ساختمان داده و الگوریتم ها (CE203)

جلسه دهم:
هرم و مرتب سازی هرمی

سجاد شیرعلی شهرضا

پاییز 1401

شنبه، 7 آبان 1401

Based on Washington University CSE 373 (Autumn 2020 by Hunter Schafer), University of North Florida COP2220 (Chip Klostermeyer), and University of Illinois Chicago CS 201 (Spring 2014 by Bing Liu) slides

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 6
- امتحانک دوم:
  - دوشنبه همین هفته، 9 آبان 1401
  - در طی ساعت کلاس به صورت حضوری و تشریحی

Interface vs. Implementation

Interface: the operations of an ADT

Implementation: the code of a data structure

Interface vs. Implementation

Interface: the operations of an ADT

- What you see on documentation web pages

Implementation: the code of 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 of 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 of 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 of 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 of 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

Examples of 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	*This lecture!*

Examples of 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	World Cup News
2	Global Warming Solutions

Java Implementation

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
  - poll() must search the list
  - peek() must search the list

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
  - poll() highest priority element at front
  - peek() same

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

Heap

- A binary tree satisfying 2 properties:

Do not confuse with heap memory.
Different use of the word heap.

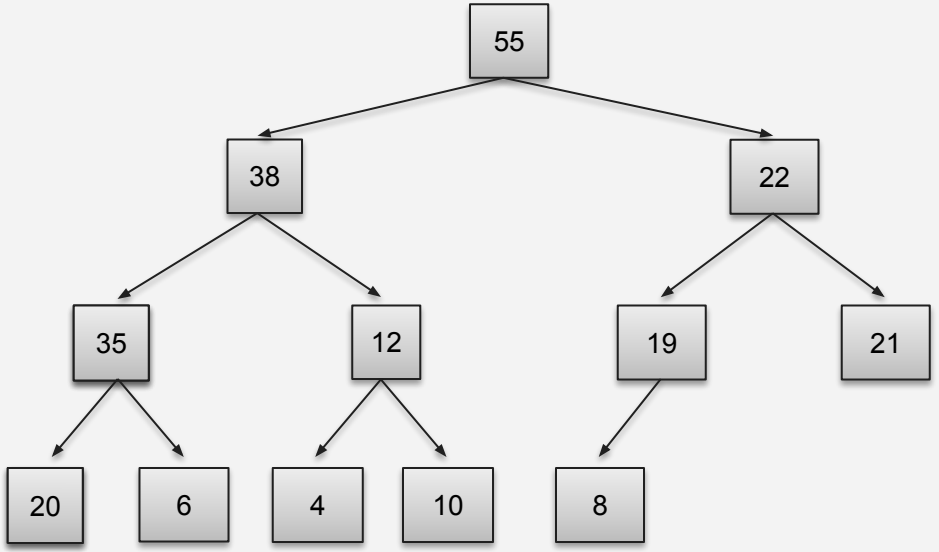
Heap

- A binary tree satisfying 2 properties:
  - **Completeness**
    - Every level of the tree (except last) is completely filled
    - Nodes on the last level are as far left as possible

Do not confuse with heap memory.
Different use of the word heap.

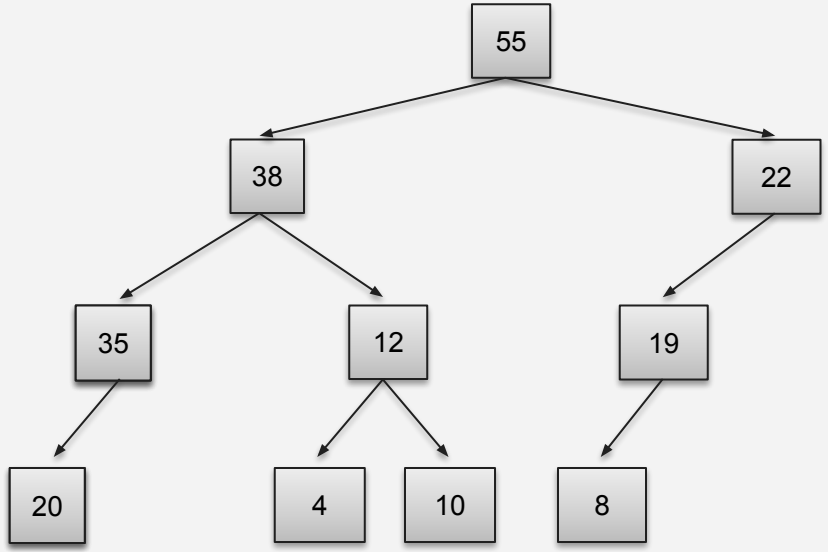
Completeness

- Every level of the tree (except last) is completely filled
- Nodes on the last level are as far left as possible



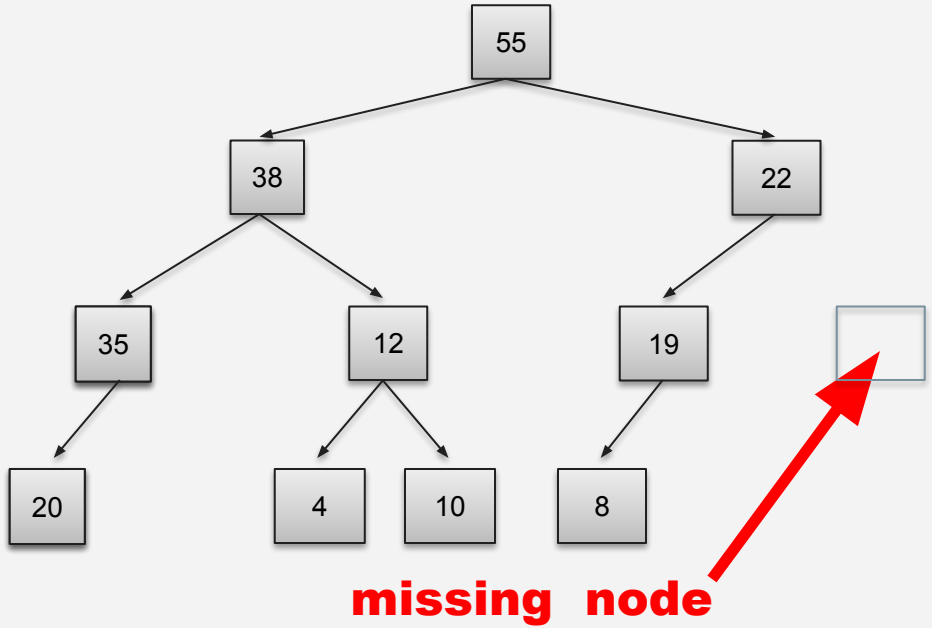
Completeness

- Not a heap because:



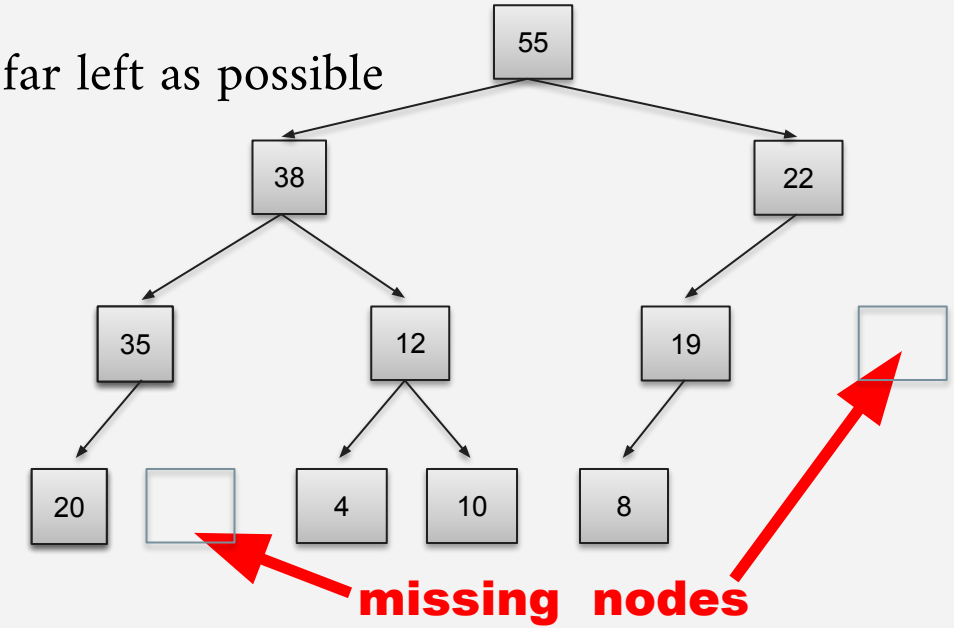
Completeness

- 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



Heap

- A binary tree satisfying 2 properties:
  - **Completeness**
    - Every level of the tree (except last) is completely filled
    - Nodes on the last level are as far left as possible
  - **Heap-order**

Heap

- A binary tree satisfying 2 properties:
  - **Completeness**
    - Every level of the tree (except last) is completely filled
    - Nodes on the last level are as far left as possible
  - **Heap-order**
    - Max-Heap: every element in tree is <= its parent

"max on top"

Heap

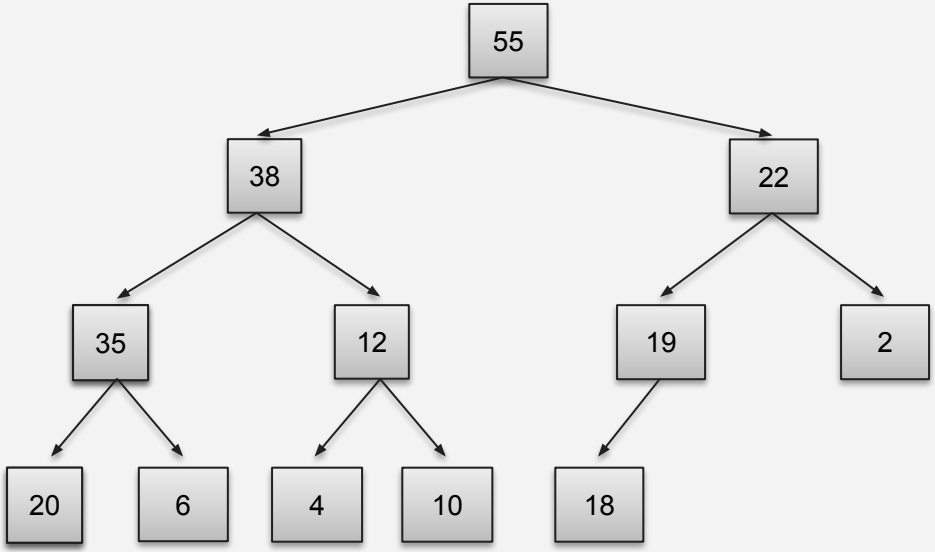
- A binary tree satisfying 2 properties:
  - **Completeness**
    - Every level of the tree (except last) is completely filled
    - Nodes on the last level are as far left as possible
  - **Heap-order**
    - Max-Heap: every element in tree is <= its parent
    - Min-Heap: every element in tree is >= its parent

"max on top"

"min on top"

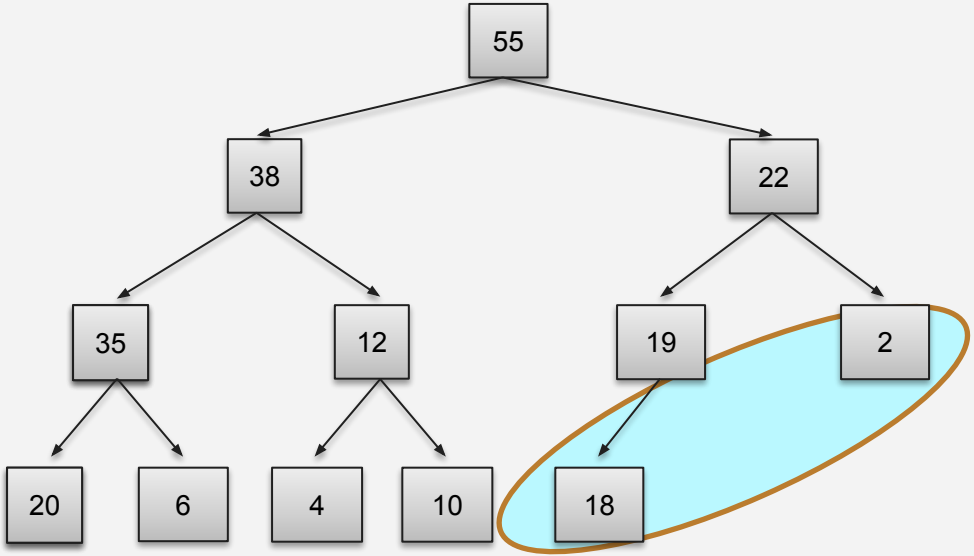
Heap-order (max-heap)

Every element is <= its parent



Heap-order (max-heap)

Every element is <= its parent



Note: Bigger elements can be deeper in the tree!

