CS1102: Lecture 10

Priority Queues and Heaps

Chapter 12: pages 621-640

Outline

- What is the ADT priority queue?
- What are the operations supported?
- heap (max-heap, min-heap)
  - heapInsert (O(log N))
  - heapDelete (O(log N))
  - heapRebuild (O(log N))
  - heapify (O(N))   - heap construction algorithm
- heapSort (O(N log N))
- Java API: PriorityQueue<E>

What is a Priority Queue?

A Special form of queue from which items are removed according to their designated priority and not the order in which they entered.

Priority Queue - Examples

- A “to-do” list with priorities
- Scheduling jobs in operating systems (OS)

- Go to Takashimaya (6)
- Work out in gym (5)
- Prepare CS1102 lecture slides (1)
- Go to department tea (4)
- ...

Priority queue operations

- Create an empty priority queue
- Insert an item with a given key
- Remove the item with maximum (or minimum) key value
- Determine whether a priority queue is empty.

Priority Queue – Unsorted list implementation

- **Insertion**: insert an element to end of a list. Complexity is O(1).
- **Removing maximum key value**: traverse the list to find the element of maximum key value and remove it. Complexity is O(n).

Priority Queue – Sorted list implementation

- **Insertion**: O(n).
- **Removing maximum key value**: O(1).

Priority Queue – Binary Search Tree implementation

- **Insertion**: $O(h)$, where h is the height of the BST.
- **Removing maximum key value**: $O(h)$.
- The height of a BST is not always $O(\log n)$.

Priority Queue – AVL Tree implementation

- AVL Tree is not covered in this course. Just for your information only.
- An AVL Tree is a self-balancing height balanced binary search tree.
- The height of an AVL tree is O(log n).
- **Insertion**: O(log n).
- **Removing maximum key value**: O(log n).

Priority Queue – Heap implementation (Summary)

- **Insertion**: O(log n).
- **Removing maximum key value**: O(log n).
- **Heap construction** O(n).

  How about for building an AVL tree?

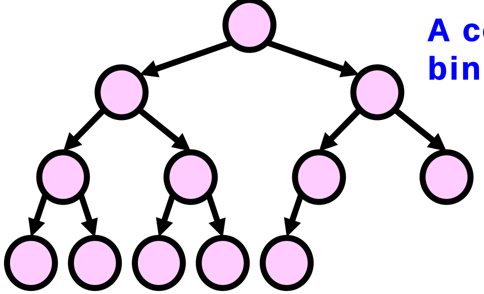
  Ans: O(n log n)
- **Peek** the maximum key value: O(1).

  How about for an AVL tree?

  Ans: O(log n)

Heap

Heap is the most appropriate data structure for realizing the ADT priority queue.

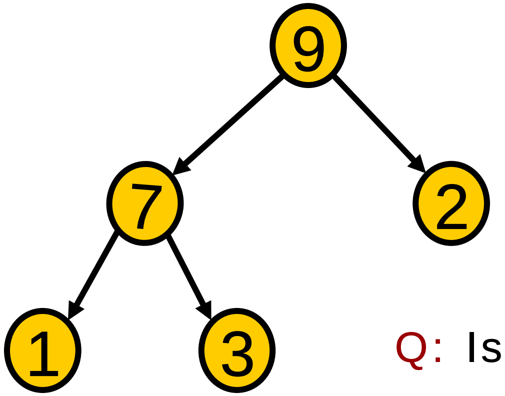


Definition: Heap

- A (binary) **heap** is a **complete binary** tree
  - either is empty,
  - or satisfies the **heap property**:

  for every node *v*, the search key in *v* is greater or equal to those in the children of *v*. It is called a max heap.

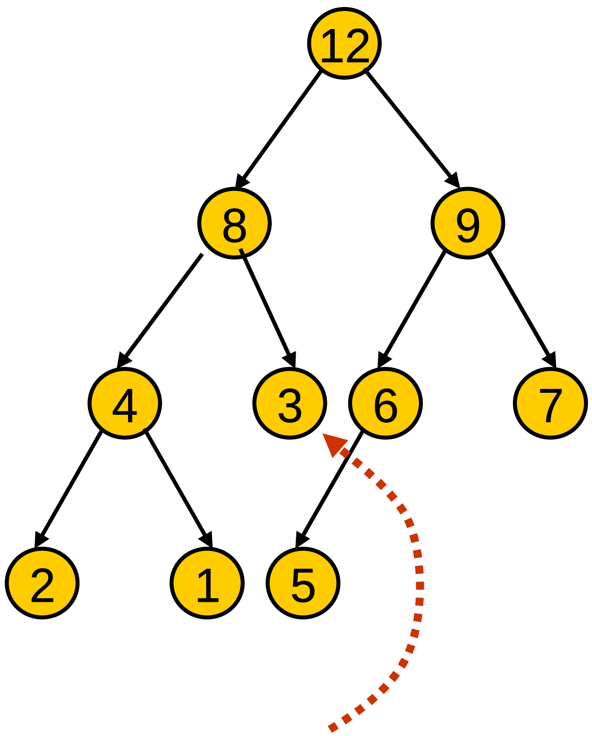
Note: The keys on the right subtree could be smaller than the keys on the left subtree



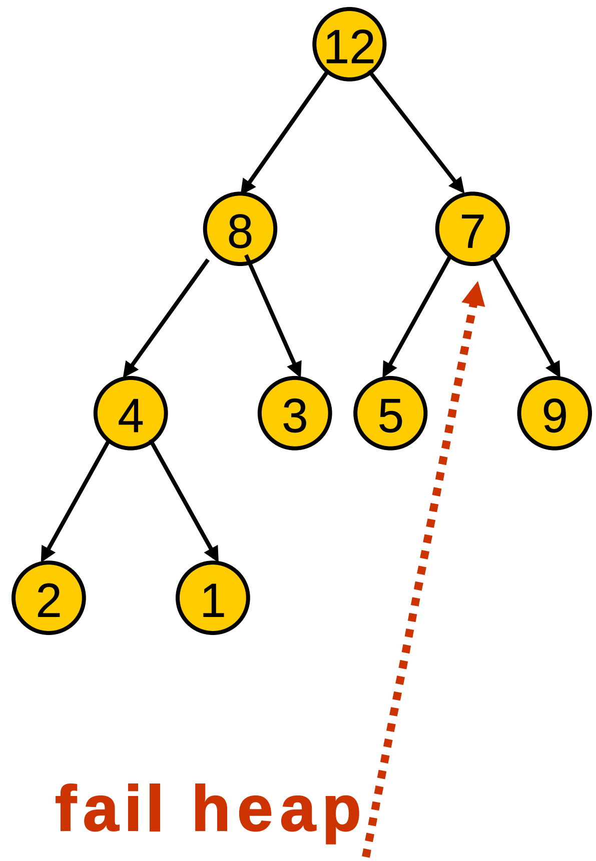
Q: What are the nice properties of complete binary trees?

