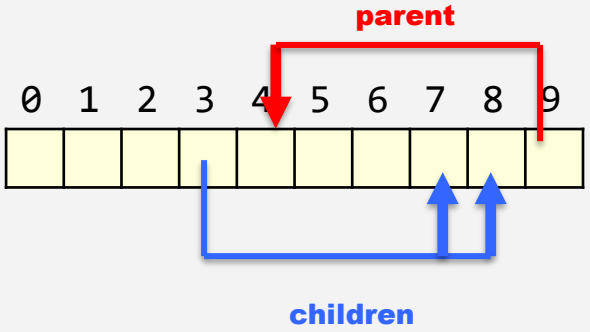
# Represent tree with array

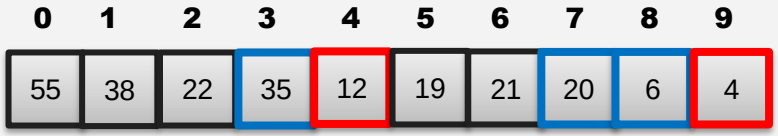
- Store node number i in:

  **b[i]**
- Children of **b[k]** are:

  **b[2k +1]** and **b[2k +2]**
- Parent of **b[k]** is **b[**$\lfloor (k-1)/2 \rfloor$**]**




سوال؟

# Constructor

```
class Heap<E> {
  E[] b; // heap is b[0..n-1]
  int n;

  /** Create heap with max size */
  public Heap(int max) {
    b= new E[max];
    // n == 0, so heap invariant holds
    // (completeness & heap-order)
  }
}
```

# peek()

```
/** Return largest element
  * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    return b[0];   // largest value at root.
}
```

# add()  (assuming enough room in array)

```
class Heap<E> {

  /** Add e to the heap */
  public void add(E e) {
    b[n]= e;
    n= n + 1;
    bubbleUp(n - 1); // on next slide
  }
}
```

# add()  heap is in b[0..n-1]

```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {

    // inv: p is parent of k and every element
    // except perhaps k is <= its parent
    while (                                    ) {




    }
}
```

# add()  heap is in b[0..n-1]

```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {
    int p = (k-1)/2;
    // inv: p is parent of k and every element
    // except perhaps k is <= its parent
    while (                                    ) {




    }
}
```

# add()  heap is in b[0..n-1]

```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {
    int p = (k-1)/2;
    // inv: p is parent of k and every element
    // except perhaps k is <= its parent
    while ( k > 0  &&  b[k] > (b[p]) ) {




    }
}
```

# add()  heap is in b[0..n-1]

```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {
    int p = (k-1)/2;
    // inv: p is parent of k and every element
    // except perhaps k is <= its parent
    while ( k > 0  &&  b[k] > (b[p]) ) {

        swap(b, k, p);
        k= p;
        p= (k-1)/2;

    }
}
```


سوال؟

# poll()  heap is in b[0..n-1]

```
/** Remove and return the largest element
  * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E v= b[0];    // largest value at root
    n= n - 1;     // move last
    b[0]= b[n];   // element to root
    bubbleDown(); // on next slide
    return v;
}
```

# poll()

```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {


   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while (                          ) {




    }
}
```

# poll()

```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
   int k= 0;
   int c= biggerChild(k); // on next slide
   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while (                          ) {




   }
}
```

# poll()

```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
   int k= 0;
   int c= biggerChild(k); // on next slide
   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while ( c < n &&  b[k] < b[c] ) {




   }
}
```

# poll()

```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
   int k= 0;
   int c= biggerChild(k); // on next slide
   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while ( c < n &&  b[k] < b[c] ) {

      swap(b, k, c);
      k= c;
      c= biggerChild(k);

   }
}
```

# poll()

```
/** Return index of bigger child of node k */
public int biggerChild(int k) {
    int c= 2*k + 2;      // k's right child
    if (c >= n || b[c-1] > b[c])
       c= c-1;
    return c;
}
```