Heap

- A binary tree satisfying 2 properties:
  - **Completeness**
    - Every level of the tree (except last) is completely filled
    - Nodes on the last level are as far left as possible
  - **Heap-order**
    - Max-Heap: every element in tree is <= its parent
    - Min-Heap: every element in tree is >= its parent
- **Primary operations**
  - add(e): add a new element to the heap
  - poll(): delete the max element and return it
  - peek(): return the max element

Priority queues with Heaps

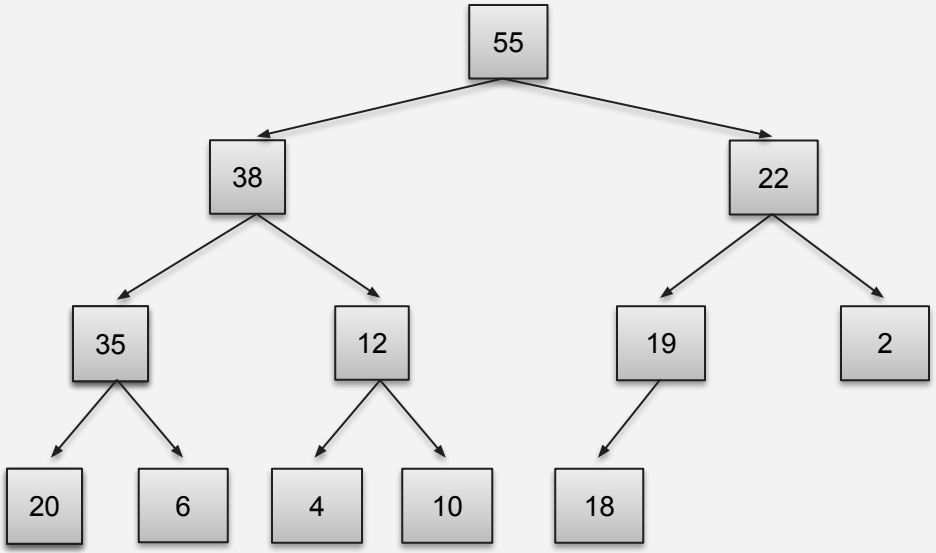
- Each heap node contains priority of a queue item
- Run time:
  - add(): O(log n)
  - poll(): O(log n)
  - peek(): O(1)
- No linear time operations: better than lists

سوال؟

الگوریتم های هرم

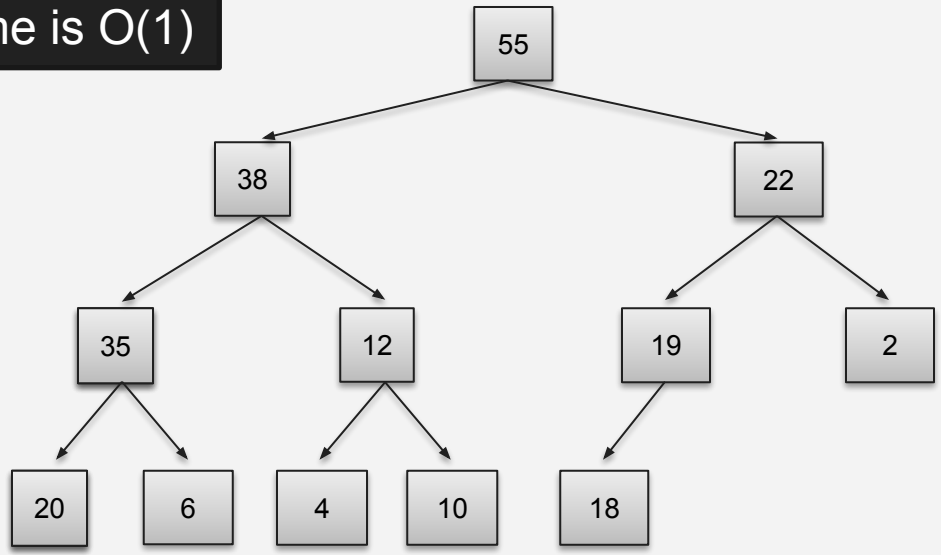
چگونگی تغییر هرم

Peek



Peek

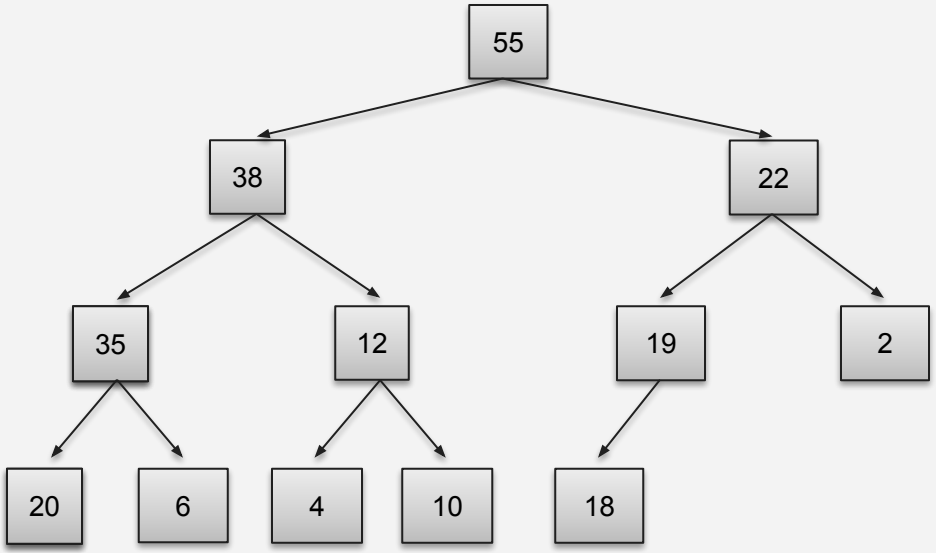
Time is O(1)



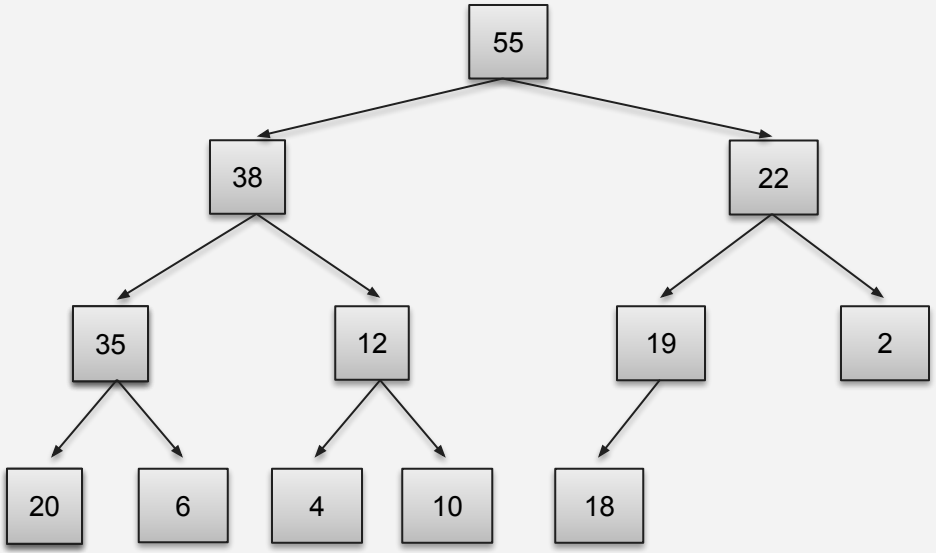
50

1. **Return root value**

Add

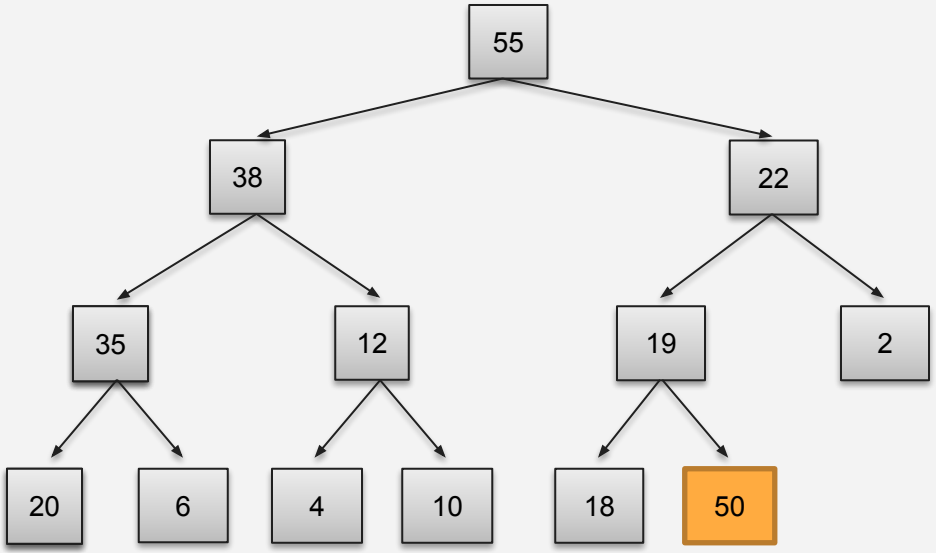


Add



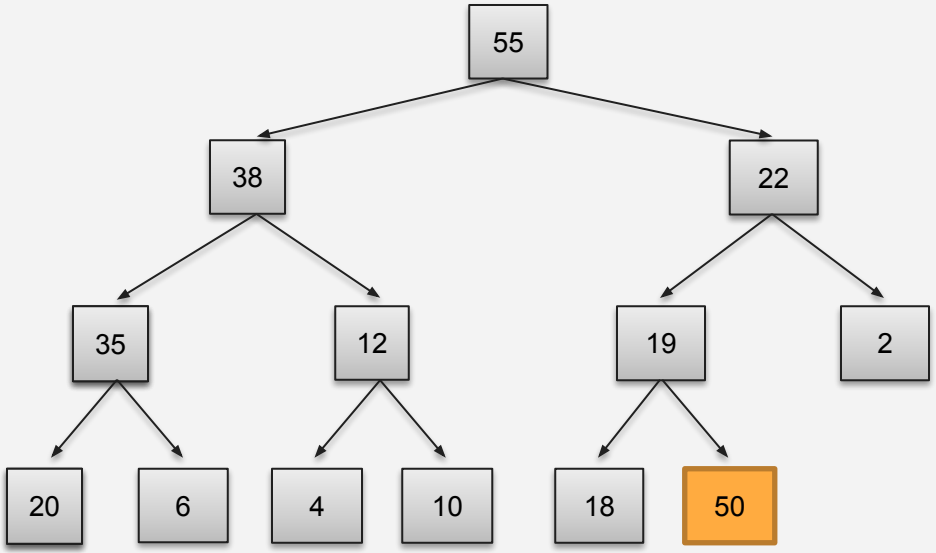
1. **Put in the new element in a new node (leftmost empty leaf)**

