Lecture 11: Priority Queue and Heap

Read: Carrano, Chpt.11.

"Good" characteristics of BST:

- Simplicity in concept & implementation.
- Objects stored in a BST can easily be sorted.
- Support general find and delete operations as well as special searchMin(Max) and deleteMin(Max) operations.
- Very efficient on average with $T_a(n) = O(\lg n)$.

"Bad" characteristics of BST:

- Worst-case complexity depends on height of tree; hence, $T_w(n) = O(n)$.
- Less efficient when many items are having identical keys.

Q: Can we design an efficient ADT with $T_w(n) = T_a(n) = O(\lg n)$?

Use Priority Queue.

ADT: Priority Queue.

A collection class in which its object has been assigned a priority with the following operations:

1. createPQ():
2. destroyPQ():
3. PQIsEmpty():
4. ***PQInsert(in newItem: PQItemType) throw PQException***
5. ***PQDelete(out priorityItem: PQItemType) throw PQException***

    // delete min (or max) priority object
6. PQSize():

Remark: A PQ does not always support general delete and find operations.

Designing a Priority Queue:

Simplest Approaches:

Use a sorted or unsorted array/linked list.

Better Approach:

Use a BST.

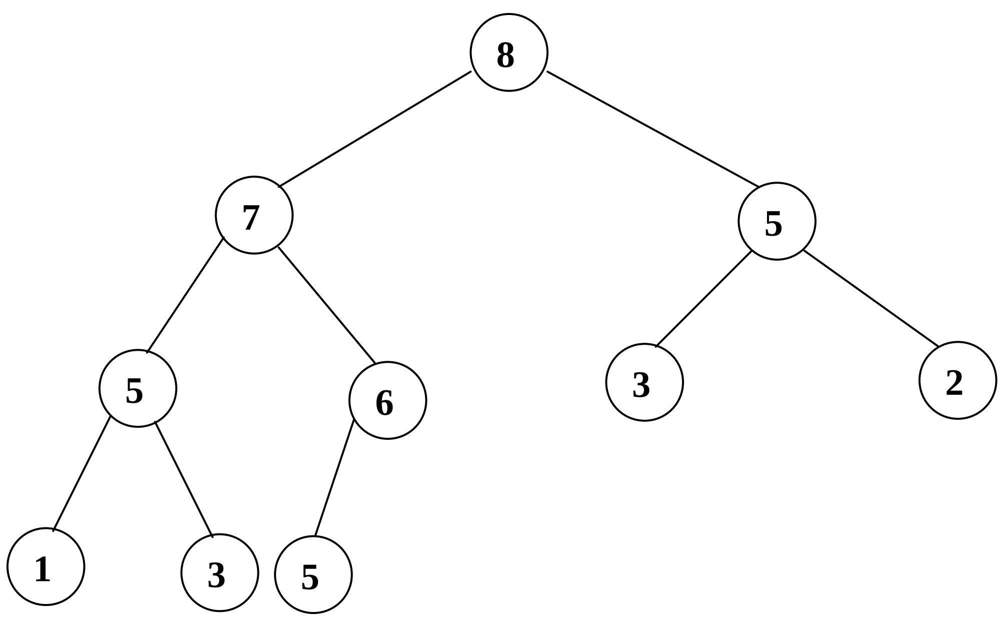
Best Approach:

Use a heap.

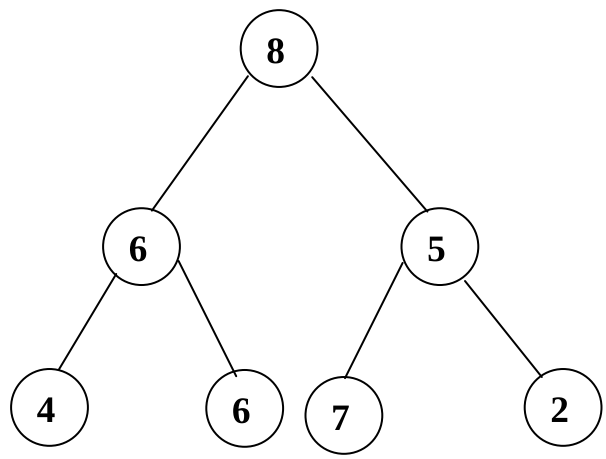
Defn: A ***max*** (binary) ***heap*** H is a complete binary tree satisfying the ***heap-ordered tree property*** such that the priority of any node $\geq$ priority of all its descendants.

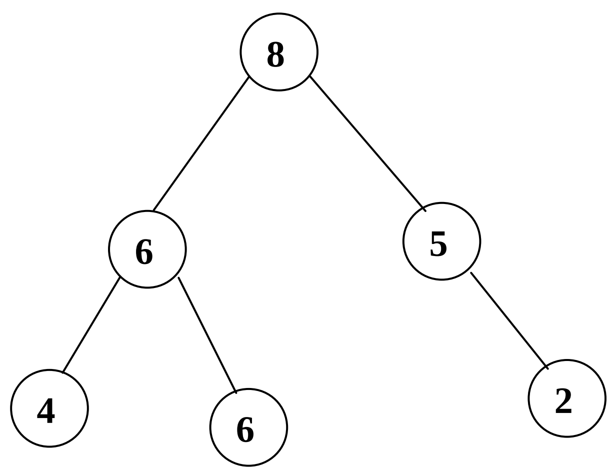
Remark: A maximum priority element occupies the root of the max heap H.

Example: A max heap H.



Example: The following binary trees are not heaps.





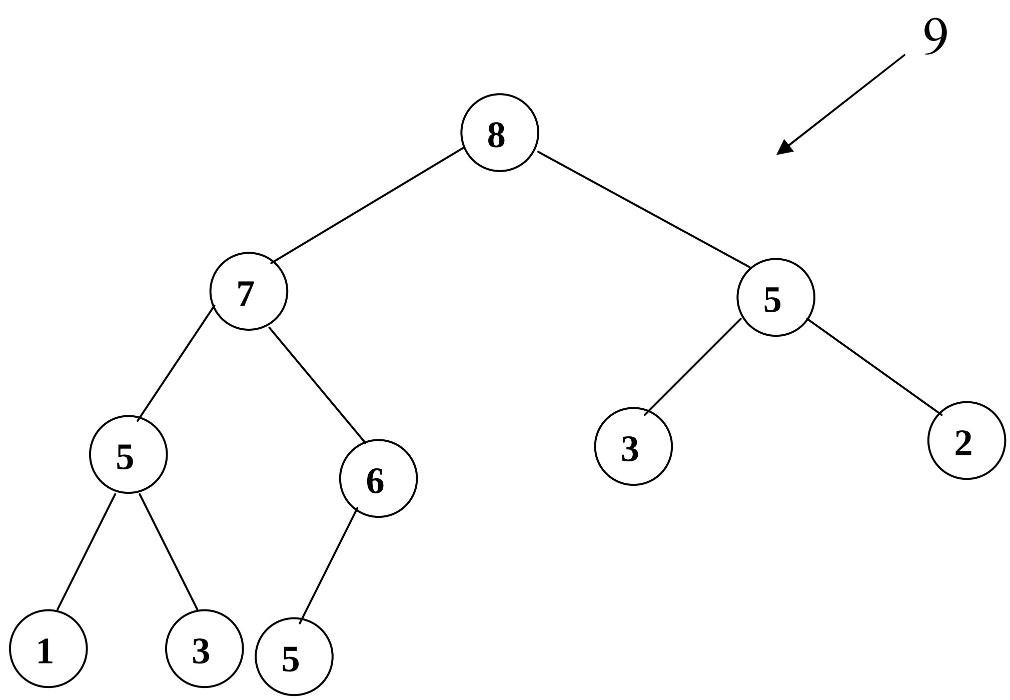
Q: Can you define a ***min heap***?

A ***min heap*** is a complete binary tree H such that the priority of any node ≤ priority of all its descendants.

Implementation of Heap:

Let's consider some typical heap operation(s) to motivate our selection of data structures for the implementation of heap.

Consider inserting 9 into the following heap H.



Observe that, after insertion, we must get back a complete binary tree satisfying the heap-ordered tree property!

Q: Where should we insert the new node in order to get back a heap?

