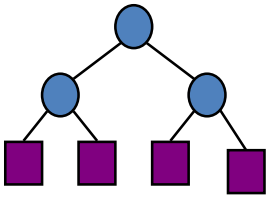
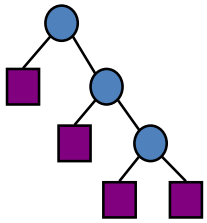
Heaps and Priority Queues

1

Quick Recap of Simple BSTs

- **Notation**: **n**: number of nodes, **e**: number of external nodes, **i**: number of internal nodes, **h**: height.





- **Properties:**
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

A **full binary tree** (sometimes proper **binary tree** or 2-**tree or a strict tree**) is a **tree** in which every node other than the leaves has two children.
A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
e = no. of edges, i = no. of nodes, h =height

2

Quick Recap of Simple BSTs

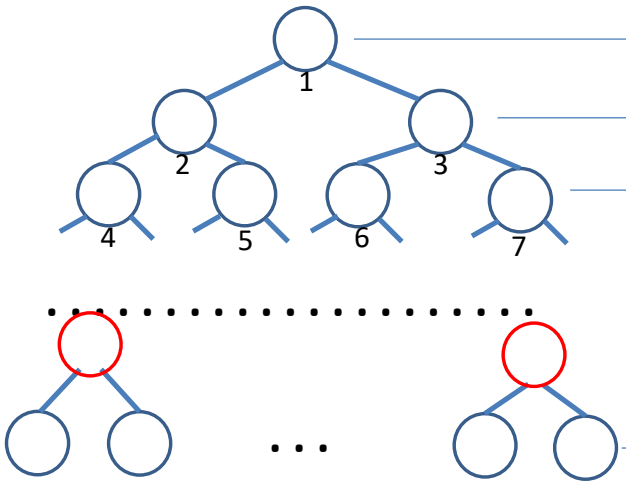
- **Traversals**:
  - Preorder: visit, left, right
  - Inorder: left, visit, right
  - Postorder: left, right, visit
- The **depth** of a node v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also be recursively defined as follows: (1) If v is the root, then the depth of v is 0; (2) otherwise, the depth of v is one plus the depth of the parent of v. The **height** of a node v is (1) 0, if v is an external node and (2) is one plus the maximum height of a child of v.
- **Creation, Insertion, deletion, Look-up, Find_max, Find_min, and successor/predecessor** (usually with respect to inorder traversal); they are all O(h) operations; in the worst case O(n).
- **Applications**: Build trees for arithmetic expressions, print/evaluate expression trees, Range Queries, Trimming a BST

Perfect Binary Trees

A **perfect binary tree** with N nodes has:

- $\lfloor \lg N \rfloor + 1$ levels
- height $\lfloor \lg N \rfloor$
- $\lceil N/2 \rceil$ leaves (half the nodes are on the last level)
- $\lfloor N/2 \rfloor$ internal nodes (half the nodes are internal)

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$



Level	Nodes per level	Sum of nodes from root up to this level	Heap height
0	2^0 (=1)	$2^1 - 1$ (=1)	n-1
1	2^1 (=2)	$2^2 - 1$ (=3)	n-2
2	2^2 (=4)	$2^3 - 1$ (=7)	n-3
...	...		
i	2^i	$2^{i+1} - 1$	n-1-i
...	...		
n-2	2^{n-2}	$2^{n-1} - 1$	1
n-1	2^{n-1}	$2^n - 1$	0

4

What is a Heap

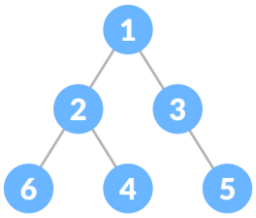
A Binary Heap is a Binary Tree with the following properties:

- It is a Complete Binary Tree. A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left. This property of Binary Heap makes them suitable to be stored in an array.
- A Binary Heap is either a Min Heap or a Max Heap.

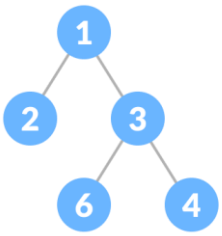
  - In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in the Binary Tree.

  - Similarly, in a Max Binary Heap, the key at the root must be maximum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree

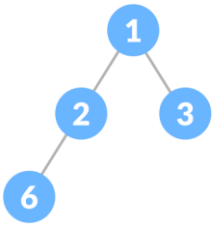
Complete vs. Full Binary Tree



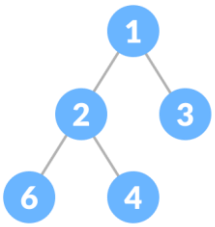
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✗ Complete Binary Tree



✗ Full Binary Tree
✓ Complete Binary Tree



✓ Full Binary Tree
✓ Complete Binary Tree

A **full** binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

A **complete** binary tree is a binary tree in which all the levels are filled except possibly the lowest one, which is filled from the left.

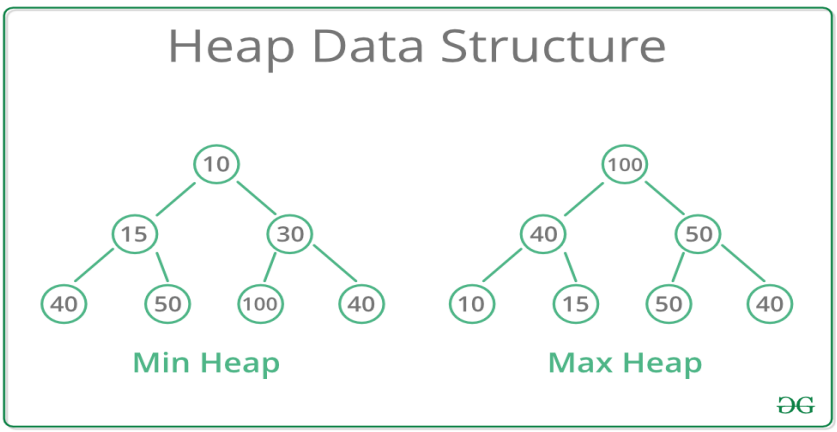
A complete binary tree is called a **perfect** binary tree if all the levels are filled.

6

6

Heap Data Structure

- A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:
  - Max-Heap: In a Max-Heap the key present at the root node must be larger than or equal to all the keys present at all its children. The same property must be recursively true for all sub-trees in that Binary Tree.
  - Min-Heap: In a Min-Heap the key present at the root node must be less than or equal to all keys present at all its children. The same property must be recursively true for all sub-trees in that Binary Tree.