Add



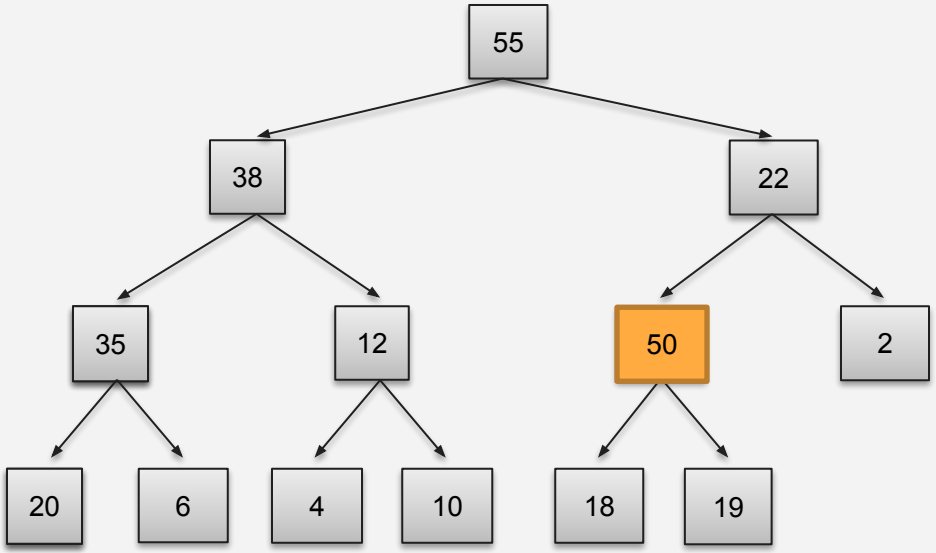
1. **Put in the new element in a new node (leftmost empty leaf)**

Add



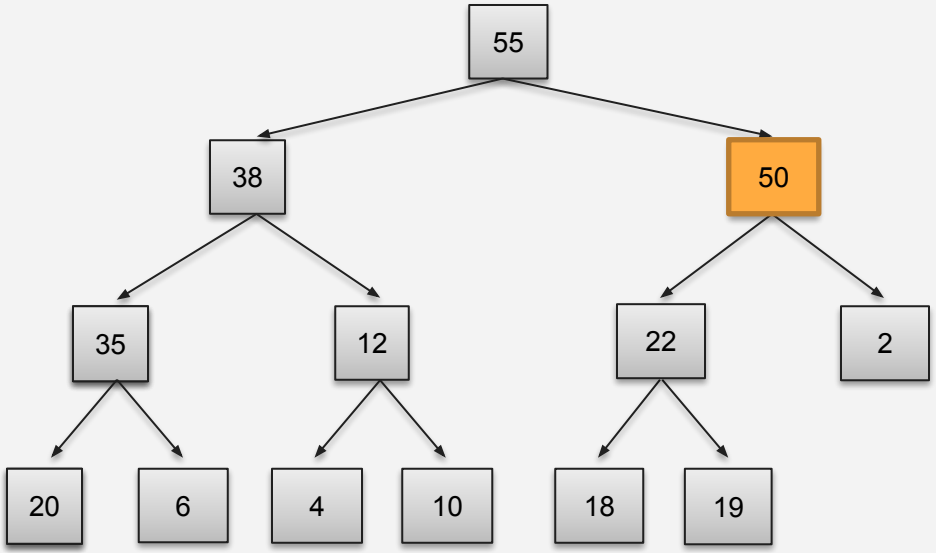
1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

Add



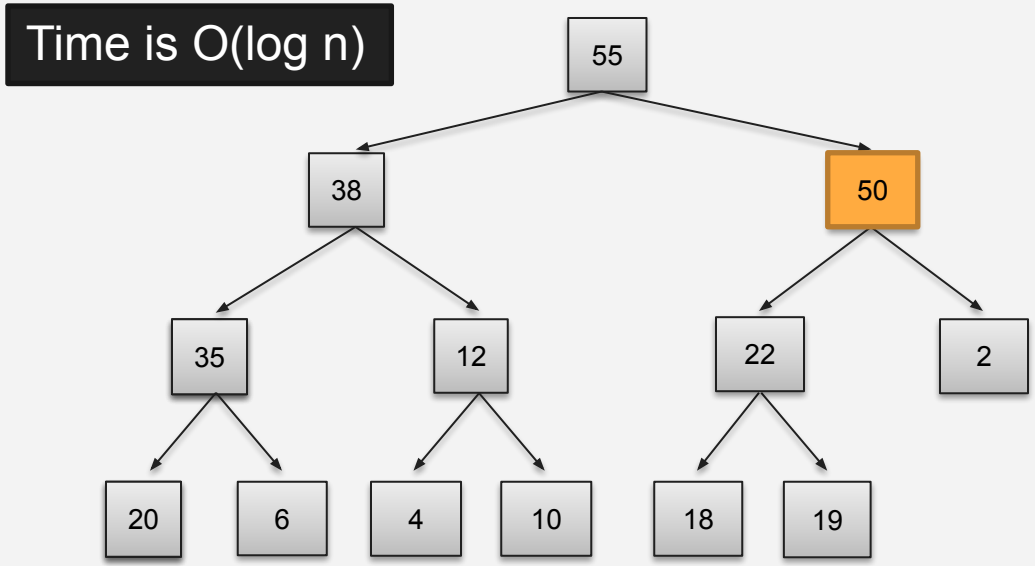
1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

Add



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

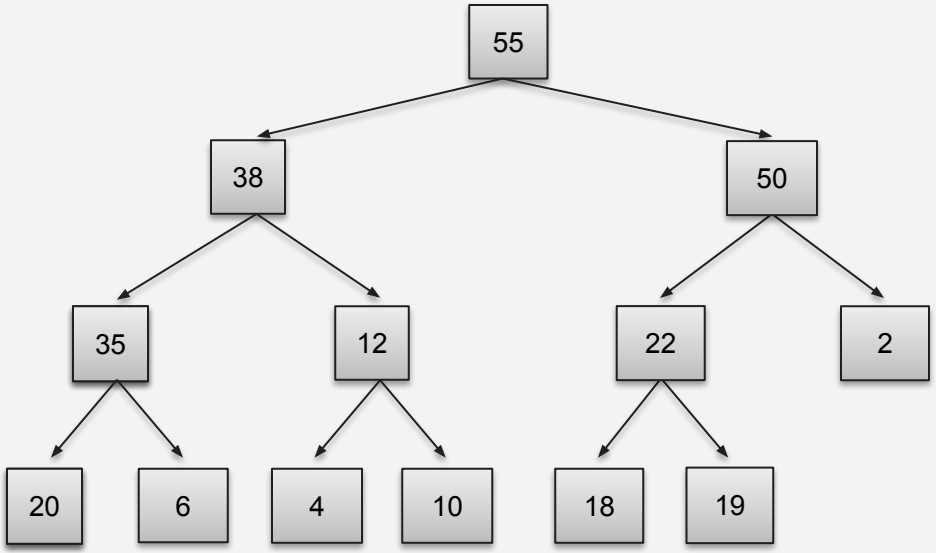
Add



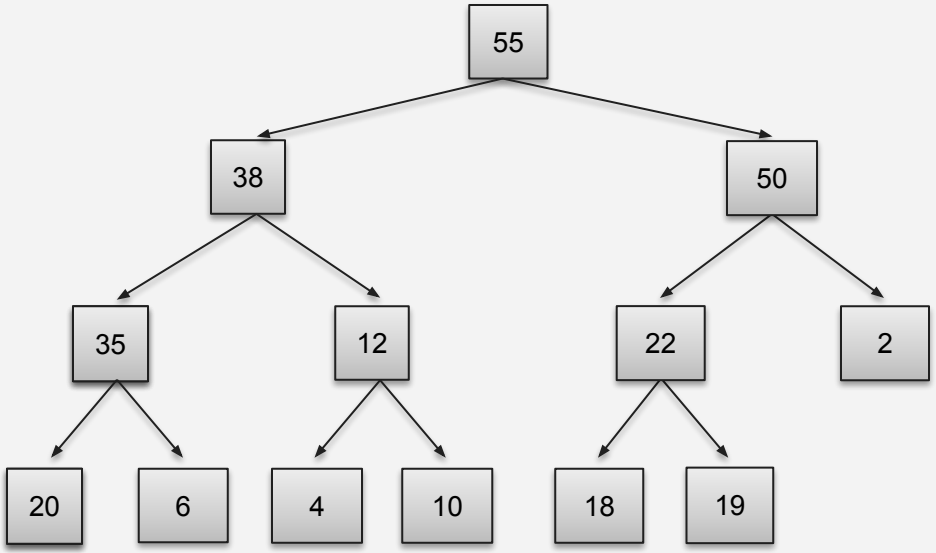
1. **Put in the new element in a new node (leftmost empty leaf)**
2. **Bubble new element up while greater than parent**

سوال؟

Poll

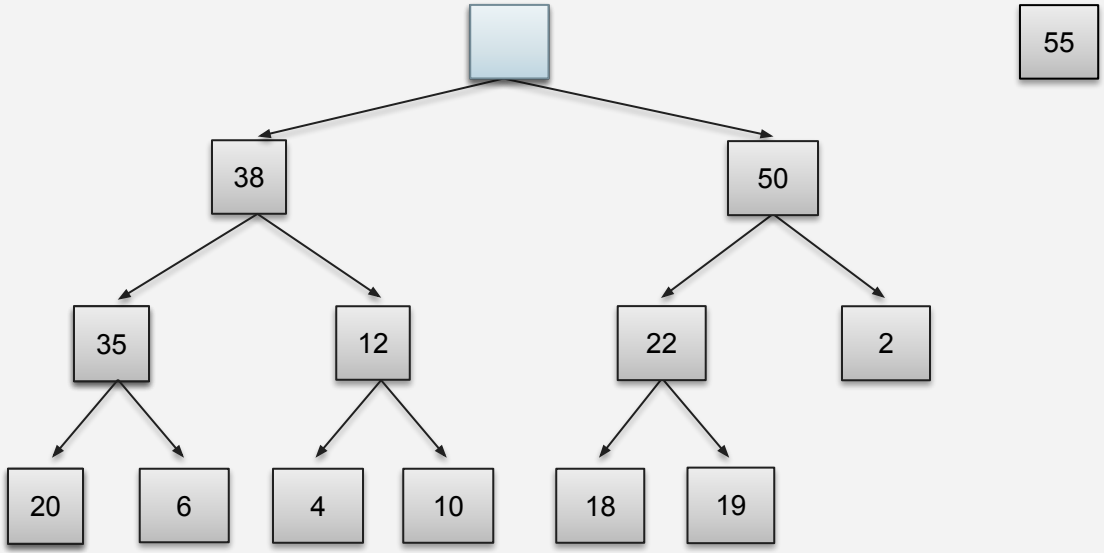


Poll



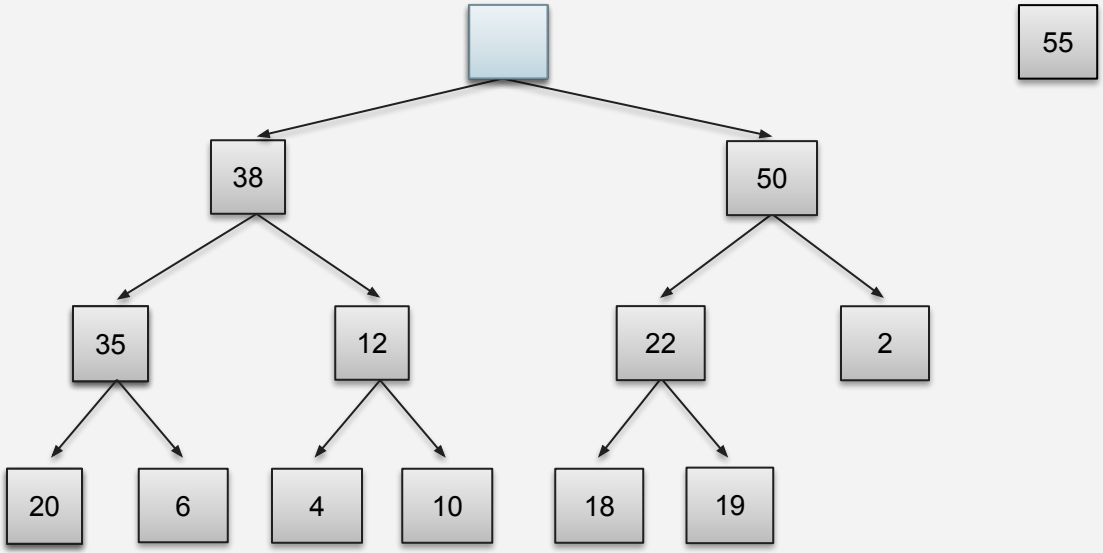
1. **Save root element in a local variable**

