COMP20003 Workshop Week 9

Priority Queue
Heaps & Binary Heaps
Heap Sort

LAB:

- Heap Sort

Important: Bring papers and pens to the last 3 workshops!

ADT: Queue vs. Priority Queue

rear front

Queue: elements are dequeued in the FIFO order.

enqueue, dequeue can be easily implemented with O(1) time complexity using linked lists

'*Can I borrow your baby?...*'

Priority Queue : element with highest priority must be dequeued first!

How can we efficiently implement dequeue (and enqueue)?

Yet Another ADT: Priority Queue

PQ: queue, where each element is associated with a *priority* (or *weight*), and the elements will be *dequeued* following the order of priority.

'Can I borrow your baby?...'

Main operations:

- ***enqueue***: `enPQ(PQ, item)` (supposing easy access to weight inside item), or
  `enPQ(PQ, weight, item)`
- ***dequeue***: `dePQ(PQ)` removes & returns the highest-priority element of `PQ`. Normally named as
  `deleteMax(PQ)`, if *higher priority* means *bigger*, or
  `deleteMin(PQ)`, if *higher priority* means *smaller*
- ***changeWeight***: change the weight of an item: `changeWeight(PQ, item)`
  or : `changeWeight(PQ, newWeight, item)`
- ***create*** : `makePQ()` – make an empty `PQ` or create a `PQ` from a set of items
- *check for being empty*: `isEmptyPQ(PQ)`

possible concrete data structures for PQ

Concrete Data Structure	Time complexity of			
	construction a PQ of n elements	enPQ	dePQ	peek
unsorted arrays or linked list				
sorted arrays or linked lists				
BST				
AVL				
hash table				

Example: priority= max

Unsorted array/list: 9 2 7 5 6 8 3

Sorted array/list: 2 3 5 6 7 8 9

Related:
Ex 9.4

check your answers: possible concrete data structures for PQ

Concrete Data Structure	Time complexity of				Notes
	make PQ of n elements	enPQ	dePQ	peek	
unsorted arrays or linked list	**O(n)**	**O(1)**	O(n)	O(n)	
sorted arrays or linked lists	O(n logn)	O(n)	**O(1)**	**O(1)**	
BST	*the worst cases are the same as in sorted linked lists*				
AVL	O(nlogn)	O(logn)	O(logn)	O(logn)	space-inefficient (and poor cache locality) compared to Heap
hash table (using the priority as a key when hashing)	*Generally not efficient:* **O(n)** time for dequeue because a hash table is an *unordered data structure*. To find the highest-priority element (for peek or dequeue), the entire hash table must be scanned to locate the maximum value.				
Heap or **Binary Heap**	Θ(n)	O(logn)	O(logn)	O(1)	space efficient (great cache locality)

Peer Activity: Hashing Priorities?

Can an efficient priority queue that prioritises lower distances be implemented on a hash table with these constraints? Why?

a. Yes, it can.

b. No, it cannot.

Consider the following **constraints:**

- keys

  supposing r is small

  - range is known ($0 \leq$ key $\leq r$)
  - hashed with the function `hash(key) = key`
- hash table

Index	0	1	...	r - 1	r
Key	...		...		

...

Peer Activity: Hashing Priorities?

Can an efficient priority queue that prioritises lower distances be implemented on a hash table with these constraints? Why?

a. Yes, it can.

b. No, it cannot.

Consider the following **constraints:**

- keys
  - unbounded ($\mathbb{N}_0$)
  - hashed with the function `hash(key) = key % 101`
- hash table

