Priority Queues, Heaps, and Heapsort

CSE 2320 – Algorithms and Data Structures
Alexandra Stefan
(includes slides from Vassilis Athitsos)
University of Texas at Arlington

Last modified: 4/19/2018

Overview

- Priority queue
  - A data structure that allows inserting and deleting items.
  - On delete, it removes the item with the HIGHEST priority
  - Implementations (supporting data structures)
    - Array (sorted/unsorted)
    - Linked list (sorted/unsorted)
    - Heap – (an array with a special "order")
- Heap
  - Definition, properties,
  - Operations:
    - increase key (swimUp), decrease key (sinkDown),
    - insert, delete, removeAny,
  - Building a heap: bottom-up (O(N)) and top-down (O(NlgN))
- Heapsort – O(NlgN)
  - Not stable
- Finding top k: with Max-Heap and with Min-Heap
- Extra: Index items – the heap has the index of the element. Heap <-> Data

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)

Priority Queue Implementations

index	0	1	2	3	4	5	6	7	8	9	10	11	12
Sorted	**1**	**1**	**1**	**2**	**3**	**3**	**3**	**4**	**4**	**5**	**5**	**7**	**9**
Unsorted	**1**	**2**	**7**	**5**	**3**	**5**	**4**	**3**	**1**	**1**	**9**	**3**	**4**

How long will it take to delete MAX?
How long will it take to insert value 2?

Arrays and linked lists (sorted or unsorted) can be used as priority queues, but they require O(N) for either insert or delete max.

Data structure	**Insert**	**Delete max**	**Create from batch of N**
Unsorted Array	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
Unsorted Linked List	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
Sorted Array	O(N) (find position, slide elements)	$\Theta(1)$	$\Theta(N\lg N)$ (e.g. mergesort)
Sorted Linked List	O(N) (find position)	$\Theta(1)$	$\Theta(N\lg N)$ (e.g. mergesort)
Heap (an array)	O(lgN) (reorganize)	O(lgN) (reorganize)	$\Theta(N)$

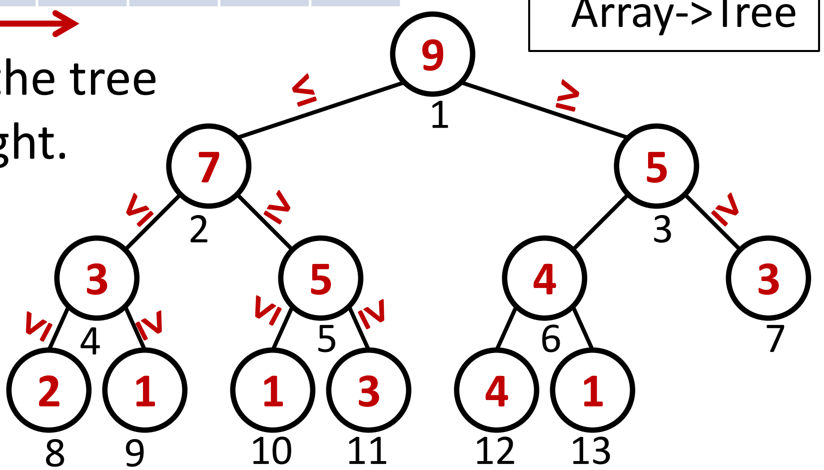
Binary Heap: Stored as Array ⇔ Viewed as Tree

A Heap is stored as an array. Here, the first element is at index 1 (not 0).
It can start at index 0, but 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

Rearrange 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.)

<table>
<tr><th colspan="3">Node at index i</th></tr>
<tr><td colspan="2">Children</td><td rowspan="2">Parent
$\lfloor i/2 \rfloor$</td></tr>
<tr><td>Left
2i</td><td>Right
2i+1</td></tr>
</table>

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 node are to the left.

Heap – Shape Property

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

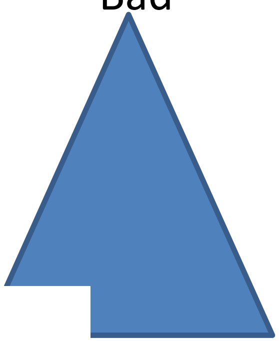
=> All levels are complete except, possibly, the last one.

