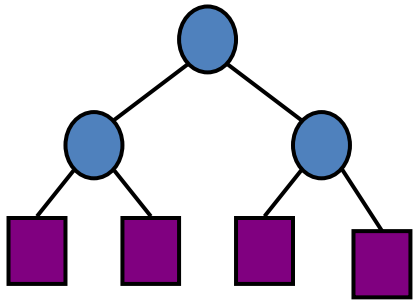
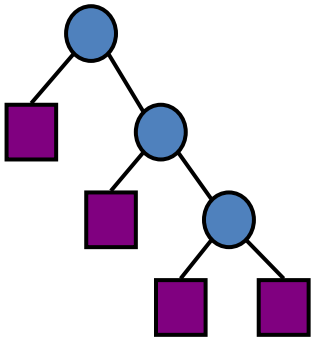
Heaps and Priority Queues

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

Quick Recap of Simple BSTs

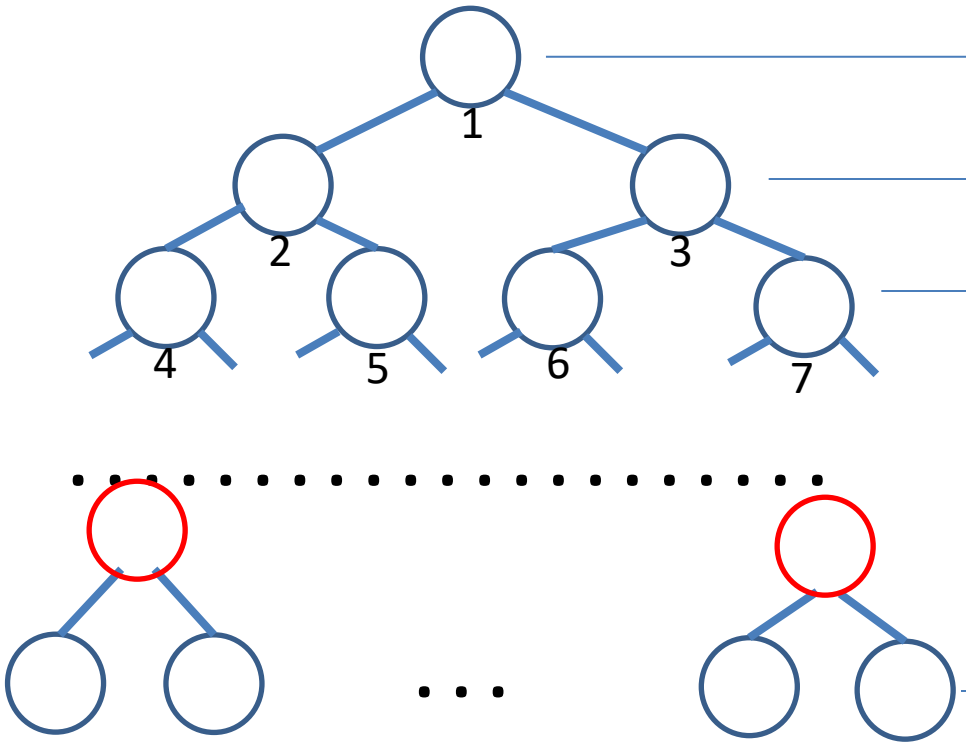
- **Traversals**:
  - Preorder: visit, left, right
  - Inorder: left, visit, right
  - Postorder: left, right, visit
- The **depth** of a node v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also 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

What is a Heap

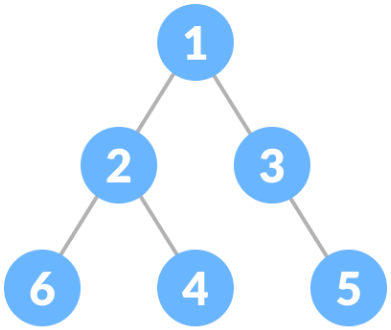
A Binary Heap is a Binary Tree with the following properties:

- 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.

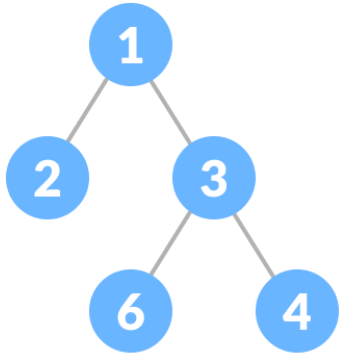
  - In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in the Binary Tree.

  - 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

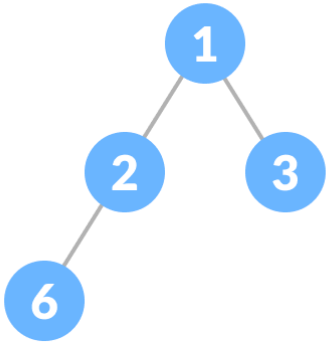
Complete vs. Full Binary Tree



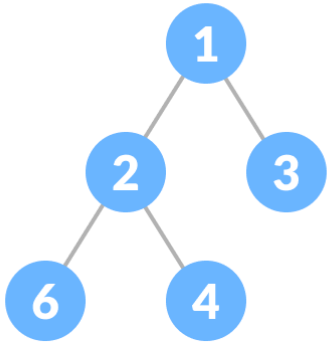
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✗ Complete Binary Tree



✗ Full Binary Tree
✓ Complete Binary Tree



✓ Full Binary Tree
✓ Complete Binary Tree

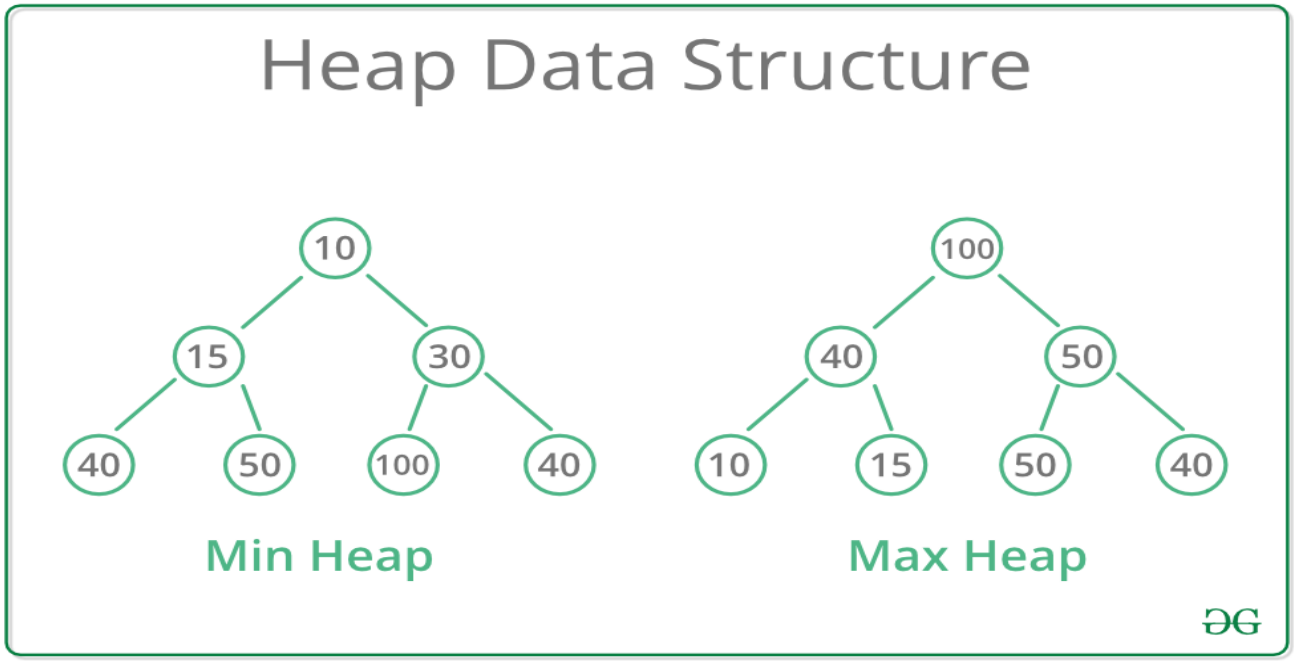
A **full** binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

A **complete** binary tree is a binary tree in which all the levels are completely 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 completely filled.

Heap Data Structure

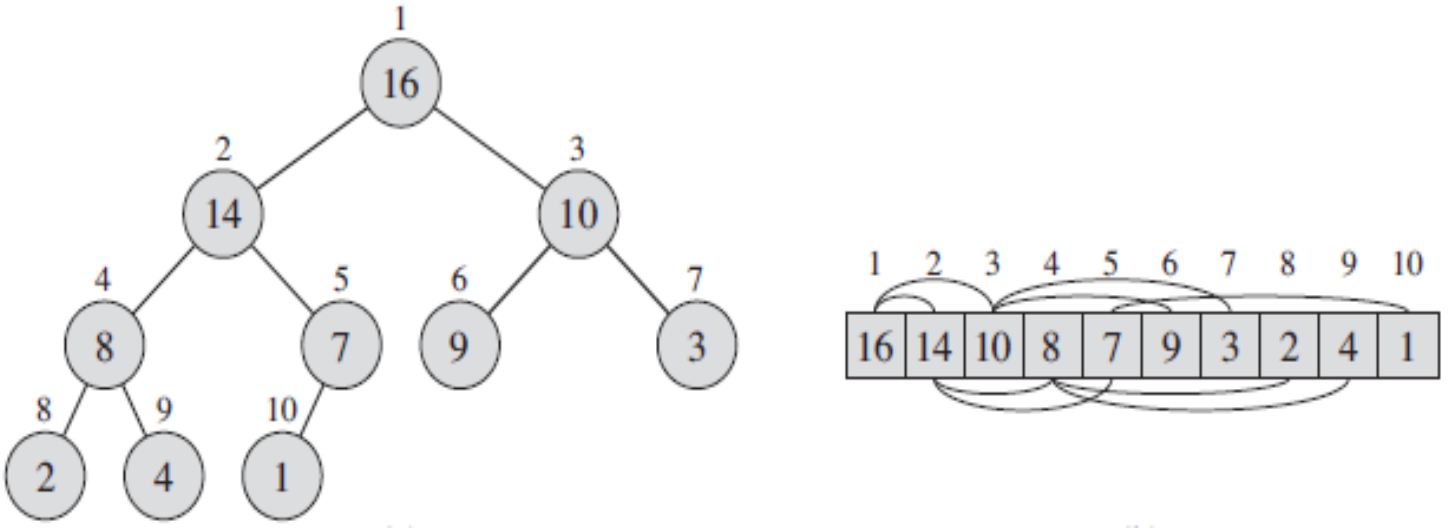
- A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:
  - **Max-Heap**: In a Max-Heap the key present at the root node must be larger than or equal to all the keys present at all its children. The same property must be recursively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



Some Notations

- The heap data structure 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

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.

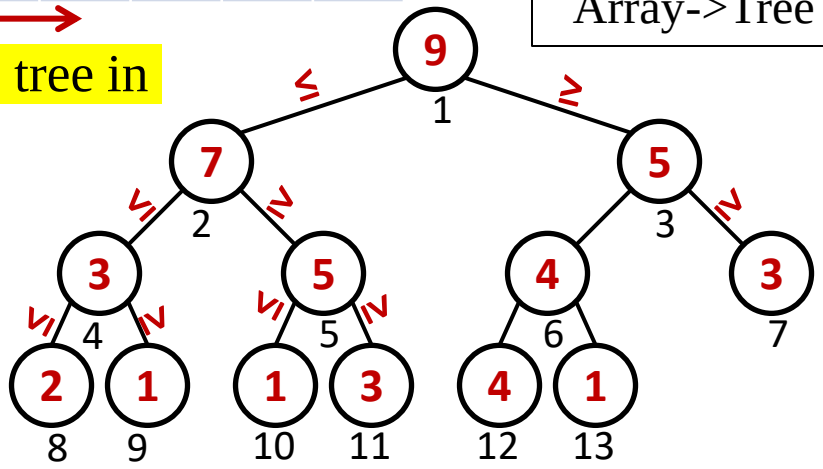
Binary Max-Heap: Stored as Array ⇔ Viewed as Tree

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.