# Efficiency

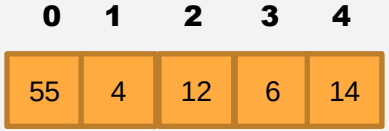
| class PriorityQueue<E> { | TIME* |
|---|---|
| boolean **add**(E e); //insert e. | O(log n) |
| E **poll**(); //remove & return max elem. | O(log n) |
| E **peek**(); //return min elem. | O(1) |
| boolean **contains**(E e); | O(n) |
| boolean **remove**(E e); | O(n) |
| int **size**(); | O(1) |
| } | |


سوال؟

# مرتب سازی هرمی

***استفاده از هرم برای مرتب سازی***

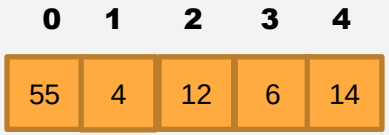
# Heapsort




**Goal: sort this array in place**
**Approach: turn the array into a heap and then poll repeatedly**

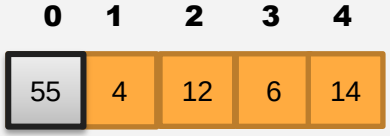
# Heapsort

**// Make b[0..n-1] into a max-heap (in place)**



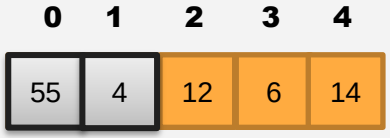
# Heapsort

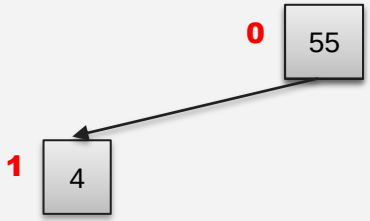
**// Make b[0..n-1] into a max-heap (in place)**




# Heapsort

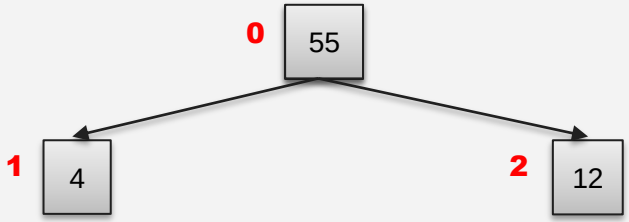
**// Make b[0..n-1] into a max-heap (in place)**



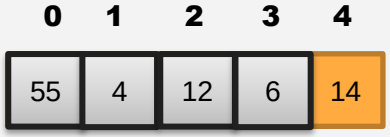
# Heapsort

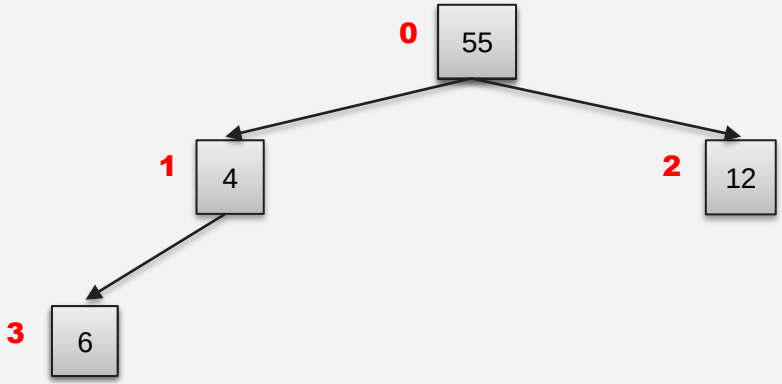
**// Make b[0..n-1] into a max-heap (in place)**