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

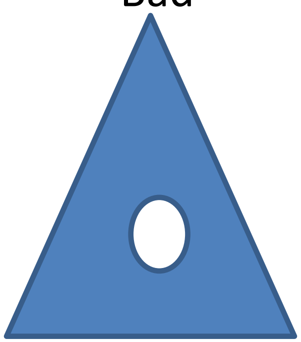
Good

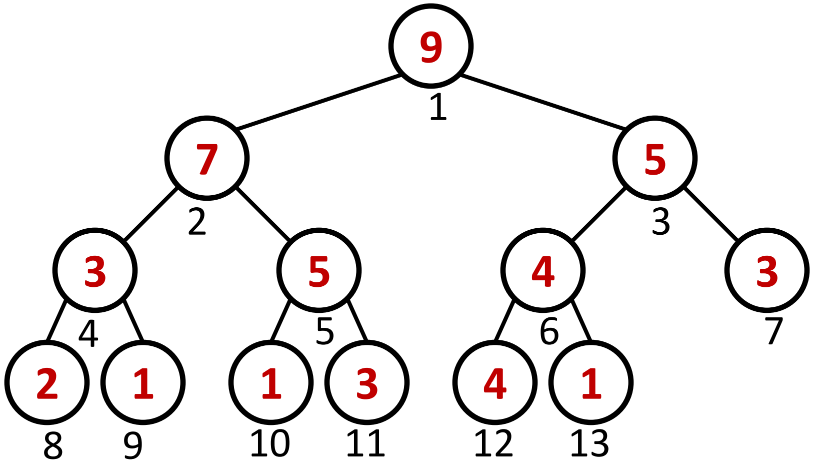
Bad

Bad



Bad

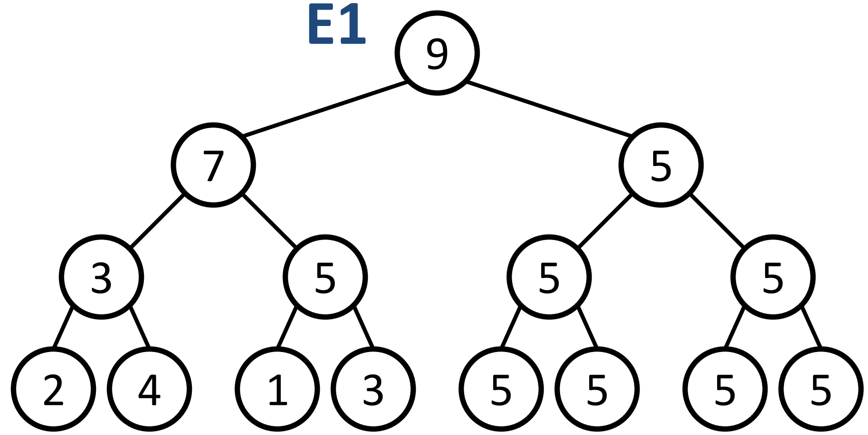




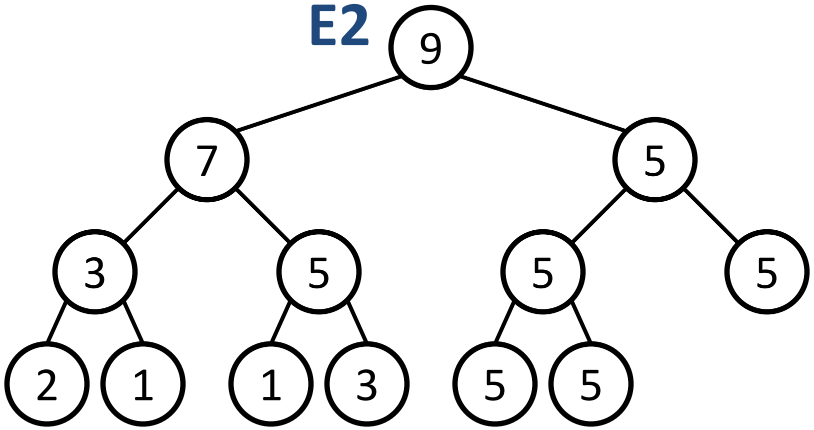
Heap Practice

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

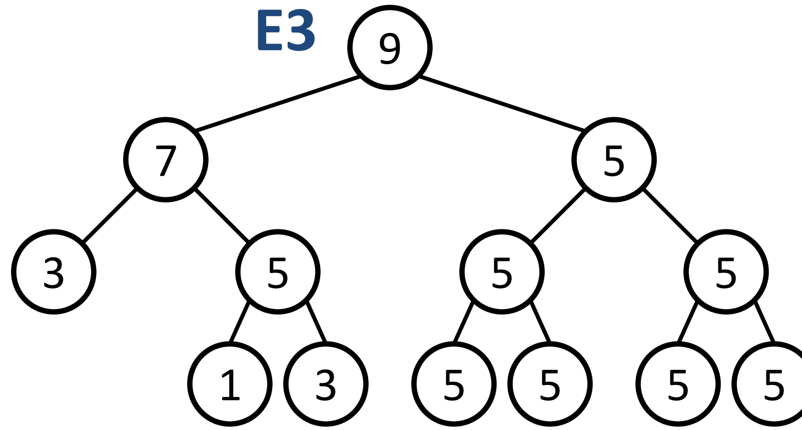
E1



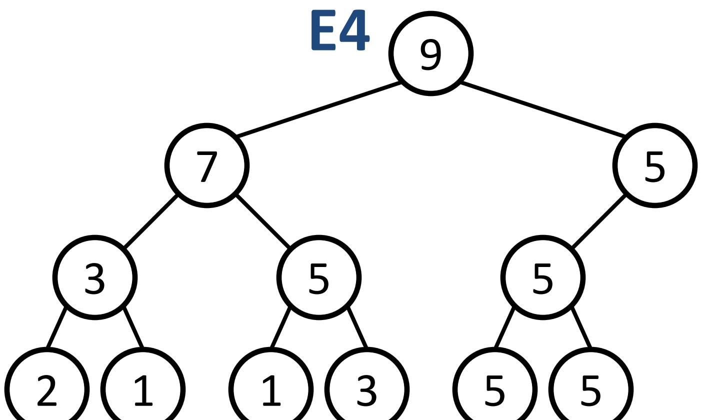
E2



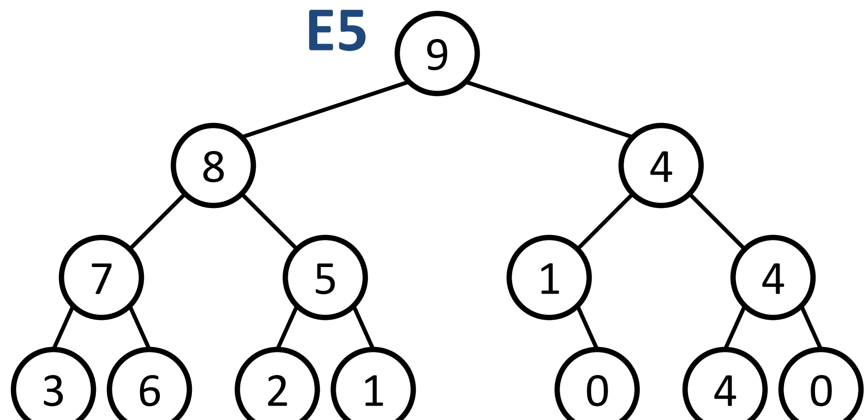
E3



E4



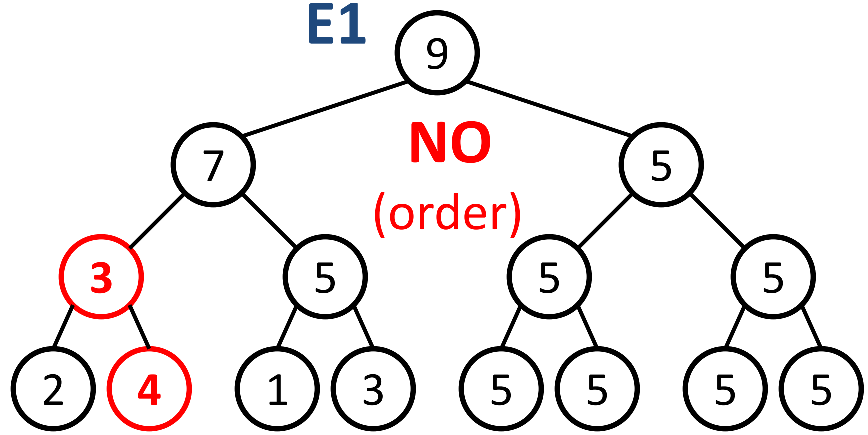
E5



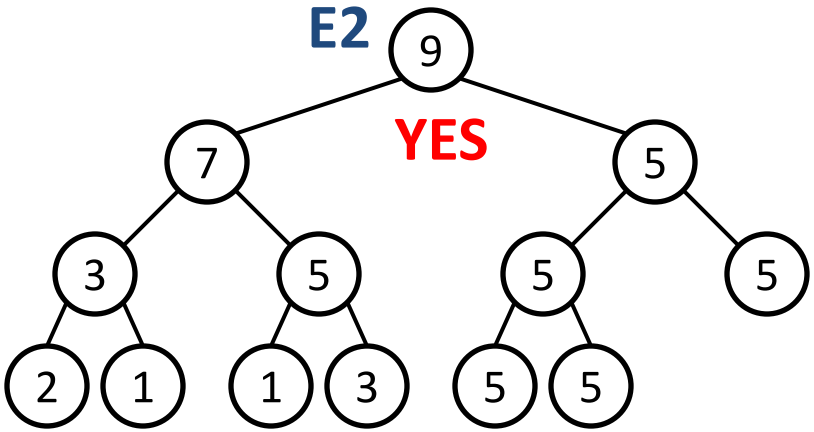
Answers

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

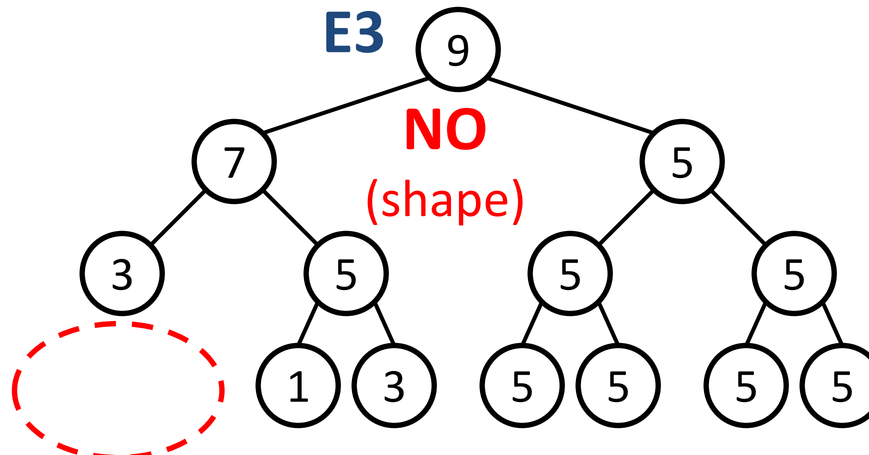
E1



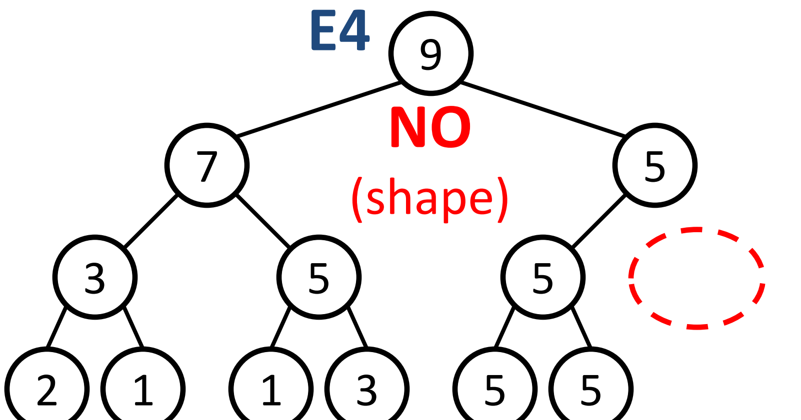
E2



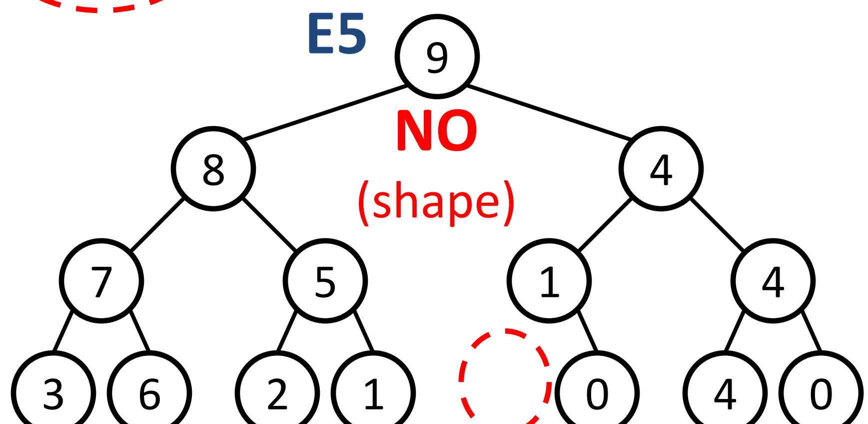
E3



E4



E5



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?

Heap-Based Priority Queues

`insert(A,key,N)` – Inserts x in A.

`maximum(A,N)` – Returns the element of A with the largest key.

`removeMax(A,N)` or `delete(A,N)`

– Removes and returns the element of A with the largest key.

`removeAny(A,p,N)`

– Removes and returns the element of A at index p.

`increaseKey(A,p,k,N)`

– Changes p’s key to be k. Assumes p’s key was initially lower than k. Apply `swimUp`

`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`.

Increase Key

(increase priority of an item)

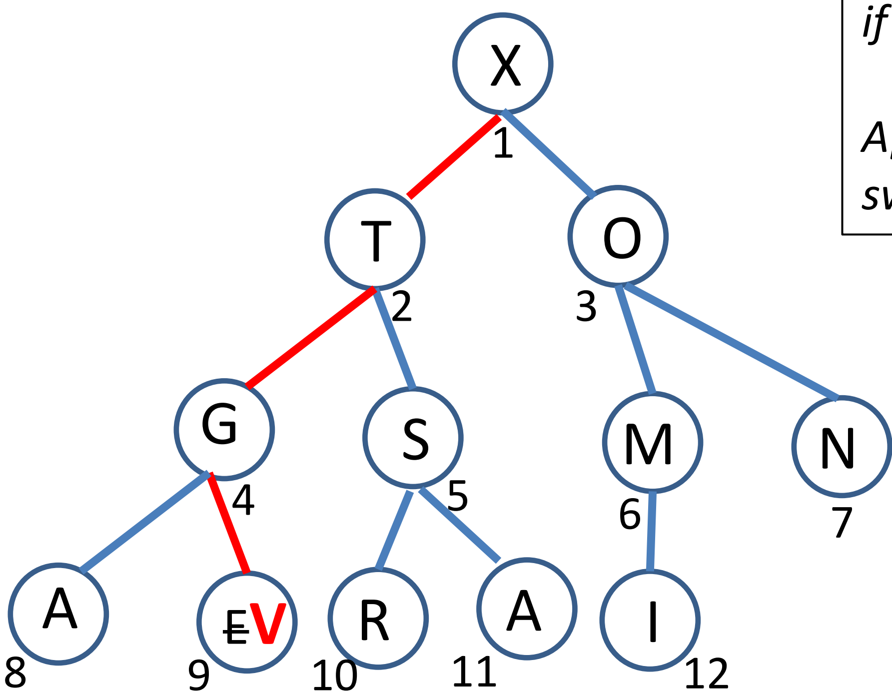
swimUp to fix it

Example: E changes to a V.

- Can lead to violation of the heap property.

swimUp to fix the heap:

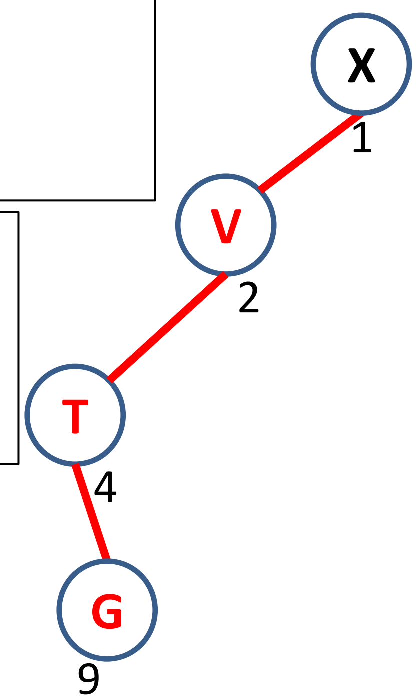
- While last modified node is not the root AND it has priority larger than its parent, swap it with his parent.
  - V not root and V>G? Yes => Exchange V and G.
  - V not root and V>T? Yes => Exchange V and T.
  - V not root and V>X? No. => STOP




```
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
```

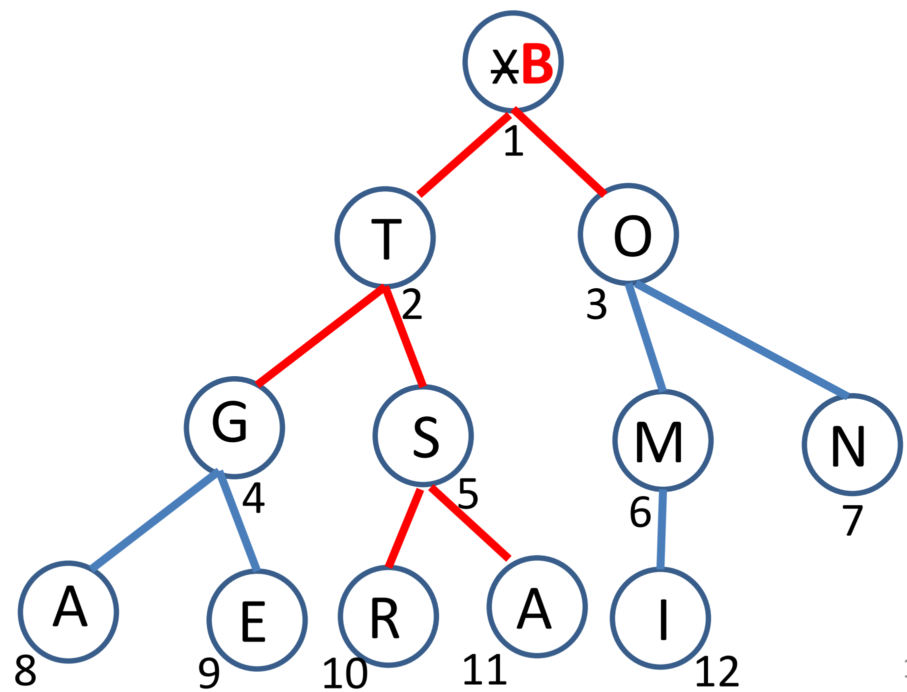
Only the red links are explored => O(lg(N))



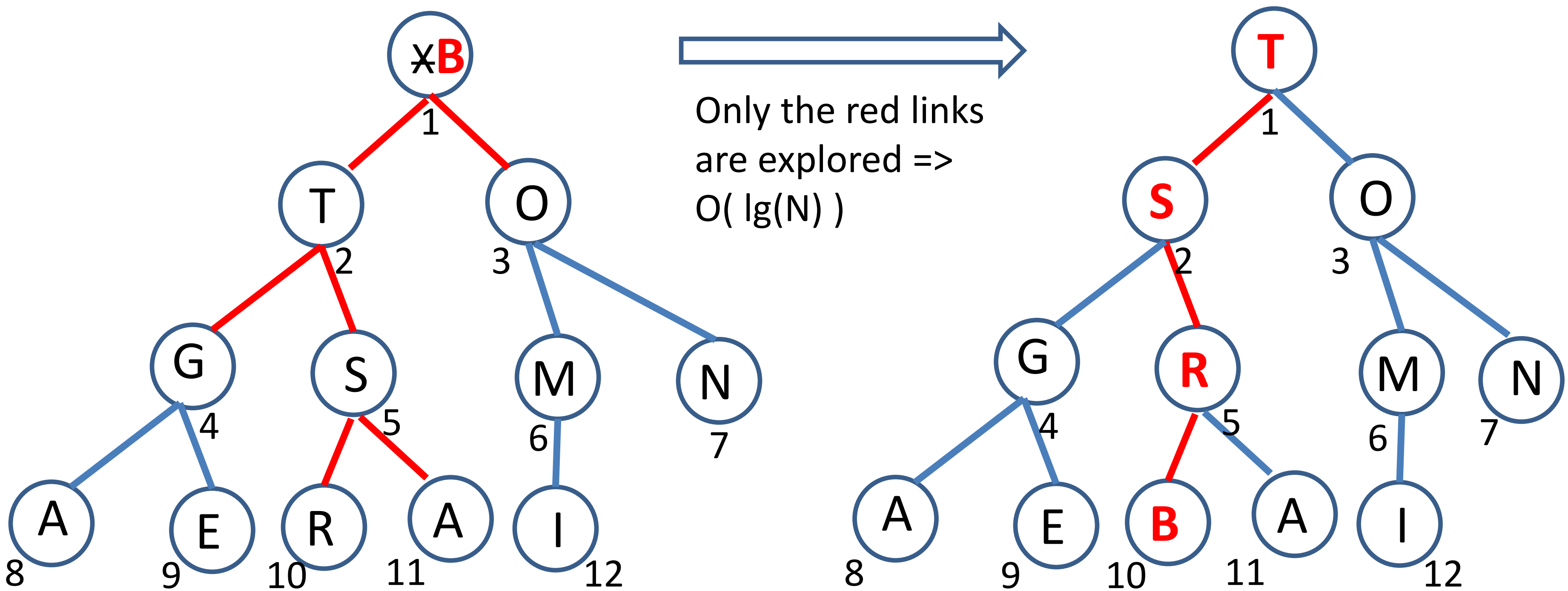
sinkDown(A,p,N)
Decrease key
(Max-Heapify/fix-down/float-down)