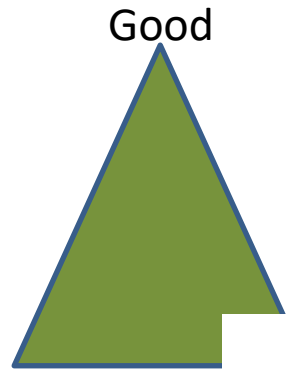
If N items =>
$h = \lfloor lgN \rfloor$
h is the height
If height h =>
$2^h \le N \le 2^{h+1}-1$

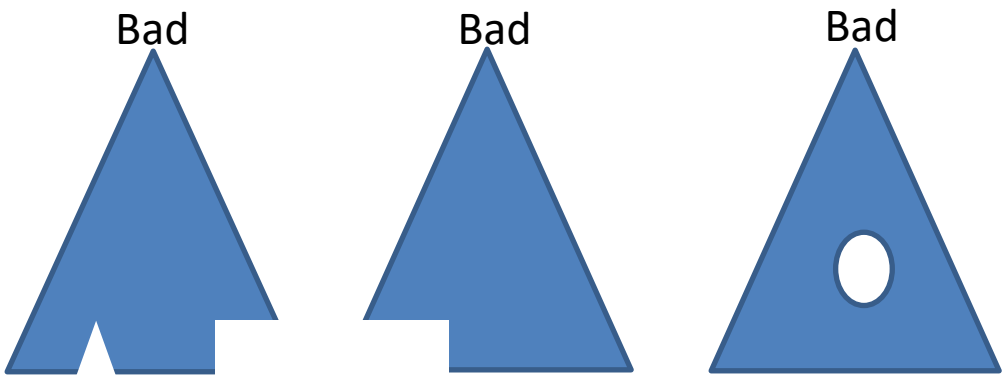
Heap – Shape Property

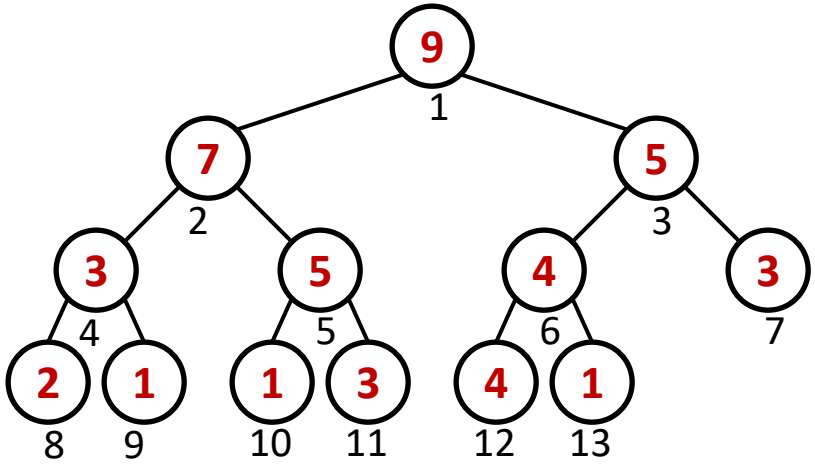
P2: Shape (complete tree: "no holes") ⇔ array storage

