Lecture 18-19

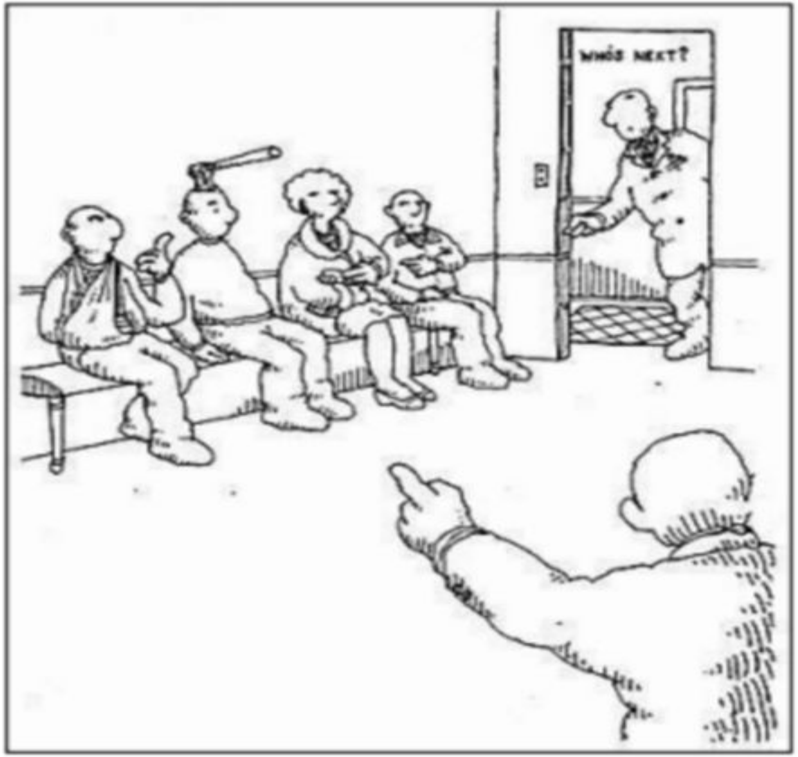
Heaps, Heap Sort
Some slides were borrowed from Prof. Alvarado.

Reminder

- Mic
- Midterm2

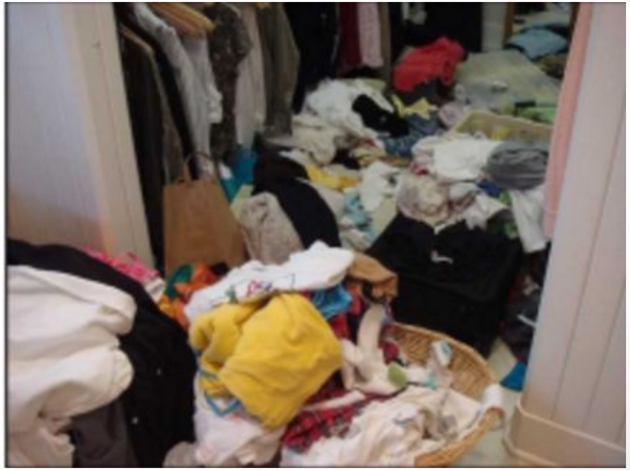
Priority Queue ADT

- Emergency Department waiting room operates as a priority queue
- Patients sorted according to seriousness, NOT how long they have waited



Unsorted linked list

- Insert new element in front
  - Easy, O(1), fast
- Remove by searching list for highest-priority item
  - Slow, may have to scan the whole list O(N)



Sorted linked list

- Always insert new elements where they go in priority-sorted order
  - Slow, O(n)
- Remove from front
  - Fast, O(1)

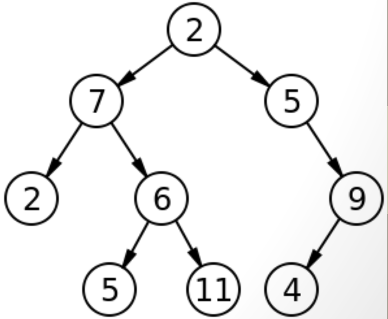
Can we combine them?

- Fast add AND fast remove/peek

- Yes! New data structure: heap.

Complete Binary tree

- Binary tree is a tree data structure in which each node has at most **two** children: left and right.
- In a **complete** binary tree *every* level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- **Height** of tree –The height of a tree is the number of edges on the longest downward path between the root and a leaf.

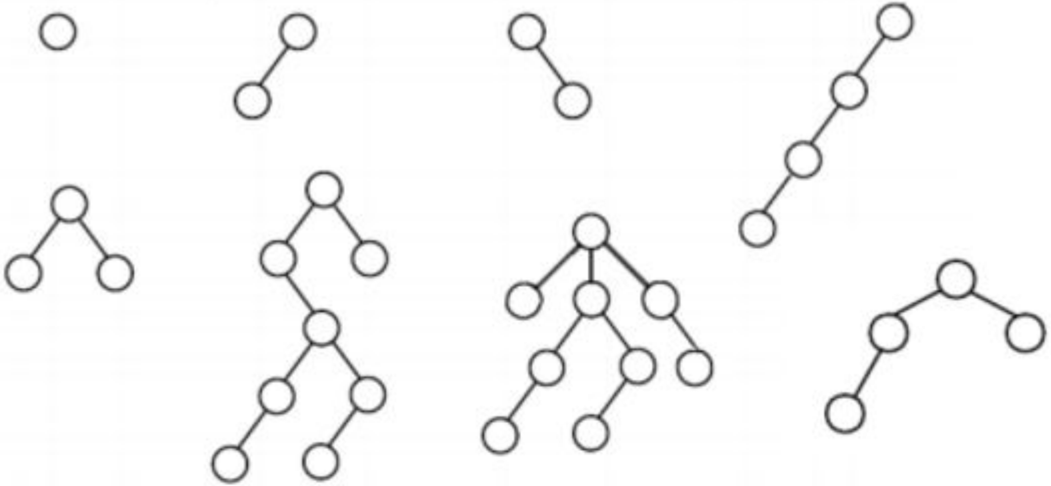


Heaps

- Heaps are ONE kind of binary tree
- They have a few *special* restrictions, in addition to the usual binary tree ADT:

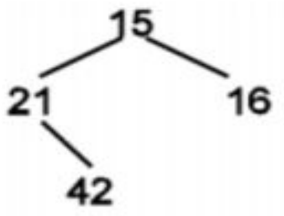
- Binary tree must be **complete**
- Ordering of data must obey **heap property**
  - **Min-heap** version: a parent's data is always ≤ its children's data
  - **Max-heap** version: a parent's data is always ≥ its children's data

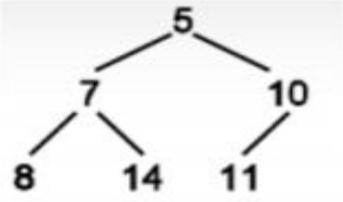
How many of these could be valid heaps?