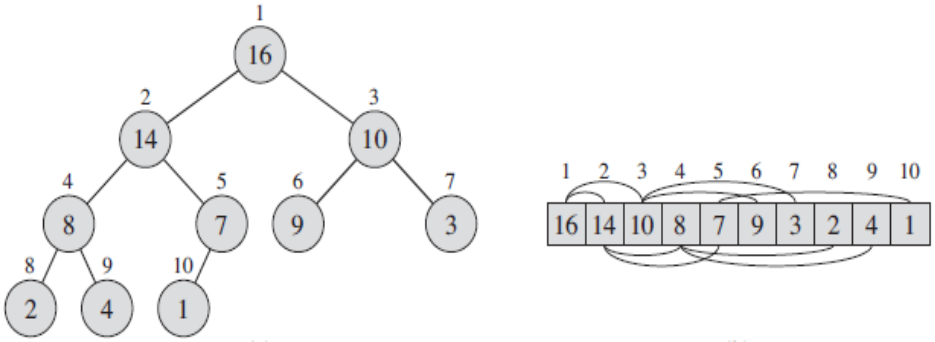


Build a mx-heap from an arbitrary array

Some Notations

- The heap data structure is useful for heapsort, but it also makes an efficient priority queue. The heap data structure is useful for many different algorithms. Note that the word "heap" also refers to "garbage-collected storage," such as the programming languages Java and Lisp provide. Our heap data structure is not garbage-collected storage.
- The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: A:length, which (as usual) gives the number of elements in the array, and A:heap-size, which represents how many elements in the heap are stored within array A.
- Although A[1 … A:length] *may* contain numbers, only the elements in A[1, … A.heap-size], where $0 \leq$ A.heap-size $\leq$ A.length, are valid elements of the heap. The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

Notations



A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has **height** three; the node at index 4 (with value 8) has height one.

PARENT(i) : return $\lfloor i/2 \rfloor$; Left(i) : return 2i; Right(i): return 2i+1

11

Observations

- As mentioned earlier, there are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap.
- In a max-heap, the max-heap property is that for every node i, **A[Parent(i)] ≥ A[i]** that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
- A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root, **A[Parent(i)] ≤ A[i]**. The smallest element in a min-heap is at the root.
- Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$. [Remember lg n denotes $\log_2 n$]

Priority Queues

- Goal – to support (efficiently) operations:
  - **Delete/remove** the max element.
  - **Insert** a new element.
  - **Initialize** (organize a given set of items).
- Useful for **online** processing
  - We do not have all the data at once (the data keeps coming or changing).

  (So far, we have seen sorting methods that work in **batch mode**: They are given all the items at once, then they sort the items, and finish.)
- Applications:
  - Scheduling:
    - flights take-off and landing, programs executed (CPU), database queries
  - Waitlists:
    - patients in a hospital (e.g. the higher the number, the more critical they are)
  - Graph algorithms (part of MST)
  - Huffman code tree: repeatedly get the 2 trees with the smallest weight.

Binary Max-Heap: Stored as Array ⇔ Viewed as Tree

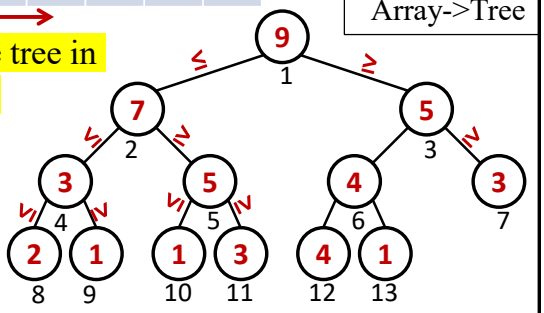
A Heap is stored as an array. Here, the first element is at index 1 (not 0).
If it starts at index 0, parent/child calculations will be: 2i+1, 2i+2, $\lfloor (i-1)/2 \rfloor$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	-	9	7	5	3	5	4	3	2	1	1	3	4	1

Practice:
Tree->Array
Array->Tree

Arrange the array data as a binary tree: Fill in the tree in level order with array data read from left to right.

Root is at index 1. (At index 0: no data or put the heap size there.)	Node at index i		
	Children		Parent
	Left 2i	Right 2i+1	$\lfloor i/2 \rfloor$



Heap properties:

P1: Order: Every node is larger than or equal to any of its children.

⇒ Max is in the root.

⇒ Any path from root to a node (and leaf) will go through nodes that have decreasing value/priority. E.g.: 9,7,5,1 (blue path), or 9,5,4,4

P2: Shape (**complete tree: "no holes")** ⇔ array storage

=> all levels are complete except for last one,

=> On last level, all nodes are to the left.



If N items =>
$h = \lfloor lgN \rfloor$
h is the height
If height h =>
$2^h \le N \le 2^{h+1}-1$

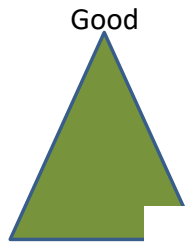
14

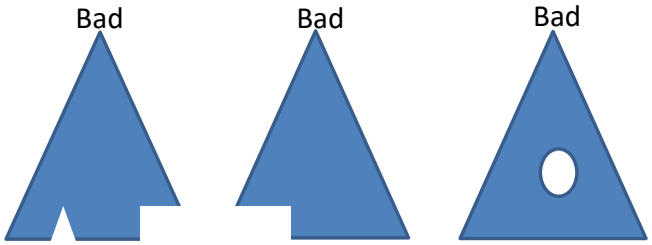
Heap – Shape Property

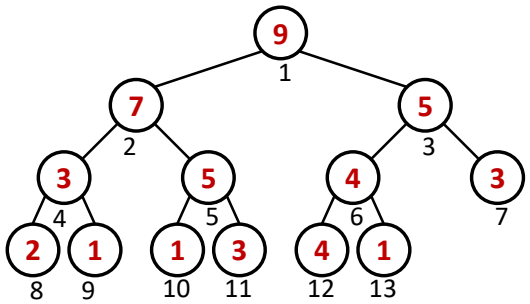
P2: Shape (**complete tree:** "no holes") ⇔ array storage

=> All levels are complete except, possibly, the last one.

=> On last level, all nodes are to the left.





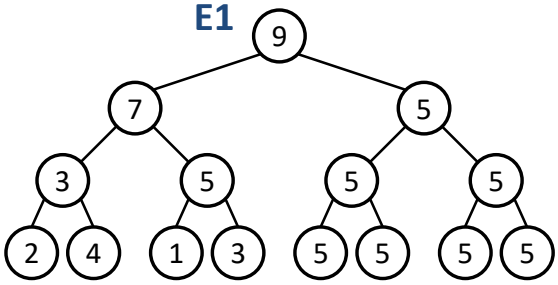


15

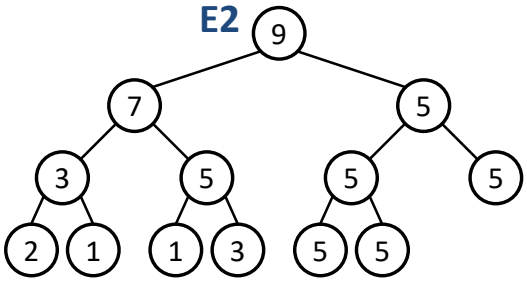
Heap Practice

For each tree, say if it is a max heap or not (Explain).

