Question 1: Answer the following questions:

(a) Let T be an ordered tree (i.e., the children of each node are ordered) with more than one node. Is it possible that the in-order traversal of T visits the nodes in the same order as the pre-order traversal? If so, give an example; otherwise argue why it cannot happen. Likewise, is it possible that post-order traversal of T visits the nodes in the reverse order as pre-order traversal? If so, give an example; otherwise argue why it cannot happen:

(b) Answer the previous questions if T is a proper binary tree with more than one node.

Solution:

(a) That the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order is not possible. A postorder traversal node will always visit an external node first whereas a preorder traversal will always visit the root node first, while.

Considering the case of a tree with only two nodes, it is possible for a preorder and a postorder traversal to visit the nodes in the reverse order.

(b) It is not possible for the post and pre order traversals to visit the nodes of a proper binary tree in the same order for the same reason in the previous question.

It is not possible for the post and pre order traversals to visit the nodes of a proper binary tree in the reverse order.

Let X be the root of a proper binary tree and let T_1 and T_2 be the left and right subtrees. A postorder traversal would visit the postorder traversal of T_1, the postorder traversal of T_2 and then node a while the preorder traversal would visit node X, the preorder traversal of T_1 and then the preorder traversal of T_2. Clearly, the postorder and preorder traversals cannot be the reverse of each other since in both cases; all the nodes of T_1 are visited before all the nodes of T_2.

Question 2: Answer the following questions:

(a) Show (diagrammatically) all the steps involved in the heap sort on the following input sequence: 7, 45, 1, 21, 2, 64, 4, 18, 9, 6. You should sort the sequence in the increasing order.

(b) Repeat part a) if the heap sort is performed **in place**.

Solution:

(a) Show how the Heap-Sort algorithm performs for the following input sequence: 7, 45, 1, 21, 2, 64, 4, 18, 9, 6.

Creating the current Binary Max Heap

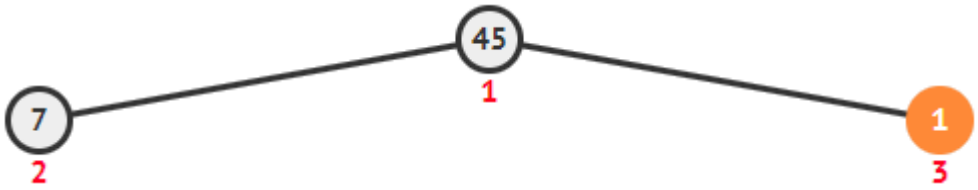
1. Insert 7
   Sequence: 45, 1, 21, 2, 64, 4, 18, 9, 6

2. Insert 45
   Sequence: 1, 21, 2, 64, 4, 18, 9, 6

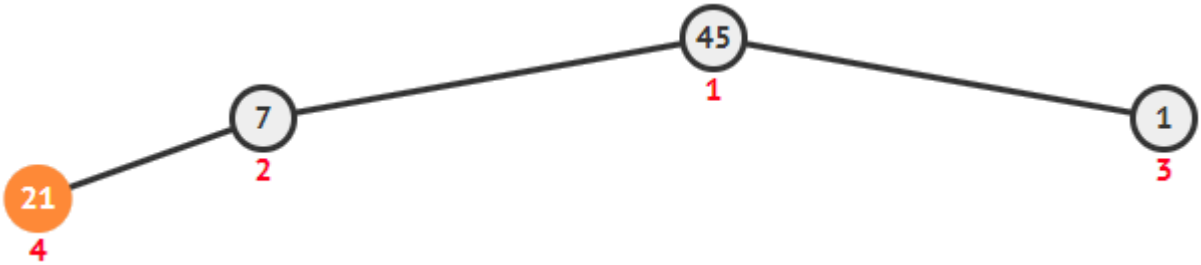
   Compare 7 and 45

   7 < 45, so swap them

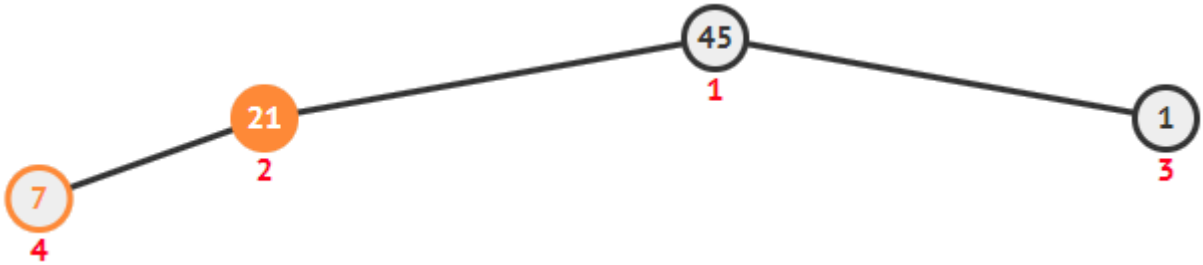
3. Insert 1, at the back of compact array
   Sequence: 21, 2, 64, 4, 18, 9, 6



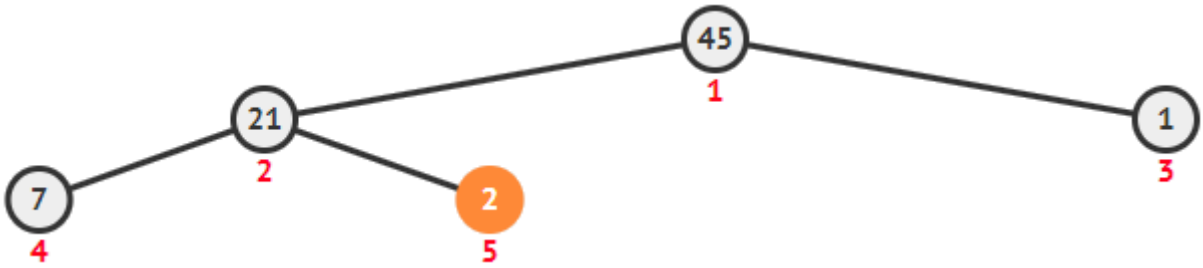
4. Insert 21, at the back of compact array
   Sequence: 2, 64, 4, 18, 9, 6
   Compare 21 and 7