- 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)
```



Decrease key
sinkDown(A,p,N)



Only the red links are explored => O(lg(N))

Applications/Usage:

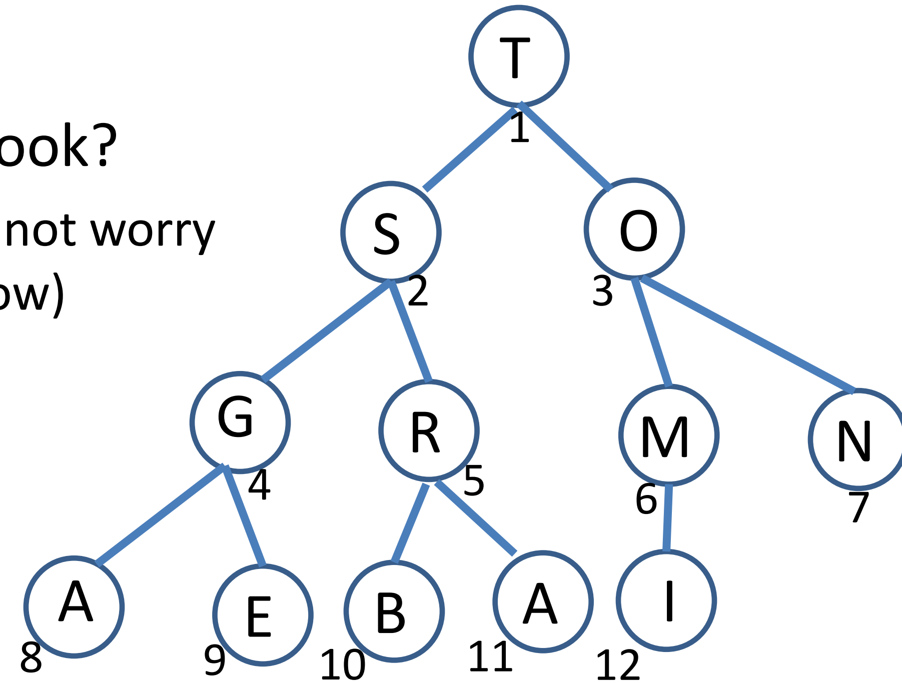
- Priority changed due to data update (e.g. patient feels better)
- Fixing the heap after a delete operation (removeMax)
- One of the cases for removing a non-root node
- Main operation used for building a heap BottomUp.

Inserting a New Record

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**	**I**

- Insert V in this heap.

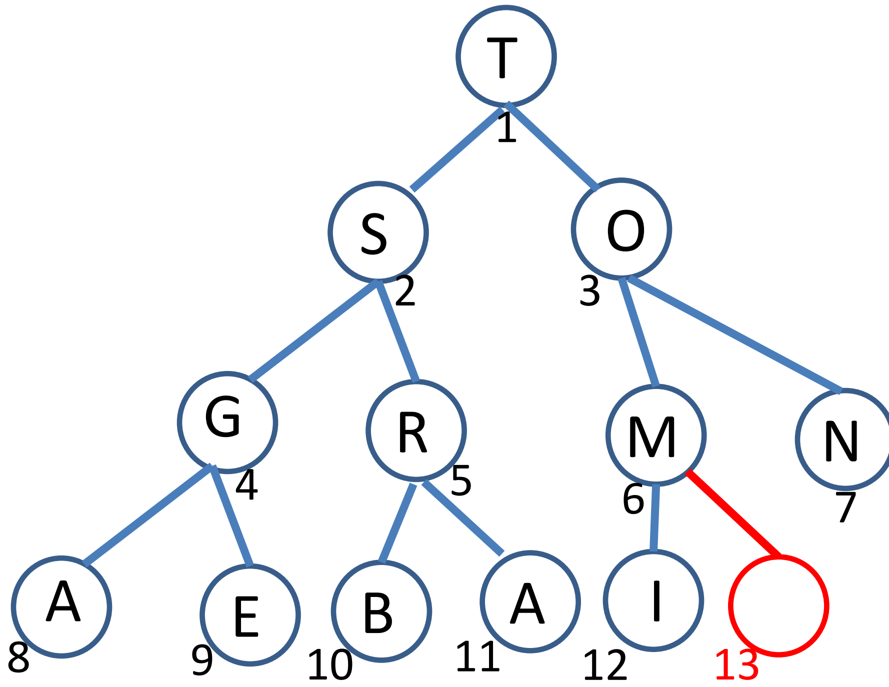
- This is a heap with 12 items.
- How will a heap with 13 items look?
  - Where can the new node be? (do not worry about the data in the nodes for now)



Inserting a New Record

index	1	2	3	4	5	6	7	8	9	10	11	12	**13**
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**I**	

- Let's insert **V**



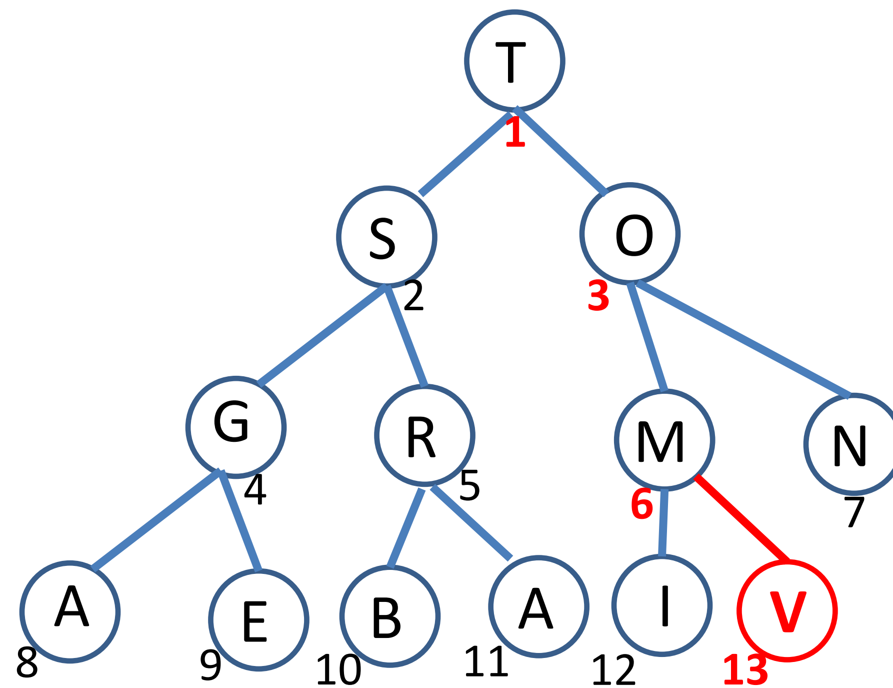
Inserting a New Record

$\lfloor 3/2 \rfloor$ $\lfloor 6/2 \rfloor$ $\lfloor 13/2 \rfloor$

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**	**V**
	T	**S**	**O**	**G**	**R**	**V**	**N**	**A**	**E**	**B**	**A**	**I**	**M**
	T	**S**	**V**	**G**	**R**	**O**	**N**	**A**	**E**	**B**	**A**	**I**	**M**
Final	**V**	**S**	**T**	**G**	**R**	**O**	**N**	**A**	**E**	**B**	**A**	**I**	**M**

- Let's insert **V**
- Put V in the last position and fix up.

```
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
```

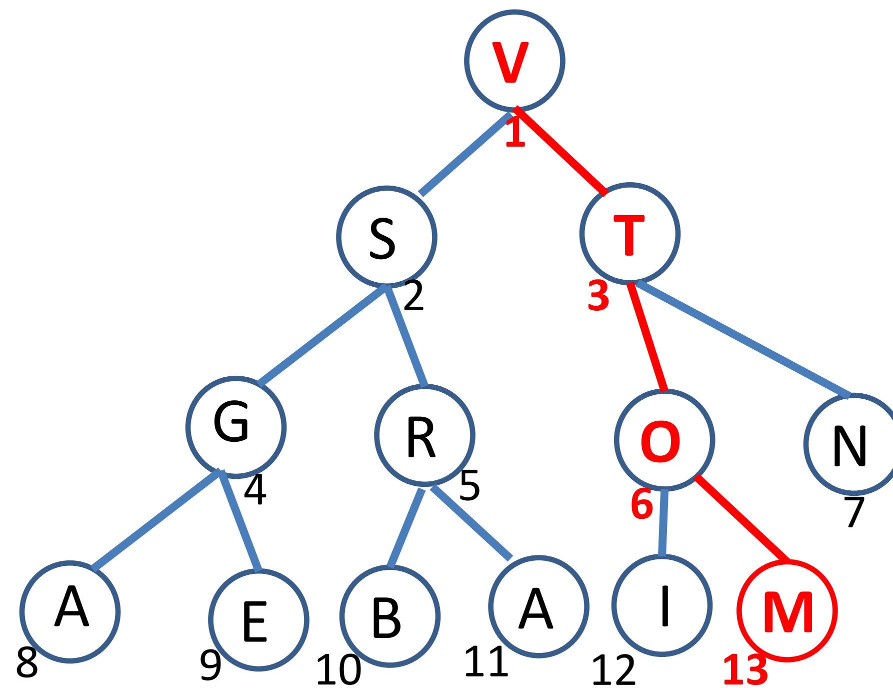


Inserting a New Record

$\underbrace{\lfloor 3/2 \rfloor}$ (indices 1–3) $\underbrace{\lfloor 6/2 \rfloor}$ (indices 4–6) $\underbrace{\lfloor 13/2 \rfloor}$ (indices 7–13)

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**	**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



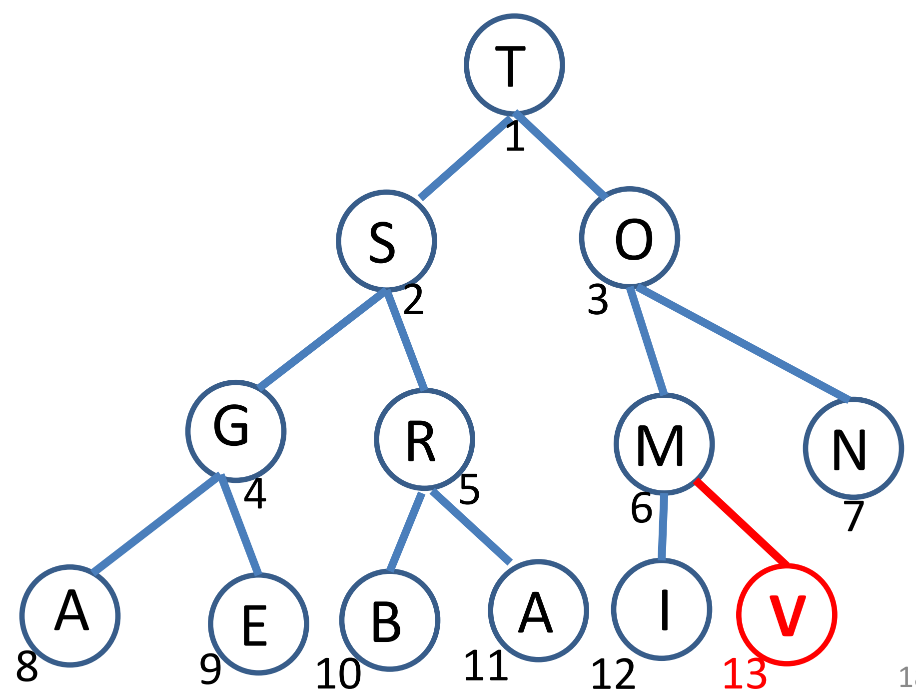
Inserting a New Record - RUNTIME

index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**I**	

```
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
```

Let N be the number of nodes in the heap.

Case	Discussion	Time complexity	Example
Best			
Worst			
General			



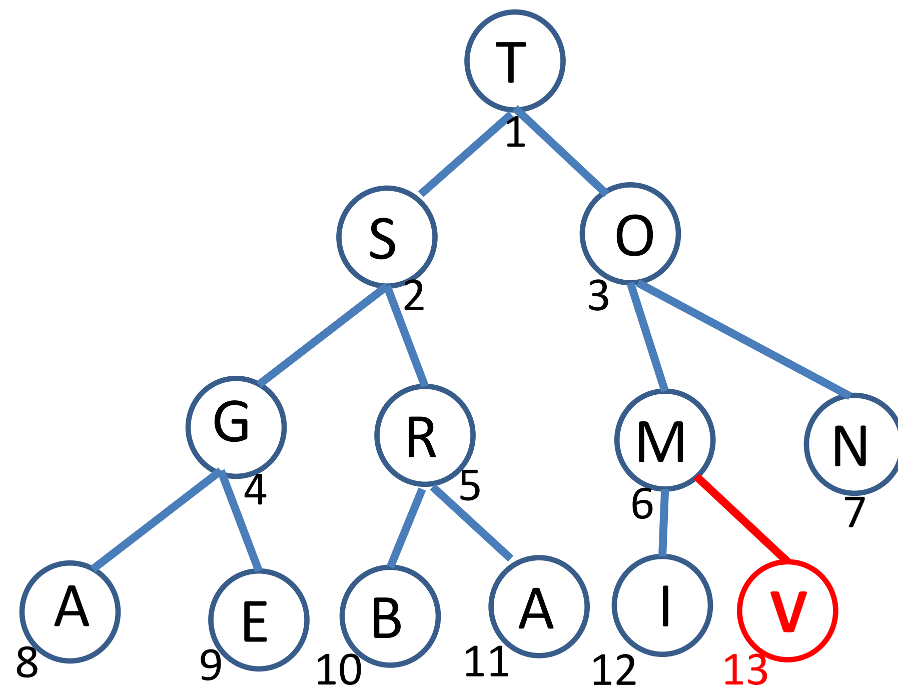
Inserting a New Record - RUNTIME

index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**I**	

```
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
```

Let N be the number of nodes in the heap.

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

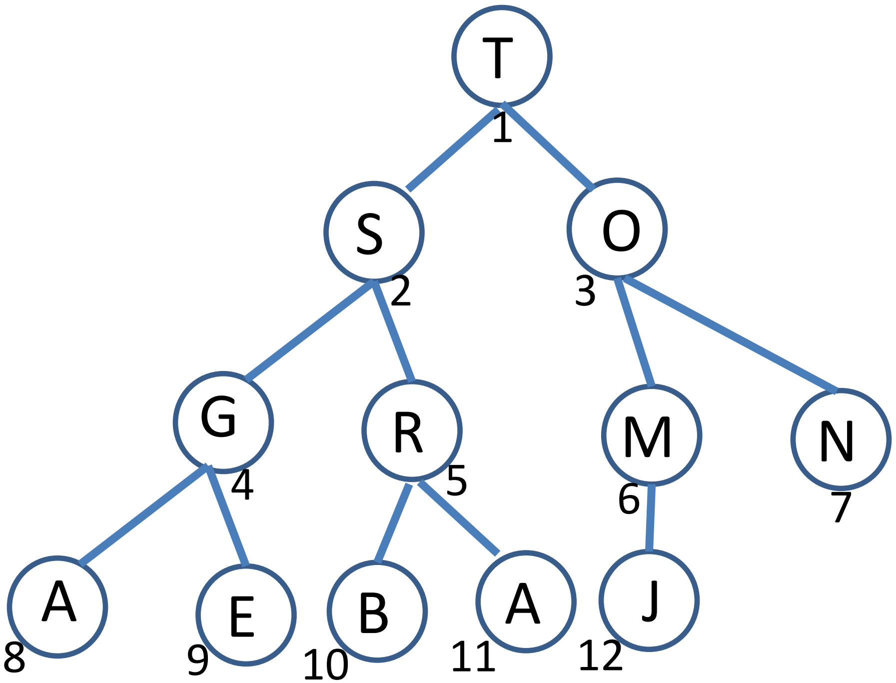
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

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?



Remove Maximum Part 1

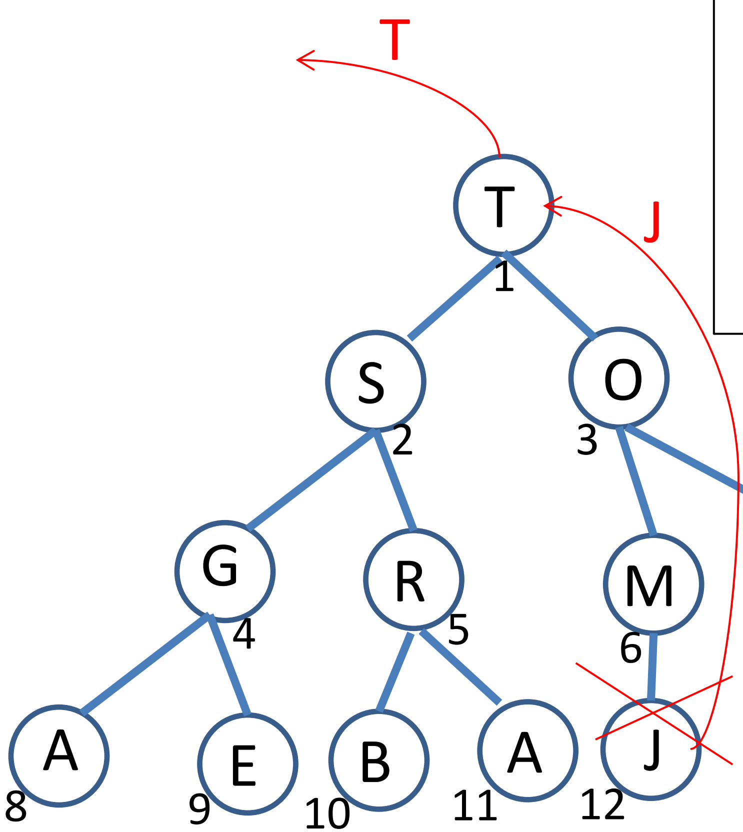
N is 12

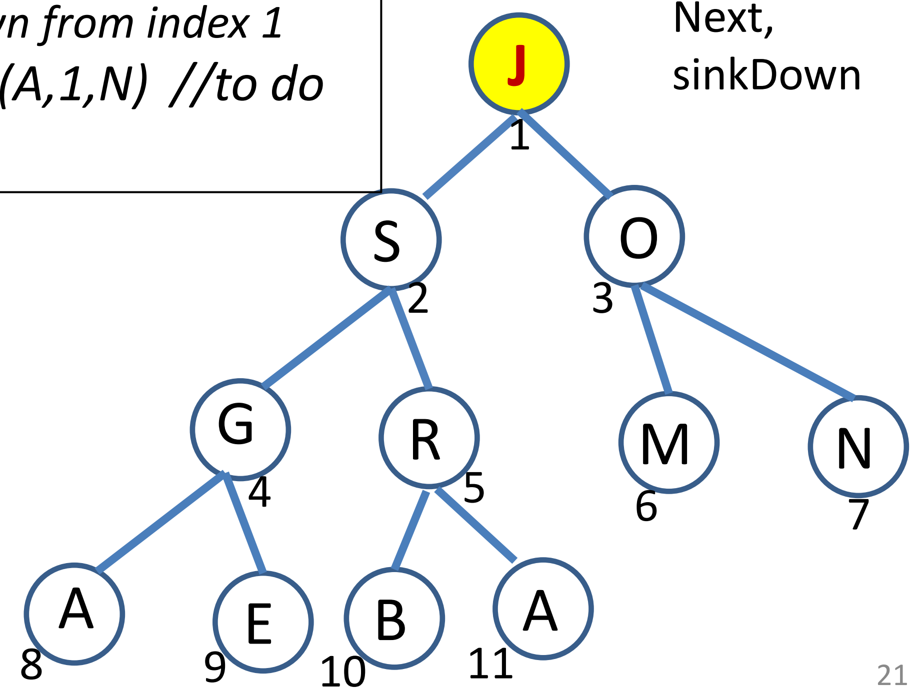
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

N is 11

index	1	2	3	4	5	6	7	8	9	10	11	
value	**J**	**S**	**O**	**G**	**R**	**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
```