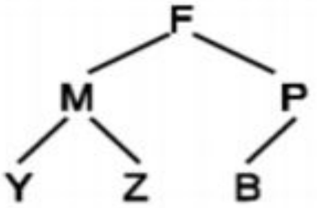


A. 0-1
B. 2
C. 3
D. 4
E. 5-8

How many of these are valid min-heaps?



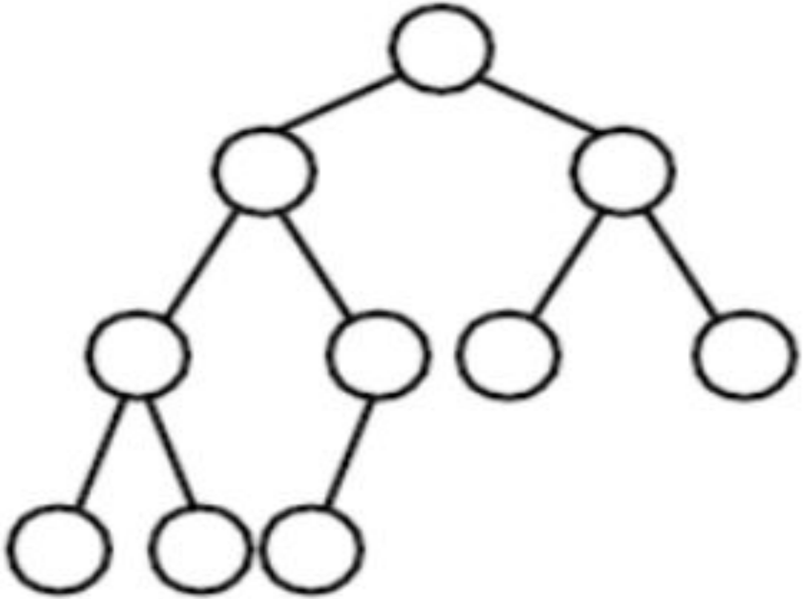




A. 0
B. 1
C. 2
D. 3

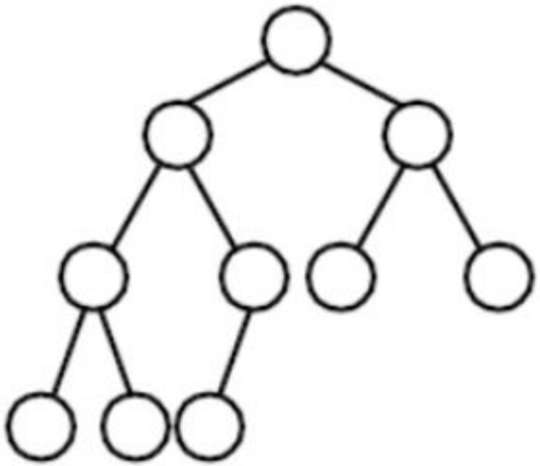
In how many places could the largest number in this **max**-heap be located ?

A. 0-2
B. 3-4
C. 5-6
D. 7-8



In how many places could the largest number in this <u>max</u>-heap be located ?

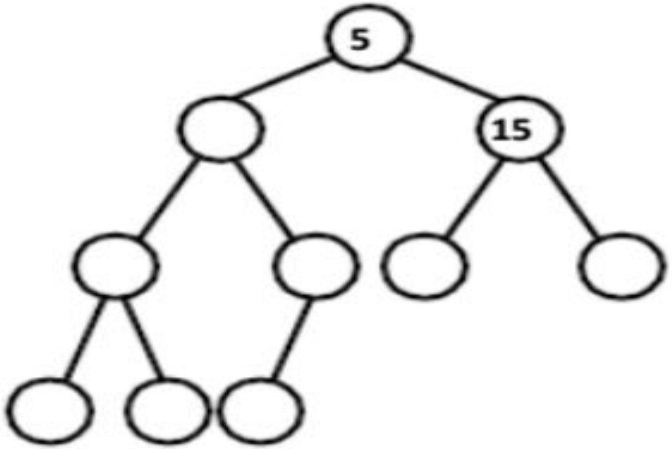
A. 0-2
B. 3-4
C. 5-6
D. 7-8



Max heaps are perfect for priority queues, because we always know where the highest priority item is

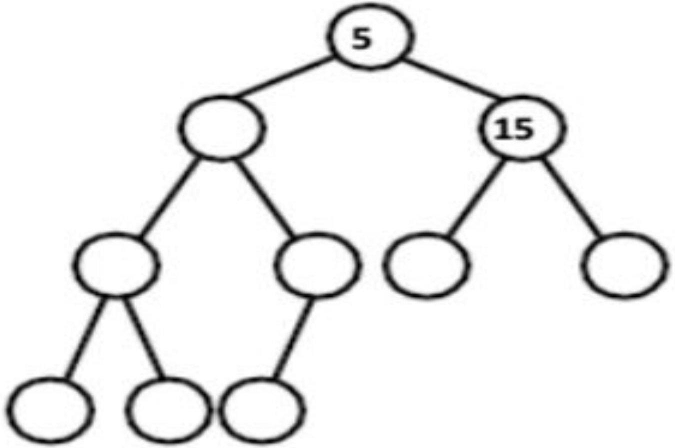
In how many places could the number 35 be located in this min-heap?

A. 0-2
B. 3-4
C. 5-6
D. 7-8



In how many places could the number 35 be located in this <u>min</u>-heap?