# Heapsort

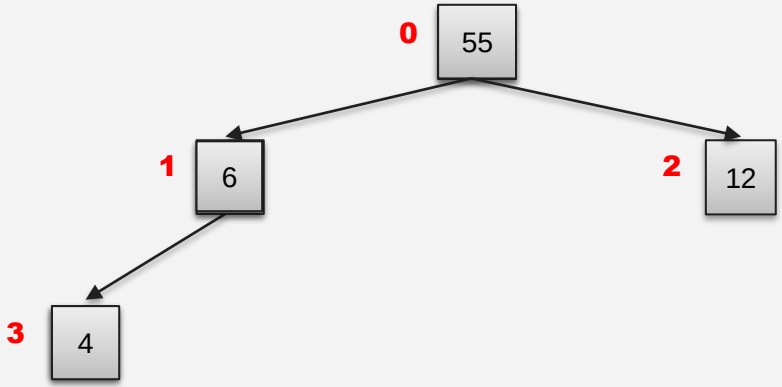
**// Make b[0..n-1] into a max-heap (in place)**



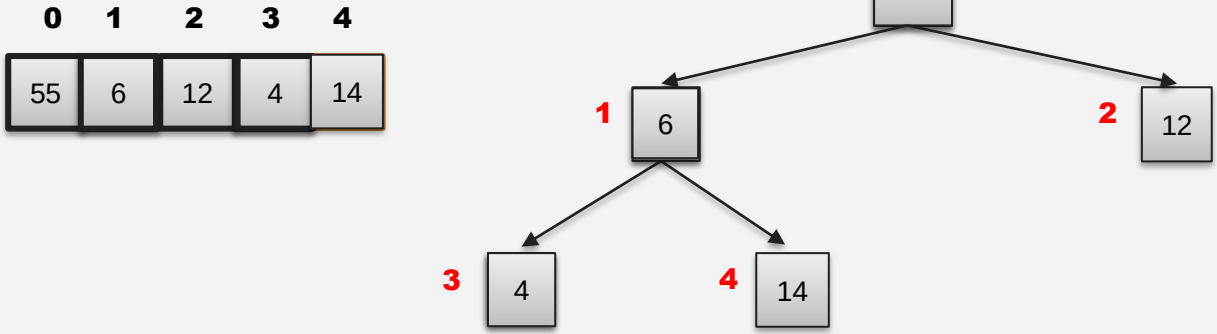
# Heapsort

**// Make b[0..n-1] into a max-heap (in place)**

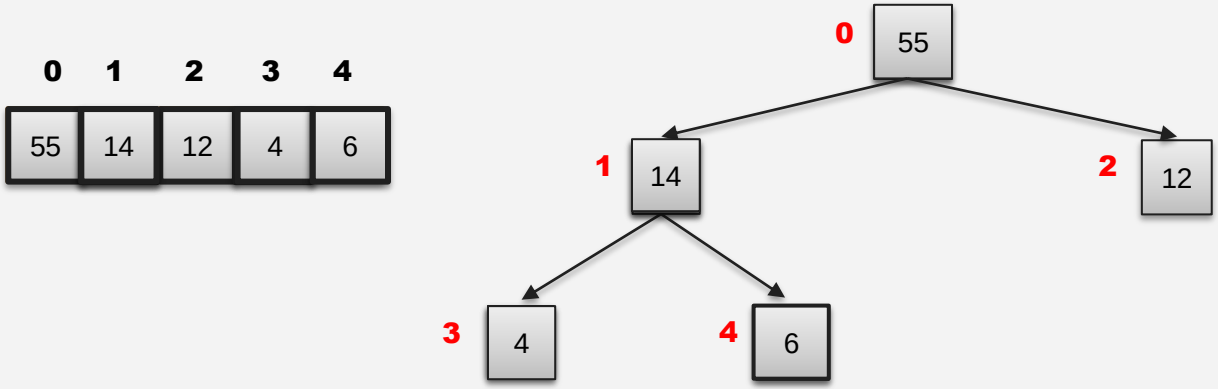