Poll



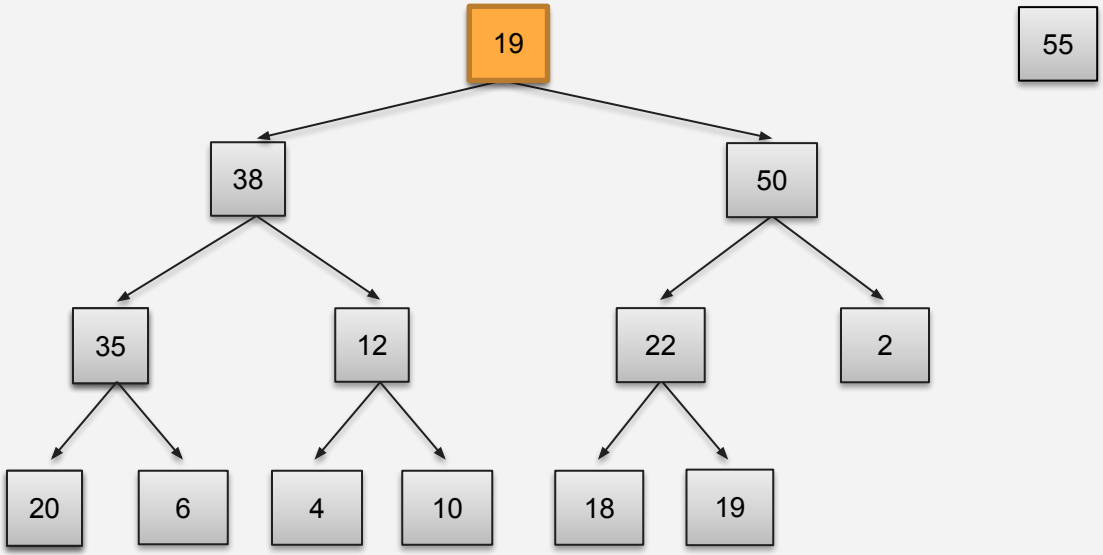
1. **Save root element in a local variable**

Poll



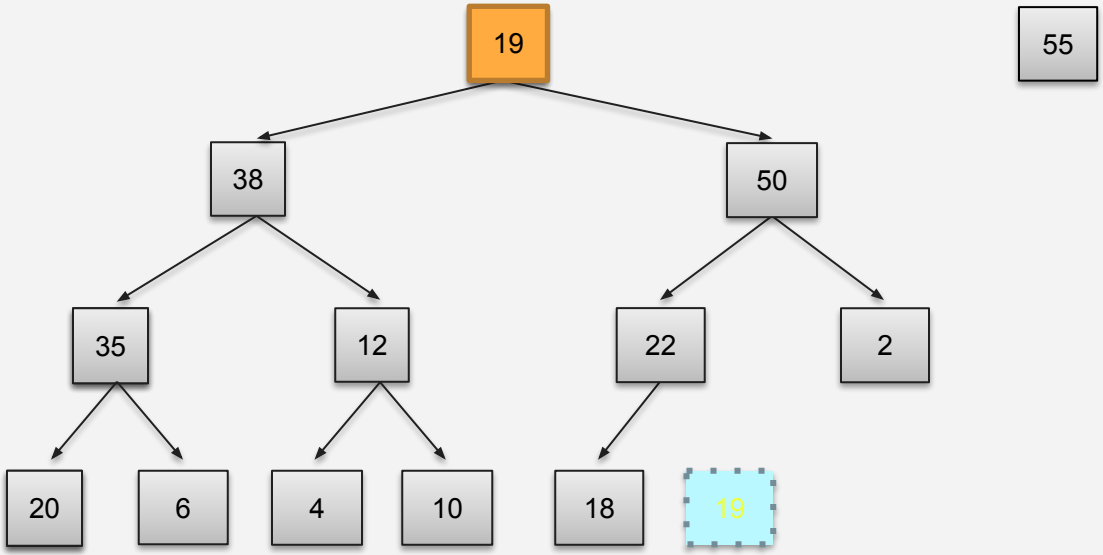
1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

Poll



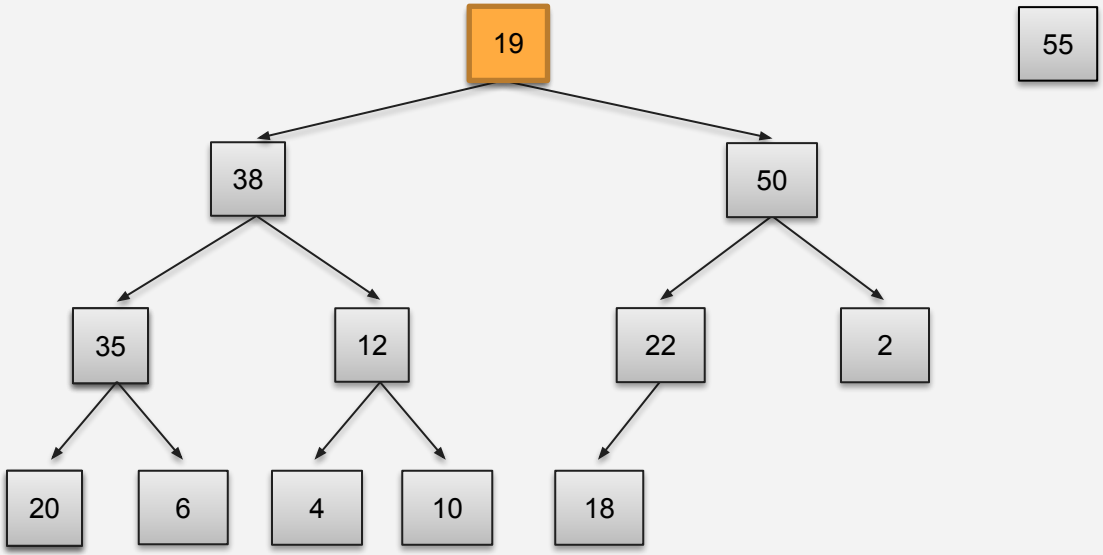
1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

Poll



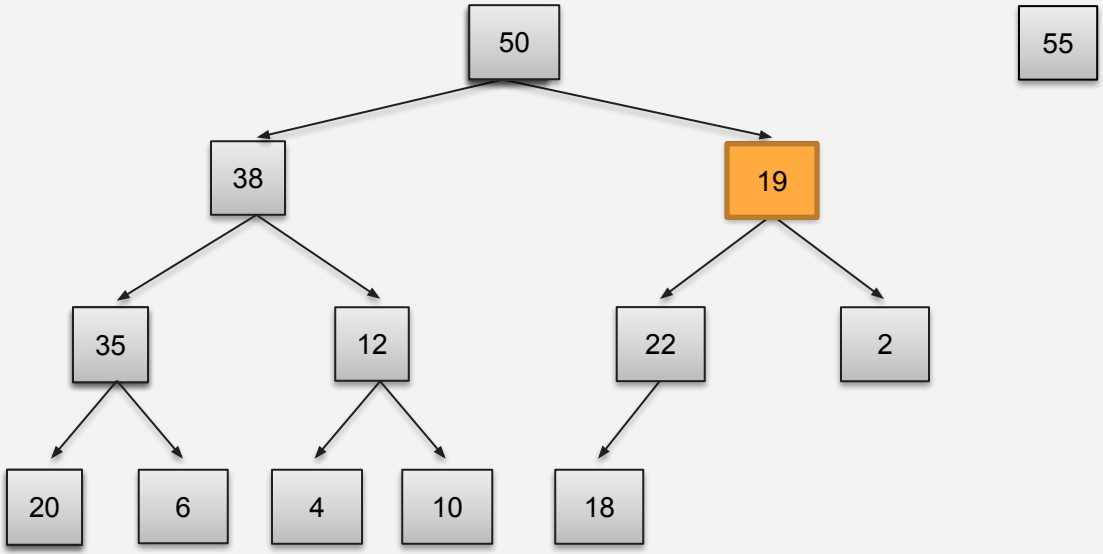
1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**

Poll



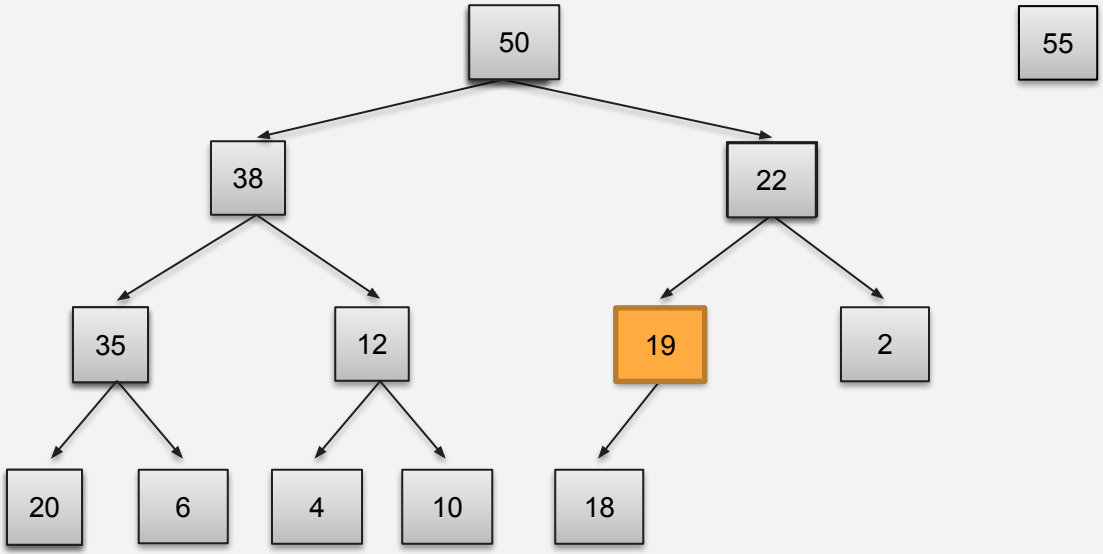
1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**