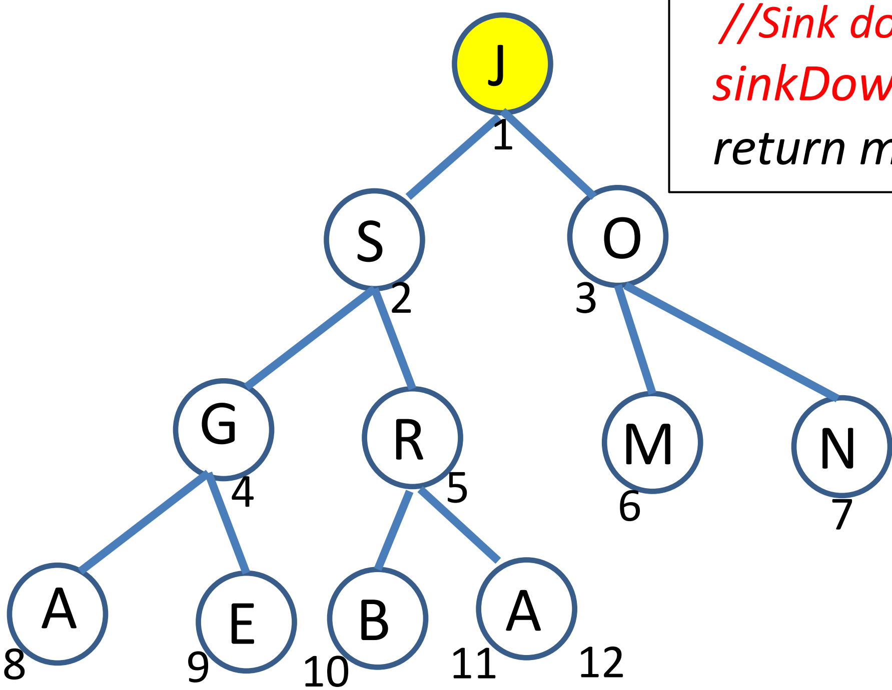
Next, sinkDown

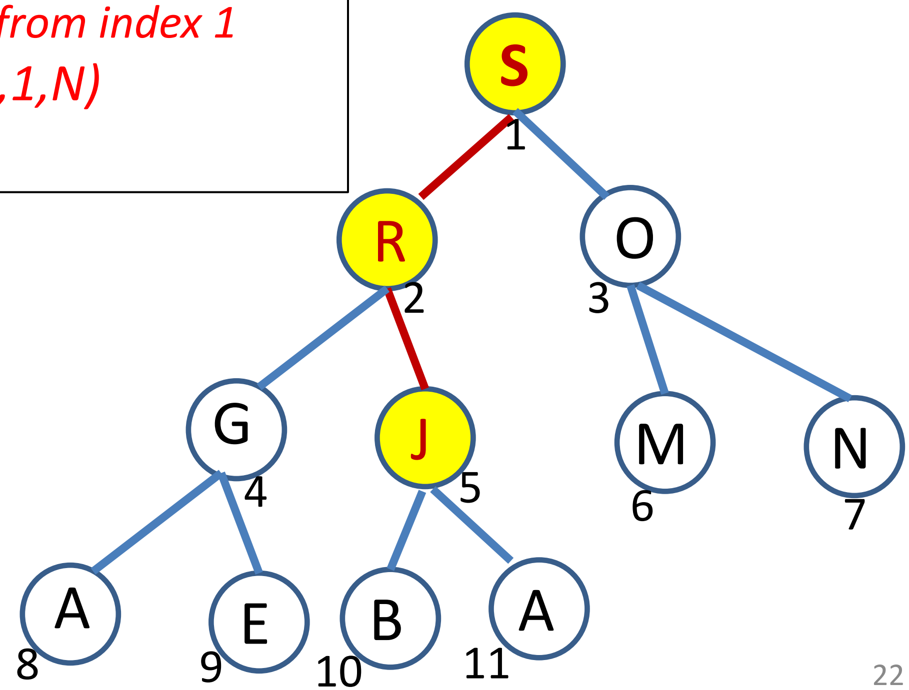
Remove Maximum Part 2

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

index	1	2	3	4	5	6	7	8	9	10	11	
value	**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)
return mx





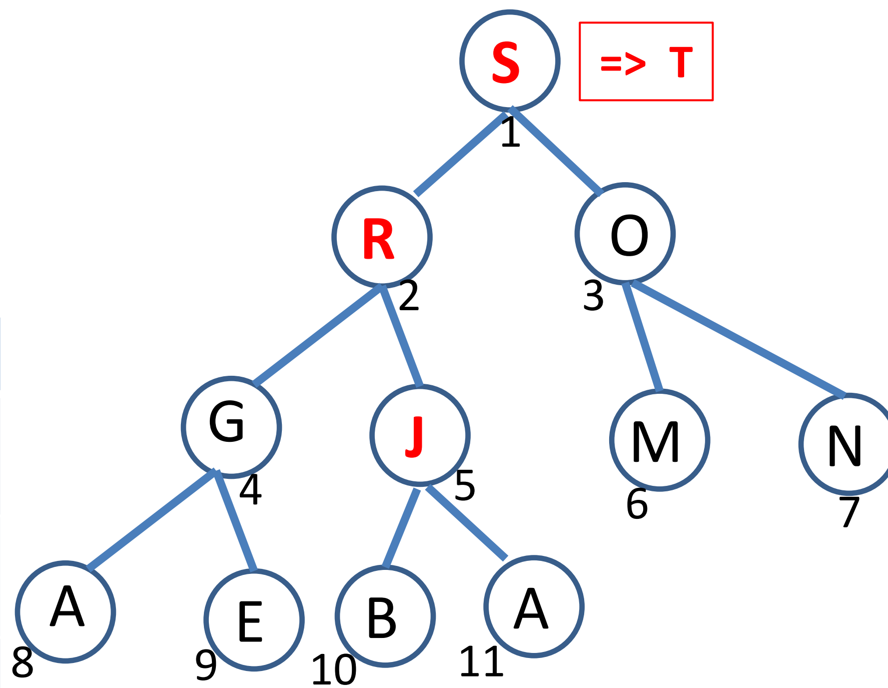
Remove the Maximum - RUNTIME

N

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

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)
return mx

Case	Discussion	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)	



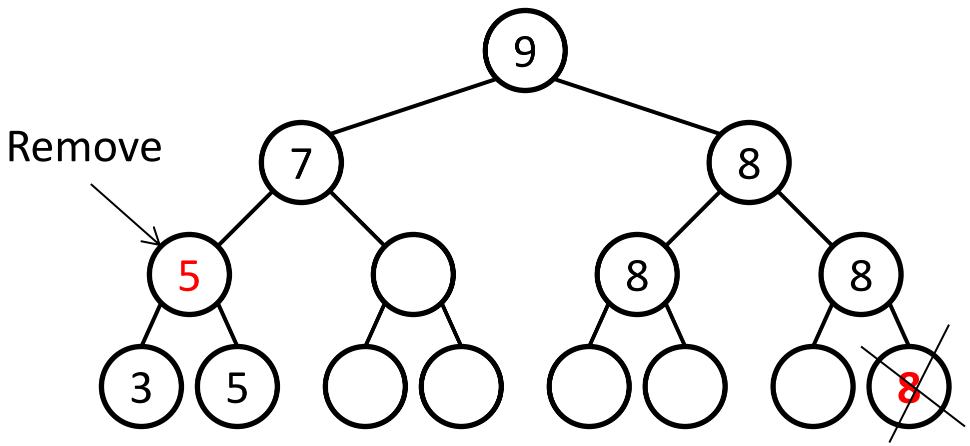
Removal of a Non-Root Node

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
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)

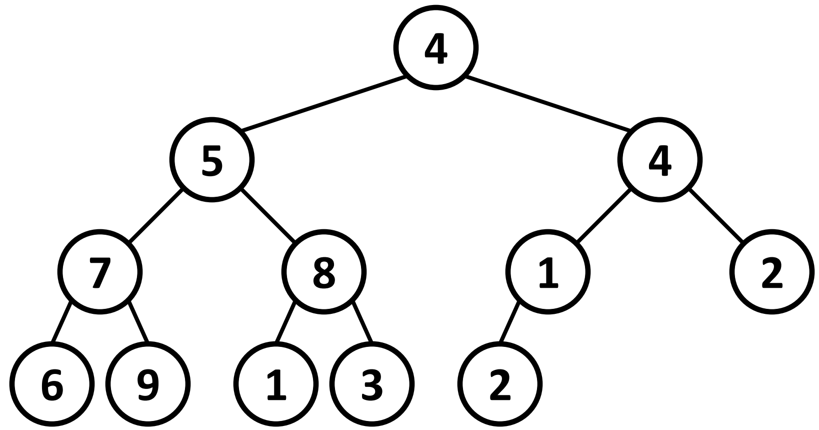
- Deletion:
  - Will always delete the maximum element. This element is always at the top of the heap (the first element of the heap).
  - Deletion of the maximum element:
    - Exchange the first and last elements of the heap.
    - Decrement heap size.
    - Fix down to restore the heap property.
    - Return A[heap_size+1] (the original maximum element).
    - Time = O(lg N)

Batch Initialization

- Batch initialization of a heap
  - The process of converting an unsorted array of data into a heap.
  - We will see 2 methods:
    - top-down and
    - bottom-up.

Batch Initialization Method	Time	Extra space (in addition to the space of the input array)
Bottom-up (fix the given array)	Θ (N)	$\Theta(1)$
Top-down	O(N lg N)	$\Theta(1)$ (if "insert" in the same array: heap grows, orig array gets smaller) $\Theta(N)$ (if insert in new array)

Bottom-Up Batch Initialization



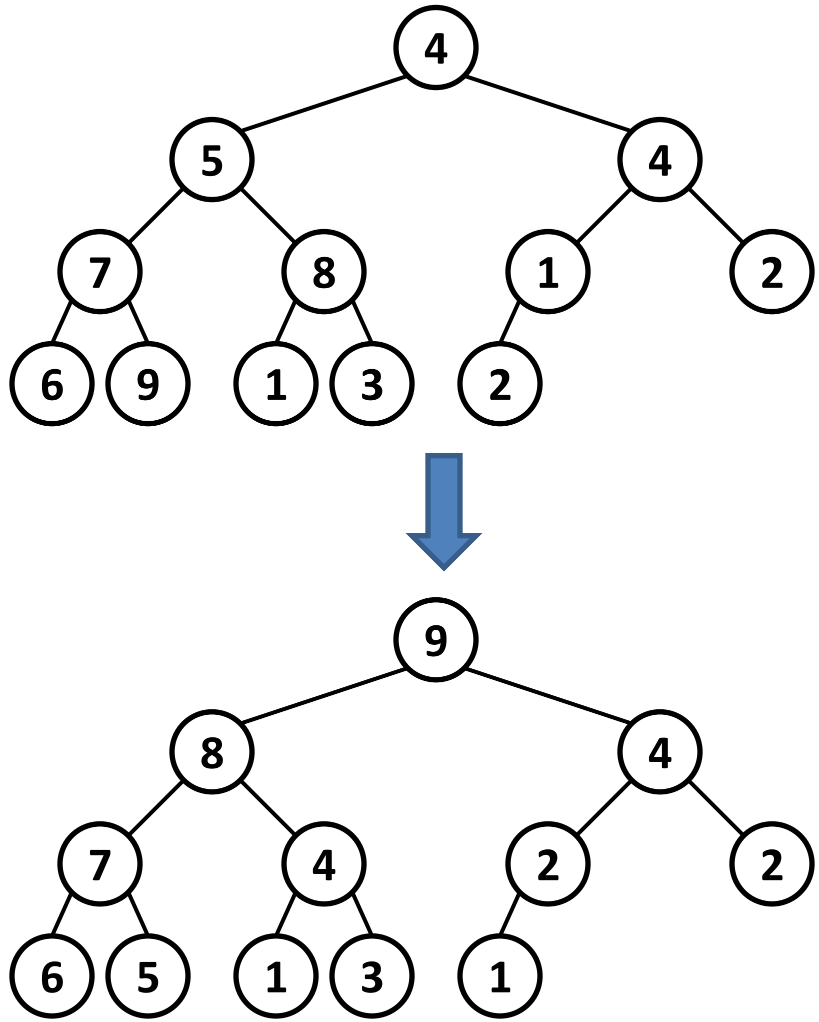
Turns array A into a heap in O(N).
(N = number of elements of A)

buildMaxHeap(A,N) *//Θ(N)*
*for (p = (*N)/2; p>=1; p--)*
    sinkDown(A,p,N)

Time complexity: O(N)
For explanation of this time complexity see extra materials at the end of slides.- Not required.

- See animation: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - Note that they do not highlight the node being processed, but directly the children of it as they are compared to find the larger one of them.

Bottom-Up Batch Initialization



Turns array A into a heap in O(N).
(N = number of elements of A)

buildMaxHeap(A,N) *//Θ(N)*
*for (p = (*N)/2; p>=1; p--)*
    sinkDown(A,p,N)

Time complexity: O(N)
For explanation of this time complexity see extra materials at the end of slides.- Not required.

- See animation: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - Note that they do not highlight the node being processed, but directly the children of it as they are compared to find the larger one of them.