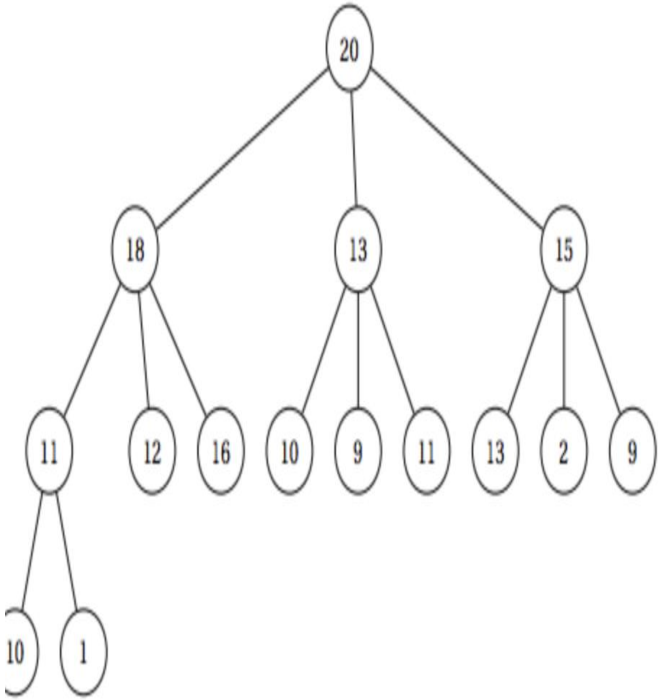
Index	0	1	...	99	100
Key	...		...		

...

Binary Heap – An Efficient Data Structure for PQ

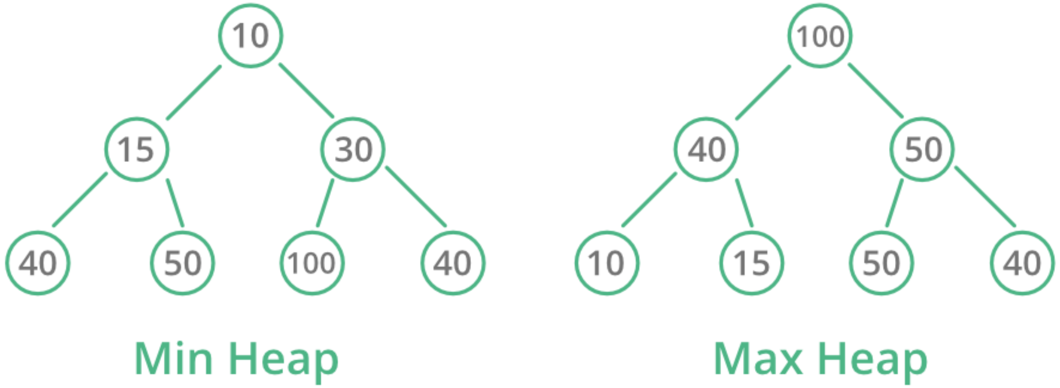
Heap?