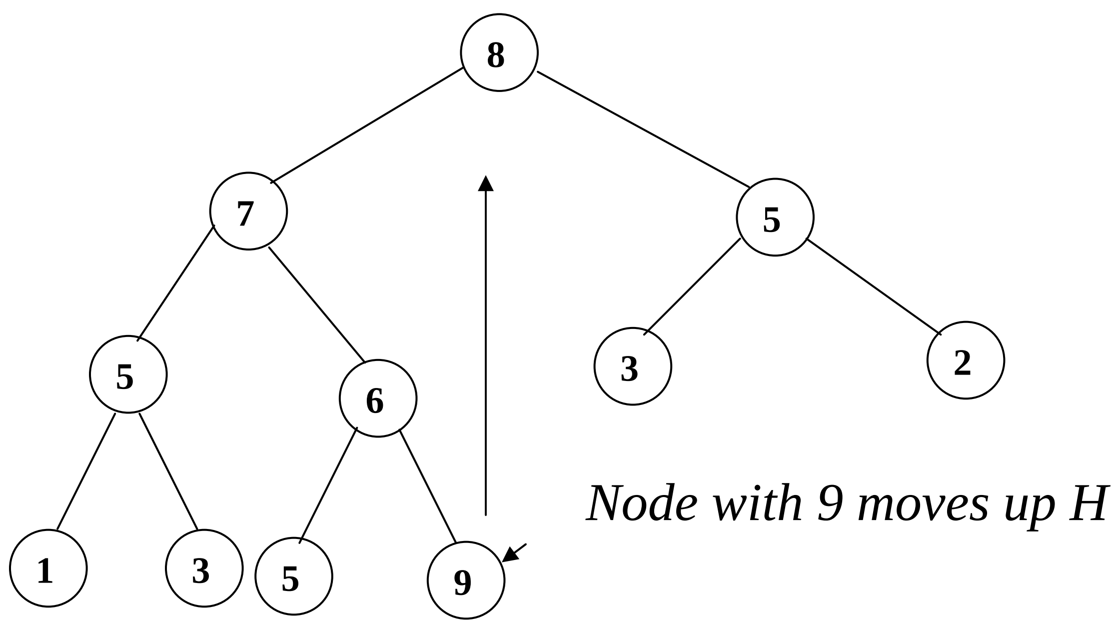
General Approach:

Always use a two-steps process:

(1) After insertion/deletion, maintain a complete binary tree structure for H.

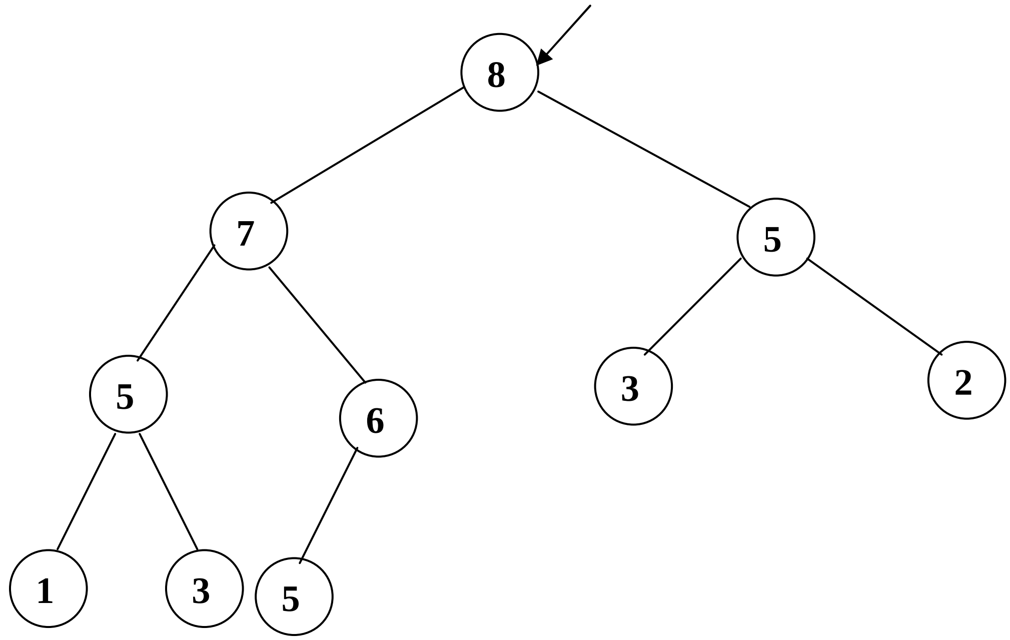
(2) Re-structure H from (1) so that it will satisfy the heap-ordered tree property.

Q: Where will 9 go in order to get back a complete binary tree?

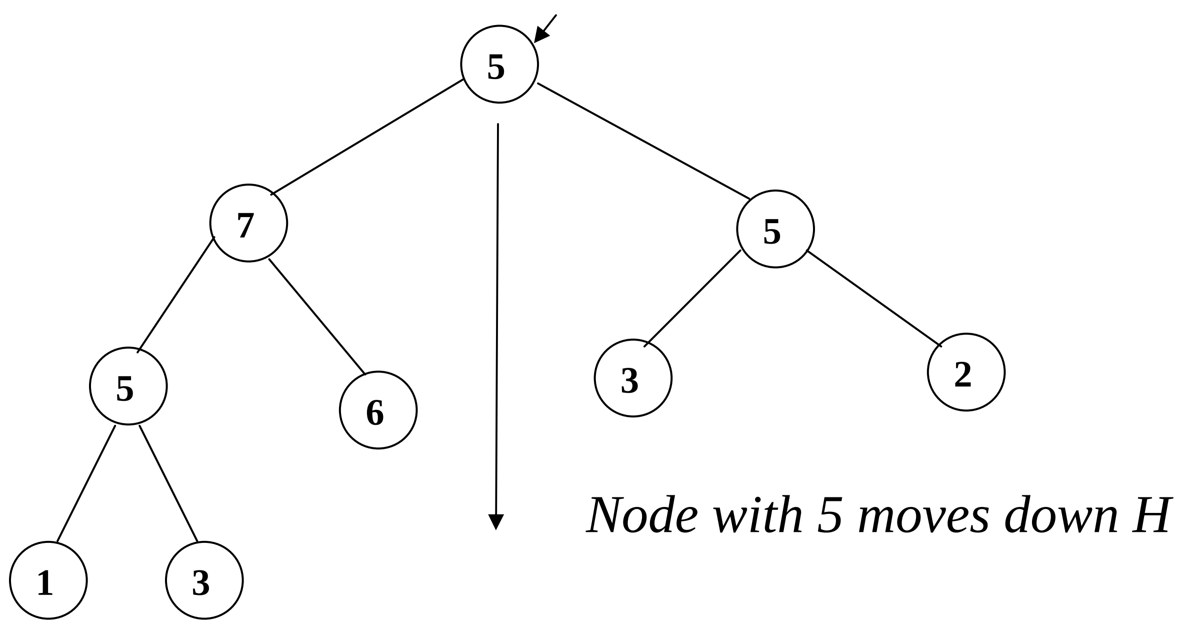


The node with key 9 must now traverse up the tree in order to get back the heap-ordered tree property.

Consider deleting max node (root) from the heap H.



Using similar two-step approach, we replace node 8 with the last node (node 5) in level-order traversal of H.



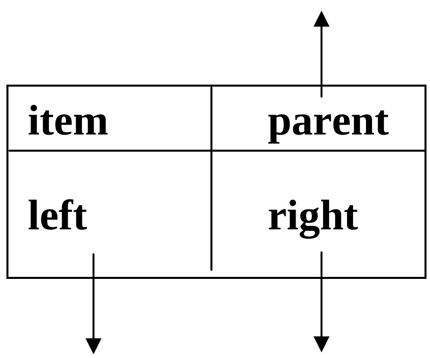
The node with key 5 must now traverse down the tree in order to get back the heap-ordered tree property.

Q: What is the most efficient data structure to implement a heap?

1. Pointer Implementation of Heap:

To facilitate the upward/downward movement in insert/delete operations, we must have parent/children information.

Consider the following node structure:



Q: Is this a good data structure?

No! This data structure is very inefficient in supporting insertion and deletion.

Why?

Remark: Do not use pointers to implement a heap!

2. Array implementation of heap:

Let H be a heap with n nodes. H can be represented using an array A[0:Max_Queue-1] such that

(1) Root of T at A[0],
(2) For any node A[i], the left child (right child) of A[i] is at A[2i+1] (A[2i+2]) if exists.

