Priority Queues, Binary Heaps, and Heapsort

CSE 3318 – Algorithms and Data Structures

Alexandra Stefan
(includes slides from Vassilis Athitsos)
University of Texas at Arlington

Food for thought

- 23. Merge k Sorted Lists
- 88. Merge Sorted Array
- 1046. Last Stone Weight

- Find top-k largest items in an array

- Remove item with highest priority in a collection

Priority Queues

- Goal – to support operations:
  - **Delete/remove** the **max element**.
  - **Insert** a new element.
  - **Initialize** (organize a given set of items).
- PriorityQueue (Java, min-queue) , priority_queue (C++, max-queue)
- Min-priority Queues – easy implementation, or adapting existing ones.
- Applications:
  - Sorting
  - 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.
  - To solve other problems (see Top-k here and others on leetcode)
  - 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.)

Behavior of a max-priority queue

Insert in empty Max-PQ in this order:
5, 3, 9, 1, 2

Max-PQ	**operation**	**out**
5, 3, 9, 1, 2	remove() ->	9
5, 3, 1, 2 ,	remove() ->	5
3, 1, 2 ,	insert(7)	
7, 3, 1, 2,	remove() ->	7
3, 1, 2,	remove() ->	3
, 1, 2,	remove() ->	2
, 1,	remove() ->	1

Overview

- Priority queue
  - A data structure that allows inserting and deleting items.
  - *On remove, it removes the item with the HIGHEST priority*
    - *To remove the LOWEST just change the comparison function*
  - Implementations (supporting data structures)
    - Array (sorted/unsorted)
    - Linked list (sorted/unsorted)
    - **Heap – (an array with a special "order")**
      - Advanced heaps: Binomial heap, Fibonacci heap – not covered
- Binary Heap
  - Definition, properties,
  - Operations (each is O(lgN))
    - swimUp, sinkDown,
    - insert, remove, removeAny
  - Building a heap: bottom-up (O(N)) and top-down (O(NlgN))
- Heapsort – O(NlgN) time, O(1) space, https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - Not stable, not adaptive
- Finding top k: with Max-Heap and with Min-Heap
- Extra: Index items – the heap has the index of the element. Heap <-> Data

Priority Queue Implementations

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Unsorted	**1**	**2**	**7**	**5**	**3**	**5**	**4**	**3**	**1**	**1**	**9**	**3**	**4**			
Sorted	**1**	**1**	**1**	**2**	**3**	**3**	**3**	**4**	**4**	**5**	**5**	**7**	**9**			

How long will it take to remove MAX?

How long will it take to insert value 2? How about value 10?

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

Data structure	**Insert**	**Remove max**	**Create from batch of N**
Unsorted Array	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$ (if can use the original array)
Unsorted Linked List	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$ (if use the original linked list)
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)
Binary Heap (an array)	O(lgN) (reorganize)	O(lgN) (reorganize)	$\Theta(N)$
Special Heaps (Binomial heap, Fibonacci heap)			

Can you use a (balanced) tree to implement a priority queue (e..g. BST, 2-3-4 tree) ?

Review

- Complete tree and nearly complete tree
- Array traversal using
  - idx = idx/2   (TC?)
  - left = idx*2, right = idx*2+1

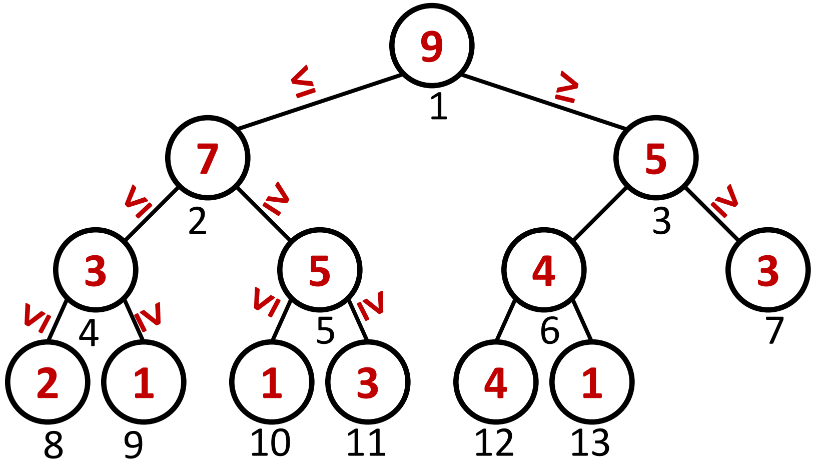
Binary Heap

Notes

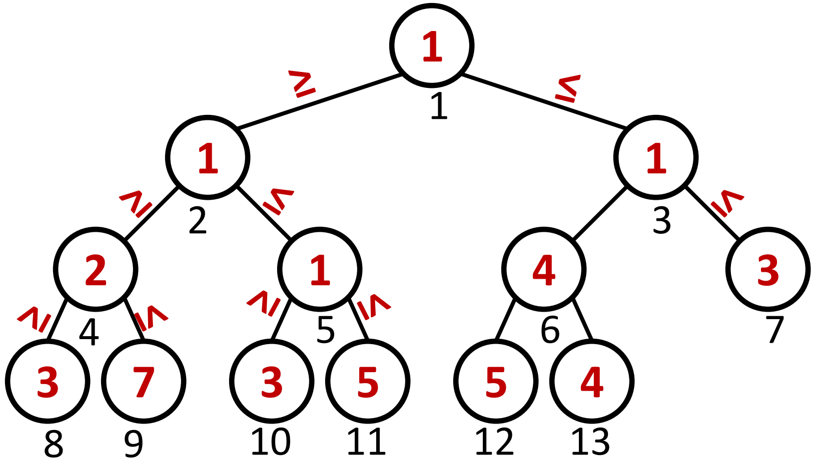
- **A binary heap is an array**
- We will view this array as a nearly complete tree
- The 1st element in the array is at index 1, not 0, in all the given code

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

Binary Max-Heap



Binary Min-Heap



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

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). It can start at index 0 as well.

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

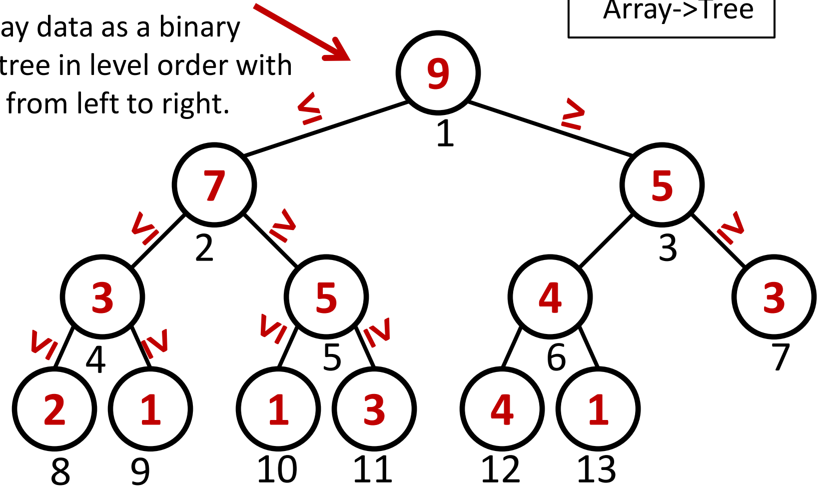
Index computation when 1st item is at index 1 (root is at index 1)

```
int left(int idx)     {return  idx*2     ;}
int right(int idx)    {return (idx*2)+1  ;}
int parent(int idx)   {return  idx/2     ;}
E.g.:
left(4)   -> ___
right(4)  -> ___
parent(4)-> ___

left(5)   -> ___
right(5)  -> ___
parent(5)-> ___
```

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.



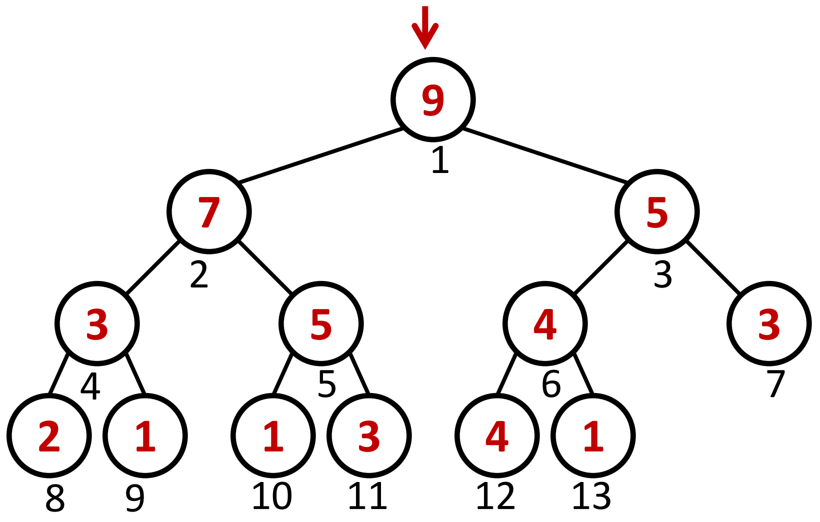
Index calculation

Index computation when 1st item is *at index 1 (root is at index 1)*

```
int left(int idx)     {return  idx*2     ;}
int right(int idx)    {return (idx*2)+1  ;}
int parent(int idx)   {return  idx/2     ;}
E.g.:
left(4)   -> ___
right(4)  -> ___
parent(4)-> ___

left(5)   -> ___
right(5)  -> ___
parent(5)-> ___
index of 1st item: _____
index of last item (based on size N): _____
```

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

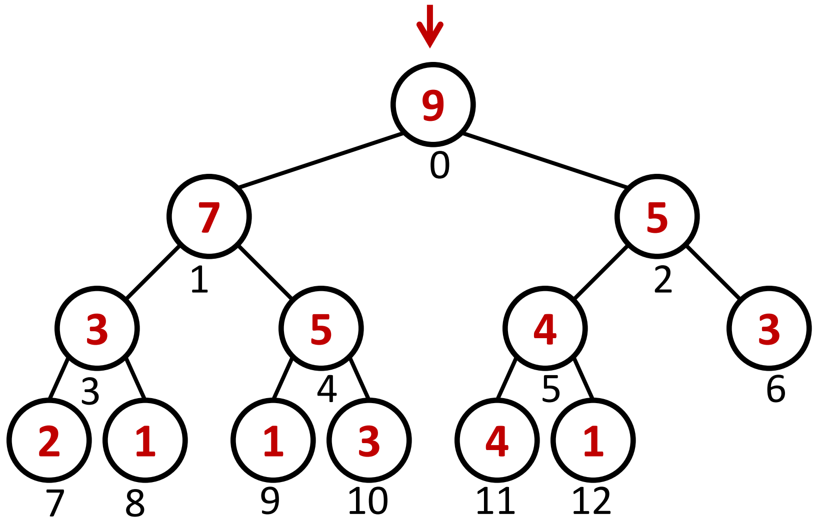


Index computation when 1st item is *at index 0 (root is at index 0)*

```
int left(int idx)     {return (idx*2)+1 ;}
int right(int idx)    {return (idx*2)+2 ;}
int parent(int idx)   {return (idx-1)/2 ;}
E.g.:
left(4)   -> ___
right(4)  -> ___
parent(4)-> ___

left(5)   -> ___
right(5)  -> ___
parent(5)-> ___
index of 1st item: _____
index of last item (based on size N): _____
```

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



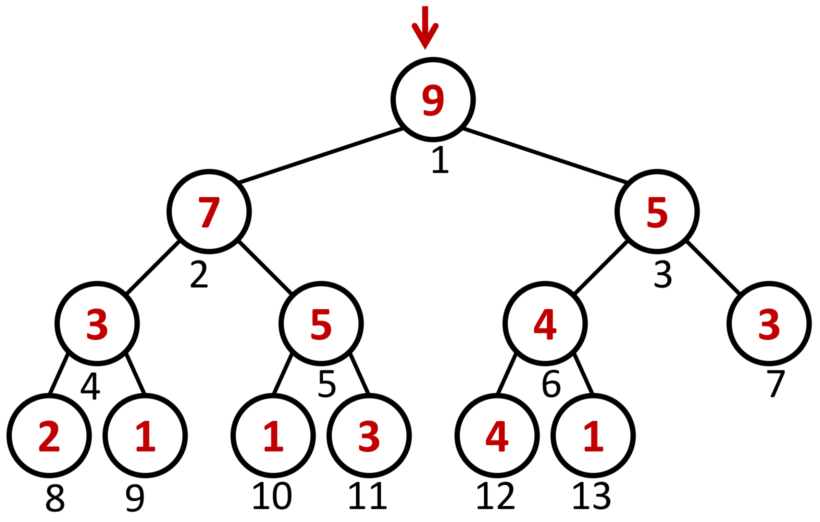
Index calculation

Index computation when 1st item is *at index 1 (root is at index 1)*

```
int left(int idx)     {return  idx*2     ;}
int right(int idx)    {return (idx*2)+1  ;}
int parent(int idx)   {return  idx/2     ;}
E.g.:
left(4)   ->  8
right(4)  ->  9
parent(4)->  2

left(5)   -> 10
right(5)  -> 11
parent(5)->  2
index of 1st item: 1
index of last item (based on size N): N
```

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



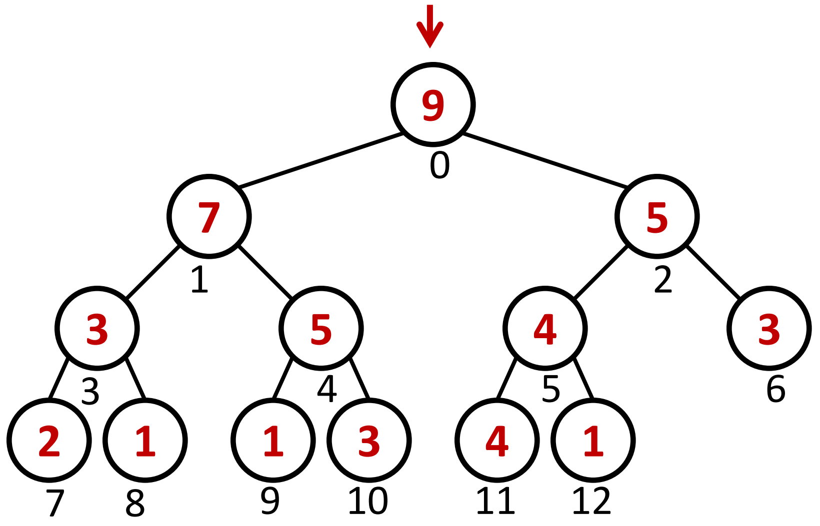
Index computation when 1st item *is at index 0 (root is at index 0)*

```
int left(int idx)     {return (idx*2)+1 ;}
int right(int idx)    {return (idx*2)+2 ;}
int parent(int idx)   {return (idx-1)/2 ;}
E.g.:
left(4)   ->  9
right(4)  ->  10
parent(4)->  1

left(5)   -> 11
right(5)  -> 12
parent(5)->  2
index of 1st item: 0
index of last item (based on size N): N-1
```

fixed

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



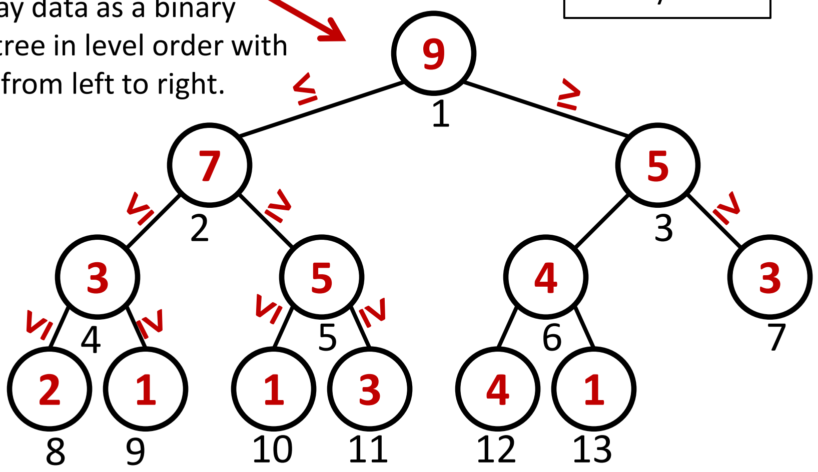
Binary Max-Heap: Stored as Array ⇔ Viewed as Tree

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

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

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.