Q: Is it a heap?

Negative heap examples



not complete

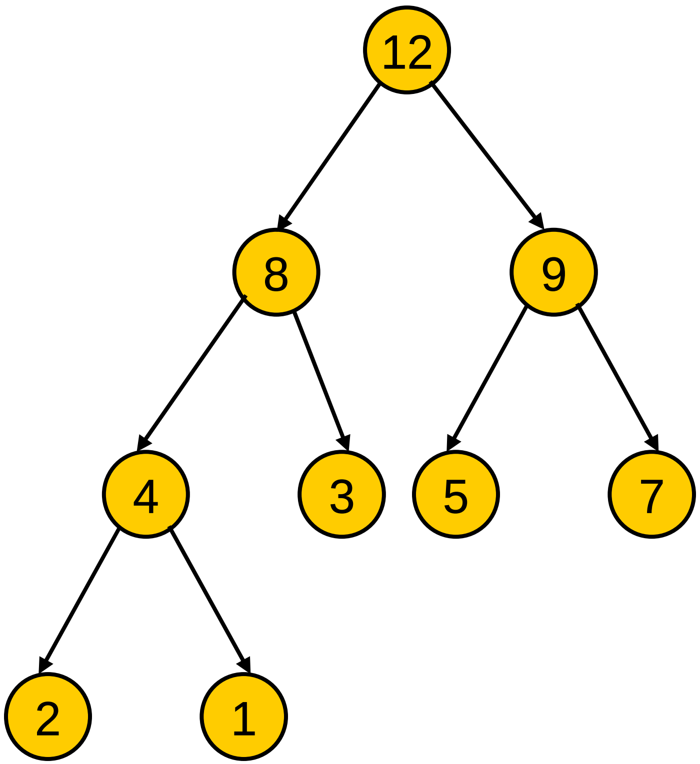


fail heap property

Compare heap with binary search tree (BST)

- Both are binary trees.
- Difference
  - Heap maintains heap property.
    - It is not a binary search tree
  - BST maintains BST property.
    - It is not a heap

Representation of heaps using arrays



0	12
1	8
2	9
3	4
4	3
5	5
6	7
7	2
8	1

Recall

left(i) = 2*i+1

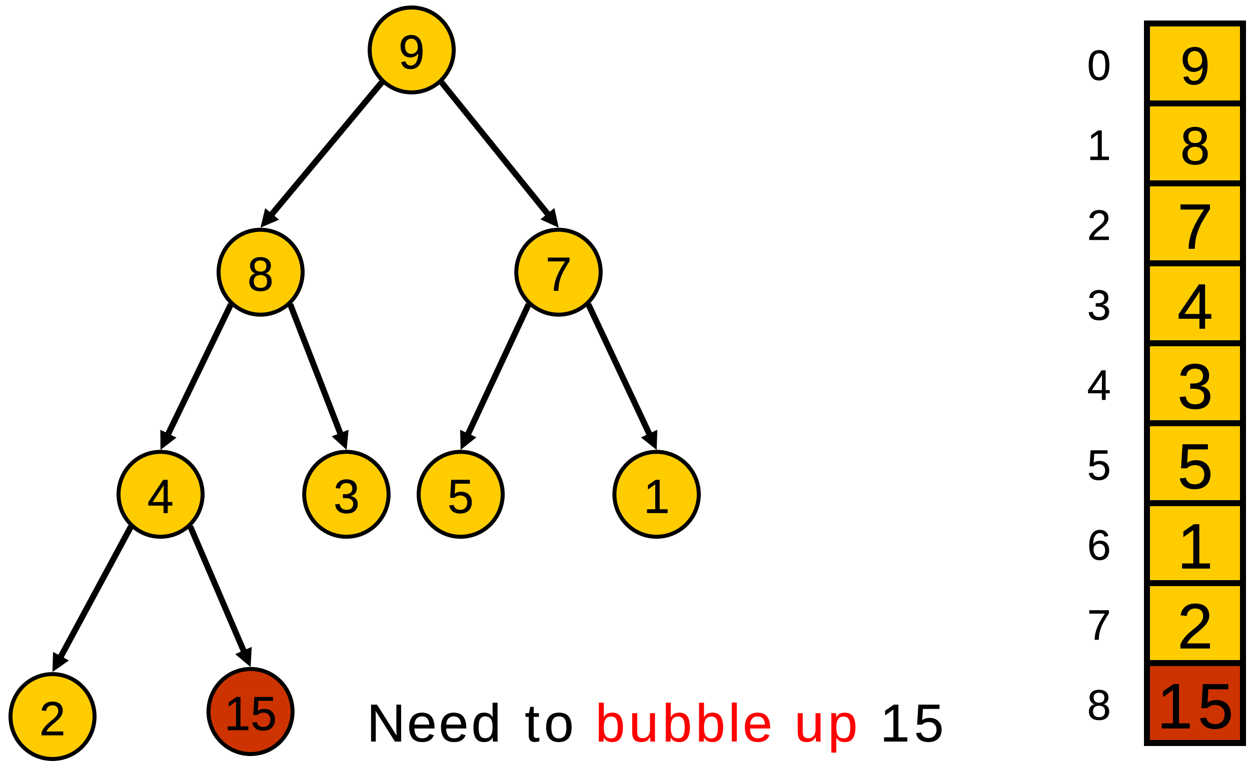
right(i) = 2*i+2

parent(i) = floor((i-1)/2)

Note: Recall that a heap is a complete binary satisfying heap property

Insert an item

Insert 15

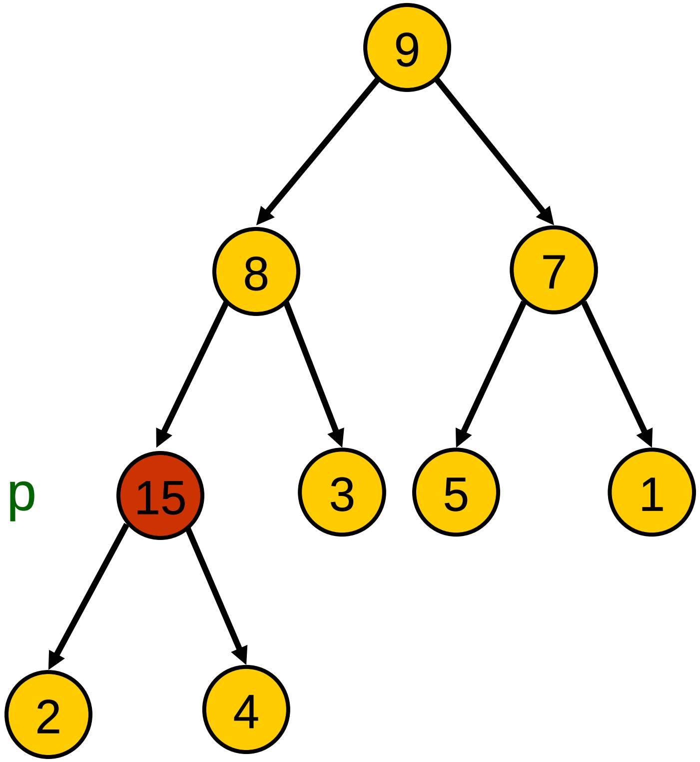


Need to bubble up 15

To insert an item, we first **append** it to the array. This may **violate** the heap property.