Poll



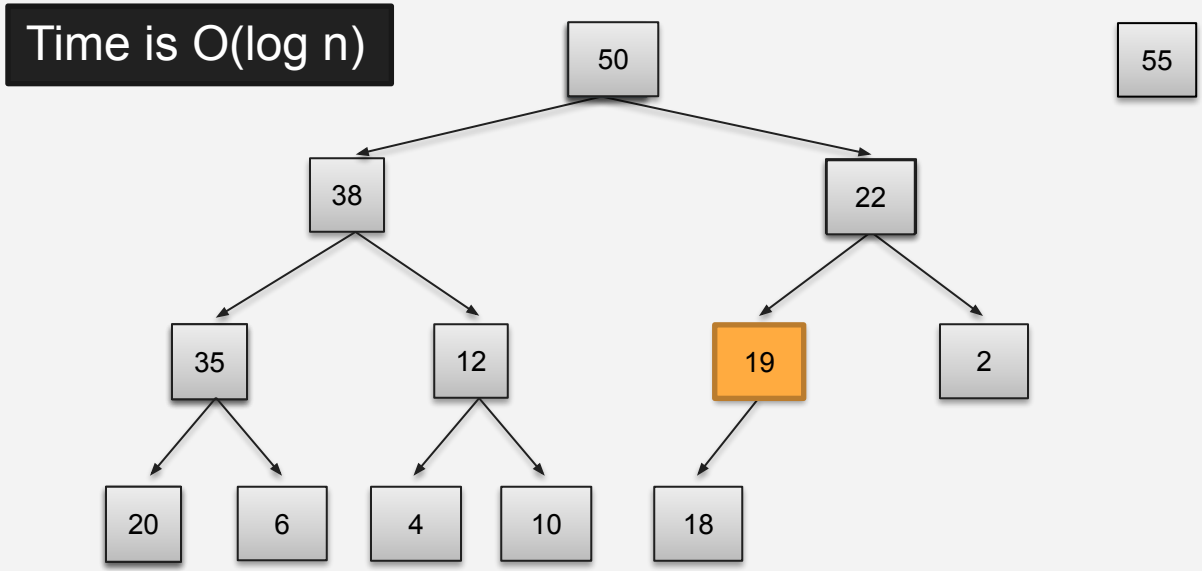
1. **Save root element in a local variable**
2. **Assign last value to root, delete last node.**
3. **While less than a child, switch with bigger child (bubble down)**

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)**

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)**

سوال؟

پیاده سازی هرم

چگونگی پیاده سازی هرم

Tree Implementation

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

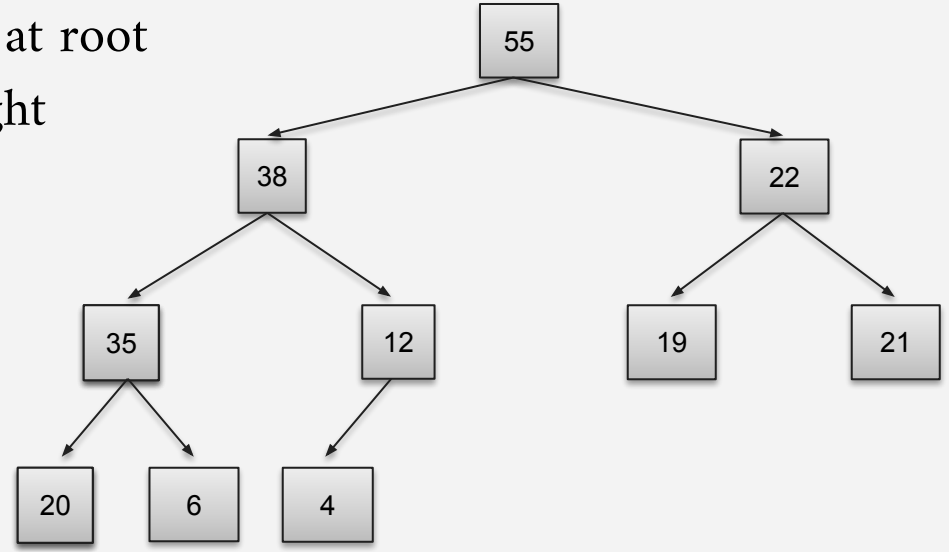
- Tree is complete, so more space-efficient implementation is possible …