Ternary Heap

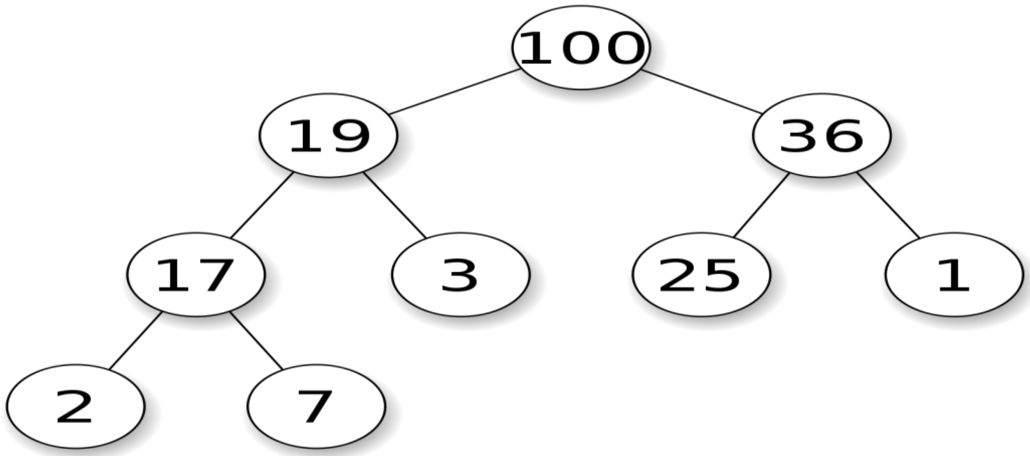


Binary heap: an efficient implementation for priority queue

Depending on the priority, we can have min-heap or max-heap



Heap: requirements

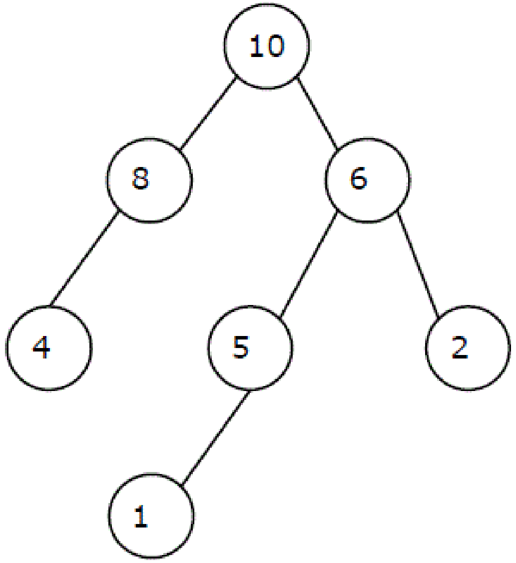
Example	**Conditions**
	1. The tree is complete. 2. *The heap property*: each node has a higher priority (here, is not smaller) than any of its descendants (or equivalently, just its children).

20

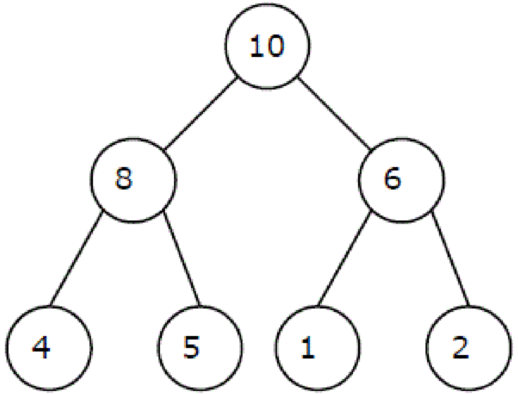
2

which one is a binary heap?

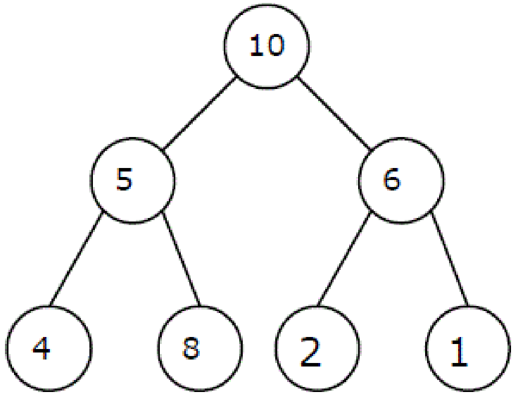
(A)



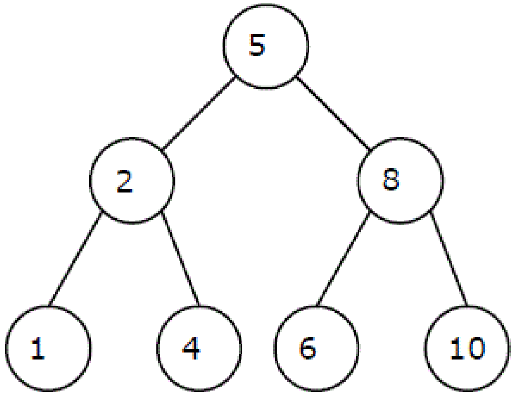
(B)



(C)

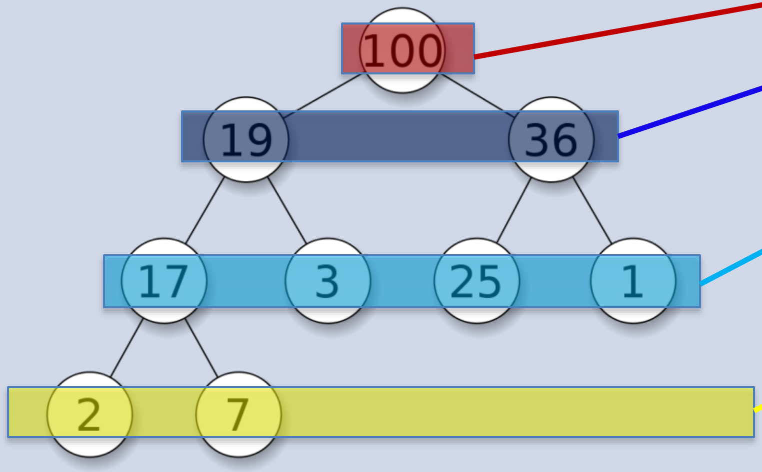


(D)



Binary Heap is implemented as an array!