Re-establish heap property

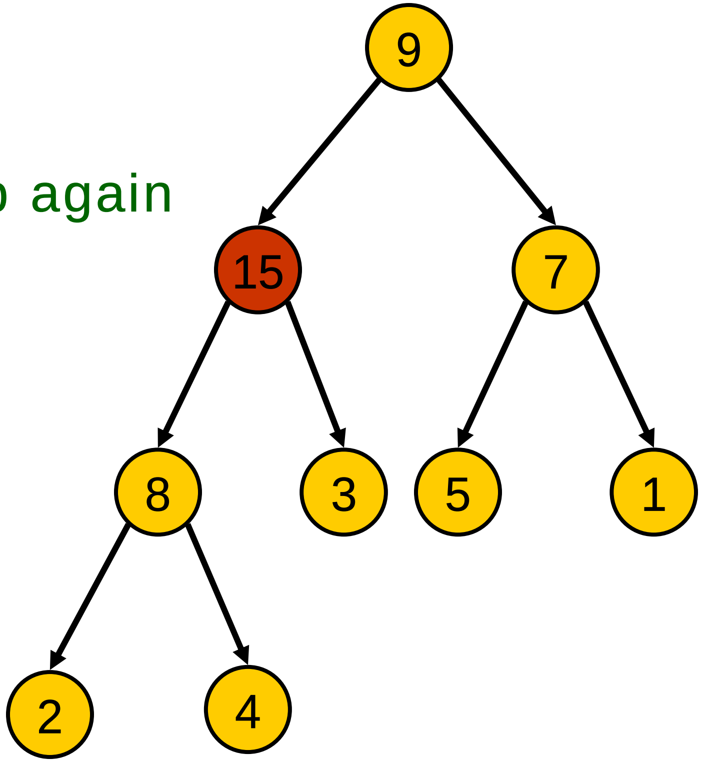
15 is bubbled up



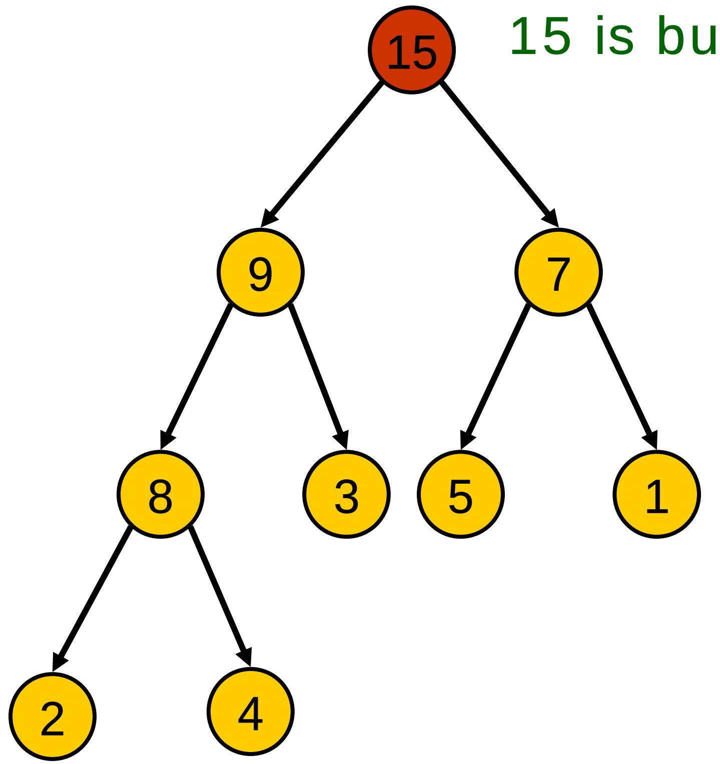
We re-establish the heap property by moving the newly inserted item up the tree. This operation is called **bubble up**.

Re-establish heap property

15 is bubbled up again



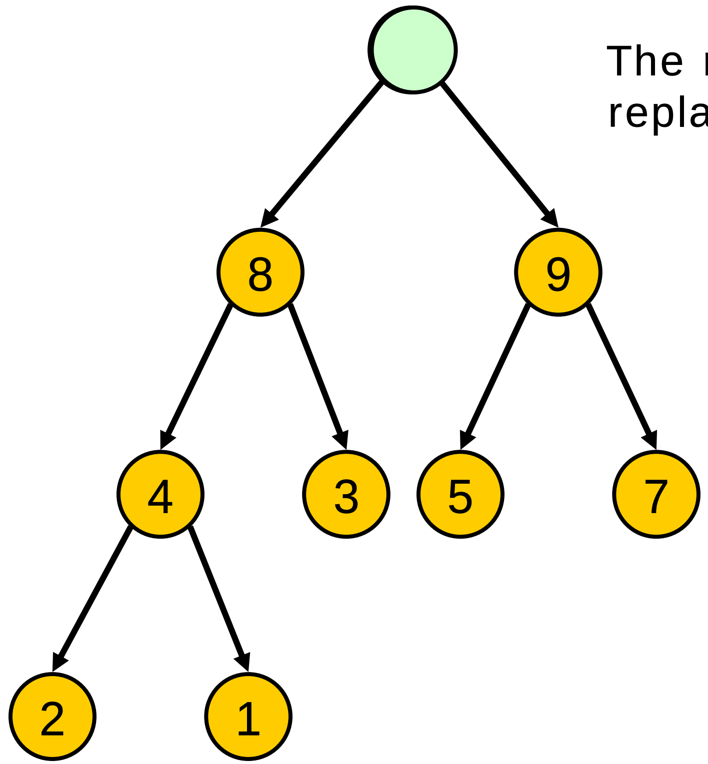
Re-establish heap property



15 is bubbled up the third time

15 is bubbled up to its final position. The heap is re-established.