Array Implementation

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

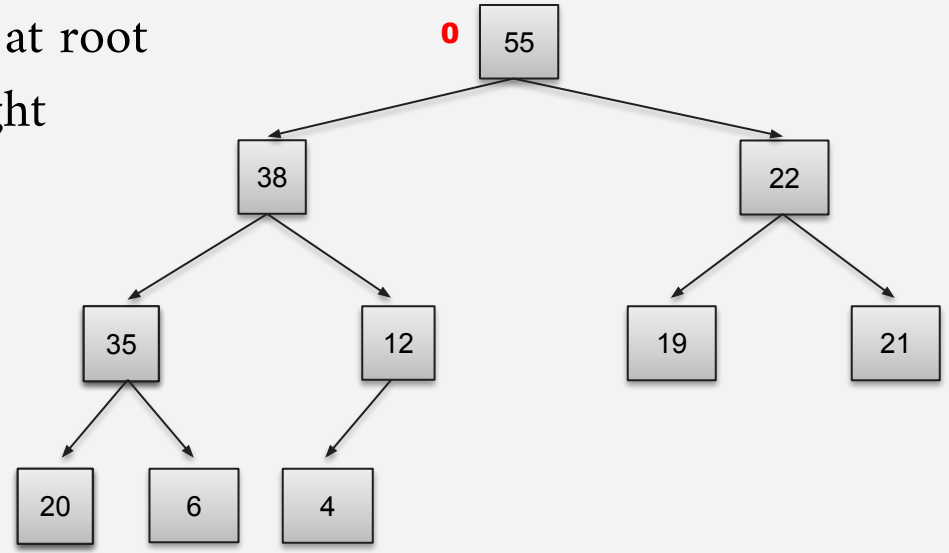
Numbering Tree Nodes

- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



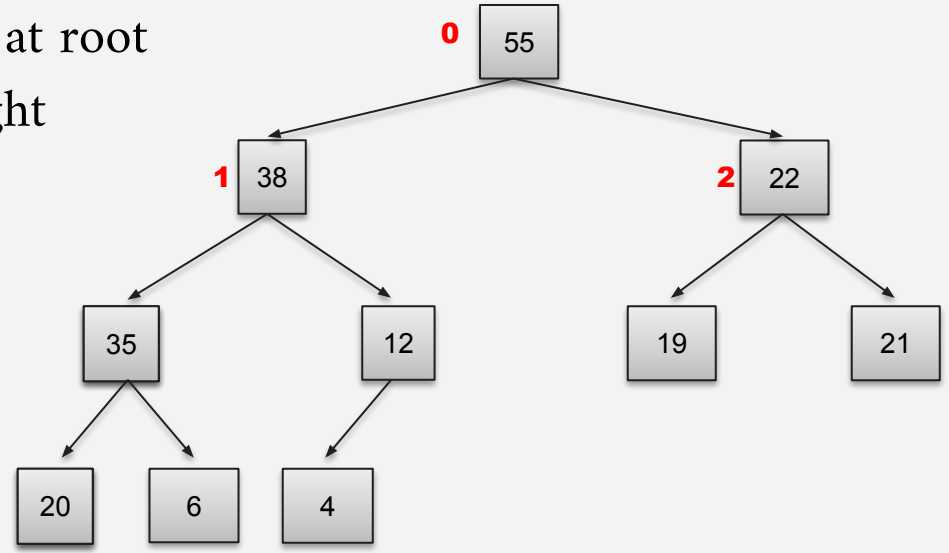
Numbering Tree Nodes

- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



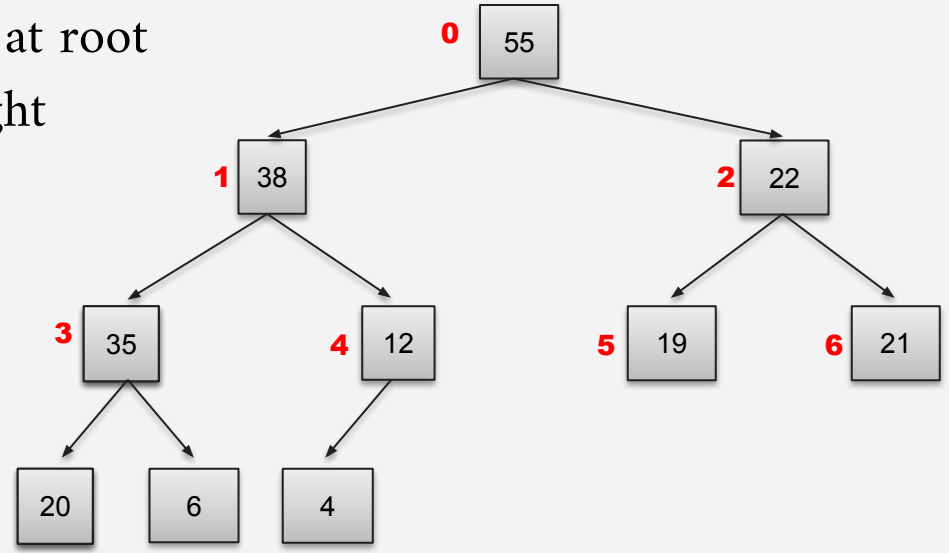
Numbering Tree Nodes

- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



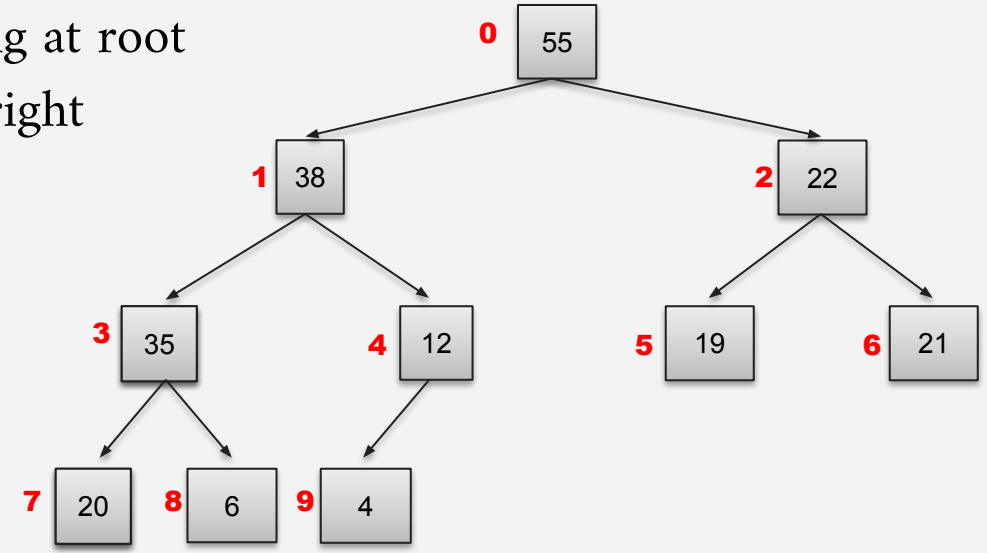
Numbering Tree Nodes

- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



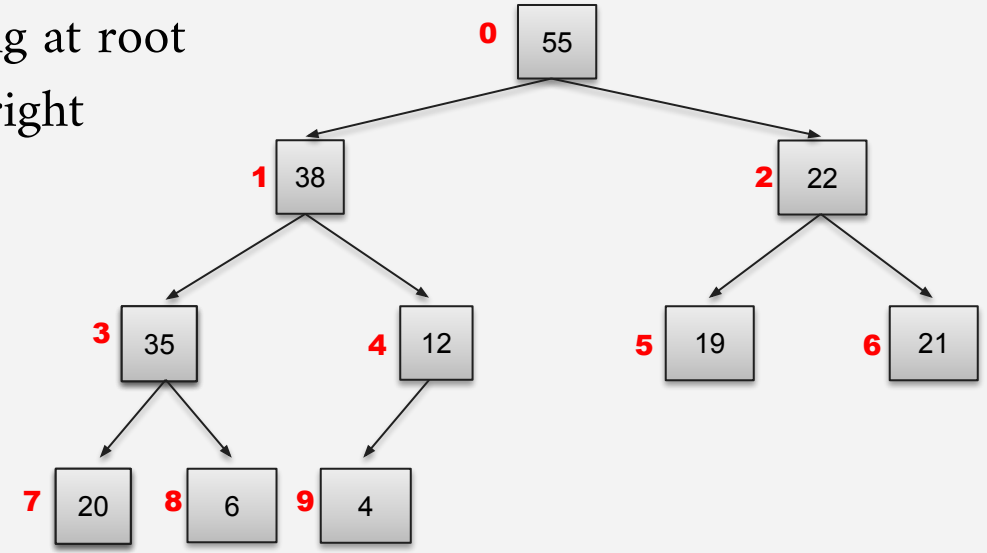
Numbering Tree Nodes

- 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

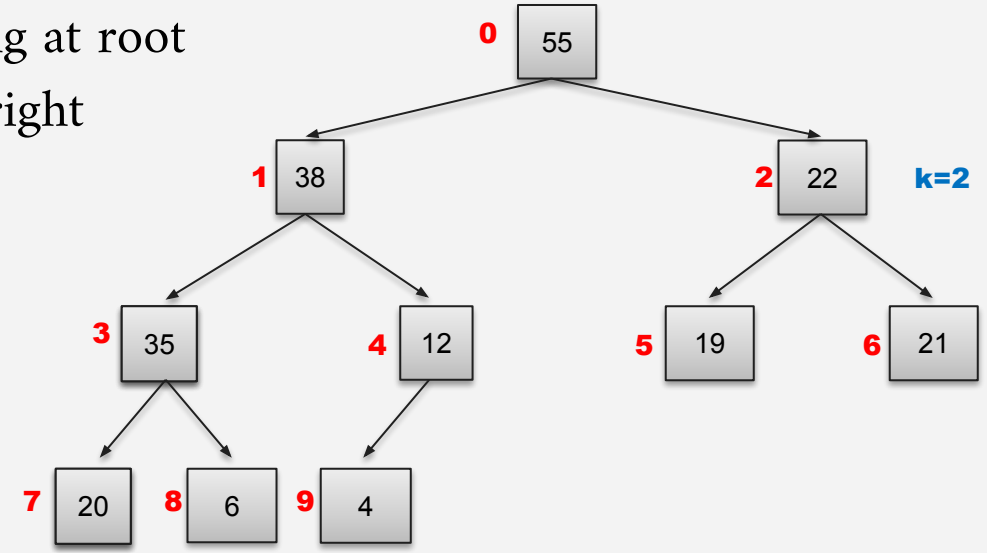


Children of node **k** are nodes **2k+1** and **2k+2**

Parent of node **k** is node **(k-1)/2**

Numbering Tree Nodes

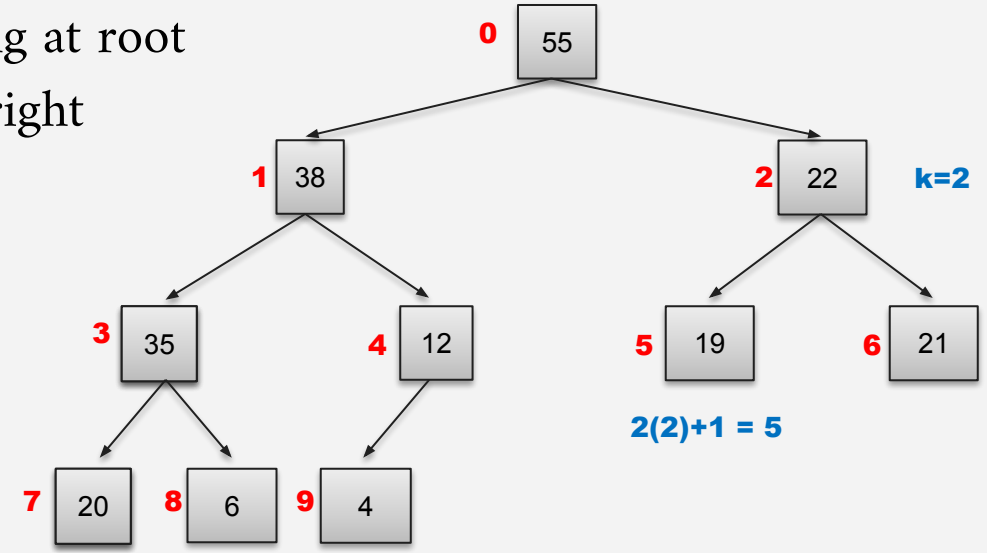
- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



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

Numbering Tree Nodes

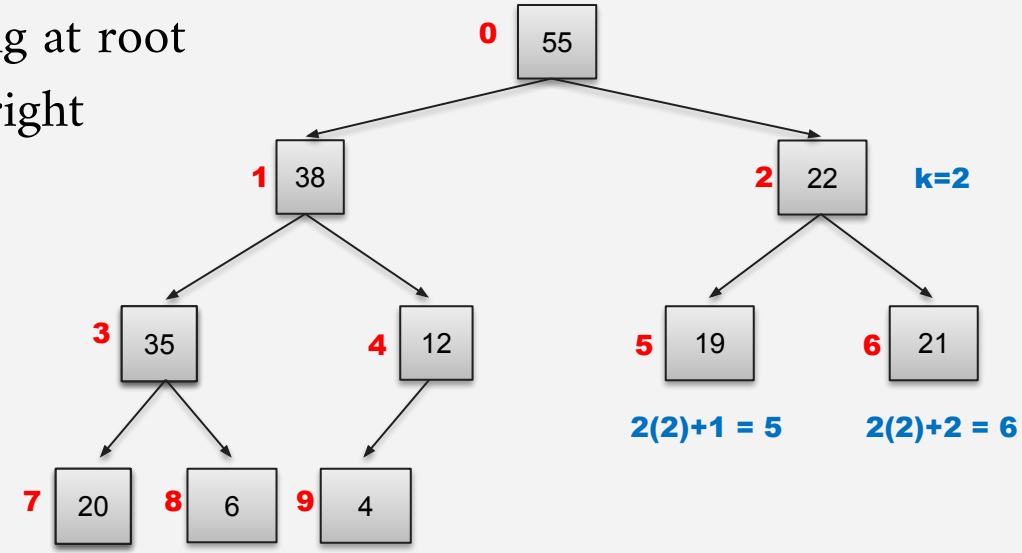
- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



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

Numbering Tree Nodes

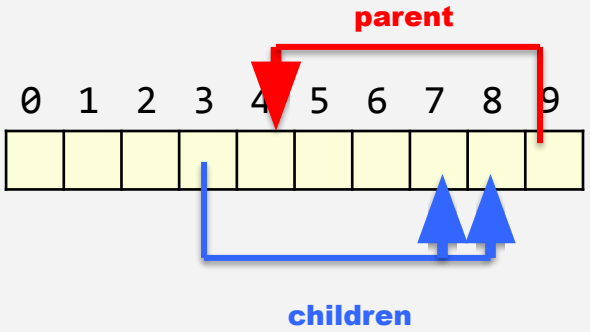
- Number node starting at root
- Row by row, left to right
- Same order as level-order traversal



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

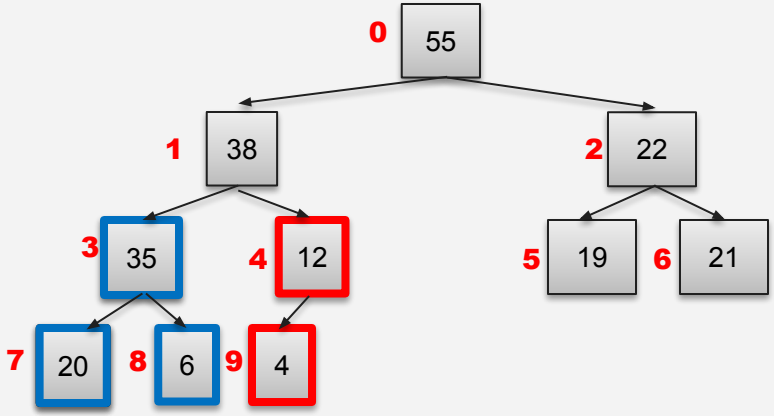
Represent Tree with Array

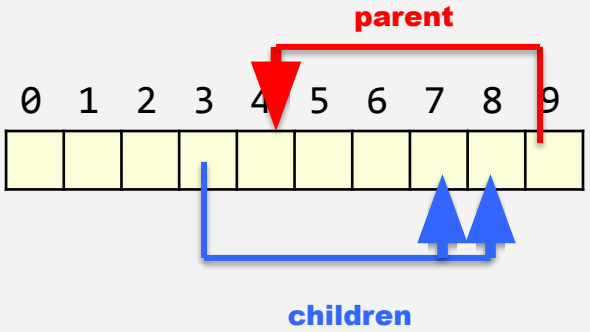
- Store node number **i** in **b[i]**
- Children of **b[k]**:
  - **b[2k+1]** and **b[2k+2]**
- Parent of **b[k]**:
  - **b[(k-1)/2]**

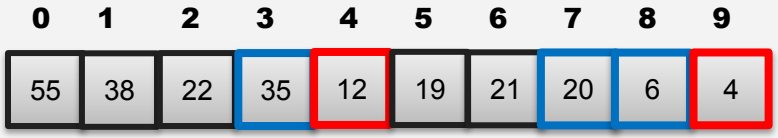


Represent Tree with Array

- Store node number **i** in **b[i]**
- Children of **b[k]**:
  - **b[2k+1]** and **b[2k+2]**
- Parent of **b[k]**:
  - **b[(k-1)/2]**







سوال؟

Constructor

heap is in b[0..n-1]

```
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
  }
}
```

BubbleUp

```
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 (                          ) {



    }
}
```

BubbleUp

```
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 (                          ) {



    }
}
```

BubbleUp

```
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]) ) {




    }
}
```

BubbleUp

```
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

```
/** 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
    heapify();    // on next slide
    return v;
}
```

Poll - bubbleDown

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


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




    }
}
```

Poll- bubbleDown

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

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

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

```
/** 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)
}
```

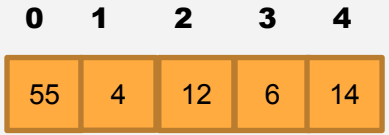
سوال؟

مرتب سازی هرمی

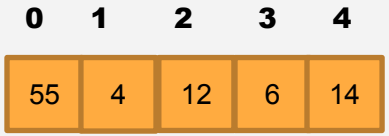
استفاده از هرم برای مرتب سازی

Heapsort

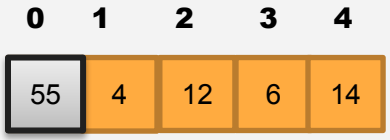
- Goal: sort this array in place
- Approach: turn the array into a heap and then poll repeatedly



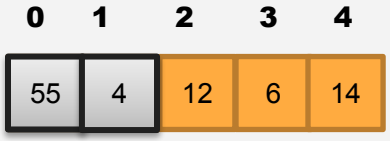
Convert into a Max-Heap (in-place)

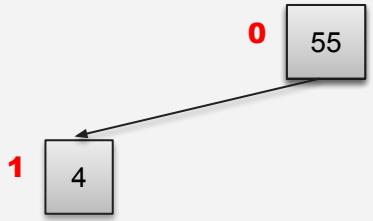


Convert into a Max-Heap (in-place)