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

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



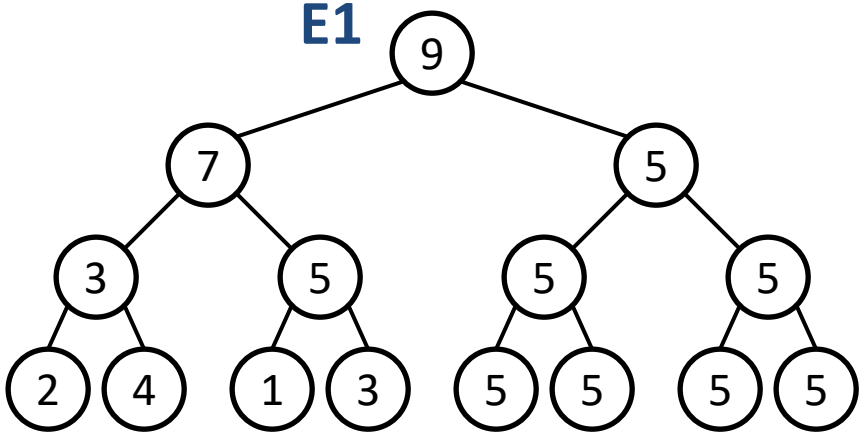




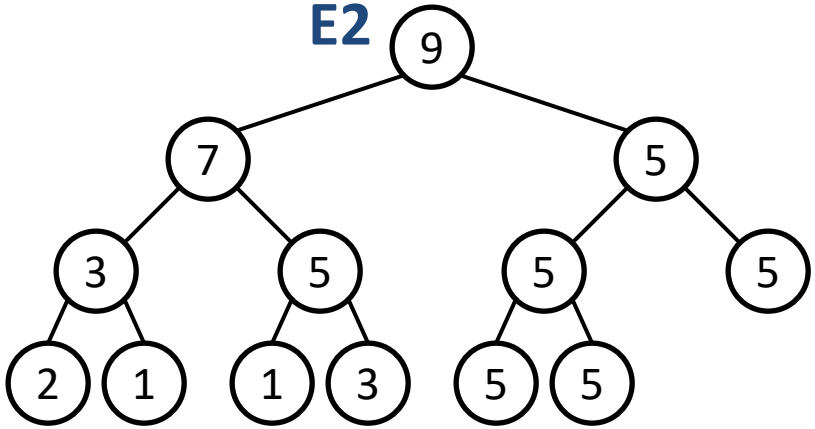
Heap Practice

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

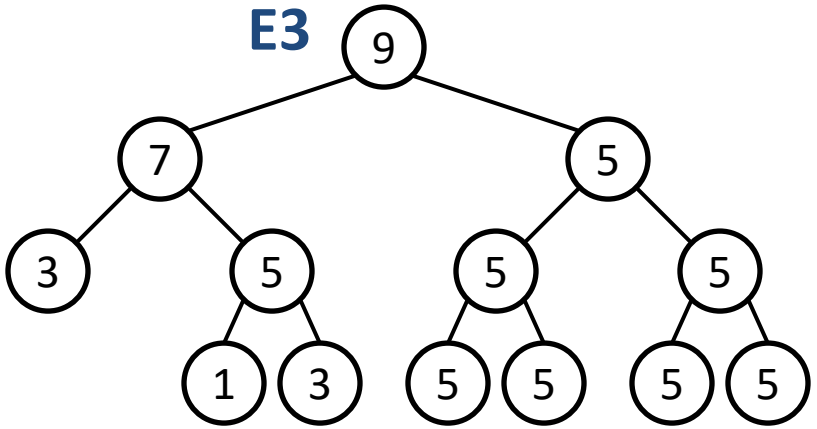
E1



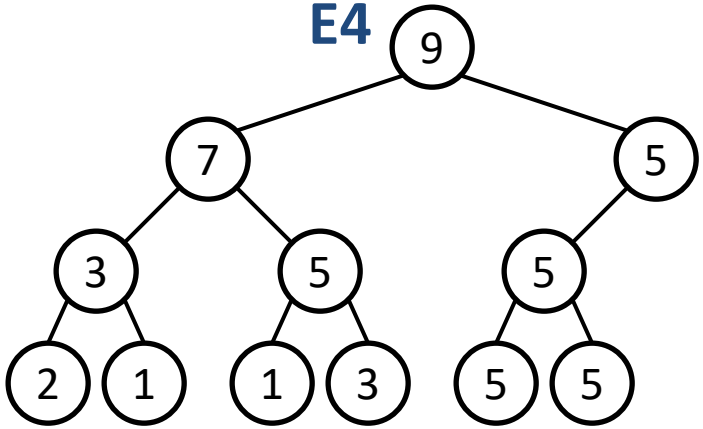
E2



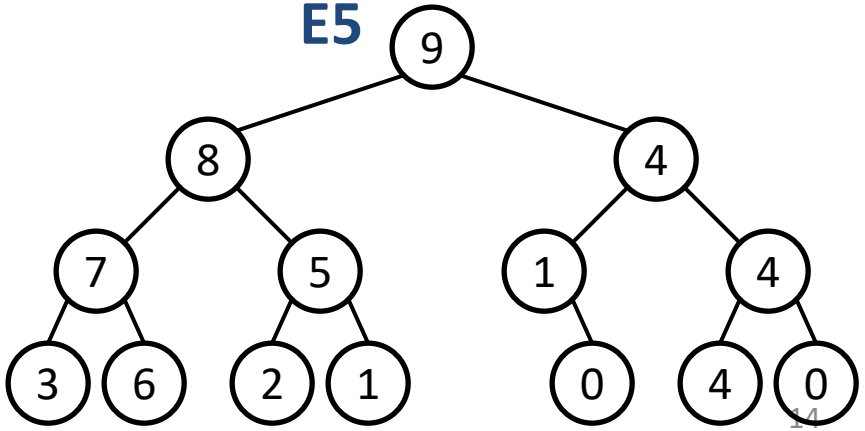
E3



E4

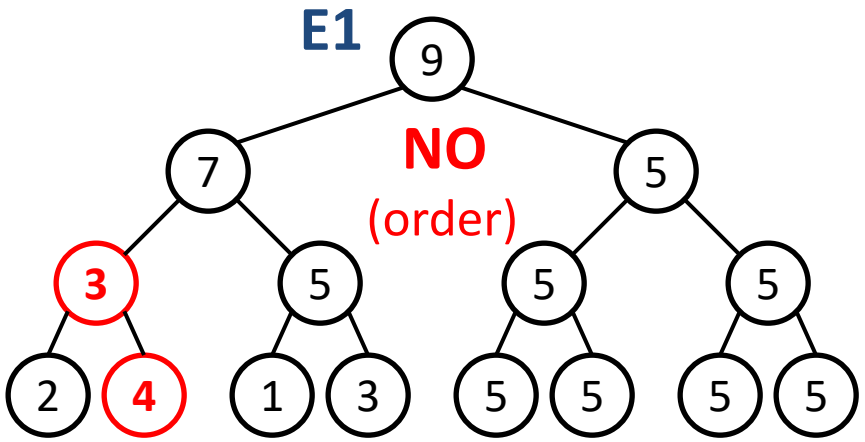


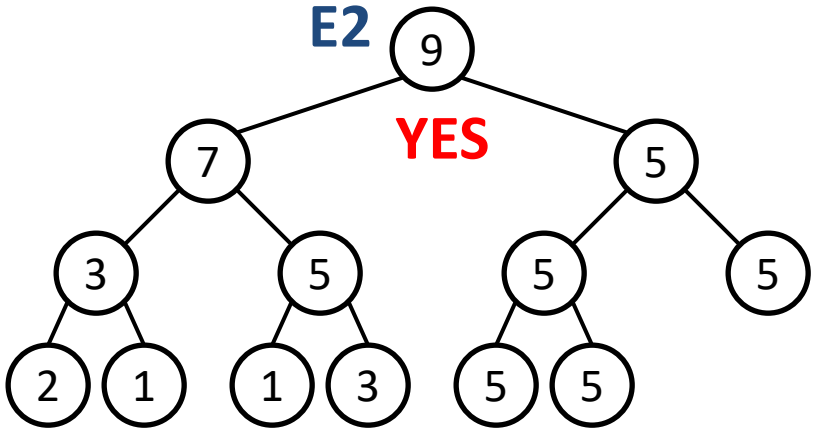
E5

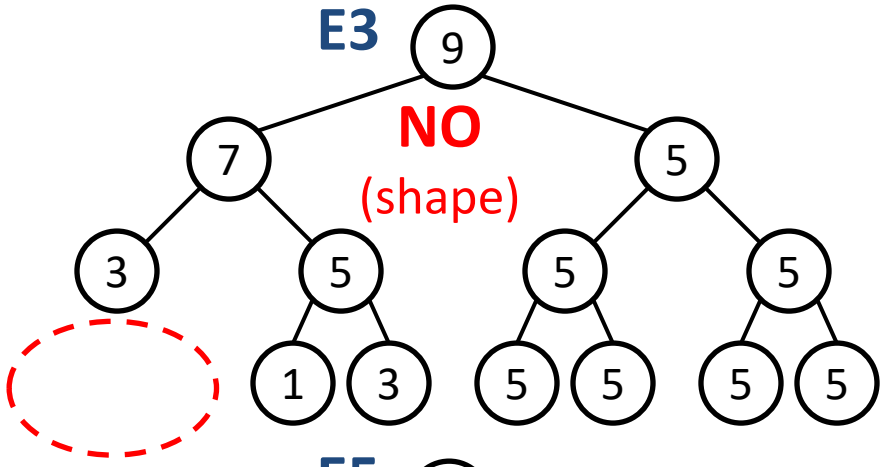


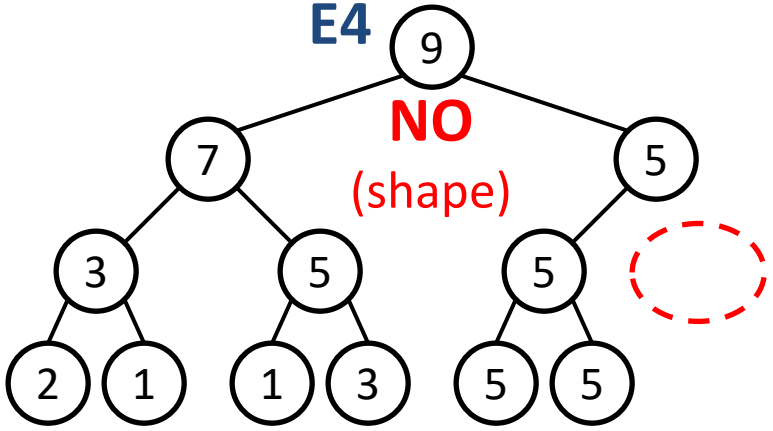
Answers

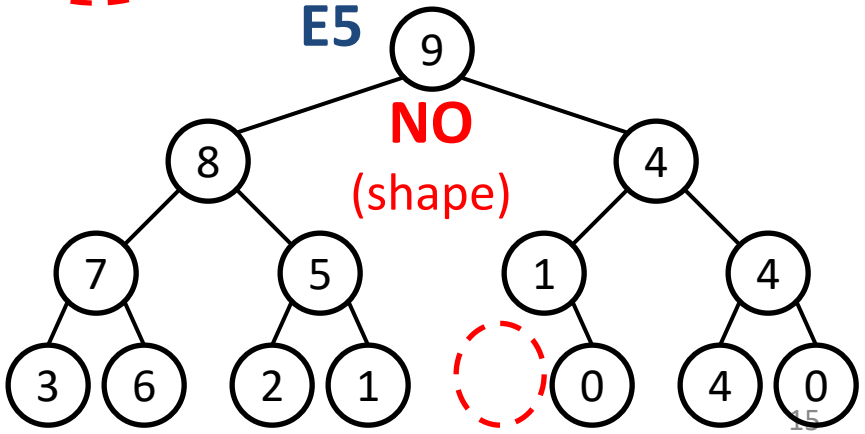
For each tree, say if it is a max heap or not.











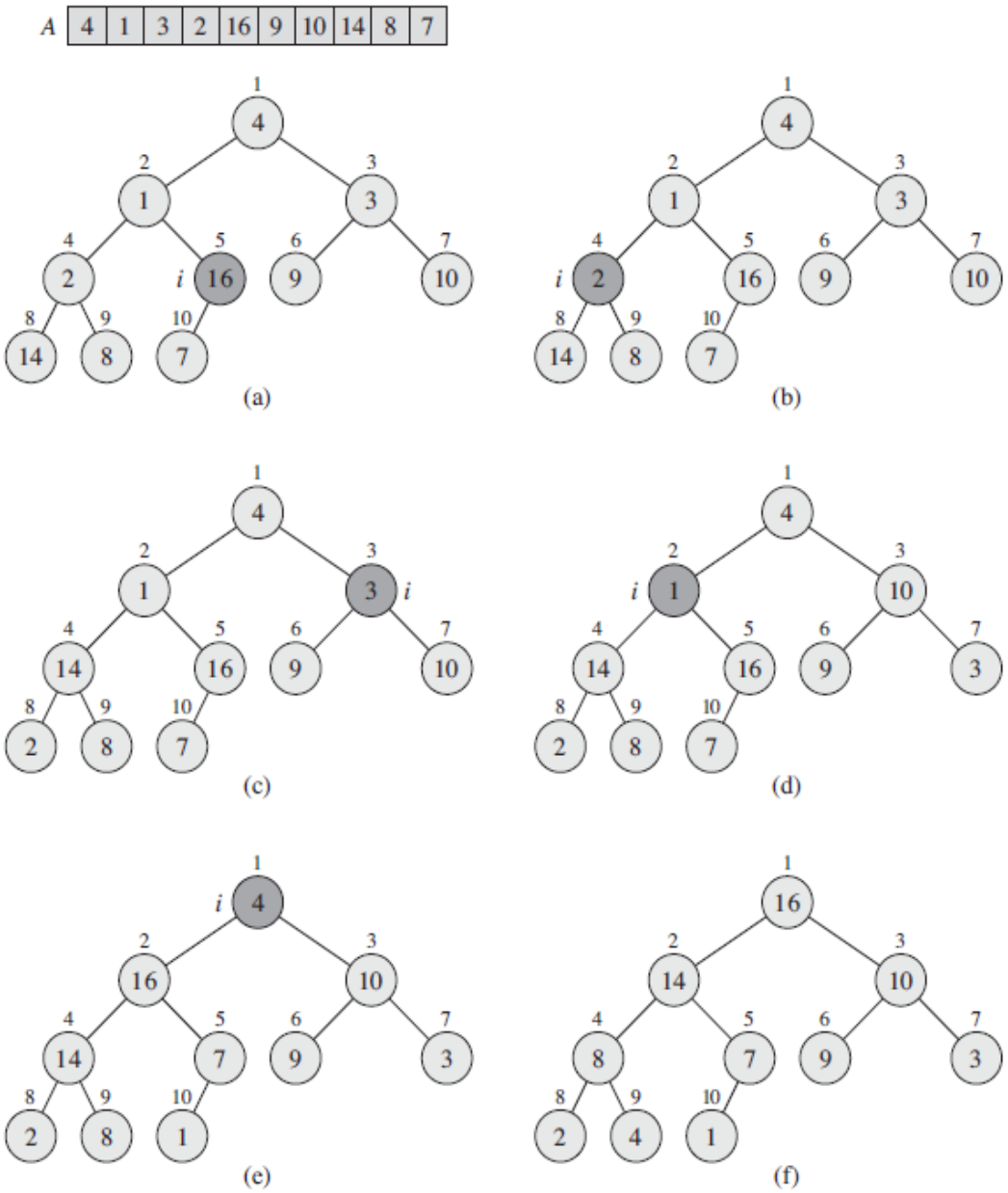
Examples and Exercises

- 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.