Remove the max item

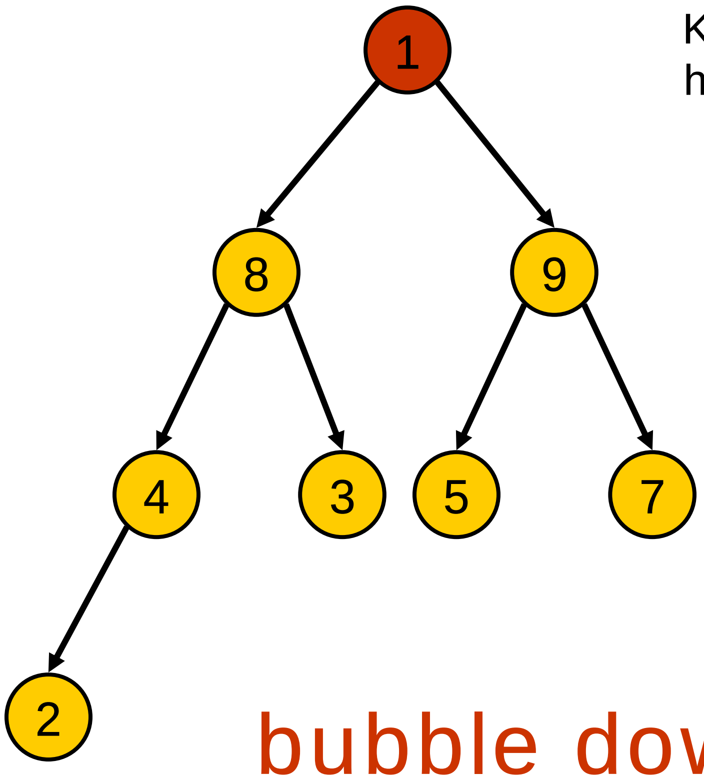


The max item 15 (the root) is replaced by the last item 1.

To remove the maximum item (the root item, **12** in this example), we take the **last item** (**1** in this example) in the heap and replace the **root** with it. This may violate the heap property.

Q: Why the **last item**?

Re-establish heap property

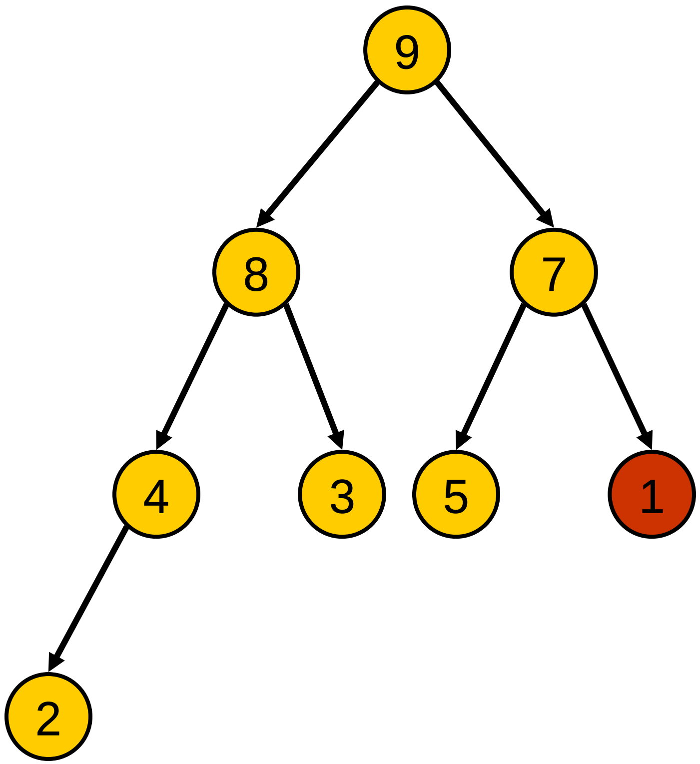


Key value 1 violates the heap propery

bubble down

If new root node violates the heap property, then we **swap** the new root with its child with **bigger key** value (**9** in this example), and repeat the process until no violation on the heap property. This is called **bubble down**.

Re-establish heap property



bubble down again

Key value 1 is now in its final position. The heap property is re-established.

Code - using array to implement heap

```
public class Heap {
   private int MAX_HEAP = 100;
   private int [] items;
   private int size;          // size of the heap

   public Heap() {
     items = new int[MAX_HEAP];
     size = 0;
   }

   public boolean heapIsEmpty() {
     return size == 0;
   }
```

heapInsert (bubble up) - using iteration

```
public void heapInsert (int newItem) throws HeapException {
    if (size < MAX_HEAP) {
      items[size] = newItem;           // append the new item to the array
      int place = size;                // place is the index of the new item
      int parent = (place - 1)/2;      // find the parent node index
      while ( (parent >= 0) && (items[place] > items[parent]) {
              // heap property violated, need to bubble up

          int temp = items[place];
          items[place] = items[parent];
          items[parent] = temp;

          place = parent;
          parent = (place - 1)/2;
      }
      ++size;
    }
    else
      throw new HeapException("HeapException: Heap full");
}
```

heapDelete – remove the **maximum** key value

```
public int heapDelete() {
      int rootItem = 0;
      if (!heapIsEmpty()) {
         rootItem = items[0];     // rootItem is set to the max key value
         items[0] = items[--size]; //replace the root by the last item
         heapRebuild (0);      // to rebuild the help – bubble down
      }
      return rootItem;      // return the maximum key value
   }
```

heapRebuild (bubble down) – recursive

- the root node may violate the heap property, bubble down to re-establish the heap recursively

```
protected void heapRebuild (int root) {
    int child = 2 * root + 1;          // left child
    if (child < size) {                // there is a left child
      int rightChild = child + 1;  // right child

      if ( (rightChild < size) &&    // there is a right child
          (items[rightChild] > items[child]) )
        child = rightChild;           // choose child with bigger key value

      if ( items[root] < items[child] ) {
         // bubble down - swap root with bigger child
        int temp = items[root];
        items[root] = items[child];
        items[child] = temp;

        heapRebuild(child);    // recursive call
      }
    }
}
```

Running time of heapDelete

- How many calls to **heapRebuild**?
- Go down 1 level after each call to heapRebuild**.**
- number of heapRebuild calls < height of the complete binary tree
- The height of a complete binary tree is log n. Q: Why?
- Worst case running time is $O(h) = O(\log n)$
- How about the complexity of heapInsert?