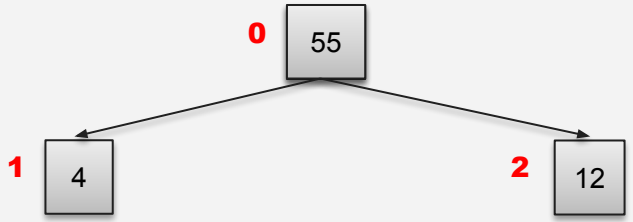


Convert into a Max-Heap (in-place)

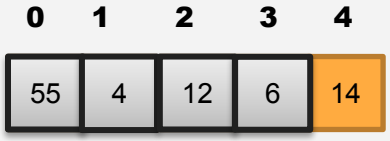


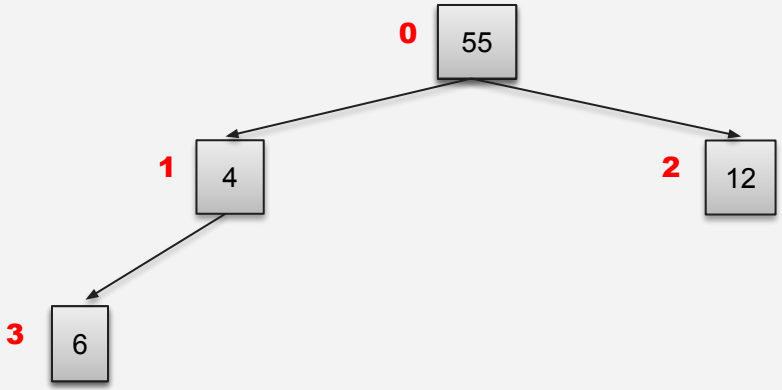


Convert into a Max-Heap (in-place)

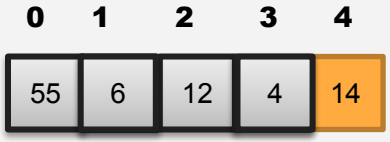


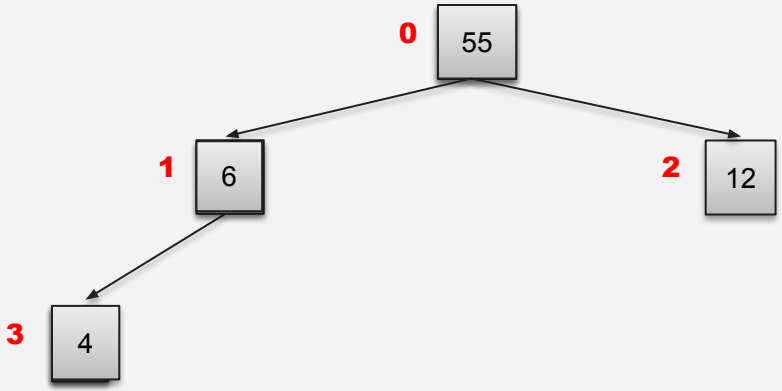
Convert into a Max-Heap (in-place)



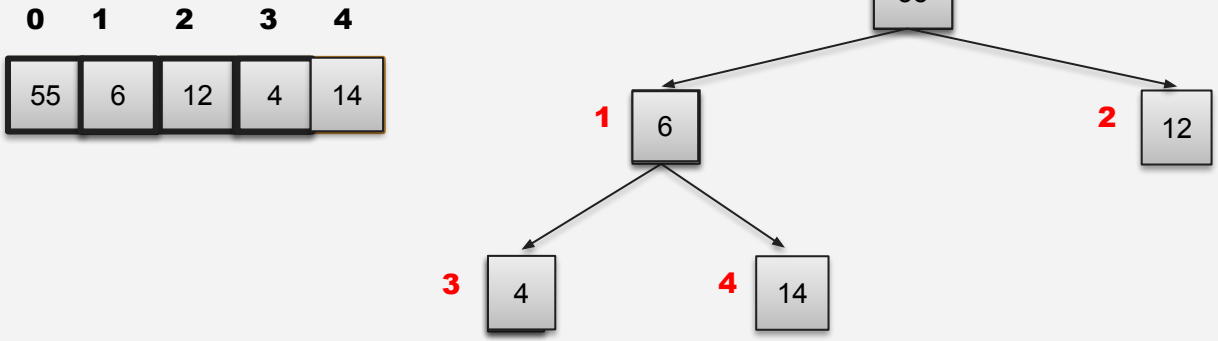


Convert into a Max-Heap (in-place)

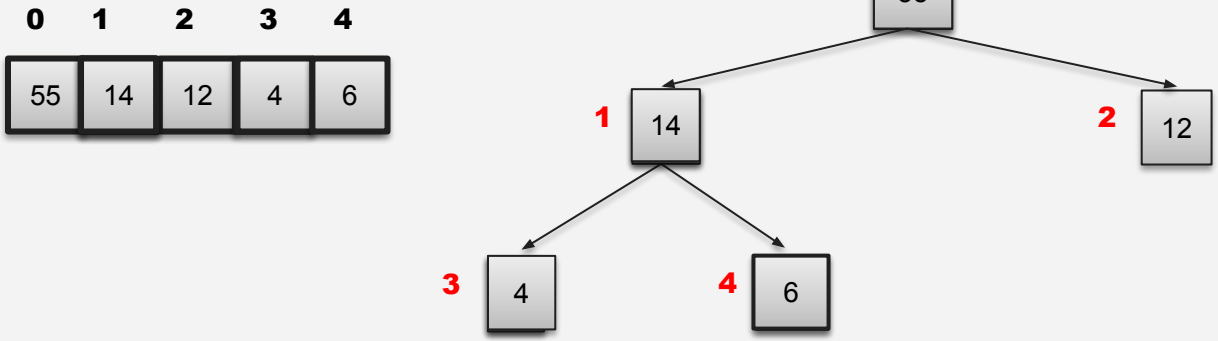




Convert into a Max-Heap (in-place)

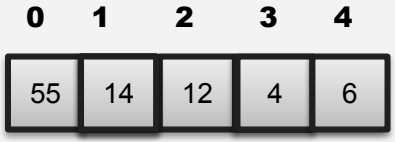


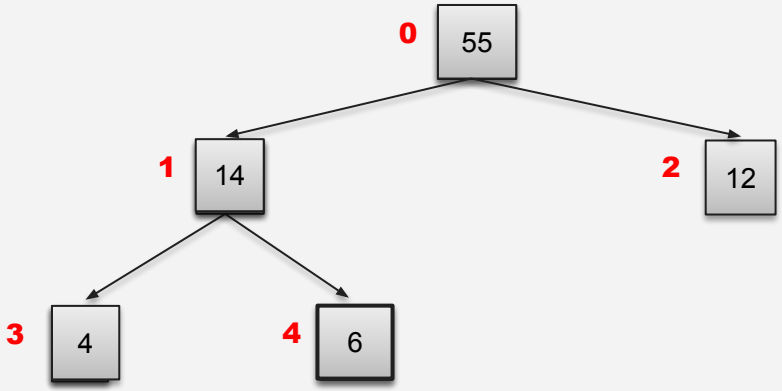
Convert into a Max-Heap (in-place)



Convert into a Max-Heap (in-place)

Runtime: O(n log n)

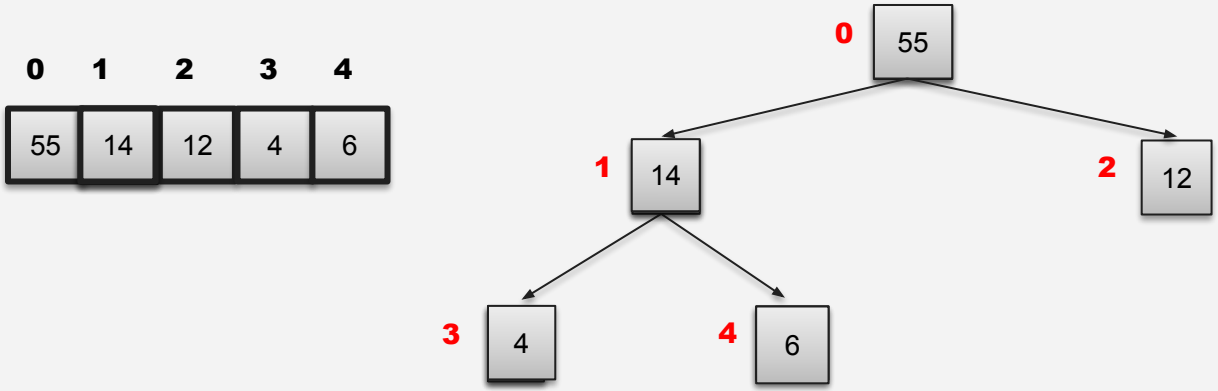




سوال؟

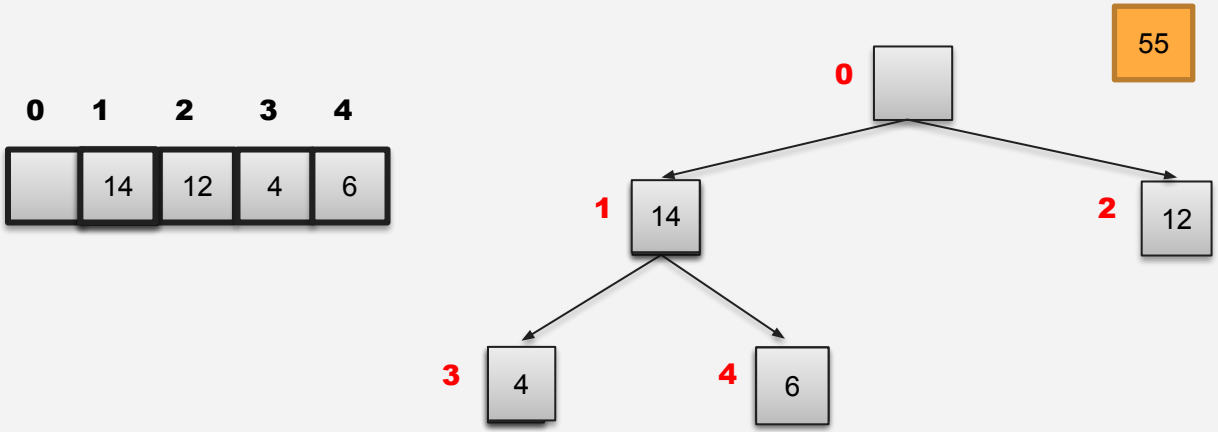
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



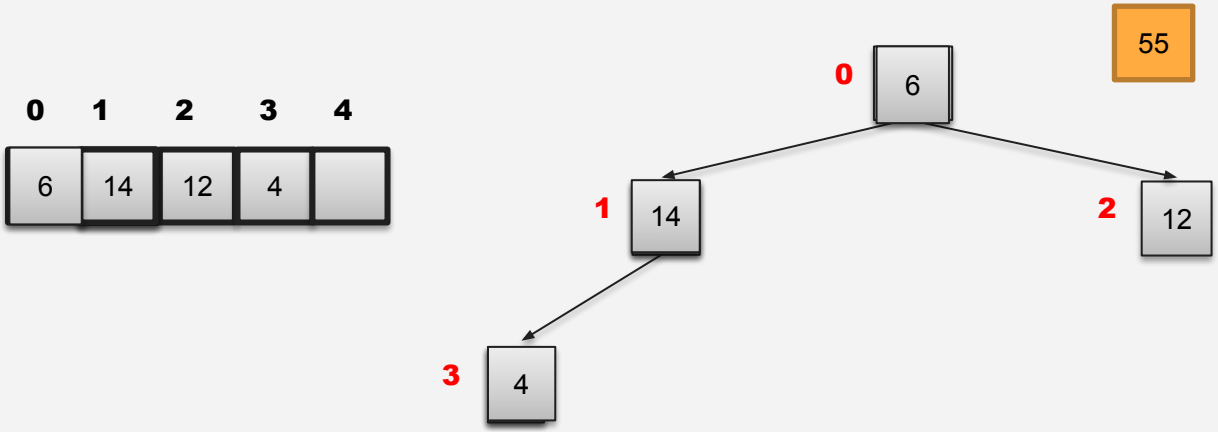
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