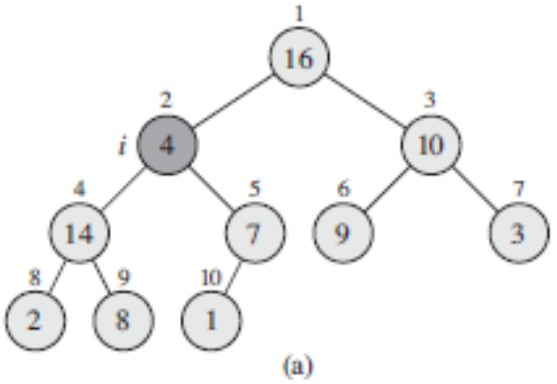
```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```



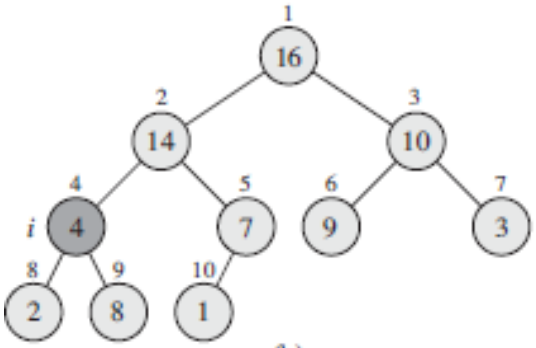
Example

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.

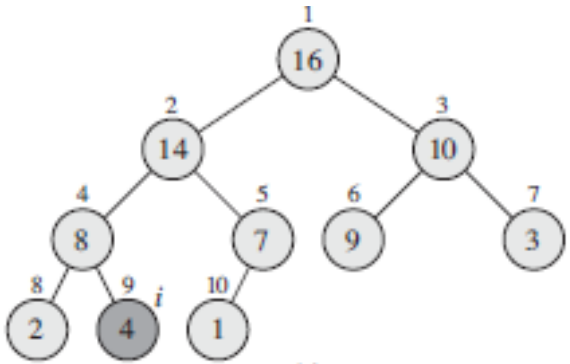
Example + Runtime



(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 $T(n) = O(\log_2 n)$.

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.

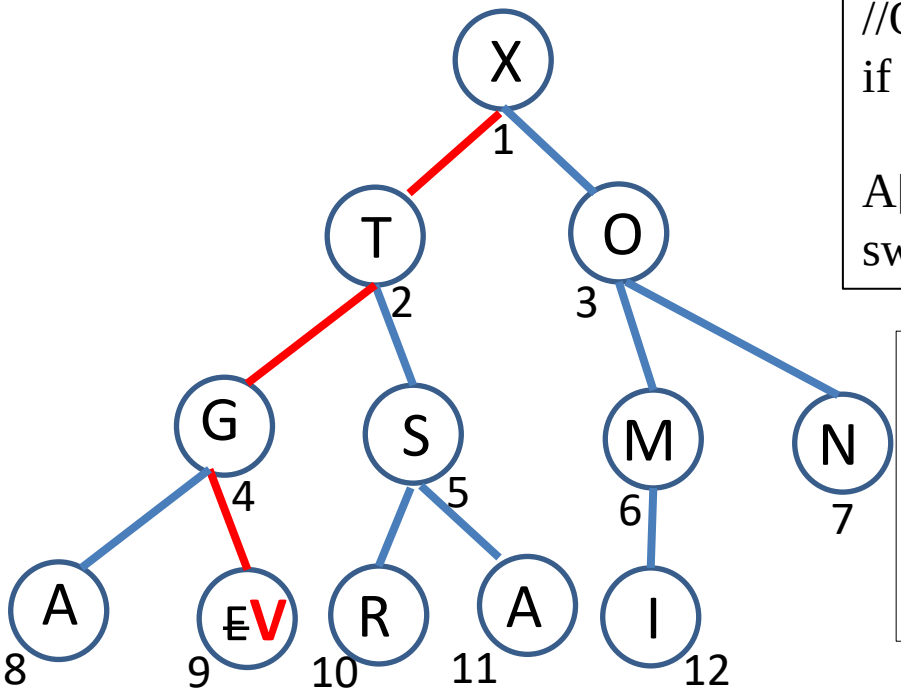
- Can lead to violation of the heap property.

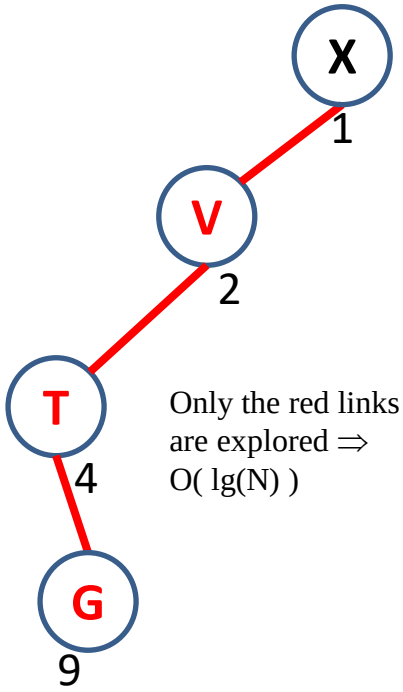
swimUp to fix the heap:

- 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))

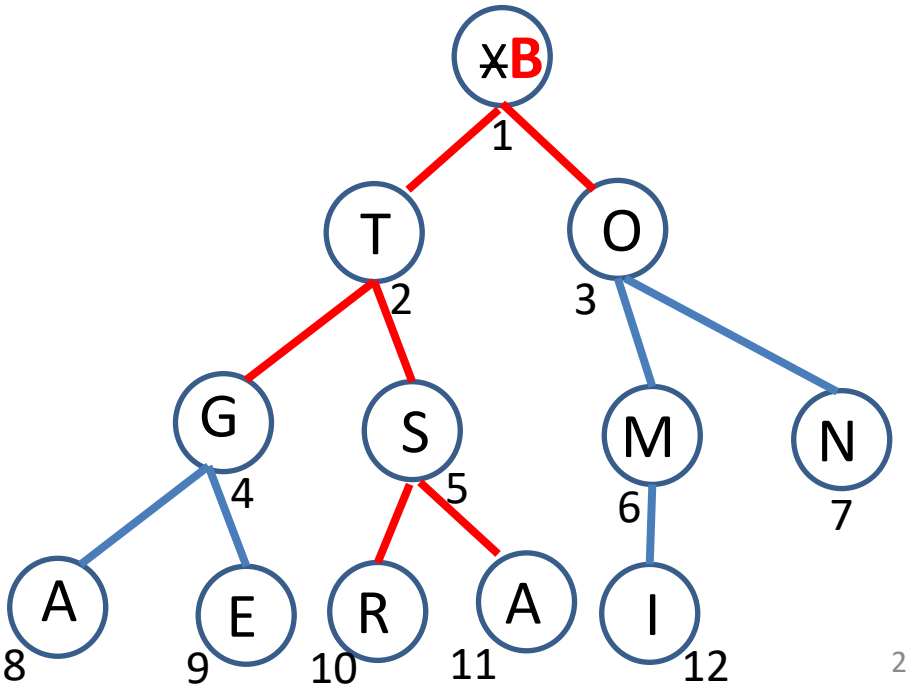
sinkDown(A,p,N)

Decrease key

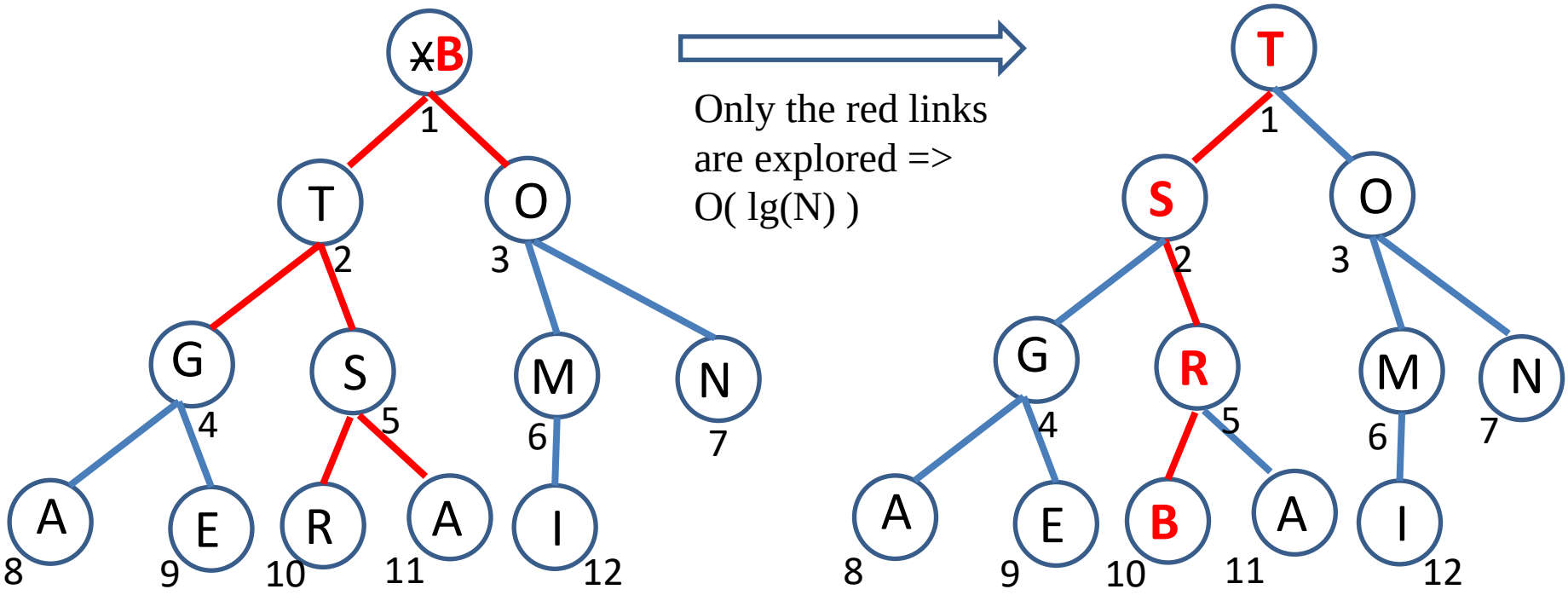
(Max-Heapify/fix-down/float-down)

- 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)  }
```