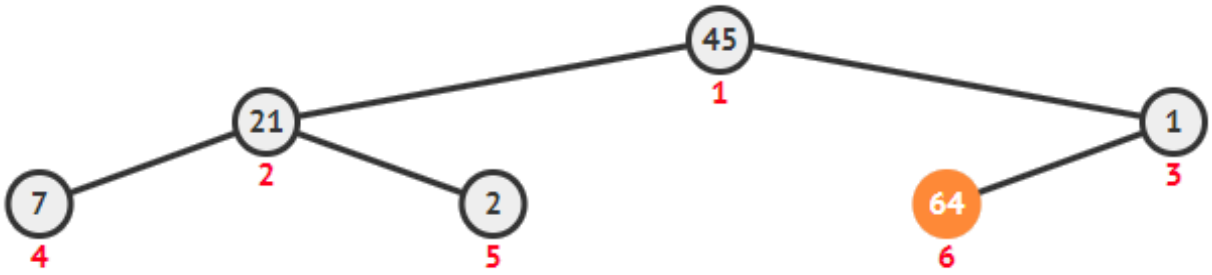
   7 < 21, so swap them



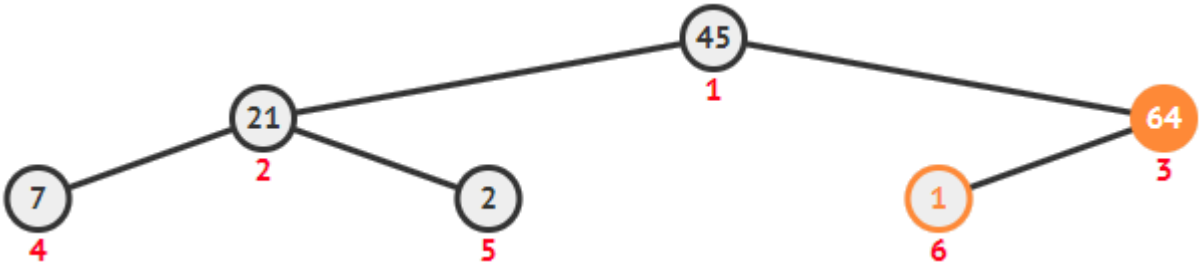
5. Insert 2, at the back of compact array
   Sequence: 64, 4, 18, 9, 6



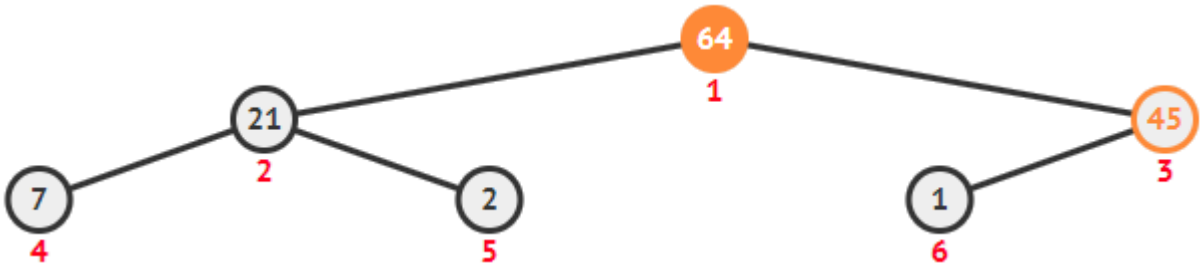
6. Insert 64, at the back of compact array
   Sequence: 4, 18, 9, 6
   Compare 64 and 1



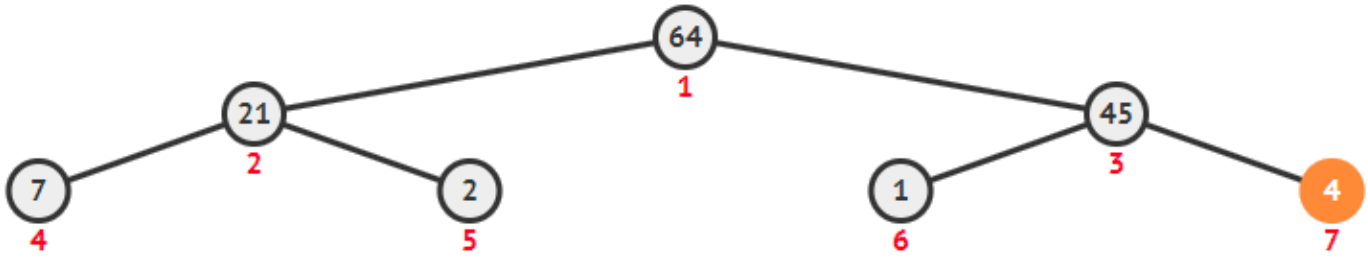
1 < 64, so swap them.