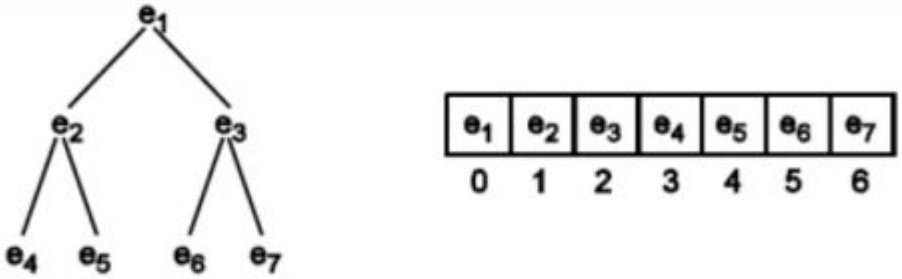
A. 0-2
B. 3-4
C. 5-6
D. 7-8

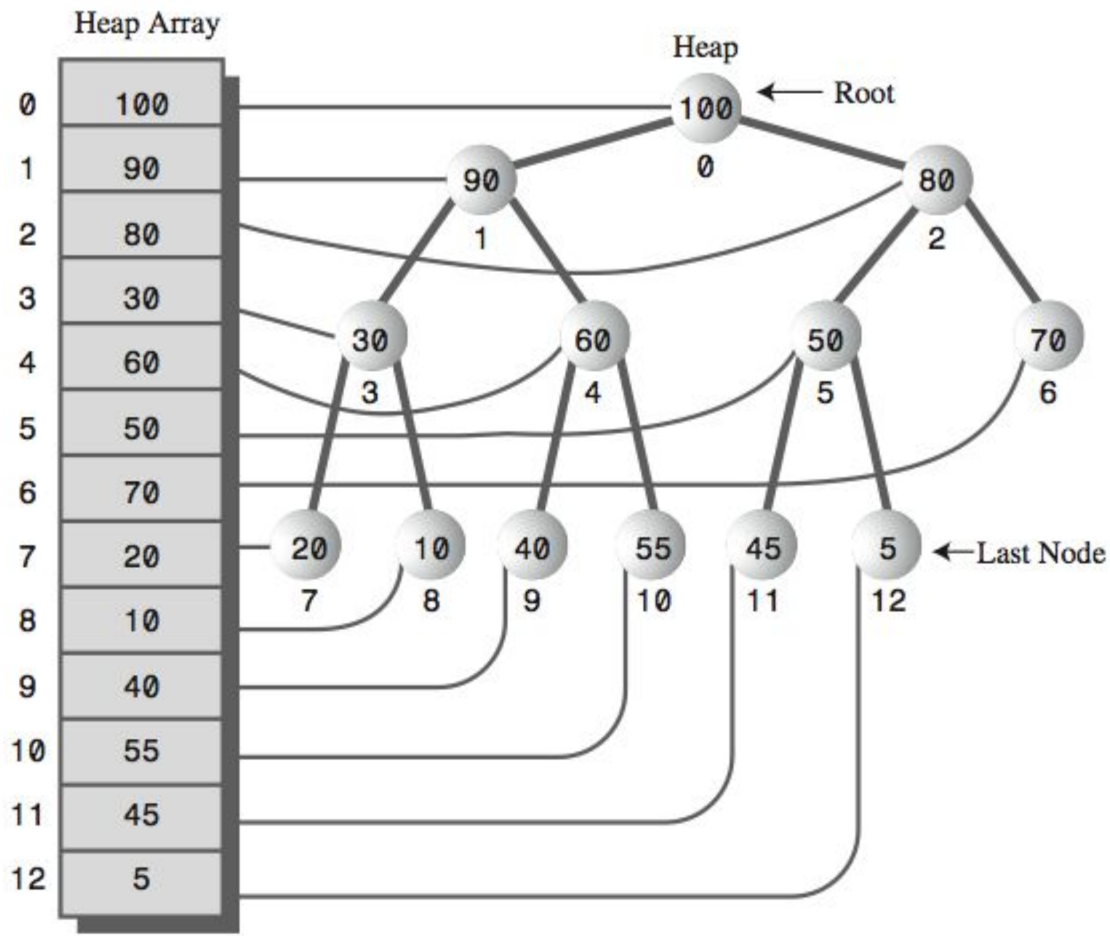


Min heaps also good for priority queues, if "high priority" in your system actually means *low* value (i.e. 1 means most important)

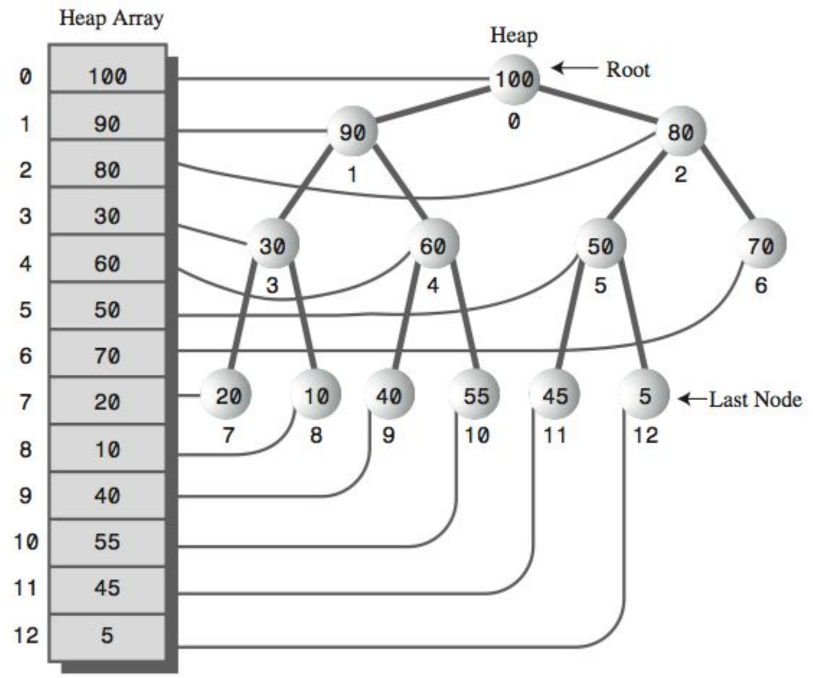
Heap implementation

- We actually do NOT typically use a node object to implement heaps
- Because they must be **complete**, they fit nicely into an array, so we usually do that



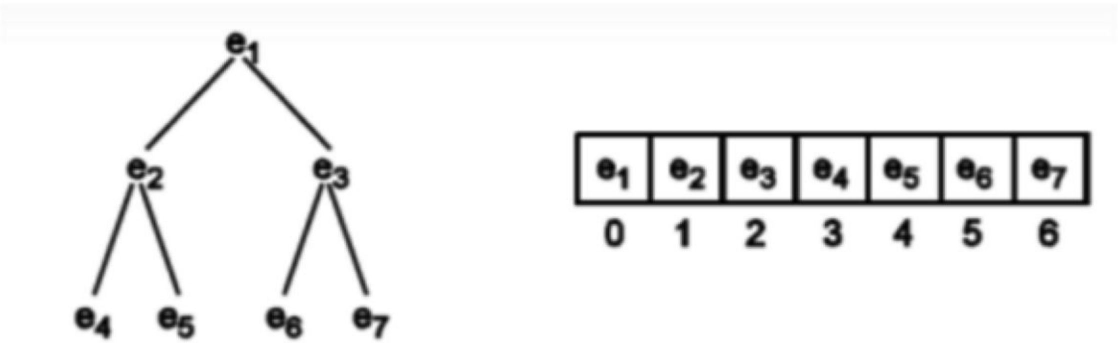

Heap Array
Heap
Root
Last Node
0 100
1 90
2 80
3 30
4 60
5 50
6 70
7 20
8 10
9 40
10 55
11 45
12 5

Parent?