# Heapsort

**// Make b[0..n-1] into a max-heap (in place)**



# Heapsort

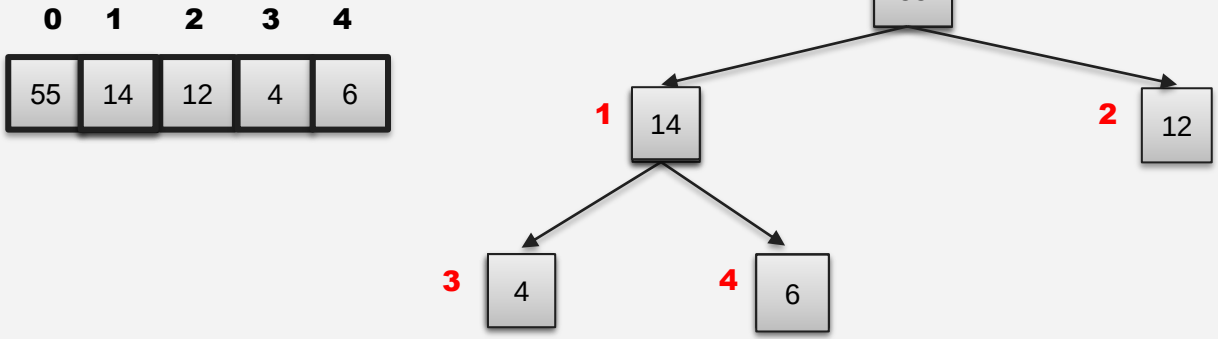
**// Make b[0..n-1] into a max-heap (in place)**




سوال؟

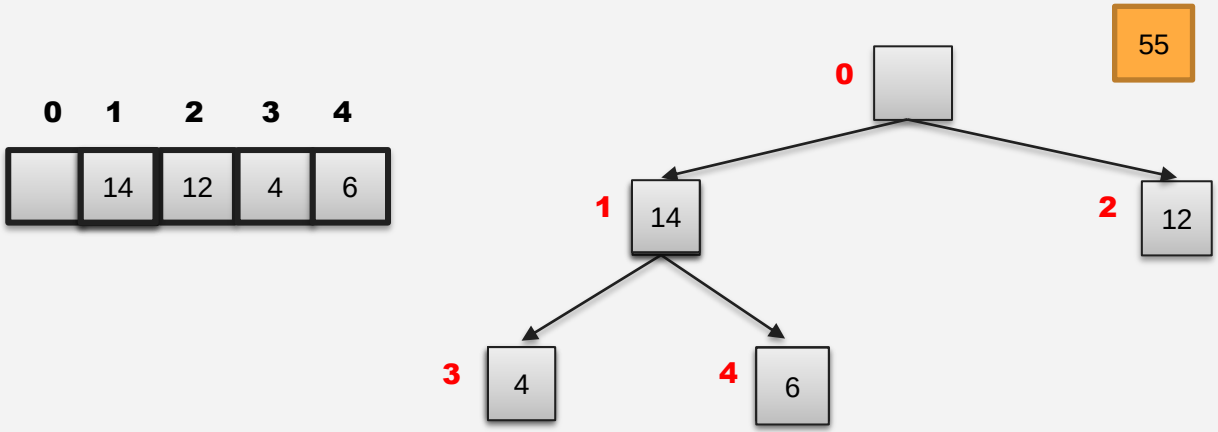
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```