Bottom-Up - Example

- Convert the given array to a heap using bottom-up:

(must work in place):

5, 3, 12, 15, 7, 34, 9, 14, 8, 11.

Priority Queues and Sorting

- Sorting with a heap:
  - Given items to sort:
  - Create a priority queue that contains those items.
  - Initialize result to empty list.
  - While the priority queue is not empty:
    - Remove max element from queue and add it to beginning of result.

- Heapsort – $\Theta(N \lg N)$
  - builds the heap in $O(N)$.

Heapsort

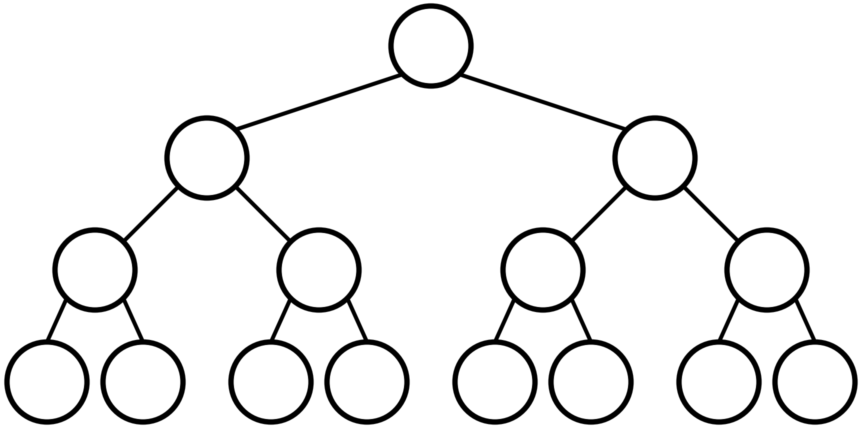
In order to sort an array in increasing order, use a Max-Heap. It will allow you to share the same array for the shrinking heap and growing sequence of sorted numbers.

In the pseudocode below:

- Where is the sorted data?
- How does the heap get shorter?

Give an example that takes Θ(N lg N) .
Give an example that takes Θ(N)
(extreme case: all equal).

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



Is Heapsort stable? - NO

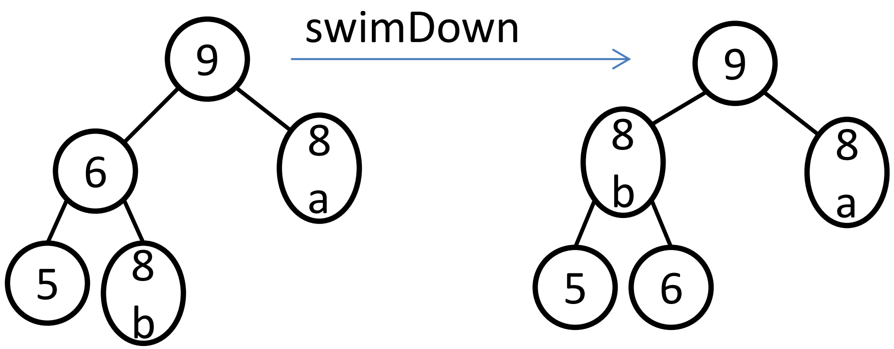
- Both of these operations are unstable:
  - swimDown
  - Going from the built heap to the sorted array (remove max and put at the end)