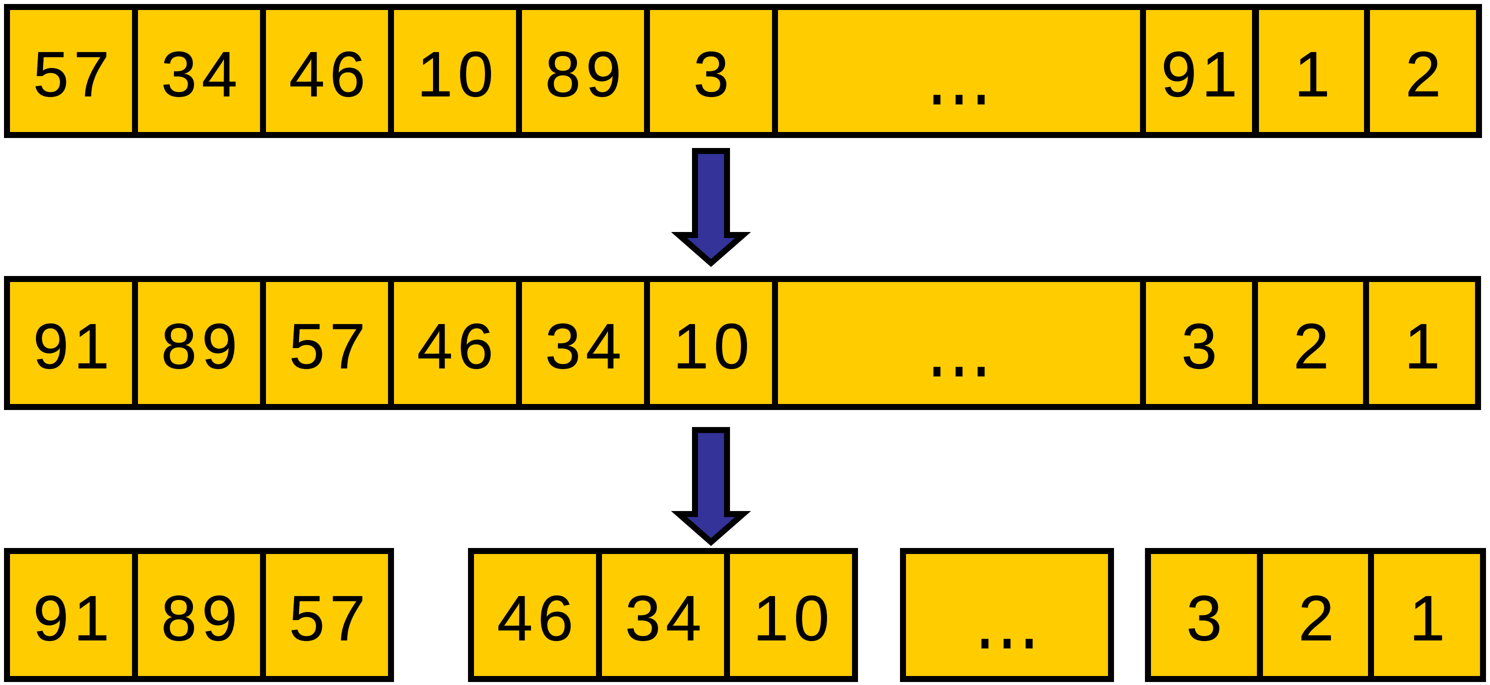
Heap Construction

To construct a heap from a given set of key values

Display ranked web pages

- Display 10 pages at a time in order of decreasing page rank scores

Sort the rank scores



- Sort the web pages according to their rank scores
- Traverse the sorted list

Running times

- Sorting $O(n \log n)$
  n: total number of pages
- Traversing $O(k)$
  k: number of pages requested
- Total running time:

$$O(n \log n) + O(k)$$
$$<= O(n \log n) + O(n), \text{ since } k<n$$
$$= O(n \log n)$$

Q: Can we do better? Maybe

Idea

- Build a heap of scores
- Remove the top 10 pages at a time

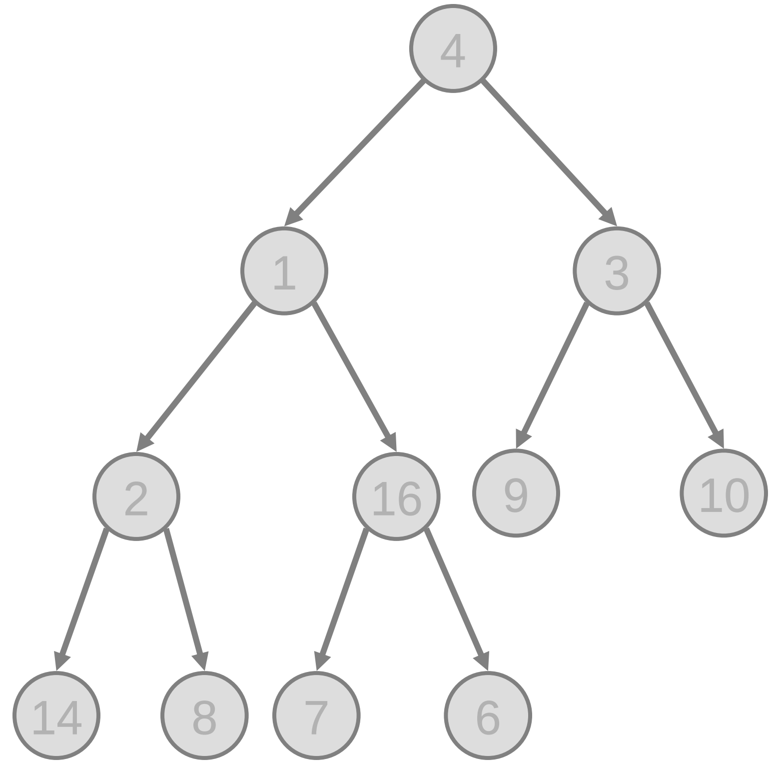
Heap construction

- Recall **heap property**: for every node *v*, the search key in *v* is greater or equal to those in the children of *v*
- Build the heap recursively from **bottom up**

Note: Alternatively, we can insert nodes one at a time into a heap. This corresponds to building a heap from the **top downwards**.

Heap construction example

0	4
1	1
2	3
3	2
4	16
5	9
6	10
7	14
8	8
9	7
10	6



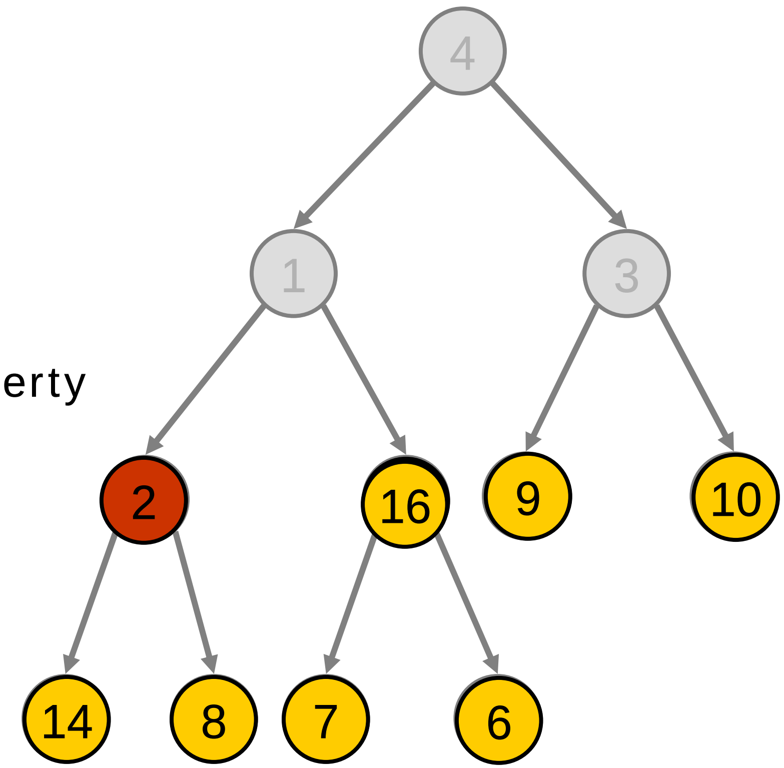
The data in the array represents the complete binary tree on the right.