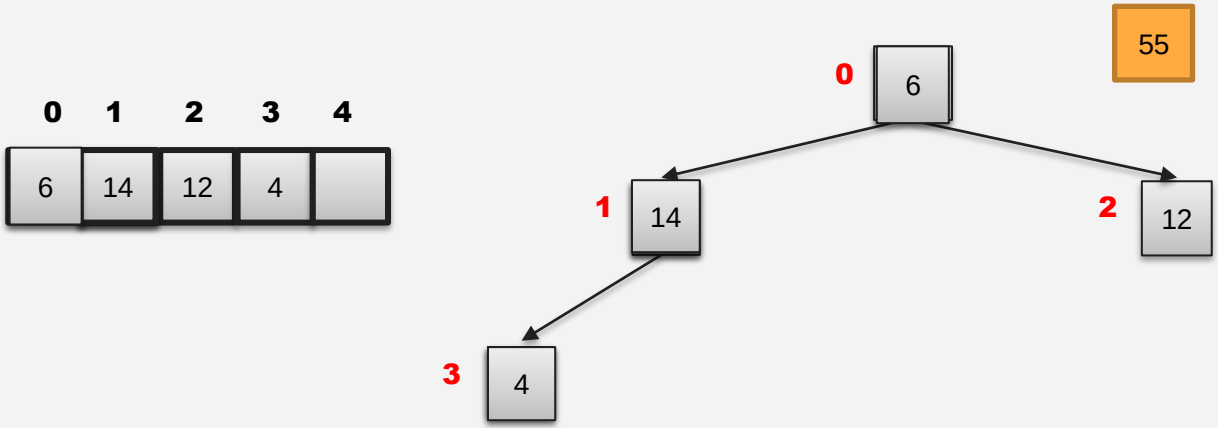
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```



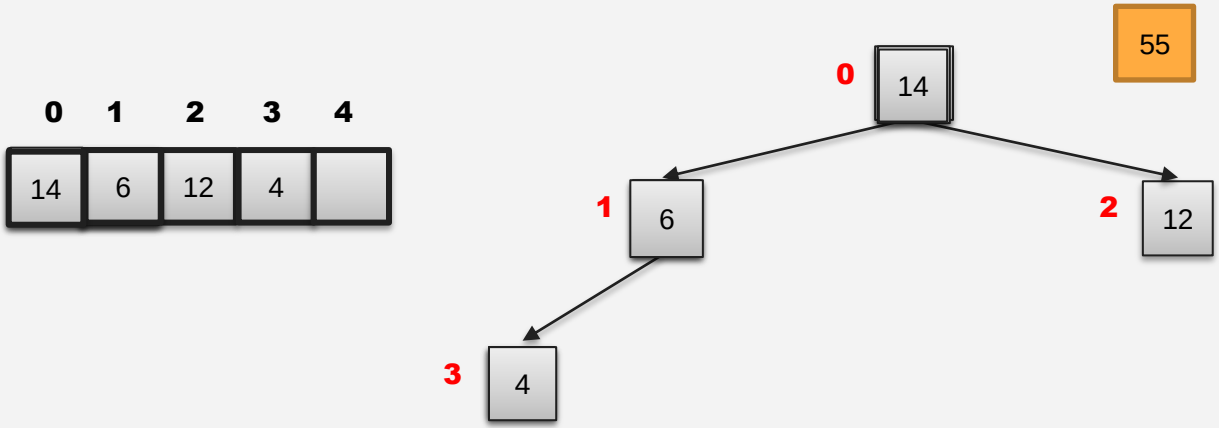
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```



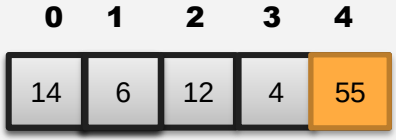
# Heapsort

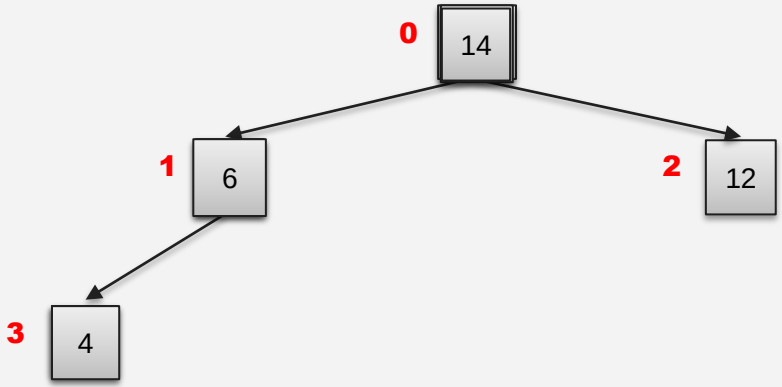
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```