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

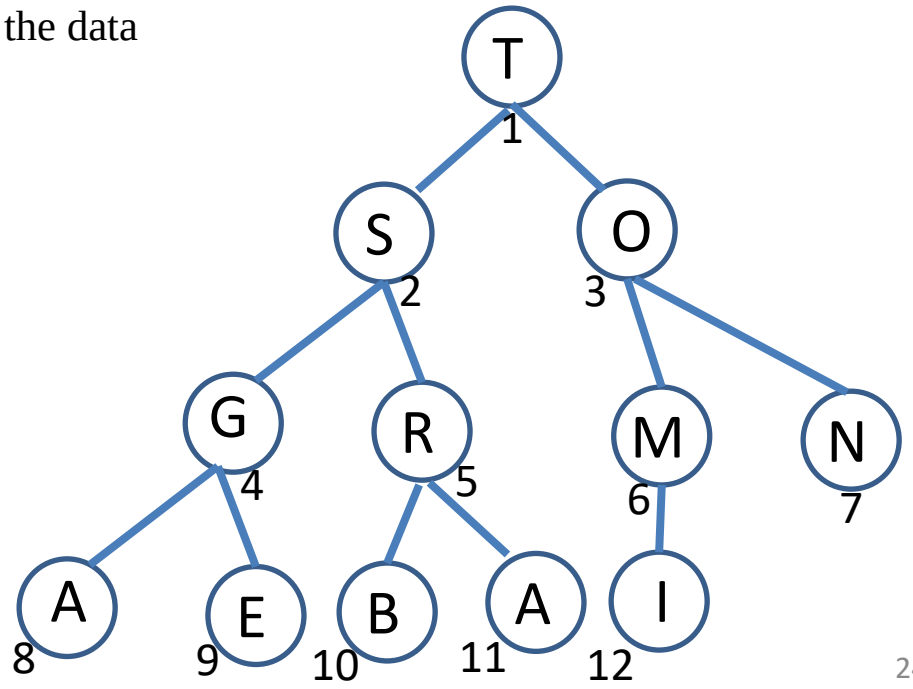
⌊3/2⌋ ⌊6/2⌋ ⌊13/2⌋

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  }
```

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

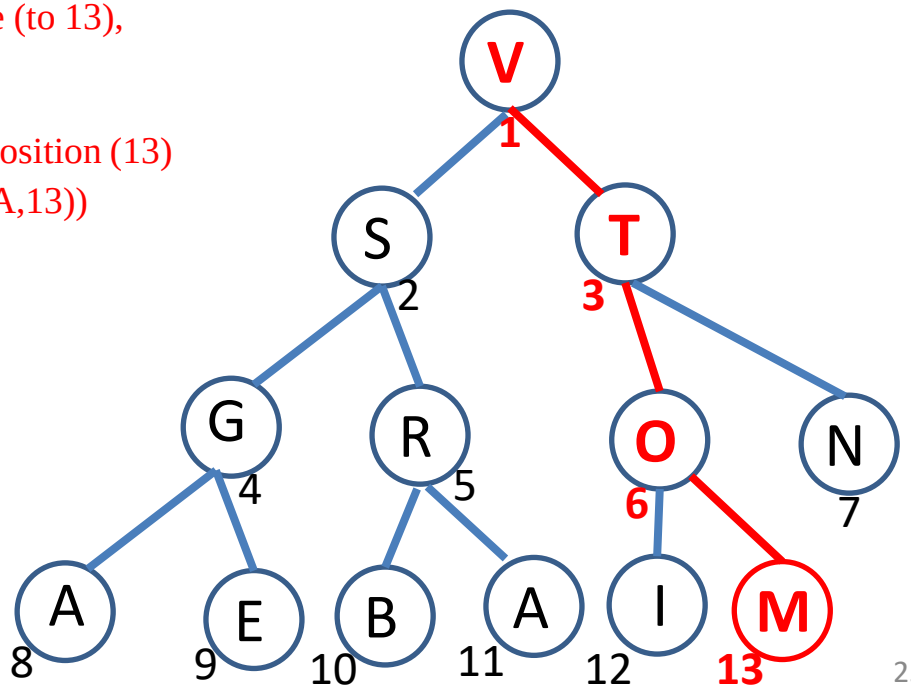
```
insert(A,newKey,&N)
(*N) = (*N)+1 // permanent change
//same as increaseKey:
i = (*N)
A[i] = newKey
while ((i>1)&&(A[i]>A[i/2])) {
   swap: A[i] <-> A[i/2]
   i = i/2  }
```

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

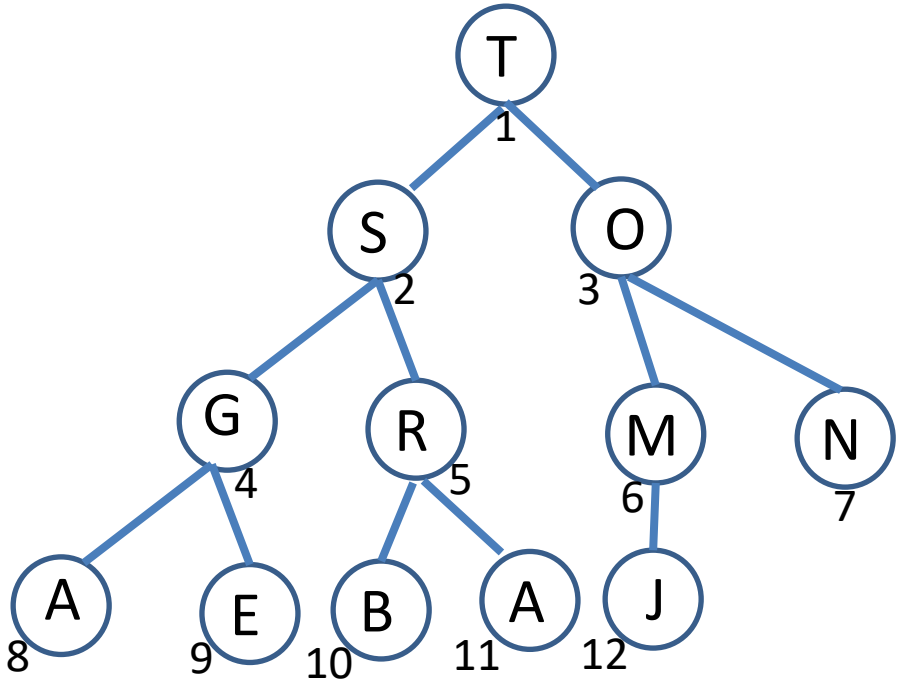
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

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?



Remove Maximum

N is 12

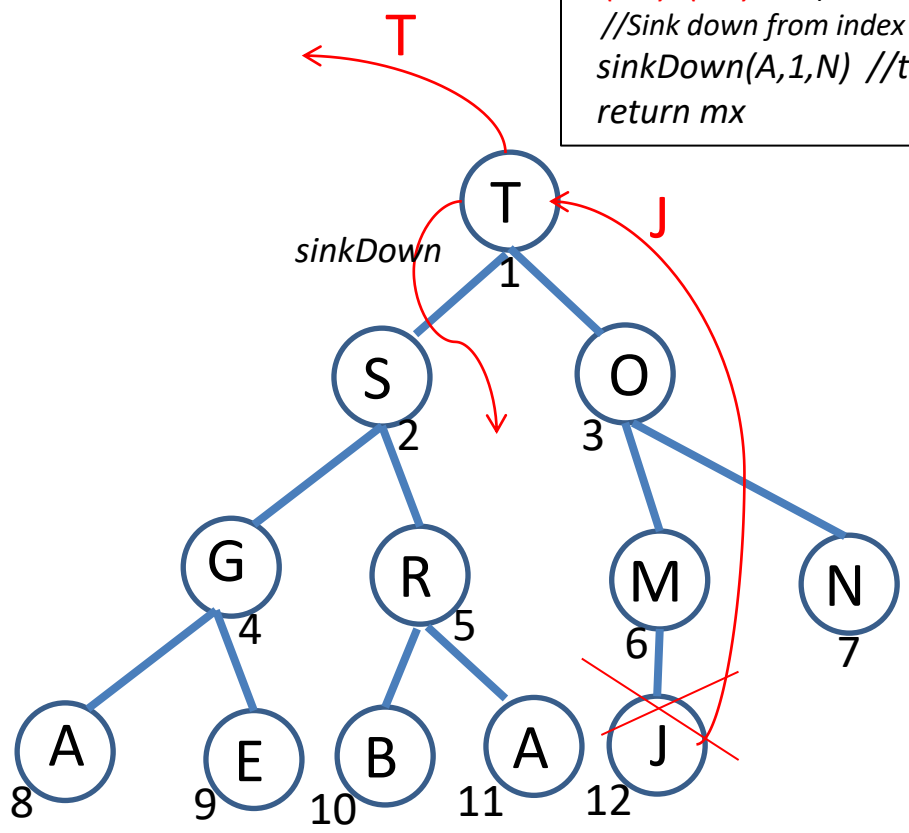
index	1	2	3	4	5	6	7	8	9	10	11	12
value	**T**	**S**	**O**	**G**	**R**	**M**	**N**	**A**	**E**	**B**	**A**	**J**

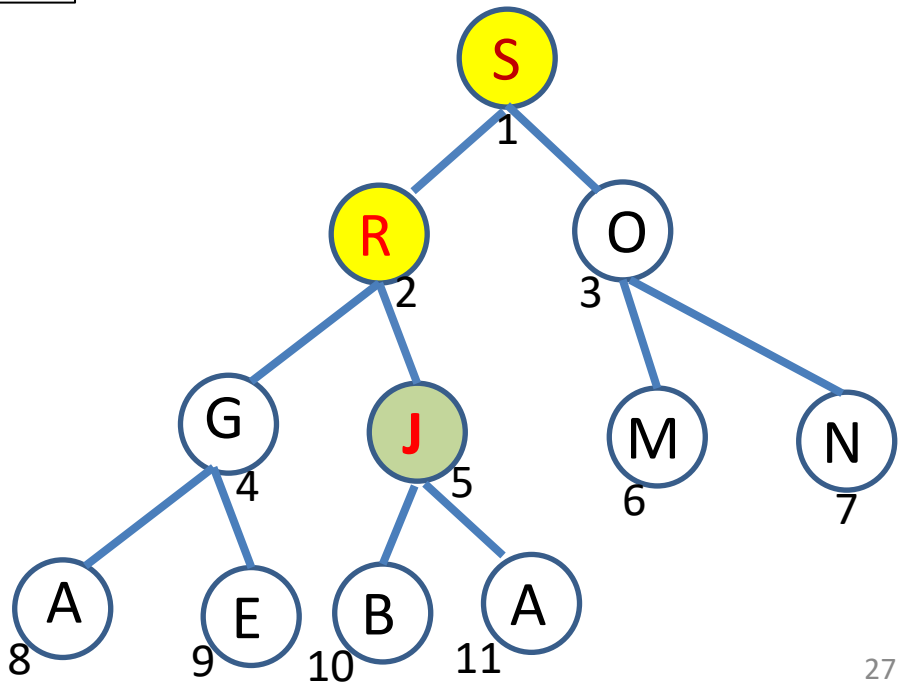
N is 11

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**	

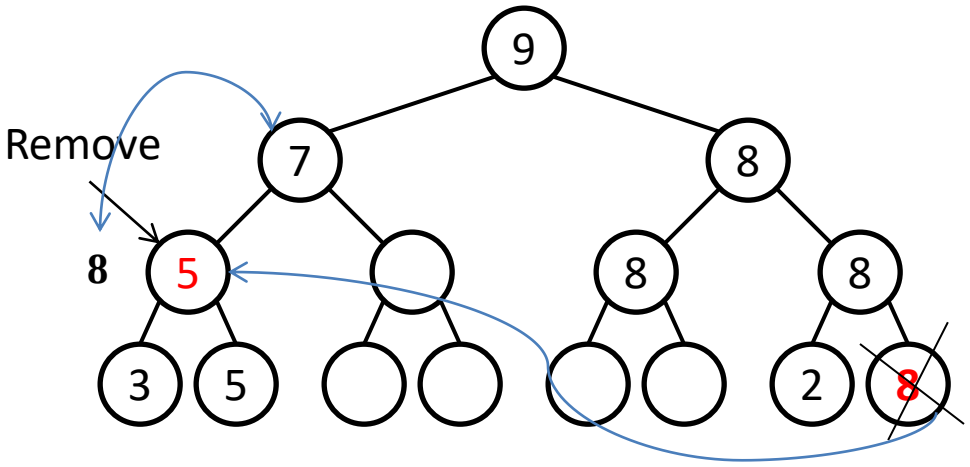
```
removeMax(A,&N)  // Θ(lgN)
  mx = A[1]
  A[1] = A[(*N)]
  (*N)=(*N)-1 //permanent
  //Sink down from index 1
  sinkDown(A,1,N)  //to do
  return mx
```

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)***	





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.

Build a heap from an arbitrary array

- 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

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

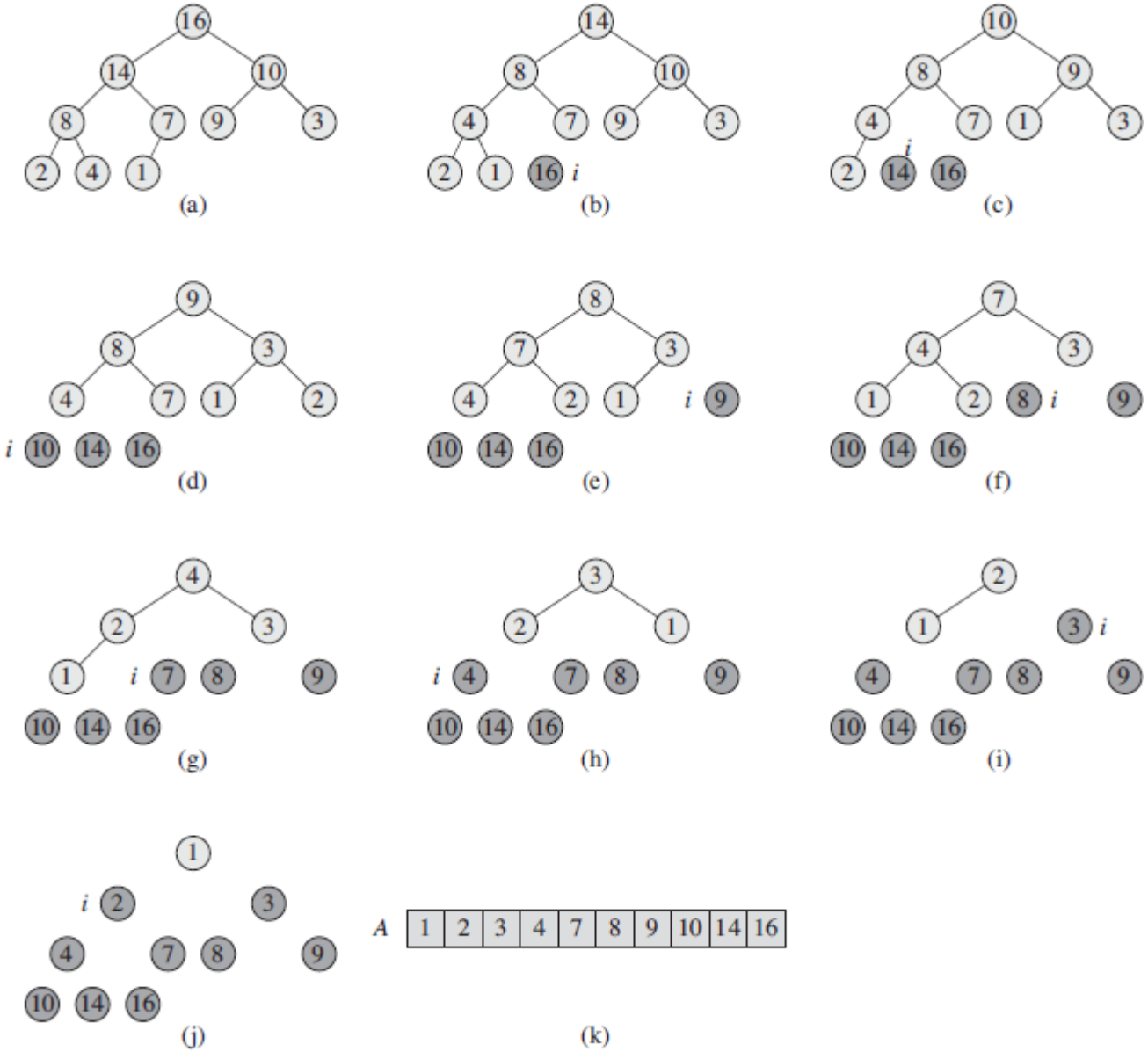
- Solving the recurrence one can show that **the time complexity is O(n)**

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. (b)–(j) The max-heap just after each call of MAX-HEAPIFY, 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

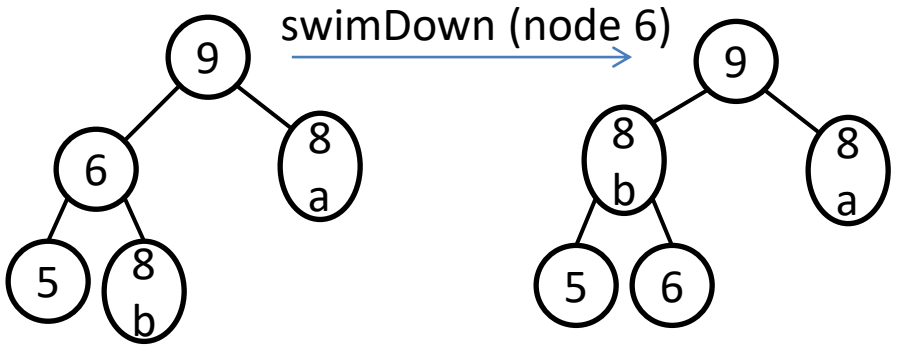
- Both operations are unstable:
  - swimDown (sinkDown)
  - Going from the built heap to the sorted array (remove max and put at the end)

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

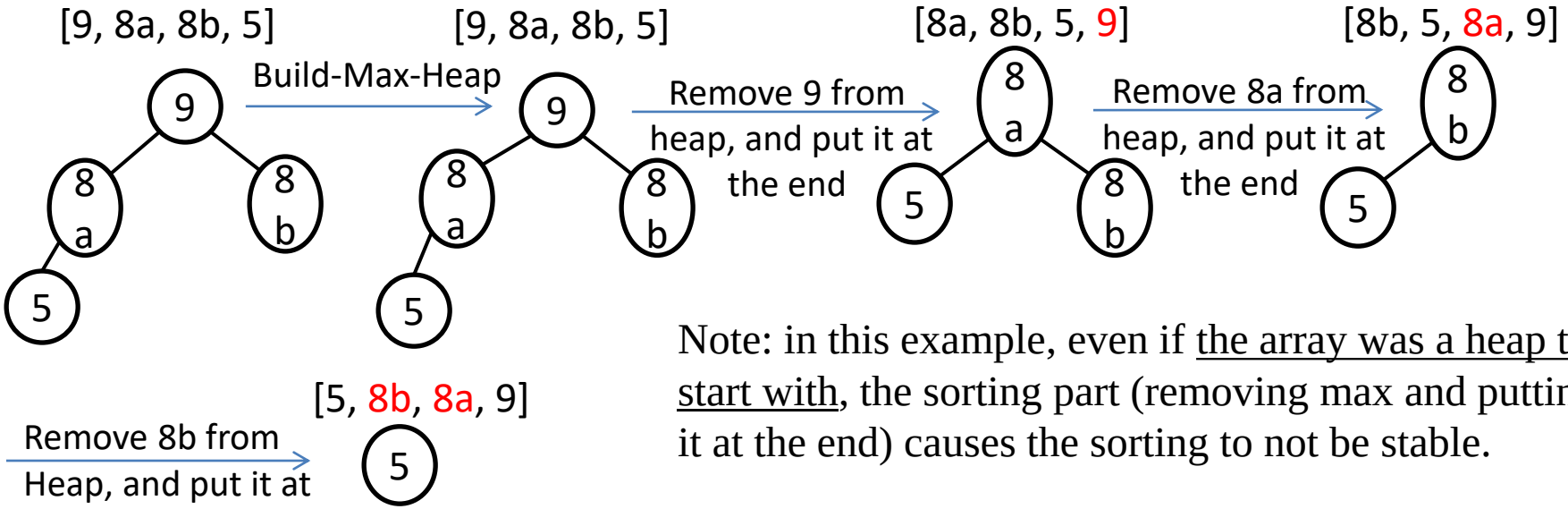
```
sinkDown(A,p,N)
    left = 2*p          // index of left child of p
    right = (2*p)+1 // index of right child of p
    index=p
    if (left≤(*N)&&(A[left]>A[index])
        index = left
    if (right≤(*N))&&(A[right]>A[index])
        index = right
    if (index!=p) {
        swap A[p] <-> A[index]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

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)