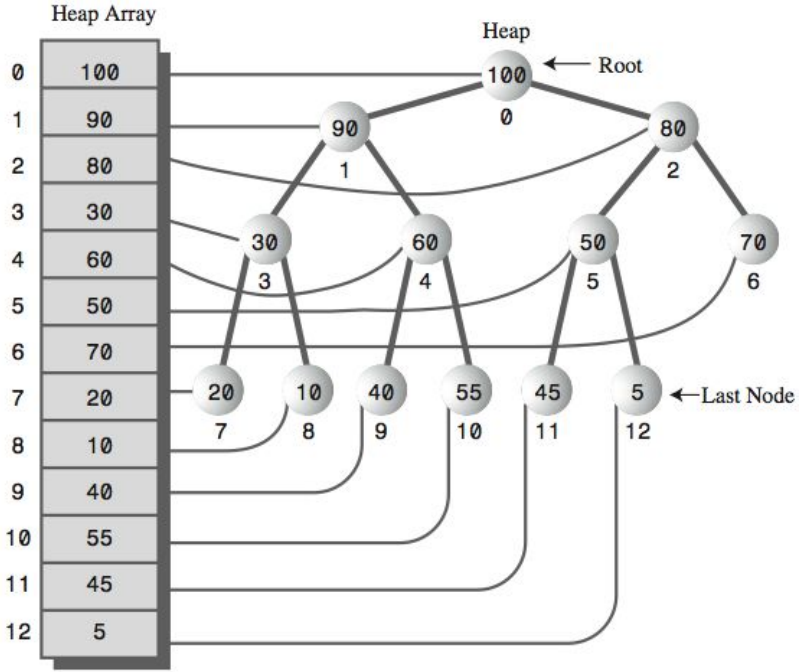
- For a node in array index i: Parent is at array index:
  - A. i – 2
  - B. i / 2
  - C. (i – 1)/2
  - D. 2i

Heap in an array



- For tree of height h, array length is $2^{h+1}-1$
- For a node in array index i:
  - Left child is at array index:
    A. i +1
    B. i + 2
    C. 2i
    D. 2i + 1

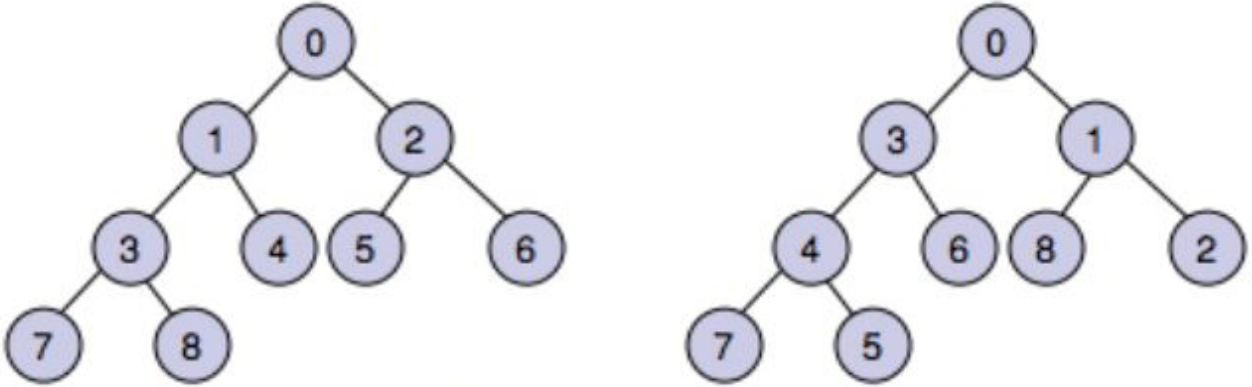
Right child?



- For a node in array index i: Right child is at array index:
  - A. i +1
  - B. i + 2
  - C. 2i
  - D. 2i + 2

Heap concept

Here's an example of 2 possible heaps on the numbers 0-8.



Note that the Heap Property implies that the minimum value is always at the root.

So the min will be the first value to leave the heap.

The Shape Property implies that tree is always perfectly balanced.

Every heap with same number of nodes has the same shape.

Is it a max-heap?

- Arr = (10, 5, 3, 6, 2, 1)

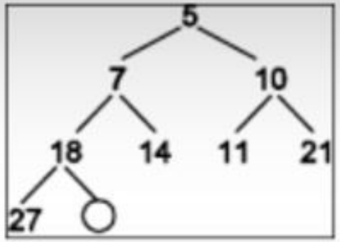
- A: Yes
- B: No

HEAP INSERT AND DELETE

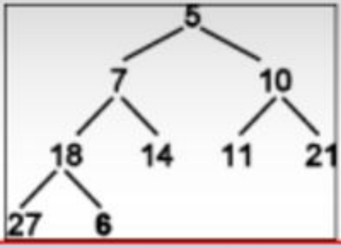
Insert

- Arr = (10, 6, 3, 5, 2, 1)
  - Insert 4. Where should it go??
- Need to minimize time for each operation.
- What place is the fastest to add?
  - To the end of the array: O(1).
  - Preserves completeness of the tree.

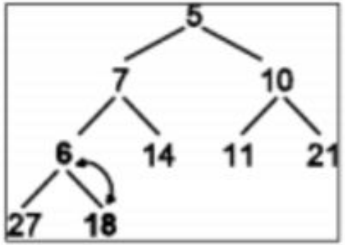
Insert



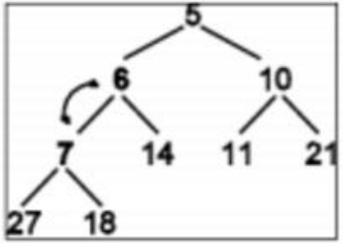
(a) A minheap prior to adding an element. The circle is where the new element will be put initially.



(b) Add the element, 6, as the new rightmost leaf. This maintains a complete binary tree, but may violate the minheap ordering property.



(c) "Bubble up" the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.

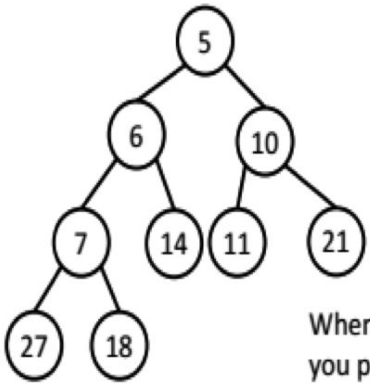


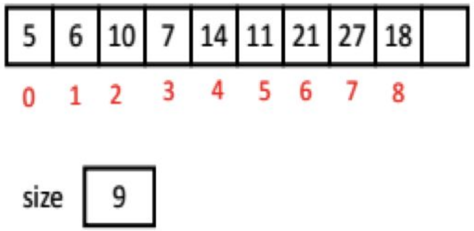
(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.

How long does it take to insert 1 element?