Heap construction example

0	4
1	1
2	3
3	2
4	16
5	9
6	10
7	14
8	8
9	7
10	6

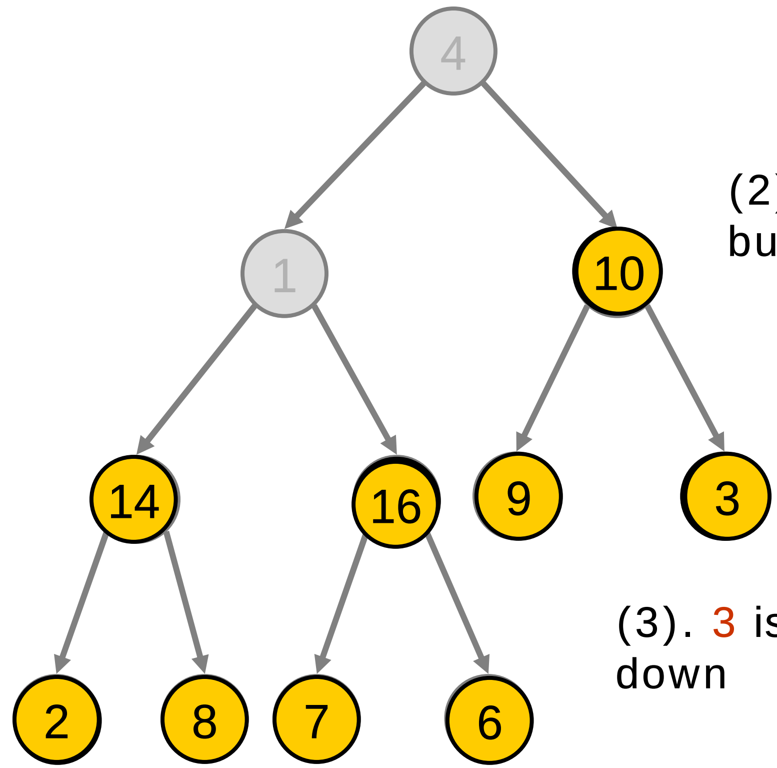
(1). No heap property violation at 16.

(2). Heap property violation at 2.



We start by building a heap by calling the heapRebuild method from the **last internal node** back to the **root**. (i.e. **16, 2, 3, 1, 4** in this example)

Heap construction example



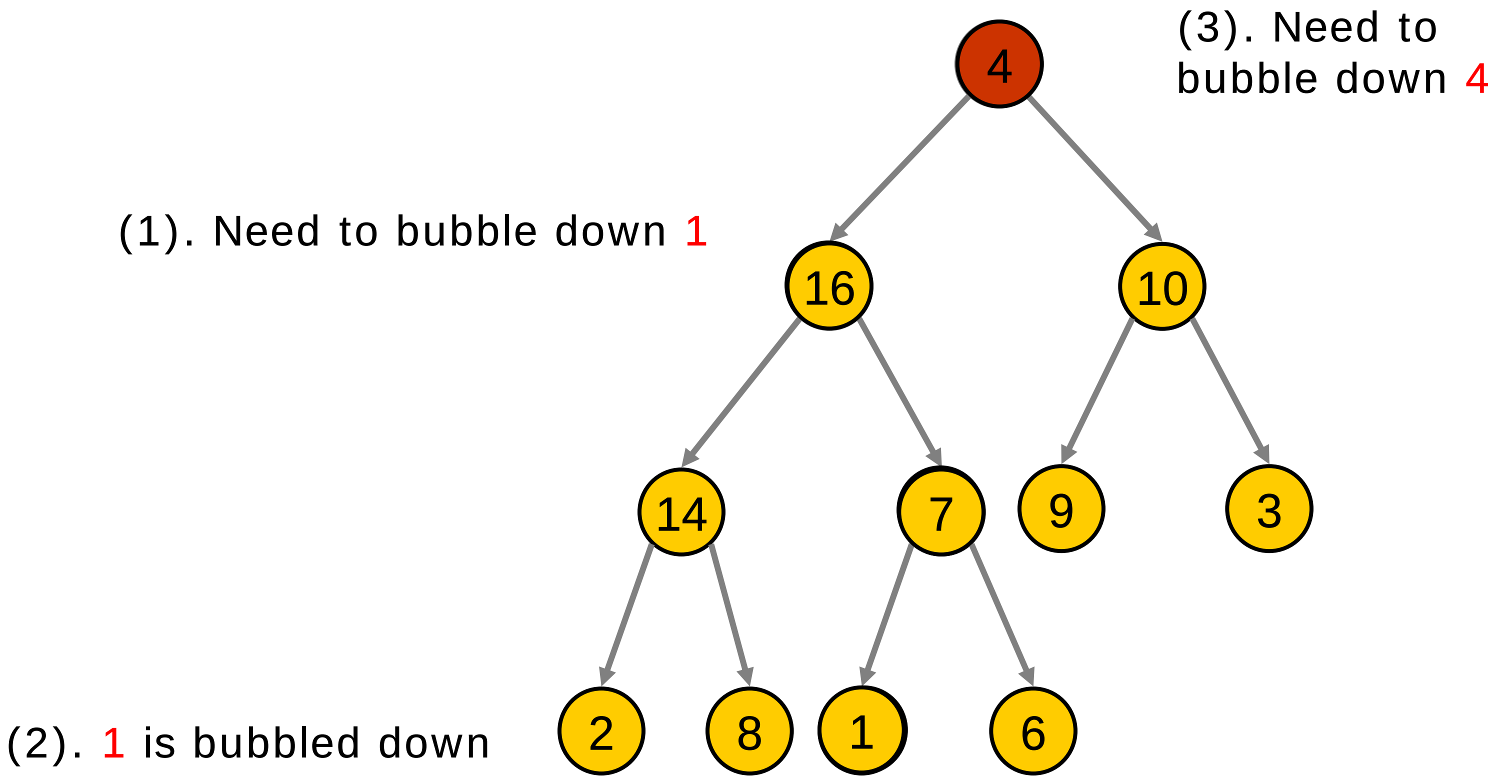
(1). 2 is bubbled down

(2). Need to bubble down 3.