Index computation when 1st item is at index 1

```
int left(int idx)    {return  idx*2    ;}
int right(int idx)   {return (idx*2)+1 ;}
int parent(int idx)  {return  idx/2    ;}
```

Heap properties:

P1: Order (heap): *The priority of every node is smaller than or equal to his parent's.*

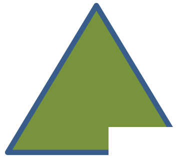
⇒ Max is in the root.

⇒ Any path from root to a node (and leaf) will go through nodes that have decreasing value/priority. E.g.: 9,7,5,1 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 in tree =>
$height, h = \lfloor lgN \rfloor$
leaves = $\lceil N/2 \rceil$

If tree height is h =>
The number of nodes in the tree is
$2^h \leq N \leq 2^{h+1}-1$

Heap – Shape Property - Nearly Complete Tree

P2: Shape (nearly complete tree: “no holes”) ⇔ array storage

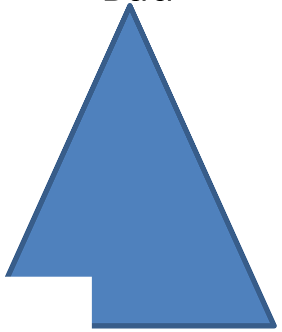
=> All levels are complete except, possibly, the last one.

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

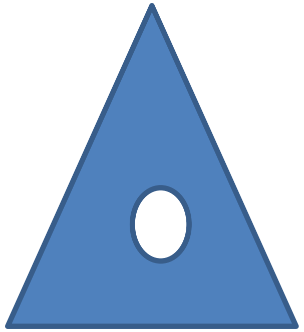
Good

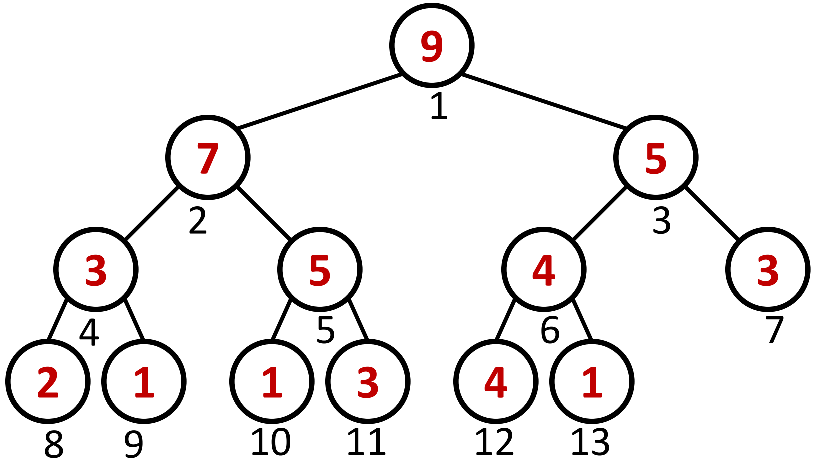
Bad

Bad



Bad





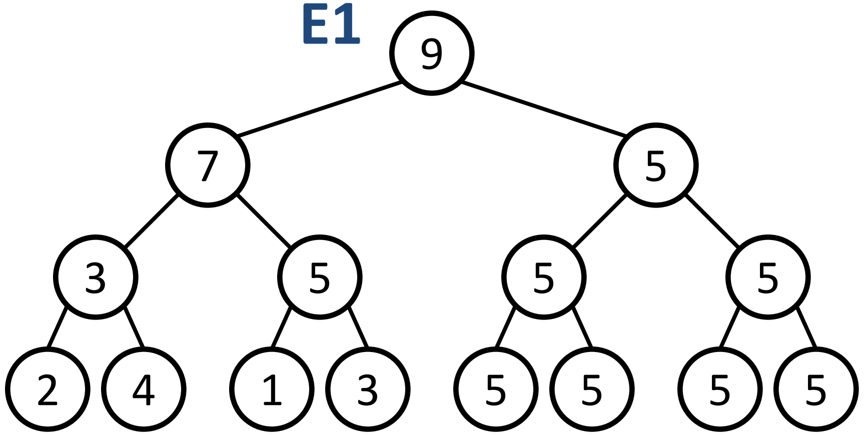
Heap Practice

For each tree, say if it is a **max-heap** or not. Check:

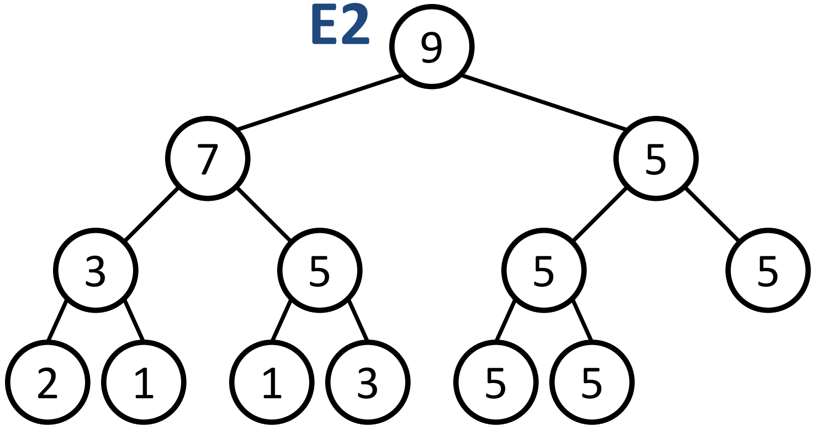
P1. Order

P2. Shape

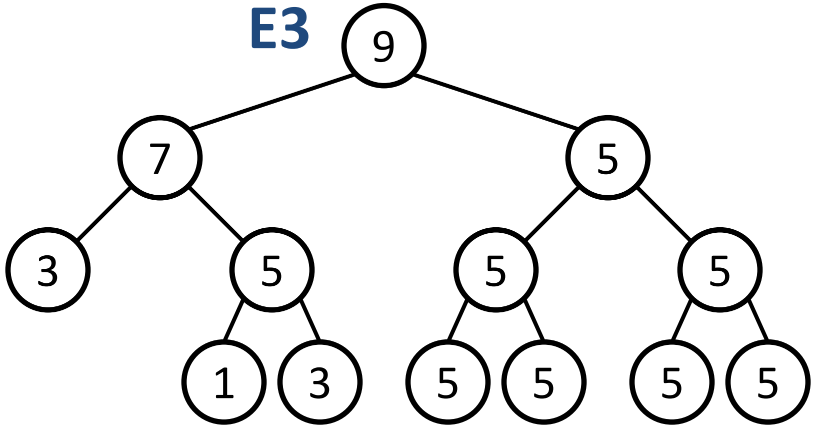
E1



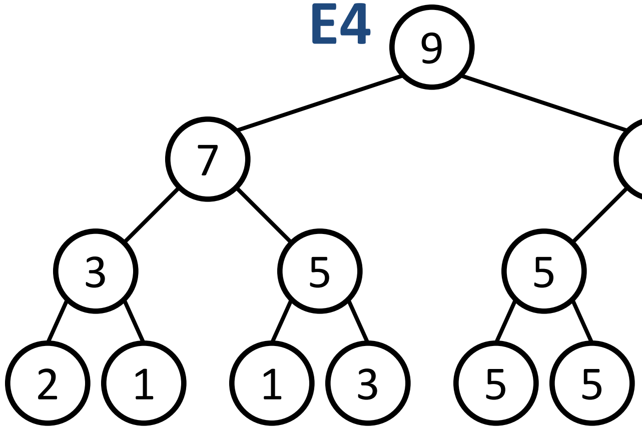
E2



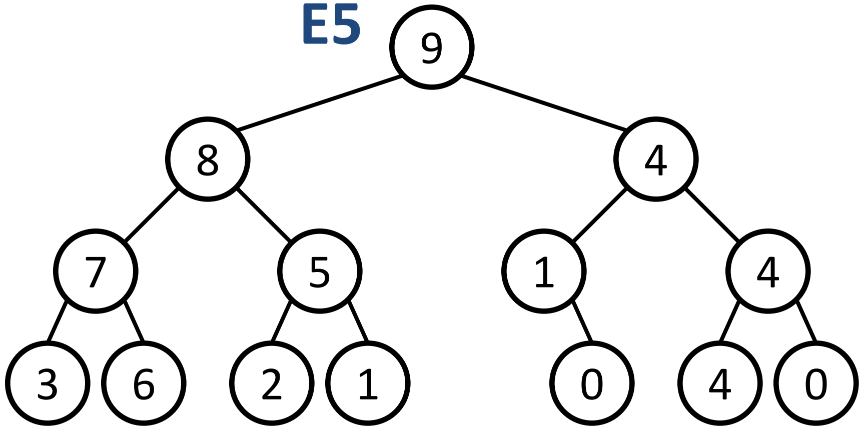
E3



E4



E5



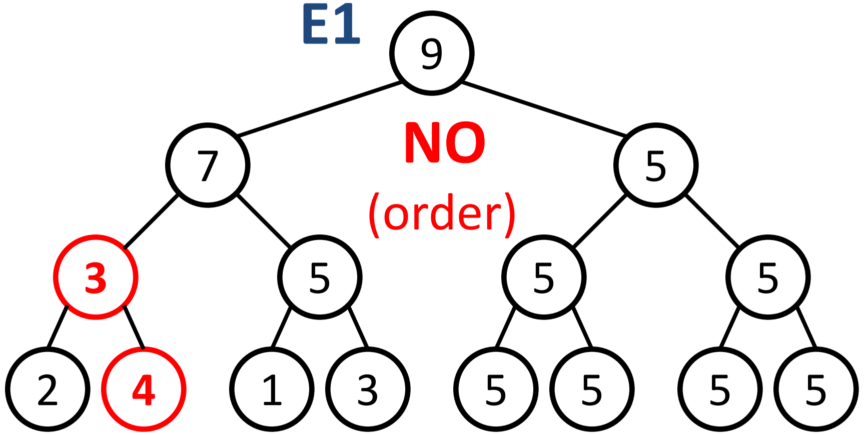
Answers

For each tree, say if it is a **max-heap** or not. Check:

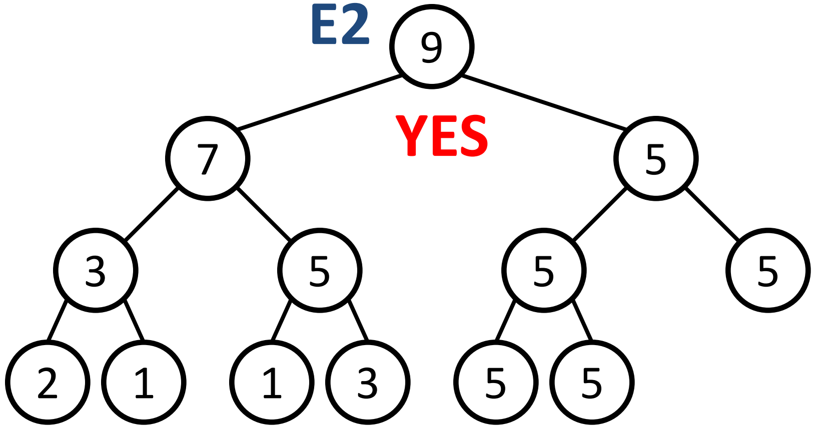
P1. Order

P2. Shape

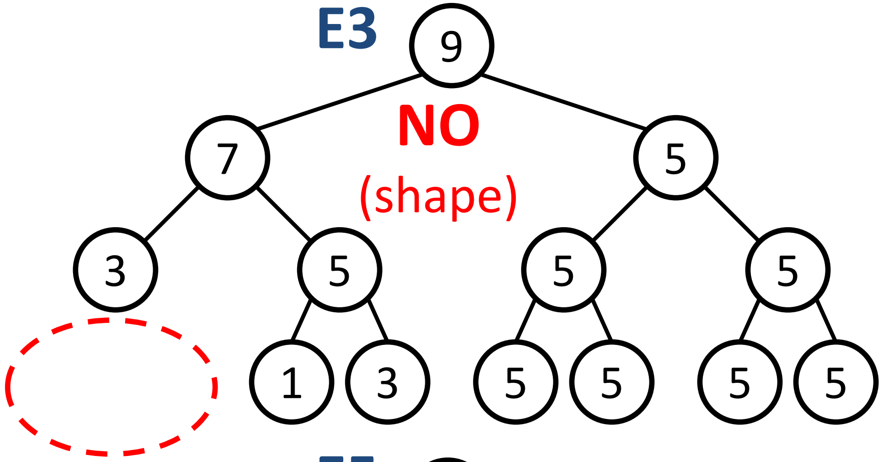
E1



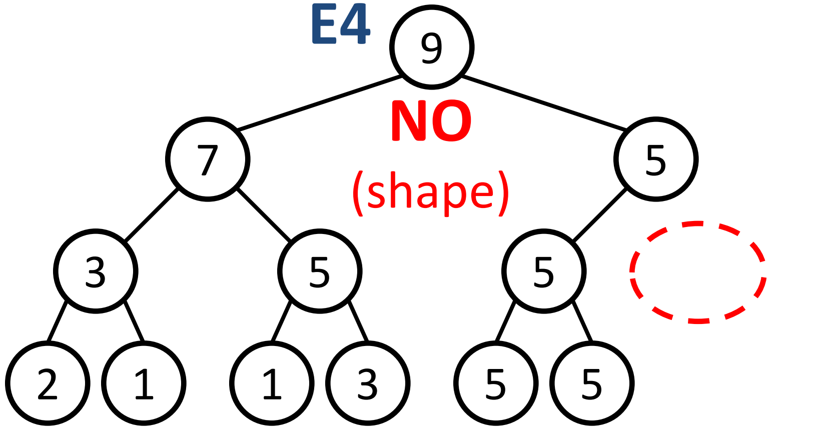
E2



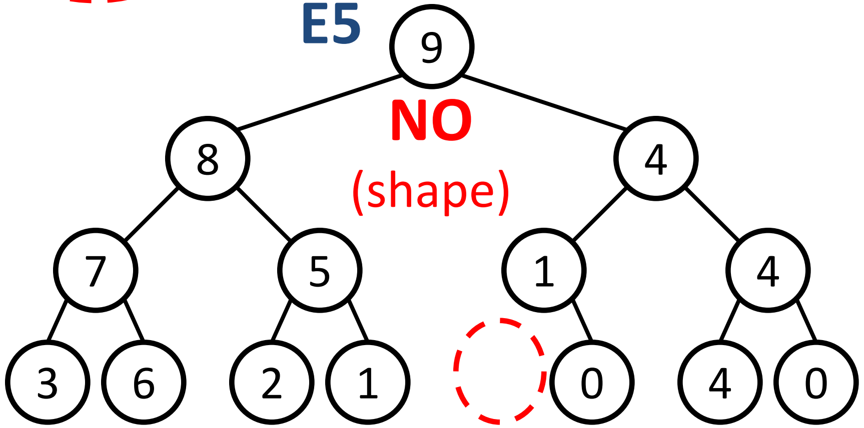
E3



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 Max-Priority Queues

Remember: N is both the size and the index of last item

`insert(int A[], int k, int * N)` – Inserts k in A. Modifies N.

`peek(int A[],int N)`

– Returns (but does not remove) the element of A with the largest key.

`remove(int A[],int * N)`

– Removes and returns the element of A with the largest (or smallest) key. Modifies N.

`increase(int A[], int p, int k)`

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

`removeAny(int A[], int p, int * N)`

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

`decrease(int A[], int p, int k, int 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 V.

- Can lead to violation of the heap property.