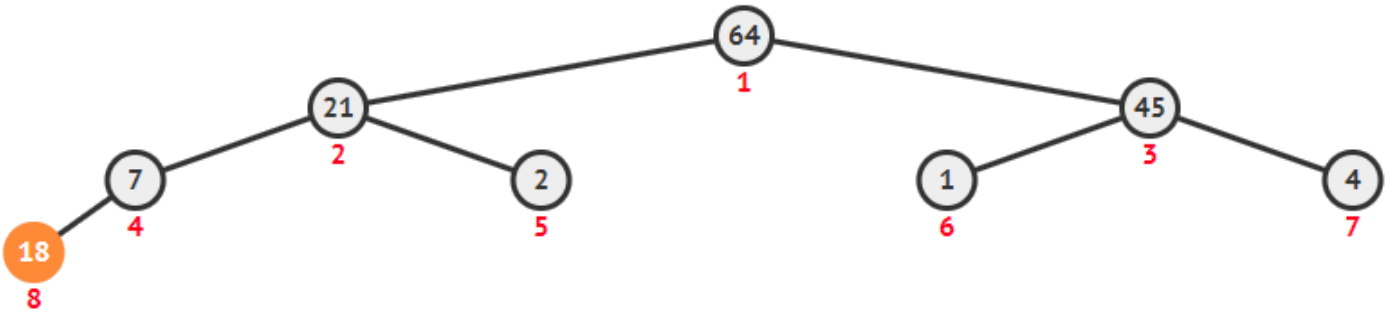
Compare 64 and 45
45 < 64, so swap them.



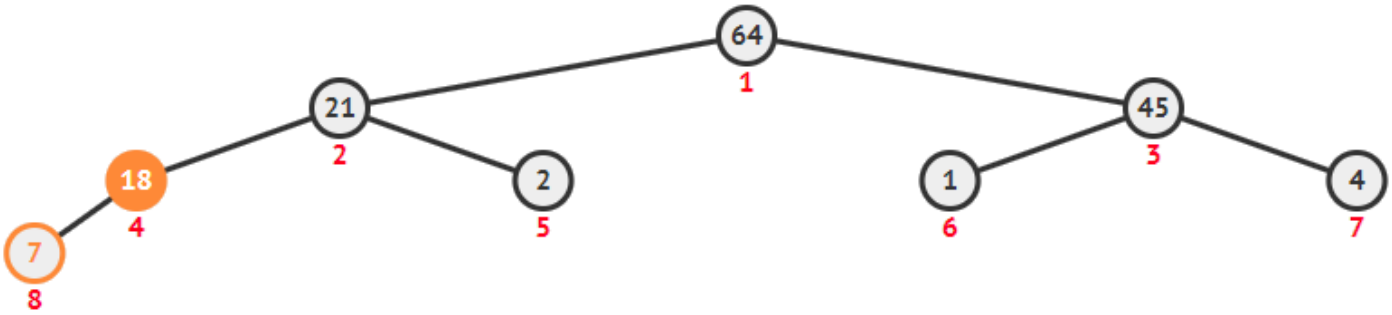
7. Insert 4, at the back of compact array
   Sequence: 18, 9, 6



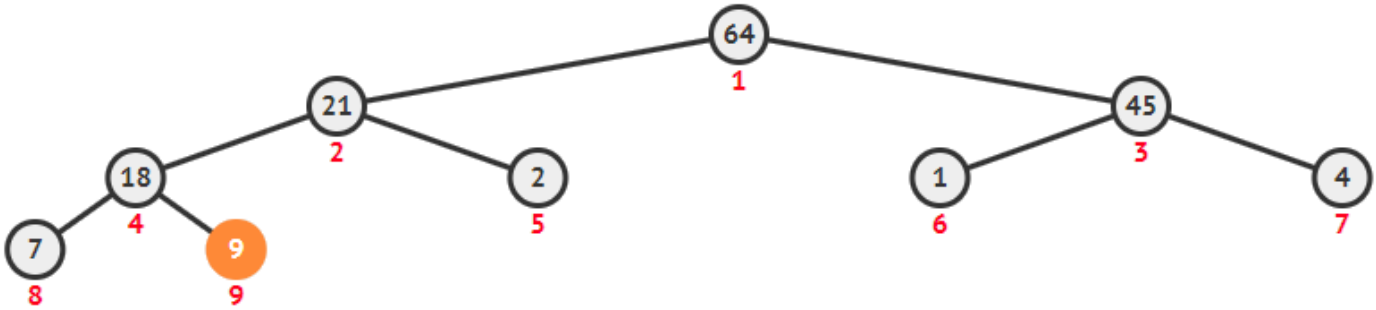
8. Insert 18, at the back of compact array
   Sequence: 9, 6