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

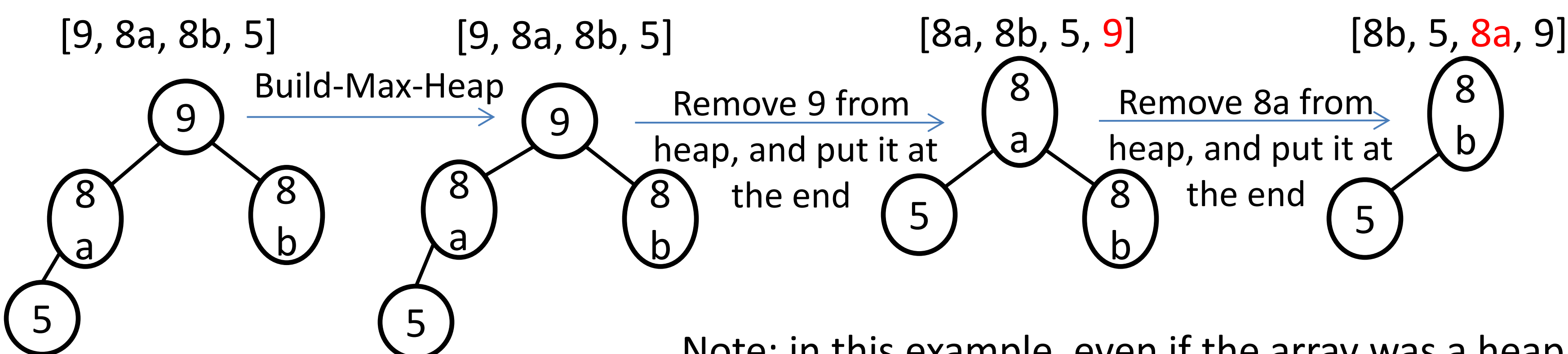
```
swimDown(A,p,N)
  left = 2*p          // index of left child of p
  right = (2*p)+1 // index of right child of p
  index=p
  if (left≤(*N)&&(A[left]>A[index])
     index = left
  if (right≤(*N))&&(A[right]>A[index])
     index = right
  if (index!=p)
     swap A[p] <-> A[index]
     sinkDown(A,index,N)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

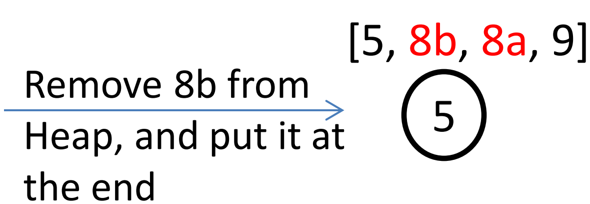
Is Heapsort Stable? - No

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:





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.

Top-Down Batch Initialization

- Build a heap of size N by repeated insertions in an originally empty heap.
  - E.g. build a max-heap from: 5, 3, 20, 15, 7, 12, 9, 14, 8, 11.
- Time complexity? **O(NlgN)**
  - N insertions performed.
  - Each insertion takes O(lg X) time.
    - X- current size of heap.
    - X goes from 1 to N.
  - In total, worst (and average) case: Θ(N lg N)
    - T(N) = Θ(NlgN).
      - The last N/2 nodes are inserted in a heap of height (lgN)-1. => T(N) = Ω(NlgN).
        - » Example that results in Θ(N)?
      - Each of the N insertions takes at most lgN. => T(N) = O(NlgN)

Finding the Top k Largest Elements

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
  - May require extra space if cannot modify the array (build heap in place and remove k)
  - Time: $\Theta(N + k*\lg N)$
    - (build heap: $\Theta(N)$, k delete ops: $\Theta(k*\lg N)$)
- 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 (constant space: k)
  - Version 1: Time: $\Theta(k + (N – k)*\lg k)$ (build heap + (N-k) insert & delete)
  - Version 2 (get the top k sorted): Time: $\Theta(k + N*\lg k) = \Theta(N\lg k)$

    (build heap + (N-k) insert & delete + k delete)

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.

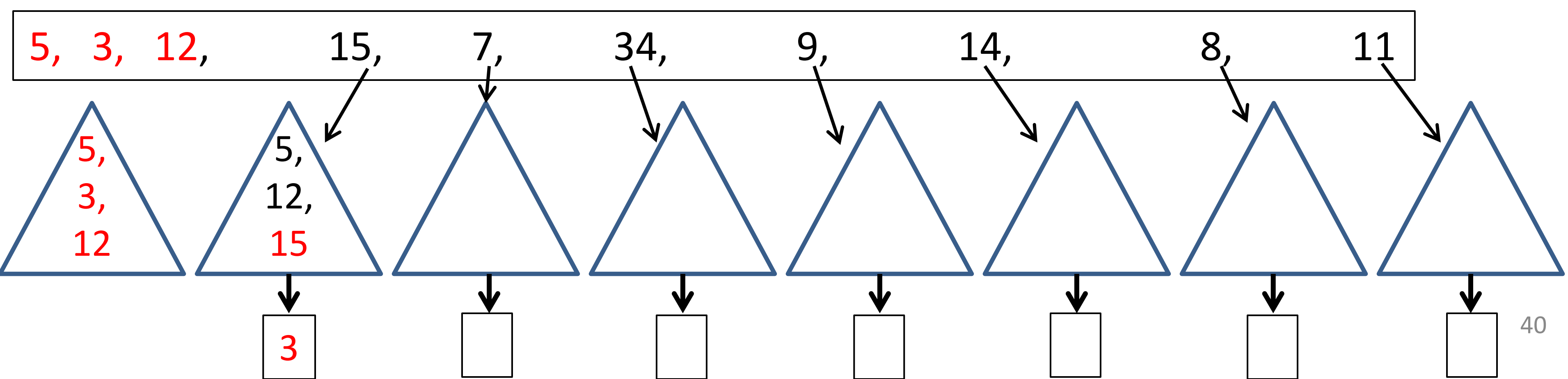
Max-Heap Method Worksheet

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

Top k Largest with Min-Heap
Worksheet

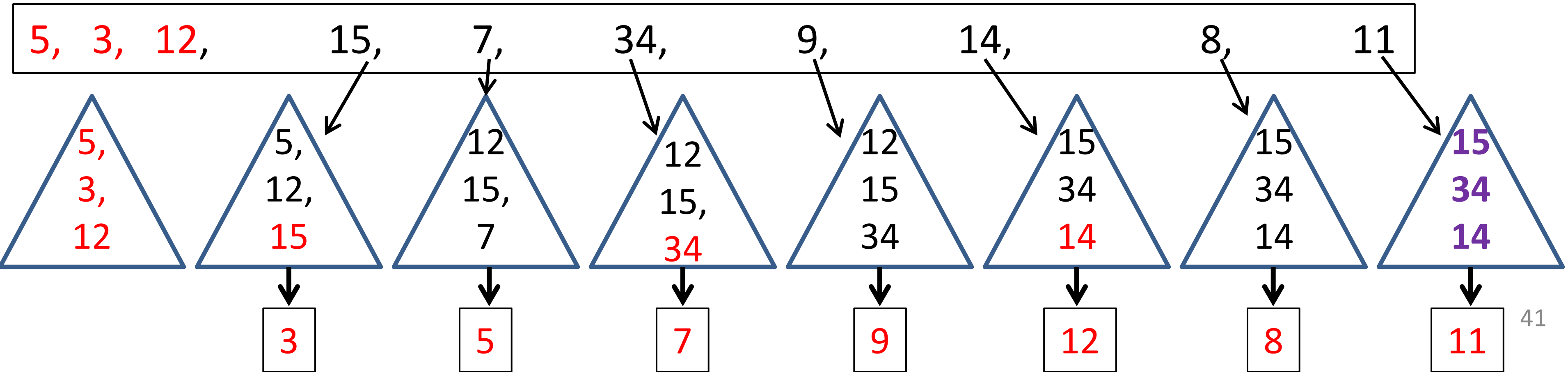
- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.

  (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.

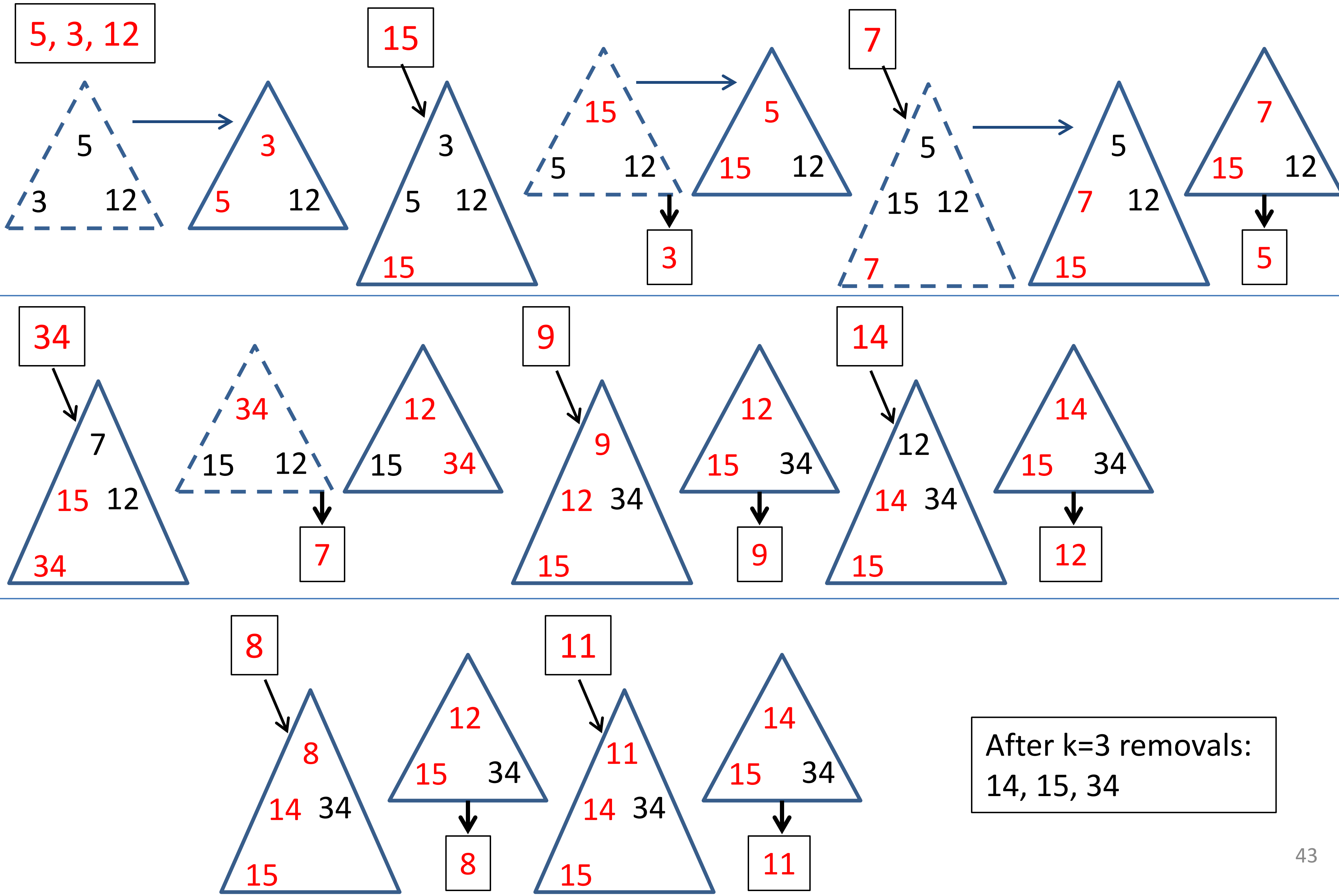


Top k Largest with Min-Heap

- 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.



After k=3 removals:
14, 15, 34

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)

- Work done in the slides: Delete, top k, index heaps,…
  - Delete is: delete_max or delete_min.

Extra Materials

Running Time of BottomUp Heap Build

- How can we analyze the running time?
- To simplify, suppose that last level if complete: => $N = 2^n - 1$ (=> last level is (n-1) => heap height is (n-1) = lgN) (see next slide)
- Counter *p* starts at value $2^{n-1} - 1$.
  - That gives the last node on level n-2.
  - At that point, we call *swimDown* on a heap of height 1.
  - For all the (2^{n-2}) nodes at this level, we call *swimDown* on a heap of height 1 (nodes at this level are at indexes *i* s.t. $2^{n-1}-1 \geq i \geq 2^{n-2}$).

……

  - When *p* is 1 ($=2^0$) we call *swimDown* on a heap of height n-1.

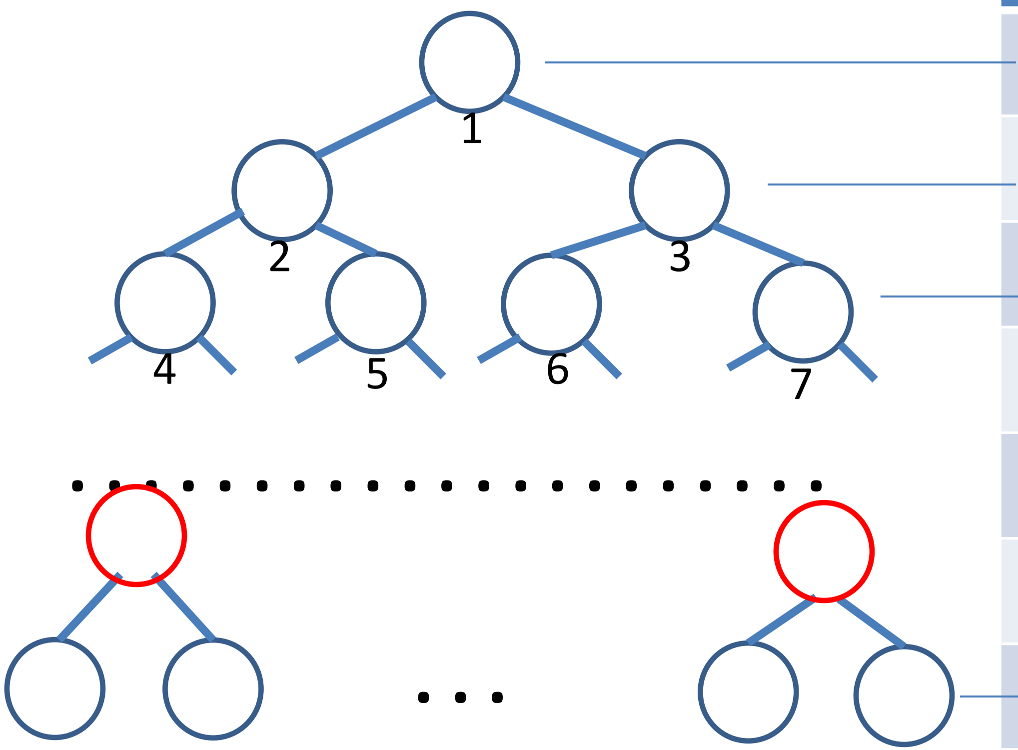
```
buildMaxHeap(A,N)
   for (p = N/2; p>=1; p--)
      sinkDown(A,p,N)
```

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

Running Time: O(N)

Counter from:	**Counter to:**	**Level**	**Nodes** per level	**Height** of heaps rooted at these nodes	**Time** per node (fixDown)	**Time** for fixing all nodes at this level
2^{n-2}	$2^{n-1} – 1$	n-2	2^{n-2}	1	O(1)	$O(2^{n-2} * 1)$
2^{n-3}	$2^{n-2} – 1$	n-3	2^{n-3}	2	O(2)	$O(2^{n-3} * 2)$
2^{n-4}	$2^{n-3} – 1$	n-4	2^{n-4}	3	O(3)	$O(2^{n-4} * 3)$
…						
$2^0 = 1$	$2^1-1 = 1$	0	$2^0 = 1$	n – 1	O(n-1)	$O(2^0 * (n-1))$

- To simplify, assume: **$N = 2^n - 1$**.
- The analysis is a bit complicated . Pull out 2^{n-1} gives: $2^{n-1}\sum_{k=1}^{n-1} kx^k \le \sum_{k=1}^{\infty} kx^k \rightarrow 2^{n-1}\frac{x}{(1-x)^2}$ for $x=\frac{1}{2}$ because $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, for $|x|<1$,
- Total time: sum over the rightmost column: $O(2^{n-1})$ => **O(N) (linear!)**

Index Heap, Handles

- So far:
  - We assumed that the actual data is stored in the heap.
  - We can increase/decrease priority of any specific node and restore the heap.
- In a real application we need to be able to do more
  - Find a particular record in a heap
    - John Doe got better and leaves. Find the record for John in the heap.
    - (This operation will be needed when we use a heap later for MST.)
  - You cannot put the actual data in the heap
    - You do not have access to the data (e.g. for protection)
    - To avoid replication of the data. For example you also need to frequently search in that data so you also need to organize it for efficient search by a different criteria (e.g. ID number).

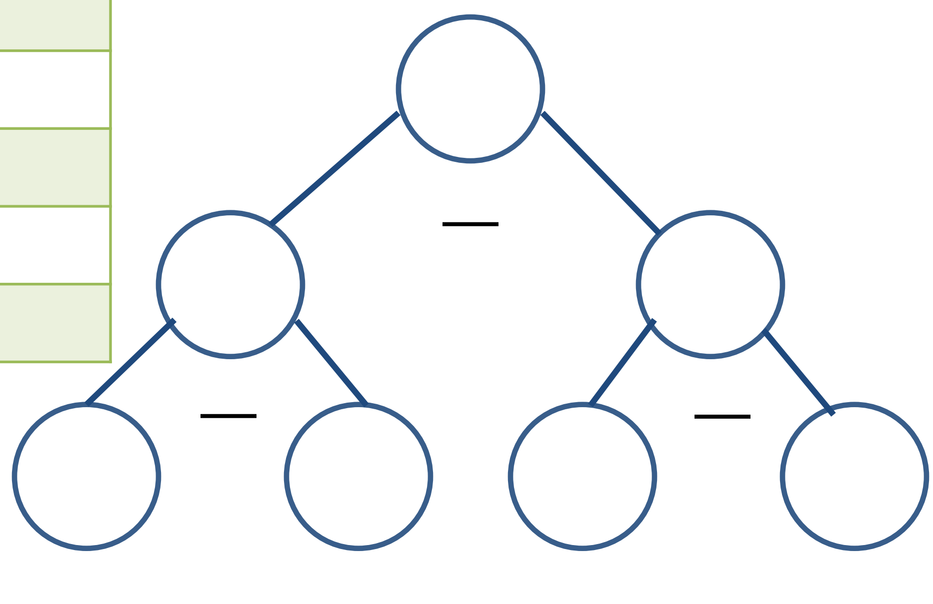
Index Heap Example - Workout

1. Show the heap with this data (fill in the figure on the right based on the **HA array**).
   1. For each heap node show the corresponding array index as well.

Index	**HA** **(H->A)**	**AH** **(A->H)**	**Name**	**Priority**	**Other data**
1	**5**	**2**	Aidan	10	
2	**1**	**4**	Alice	7	
3	**4**	**5**	Cam	10	
4	**2**	**3**	Joe	13	
5	**3**	**1**	Kate	20	
6	**6**	**6**	Mary	4	
7	**7**	**7**	Sam	6	

HA – Heap to Array (*the actual heap*)

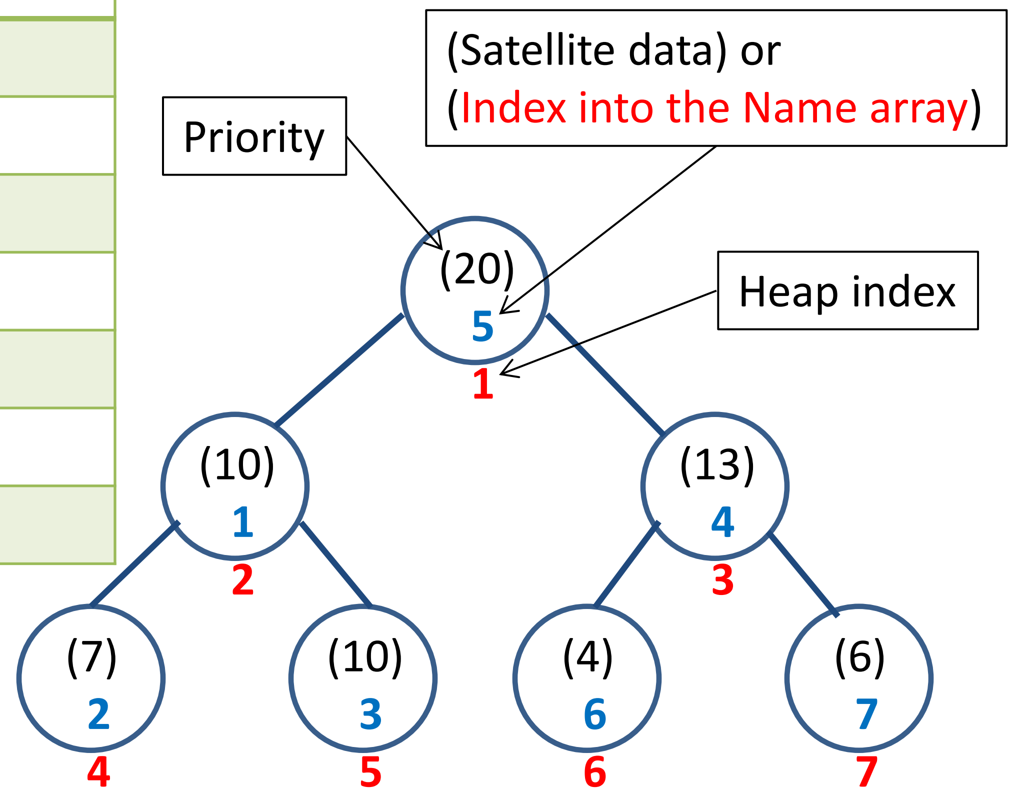
AH – Array to Heap