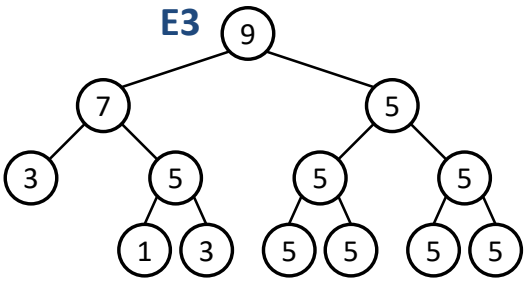
E1



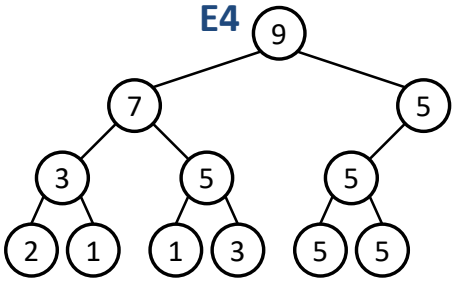
E2



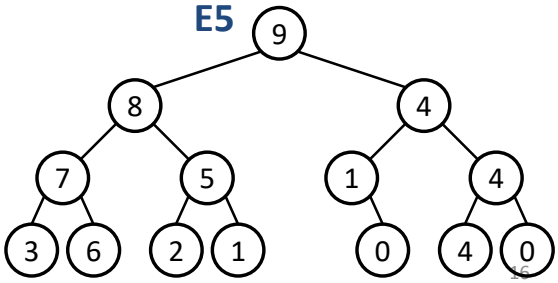
E3



E4



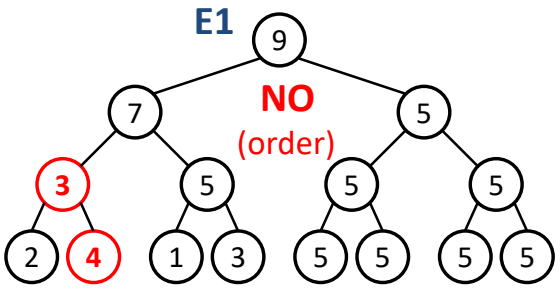
E5

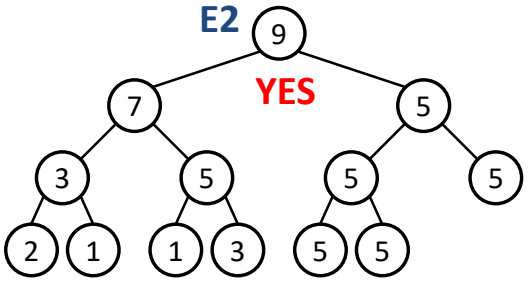


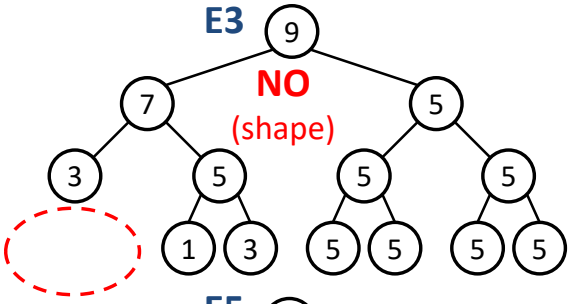
16

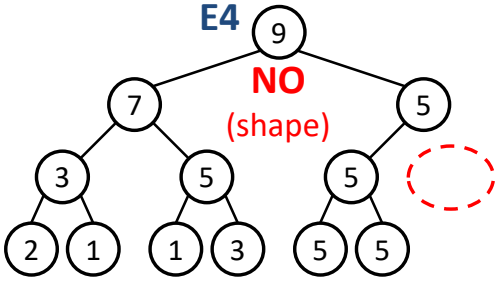
Answers

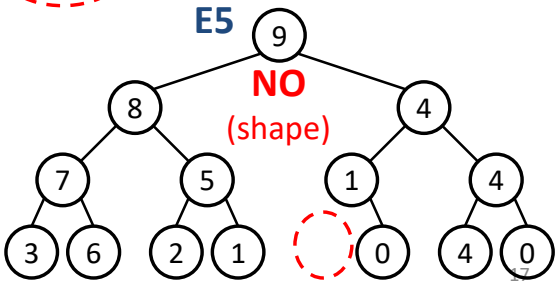
For each tree, say if it is a max heap or not.











17

Examples and Exercises

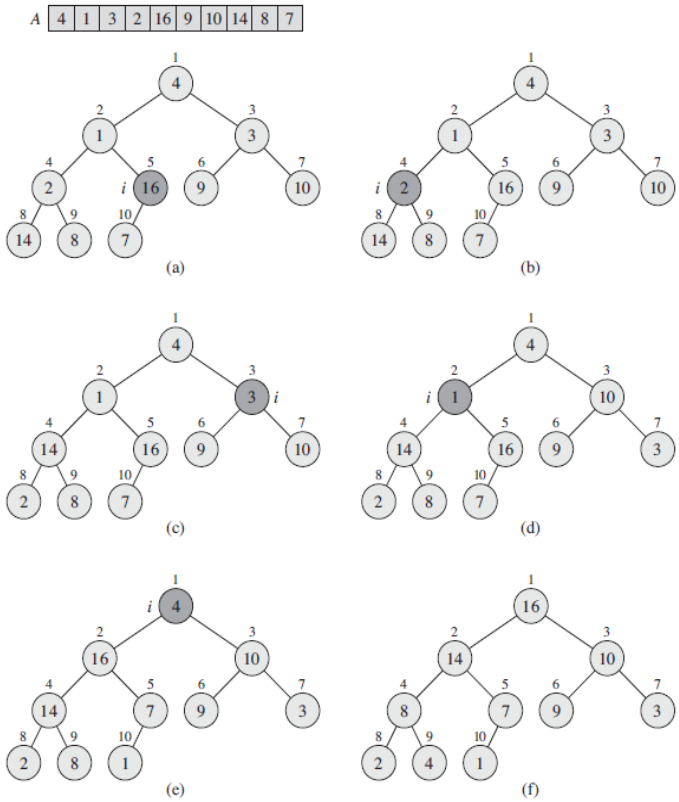
- Invalid heaps
  - Order property violated
  - Shape property violated ('tree with holes')
- Valid heaps ('special' cases)
  - Same key in node and one or both children
  - 'Extreme' heaps (all nodes in the left child are smaller than any node in the right child or vice versa)
  - **Min-heaps**
- Where can these elements be found in a Max-Heap?
  - Largest element?
  - 2-nd largest?
  - 3-rd largest?

Maintaining the heap property

- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.
- Its inputs are an array A and an index i into the array.
- When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are maxheaps, but that A[i] might be smaller than its children, thus violating the max-heap property.
- MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

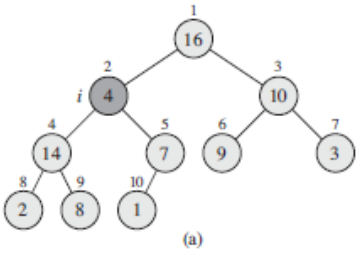

19



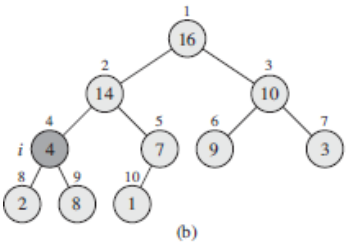
Example

The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP.
(a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i).
(b) The data structure that results. The loop index i for the next iteration refers to node 4.
(c)–(e) Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps.
(f) The max-heap after BUILD-MAX-HEAP finishes.