- A: $O(1)$
- B: $O(\log n)$
- C: $O(n)$
- D: $O(n \log n)$
- E: $O(n^2)$

Heap insert



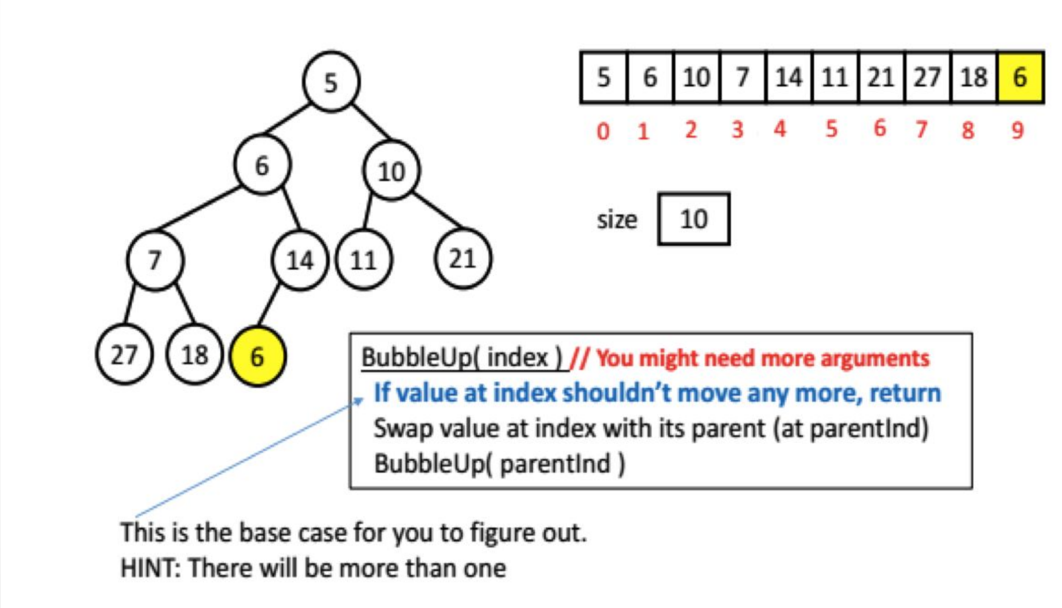


When you add an element to a heap, at what index do you put it initially?

A. 0
B. size – 1
C. size
D. Somewhere else

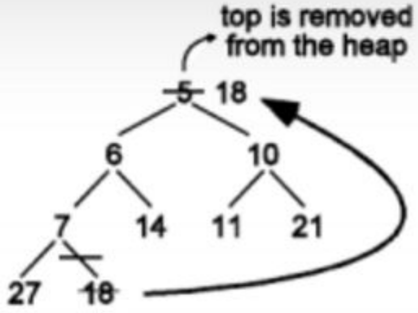
Bubble up

- To insert an element, you will need a helper method called BubbleUp. I suggest recursion, and here's a very rough recursive algorithm, though it's up to you to figure out the base case. You can talk to others in the class about this.

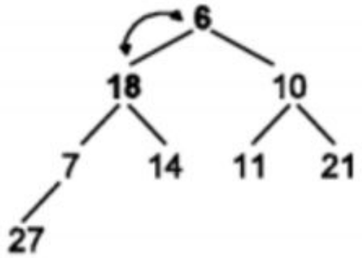


Heap delete

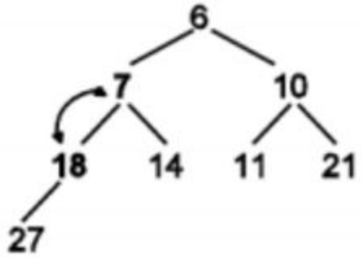
(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.



(b) "Trickle down" the element. Swapping top with the smaller of its two children leaves top's right subtree a valid heap. The subtree rooted at 18 still needs fixing.



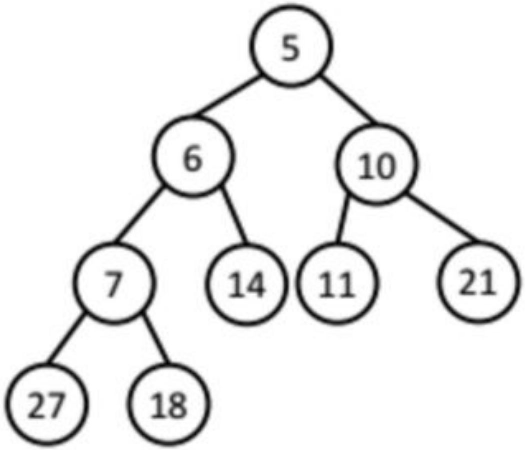
(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored



Heap remove (smallest/largest)

- When you remove an element from an array-backed heap, at what index is the element to remove located? Assume the variable size stores the number of elements currently in the heap, and arr.length is the length of the array storing the heap.
  A. 0
  B. 1
  C. size – 1
  D. arr.length – 1
  E. You can't tell with the information given

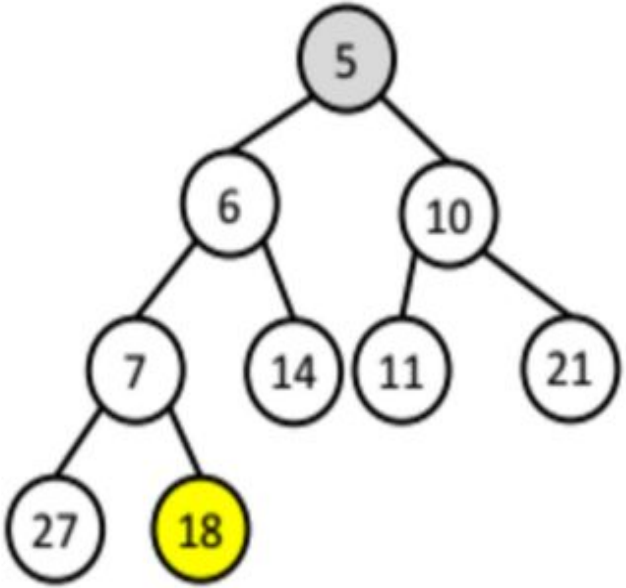
Heap remove (smallest/largest)

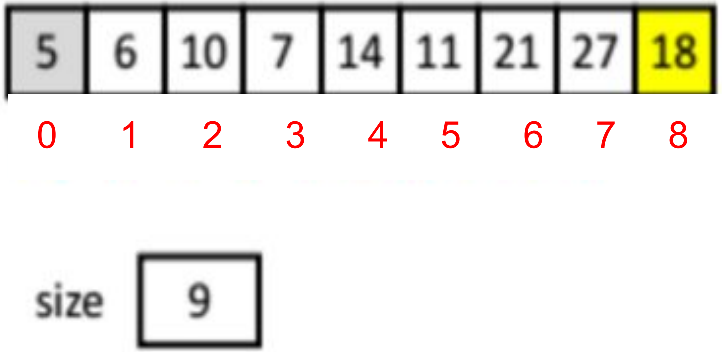


5	6	10	7	14	11	21	27	18
0	1	2	3	4	5	6	7	8

size 9

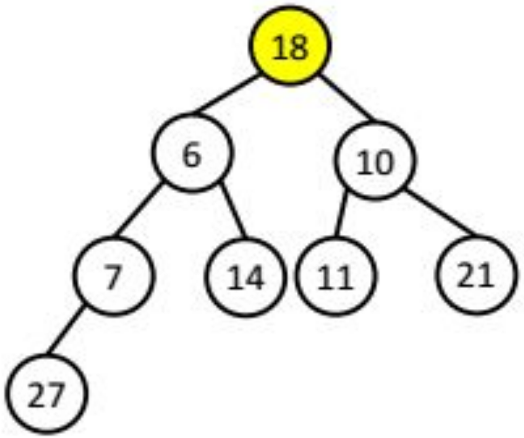
Heap remove (smallest/largest)