Visualisation: as a complete binary tree



Implementation: using arrays

idx	0	1	2	3	4	5	6	7	8	9
val	×	100	19	36	17	3	25	1	2	7

note: H[1] is for the root

H[0] not used

Heap is H[1..n]

- level i occupies 2^i cells in array H[1..n] from idex 2^i to 2^{i+1}-1

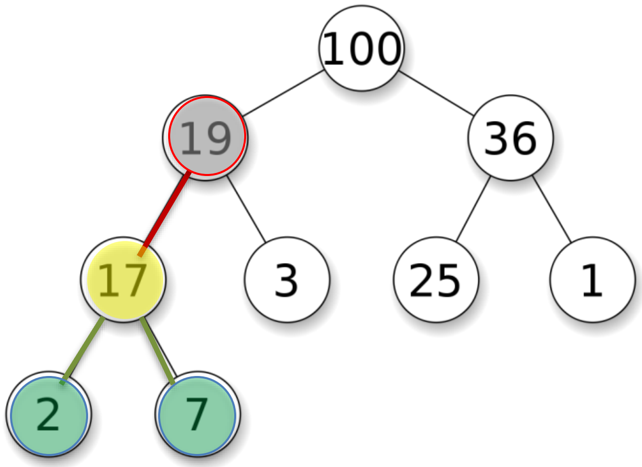
Binary Heap is implemented as an array:
→ efficient locating parent or children of the node at index i

- left child of H[i] is H[2*i]
- right child of H[i] is H[2*i+1]

parent of `H[i]` is `H[i/2]`

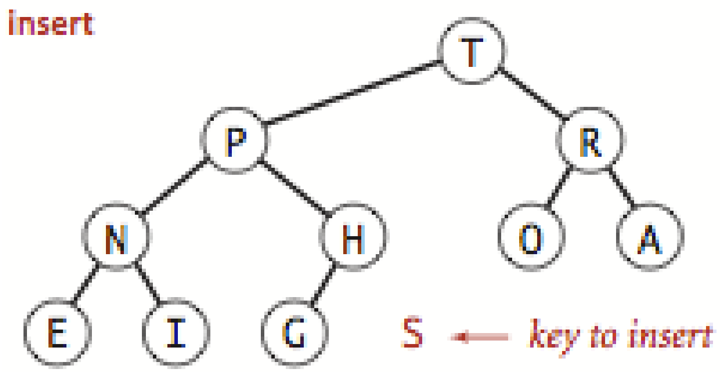
idx	0	1	2	3	4	5	6	7	8	9
val	X	100	19	36	17	3	25	1	2	7

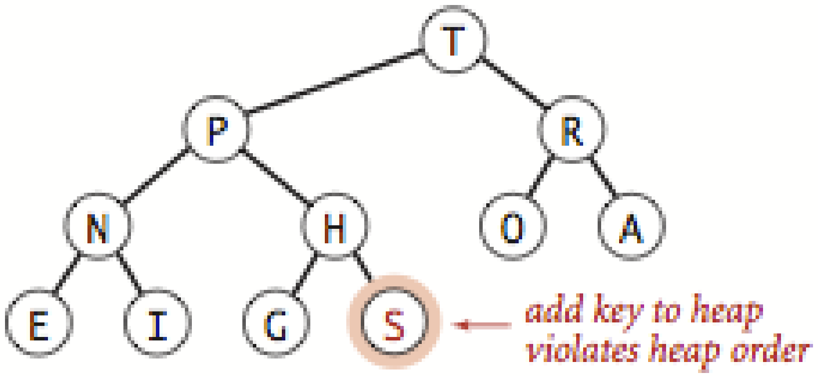
4/2 i=4 4*2 4*2+1



Insert into a heap.