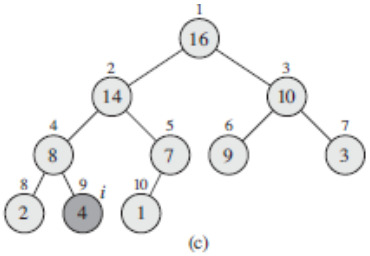
Example + Runtime



(a)



(b)



(c)

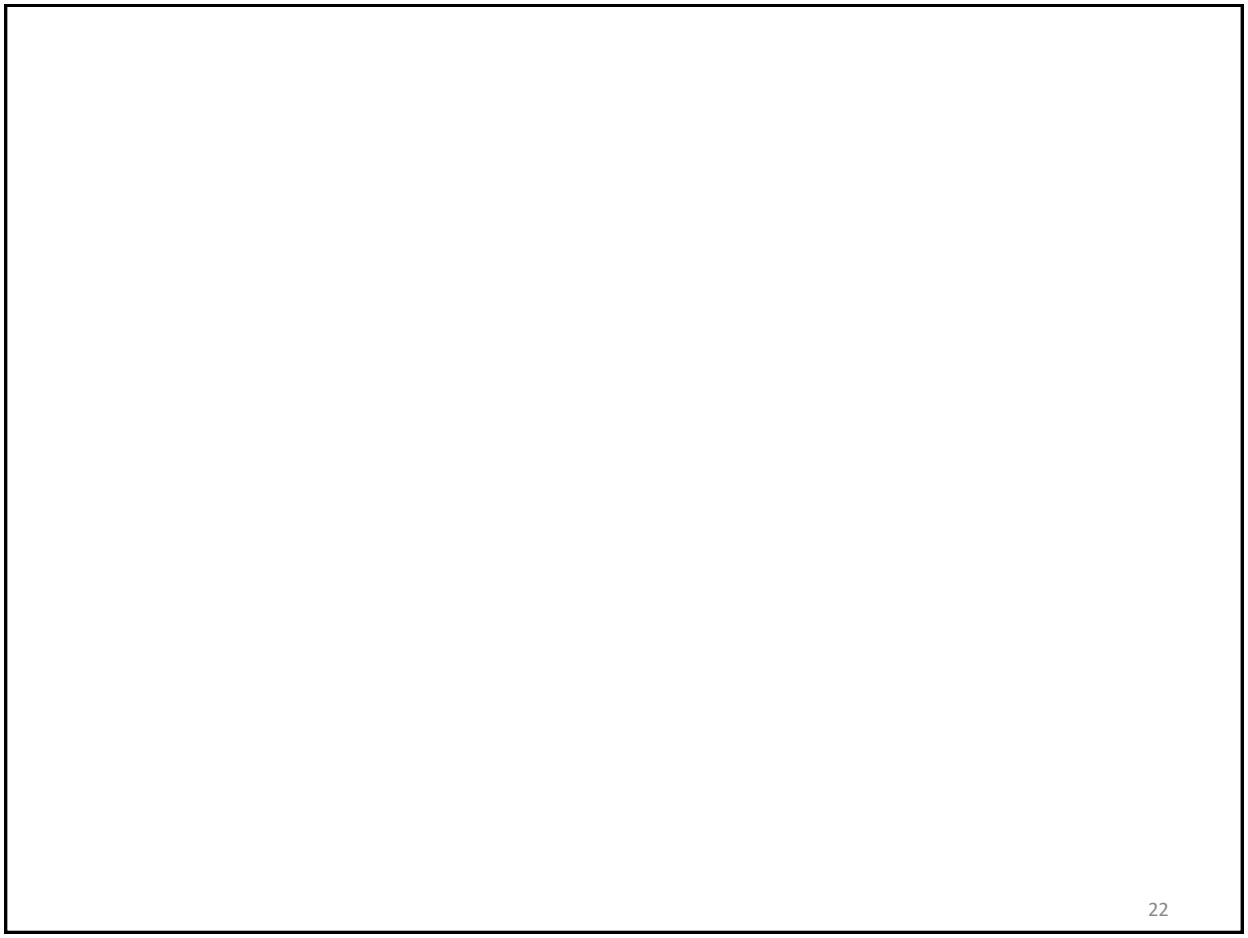
A	1	2	3	4	5	6	7	8	9	10
heap	16	4	10	14	7	9	3	2	8	1

Runtime: The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements A[left(i)] and A[right(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs).

The children's subtrees each have size at most 2n/3 – the worst case occurs when the bottom level of the tree is exactly half full WHY? – and therefore we can describe the running time of MAX-HEAPIFY by the recurrence $T(n) \leq T(2n/3) + \Theta(1)$ which will have a solution of $T(n) = O(\log_2 n)$.

21

21



22

Operations: Heap-Based Priority Queues

1. insert (A, key, N)– Inserts key in A. [A is the array; N is the size of heap]
2. removeMax (A, N) or delete(A, &N)
   - – Removes and returns the element of A with the largest key.
3. removeAny (A, p, N)
   - – Removes and returns the element of A at index p.
4. increaseKey (A, p, k, N)
   - – Changes p's key to be k. Assumes p's key was initially lower than k. Apply swimUp
5. decreaseKey (A, p, k, N)
   - – Changes p's key to be k. Assumes p's key was initially higher than k.
     - Decrease the priority and apply sinkDown.

Note: A[1 …L] is the array, p is a given index, k is a given integer, heap-size = N, assume N < L

Increase Key

(increase priority of an item)

swimUp to fix it

Example: E changes to a V.

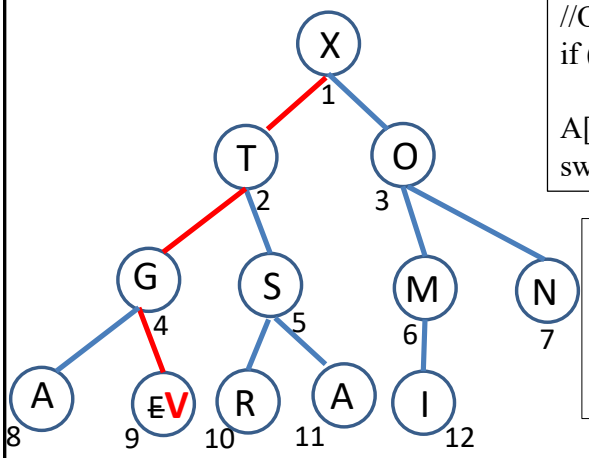
- Can lead to violation of the heap property.