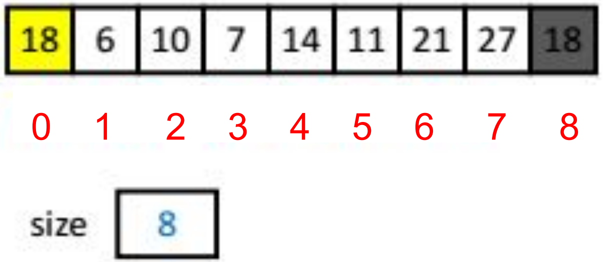





```
theHeap[0] = theHeap[size-1];
```

Heap remove (smallest/largest)



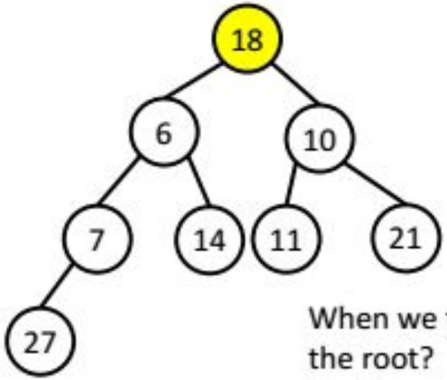



```
theHeap[0] = theHeap[size-1];
size--;
trickleDown( ____?____ );
```

Now 18 needs to trickle down… this should be a separate method. Can be iterative or recursive, but recursive is easier, really!

Trickle down (min heap)

This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.



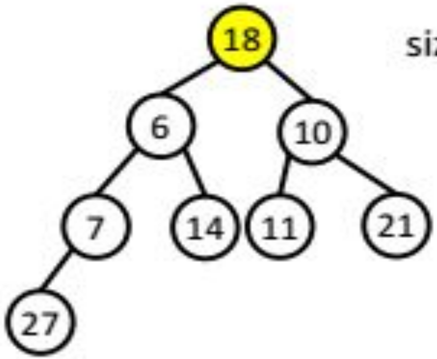
0	1	2	3	4	5	6	7	8
18	6	10	7	14	11	21	27	18

size 8

When we trickle 18 down, which value should become the root?

A. 6
B. 10
C. 27
D. 18 should stay there
E. Other

Trickle down



size 8

18	6	10	7	14	11	21	27	18
0	1	2	3	4	5	6	7	8

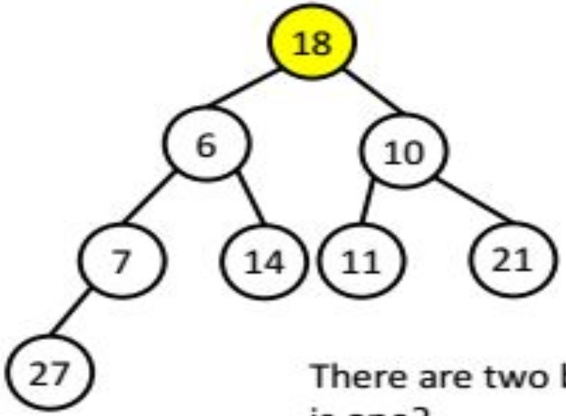
Assume the node to be trickled down is at `index` and that its left and right children are at `lInd` and `rInd`, respectively. Also assume the node at index has two children. Which line correctly completes the code below?

```
if ( ____________________ ) childInd = lInd;
else childInd = rInd;
swap( theHeap, childInd, index );
```

```
A. lInd < rInd
B. lInd < index
C. theHeap[lInd] < theHeap[rInd]
D. theHeap[lInd] < theHeap[index]
```

TrickleDown (min heap)

This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.



18	6	10	7	14	11	21	27	18
0	1	2	3	4	5	6	7	8

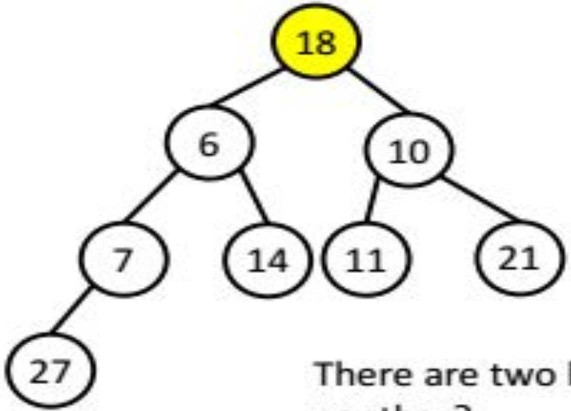
size 8

There are two base cases for this method. Which of these is one?

A. 18 is in a leaf node
B. 18 is in a node with one child
C. 18 is a node with 2 children
D. 18 is at the root of the heap

TrickleDown (min heap)

This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.



18	6	10	7	14	11	21	27	18
0	1	2	3	4	5	6	7	8

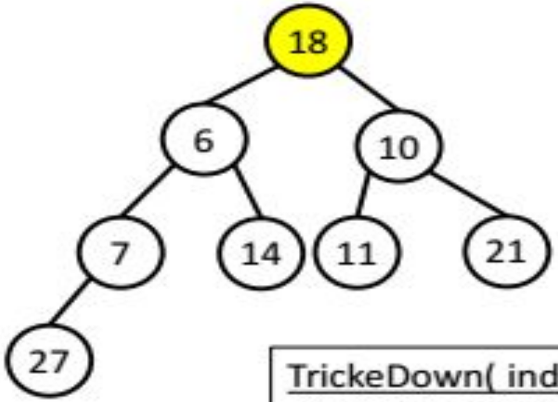
size 8

There are two base cases for this method. Which of these is another?

A. 18 has no children less than itself
B. 18 has no more than one child less than itself.
C. 18 has exactly one child, which is greater than or equal to it

TrickleDown (min heap)

Here's a rough recursive algorithm for trickleDown. It's up to you to translate this to code! And careful, because there are subtleties not mentioned here (e.g., what if the node has only one child?



0	1	2	3	4	5	6	7	8
18	6	10	7	14	11	21	27	18

size 8

```
TrickeDown( index ) // You might need more arguments
  If value at index is a leaf, return
  If value at index has no children less than it, return
  Swap value at index with its smaller child (at childInd)
  TrickleDown( childInd )
```

Finding an arbitrary node

- Heaps offer fast access to the *largest/smallest* node in the heap.

- What if the element that the user wants to find is NOT the largest/smallest element?
- A: Since it is a binary tree we can find it fast O(log n)
- B: The binary tree does not help here and we need to search through the entire array O(n).

Finding an arbitrary node

- If the element is not the largest/smallest, then it could be anywhere in the heap.
- This contrasts with binary *search* trees
- Hence, to find an element within a heap, we **must** search through the entire heap.

Removing an arbitrary node n (max heap)

- Let assume we found the *index* of node n.
- Then we can swap the last node in the heap (rightmost child of last level) with n.
- Then we just *trickleDown* that node and we are done. Right?

A: Yes, we can use the same algorithm

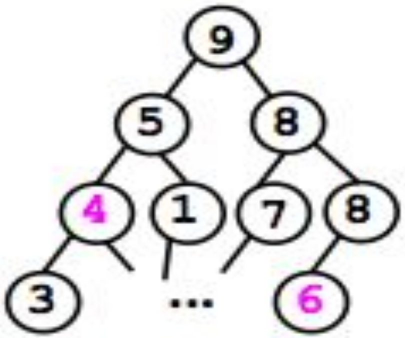
B: No, the solution is different

Wrong!