tree visualisation





in the implemented array

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

H= [T,P,R,N,H,O,A,E,I,G,]

H has 10 elements

Insert S

Just added H[11]= S

parent of H[11] is H[11/2] ie. H[5]

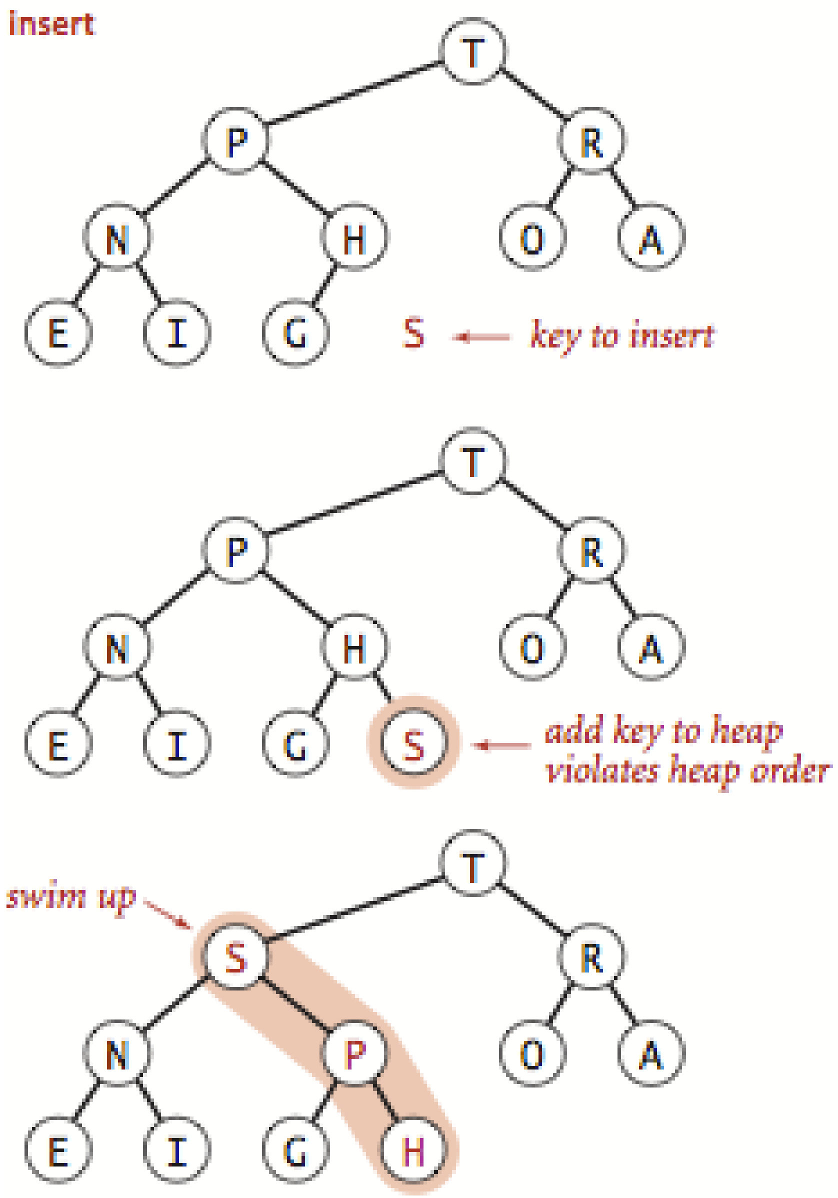
index 5=11/2 11

H= [T,P,R,N,H,O,A,E,I,C,S]

in this case H[11] and its parent H[5] violate the heap order

→ need to repair using **upheap**

Insert a new elem into a heap using upheap. Complexity= ?



upheap

when a child node violates the heap order: *repeatedly swap the child with its parent (if exist) until having no violation*

Complexity: $O(\ ?\ )$, $\theta(\ \ )$

Need to promote H[11] up using **upheap(h,i=11)**, which repeatedly swap node i with its parent.