Index Heap Example - Solution

HA – Heap to Array
AH – Array to Heap

Index	HA (H->A)	AH (A->H)	Name	Priority	Other data
1	5	2	Aidan	10	
2	1	4	Alice	7	
3	4	5	Cam	10	
4	2	3	Joe	13	
5	3	1	Kate	20	
6	6	6	Mary	4	
7	7	7	Sam	6	



Property:
HA(AH(j) = j e.g. HA(AH(5) = 5
AH(HA(j) = j e.g. AH(HA(1) = 1

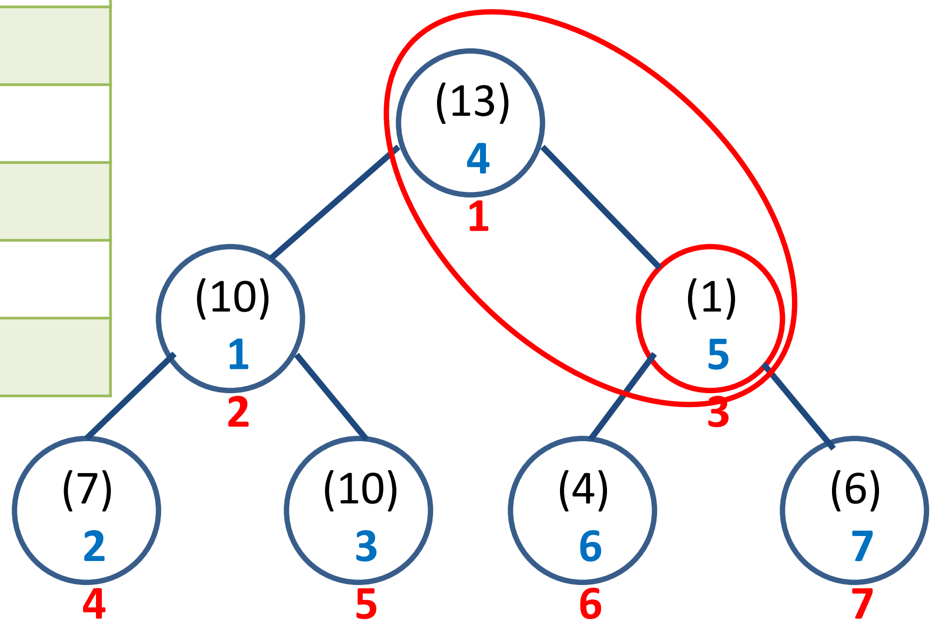
Decrease Kate's priority to 1. Update the heap.
To swap nodes p_1 and p_2 in the heap: HA[p_1]<->HA[p_2], and AH[HA[p_1]] <-> AH[HA[p_2]].

Index Heap Example

Decrease Key – (Kate 20 -> Kate 1)

HA – Heap to Array
AH – Array to Heap

Index	HA (H->A)	AH (A->H)	Name	Priority	Other data
1	~~5~~ 4	2	Aidan	10	
2	1	4	Alice	7	
3	~~4~~ 5	5	Cam	10	
4	2	~~3~~ 1	Joe	13	
5	3	~~1~~ 3	Kate	~~20~~ 1	
6	6	6	Mary	4	
7	7	7	Sam	6	



Property:
HA(AH(j) = j e.g. HA(AH(5) = 5
AH(HA(j) = j e.g. AH(HA(1) = 1

Decrease Kate's priority to 1. Update the heap.
To swap nodes 1 and 3 in the heap: HA[1]<->HA[3], and AH[HA[1]] <-> AH[HA[3]].

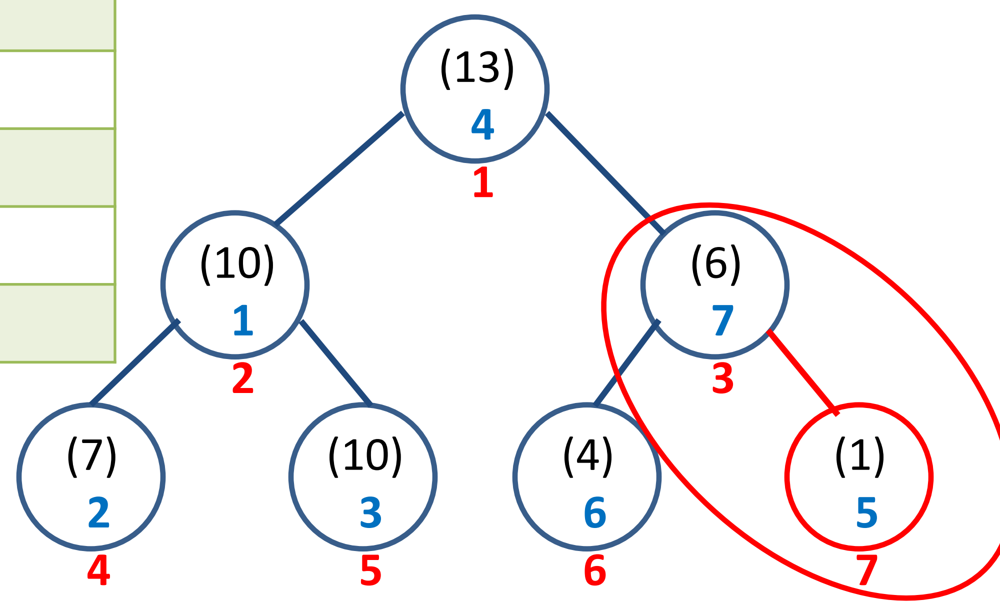
Index Heap Example

Decrease Key - cont

HA – Heap to Array
AH – Array to Heap

Index	HA (H->A)	AH (A->H)	Name	Priority	Other data
1	~~5~~ 4	2	Aidan	10	
2	1	4	Alice	7	
3	~~4~~ ~~5~~ 7	5	Cam	10	
4	2	~~3~~ 1	Joe	13	
5	3	~~1~~ ~~3~~ 7	Kate	~~20~~ 1	
6	6	6	Mary	4	
7	~~7~~ 5	~~7~~ 3	Sam	6	

Property:
HA(AH(j) = j e.g. HA(AH(5) = 5
AH(HA(j) = j e.g. AH(HA(1) = 1



Continue to fix down 1. Update the heap.
To swap nodes 3 and 7 in the heap: HA[3]<->HA[7], and AH[HA[3]] <-> AH[HA[7]].

Removed, detailed slides

Heap Operations

- Initialization:
  - Given N-size array, **heapify** it.
  - Time: Θ(N). Good!
- Insertion of a new item:
  - Requires rearranging items, to maintain the **heap property**.
  - Time: O(lg N). Good!
- Deletion/removal of the largest element (max-heap):
  - Requires rearranging items, to maintain the **heap property**.
  - Time: O(lg N). Good!
- **Min-heap is similar.**

Heap

- Intuition
  - Lists and arrays: not fast enough => Try a tree ('fast' if 'balanced').
  - Want to remove the max fast => keep it in the root
  - Keep the tree balanced after insert and delete (to not degenerate to a list)
- Heap properties (when viewed as a tree):
  - Every node, N, is larger than or equal to any of his children (their keys).
    - => root has the largest key
  - Complete tree:
    - All levels are full except for possibly the last one
    - If the last level is not full, all nodes are leftmost (no 'holes').
    - $\Leftrightarrow$ stored in an array
- This tree can be represented by an array, A.
  - Root stored at index 1,
  - Node at index i has left child at 2i, right child at 2i+1 and parent at $\lfloor i/2 \rfloor$

Binary Heap:
Properties (Invariants)

A valid heap will always have these properties. (Also called invariants.)
They will be preserved even after delete and insert.

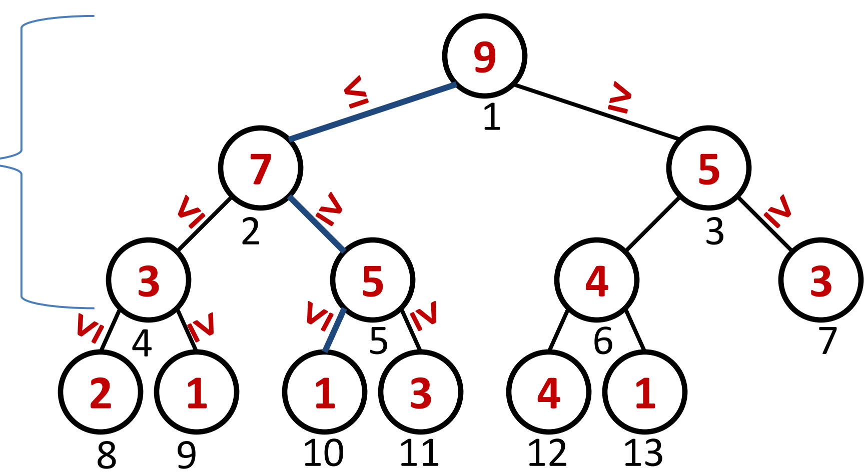
P1: Order: Every node, N, is larger than or equal to any of its children.

=> 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

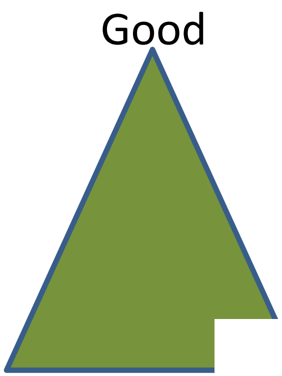
P2: Shape (**complete tree**)

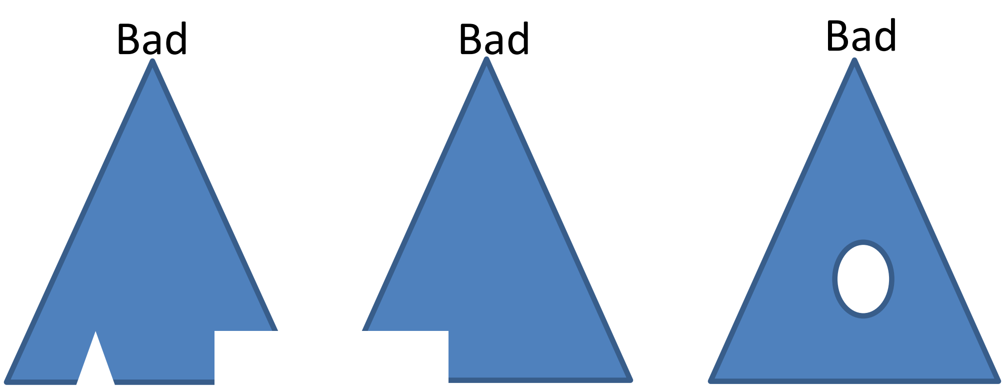
- When viewed as a tree:
- All levels are full except for possibly the last level.
  - => Heap height = $\lfloor \lg N \rfloor$
  - => If height h => $2^h \leq N \leq 2^{h+1}-1$
- On last level all nodes are leftmost:
- ⇔ array storage

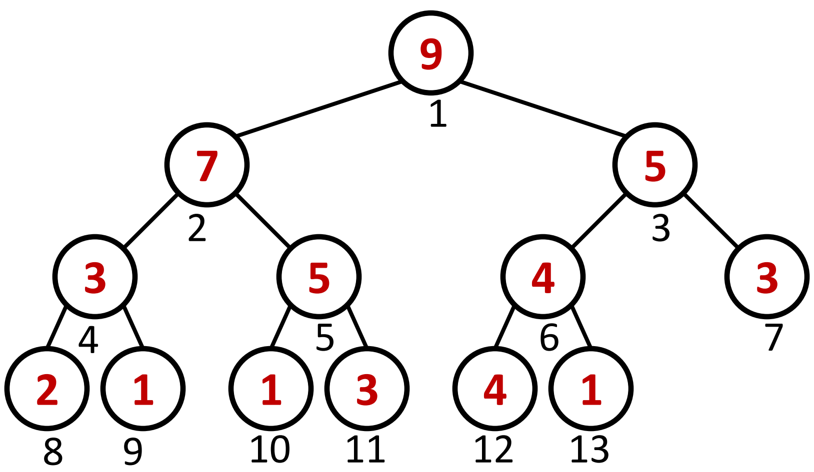


Heap – Shape Property

- A binary tree representing a heap has to be **complete**:
  - All levels are full, except possibly for the last level.
  - At the last level:
    - Nodes are placed on the left.
    - Empty positions are placed on the right.
  - There is “no hole”

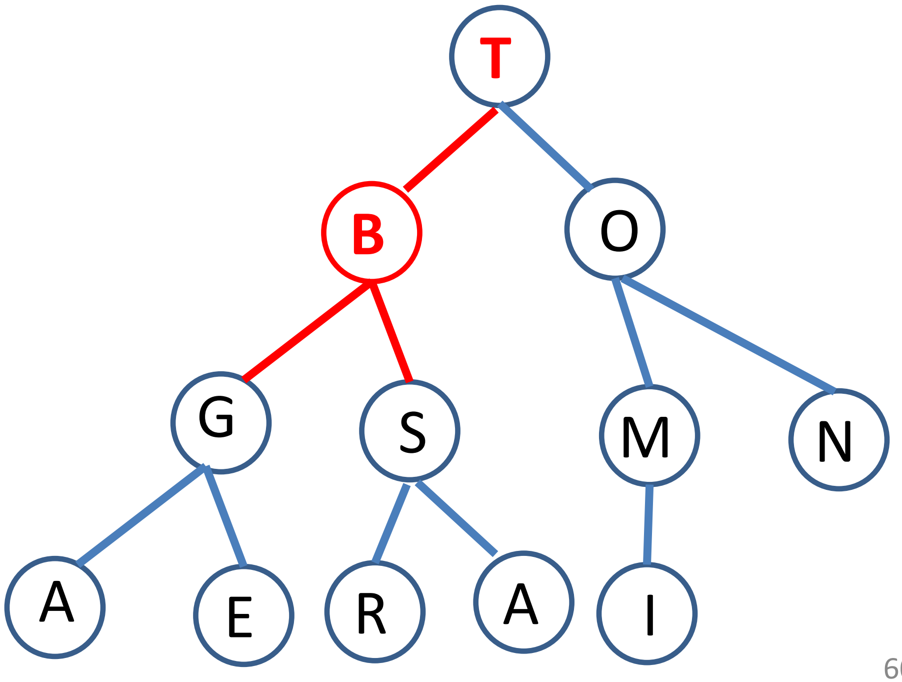






swimDown

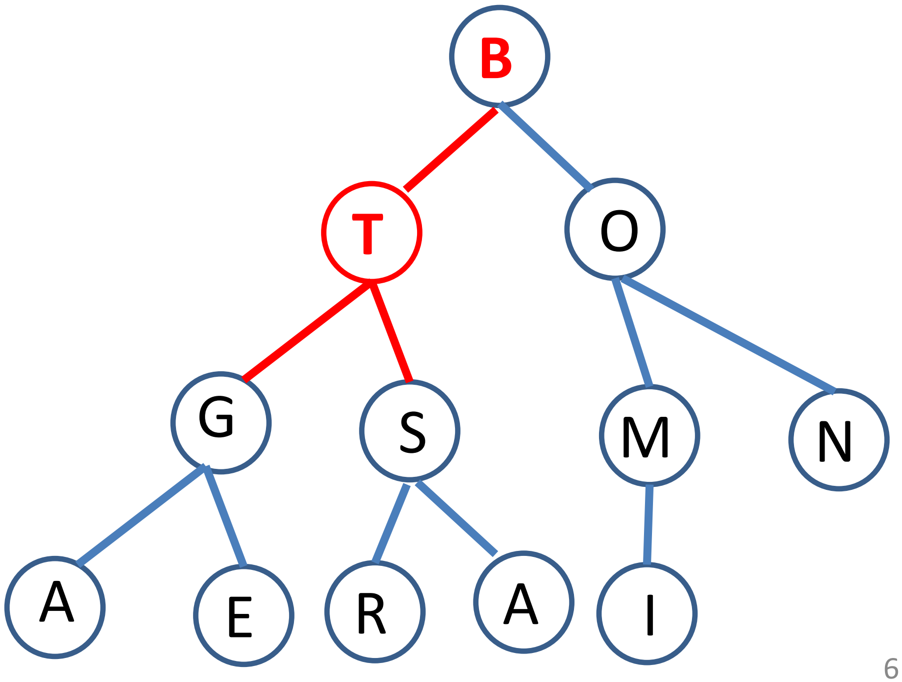
- B will move down until in a good position.
- Exchange B and T.



swimDown

- B will move down until in a good position.

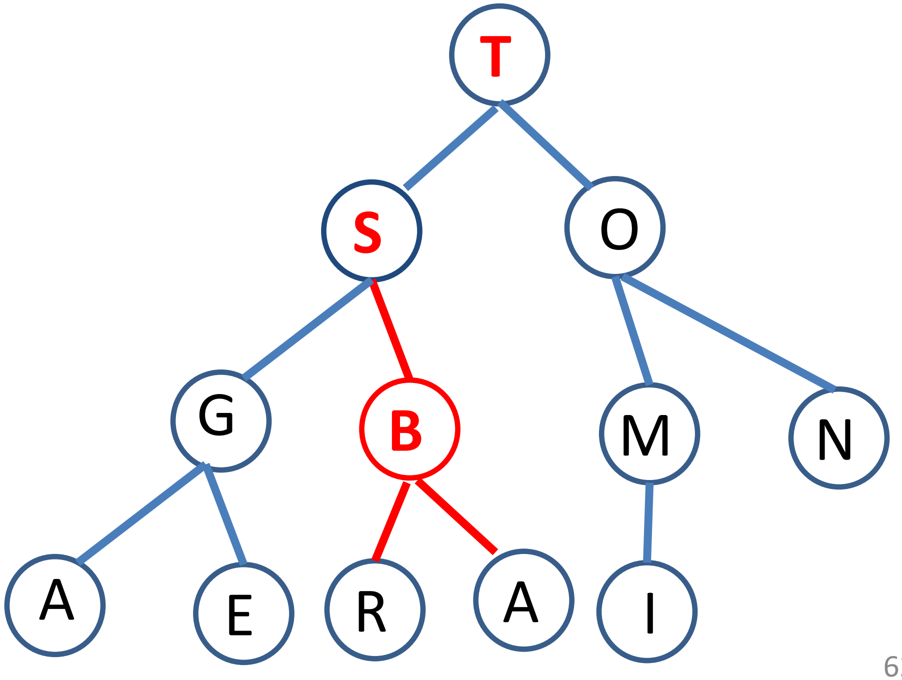
- Exchange B and T.
- Exchange B and S.



swimDown

- B will move down until in a good position.

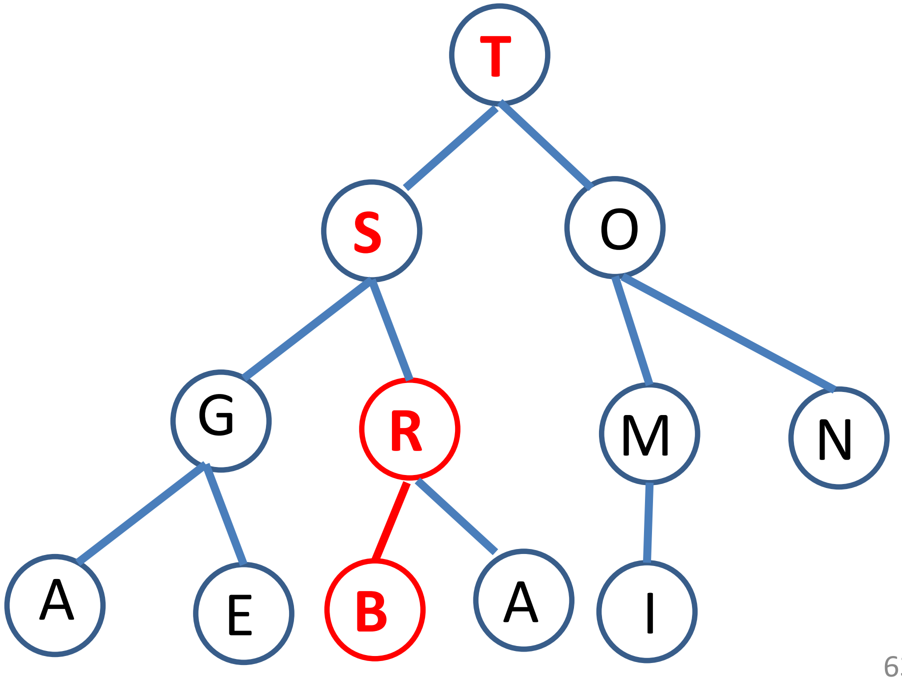
- Exchange B and T.
- Exchange B and S.
- Exchange B and R.



swimDown

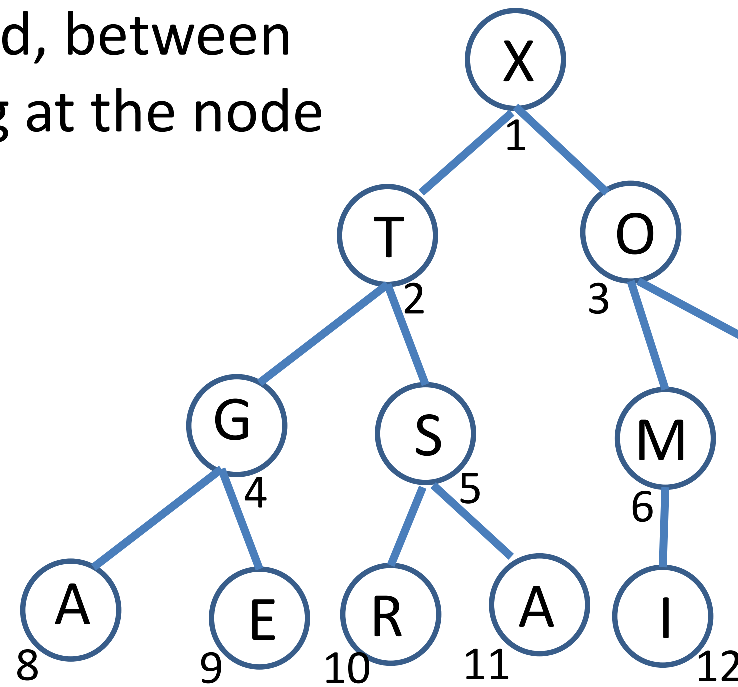
- B will move down until in a good position.

- Exchange B and T.
- Exchange B and S.
- Exchange B and R.



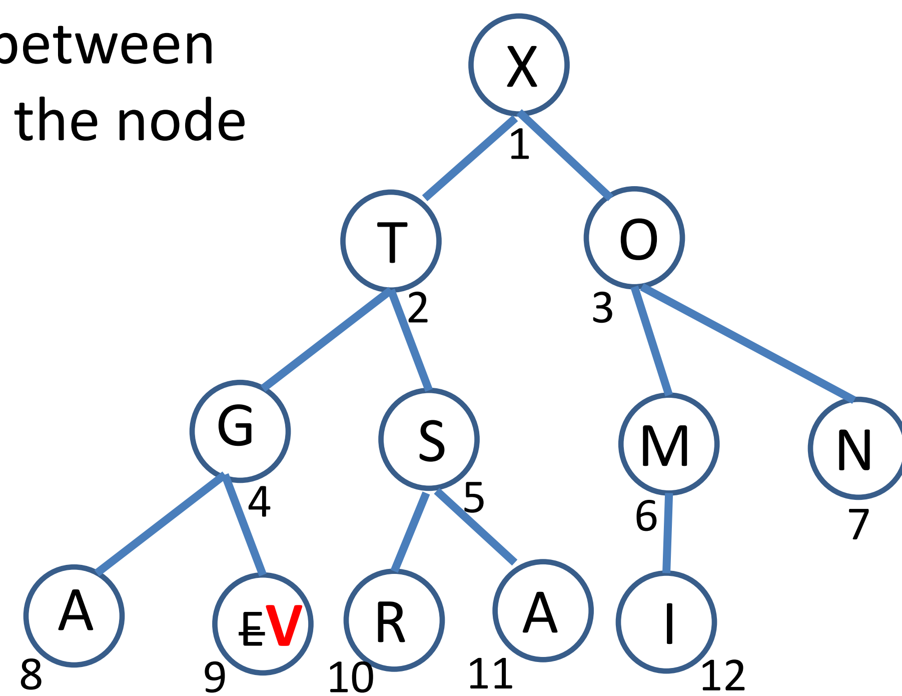
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.



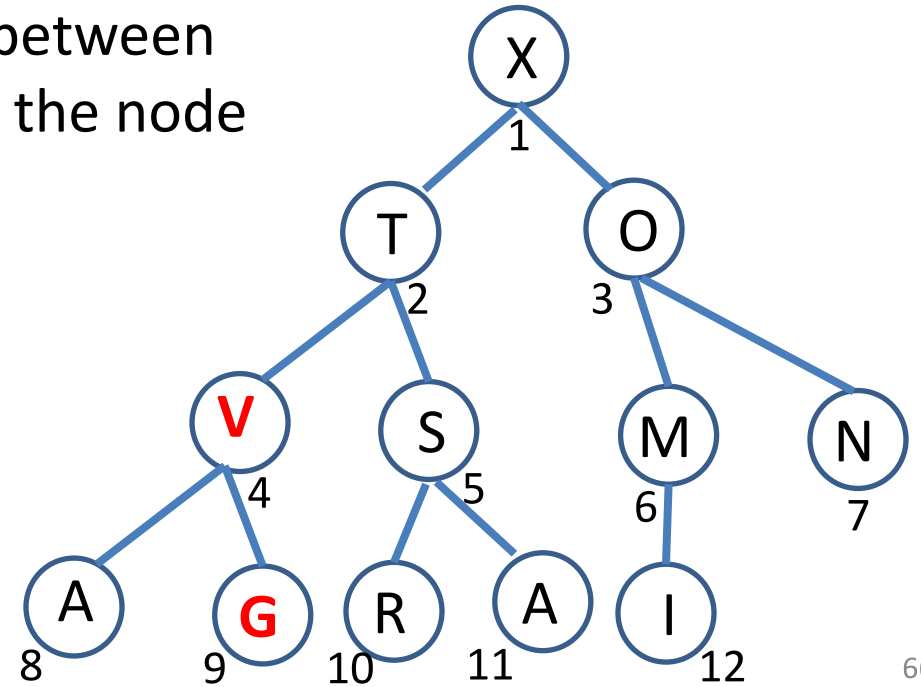
Increasing a Key

- Also called “increasing the priority” of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.



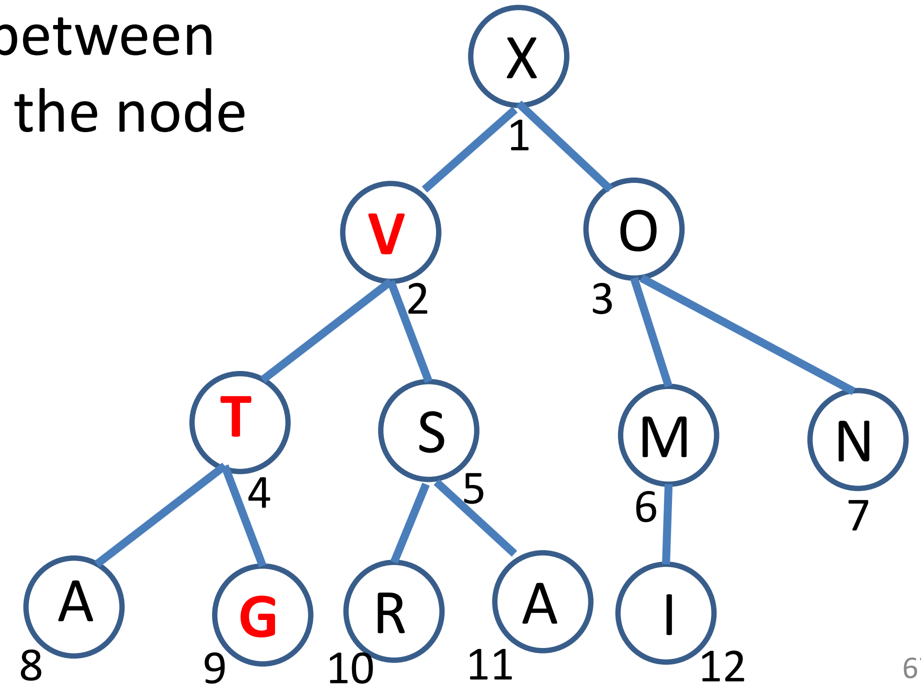
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.
  - Exchange V and G. Done?



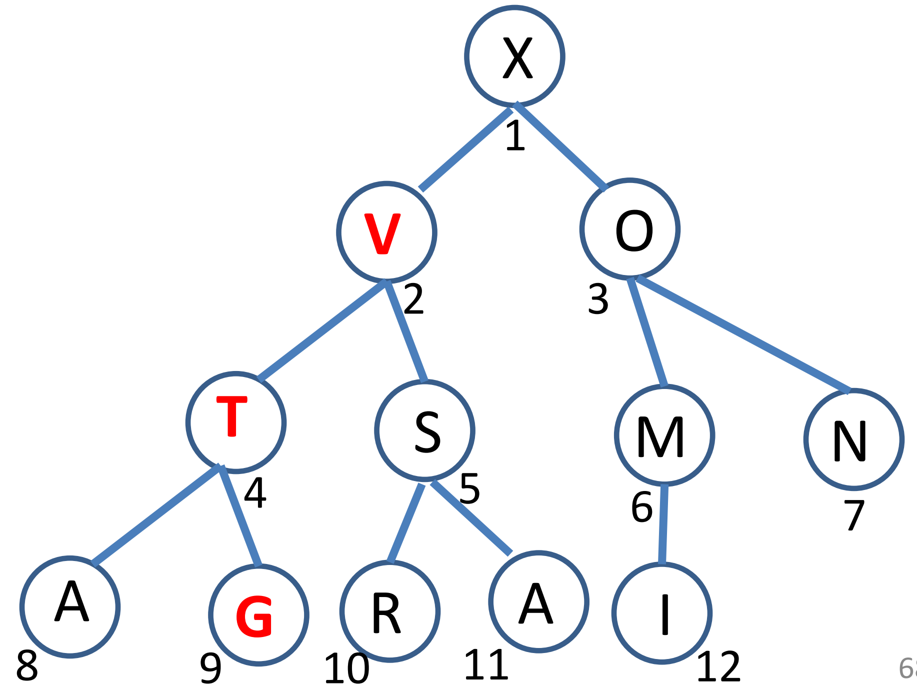
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.
  - Exchange V and G.
  - Exchange V and T. Done?



Increasing a Key

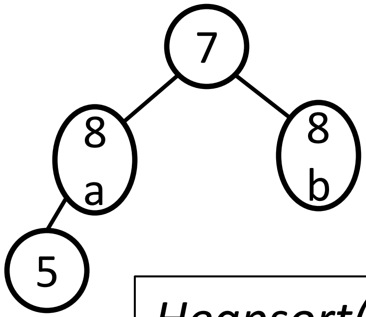
- Also called "increasing the priority" of an item.
- Can lead to violation of the heap property.
- ***Swim up*** to fix the heap:
  - While last modified node has priority larger than parent, swap it with his parent.
- Example:
  - An E changes to a V.
  - Exchange V and G.
  - Exchange V and T. Done.



Is Heapsort stable?

- If both children are the same value, and the parent value (9) needs to move down, the value from which child will be promoted (8a or 8b))?

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

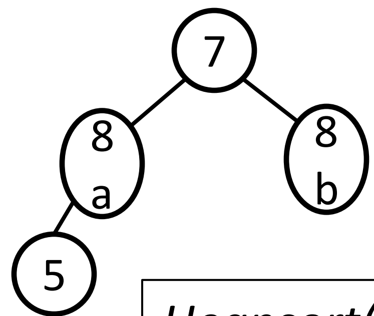