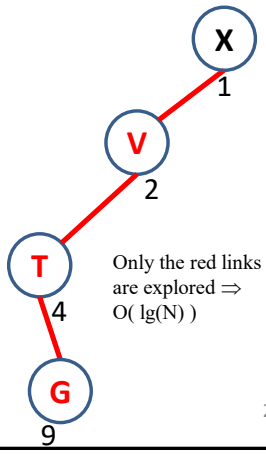
swimUp to fix the heap:

- While last modified node is not the root AND it has priority larger than its parent, swap it with his parent.
  - V not root and V>G? Yes => Exchange V and G.
  - V not root and V>T? Yes => Exchange V and T.
  - V not root and V>X? No. => STOP

```
increaseKey(A,i,newKey)
//O(lg(N))
if (A[i]>newKey)
    Not an increase. Exit.
A[i]=newKey
swimUp(A,i)
```

```
swimUp(A,i)  //O(lg(N))
while
((i>1)&&(A[i]>A[i/2])){
   swap: A[i] <-> A[i/2]
   i = i/2  }
```





Only the red links are explored ⇒ O(lg(N))

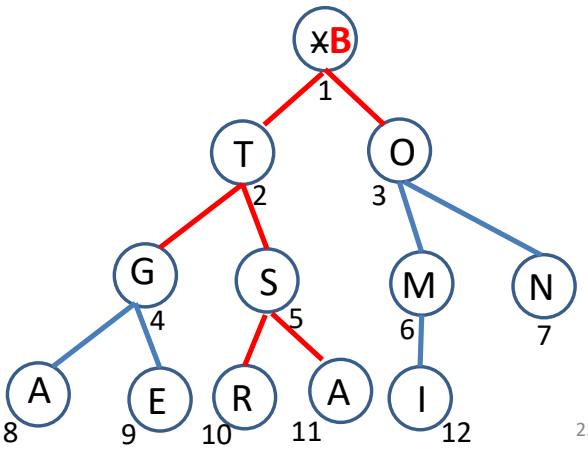
sinkDown(A,p,N)

Decrease key

(Max-Heapify/fix-down/float-down)

- Makes the tree rooted at p be a heap.
  - Assumes the left and the right subtrees are heaps.
  - Also used to restore the heap when the key, from position p, decreased.
- How:
  - Repeatedly exchange items as needed, between a node and his **largest** child, starting at p.
- e.g.: X was a B (or decreased to B).
- B will move down until in a good position.
  - T>O && T>B => T <-> B
  - S>G && S>B => S <-> B
  - R>A && R>B => R <-> B
  - No left or right children => stop

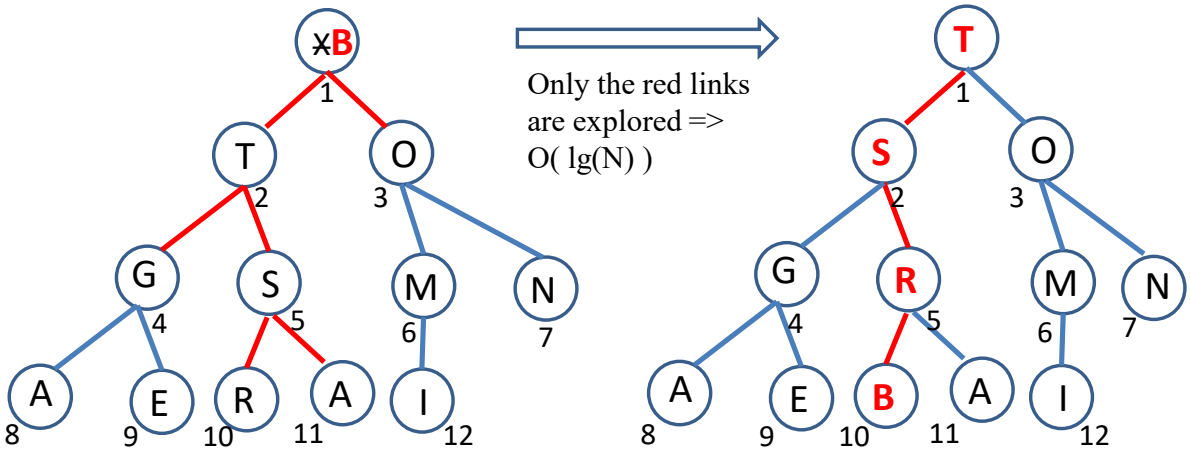
```
sinkDown(A,p,N)     - O(lgN)
left = 2*p          // index of left child of p
right = (2*p)+1 // index of right child of p
index=p
if (left≤N)&&(A[left]>A[index])
      index = left
if (right≤N)&&(A[right]>A[index])
      index = right
if (index!=p)  {
      swap A[p] <-> A[index]
      sinkDown(A,index,N)  }
```



25

25

Decrease key
sinkDown(A,p,N)



Applications/Usage:

- Priority changed due to data update (e.g. patient feels better)
- Fixing the heap after a delete operation (removeMax)
- One of the cases for removing a non-root node
- Main operation used for building a heap BottomUp.

Insert a New Record in a Heap

⌊3/2⌋ ⌊6/2⌋ ⌊13/2⌋

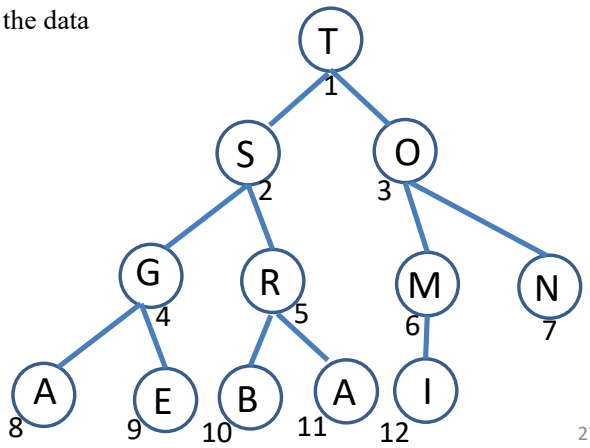
index	1	2	3	4	5	6	7	8	9	10	11	12	
Original	T	S	O	G	R	M	N	A	E	B	A	I	

Insert V in this heap. This is a heap with 12 items.

- Where can the new node be? (do not worry about the data in the nodes for now)

Time complexity? Best: Worst: General:

```
insert(A,newKey,&N)
(*N) = (*N)+1 // permanent change
//same as increaseKey:
i = (*N)
A[i] = newKey
while ((i>1)&&(A[i]>A[i/2])) {
   swap: A[i] <-> A[i/2]
   i = i/2 }
```



27

Inserting a New Record

⌊3/2⌋ ⌊6/2⌋ ⌊13/2⌋

index	1	2	3	4	5	6	7	8	9	10	11	12	13
Original	T	S	O	G	R	M	N	A	E	B	A	I	
Increase and Put V	T	S	O	G	R	M	N	A	E	B	A	I	V
1st iter	T	S	O	G	R	V	N	A	E	B	A	I	M
2nd iter	T	S	V	G	R	O	N	A	E	B	A	I	M
3rd iter,Final	V	S	T	G	R	O	N	A	E	B	A	I	M

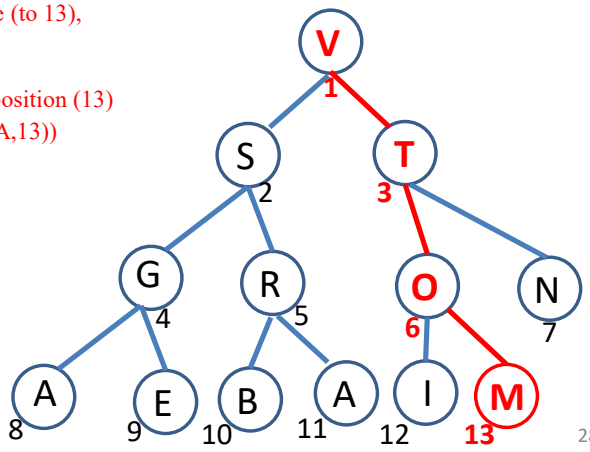
```
insert(A,newKey,&N)
(*N) = (*N)+1 // permanent change
//same as increaseKey:
i = (*N)
A[i] = newKey
while ((i>1)&&(A[i]>A[i/2])) {
   swap: A[i] <-> A[i/2]
   i = i/2  }
```