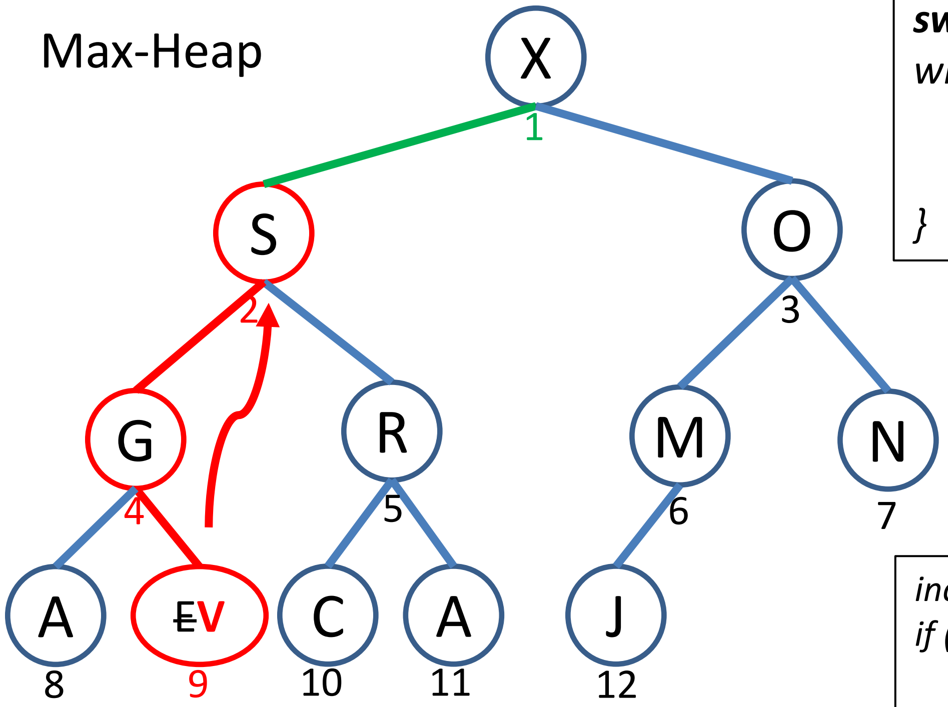
swimUp V to fix the heap:

Idea: While last modified node is not the root AND it has priority larger than its parent, swap it with his parent and the parent becomes the last modified node.

- 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

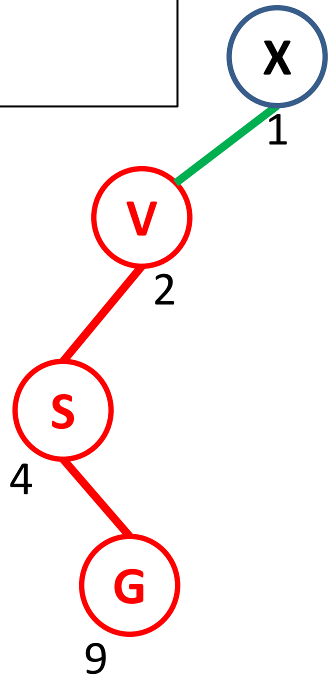
Letters in alphabetical order:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Max-Heap




```
swimUp(int* A, int idx)  //O(lg(N))
while ( (idx>1) && (A[idx]>A[parent(idx)] ){
   swap: A[idx] ,  A[parent(idx)]
   idx = parent(idx)
}
```

O(lg(N)) TC
b.c. only the red links are explored)




```
increase(int* A, int idx, int k) //O(lgN)
if (A[idx]<k)  {
    A[idx]=k
    swimUp(A,idx)
}
// Else reject operation
```

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		
Original	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**C**	**A**	**J**		

Insert **V** in this heap.

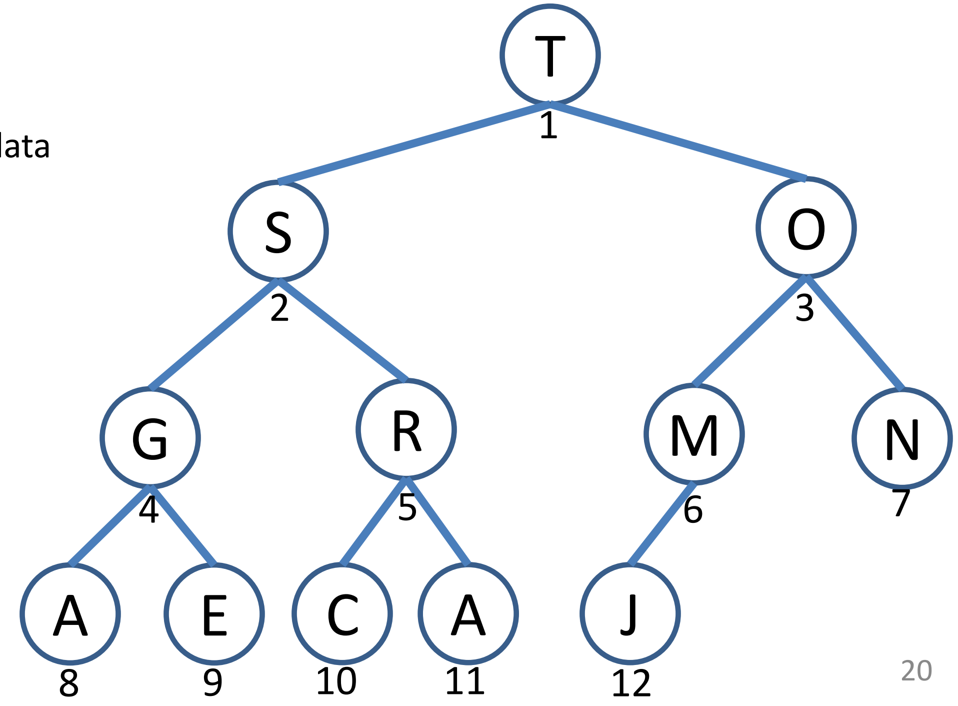
- This is a heap with 12 items.

- How will a heap with 13 items look? (What shape?)

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

Time complexity? Best: Worst: General:

```
insert(int* A, int newKey, int* N)
    (*N) = (*N)+1 // permanent change
    idx  = (*N)       // index of increased node
    A[idx] = newKey
    swimUp(A,idx)
```



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	J		
Increase and Put V	T	S	O	G	R	M	N	A	E	B	A	J	V	
1st iter	T	S	O	G	R	V	N	A	E	B	A	J	M	
2nd iter	T	S	V	G	R	O	N	A	E	B	A	J	M	
3rd iter,Final	V	S	T	G	R	O	N	A	E	B	A	J	M	

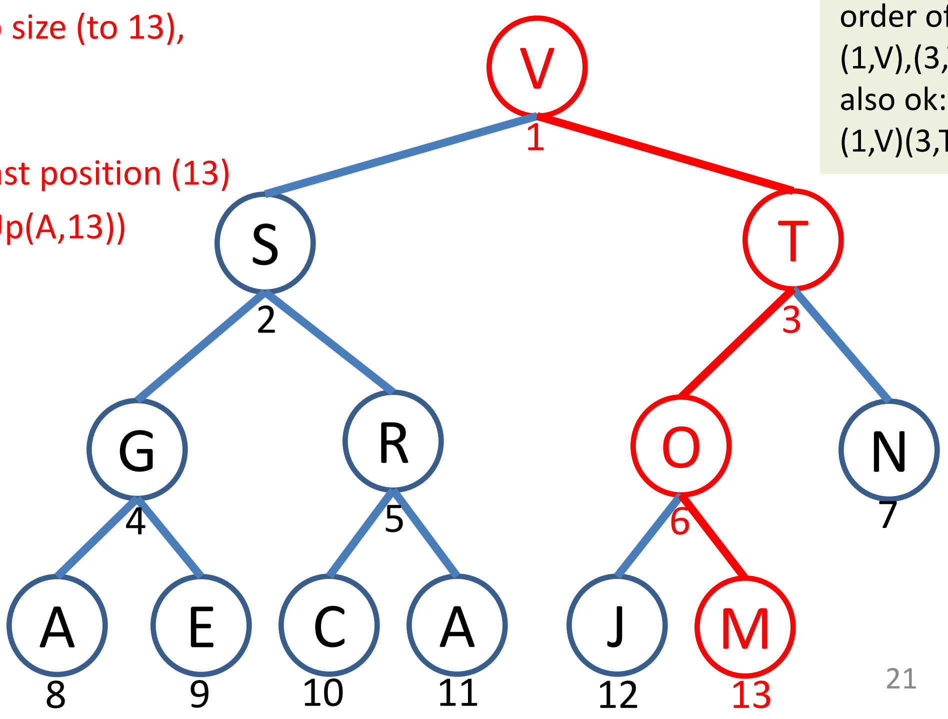
```
insert(int* A, int newKey, int* N)
   (*N) = (*N)+1 // permanent change
   //same as increaseKey:
   idx = (*N)
   A[idx] = newKey
   swimUp(A,idx)
```

Increase heap size (to 13),

Put V in the last position (13)

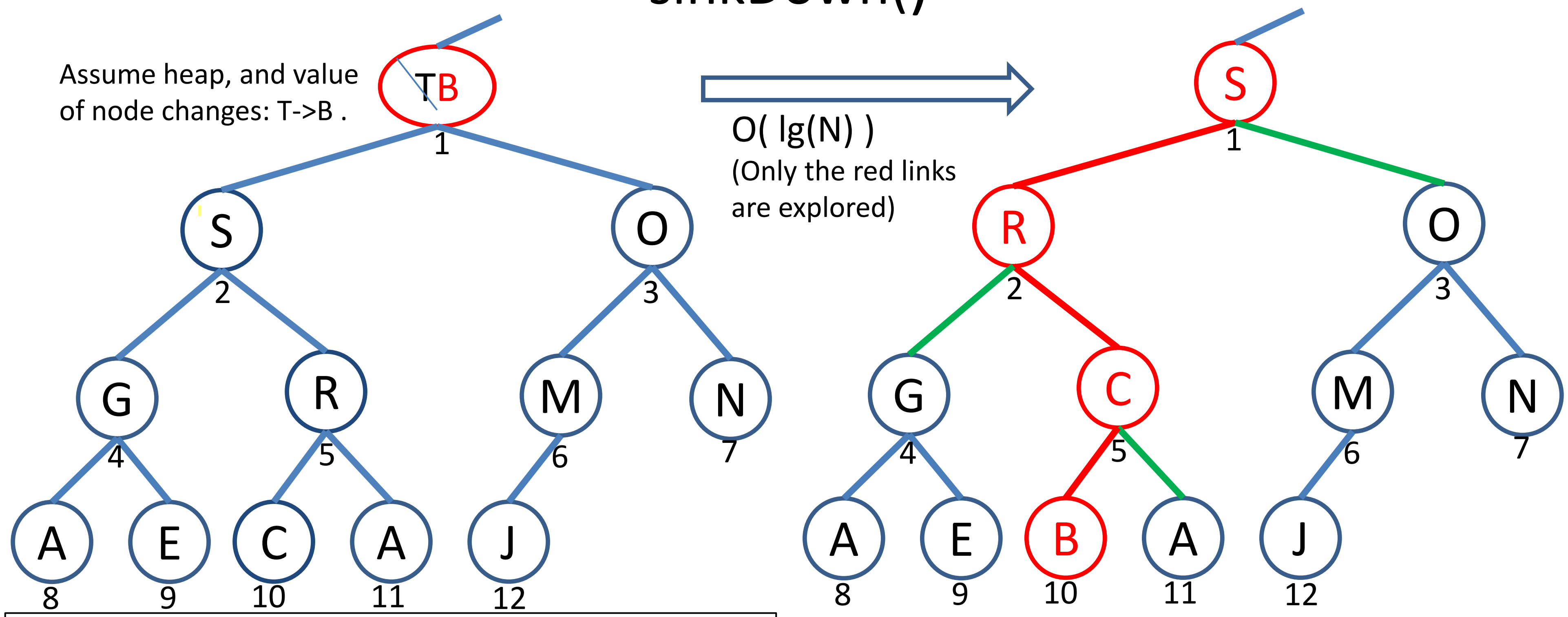
Fix up (swimUp(A,13))

Canvas format for the answer: (index, value) of updated nodes in result heap, listed in increasing order of indexes:
(1,V),(3,T),(6,O),(13,M)
also ok:
(1,V)(3,T)(6,O)(13,M)



Case	Discussion	Time complexity	Example
Best	1	Θ(1)	insert B (not V)
Worst	Height of heap	Θ(lgN)	Shown here
General		***O(lgN)***	

sinkDown()



Assume node p is smaller than one (or both) of his children, AND ***the subtrees rooted at the children are heaps***.

Make the entire tree rooted at p be a heap:

- Repeatedly exchange items as needed, between a 'bad' node and his **<u>largest</u>** child, starting at p.
- Stop when in good place (parent is larger than both children) or it has no children

Applications/Usage:

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

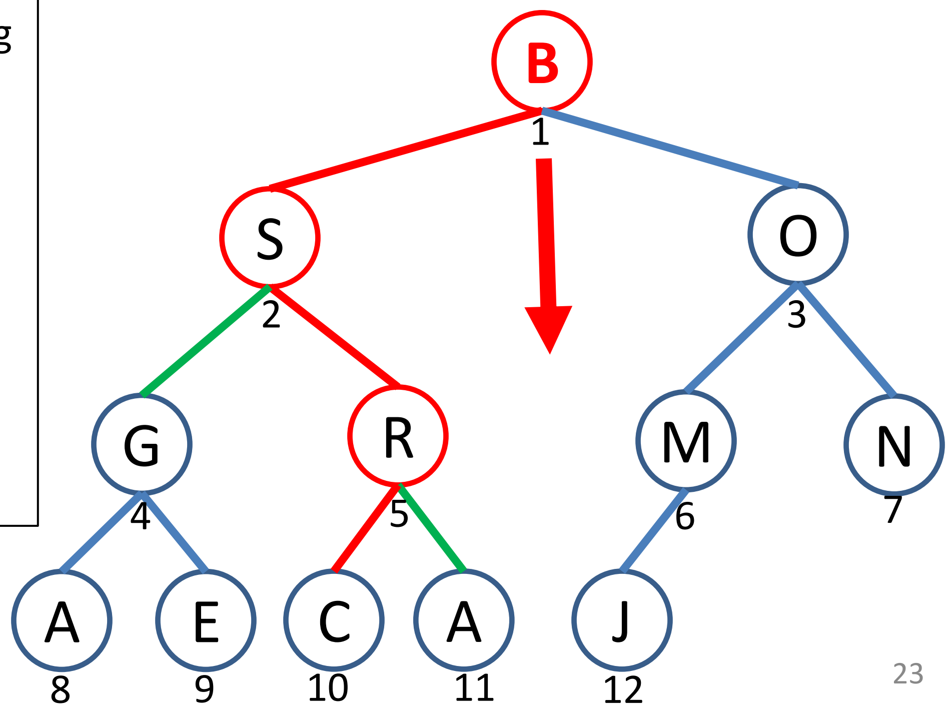
sinkDown(A,p,N)

Decrease key

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

- Makes the tree rooted at index 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.: T(root value) is decreased to B.
- B will move down until in a good position.
  - S>O && S>B => S <-> B
  - R>G && R>B => R <-> B
  - C>A && C>B => C <-> B
  - No left or right children (or) => stop

```
// push down DATA from node at index p if needed
sinkDown(int* A, int p, int N)    - O(lgN)
le = left(p)        // index of left child of p
ri = right(p) // index of right child of p
imv = idxOfMaxValue(A,p,le,ri,N)
if (imv != p) {
   swap  A[imv] ,  A[p]
   sinkDown(A, imv, N)
}
//idxOfMaxValue MUST check that left and right are valid indexes
```



sinkDown(A,p,N)

idxOfMaxValue code

Code tracing

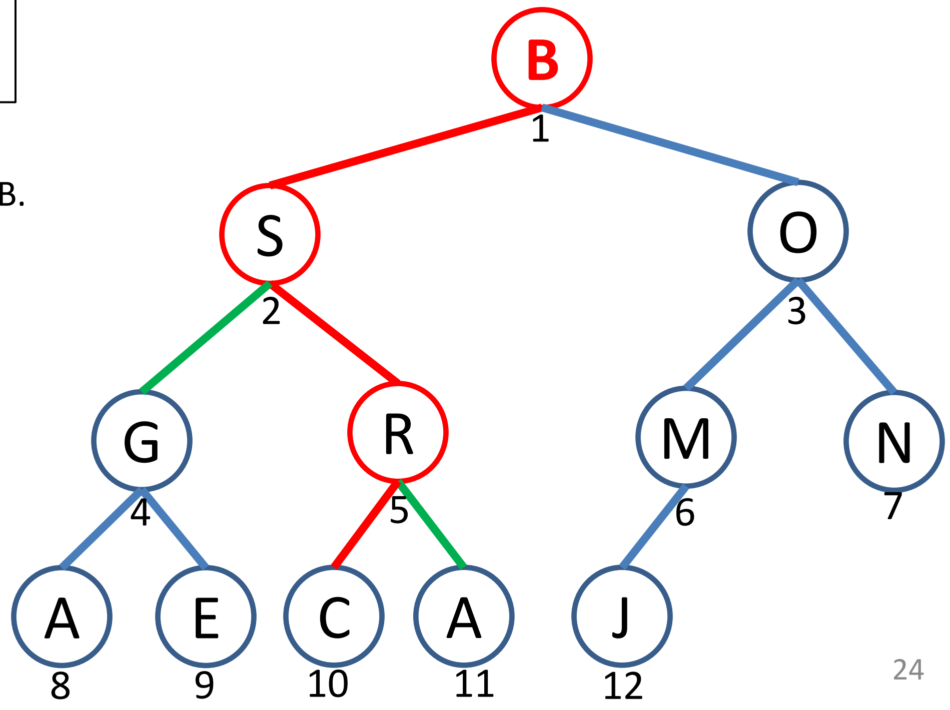
```
// idxOfMaxValue MUST check valid indexes le<=N and ri<=N
int idxOfMaxValue(int* A,int p,int le,int ri,int N){
    int imv=p; // so far p is the index of the largest value

    // there is a left child and he is bigger than the parent
    if ((le≤N)&&(A[le]>A[imv]))
        imv = le;   // set imv to index of left child

    // There is a right child and it is bigger than max value seen
    if ((ri≤N)&&(A[ri]>A[imv]))
        imv = ri;  // set imv to index of left child
    return imv;
}
```

```
sinkDown(A,p,N)      - O(lgN)
le = left(p)   // left child of p
ri = right(p)  // right child of p
imv = idxOfMaxValue(A,p,le,ri,N);
if(imv != p){
    swap A[imv] , A[p]
    sinkDown(A,imv,N)
}
//idxOfMaxValue MUST check that left and right are valid indexes
```

Trace the code for sinkDown(A,1,12) , i.e. N=12, p=1 and A[1] is B.