```
Heapsort(A,N)
1   buildMaxHeap(A,N)
2   for (p=(*N); p≥2; p--)
3       swap A[1] <-> A[p]
4       (* N)= (*N)-1
5       swimDown(A,p,N)
```

Is Heapsort stable? - NO

- If both children are the same value, and the parent value (9) needs to move down, the value from which child will be promoted (8a or 8b))? **Ans: left child**
- Both of these operations are unstable:
  - swimDown
  - Going from the built heap to the sorted array (remove max and put at the end)

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



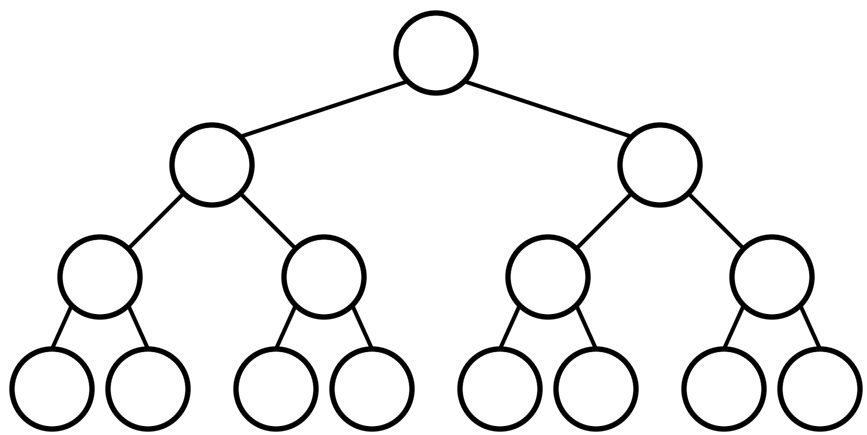

```
Heapsort(A,N)
   buildMaxHeap(A,N)
   for (p=(*N); p≥2; p--)
       swap A[1] <-> A[p]
       (* N)= (*N)-1
       swimDown(A,p,N)
```

Bottom-Up Batch Initialization

Turns array A into a heap in O(N).
(N = number of elements of A)

buildMaxHeap(A,N) *//Θ(N)*
*for (p = (*N)/2; p>=1; p--)*
    sinkDown(A,p,N)

Time complexity: O(N)
For explanation of this time complexity see extra materials at the end of slides.- Not required.



- See animation: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - Note that they do not highlight the node being processed, but directly the children of it as they are compared to find the larger one of them.