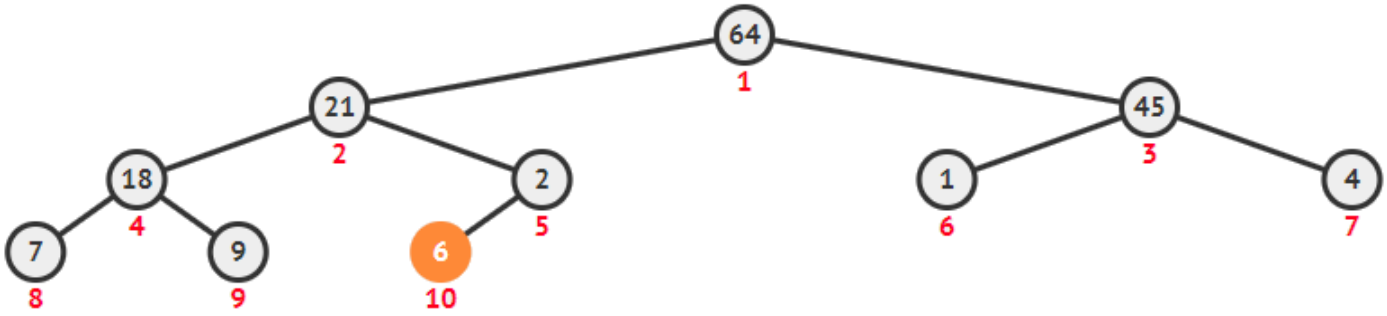
Compare 7 and 18
7 < 18, so swap them



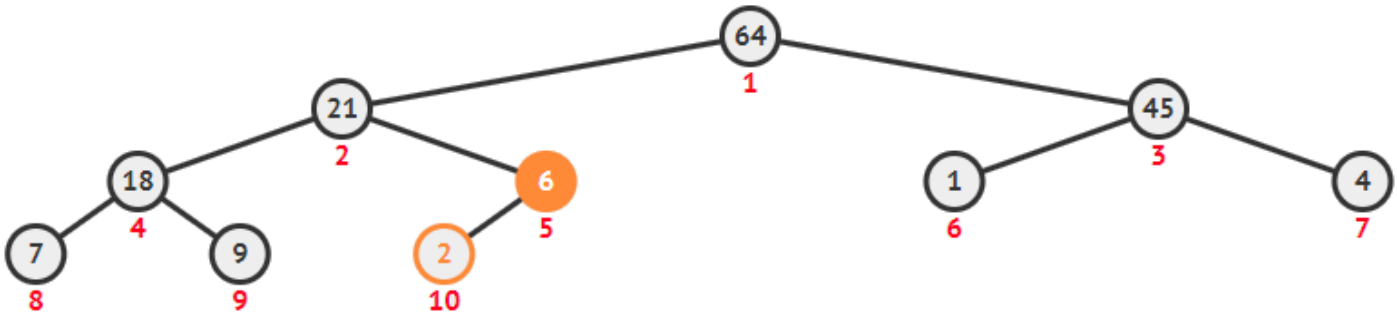
9. Insert 9, at the back of compact array
   Sequence: 6



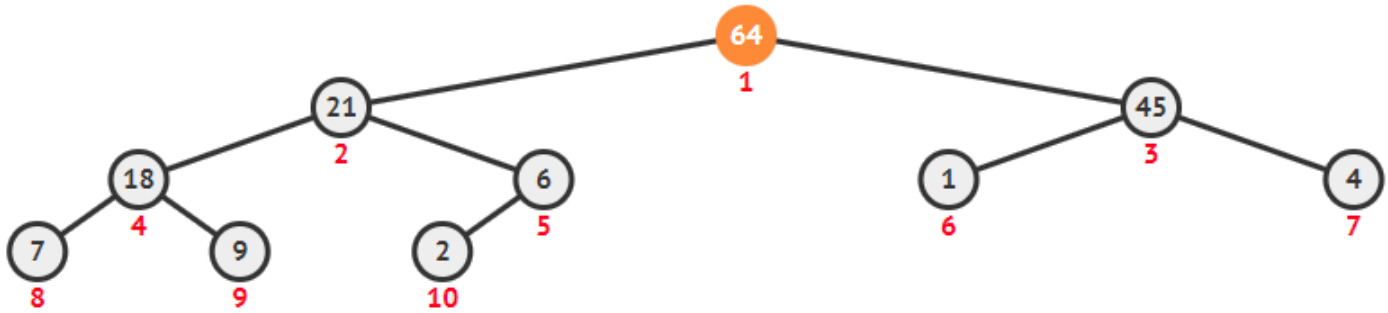
10. Insert 6, at the back of compact array
    Sequence: empty