Remove

N is 12

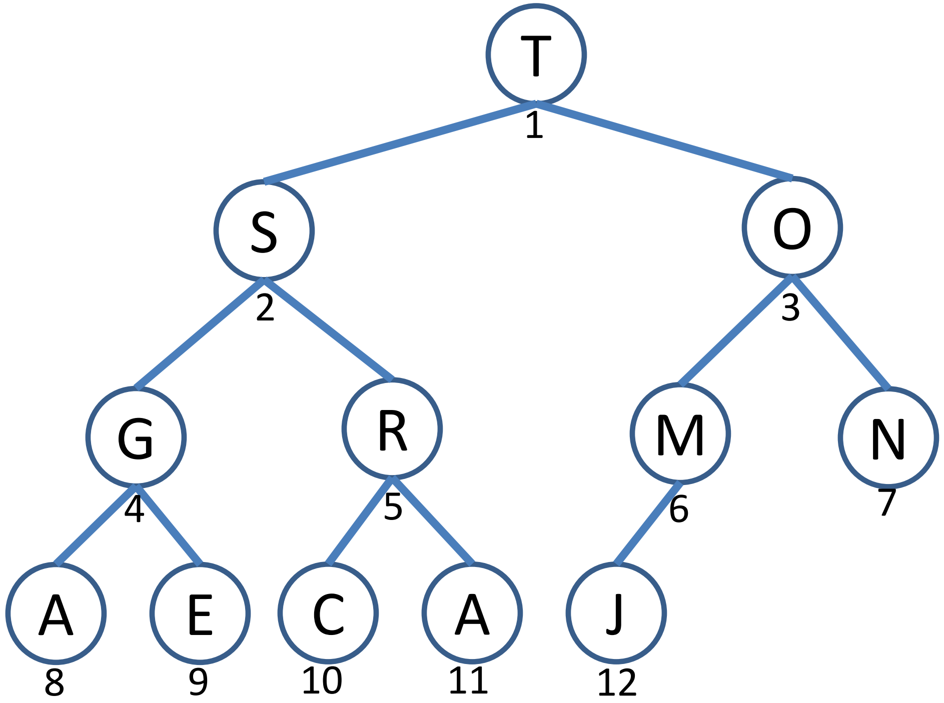
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**C**	**A**	**J**

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

N is 12

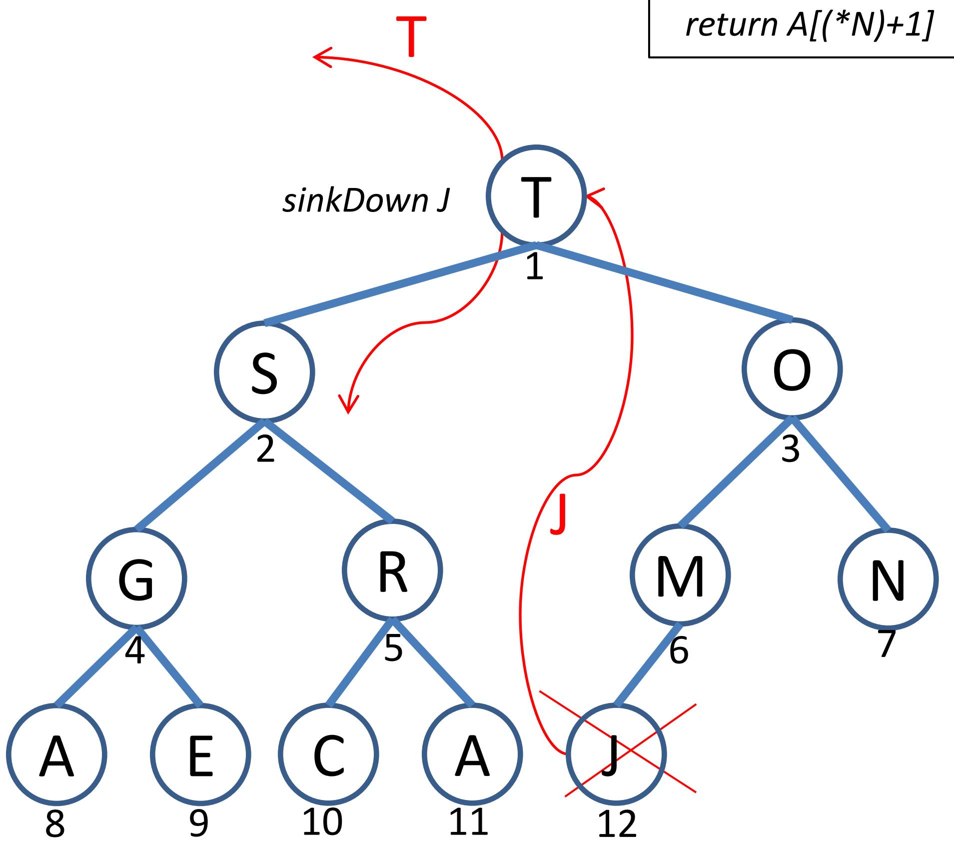
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**C**	**A**	**J**

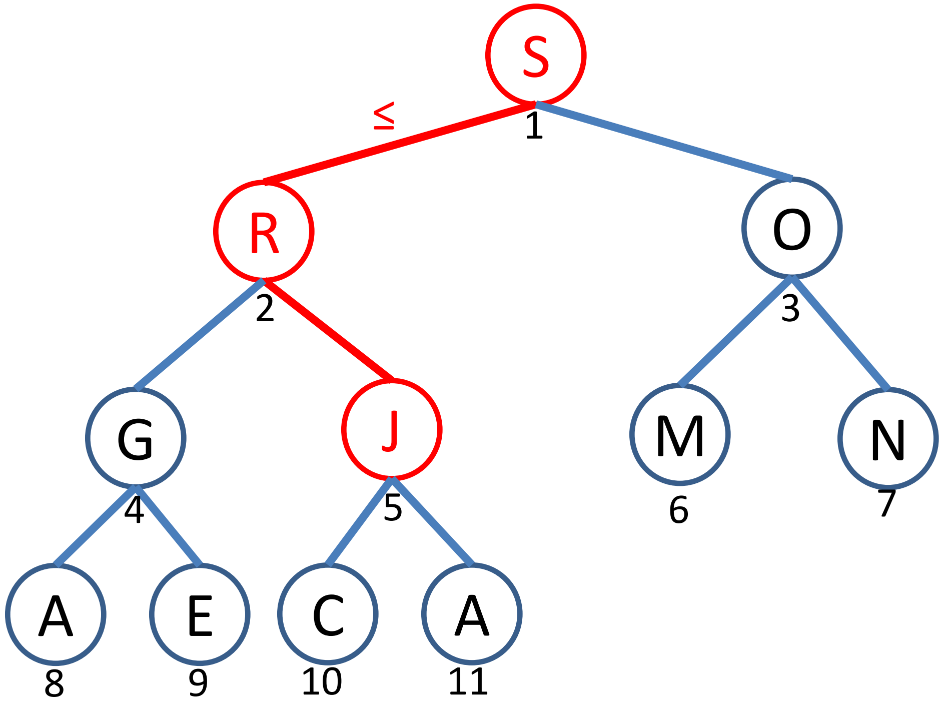
N is 11

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

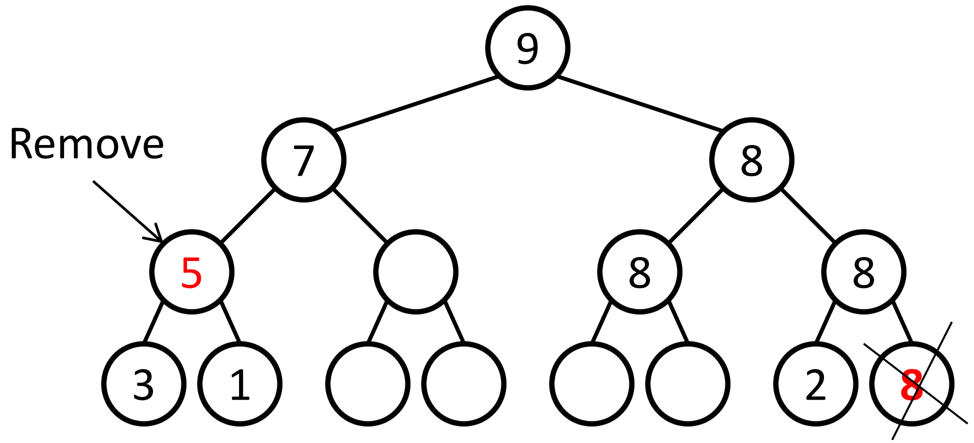
remove(int A, int* N) // O(lgN)*
*swap A[1] and A[(*N)]*
*(*N)=(*N)-1 //permanent*
*sinkDown(A,1, *N)*
*return A[(*N)+1]*

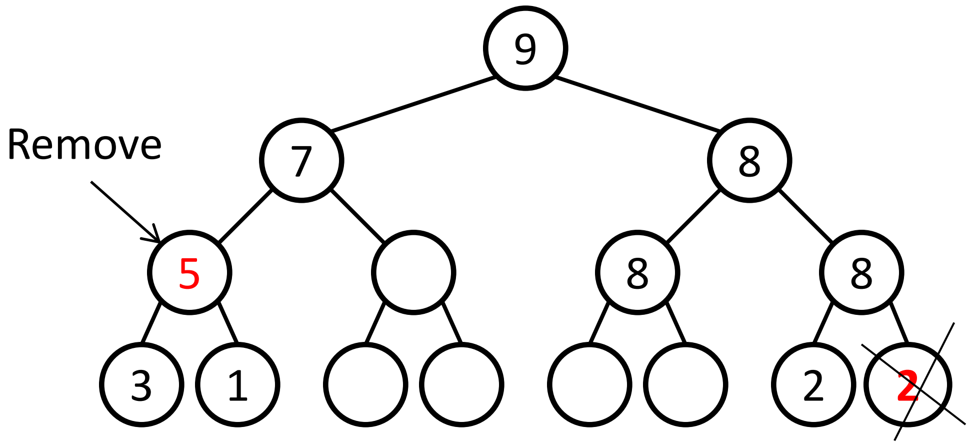
Case	Discussion	Time Complexity	Example
Best	1	Θ(1)	All items have the same value
Worst	Height of heap	Θ(lgN)	Content of last node was A
General	1<=…<=lgN	***O(lgN)***	





Removal of a Non-Root Node





Give examples where new priority is:
- Increased
- Decreased

```
removeAny(int* A, int p, int* N)    // O(lgN)
   temp = A[p]
   A[p] = A[(*N)]
   (*N)=(*N)-1 //permanent
  //Fix H at index p
  if (A[p] > A[parent(p)])
              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)
  - Space: O(1)

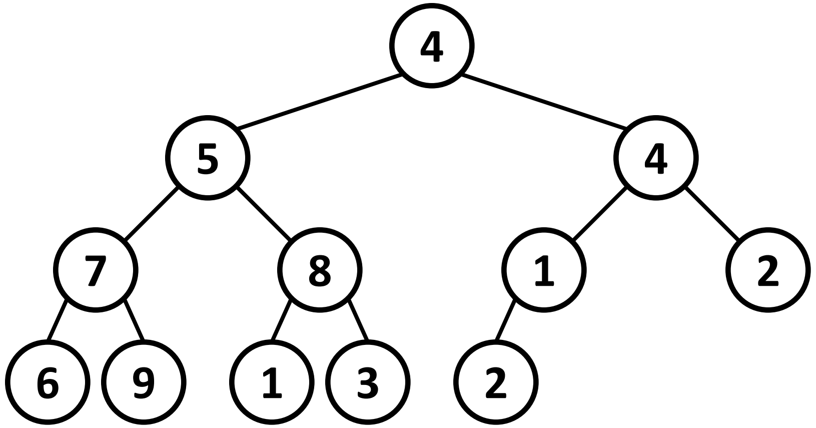
- Deletion:
  - Will always remove the largest element. This element is always at the top of the heap (the first element of the heap).
  - Deletion of the maximum element:
    - Exchange the first and last elements of the heap.
    - Decrement heap size.
    - Fix down to restore the heap property.
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

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.
  - To work in place, we would need to start at index 0. Here I still use a heap that starts at index 1 (for consistency with the other slides), but in reality, if the array is full, we cannot make the cell at index 0 empty.

Batch Initialization Method	Time	Extra space (in addition to the space of the input array)
Bottom-up (fix the given array)	Θ (N)	Θ(1)
Top-down (start with empty heap and insert items one-by-one)	O(N lg N)	Θ(1) (if "insert" in the same array: heap grows, original array gets smaller) Θ(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)

```
//Assumes data in A starts at index 1 (not 0)
buildMaxHeap(int* A, int 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 build - Step-by-step

Space: O(1)

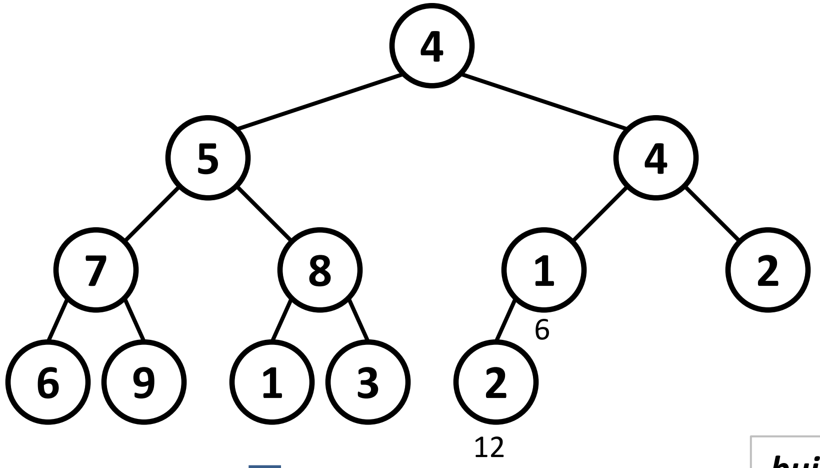
Time complexity: O(N) - Intuition: the bigger the height of the heap to fix, the SMALLER the number of heaps of that height that need to be fixed.