- The above procedure worked for removeLargest() because we always started from the **top** (root) of the heap.
  - By trickling down from the top, we guarantee that every sub-tree (starting from the very top) is a **valid** heap.
- When removing an *arbitrary* node, the trickleDown process will “fix” the sub-tree rooted at n, but not necessarily the whole tree.

Removing an arbitrary node

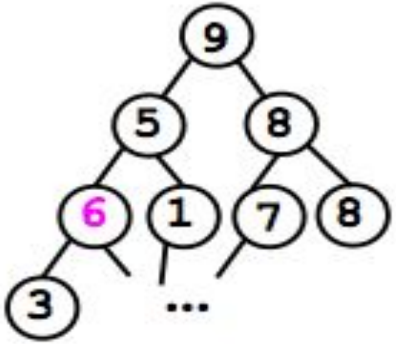
- Suppose we wish to remove the node containing 4.
- If we just replace it with the "last" node (6)...



Valid heap.

Removing an arbitrary node

- ...then the `trickleDown()` method will do nothing (6 is already bigger than its children).
- Moreover, 6 is now bigger than its parent -- a *violation of the heap condition*.



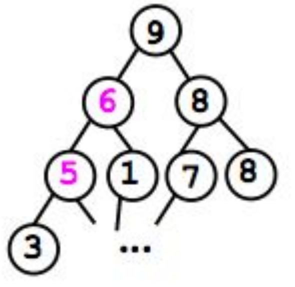
Invalid heap.

Removing an arbitrary node

```
void remove (..)
{
   find the node n,
      if ( n > heap[lastIndex])
          swap n with heap[lastIndex]
          --?--
      else
         swap n with heap[lastIndex]
         --?--
}
```

Removing an arbitrary node

```
void remove (..)
{
    find the node n,
        if ( n > heap[lastIndex])
            swap n with heap[lastIndex]
            trickleDown
        else
           swap n with heap[lastIndex]
           BubbleUp
}
```



Time cost

- Find ?

- Remove
  - Find ?
  - Swap ?
  - Either trickle node down or bubble up ?

Time cost

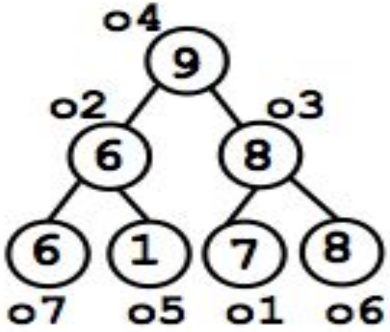
- Find O(n)
  - Search through all nodes.
- Remove
  - Find O(n)
  - Swap O(1)
  - Either trickle node down or bubble up O(log n)

- Total is O(n)+O(1)+O(log n)=O(n).

Increasing/decreasing priority

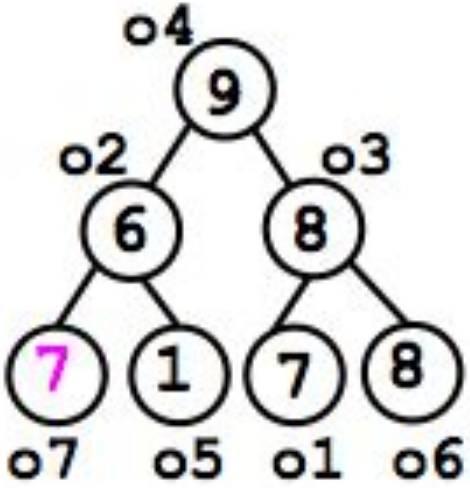
- Example:

```
heap.add(o1);  // Priority 7
heap.add(o2);  // Priority 6
...
heap.add(o7);  // Priority 5
```



Are we in trouble??

- A: Yes, the heap is ruined ;(
- B: No, just BubbleUp!



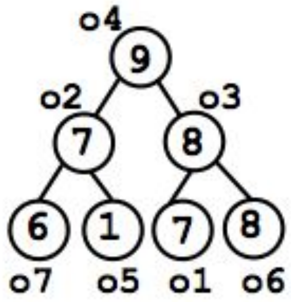
Increasing/decreasing priority

- Example:

```
heap.add(o1);  // Priority 7
heap.add(o2);  // Priority 6
...
heap.add(o7);  // Priority 5
```

- Later on:

```
heap.increasePriority(o7);
```



Done.

Binary heap vs d-heap (PA07)