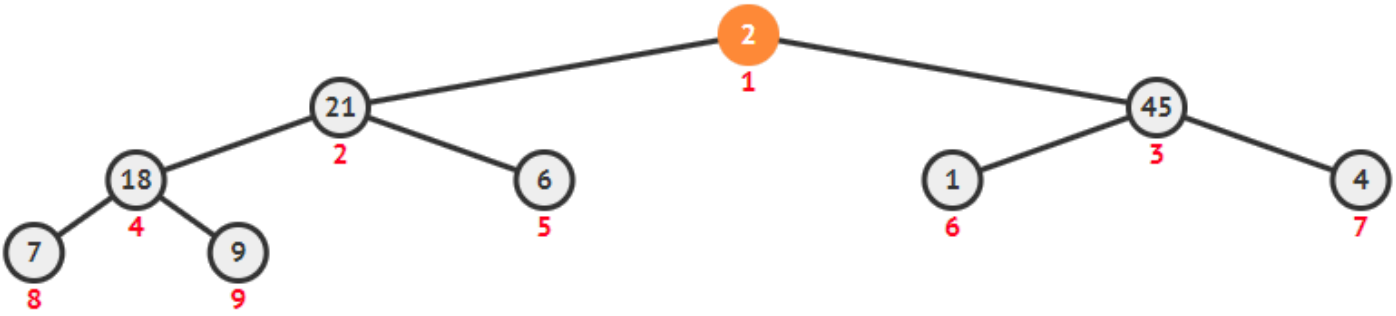
Compare 2 and 6
2 < 6, so swap them



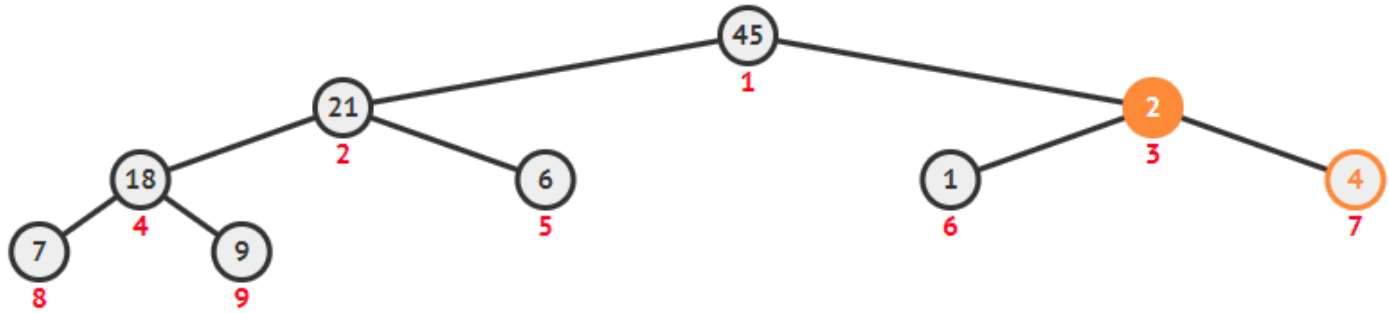
Sorting:
Root stores the max item.



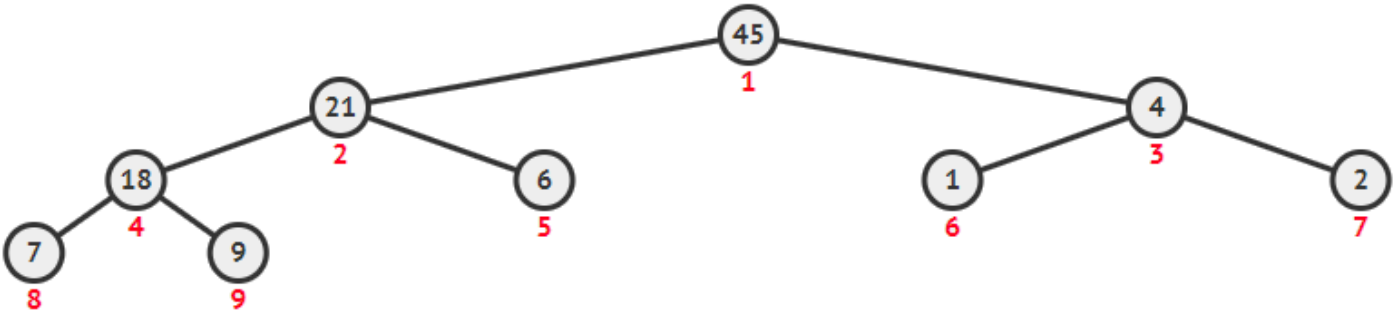
Take out the root 64.
Replace root with the last leaf 2.



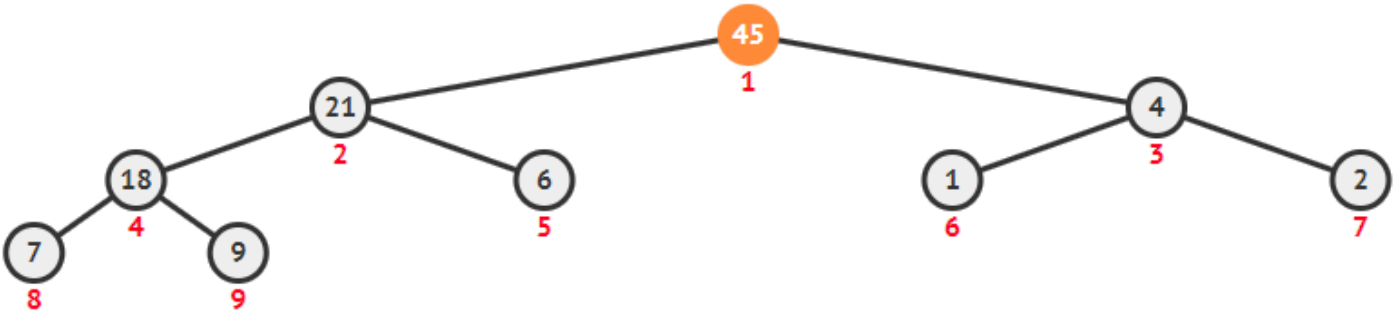
Compare the 2 child nodes (21 and 45), 45>21, so compare the new root with the leaf on its right 45; 2 < 45, so swap them.



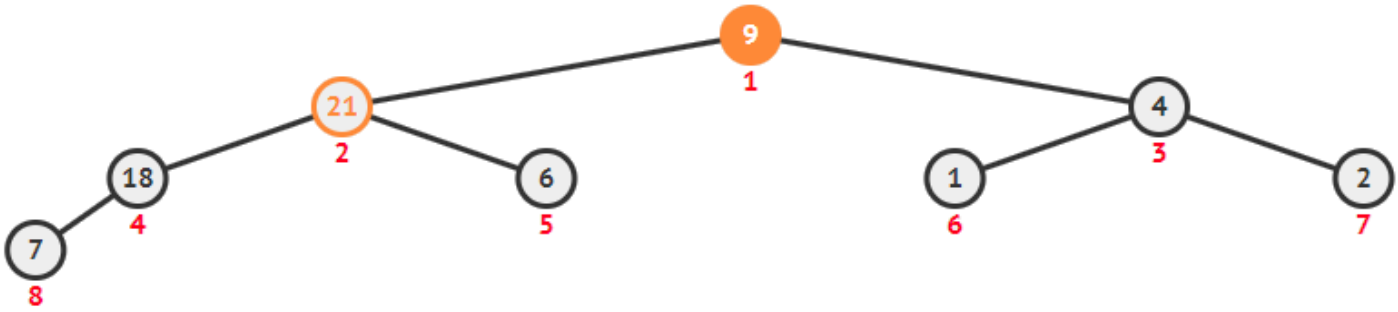
Compare the 2 child nodes (1 and 4), 4 > 1, so compare that new leaf with the leaf on its right; 2 < 4, so swap them.