When fixing one heap (by hand,on paper), apply swimDown correctly: swap as long as needed (not just one level)

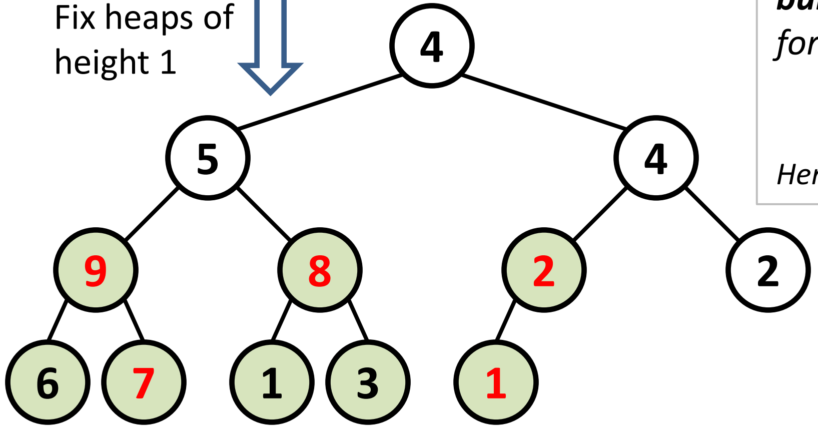
buildMaxHeap(int* A, int N) *//Θ(N) // Assumes data in A starts at 1*
for (p = N/2; p>=1; p--) // start from parent of last node, stop at root
    sinkDown(A,p,N) // makes heap at p if left and right are heaps

Here: last node is at index N=12, its parent is at index p=N/2 =12/2 = 6 => start from node at index 6

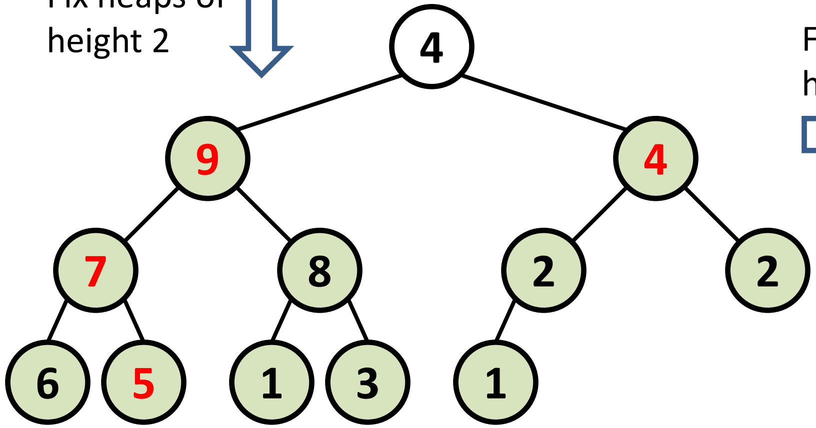
	0	1	2	3	4	5	6	7	8	9	10	11	12
	-												



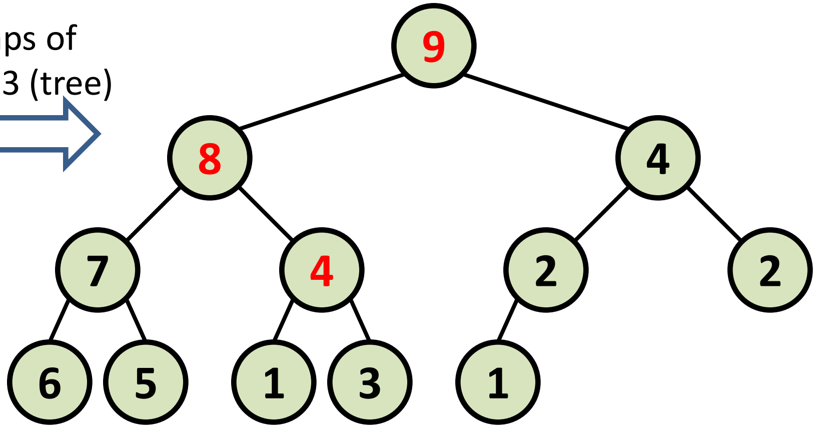
Fix heaps of height 1



Fix heaps of height 2

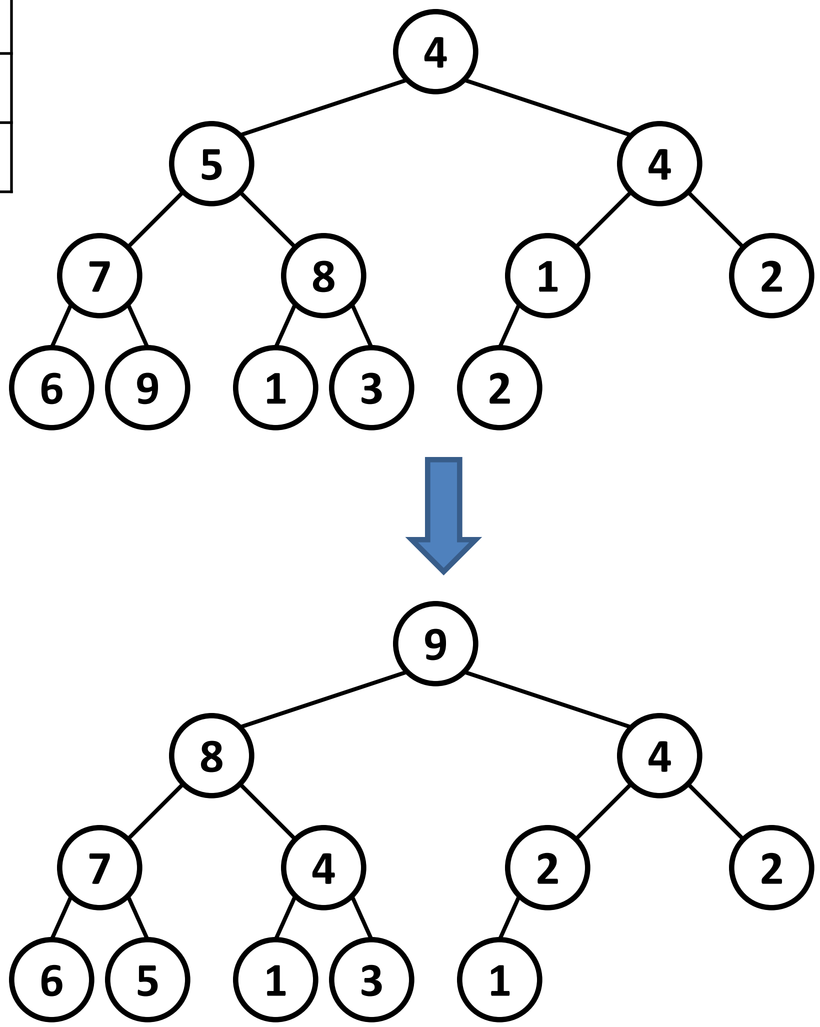


Fix heaps of height 3 (tree)



Bottom-Up Batch Initialization

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



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

buildMaxHeap(int* A, int 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

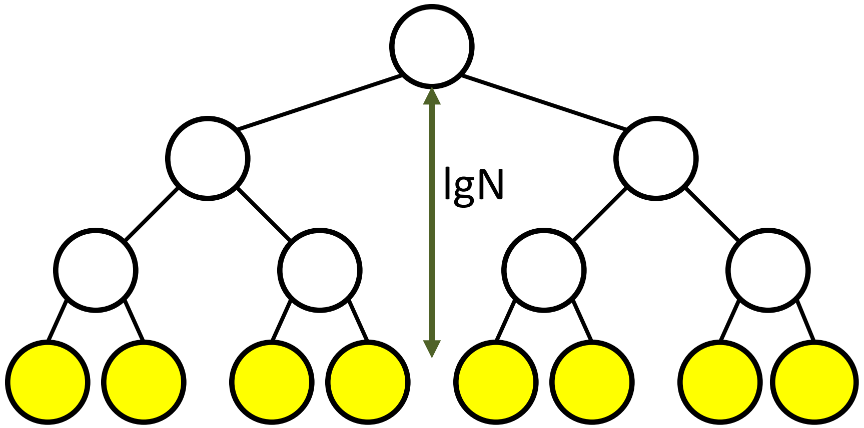
- Convert the given array to a heap using bottom-up:

(must work in place):

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

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.
- Space complexity: **O(1)** (smart implementation)
  - O(N) if not using the smart implementation and a copy is made
- 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.
  - worst case: Θ(N lg N) (for building the heap)
    - The last N/2 nodes (yellow) are inserted in a heap of height (lgN)-1. => T(N) = Ω(NlgN) (worst case).
      - Example that results in Θ(N)?
    - Each of the N insertions takes at most lgN. => T(N) = O(NlgN)
    - => T(N) = Θ(NlgN) (b.c. T(N) = Ω(NlgN) and T(N) = O(NlgN))



Using Heaps

- See leetcode problems tagged with Heap
- Learn how to use a PriorityQueue object from your favorite language
  – Check solutions posted under Solution, but also under Discussions on leetcode. You may find very nice code samples that show a good usage of the library functions.

Priority Queues and Sorting

- Sorting with a max-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.

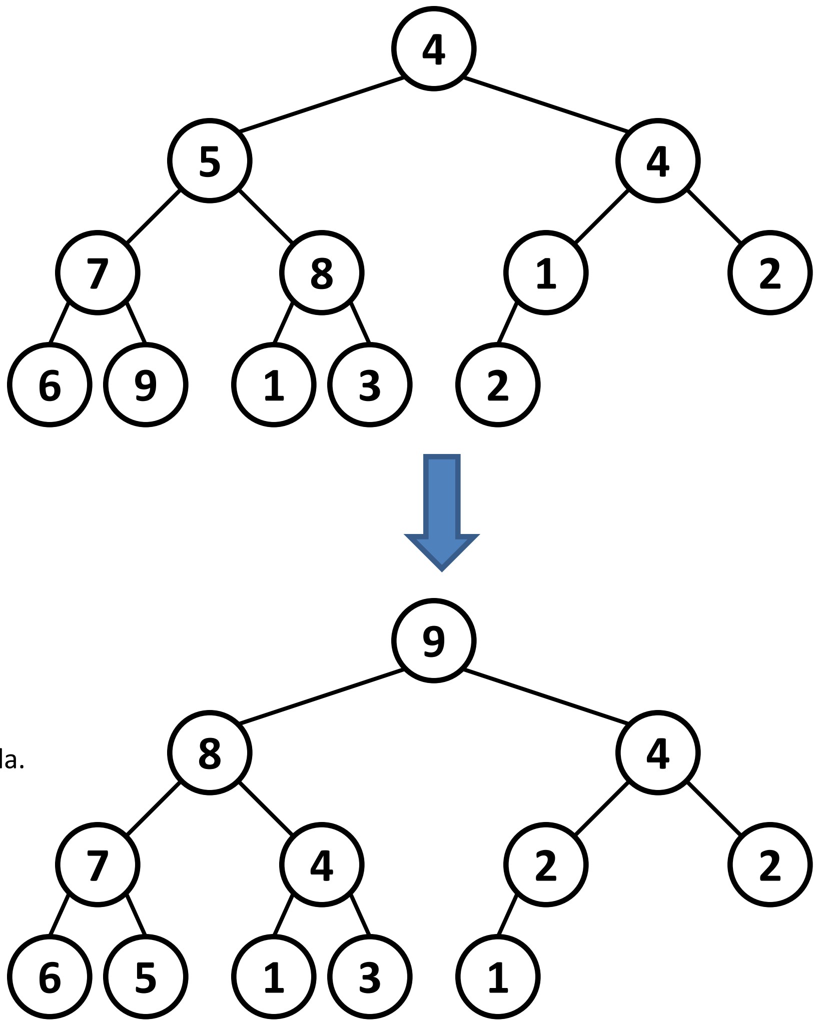
- Heapsort – Θ(NlgN) time, Θ(1) space
  - builds the heap in O(N).
  - NlgN from repeated remove operations
    - N/2 remove max operation can take O(lgN) each => O(NlgN)
  - Not stable, not adaptive

Heapsort

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12
Orig array	-	4	5	4	7	8	1	2	6	9	1	3	2
Heap	-												
1^{st} remove	-												
2^{nd} remove	-												
…	-												
	-												
	-												
	-												
	-												
	-												
	-												
	-												
	-												

Note: it also works for data starting at index 0, with correct child/parent index formula.

```
Heapsort(A,N)              //T(N) = ______
   buildMaxHeap(A,N)       // _________
   while ( N>1 )  {        // ________
      remove(A,&N)
   }
```

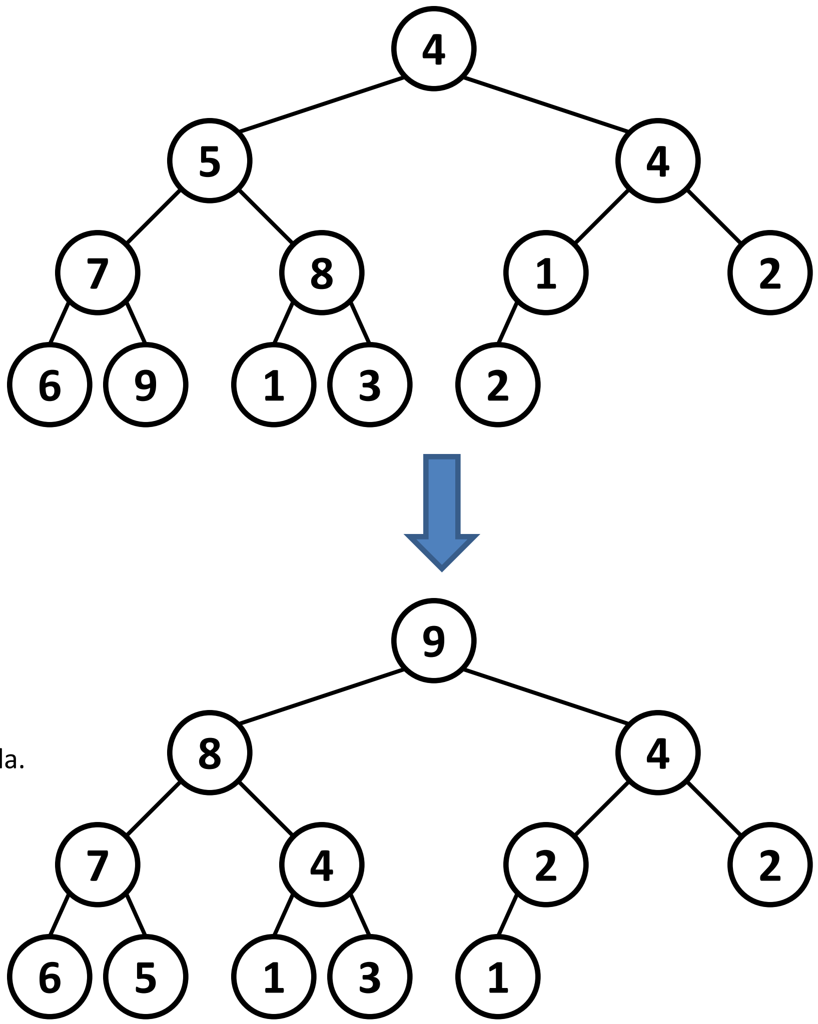


See animation: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
(Note that they do not highlight the node being processed, but directly the children of it as they are compared to find the larger one of them.)