Remarks:

(1) Parent of a node A[i] is at A[(i-1)/2] if exists.
(2) For $n > 1$, A[i] is a leaf node in H iff $2i \geq n$.

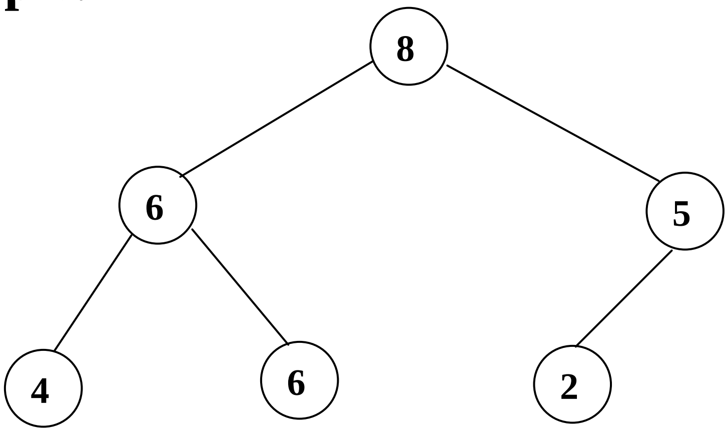
An Alternate Array Implementation:

Skip A[0] and store root of H at A[1].
Hence,

(1) Parent of node A[i] is at A[$\lfloor i/2 \rfloor$].
(2) Left child (right child) of a node A[i] is at A[2i] (A[2i+1]).
(3) A[i] is a leaf iff $2i > n$.

Example:

H:



Array representation of H using A[0]:

last

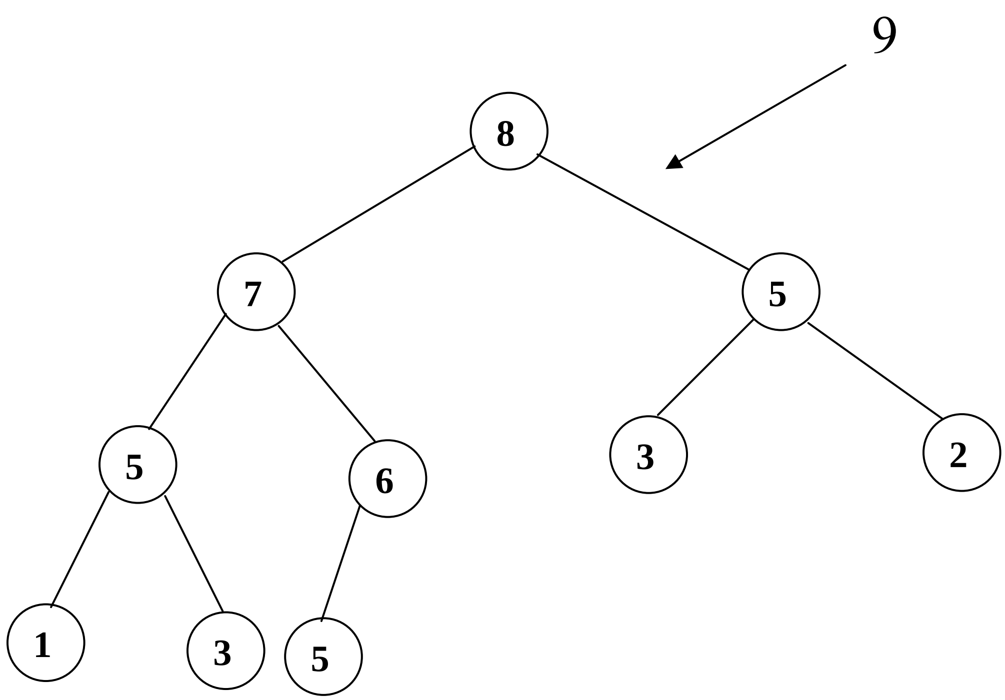
8	6	5	4	6	2	…		

Array representation of H without using A[0]:

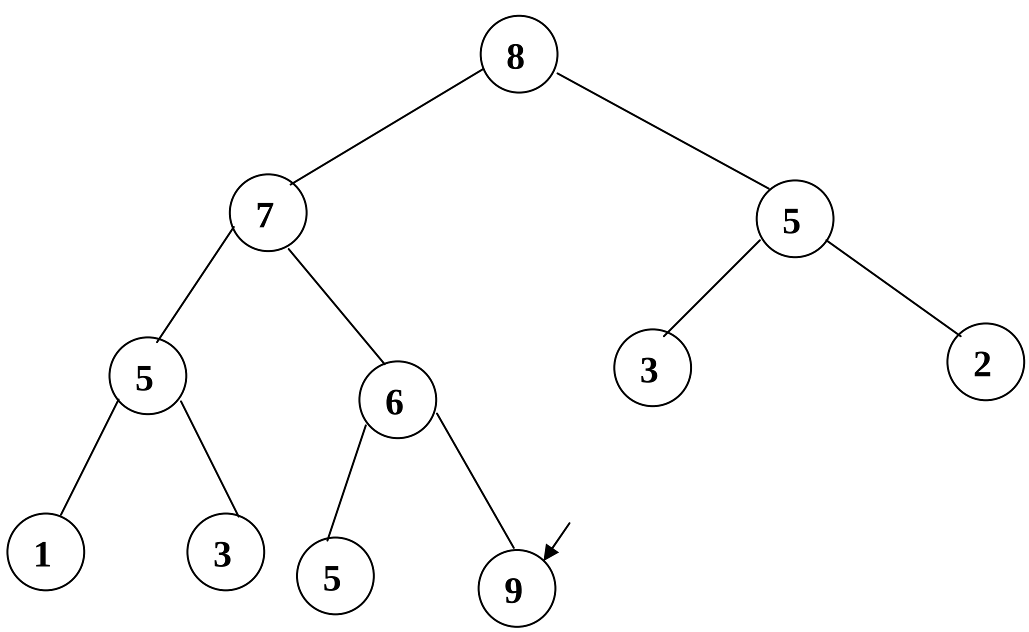
last

	8	6	5	4	6	2	…	

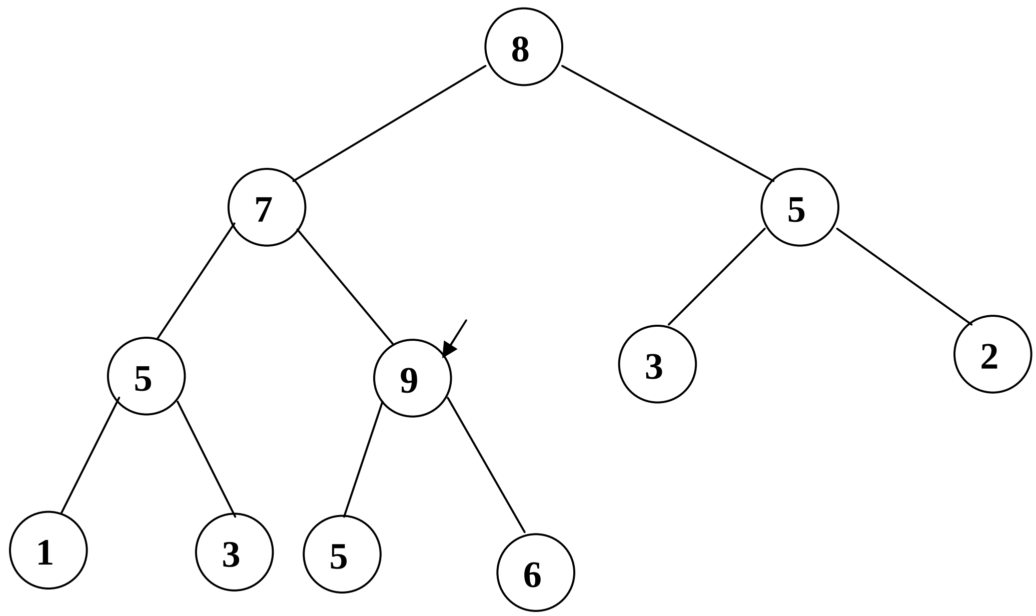
Consider inserting 9 into a max heap H.