2/2=1 2=5/2 5=11/2 11

[T,P,R,N,H,O,A,E,I,C,S]

S H

➔[T,P,R,N,S,O,A,E,I,C,H]

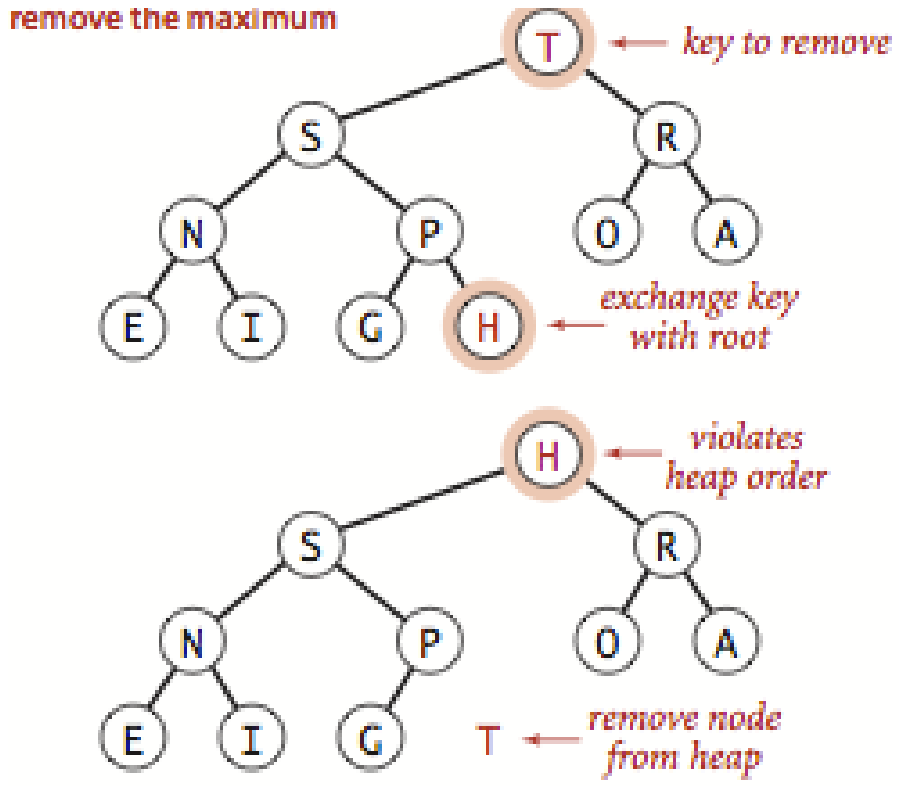
S P

➔[T,S,R,N,P,O,A,E,I,C,S]

S

Complexity= ?

deletemax: delete (and returns) the heaviest. Complexity=

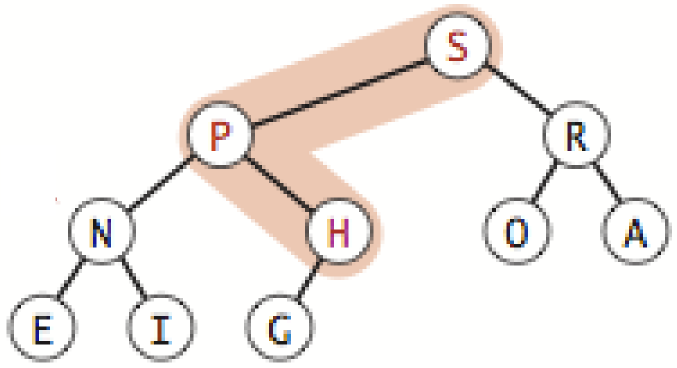


Heap= [T,S,R,N,P,O,A,E,I,G,H]

To remove (the heaviest, the root):

- swap root T with the last leaf H
- decrease number of elements in heap
- new root will likely violate the heap order: repair that by doing **downheap**

deletemax: delete (and returns) the heaviest. Complexity=



downheap= repeatedly swap node with its *heaviest child* until having no violation

Complexity: O(log n)

Notes: Here upheap(H, node) was used for insertion, and downheap(H, node) for deletion. The operations can be performed for any node of the heap.
For example, when changing the priority of a node in a heap.

How to efficiently build a heap with n elements?

- Solution 1: insert each element into the (initially empty) heap, and do upheap after each insertion.