Heapsort

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12
Orig array	-	4	5	4	7	8	1	2	6	9	1	3	2
Heap	-	9	8	4	7	4	2	2	6	5	1	3	1
1st remove	-	8	7	4	6	4	2	2	1	5	1	3	9
2nd remove	-	7	6	4	5	4	2	2	1	3	1	8	9
…	-	6	5	4	3	4	2	2	1	1	7	8	9
	-	5	4	4	3	1	2	2	1	6	7	8	9
	-	4	3	4	1	1	2	2	5	6	7	8	9
	-	4	3	2	1	1	2	4	5	6	7	8	9
	-	3	2	2	1	1	4	4	5	6	7	8	9
	-	2	1	2	1	3	4	4	5	6	7	8	9
	-	2	1	1	2	3	4	4	5	6	7	8	9
	-	1	1	2	2	3	4	4	5	6	7	8	9
	-	1	1	2	2	3	4	4	5	6	7	8	9

Note: it also works for data starting at index 0, with correct child/parent index formula.



```
Heapsort(A,N)          //T(N) = O(NlgN)
   buildMaxHeap(A,N)      // Θ(N)
   for ( N>1 ) {       // Θ(N)       }
       remove(A,&N) // O(lgN)        }  O(NlgN)
```

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.)

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

Is Heapsort stable? - NO

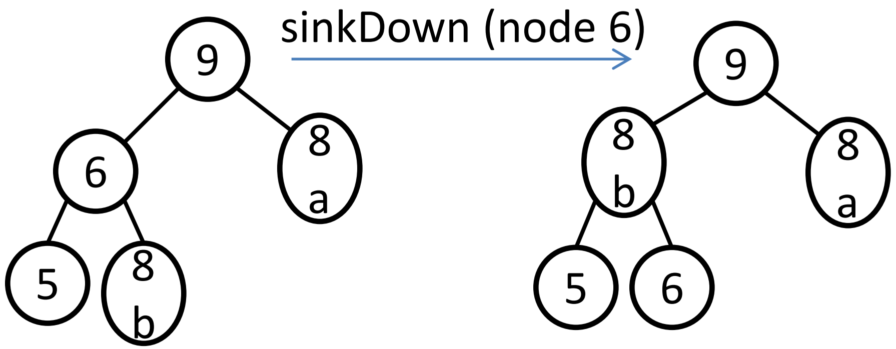
- Both of these operations are unstable:
  - sinkDown
  - Going from the built heap to the sorted array (remove max and put at the end)

```
Heapsort(A,N)
1   buildMaxHeap(A,N)
2  while ( N>1 )
3        remove(A,&N)
```

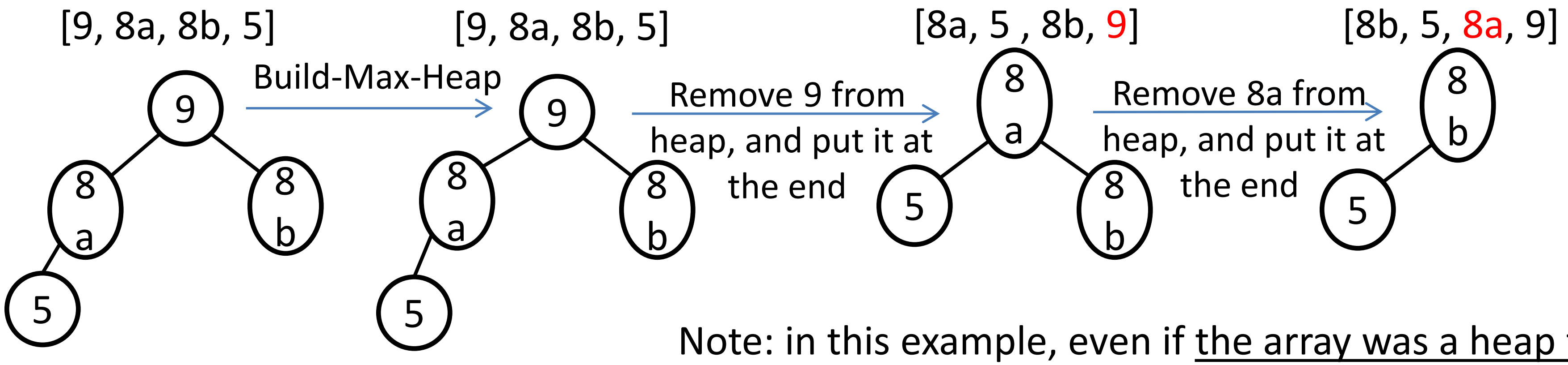
```
sinkDown(A,p,N)  //recursive
    left = left(p)          // index of left child of p
    right = right(p)     // index of right child of p
    index=p
    if (left≤(*N)&&(A[left]>A[index])
       index = left
    if (right≤(*N))&&(A[right]>A[index])
       index = right
    if (index!=p) {
     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

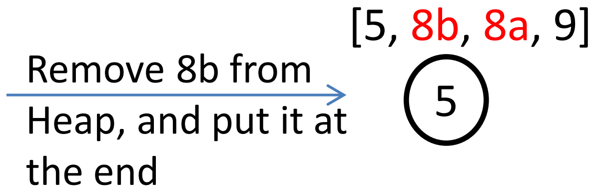
Is Heapsort Stable? - No

Example 1: sinkDown 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 <u>the array was a heap to start with</u>, the sorting part (removing max and putting it at the end) causes the sorting to not be stable.

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** (need to have entire array)
  - 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 remove ops: Θ(k*lgN))
- Using a **min-heap** (good for online processing, less space)
  - Build a min-heap, H, of size **k** (from the first k elements).
  - (N-k) times perform both: *insert* and then *remove* 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: a) less space (Θ(k))
b) good for online processing(maintains top-k at all times)
  - Version 1: Time: Θ(k + (N – k)*lgk) (build heap + (N-k) insert & remove)
  - Version 2 (get the top k sorted): Time: Θ(k + N*lgk) = Θ(Nlgk)
(build heap + (N-k) insert & remove + k remove)

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.

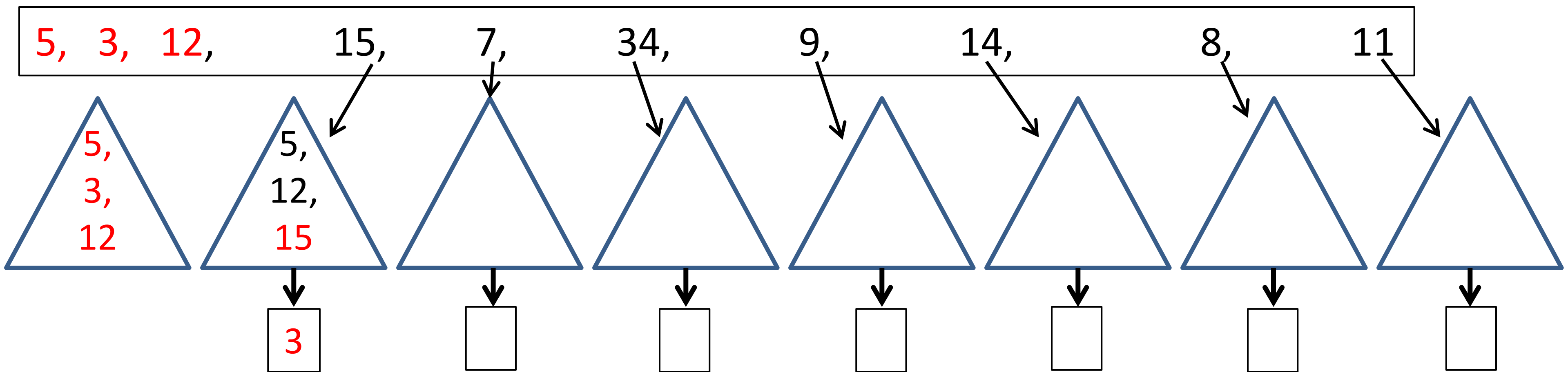
Max-Heap Method Worksheet

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

Top k Largest with Min-Heap Worksheet

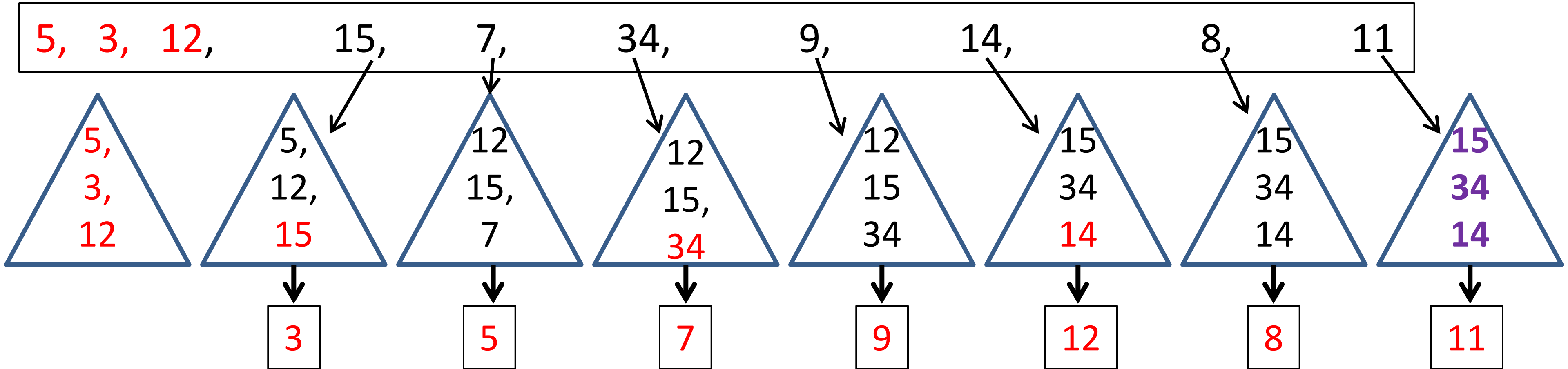
- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.

  (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 times (where 7=N-k) : 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.

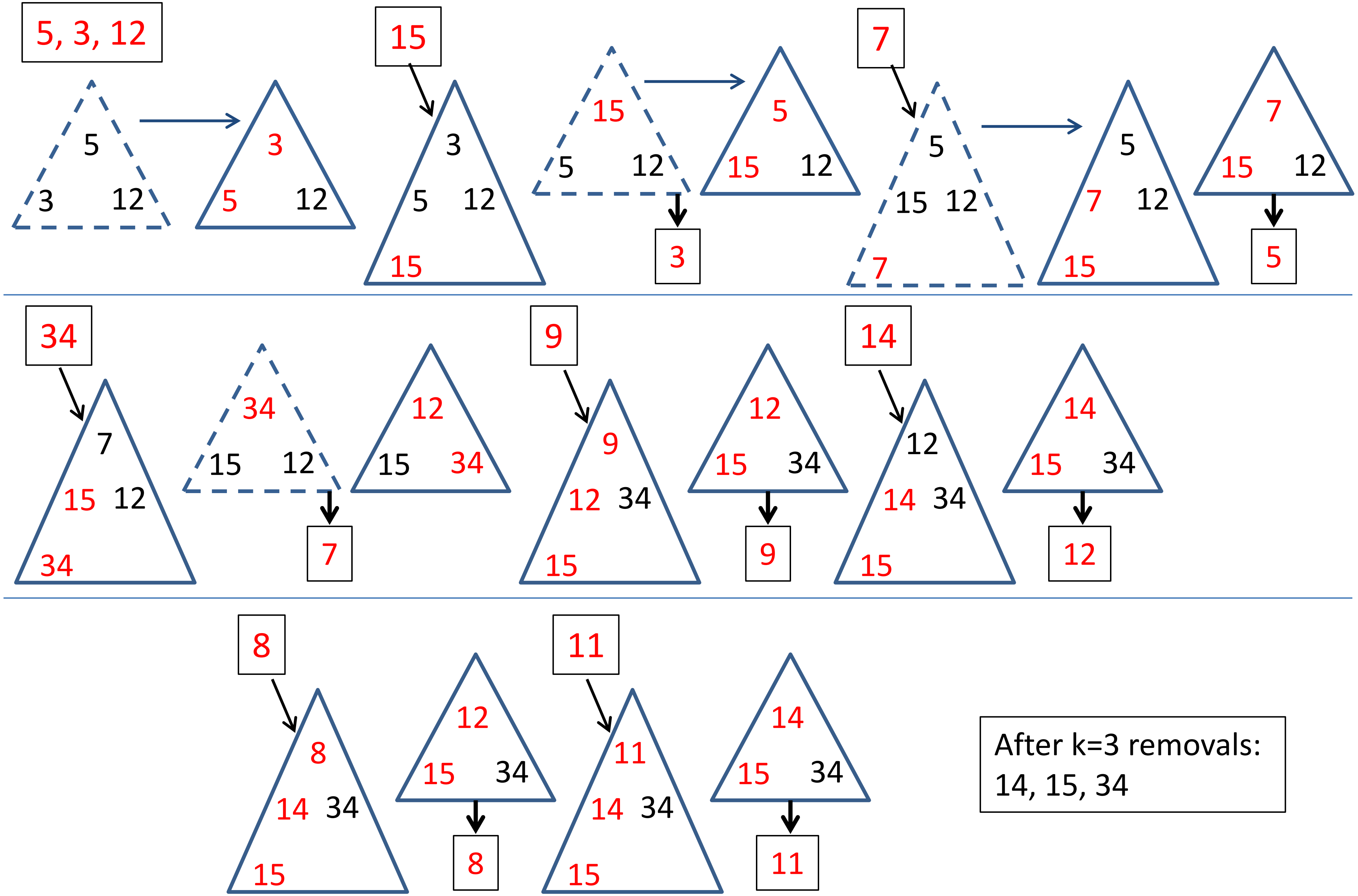


Top k Largest with Min-Heap

- Show the actual heaps and all the steps (insert, remove, 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: remove, top k,…
  - remove() does: remove_max or remove_min based on what type of heap it is.

- To learn using the library: use a MinPriority Queue (Java) as a MaxHeap by providing a comparator that compares for > instead of <

- Extra, not required, but interesting: index heaps (similar idea to indirect sorting)

Extra Materials
not required

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
    - The heap structure is derived from another data structure
    - 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).

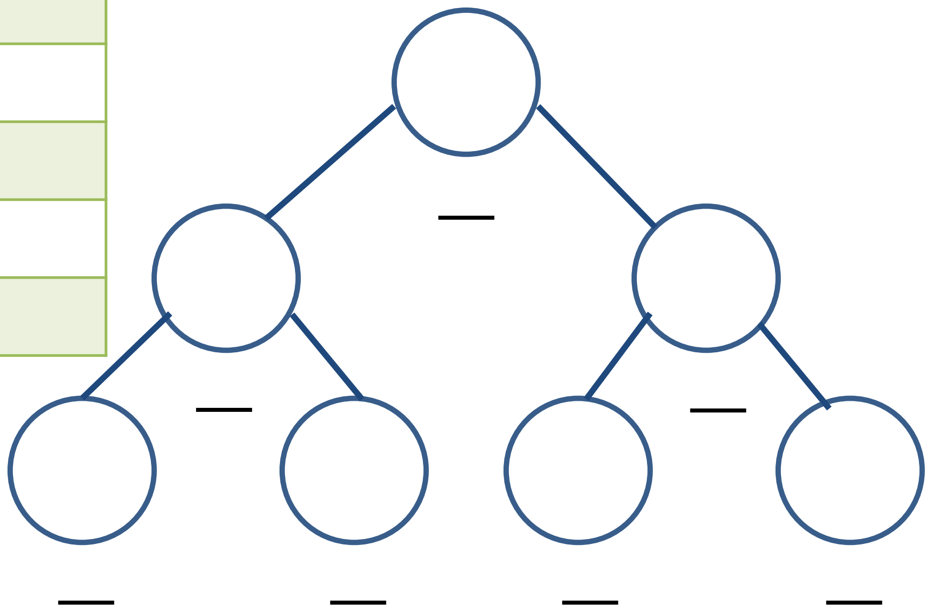
Index Heap Example - Workout

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**
0	**4**	**1**	Aidan	10	
1	**0**	**3**	Alice	7	
2	**3**	**4**	Cam	10	
3	**1**	**2**	Joe	13	
4	**2**	**0**	Kate	20	
5	**5**	**5**	Mary	4	
6	**6**	**6**	Sam	6	

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

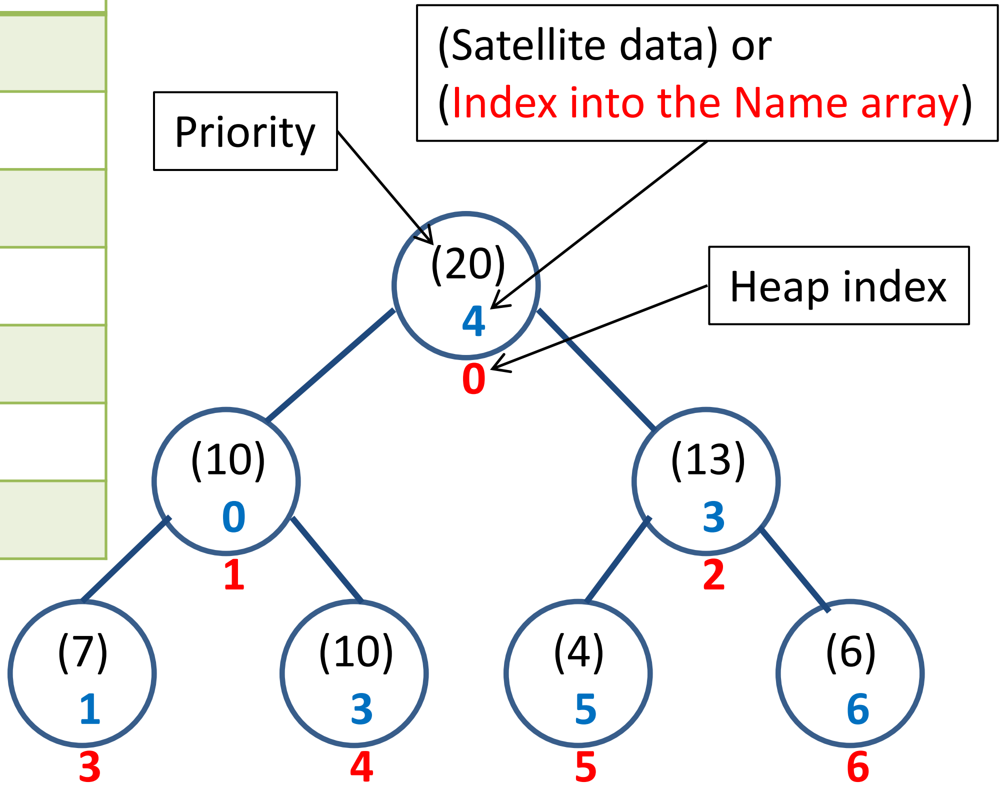
AH – Array to Heap



Index Heap Example - Solution

HA – Heap to Array (HA[0] has index into Name array
AH – Array to Heap

Index	HA (H->A)	AH (A->H)	Name	Priority	Other data
0	4	1	Aidan	10	
1	0	3	Alice	7	
2	3	4	Cam	10	
3	1	2	Joe	13	
4	2	0	Kate	20	
5	5	5	Mary	4	
6	6	6	Sam	6	



Property:
HA(AH(j) = j e.g. HA(AH(4) = 4
AH(HA(j) = j e.g. AH(HA(0) = 0

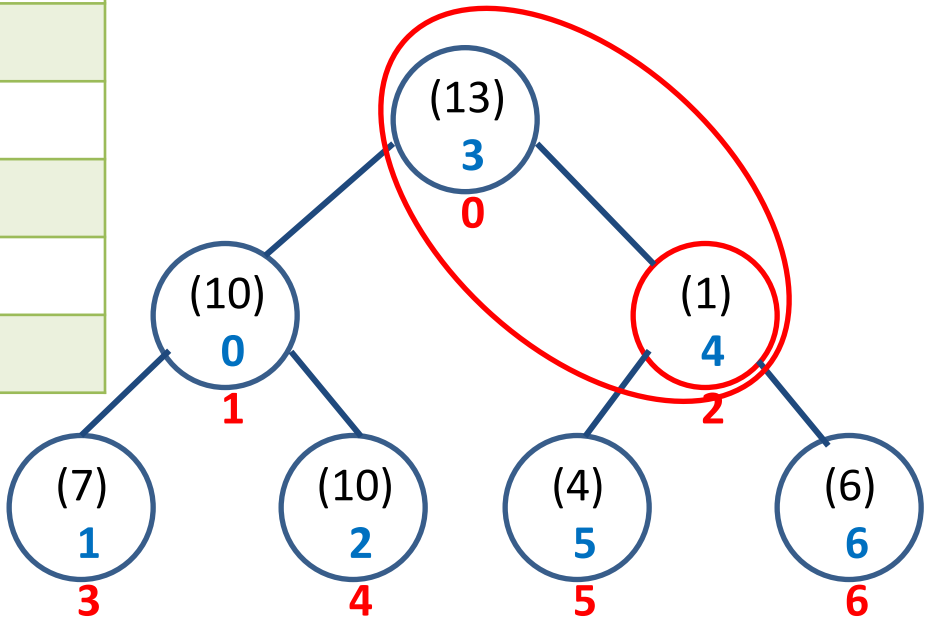
Decrease Kate's priority to 1. Update the heap.
To swap nodes p_1 and p_2 in the heap: $HA[p_1]$<->$HA[p_2]$, and $AH[HA[p_1]]$ <-> $AH[HA[p_2]]$.

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
0	~~4~~ 3	1	Aidan	10	
1	0	3	Alice	7	
2	~~3~~ 4	4	Cam	10	
3	1	~~2~~ 0	Joe	13	
4	2	~~0~~ 2	Kate	~~20~~ 1	
5	5	5	Mary	4	
6	6	6	Sam	6	



Property:
HA(AH(j) = j e.g. HA(AH(4) = 4
AH(HA(j) = j e.g. AH(HA(0) = 0

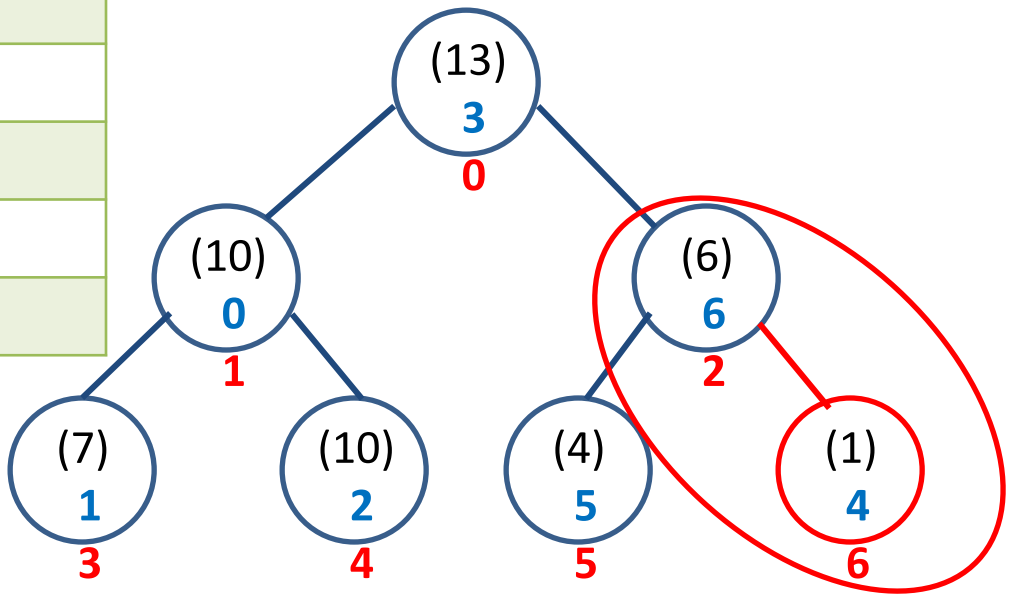
Decrease Kate's priority to 1. Update the heap.
To swap nodes 0 and 2 in the heap: HA[0]<->HA[2], and AH[HA[0]] <-> AH[HA[2]].

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
0	~~4~~ 3	1	Aidan	10	
1	0	3	Alice	7	
2	~~3~~ ~~4~~ 6	4	Cam	10	
3	1	~~2~~ 0	Joe	13	
4	2	~~0~~ ~~2~~ 6	Kate	~~20~~ 1	
5	5	5	Mary	4	
6	~~6~~ 4	~~6~~ 2	Sam	6	



Property:
HA(AH(j) = j e.g. HA(AH(4) = 4
AH(HA(j) = j e.g. AH(HA(0) = 0

Continue to fix down 1. Update the heap.
To swap nodes 2 and 6 in the heap: HA[2]<->HA[6], and AH[HA[2]] <-> AH[HA[6]].

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.

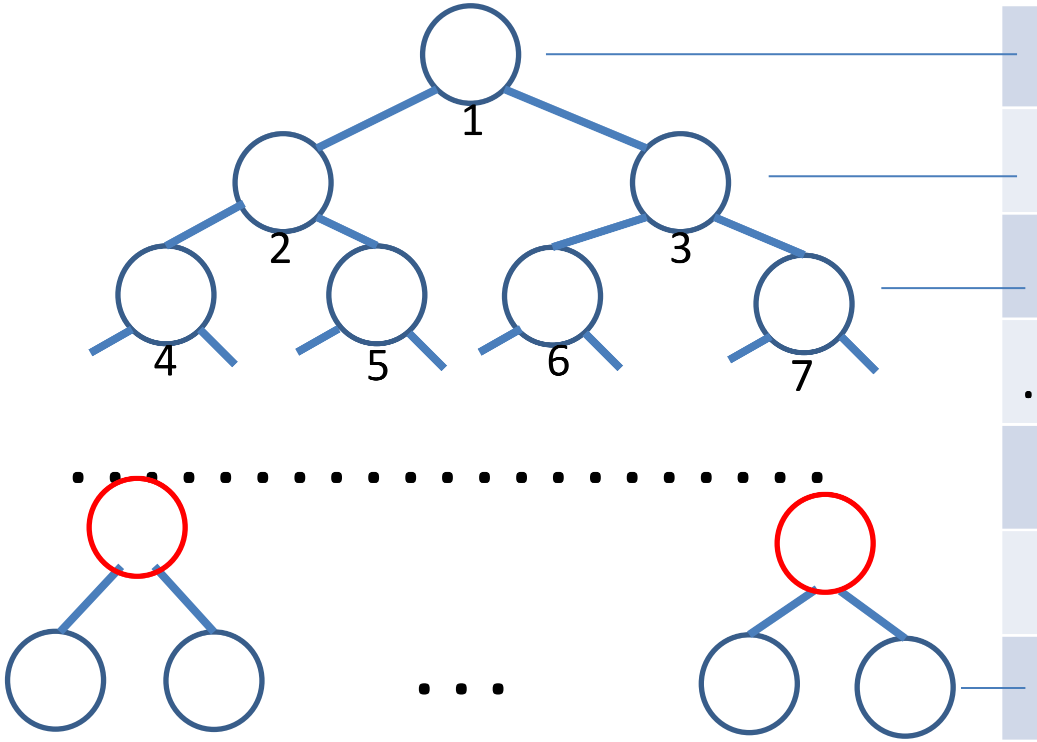
```
buildMaxHeap(A,N)
   for (p = N/2; p>=1; p--)
      sinkDown(A,p,N)
```

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: $2^{n-1}\sum_{k=1}^{n-1} kx^k \le \sum_{k=1}^{\infty} kx^k \to 2^{n-1}\frac{x}{(1-x)^2}$ 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!)**

Removed, detailed slides

sinkDown(A,p,N)

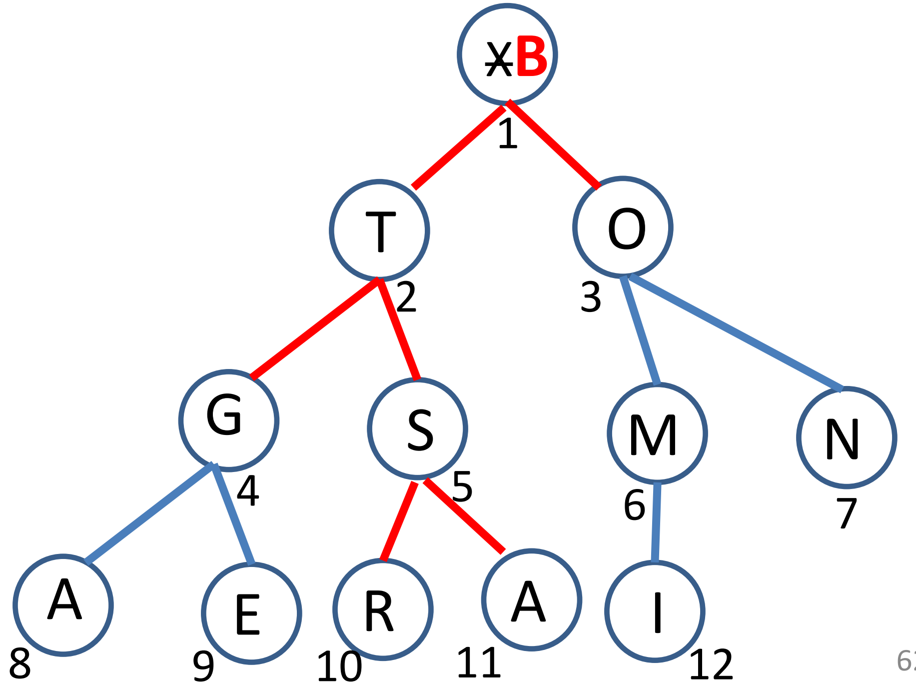
Decrease key

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

Short, but harder to understand version

- 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)  }
```