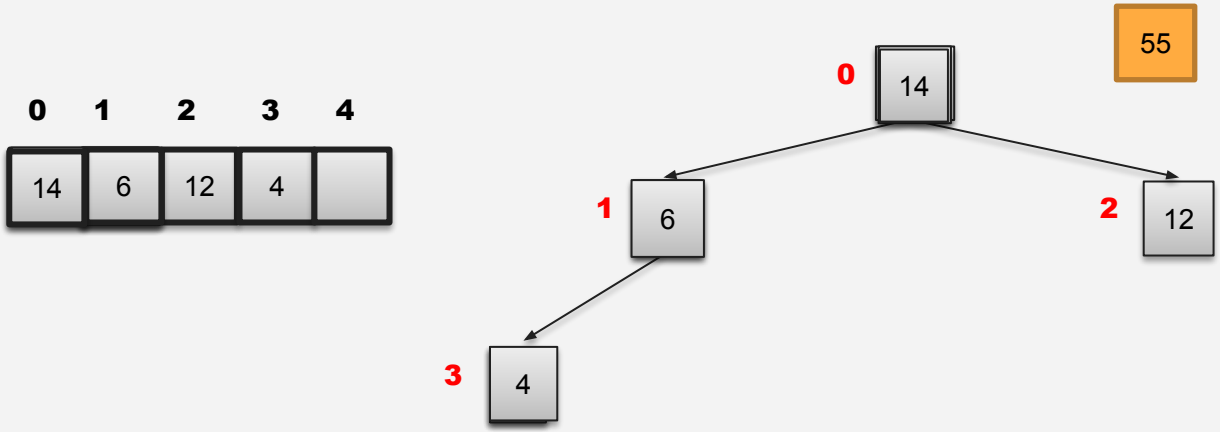
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



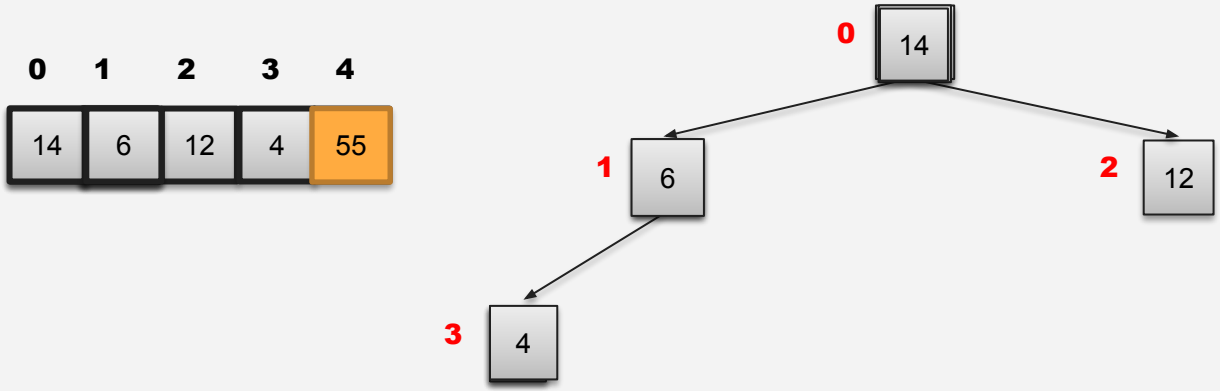
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



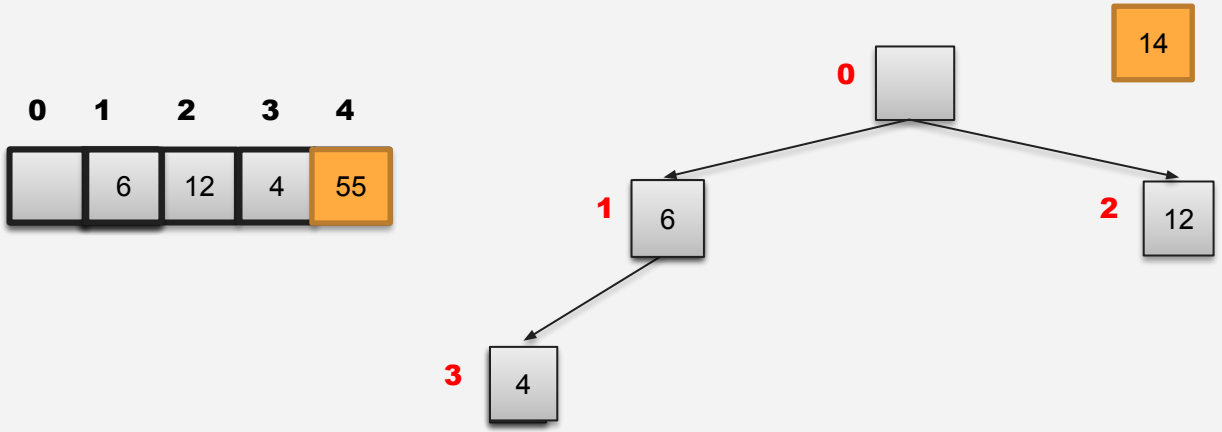
Take Max Element out of the Heap

- $b[k] = \text{poll}()$
  - i.e., take max element out of heap



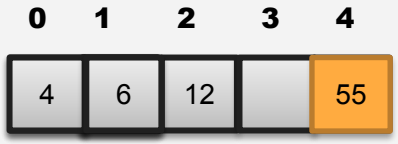
Take Max Element out of the Heap

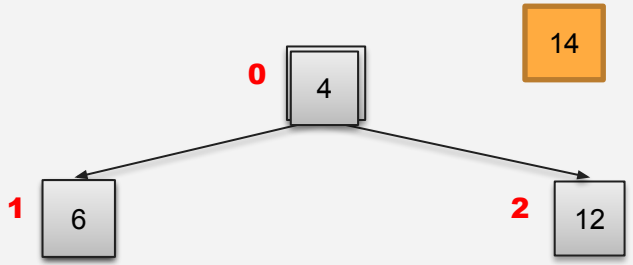
- b[k] = poll()
  - i.e., take max element out of heap



Take Max Element out of the Heap

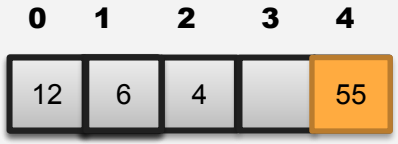
- b[k] = poll()
  - i.e., take max element out of heap

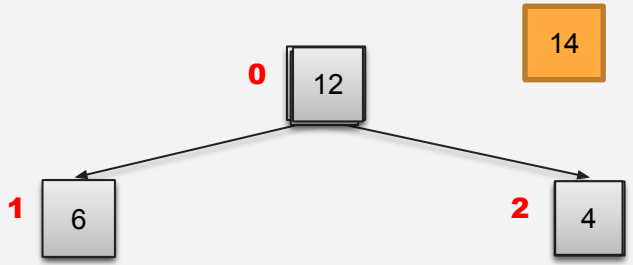




Take Max Element out of the Heap

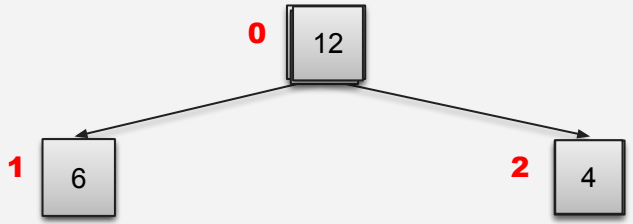
- b[k] = poll()
  - i.e., take max element out of heap





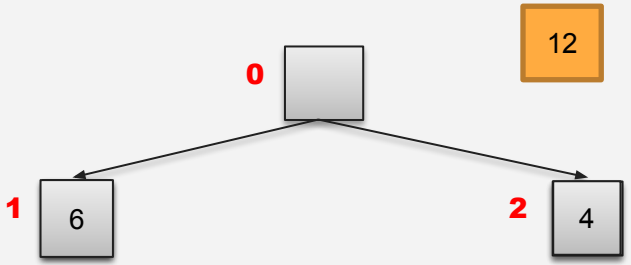
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



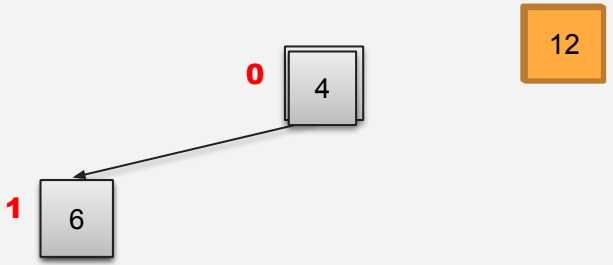
Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap



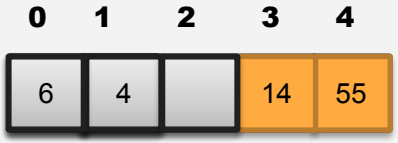
Take Max Element out of the Heap

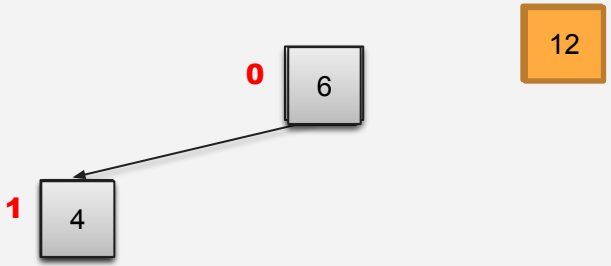
- $b[k] = \text{poll}()$
  - i.e., take max element out of heap



Take Max Element out of the Heap

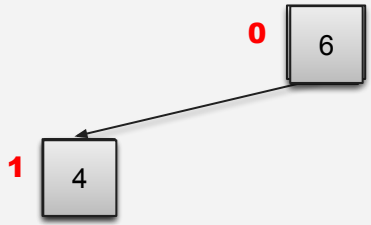
- b[k] = poll()
  - i.e., take max element out of heap





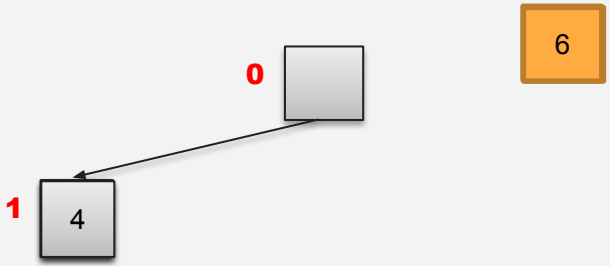
Take Max Element out of the Heap

- $b[k] = \text{poll}()$
  - i.e., take max element out of heap



Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap

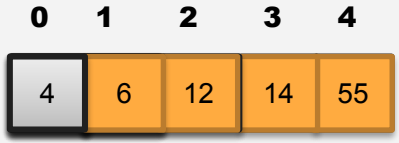


Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap

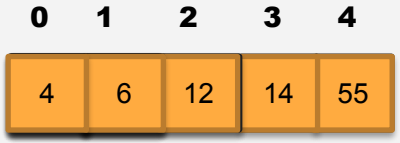
Take Max Element out of the Heap

- $b[k] = \text{poll}()$
  - i.e., take max element out of heap



Take Max Element out of the Heap

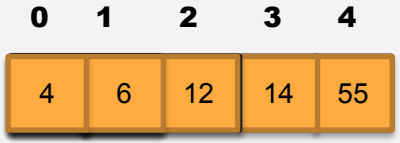
- $b[k] = \text{poll}()$
  - i.e., take max element out of heap



Take Max Element out of the Heap

- b[k] = poll()
  - i.e., take max element out of heap

Runtime: O(n log n)



سوال؟