Top k Largest with Max-Heap

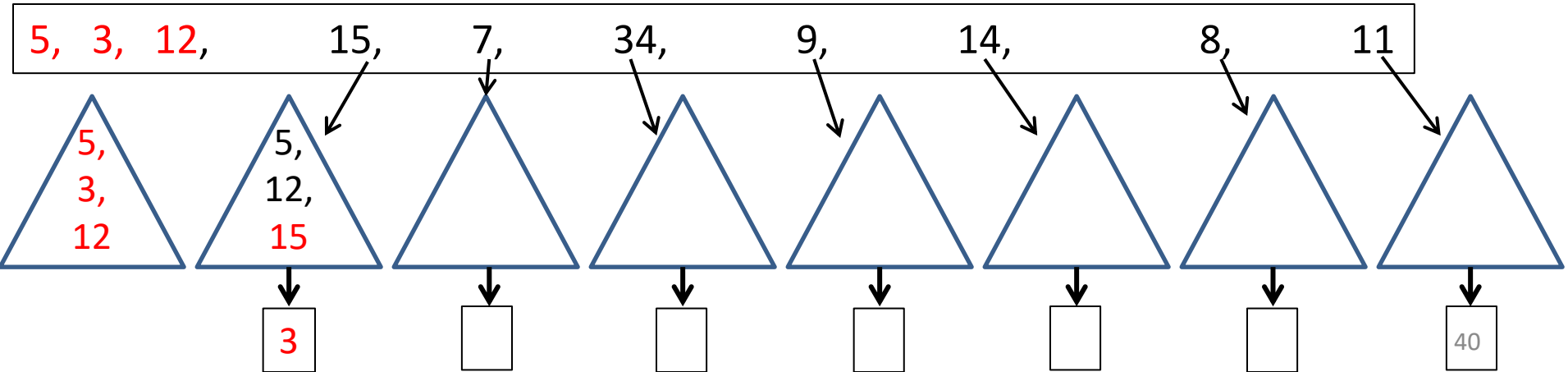
- Input: N = 10, k = 3, array: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)
- Method:
  - Build a max heap using bottom-up
  - Delete/remove 3 (=k) times from that heap
    - **What numbers will come out?**
- Show all the steps (even those for bottom-up build heap). Draw the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

Max-Heap Method Worksheet

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

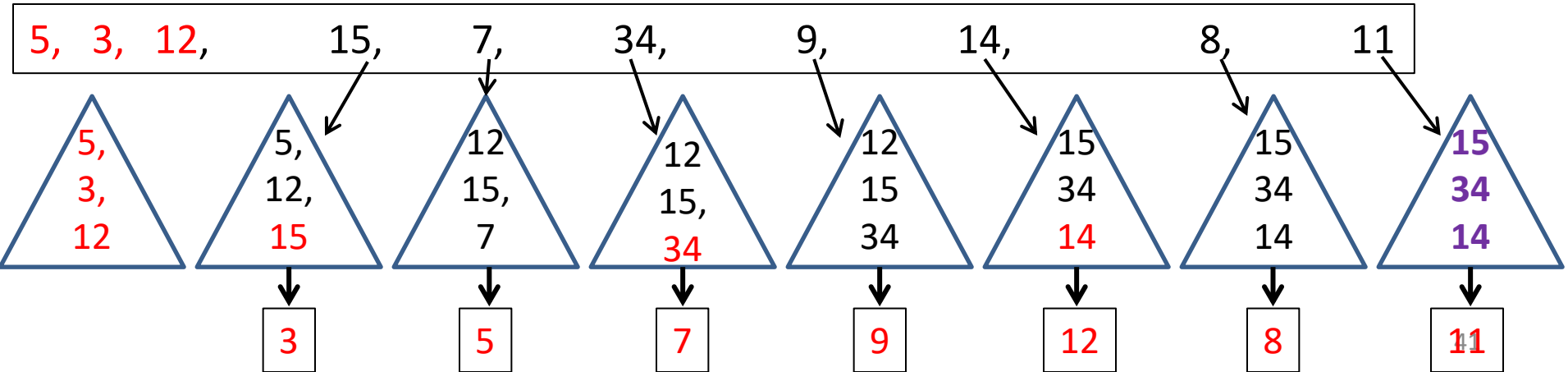
Top k Largest with Min-Heap Worksheet

- 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.

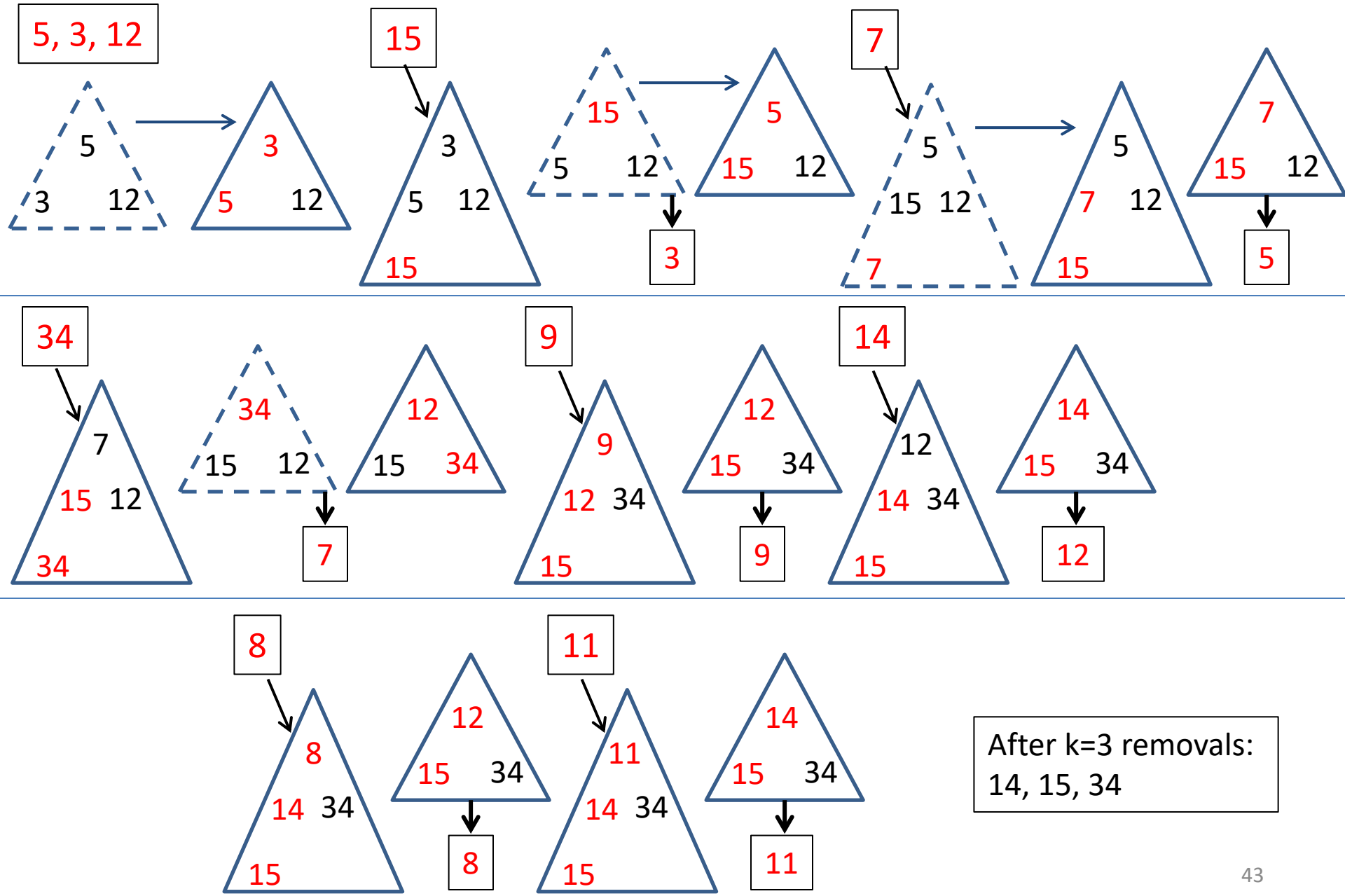


Top k Largest with Min-Heap

- 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.

Binomial heaps, Fibonacci heaps, & Applications

Binomial heaps, Fibonacci heaps, & Applications

Fibonacci Heaps

- The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "**mergeable heap.**" Second, several Fibonacci-heap operations **run in constant amortized time**, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable Heap

A. A mergeable heap is any data structure that supports the following five operations, in which each element has a key:
   1. MAKE-HEAP() creates and returns a new heap containing no elements.
   2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
   3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
   4. EXTRACT (DELETE) -MIN (H) **deletes** the element from heap H whose key is minimum, returning a pointer to the element.
   5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

B. In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:
   1) DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.
   2) DELETE.H (H, x) deletes element x from heap H.

Note: We assume our default mergeable heaps are mergeable min-heaps.

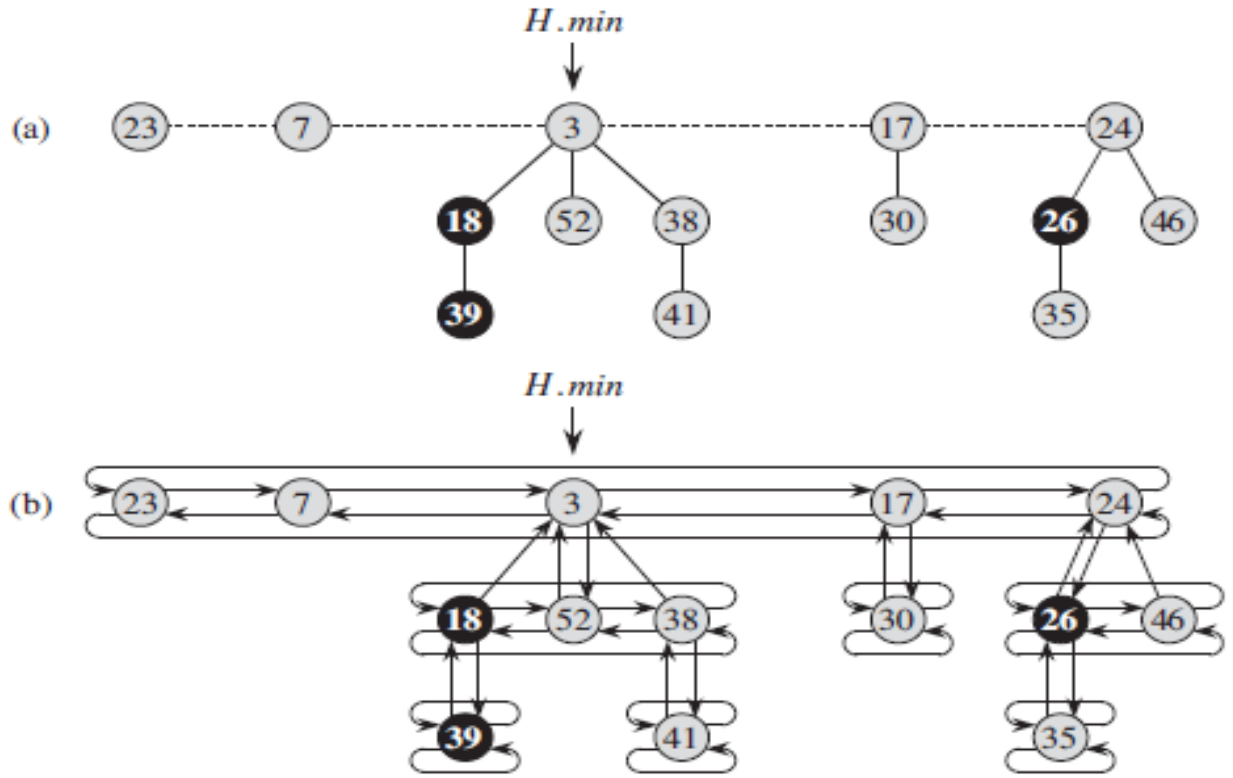
Running times for operations on Mergeable heaps

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n.

Structure of Fibonacci Heaps

- A Fibonacci heap is a collection of rooted trees that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. Figure (a) shows an example of a Fibonacci heap.
- As Figure (b) shows, each node x contains a pointer x.p [up arrows] to its parent and a pointer x.child to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the child list of x. Each child y in a child list has pointers y.left and y.right that point to y's left and right siblings, respectively. If node y is an only child, then y.left = y.right = y. Siblings may appear in a child list in any order.

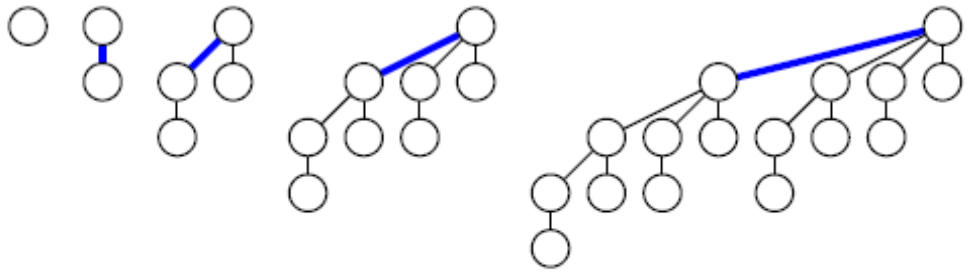


Structure of Fibonacci Heaps (contd.)