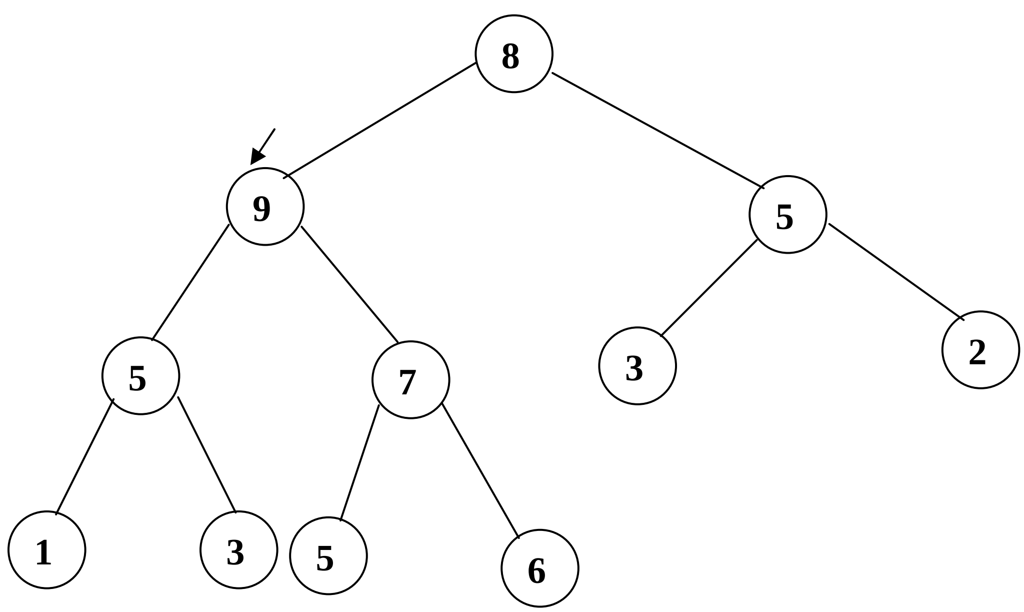
Inserting 9:



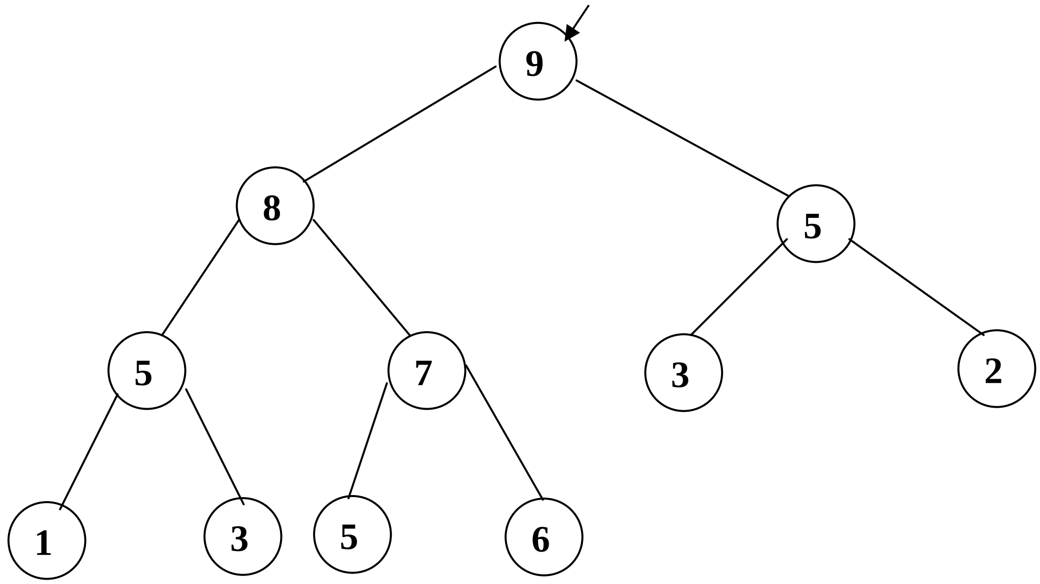
Compare 9 with its parent 6 and swap:



Compare 9 with its parent 7 and swap:



Compare 9 with its parent 8 and swap:



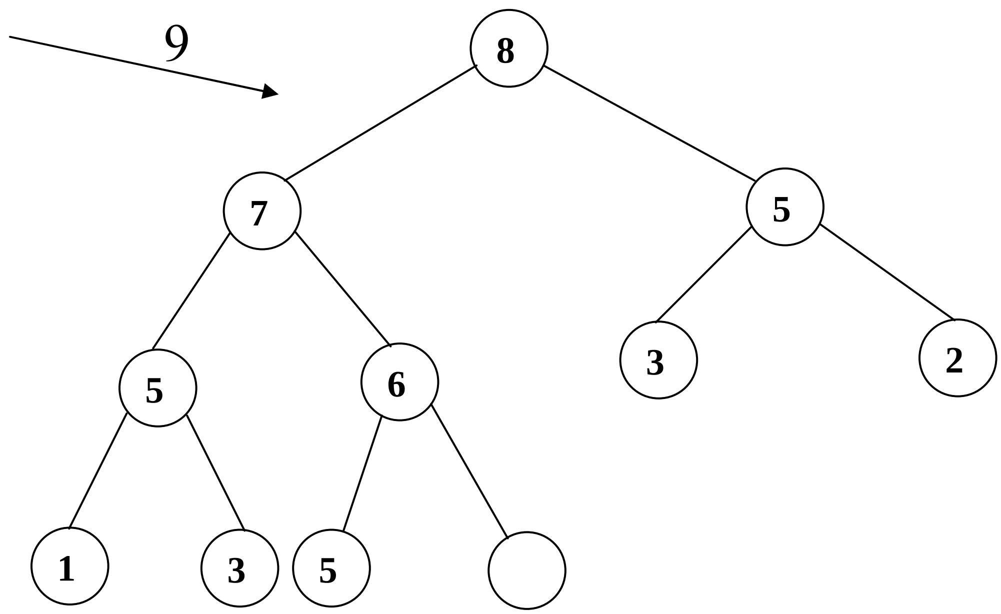
Since node 9 has no parent; process terminates.

In general, for inserting a new item X into H, we need to find a location to insert X along the path from X (after insertion) to the root of H by *repeatedly comparing X with his parent, grandparent, ..., until either a node with priority ≥ X is found or the root is reached*.

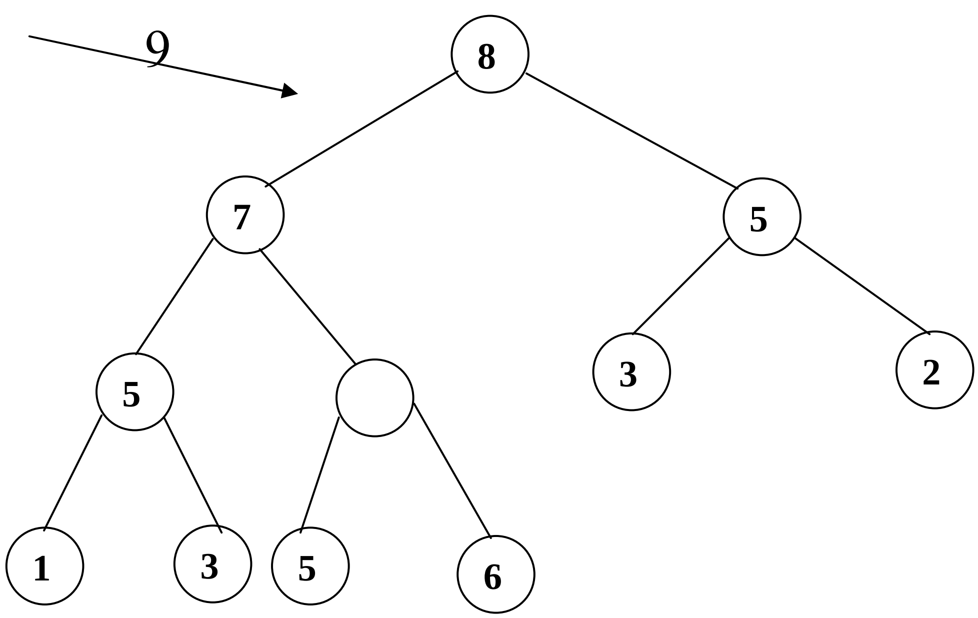
Remark: Insert (remove) operation can be somewhat speeded up if we will attempt to find the *final* location for X first instead of placing X into various locations in H only find out later that it needs to be moved to another location again.