HEAP SORT

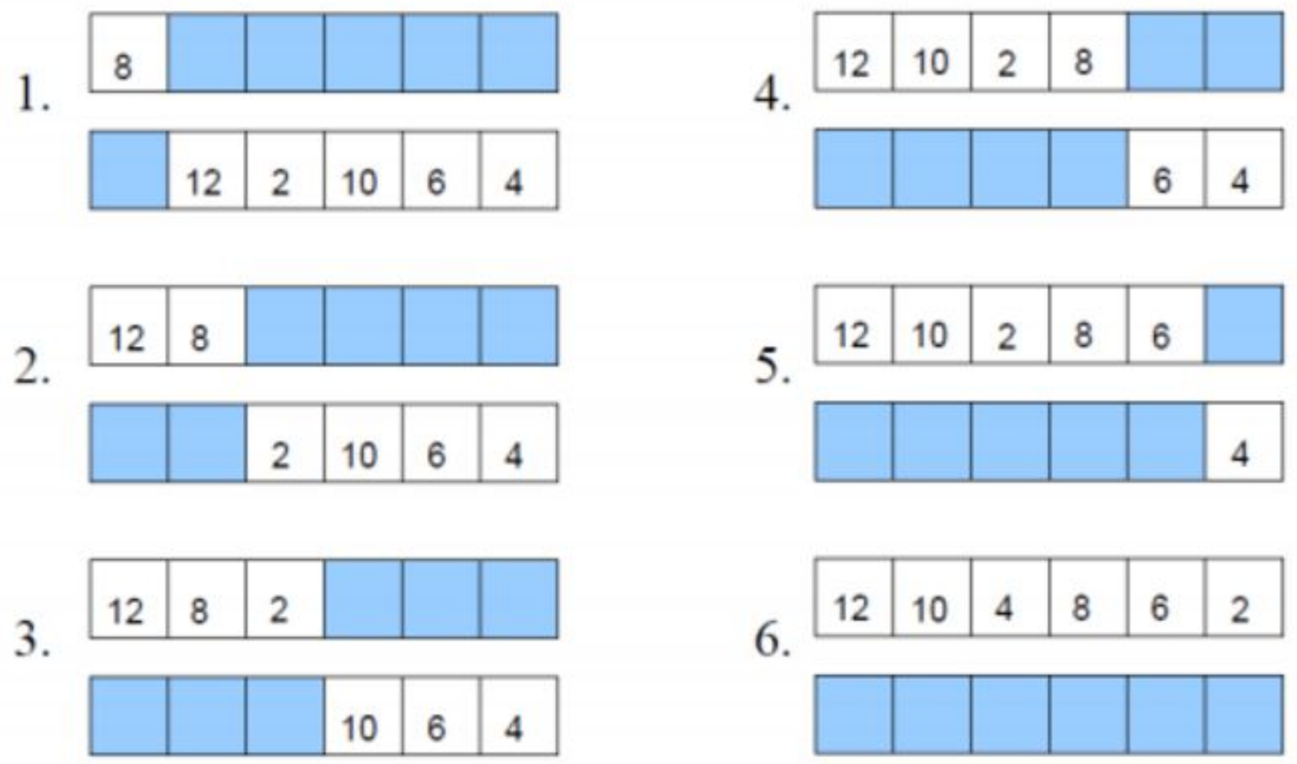
Heapsort

Heapsort is super easy

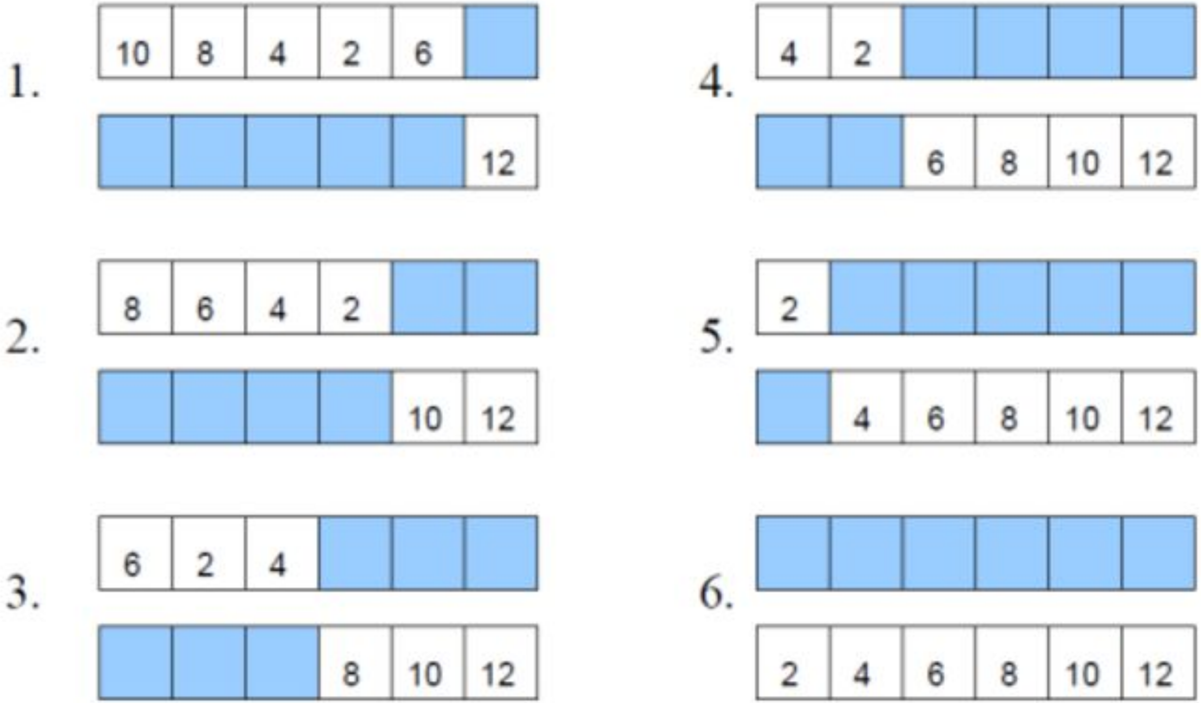
1. Insert unsorted elements one at a time into a heap until all are added
2. Remove them from the heap one at a time (we will always be removing the next biggest item, for max-heap; or next smallest item, for min-heap)

- Unlike mergesort, we don’t need a separate array for our workspace.
- We can do it all in place in one array (the same array we were given as input)

Build heap by inserting elements one at a time:



Sort array by removing elements one at a time:



Build heap by inserting elements one at a time IN PLACE:

1.

8	12	2	10	6	4

2.

12	8	2	10	6	4

3.

12	8	2	10	6	4

4.

12	10	2	8	6	4

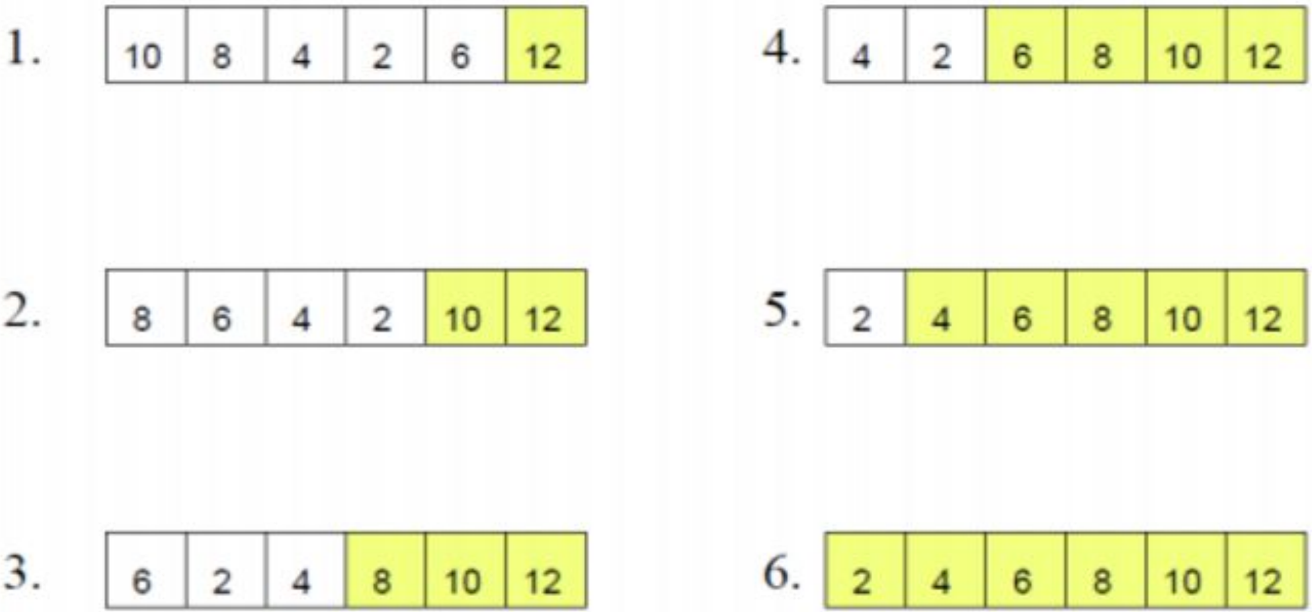
5.

12	10	2	8	6	4

6.

12	10	4	8	6	2

Sort array by removing elements one at a time IN PLACE:



Complexity of heapsort (best, then again for worst)?

- A: O(1)
- B: O(log n)
- C: O(n)
- D: O(n log n)
- E: O(n^2)