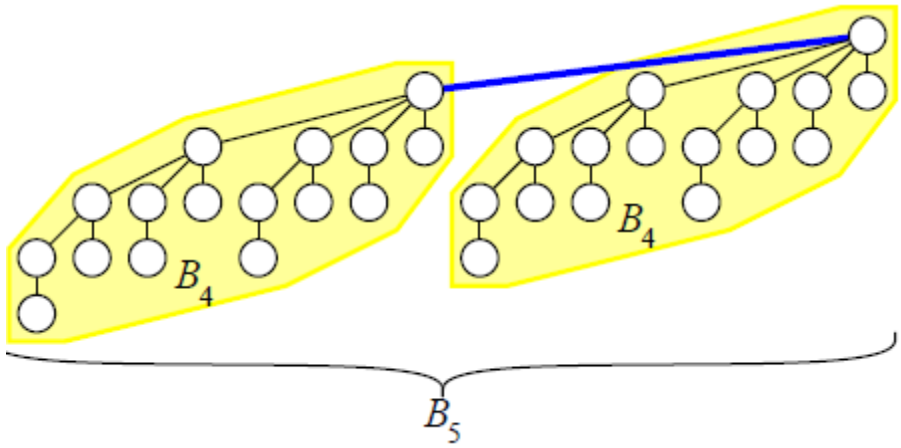
- Circular, doubly linked lists have two advantages for use in Fibonacci heaps.
  - First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in O(1) time.
  - Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in O(1) time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally.
- Each node has two other attributes: x.degree [# of children of x], The boolean-valued attribute x.mark [to indicate whether node x has lost a child since the last time x was made the child of another node].
- Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Initially, set all mark attributes to FALSE.
- We access a given Fibonacci heap H by a pointer H.min to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap. If more than one root has a key with the minimum value, any such root may serve as the minimum node. When a Fibonacci heap H is empty, H.min is NIL.
- The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap. The pointer H.min thus points to the node in the root list whose key is minimum. [Trees may appear in any order within a root list].
- We rely on another attribute for a Fibonacci heap H, H.n, the number of nodes currently in H.

Binomial Trees and Fibonacci Heaps

- Fibonacci heap is a circular doubly linked list, with a pointer to the minimum key, but the elements of the list are not single keys. Instead, we collect keys together into structures called binomial heaps. Binomial heaps are trees that satisfy the heap property – every node has a smaller key than its children | and have the following special structure (next page).



Binomial trees of order 0 through 5.

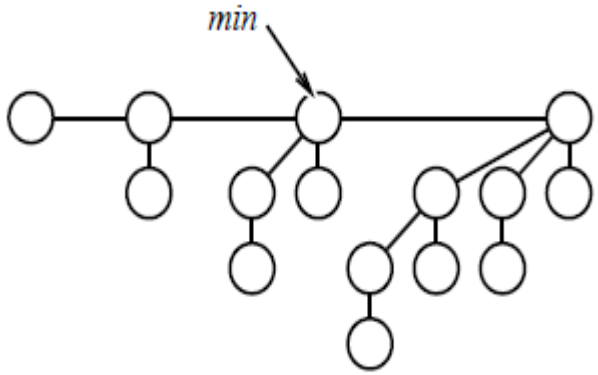


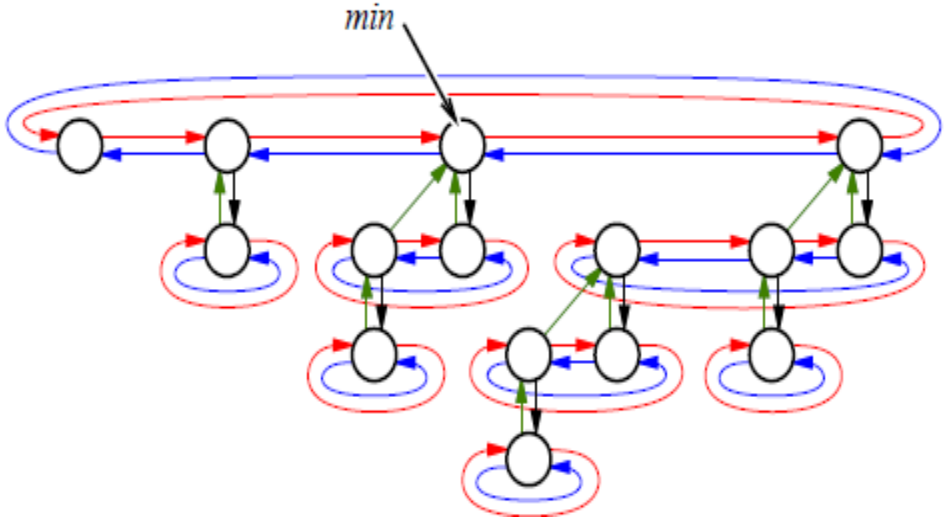
Properties of Binomial Trees

- The root of B_k has degree k.
- The children of the root of B_k are the roots of $B_0, B_1, \ldots, B_{k-1}$.
- B_k has height k.
- B_k has 2^k nodes.
- B_k can be obtained from B_{k-1} by adding a new child to every node.
- B_k has $\binom{k}{d}$ nodes at depth d, for all $0 \leq d \leq k$.
- B_k has 2^{k-h-1} nodes with height h, for all $0 \leq h < k$, and one node (the root) with height k.

Note: Every node in a Fibonacci heap points to four other nodes: its parent, its `next' sibling, its "previous" sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked root list. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.

Example





- With this representation, we can add or remove nodes from the root list, merge two root lists together, link one two binomial tree to another, or merge a node's list of children with the root list, in constant time, and we can visit every node in the root list in constant time per node.
- Having established that these primitive operations can be performed quickly, we never again need to think about the low-level representation details.
- Every node in a Fibonacci heap points to four other nodes: its parent, its `next' sibling, its `previous' sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked root list. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.

Operations on Fibonacci Heaps

- The **Insert, Merge, and FindMin** algorithms for Fibonacci heaps are exactly like the corresponding algorithms for linked lists.
- Since we maintain a pointer to the minimum key, **FindMin** is trivial.
- To insert a new key, we add a single node (which we should think of as a B_0) to the root list and (if necessary) update the pointer to the minimum key.
- To **merge** two Fibonacci heaps, we just merge the two root lists and keep the pointer to the smaller of the two minimum keys. Clearly, all three operations take O(1) time.
- DeleteMin is a bit more complicated.
  1. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes O(1) time.
  2. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $\Theta(n)$ time in the worst case.
  3. To bring down the amortized deletion time, we apply a **Cleanup algorithm**, which links pairs of equal-size binomial heaps until there is only one binomial heap of any size.

Cleanup Algorithm

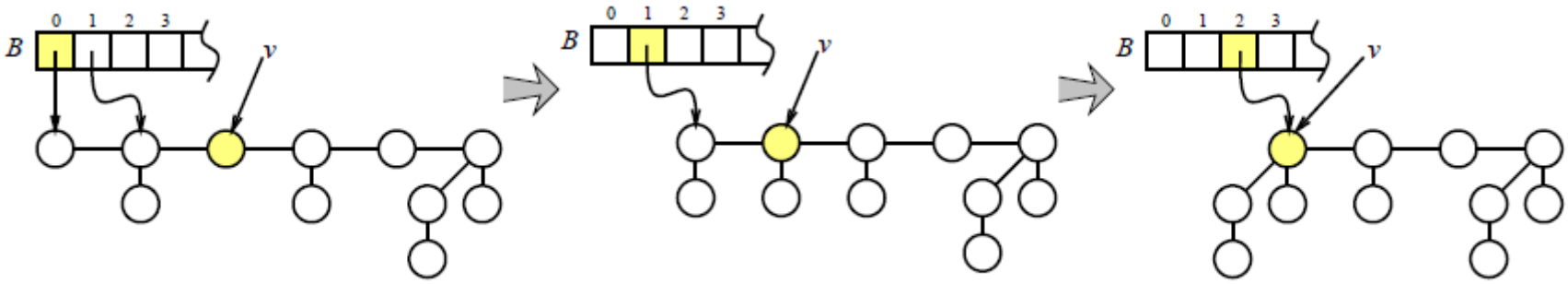
- We will see the Cleanup algorithm in more detail, so we can analyze its running time.
- The following algorithm maintains a global array B[1…⌊lg n⌋], where B[i] is a pointer to some previously-visited binomial heap of order i, or Null if there is no such binomial heap. Notice that Cleanup simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. We split off the part of the algorithm that merges binomial heaps of the same order into a separate subroutine MergeDupes.

```
CLEANUP:
  newmin ← some node in the root list
  for i ← 0 to ⌊lg n⌋
      B[i] ← NULL

  for all nodes v in the root list
      parent(v) ← NULL    (⋆)
      if key(newmin) > key(v)
          newmin ← v
      MERGEDUPES(v)
```

```
MERGEDUPES(v):
  w ← B[deg(v)]
  while w ≠ NULL
      B[deg(v)] ← NULL
      if key(v) ≤ key(w)
          swap v ⇆ w
      remove w from the root list    (⋆⋆)
      link w to v
      w ← B[deg(v)]
  B[deg(v)] ← v
```

Cleanup Algorithm



MERGEDUPES(v), ensuring that no earlier root has the same degree as v.

- Notice that MergeDupes is careful to merge heaps so that the heap property is maintained – the heap whose root has the larger key becomes a new child of the heap whose root has the smaller key. This is handled by swapping v and w if their keys are in the wrong order.
- The running time of Cleanup is O(r'), where r' is the length of the root list just before Cleanup is called. The easiest way to see this is to count the number of times the two starred lines can be executed: line (*) is executed once for every node v on the root list, and line (**) is executed at most once for every node w on the root list. Since DeleteMin does only a constant amount of work before calling Cleanup, the running time of DeleteMin is O(r') = O(r + deg(min)) – NOT GOOD. where **r is the number of roots before DeleteMin begins, and min is the node deleted.**
- **Note:** Although deg(min) is at most lg n, we can still have r = Θ(n) (for example, if nothing has been deleted yet), so the worst-case time for a DeleteMin is Θ(n). After a DeleteMin, the root list has length O(log n)

Amortized Analysis of DeleteMin

- To bound the amortized cost, observe that each insertion increments r. If we charge a constant `cleanup tax' for each insertion and use the collected tax to pay for the Cleanup algorithm, the unpaid cost of a DeleteMin is only O(deg(min)) = O(log n).
- More formally, **define the potential of the Fibonacci heap to be the number of roots**. Recall that the amortized time of an operation can be defined as its actual running time plus the increase in potential, provided the potential is initially zero (it is) and we never have negative potential (we never do). Let r be the number of roots before a DeleteMin and let r'' denote the number of roots afterwards. The actual cost of DeleteMin is r + deg(min), and the number of roots increases by r'' – r, so the amortized cost is r'' + deg(min). Since r'' = O(log n) and the degree of any node is O(log n), **the amortized cost of DeleteMin is O(log n).**
- Each **Insert** adds only one root, so its amortized cost is still constant. A Merge doesn't change the number of roots, since the new Fibonacci heap has all the roots from its constituents and no others, so its **amortized cost is also constant**.

Decreasing Keys

- In some applications of heaps, we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$, and then use DeleteMin. Here is an algorithm to decrease the key of a node in a Fibonacci heap; the algorithm will take O(log n) time in the worst case, but the amortized time will be only O(1). Our algorithm for decreasing the key at a node v follows two simple rules.
  - 1. Promote v up to the root list. (This moves the whole subtree rooted at v.)
  - 2. As soon as two children of any node w have been promoted, immediately promote w.
- In order to enforce the second rule, we now mark certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the only way to unmark a node. (Specifically, splicing nodes into the root list during a DeleteMin is not considered a promotion.)
- We provide a more formal description of the algorithm next. The input is a pointer to a node v and the new value k for its key.