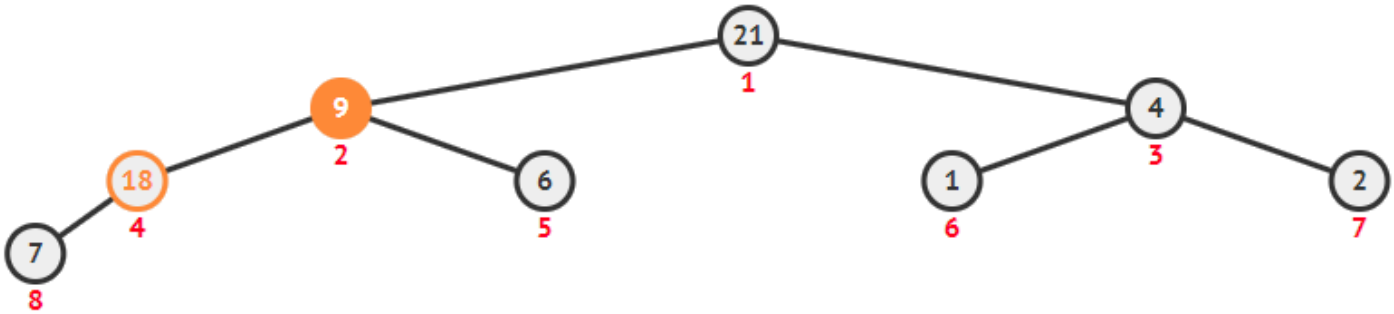
ExtractMax() has been done. There returns the max item: 64.
The partial sorted order is -1,-1,-1,-1,-1,-1,-1,-1,-1,64.
Perform another ExtractMax() iteration with the new root stores the max item 45.



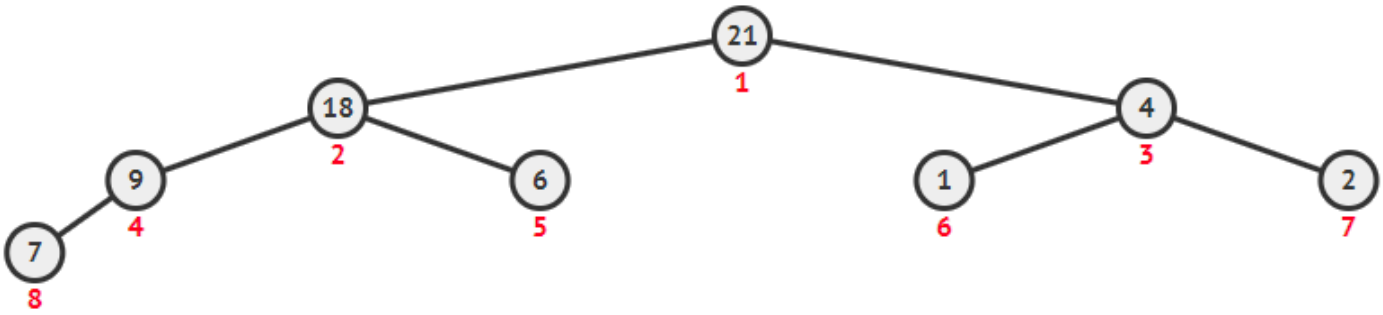
Take out the root 45.
Replace root with the last leaf 9.



Compare the 2 child nodes (21 and 4), 21 > 4, so compare the new root with the leaf on its left; (21 > 9), so do not swap them.



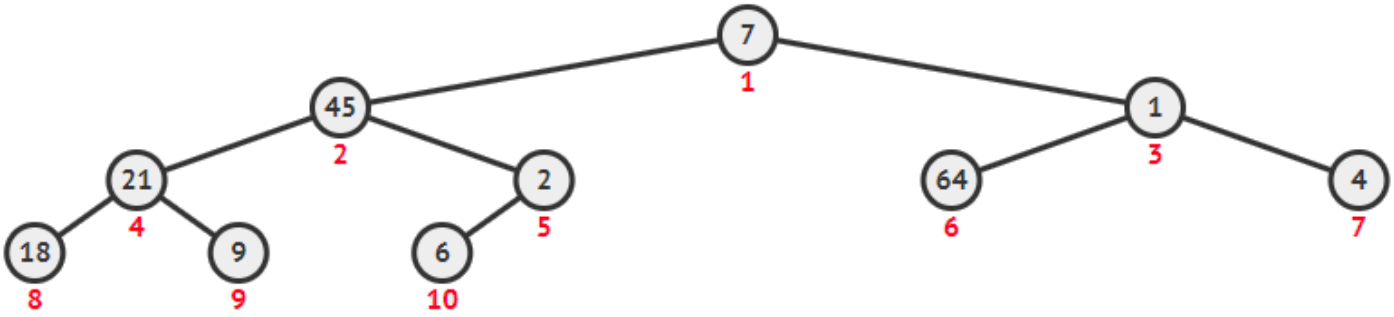
Compare the 2 child nodes (18 and 6), 18 > 6, so compare that new leaf with the leaf on its left; (9 < 18), so swap them.