Example: Consider inserting 9 into the heap H.

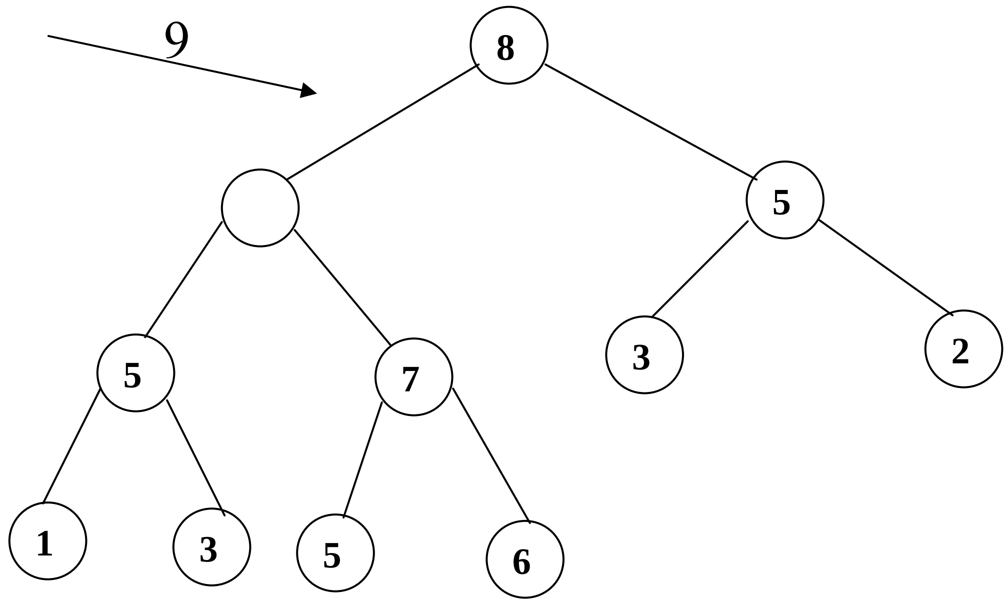
Creating new location for 9:



Compare 9 with its parent 6; 6 moves down:

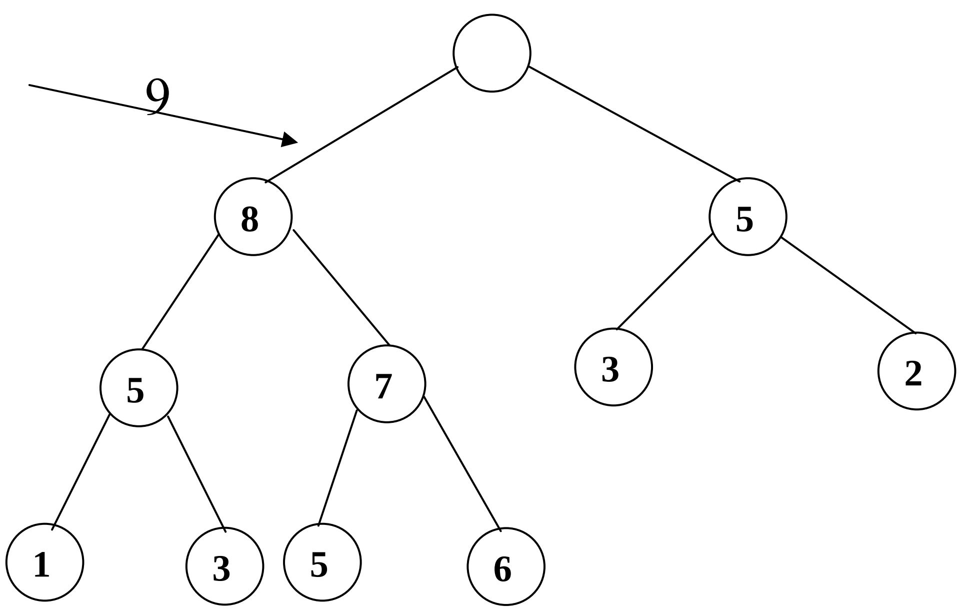


Compare 9 with its parent 7; 7 moves down:

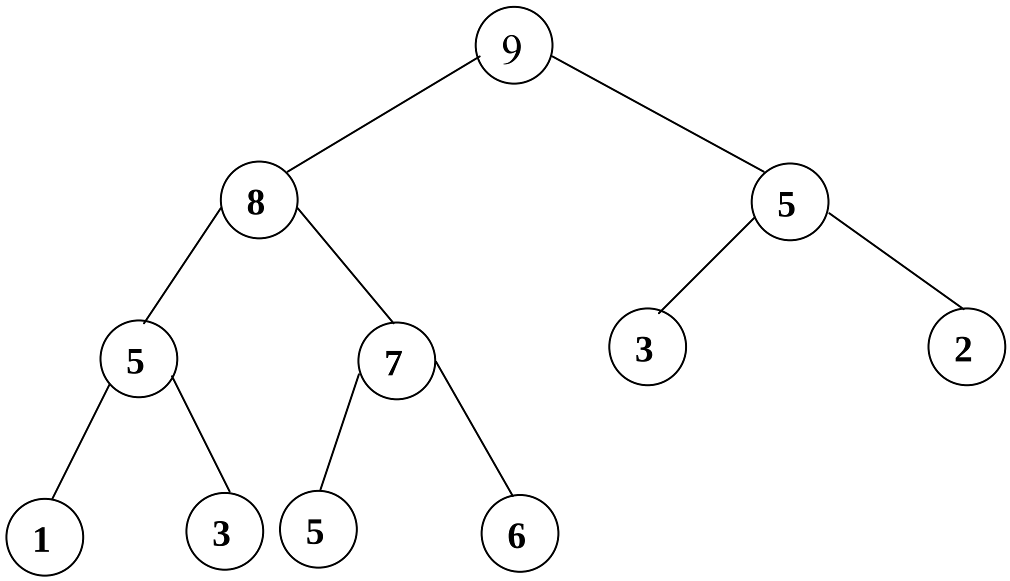


↓

Compare 9 with its parent 8; 8 moves down:



Insert 9 into its final location; process terminates:



Using Array implementation:

9

0	1	2	3	4	5	6	7	8	9	10	…			
8	7	5	5	***6***	3	2	1	3	5					

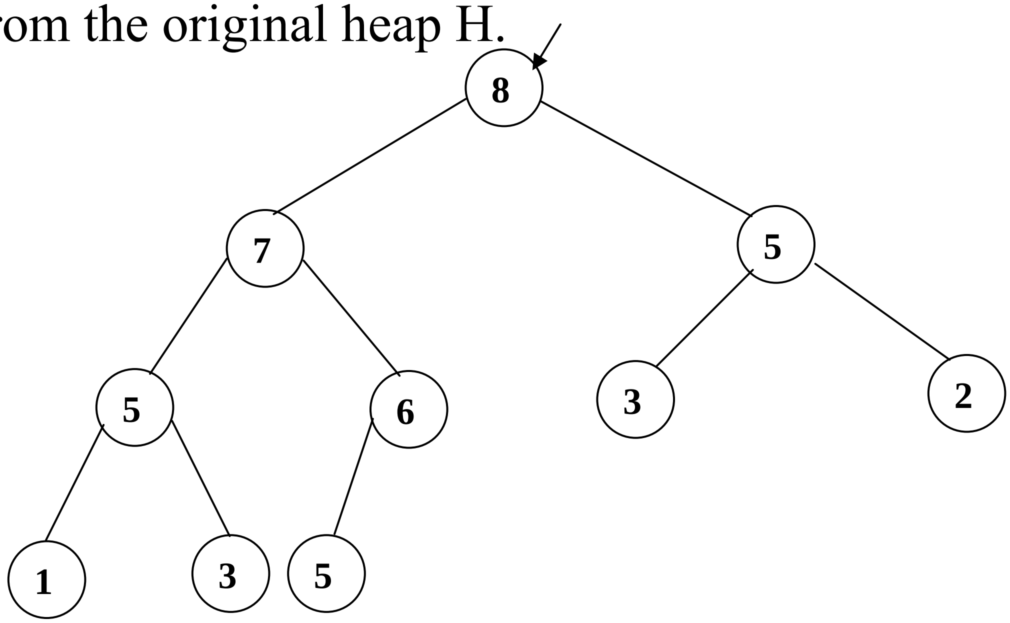
0	1	2	3	4	5	6	7	8	9	10	…			
8	***7***	5	5		3	2	1	3	5	6				