Decreasing Keys

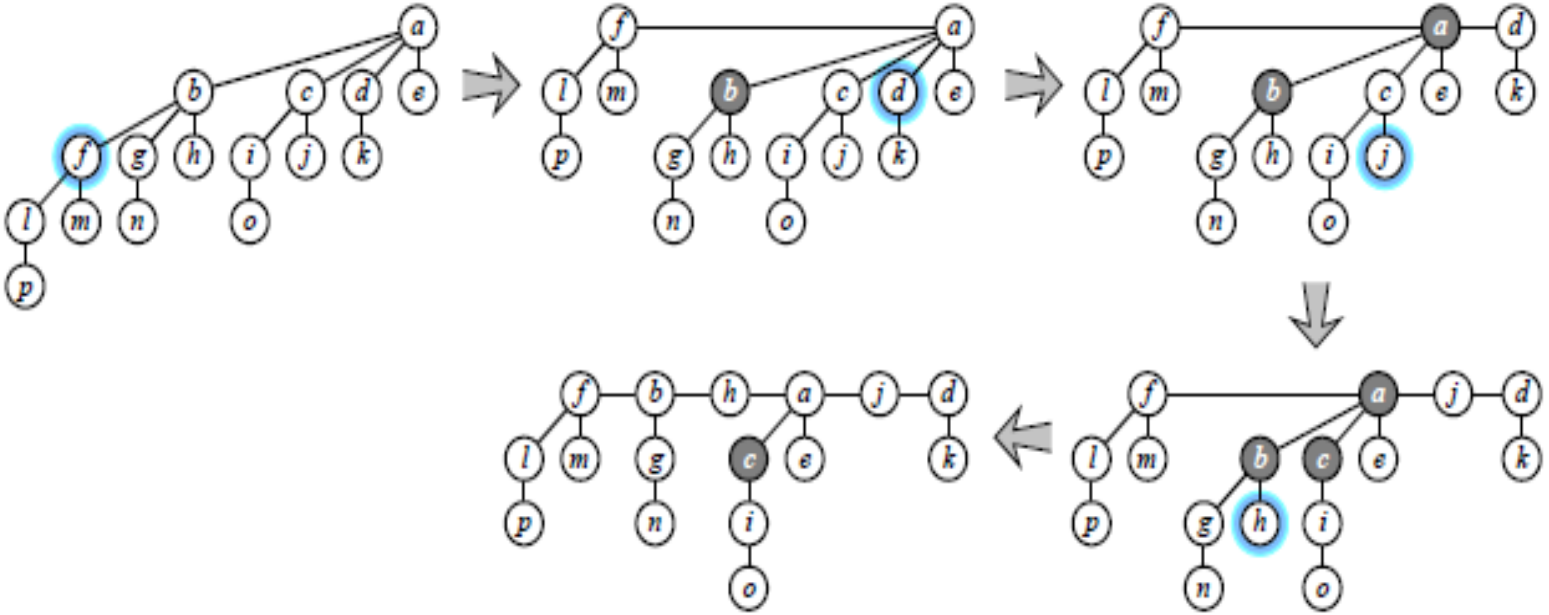
- The input is a pointer to a node v and the new value k for its key.

```
DecreaseKey(v, k):
    key(v) ← k
    update the pointer to the smallest key
    Promote(v)
```

```
Promote(v):
    unmark v
    if parent(v) ≠ Null
        remove v from parent(v)'s list of children
        insert v into the root list
        if parent(v) is marked
            Promote(parent(v))
        else
            mark parent(v)
```

- The Promote algorithm calls itself recursively, resulting in a `cascading promotion'. Each consecutive marked ancestor of v is promoted to the root list and unmarked, otherwise unchanged.
- The lowest unmarked ancestor is then marked, since one of its children has been promoted.
- The time to decrease the key of a node v is O(1+#consecutive marked ancestors of v). Binomial heaps have logarithmic depth, so if we still had only full binomial heaps, the running time would be O(log n). Unfortunately, promoting nodes destroys the nice binomial tree structure; our trees no longer have logarithmic depth! In fact, DecreaseKey runs in Θ(n) time in the **worst** case.

Example



Decreasing the keys of four nodes: first f, then d, then j, and finally h. Dark nodes are marked. DECREASEKEY(h) causes nodes b and a to be recursively promoted.

Amortized cost of Decreasing Keys

- To compute the amortized cost of DecreaseKey, we'll use the potential method, just as we did for DeleteMin.
- We need to find a potential function Φ that goes up a little whenever we do a little work and goes down a lot whenever we do a lot of work.
- DecreaseKey unmarks several marked ancestors and possibly also marks one node. So, the number of marked nodes might be an appropriate potential function here.
  - Whenever we do a little bit of work, the number of marks goes up by at most one; whenever we do a lot of work, the number of marks goes down a lot.
  - More precisely, let m and m′ be the number of marked nodes before and after a DecreaseKey
  - operation. The actual time (ignoring constant factors) is t = 1 + #consecutive marked ancestors of v and if we set Φ = m, the increase in potential is
  - $m' - m \leq 1 -$ #consecutive marked ancestors of v.
  - Since $t + \Delta\Phi \leq 2$, **the amortized cost of DecreaseKey is O(1)**.

Bounding the Degree

- Now we have a problem with our earlier analysis of DeleteMin. The amortized time for a DeleteMin is still O(r+deg(min)). To show that this equaled O(log n), we used the fact that the maximum degree of any node is O(log n), which implies that after a Cleanup the number of roots is O(log n). But now that we don't have complete binomial heaps, this `fact' is no longer obvious!
- So, let's prove it. For any node v, let |v| denote the number of nodes in the subtree of v, including v itself. Our proof uses the following lemma, which finally tells us why these things are called Fibonacci heaps.
- **Lemma:** For any node v in a Fibonacci heap, $|v| \geq F_{deg(v)+2}$.
- **Proof:** [Skip the proof if you want to] Label the children of v in the chronological order in which they were linked to v. Consider
- the situation just before the i^{th} oldest child w_i was linked to v. At that time, v had at least i – 1
- children (possibly more). Since Cleanup only links trees with the same degree, we had $deg(w_i) = deg(v) \geq i - 1$. Since that time, at most one child of w_i has been promoted away; otherwise, w_i would have been promoted to the root list by now. So currently we have $deg(w_i) \geq i - 2$.
- We also quickly observe that $deg(w_i) \geq 0$.

Bounding the Degree (contd.)

- Let s_d be the minimum possible size of a tree with degree d in any Fibonacci heap. Clearly
- $s_0 = 1$; for notational convenience, let $s_{-1} = 1$ also. By our earlier argument, the i^{th} oldest child of the root has degree at least $\max\{f_0, i-2\}$, and thus has size at least $\max(1, s_{i-2}) = s_{i-2}$. Thus, we have the following recurrence:

$$s_d \geq 1 + \sum_{i=1}^{d} s_i - 2$$

- If we assume inductively that $s_i \geq F_{i+2}$ for all $-1 \leq i < d$ (with the easy base cases $s_{-1} = F_1$ and $s_0 = F_2$), we have [Remember, by definition $|v| \geq s_{deg(v)}$]

$$s_d \geq 1 + \sum_{i=1}^{d} F_i = F_{d+2}$$

- We can easily show, by using induction, that $F_{k+2} > \phi^k$ where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. Thus, by our previous lemma, w.e get $deg(v) \leq \log_\phi |v| = O(\log |v|)$.
- Thus, since the size of any subtree in an n-node Fibonacci heap is obviously at most n, the degree of any node is O(log n), which is exactly what we wanted. Our earlier analysis is still good.

Analyzing Everything Together

- Unfortunately, our analyses of DeleteMin and DecreaseKey used two different potential functions. Unless we canfind a single potential function that works for both operations, we can't claim both amortized time bounds simultaneously. So, we need to find a potential function Φ that goes up a little during a cheap DeleteMin or a cheap DecreaseKey and goes down a lot during an expensive DeleteMin or an expensive DecreaseKey.
- Let's look a little more carefully at the cost of each Fibonacci heap operation, and its effect on both the number of roots and the number of marked nodes, the things we used as out earlier potential functions. Let r and m be the numbers of roots and marks before each operation and let r' and m' be the numbers of roots and marks after the operation.

operation	actual cost	$r' - r$	$m' - m$
INSERT	1	1	0
MERGE	1	0	0
DELETEMIN	$r + \deg(min)$	$r' - r$	0
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$

- If we guess that the correct potential function is a linear combination of our old potential functions r and m and play around with various possibilities for the coefficients, we will eventually stumble across the correct answer: **$\phi = r + 2m$**

Analyzing Everything Together

- To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

operation	actual cost	$r' - r$	$m' - m$	$\Phi' - \Phi$	amortized cost
INSERT	1	1	0	1	2
MERGE	1	0	0	0	1
DELETEMIN	$r + \deg(\min)$	$r' - r$	0	$r' - r$	$r' + \deg(\min)$
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$	$1 + m' - m$	2

- Since Lemma 1 implies that r0 + deg(min) = O(log n), we're finally done!

Fibonacci Trees

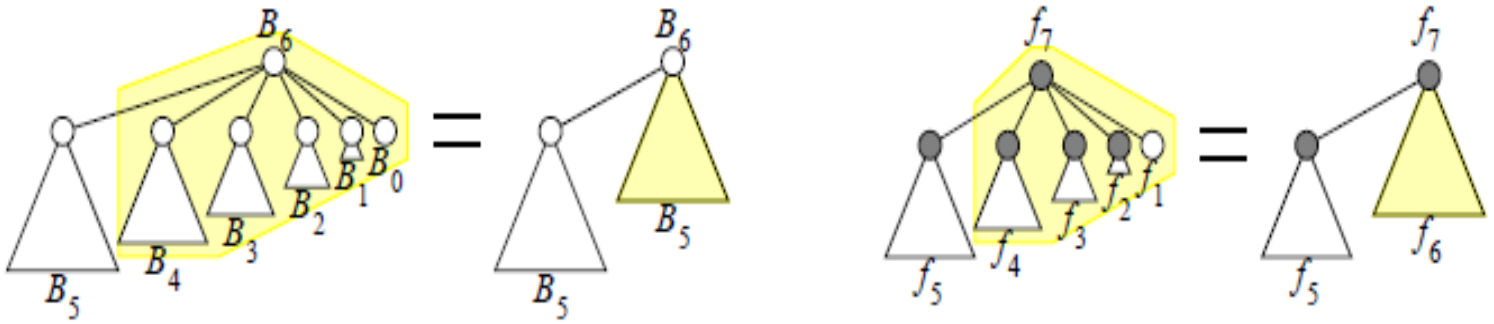
- To get a little more intuition about how Fibonacci heaps behave, let's look at a worst-case construction for our Lemma.
- Suppose we want to remove as many nodes as possible from a binomial heap of order k, by promoting various nodes to the root list, but without causing any cascading promotions. The most damage we can do is to promote the largest subtree of every node. Call the result a Fibonacci tree of order k + 1 and denote it f_{k+1}. As a base case, let f_1 be the tree with one (unmarked) node, that is, $f_1 = B_0$. The reason for shifting the index should be obvious after a few seconds.



Fibonacci trees of order 1 through 6. Light nodes have been promoted away; dark nodes are marked.

Fibonacci Trees

- Recall that the root of a binomial tree B_k has k children, which are roots of B_0, B_1 …, B_{k-1}. To convert B_k to f_{k+1}, we promote the root of B_{k-1}, and recursively convert each of the other subtrees B_i to f_{i+1}. The root of the resulting tree f_{k+1} has degree k .. 1, and the children are the roots of smaller Fibonacci trees f_1, f_2, … , f_{k-1}. We can also consider B_k as two copies of B_{k-1} linked together. It's quite easy to show that an order-k Fibonacci tree consists of an order k – 2 Fibonacci tree linked to an order k – 1 Fibonacci tree.



Comparing the recursive structures of B_6 and f_7.

- Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the kth Fibonacci number! Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the k^{th} Fibonacci number!

Properties of Fibonacci Trees

- Like binomial trees, Fibonacci trees have lots of other nice properties that easy to prove by induction.

  - The root of f_k has degree $k-2$.
  - f_k can be obtained from f_{k-1} by adding a new unmarked child to every marked node and then marking all the old unmarked nodes.
  - f_k has height $\lceil k/2 \rceil - 1$.
  - f_k has F_{k-2} unmarked nodes, F_{k-1} marked nodes, and thus F_k nodes altogether.
  - f_k has $\binom{k-d-2}{d-1}$ unmarked nodes, $\binom{k-d-2}{d}$ marked nodes, and $\binom{k-d-1}{d}$ total nodes at depth d, for all $0 \le d \le \lfloor k/2 \rfloor - 1$.
  - f_k has F_{k-2h-1} nodes with height h, for all $0 \le h \le \lfloor k/2 \rfloor - 1$, and one node (the root) with height $\lceil k/2 \rceil - 1$.

Potential function

- We use the potential method to analyze the amortized performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by t(H) the **number of trees in the root list** of H and by m(H) the **number of marked nodes** in H. We then define the potential Φ of Fibonacci heap H by $\Phi(H) = t(H) + 2m(H)$.
- The potential of the Fibonacci heap shown is 5 + 2.3 =11. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We assume that a unit of potential can pay for a constant amount of work.
- We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and so, the potential is nonnegative at all subsequent times.
- An upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.