  Complexity: O(?)

How to efficiently build a heap with n elements? heapify

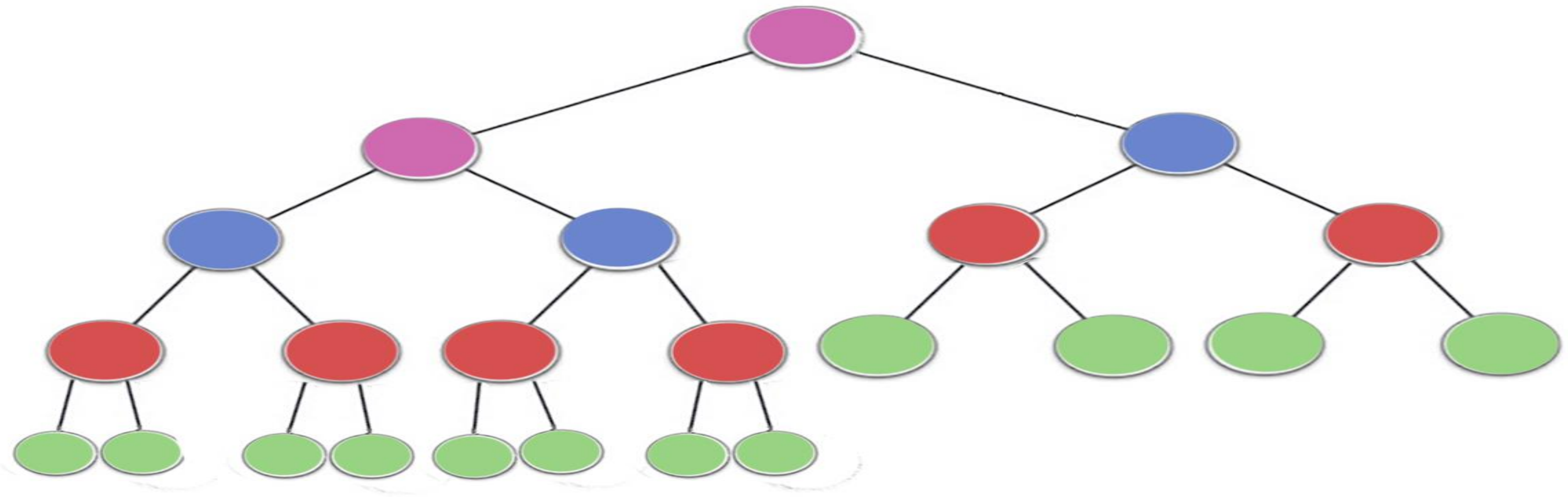
Solution 2: populate the heap array with n elements in the input order, then turn the array to a heap (ie make it to satisfy the heap condition). Algorithm:

```
for (i= ??? ; i>0; i--) {
   // for i from last parent node to first node
      downheap(h, i);
```

Related:
Ex 9.3

= **Θ(n)** (see lectures and/or ask Google for a proof)

The operation is known as Heapify/Makeheap/ Bottom-Up Heap Construction



Summary: Building a heap for n priority values

Heapsort= sorting using a heap

How? Complexity=?

Pre-questions:

- How to selection-sort by selecting the largest first?
- What is the complexity of selection sort?
- How can we have a faster select-the-largest?

Example: sort the keys: 20,3,60, 8,1,16

HeapSort summary

To sort an array A[1..n] in *increasing* order

1. Use heapify to turn A into a *maxheap*
2. while (heap A has more than 1 element):
   - delete root by :
     - first swap it with the last element of the heap, then
     - downheap the new root

Complexity of Heapsort =

= O(?)

Questions:

- What's the best case of heapsort?
- Is heapsort stable?

Example: using Heapsort, sort the keys: 20,3,60,8,1,16

Heapsort= sorting using a heap

How? Complexity=?