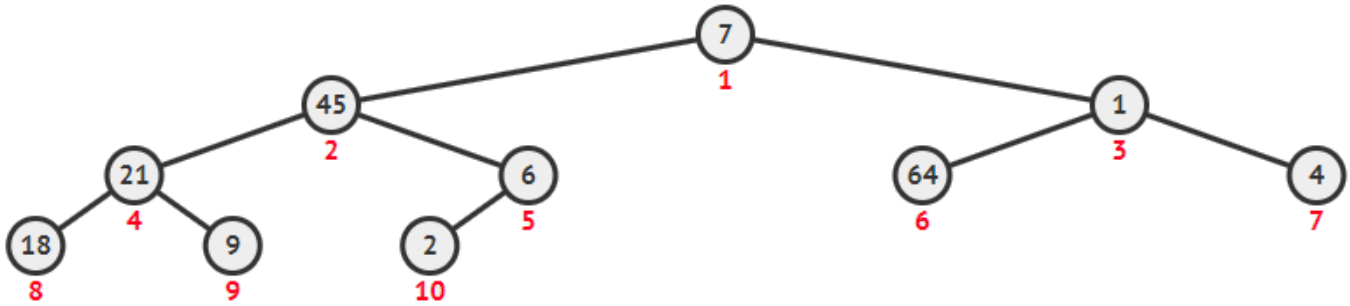
ExtractMax() has been done. There returns the max item: 45.
The partial sorted order is -1,-1,-1,-1,-1,-1,-1,-1,45,64.
Perform another ExtractMax() iteration with the new root stores the max item 21.
Please see the text below (without demo pictures)
- Take out the root 21.
- Replace root with the last leaf 7.
- The new root 7.
- 7 < 18, so swap them.
- 7 < 9, so swap them.
- ExtractMax() has been done. There returns the max item: 21.
- The partial sorted order is -1,-1,-1,-1,-1,-1,18,21,45,64.
- Perform another ExtractMax() iteration with the new root stores the max item 18.
- Take out the root 18.
- Replace root with the last leaf 2.
- The new root 2.
- 2 < 9, so swap them.
- 2 < 7, so swap them.
- ExtractMax() has been done. There returns the max item: 18.
- The partial sorted order is -1,-1,-1,-1,-1,-1,18,21,45,64.
- Perform another ExtractMax() iteration with the new root stores the max item 9.
- Take out the root 9
- Replace root with the last leaf 1.
- The new root 1.
- 1 < 7, so swap them.
- 1 < 6, so swap them.
- ExtractMax() has been done. There returns the max item: 9.
- The partial sorted order is -1,-1,-1,-1,-1,9,18,21,45,64.
- Perform another ExtractMax() iteration with the new root stores the max item 7.
- Take out the root 7
- Replace root with the last leaf 1.
- The new root 1.
- 1 < 6, so swap them.
- 1 < 2, so swap them.
- ExtractMax() has been done. There returns the max item: 7.
- The partial sorted order is -1,-1,-1,-1,7,9,18,21,45,64.
- Perform another ExtractMax() iteration with the new root stores the max item 6.
- Take out the root 6
- Replace root with the last leaf 1.
- The new root 1.
- 1 < 4, so swap them.
- ExtractMax() has been done. There returns the max item: 6.
- The partial sorted order is -1,-1,-1,6,7,9,18,21,45,64.

- Perform another ExtractMax() iteration with the new root stores the max item 4.
- Take out the root 4
- Replace root with the last leaf 1.
- The new root 1.
- 1 < 2, so swap them.
- ExtractMax() has been done. There returns the max item: 4.
- The partial sorted order is -1,-1,4,6,7,9,18,21,45,64
- Perform another ExtractMax() iteration with the new root stores the max item 2.
- Take out the root 2
- Replace root with the last leaf 1.
- The new root 1.
- ExtractMax() has been done. There returns the max item: 2.
- The partial sorted order is -1,2,4,6,7,9,18,21,45,64
- Perform another ExtractMax() iteration with the new root stores the max item 1.
- Take out the root 1
- ExtractMax() has been done. There returns the max item: 1.
- The partial sorted order is 1,2,4,6,7,9,18,21,45,64

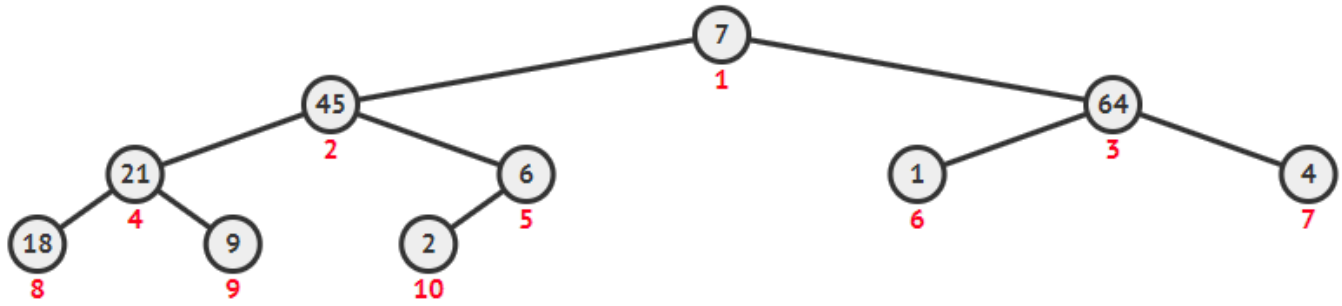
(b) Repeat part a) if the heap sort is performed **in place**.
Create(A) - O(N): 7, 45, 1, 21, 2, 64, 4, 18, 9, 6