Increase heap size (to 13),

Put V in the last position (13)

Fix up (swimUp(A,13))

Case	Discussion	Time complexity	Example
Best	1	Θ(1)	V was B
Worst	Height of heap	Θ(lgN)	Shown here
General		***O(lgN)***	



Remove the Maximum

N is 12

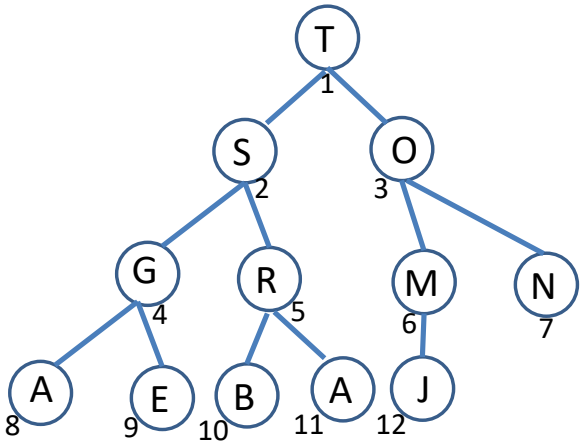
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

This is a heap with 12 items.

How will a **heap with 11 items look like**?

- What node will disappear? Think about the nodes, not the data in them.

Where is the record with the highest key?



29

Remove Maximum

N is 12

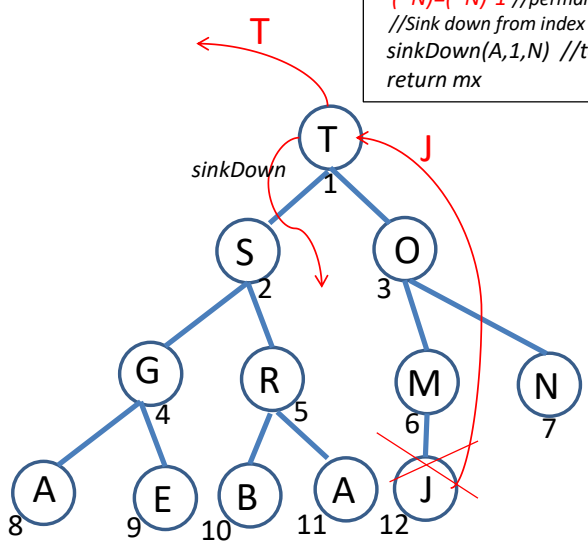
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

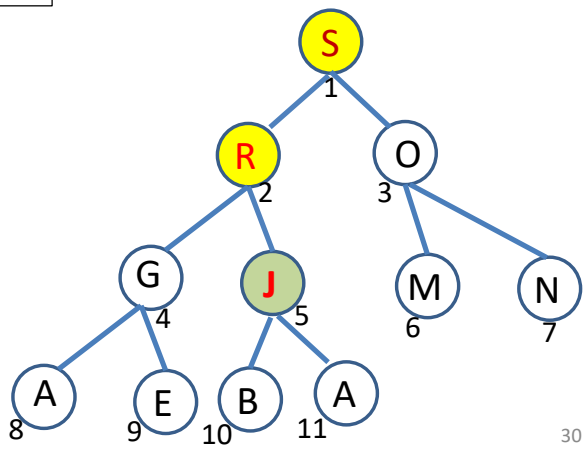
N is 11

index	1	2	3	4	5	6	7	8	9	10	11
Copy J	**J**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**
sinkDown	**S**	**R**	**O**	**G**	**J**	**M**	**N**	**A**	**E**	**B**	**A**

```
removeMax(A,&N)  // Θ(lgN)
  mx = A[1]
  A[1] = A[(*N)]
  (*N)=(*N)-1 //permanent
  //Sink down from index 1
  sinkDown(A,1,N)  //to do
  return mx
```

Case	Discussion	Time Complexity	Example
Best	1	Θ(1)	All items have the same value
Worst	Height of heap	Θ(lgN)	Content of last node was A
General	1<=…<=lgN	***O(lgN)***	

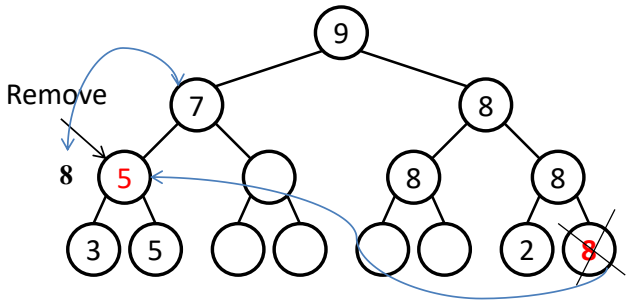




30

30

Removal of a Non-Root Node



Give examples where new priority is:
- Increased
- Decreased

removeAny(A,p,&N) // Θ(lgN)

```
temp = A[p]
A[p] = A[(*N)]
(*N)=(*N)-1 //permanent
//Fix H at index p
if (A[p]>A[p/2])
        swimUp (A,p)
else if (A[p]<temp)
        sinkDown(A,p,N)
return temp
```

Insertions and Deletions - Summary

- Insertion:
  - Insert the item to the end of the heap.
  - Fix up to restore the heap property.
  - Time = O(lg N)
- Deletion:
  - Will always delete the maximum element. This element is always at the top of the heap (the first element of the heap).
  - Deletion of the maximum element:
    - Exchange the first and last elements of the heap.
    - Decrement heap size.
    - Fix down to restore the heap property.
    - Return A[heap_size+1] (the original maximum element).
    - Time = O(lg N)
- See https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c for some applications of heaps (Not required in this course)

32

Build a heap from an arbitrary array [Build-Max_Heap]

- Consider an arbitrary array A[1…n]. We have seen that the height of a heap of n nodes is O(lg n); also, the operation max-heapify on any node x takes O(h(x)) steps.
- The height 'h' increases as we move upwards along the tree. Hence, Heapify takes different time for each node x, which is O(h(x)).
- If an arbitrary array A[1…n] is viewed as a heap, we observe that none of the leaf nodes need be considered to heapify an array; they are already heapified by definition. And there are at most $\lfloor (heapsize/2) \rfloor$ non-leaf nodes. Consider the following algorithm:

```
BUILD-HEAP(A)
   heapsize := size(A);
   for i := ⌊(heapsize/2)⌋ downto 1
      do HEAPIFY(A, i);
   end for END
```

Build a heap from an arbitrary array

- To compute the Time Complexity of building a heap, we must know the number of nodes having height h. For this we use the fact that, A heap of size n has at most $\lceil n / 2^{h+1} \rceil$ nodes with height h.
- Now to derive the time complexity, we express the total cost of **Build-Heap** as

$$T(n) = \sum_{h=0}^{\lg(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h) = O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})$$