0	1	2	3	4	5	6	7	8	9	10	…			
8		5	5	7	3	2	1	3	5	6				

0	1	2	3	4	5	6	7	8	9	10	…			
	8	5	5	7	3	2	1	3	5	6				

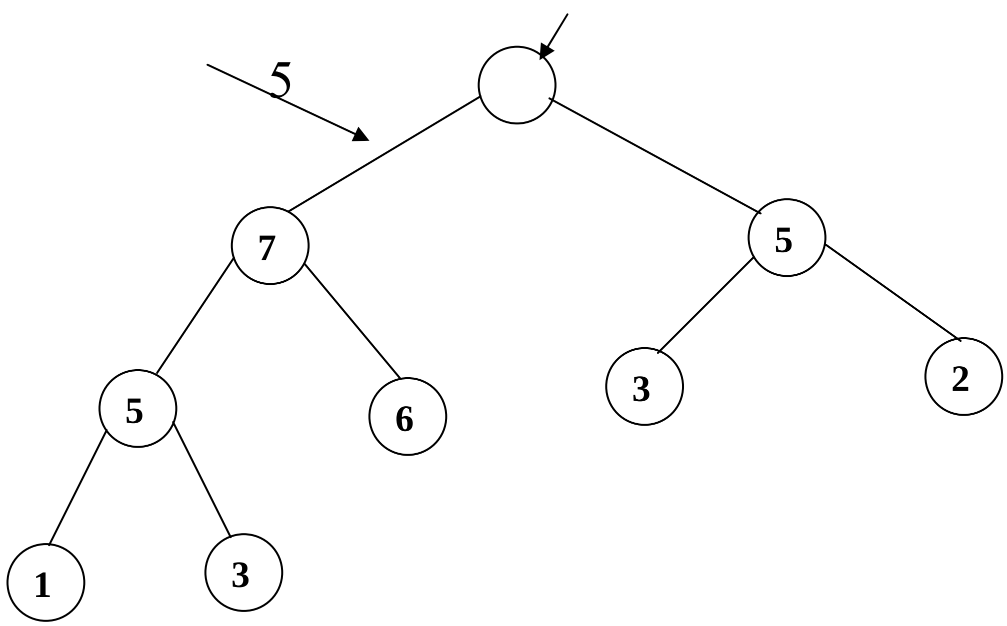
0	1	2	3	4	5	6	7	8	9	10	…			
9	8	5	5	7	3	2	1	3	5	6				

Consider deleting the highest priority item (root) from the original heap H.

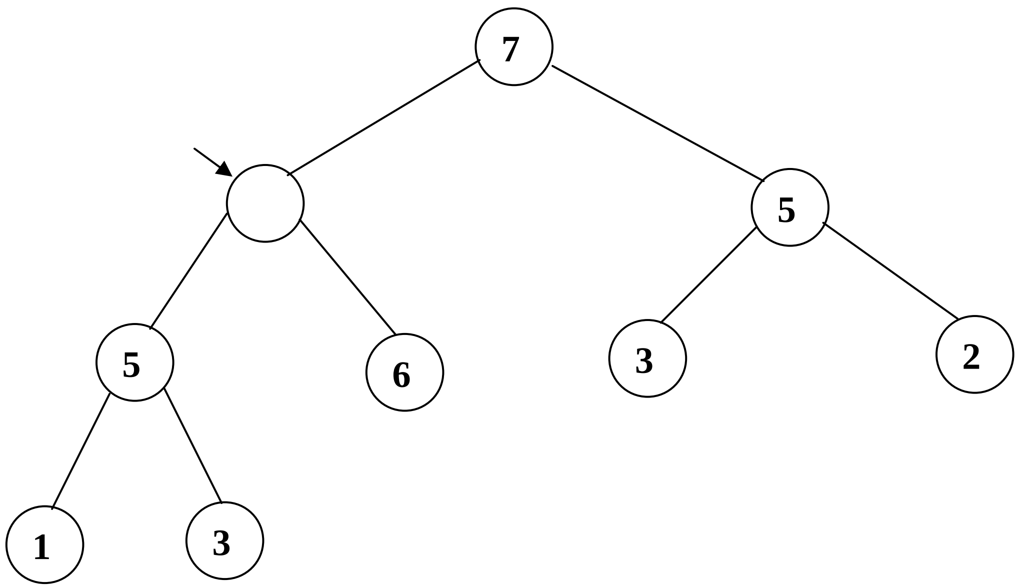


Q: What happens after 8 is removed?

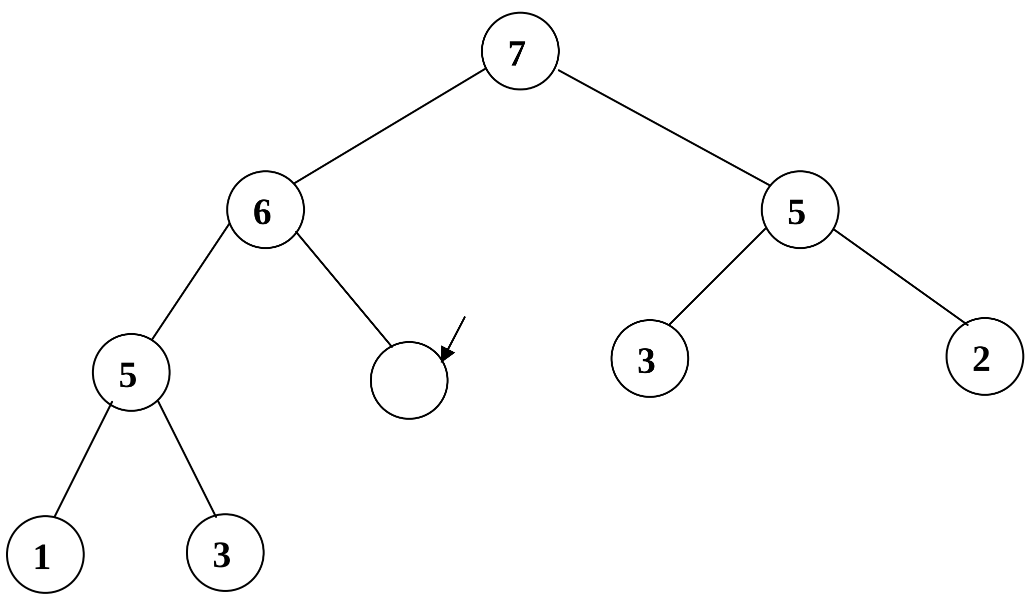
Replace the root of H with the last item (in level order) 5.



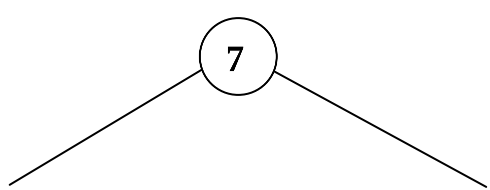
Compare 5 with its two children, swap with 7:

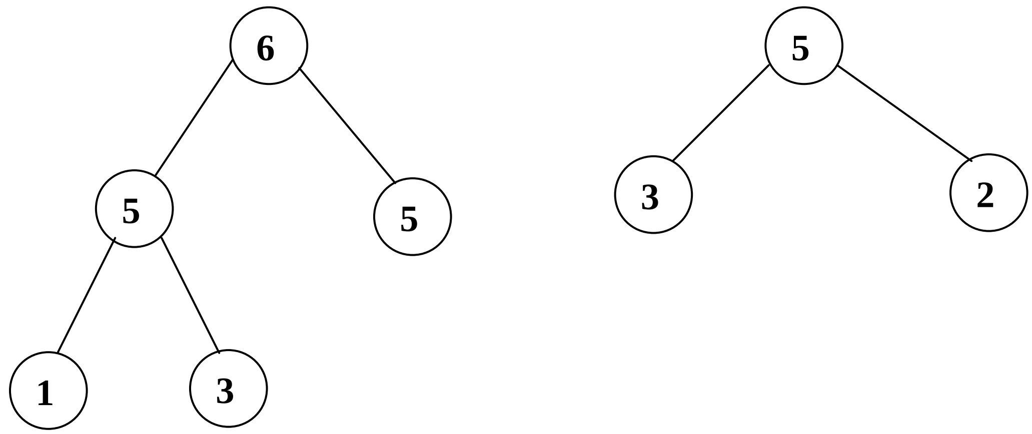


Compare 5 with its two children, swap with 6:



Insert 5 into its final location; process terminates:





In general, we replace the root (highest priority item) of H with the "last" item X in H. We then *repeatedly compare X with its child (children), swap with the larger child if necessary, until X $\geq$ its child (children if X has two children) or a leaf is reached*.