Example: sort the keys: 20,3,60, 8,1,16

Step 1: Turn the array to a maxheap using heapify

20 3 60 8 1 16

60 > only child 16 : subtree 60 is a heap

20 3 60 8 1 16

3 < larger child 8 : swap

20 8 60 3 1 16

3 has no child, is a heap

20 8 60 3 1 16

swap 20 with larger child 60

60 8 20 3 1 16

20 > only child 16 : 20 is a heap

done

complexity step 1: Θ(n)

Step 2: loop: a) swap root with the last
decrement heap length
b) downheap(heap, position= 1)

60 8 20 3 1 16 n= 6

1a: swap(60,16)

16 8 20 3 1 60 n= 5

1b: downheap(16) : swap (16,20), done

20 8 16 3 1 60 n= 5

2a: swap(20,1)

1 8 16 3 20 60 n= 4

2b: downheap(1) : swap (1,16), done

16 8 1 3 20 60 n= 4

3ab: swap (16,3), downheap(3)= swap(3,8)

8 3 1 16 20 60 n= 3

4ab: swap (8,1), downheap(1)= swap(1,3)

3 1 8 16 20 60 n= 2

5ab: swap (3,1), downheap(3)= no swap needed

1 3 8 16 20 60 n= 1, done

DONE after 5= n-1 step

complexity step 2= overall = O(n logn)

HeapSort summary

To sort an array A[1..n] in *increasing* order

1. Use heapify to turn A into a *maxheap*
2. while (heap A has more than 1 element):
   - delete root by :
     - first swap it with the last element of the heap, then
     - downheap(A,1): downheap the new root

- Complexity=
- = O(n logn)
- Questions:
  - What's the best case of heapsort? all elements equal, Θ(n)
  - Is heapsort stable? no (long-distance swap)

Heap & Heap Sort: Complexity Summary

Heap operations:

- upheap:
- downheap:
- insert/enPQ:
- deleteMax/deleteMin:
- heapify:
- heapsort:

Heap & Heap Sort: Complexity Summary

Heap operations:

- upheap(H, pos): $O(\log n)$
- downheap(H, pos): $O(\log n)$
- insert/enPQ(H, key): $O(\log n)$
- deleteMax/deleteMin(H): $O(\log n)$

- heapify: $\Theta(n)$
- heapsort: $O(n \log n)$ best case: Θ

Peer Activity: m^{th} Smallest Number

Does the upper-bound complexities of these two algorithms differ?

a. Yes, they do.

b. No, they do not.

Assume that that $m << n$.

Consider an **unsorted algorithm** that:

- gets the m^{th} smallest value
- from n unsorted values

Now consider a **sorted algorithm** that:

- sorts n values in ascending order
- indexes the m^{th} value

Adaptive (aka. Natural) Merge Sort

Bottom-up merge sort improvement

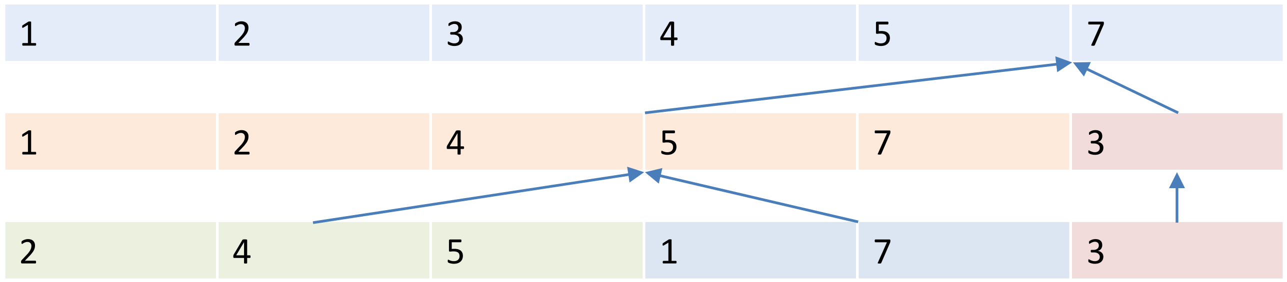
- Monotonic increasing runs already sorted
- Insert monotonic runs into queue instead of singletons

2	4	5	1	7	3

Demonstration – Adaptive Merge Sort

Bottom-up merge sort improvement

- Best Case: Θ()
- Worst Case: Θ()
- If known k= number of monotonic runs: Θ()



Lab

W9.7: Implementing heapsort