Heap Operations

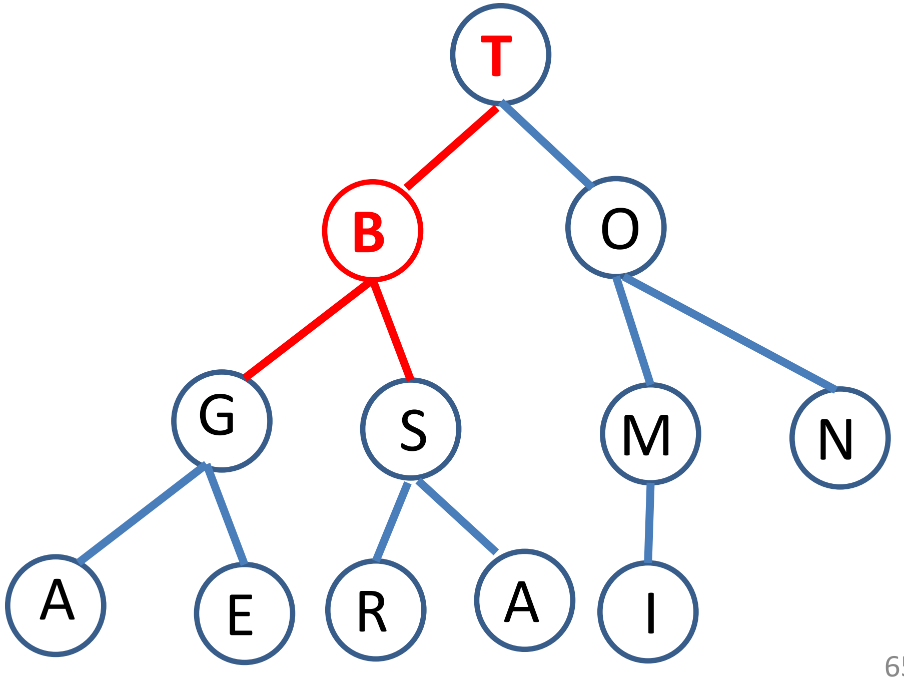
- Initialization:
  - Given N-size array, **heapify** it.
  - Time: Θ(N). Good!
- Insertion of a new item:
  - Requires rearranging items, to maintain the **heap property**.
  - Time: O(lg N). Good!
- Deletion/removal of the largest element (max-heap):
  - Requires rearranging items, to maintain the **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 remove (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 $\lfloor i/2 \rfloor$

swimDown

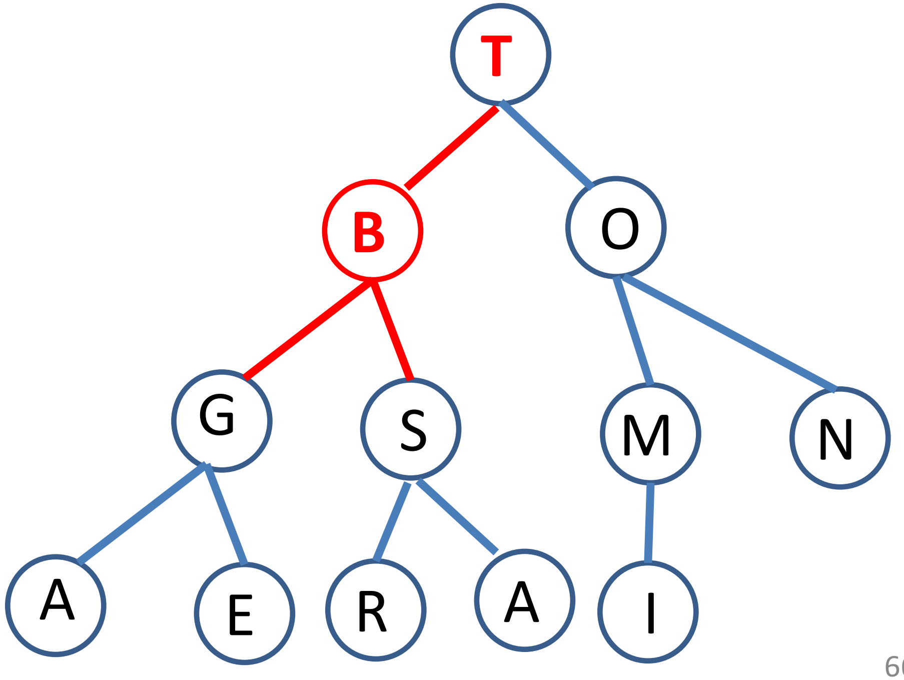
- B will move down until in a good position.
- Exchange B and T.



swimDown

- B will move down until in a good position.