(3). 3 is bubbled down

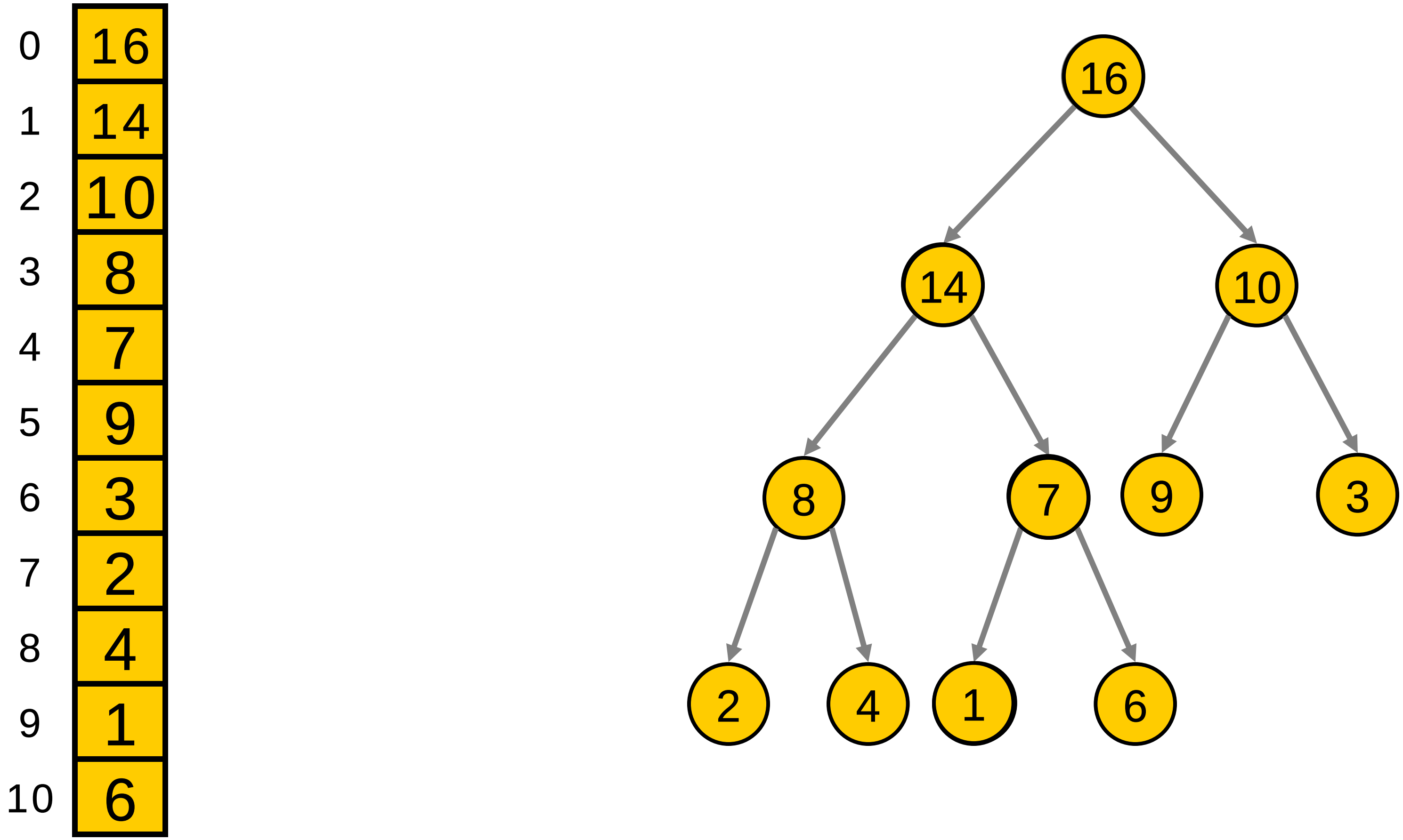
Call heapRebuild at 16, then 2, 3, 1, 4 in this example

Heap construction example



Call heapRebuild at 16, then 2, 3, 1, 4 in this example

Heap construction example



4 is bubbled down. The heap is constructed.

Heapify - Heap construction algorithm

```
protected void heapify() {
   for (int i = size/2-1; i >= 0; i--)
               heapRebuild(i);
}
```

Note: i starts from the last internal node at **size/2-1** back to the root node 0.

Heap construction algorithm's Running time

Rough count: $n/2 * O(\log n) = O(n \log n)$ // call heapRebuild **n/2** times

More careful count of No. of calls to **heapRebuild**,
level by level from bottom up:

level	No. of calls	Each call requires
-2	$n/2^2$	O(2)
-3	$n/2^3$	O(3)
-4	$n/2^4$	O(4)
...		

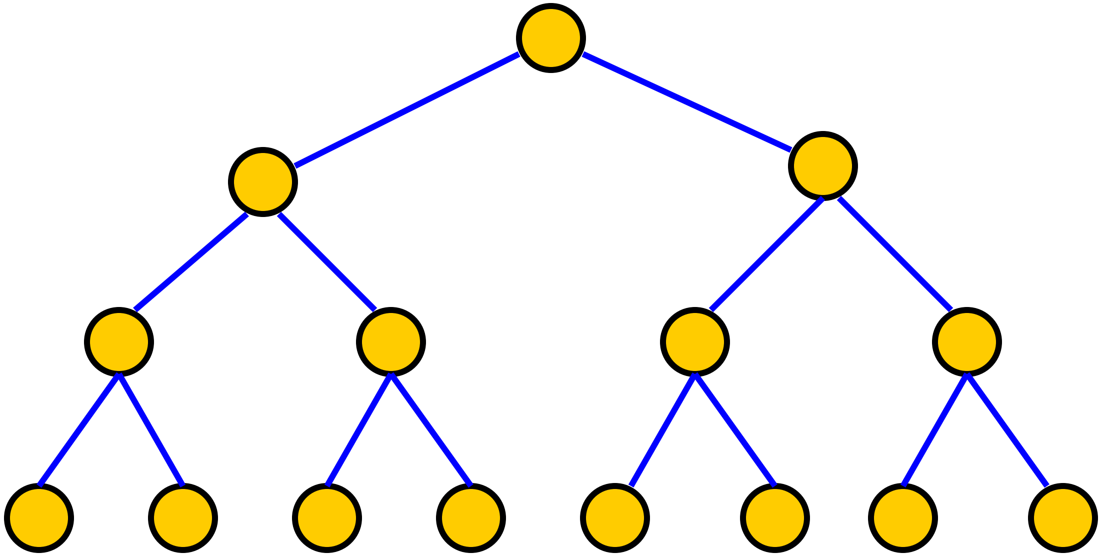
Total complexity of the heap construction algorithm:

$2 \times n/2^2 + 3 \times n/2^3 + 4 \times n/2^4 + \ldots$

$< n(\mathbf{2/2^2 + 3/2^3 + 4/2^4 + \ldots})$

$< n(3/2) = O(n)$

Running time: another derivation



Count number of edges visited in bubbling down.

number of nodes: $n = 2^h - 1$ // $h = \log n$

Total number of edges visited:

= total no. of edges – no. of edges not visited

= $(n-1) - (h-1)$

= $n-h$ = $O(n)$ Why? Ans. Because $h = \log n$

Web page ranking again

- Build a heap *O(n)*
- Retrieve top k pages *O(k log n)*

  Why? Ans. heapDelete is O(log n).
- Total running time: O(n) + O(k log n)
  - If k=n, then O(n log n)   Q: What meaning?
  - If k=20 (or some other **constant**) , then

    O(n) + O(20 log n)

    = O(n)

Retrieve top K pages is O(n) using heap!