Performance analysis:
For insert/remove operations, in the worst-case, we have to either traverse all the way from a leaf to the root (for insert) or traverse all the way from the root to a leaf (for delete). Hence, the complexities of both algorithms will depend on the height of the heap. Since a heap is a complete binary tree and a complete binary tree with n nodes has height $\lfloor \lg n \rfloor + 1$, $T_w(n) = O(\lg n)$, where n is the number of items in the heap.

Array implementation:

0	1	2	3	4	5	6	7	8	9	…		
8	7	5	5	6	3	2	1	3	***5***			

0	1	2	3	4	5	6	7	8	9	…		
	7	***5***	5	6	3	2	1	3				

0	1	2	3	4	5	6	7	8	9		…	
7		5	***5***	***6***	3	2	1	3				

0	1	2	3	4	5	6	7	8	9		…	
7	6	5	***5***		3	2	1	3				

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

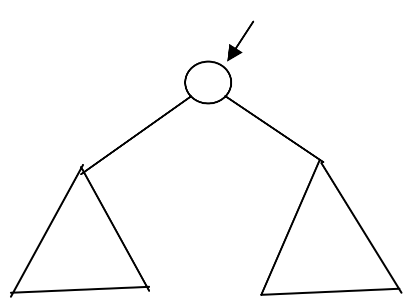
Q: How do we build the initial heap H?
Two build-heap methods:

1. **Top-down approach**: Insert items in the set, one by one, into an initially empty heap.

   $T_w(n) = O(n\lg n)$.

2. **Bottom-up approach**: First form a complete binary tree H for S according to its given order. If we scan the nodes of H in the reverse level order (leaf-to-root), observe that a leaf by itself is a heap and two heaps can be combined together with their common parent to form a bigger heap (as in delete operation), we can grow a heap for S in a bottom-up fashion.

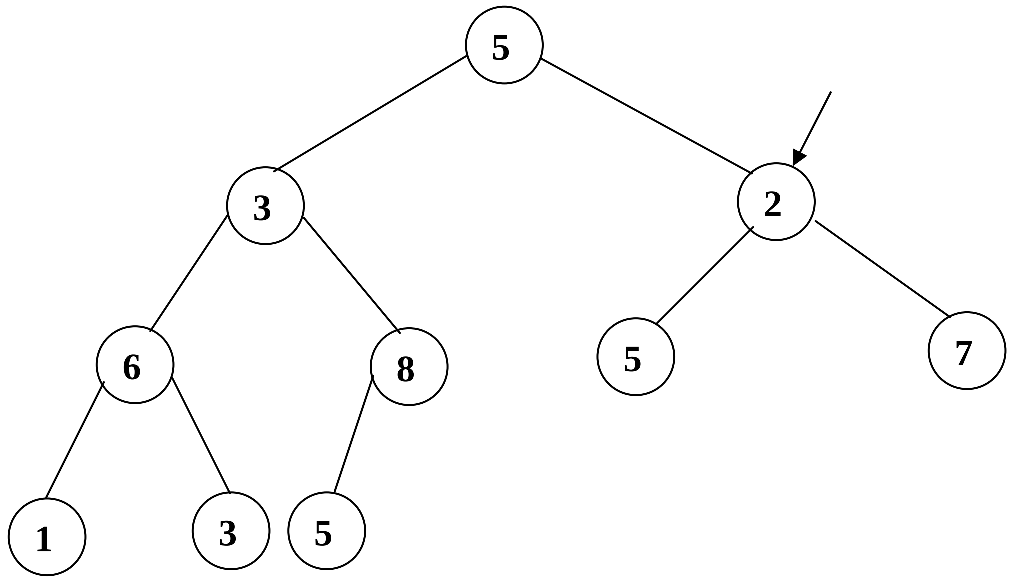
   Heapify as in delete operation by moving a root node down the tree so as to satisfy the heap-order tree property:



   $T_w(n) = O(n)$.

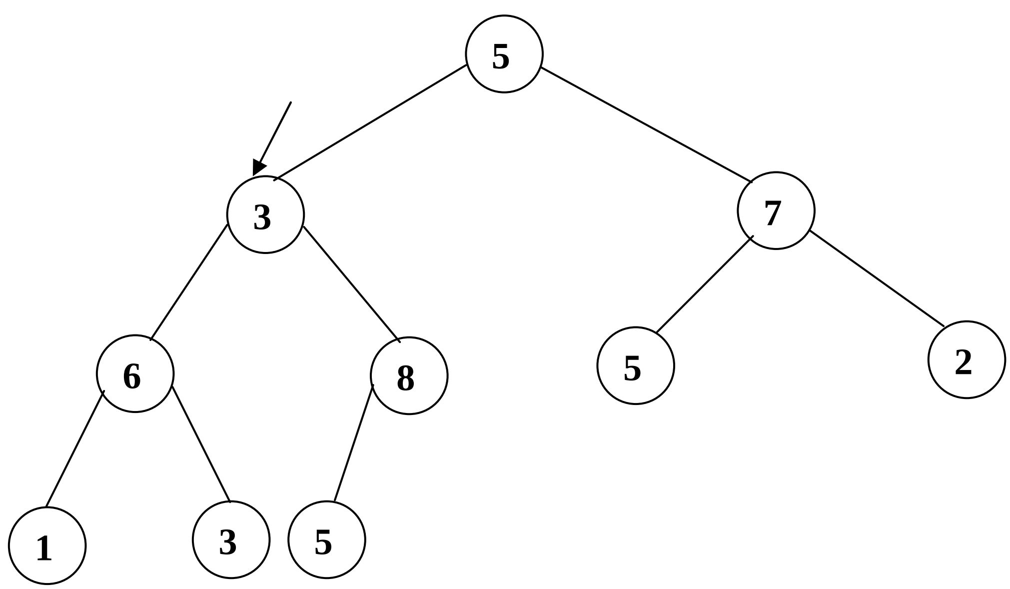
Example: Build a heap for S = {5,3,2,6,8,5,7,1,3,5}.

Form initial complete binary tree:

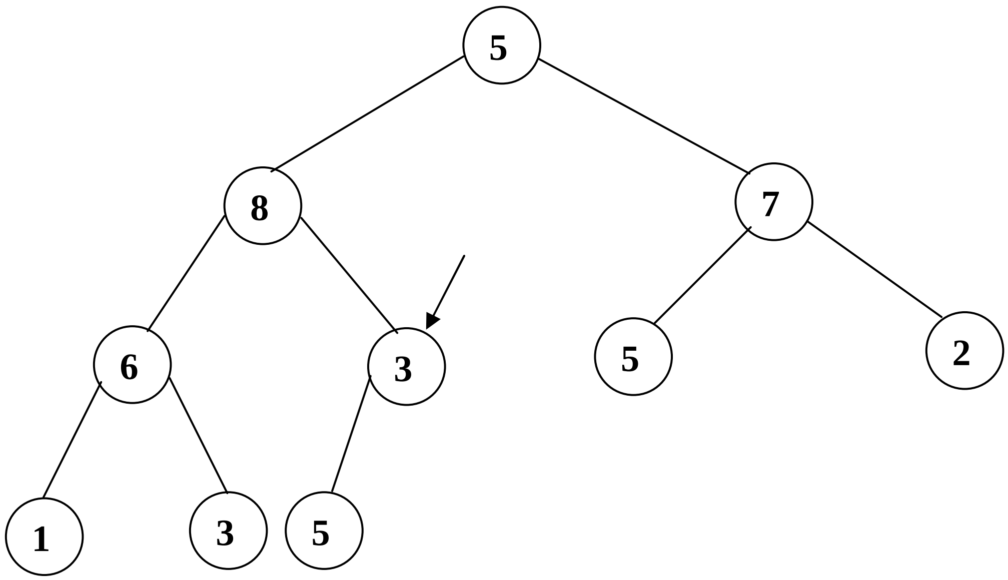


Since n = 10, first non-leaf node needs to be checked has array index $\lfloor n/2 \rfloor - 1 = 4$, follows by nodes with index 3, 2, 1, 0.