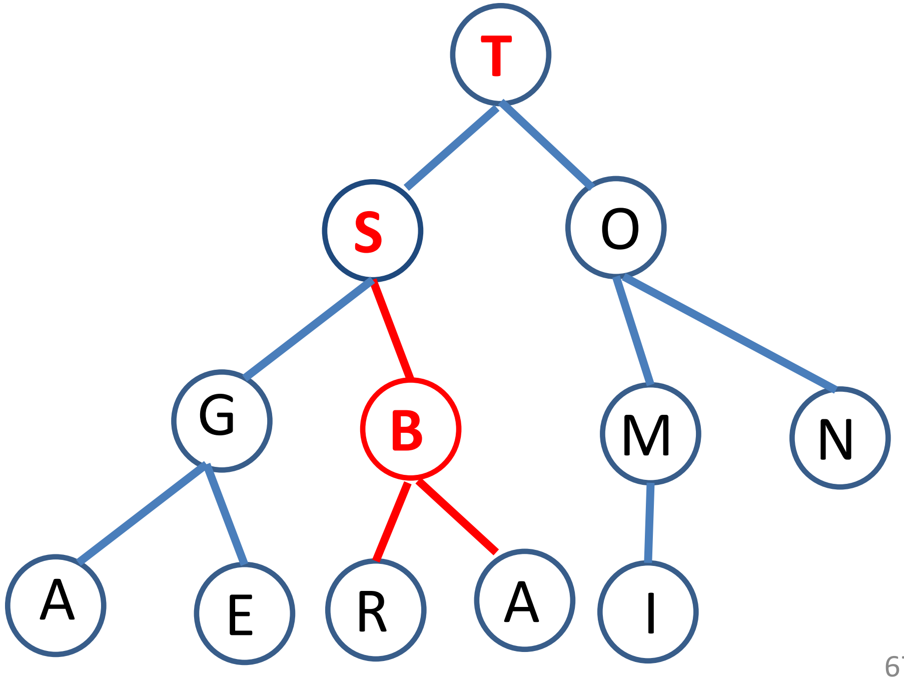
- Exchange B and T.
- Exchange B and S.



swimDown

- B will move down until in a good position.

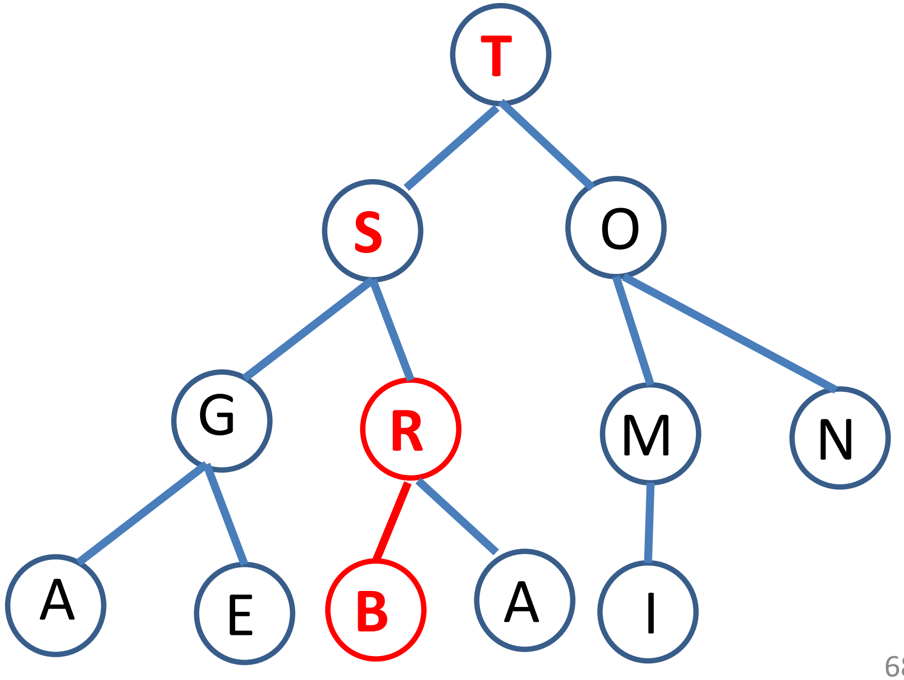
- Exchange B and T.
- Exchange B and S.
- Exchange B and R.



swimDown

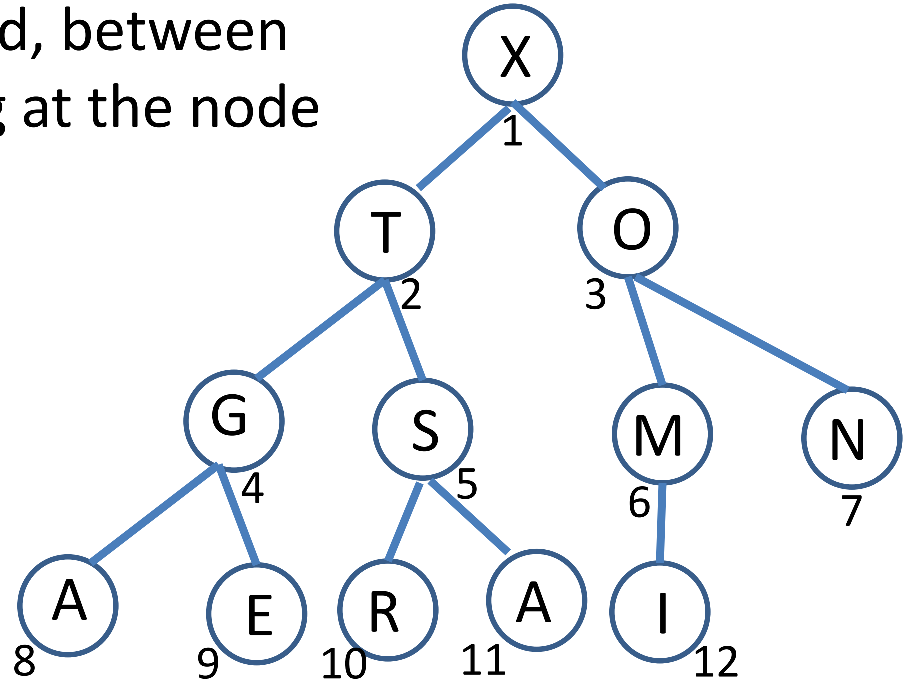
- B will move down until in a good position.

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



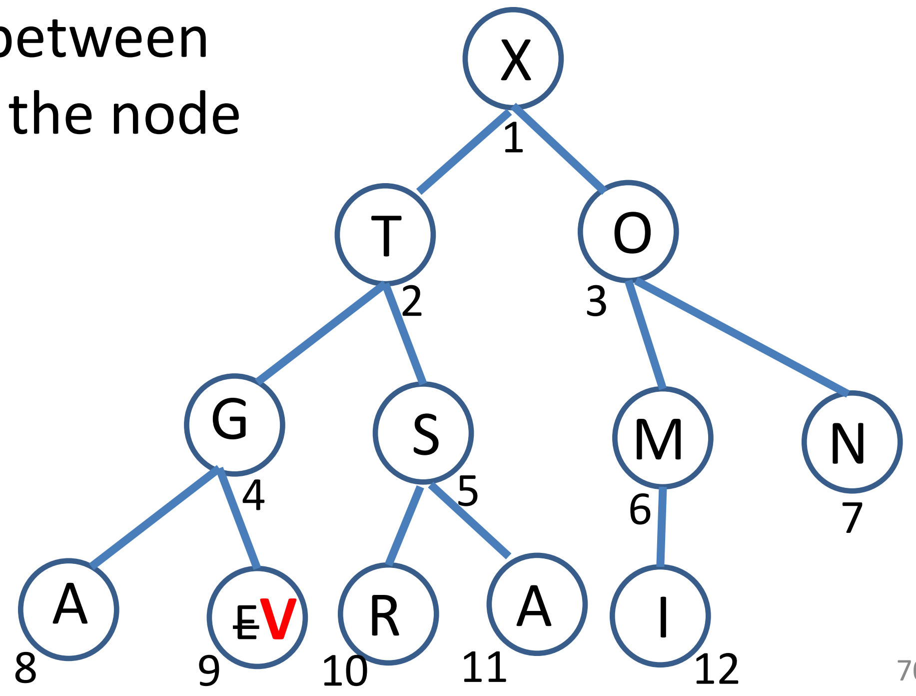
Increasing a Key

- 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.



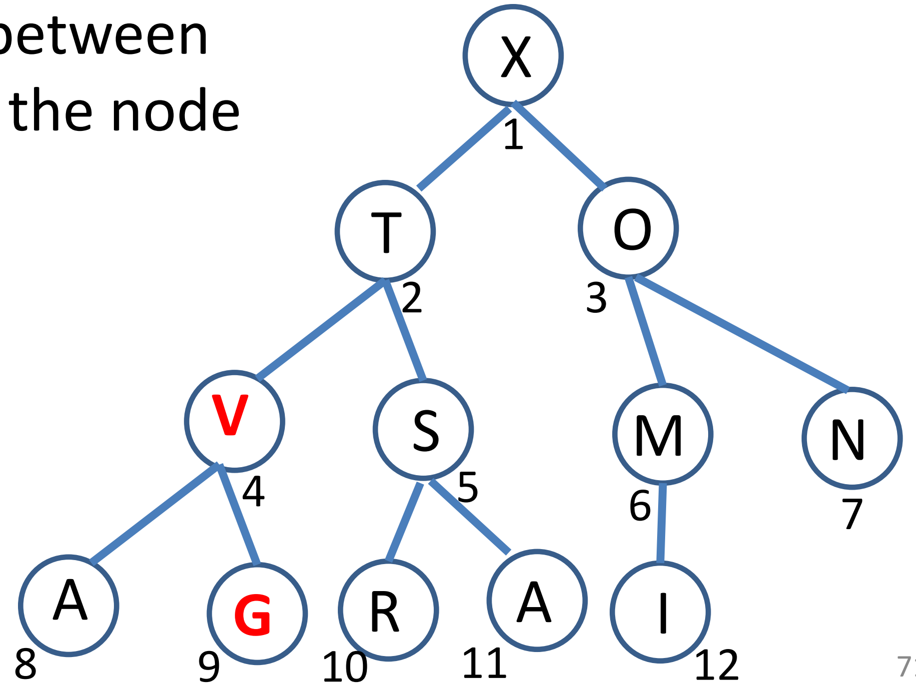
Increasing a Key

- 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.



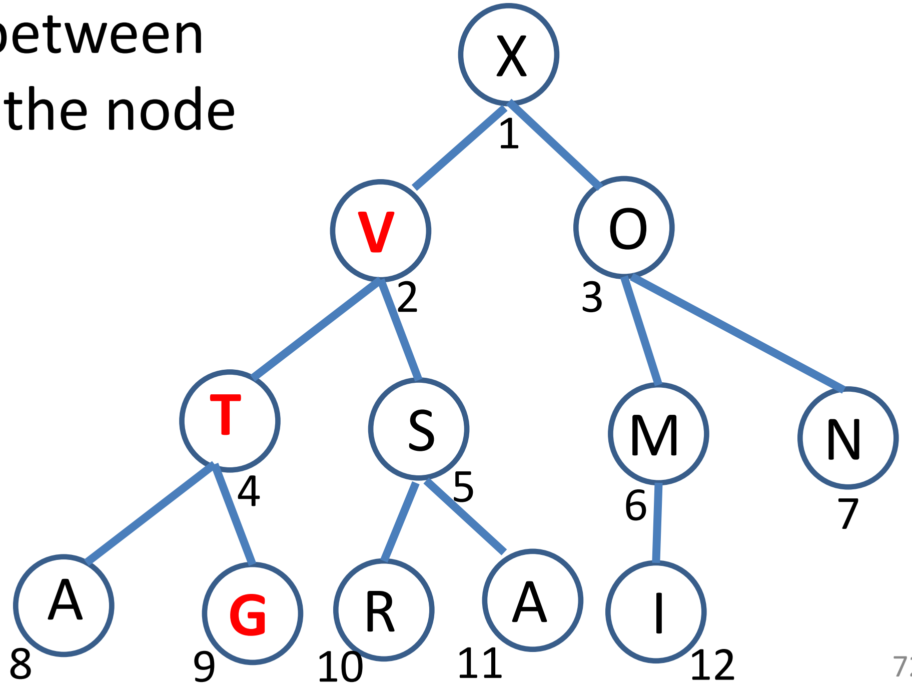
Increasing a Key

- 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?



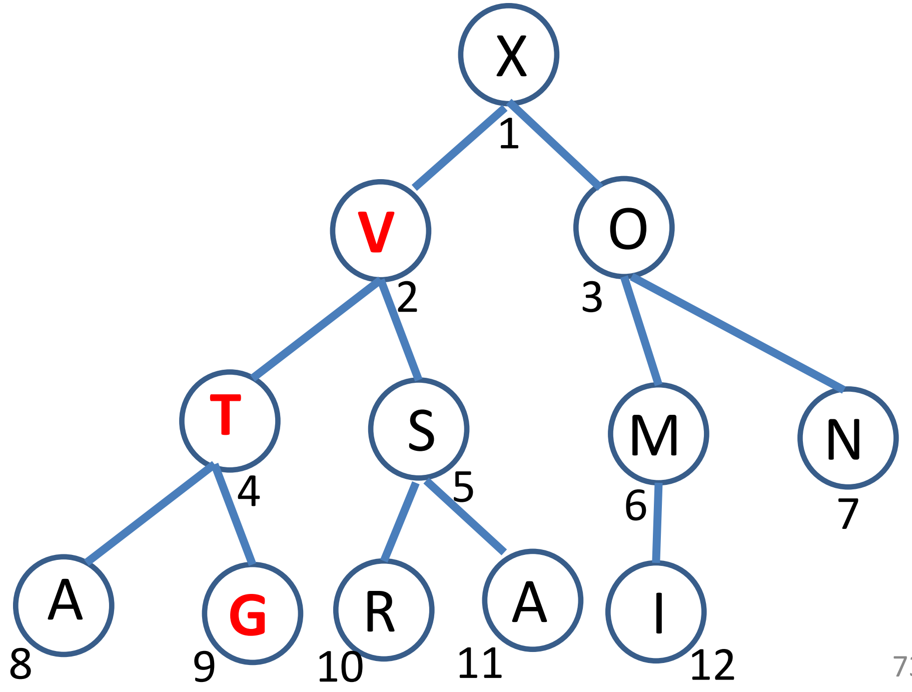
Increasing a Key

- 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.



Worksheet