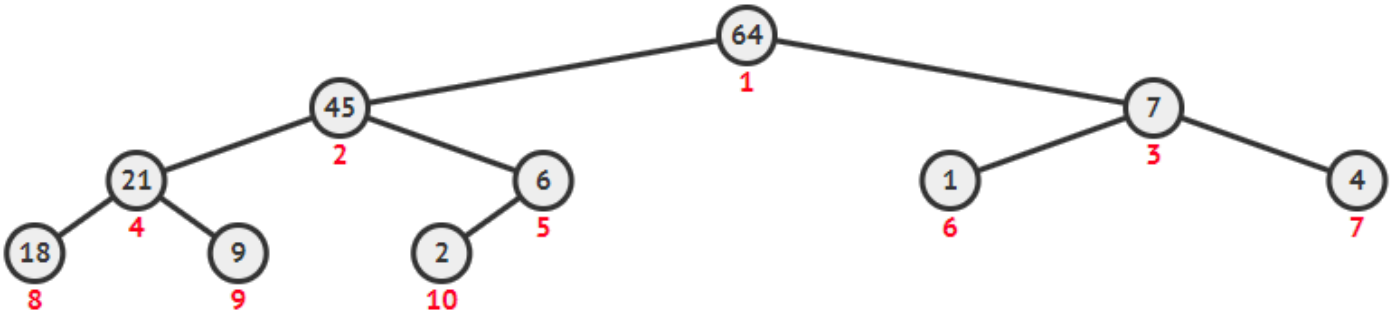
shiftDown(5) to fix the Binary Max Heap property (if necessary) of sub-tree rooted at 2.
2 < 6, so swap them.



shiftDown(4) to fix the Binary Max Heap property (if necessary) of sub-tree rooted at 21. (No swapping needed).
shiftDown(3) to fix the Binary Max Heap property (if necessary) of sub-tree rooted at 1.
1 < 64, so swap them.
shiftDown(2) to fix the Binary Max Heap property (if necessary) of sub-tree rooted at 45. (No swapping needed).



shiftDown(1) to fix the Binary Max Heap property (if necessary) of sub-tree rooted at 7.
$7 < 64$, so swap them.



Repeating the steps like question a. To heap sort the tree.

---

3. **Q3: Answer the following questions:**

(a) Bill suggested that he can implement a queue ADT using a priority queue and a constant amount of additional space. Recall that a stack supports enqueue and dequeue operations, and a priority queue supports min(), removeMin and insert operations. Explain how Bill can achieve this by showing the necessary pseudo code for implementing the stack ADT.

(b) Referring to a) above, what are the tight big-Oh time complexities of the push and pop operations of the newly designed stack given that the priority queue is implemented using a heap? Explain your answer.

Solution:

(a)

```
CLASS PriorityQueue {
  // array in sorted order, from max at 0 to min at size-1
  int maxSize;

  int[] queueArray;

  int noOfItems;

  CONSTRUCTOR PriorityQueue(int s) {
    maxSize <- s;
    queueArray <- new int[maxSize];
    noOfItems <- 0;
  }

  FUNCTION insert(int newItem) {
    int i;

    IF (noOfItems == 0)
      queueArray[noOfItems++] <- newItem; // insert at 0
    ELSE
```

```
      FOR (i = noOfItems - 1; i >= 0; i--) // start at end,
      {
        IF (newItem > queueArray[i]) // if new item larger,
          queueArray[i + 1] <- queueArray[i]; // shift upward
        ELSE
          // if smaller,
          break; // done shifting
            END ELSE
      }
      queueArray[i + 1] <- newItem; // insert it
      noOfItems++;
    END ELSE// end else (noOfItems > 0)
  }

  FUNCTION int removeMin(){
    return queueArray[--noOfItems];
  }

  FUNCTION int getMin(){
    return queueArray[noOfItems - 1];
  }

  FUNCTION boolean isEmpty(){
    return (noOfItems == 0);
  }

  FUNCTION boolean isFull(){
    return (noOfItems == maxSize);
  }

END CLASS
```

(b) The big-Oh time complexities of the push and pop operations of the newly designed stack.

```
STRUCT stackItem<int key, int value>;

// User defined stack class
CLASS Stack

    int noOfElements;
    PriorityQueue pq <- new PriorityQueue(stackItem<int, int>);

// push function increases noOfElements by 1 and
// inserts this noOfElements with the original value.
FUNCTION push(int n){
    noOfElements++;
    pq.insert(stackItem<noOfElements, n>);
}

// pops element and reduces count.
FUNCTION POP(){
    IF(pq.empty())
      OUTPUT "Nothing to pop!!!"
      END IF
    noOfElements--;
    pq.remove();
}

// returns the top element in the stack using
// cnt as key to determine top(highest priority),
```

```
// default comparator for pairs works fine in this case
FUNCTION int top(){
    stackItem temp=pq.top();
    return temp.value;
}

// return true if stack is empty
FUNCTION bool isEmpty(){
    return pq.empty();
}

END CLASS
```

The above implementation would take O(log n) time for both push and pop operations.

As we have used priority queue to implement the stack, we are assigning priority to the elements that are being pushed.

This type of stack requires elements to be processed in LIFO manner. The idea is to use a counter determining when it was pushed. This counter works as a key for the priority queue. Hence, the implementation of stack uses a priority queue of pairs, with the first element serving as the key. pair <int, int> (key, value)

---

4. **Q4: Programming Questions**: In class, we discussed the priority queue (PQ) ADT implemented using min-heap. In a minheap, the element of the heap with the smallest key is the root of the binary tree. On the other hand, a max-heap has as root the element with the biggest key, and the relationship between the keys of a node and its parent is reversed of that of a min-heap. We also discussed an array-based implementation of heaps.

In this assignment, your task is to implement your own flexible priority queue ADT using both min- and max-heap in Java. The specifications of the flexible priority queue are provided in the following. The heap(s) must be implemented from scratch using an array that is automatically extendable. You are not allowed to use any list (including arraylist), tree, vector, or heap implementation already available in Java. You must not duplicate code for implementing min- and max-heaps (i.e. the same code must be used for constructing min or max heaps. Hence think of a flexible way to parameterize your code for either min- or max heap)