# Heapsort

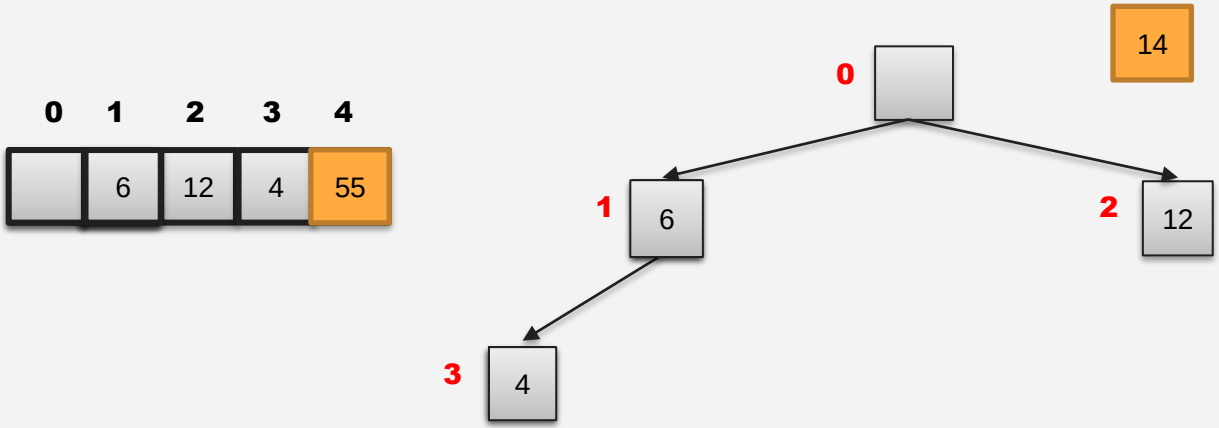
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```



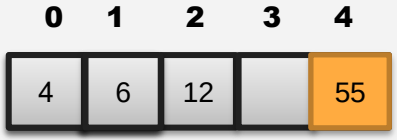
# Heapsort

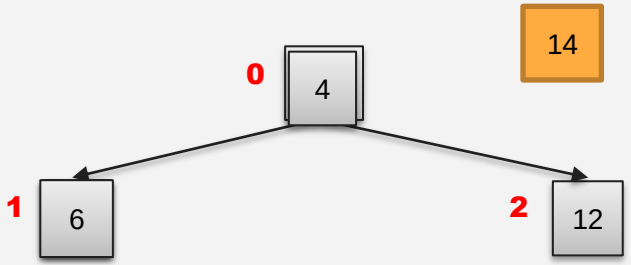
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```



# Heapsort

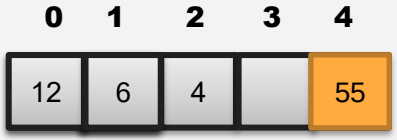
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```