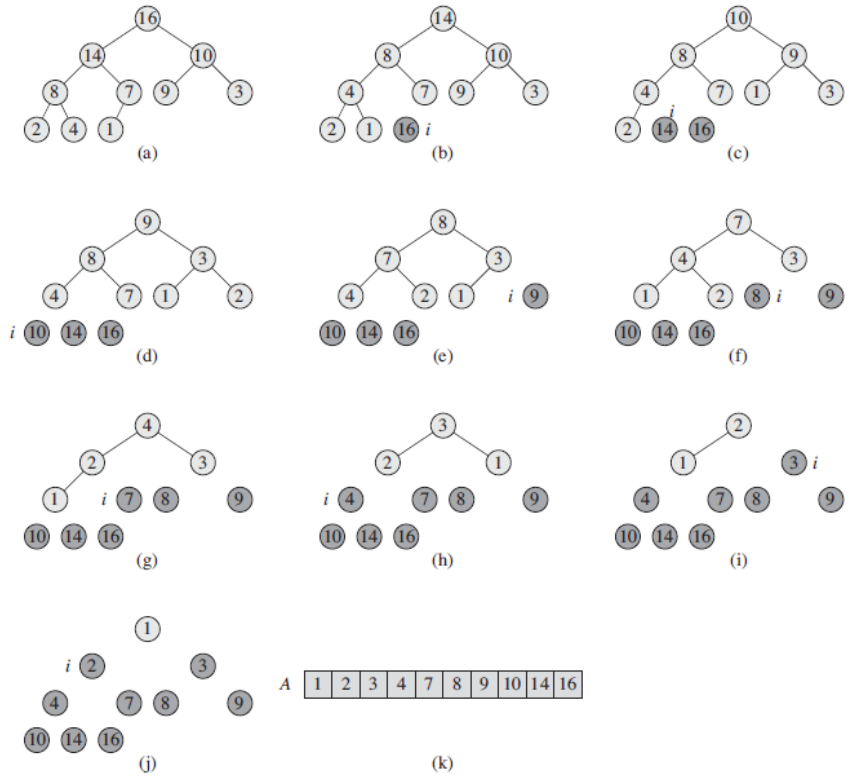
- Solving the recurrence one can show that **the time complexity is O(n)**

HeapSort

```
Heapsort(A) {
  BuildHeap(A)
  for i <- length(A) downto 2 {
    exchange A[1] <-> A[i]
    heapsize <- heapsize -1
    Heapify(A, 1)
}
```

```
BuildHeap(A) {
  heapsize <- length(A)
  for i <- floor( length/2 ) downto 1
    Heapify(A, i)
}
```

```
Heapify(A, i) {
  le <- left(i)
  ri <- right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest <- le
  else
    largest <- i
  if (ri<=heapsize) and
(A[ri]>A[largest])
    largest <- ri
  if (largest != i) {
    exchange A[i] <-> A[largest]
    Heapify(A, largest)
  }
}
```



The operation of HEAPSORT.
(a) The max-heap data structure just after BUILD-MAXHEAP has built it in line 1.
(b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap.
(k) The resulting sorted array A.

Is Heapsort stable? - NO

If a sorting algorithm is said to be "unstable", this means that for any items that rank the same, the order of the tied members is not guaranteed to stay the same with successive sorts of that collection. For a 'stable' sort, the tied entries will always end up in the same order when sorted.

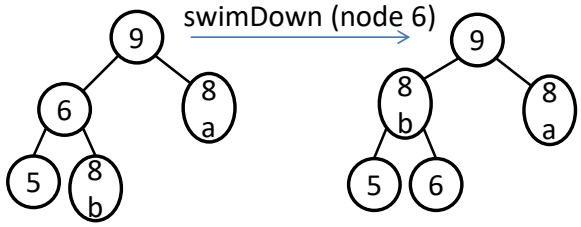
- Both of these operations are unstable: (1) swimDown (2) Going from the built heap to the sorted array (remove max and put at the end)

```
Heapsort(A,N)
1  buildMaxHeap(A,N)
2  for (p=(*N); p≥2; p--)
3      swap A[1] <-> A[p]
4      (*N) = (*N)-1
5      sinkDown(A,1,N)
```

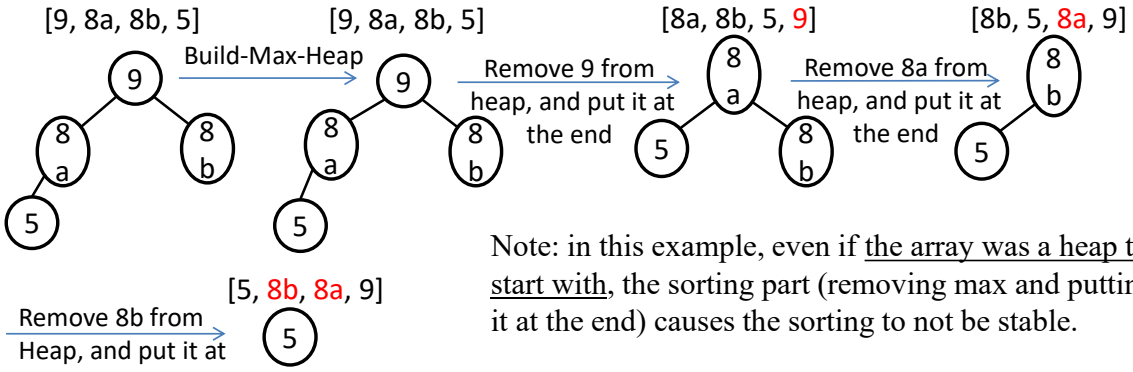
```
sinkDown(A,p,N)
   left = 2*p          // index of left child of p
   right = (2*p)+1 // index of right child of p
   index=p
   if (left≤(*N)&&(A[left]>A[index])
      index = left
   if (right≤(*N))&&(A[right]>A[index])
      index = right
   if (index!=p) {
      swap A[p] <-> A[index]
      sinkDown(A,index,N) }
```

Is Heapsort Stable? - No

Example 1: swimDown operation is not stable. When a node is swapped with his child, they jump all the nodes in between them (in the array).



Example 2: moving max to the end is not stable if there are duplicates:



Note: in this example, even if the array was a heap to start with, the sorting part (removing max and putting it at the end) causes the sorting to not be stable.

Finding the Top k Largest Elements

39

Finding the Top k Largest Elements

- Using a max-heap
- Using a min-heap

Finding the Top k Largest Elements

- Assume N elements
- Using a **max-heap**
  - Build max-heap of size **N** from all elements, then
  - remove k times
  - Requires Θ(N) space if cannot modify the array (build heap in place and remove k)
  - Time: Θ(N + k*lgN)
    - (build heap: Θ(N), k delete ops: Θ(k*lgN))
- Using a **min-heap**
  - Build a min-heap, H, of size **k** (from the first k elements).
  - (N-k) times perform both: *insert* and then *delete* in H.
  - After that, all N elements went through this min-heap and k are left so they **must be** the k largest ones.
  - advantage: less space (Θ(k))
  - Version 1: Time: Θ(k + (N – k)*lgk) (build heap + (N-k) insert & delete)
  - Version 2 (get the top k sorted): Time: Θ(k + N*lgk) = Θ(Nlgk)
    (build heap + (N-k) insert & delete + k delete)