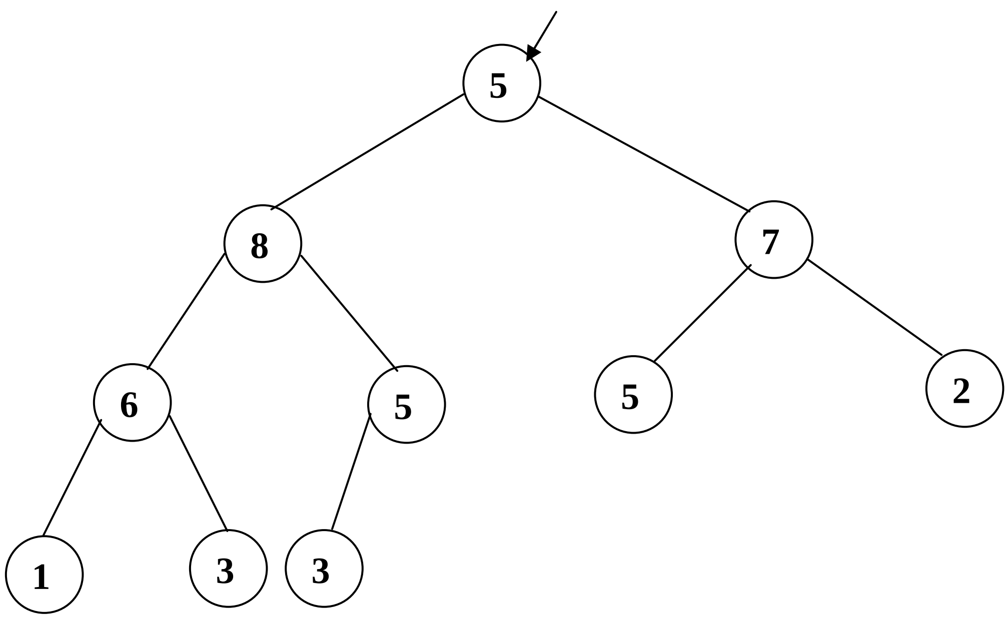
For A[2], compare 2 with 5 and 7, swap with 7:



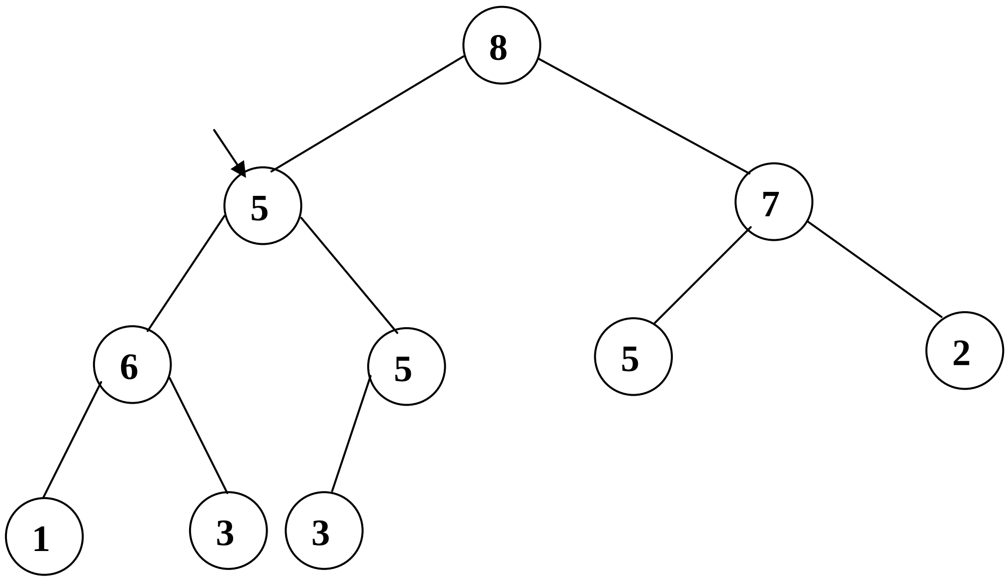
For A[1], compare 3 with 6 and 8, swap with 8:



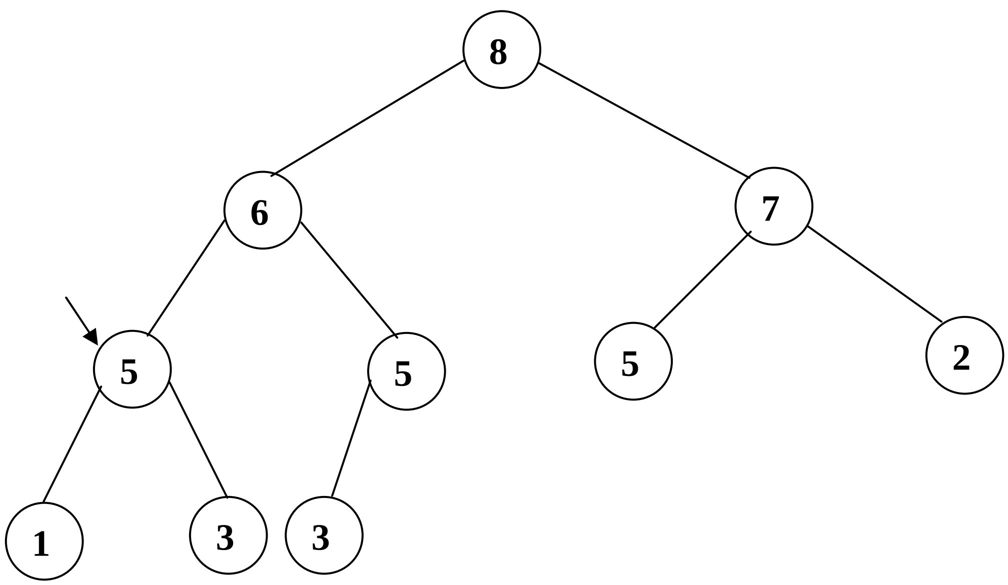
Compare 3 with 5, swap with 5:



Compare 5 with 8 and 7, swap with 8:



Compare 5 with 6 and 5, swap with 6:



Compare 5 with 1 and 3, terminates process!

HW: Repeat above constructions using array.

PQ Sorting Revisited:

1. Build a PQ Q for S.
2. Delete repeatedly until Q is empty.

Heap Sort:

1. Build a max-heap H for S.
2. Deletemax repeatedly until H is empty. (Swap max item with the current last item in array.)

Worst-Case Complexity of Heap Sort:

Building a heap: $\Theta(n)$.
DeleteMax: $\Theta(\lg n)$.

Hence, $T(n) = \Theta(n\lg n)$

Worst-Case Time Comparison of PQ Implementations:

Assuming max-heap:

PQ Operation	Heap	BST	Sorted List	Unsorted Array
Build/Organize	O(n)	O(n^2)	O(n^2)	O(n)
Insert	O(lgn)	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)	O(n)
GeneralDelete	O(n)	O(n)	O(n)	O(n)
DeleteMax	O(lgn)	O(n)	O(n)	O(n)
DeleteMin	O(n)	O(n)	O(1)	O(n)

Extension: k-heap, $k \geq 3$.

<u>2-heap</u>	**<u>k-heap</u>**
Complete binary tree	Complete k-ary tree
Heap-ordered tree	Heap-ordered tree

Implementation:

Array implementation with root at A[1].

Parent of A[i] at $A[(i+k-2)/k]$,
1^{st} child at $A[ki-k+2]$,
2^{nd} child at $A[ki-k+3]$,
3^{rd} child at $A[ki-k+4]$,
●●●
j^{th} child at $A[ki-k+j+1]$, $1 \leq j \leq k$, if exists.

Example:

Given A[i] in a 3-heap.

Parent of A[i] at $A[(i+1)/3]$,
1^{st} child at $A[3i-1]$,
2^{nd} child at $A[3i]$,
3^{rd} child at $A[3i+1]$, if exists.

HW: Compute parent-children info if rooted at A[0].