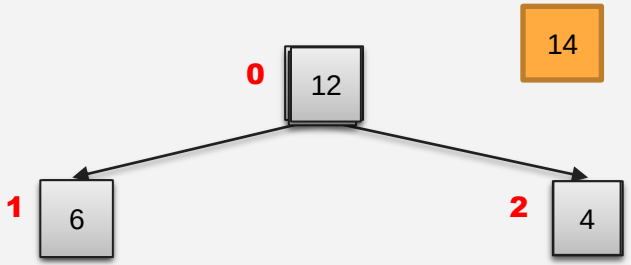


# Heapsort

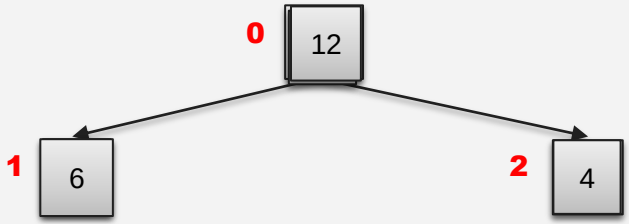
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
           b[k]= poll  – i.e., take max element out of heap.
      }
```



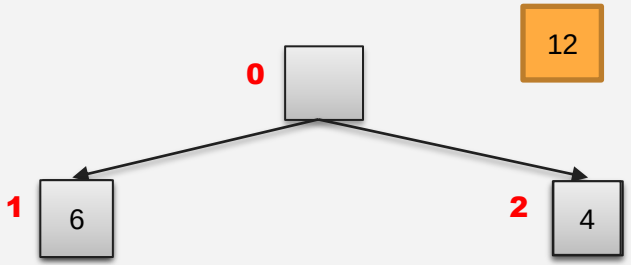
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
           b[k]= poll  – i.e., take max element out of heap.
      }
```

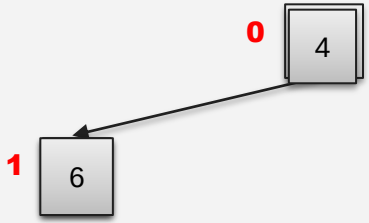
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```

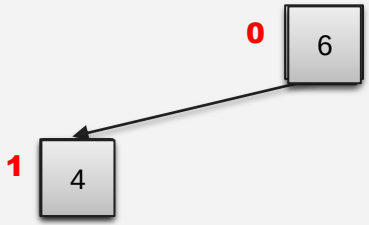
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```

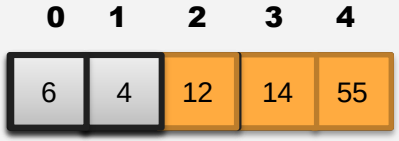

# Heapsort

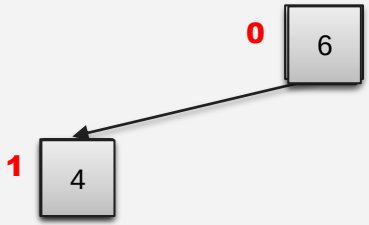
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
          b[k]= poll  – i.e., take max element out of heap.
    }
```


# Heapsort

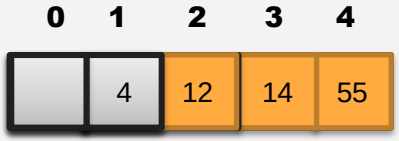
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
           b[k]= poll  – i.e., take max element out of heap.
      }
```

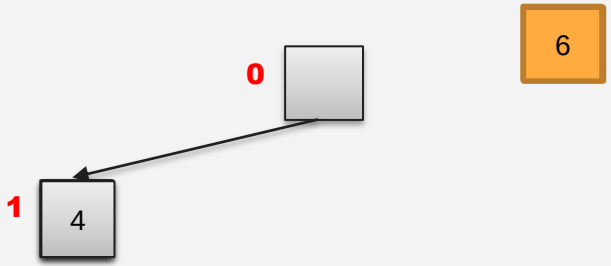


# Heapsort

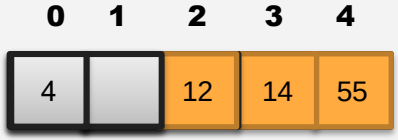
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
           b[k]= poll  – i.e., take max element out of heap.
      }
```



# Heapsort

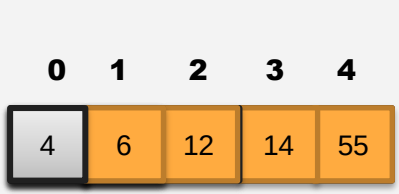
```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
           b[k]= poll  – i.e., take max element out of heap.
      }
```





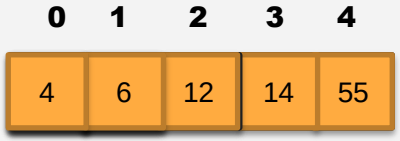
# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```




# Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
     for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
      }
```




سوال؟