Heapsort

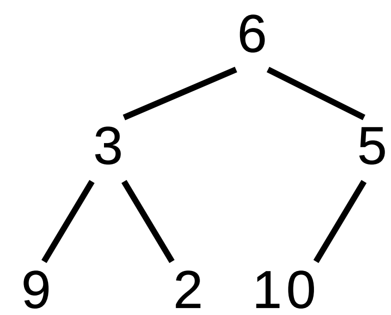
Uses a heap to sort an array a[0..n] of items.

- First transform the array into a heap, O(n)
- Then execute n steps to turn the heap into a sorted array:
  - In step k, k =1..n:
    - The array has been partitioned into two regions: the heap region a[0..n-k+1] and the sorted region a[n-k+2..n]
    - swap a[0] with a[n-k+1]
    - heapRebuild a[0..n-k].
- Heapsort worst and average cases are O(n log n).

Heapsort - Example

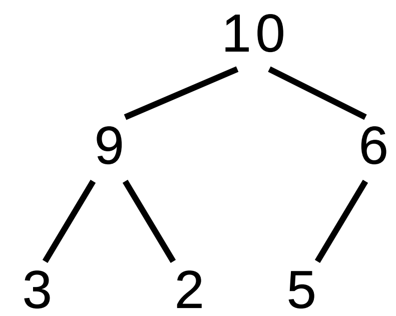
Original array

6	3	5	9	2	10



After heap construction

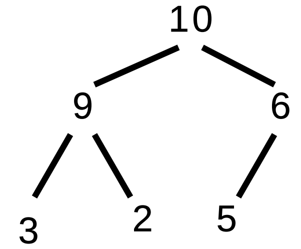
10	9	6	3	2	5



Q: What is complexity of heap construction?

Ans. O(n)

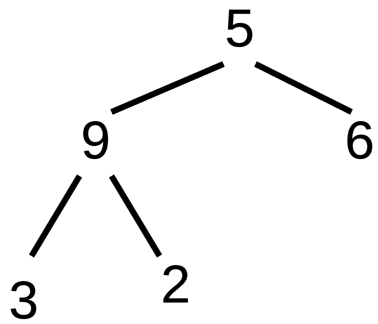
Heapsort - Example



Step 1

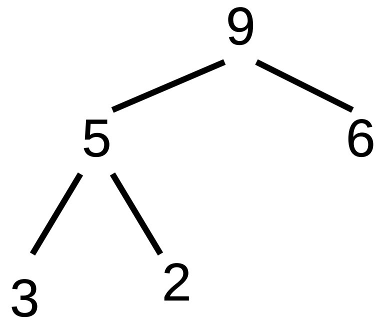
Before swap

10	9	6	3	2	5



After swap

heap					sorted
5	9	6	3	2	10

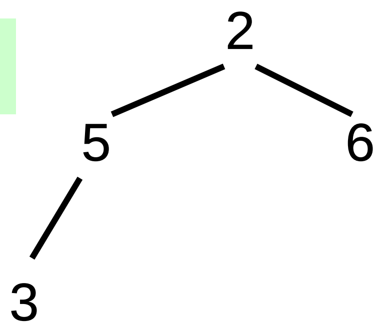


After heapRebuild

heap					sorted
9	5	6	3	2	10

Heapsort - Example

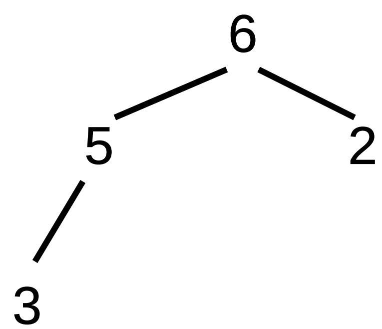
Step 2



After **swap**

heap | sorted

2	5	6	3	9	10



After **heapRebuild**

heap | sorted

6	5	2	3	9	10

Finally **Sorted**

heap | sorted

2	3	5	6	9	10

Is it in place? **Yes**

Is it stable?

Complexity? **O(n log n)**

A Heap of Queues

- Used when there are only finite number of distinct priority values
  - For example, scheduling of tasks by OS
- Each entry in the heap is a queue, one queue for each priority
- To add an item to the heap, just enqueue it to the queue of the item's priority
- To remove an item from the heap, dequeue an item from the queue with the highest priority

Summary of Heap

- Priority queue is a special form of queue where the item with maximum key value (highest priority) is removed first.
- It is best implemented by a Heap.
- Heap Insert is $O(\log n)$
- Heap Delete is $O(\log n)$
- Heap construction is $O(n)$
- Peek the maximum key value: $O(1)$.
- Retrieve Top K key value: $O(n)$
- Heap Sort is $O(n \log n)$

Q: What are the complexities of the operations if priority queue is implemented by unsorted list, sorted list, binary search tree, or balanced binary tree (e.g. AVL Tree)?

Java API: PriorityQueue<E>

PriorityQueue ()

Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering (using Comparable).

PriorityQueue (int initialCapacity, Comparator<? Super E> comparator)

Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

PriorityQueue (PriorityQueue<? extends E> c)

Creates a PriorityQueue containing the elements in the specified collection.

Java API: PriorityQueue<E> (cont.)

Java PriorityQueue<E> maintains a min-heap.

boolean offer (E o)
Inserts the specified element into this priority queue.

E peek()
Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

E poll()
Retrieves and removes the head of this queue, or null if this queue is empty.

int size()

PriorityQueue<E> Example

```
import java.util.*;

public class S {
  public static void main(String[] args) {
    PriorityQueue<Integer> pq = new
                  PriorityQueue<Integer>();
    pq.offer(20);
    pq.offer(10);
    while (pq.size()!= 0){
      System.out.println( pq.poll()) ;     // output: 10 20
    }
  }
}
```

Note: Java's PriorityQueue is a min-heap.