41

Top k Largest with Max-Heap

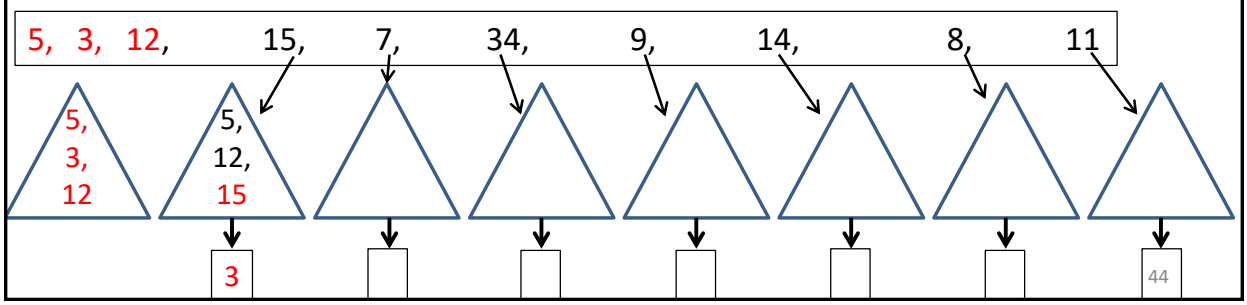
- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)
- Method:
  - Build a max heap using bottom-up
  - Delete/remove 3 (=k) times from that heap
    - **What numbers will come out?**
- Show all the steps (even those for bottom-up build heap). Draw the heap as a tree.

Max-Heap Method Worksheet

- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.

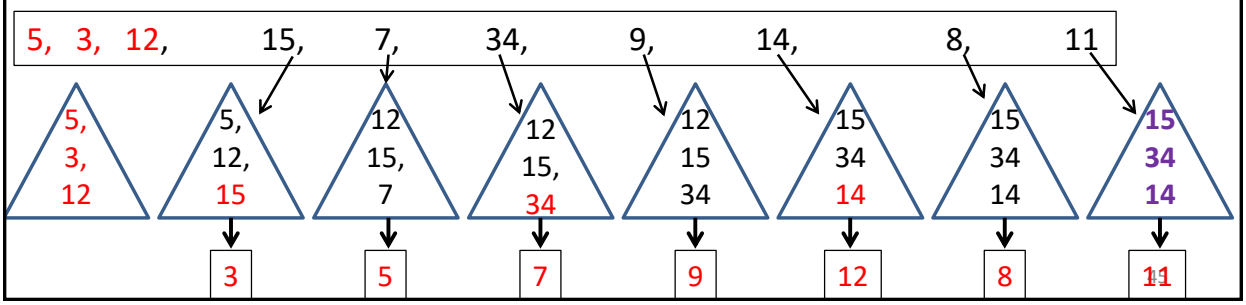
Top k Largest with Min-Heap Worksheet

- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)
- Method:
  - Build a **min heap** using bottom-up from the first 3 (=k) elements: 5,3,12
  - Repeat 7 (=N-k) times: one insert (of the next number) and one remove.
  - Note: Here we do not show the k-heap as a heap, but just the data in it.



Top k Largest with Min-Heap Answers

- What is left in the min heap are the top 3 largest numbers.
  - If you need them in order of largest to smallest, do 3 remove operations.
- Intuition:
  - the MIN-heap acts as a 'sieve' that keeps the **largest** elements going through it.



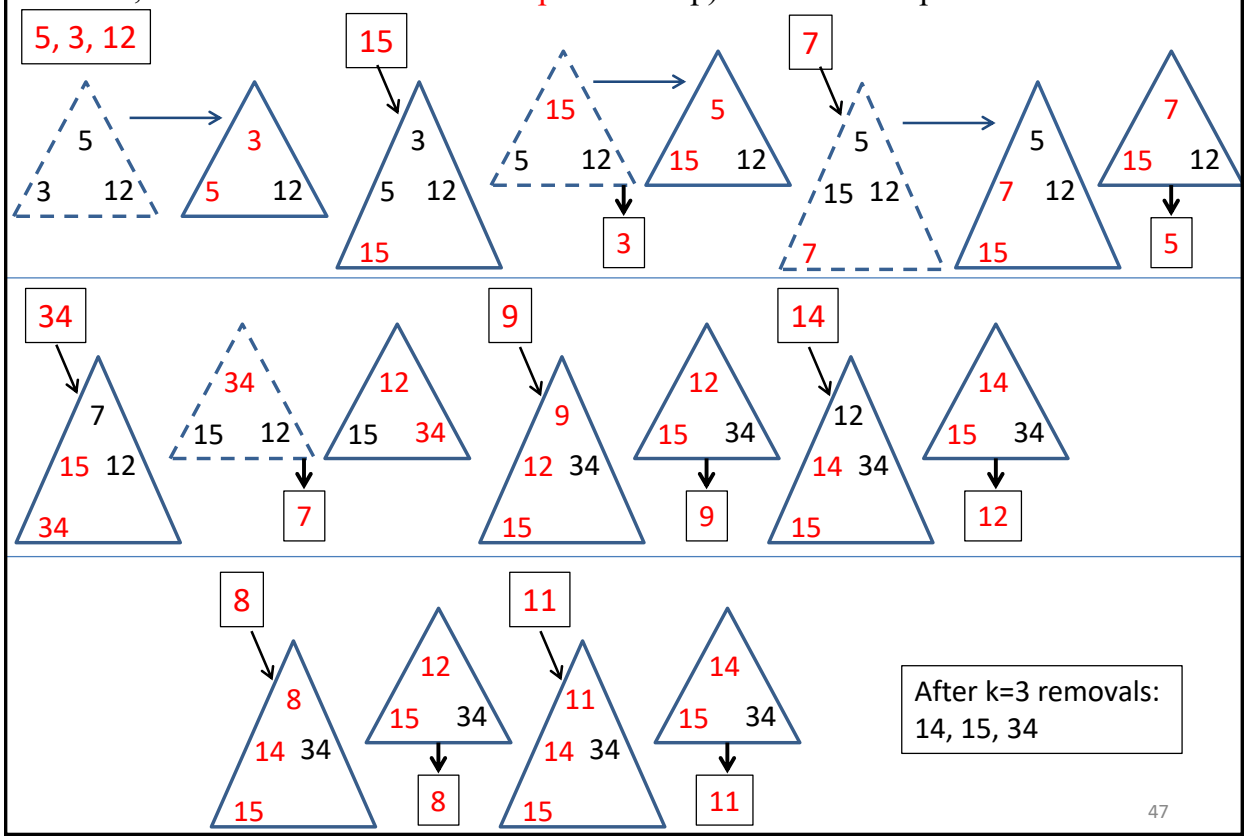
45

Top k Largest with Min-Heap

- Show the actual heaps and all the steps (insert, delete, and steps for bottom-up heap build). Draw the heaps as a tree.
  - N = 10, k = 3, Input: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.

    (Find the top 3 largest elements.)
  - Method:
    - Build a min heap using bottom-up from the first 3 (=k) elements: 5,3,12
    - Repeat 7 (=N-k) times: one insert (of the next number) and one remove.

Top largest k with MIN-Heap: Show the actual heaps and all the steps (for insert, remove, and even those for bottom-up build heap). Draw the heaps as a tree.



47

Other Types of Problems

- Is this (array or tree) a heap?
- Tree representation vs array implementation:
  - Draw the tree-like picture of the heap given by the array …
  - Given tree-like picture, give the array
- Perform a sequence of remove/insert on this heap.
- Decrement priority of node x to k
- Increment priority of node x to k
- Remove a specific node (not the max)