COMP20003 Workshop Week 9

Priority Queue
Heaps & Binary Heaps
Heap Sort

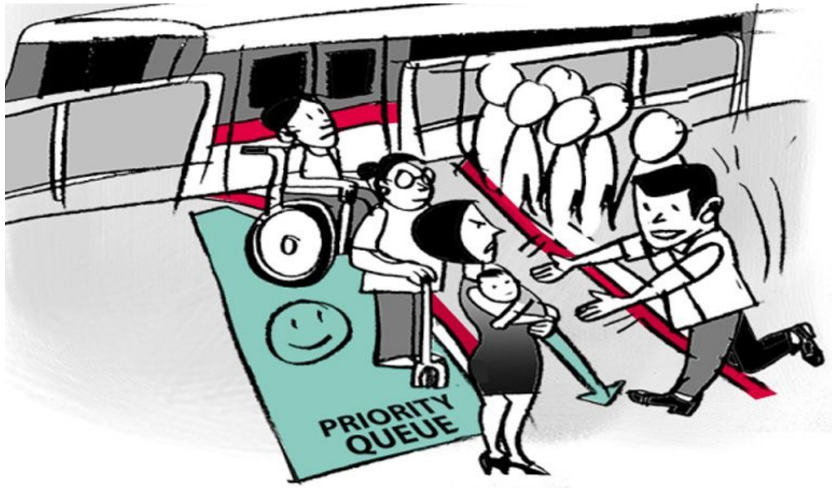
Natural Merge Sort

ADT: Queue & Priority Queue

Remember queue?

enqueue, dequeue?

What is a PQ?
practical examples?



'Can I borrow your baby?...'

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***: inserts (element, weight) into `PQ`: **`enPQ(PQ, item)`**
- ***dequeue***: removes & returns the heaviest element of `PQ`. Normally named as
  - `dePQ(PQ)`, for general case, or
  - **`deleteMax(PQ)`**, if *higher priority* means *bigger*, or
  - **`deleteMin(PQ)`**, if *higher priority* means *smaller*
- ***create***: creates an empty `PQ`: **`makePQ()`**
- *check for being empty*: **`isEmptyPQ(PQ)`**
- *changeWeight*: change the weight of a particular element of a queue
- *peek/frontier*: returns the heaviest element without removing it

possible (but bad) concrete data structures for PQ

DS	complexity of construction a PQ of n elements	complexity of dePQ	complexity of peek
unsorted arrays or linked list			
sorted arrays or linked lists			

Example: priority= max

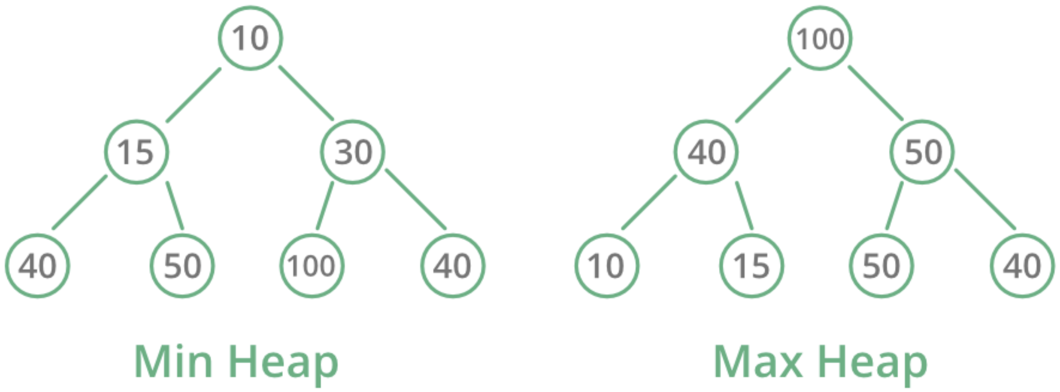
Unsorted: 9 2 7 5 6 8 3

Sorted : 2 3 5 6 7 8 9

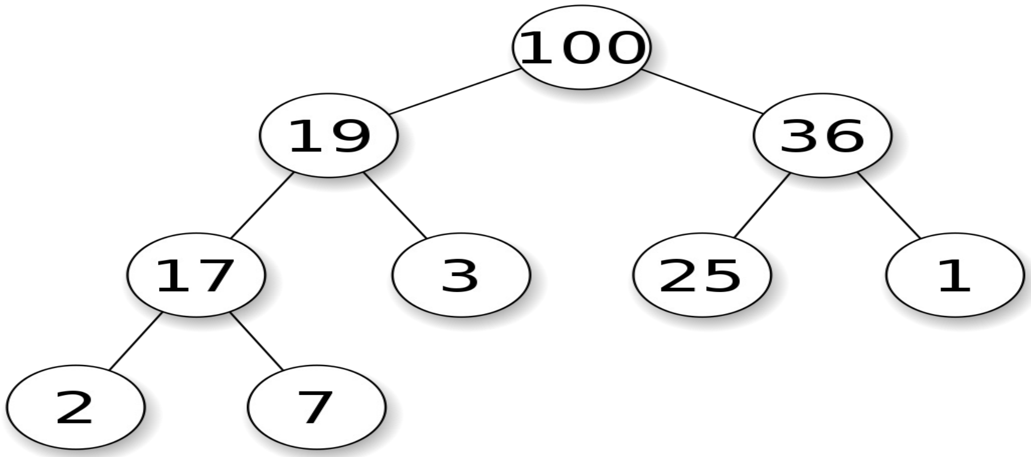
Binary Heap

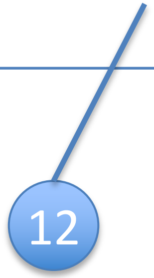
Binary heap: an implementation for priority queue

For simple keys, we can have min-heap or max-heap



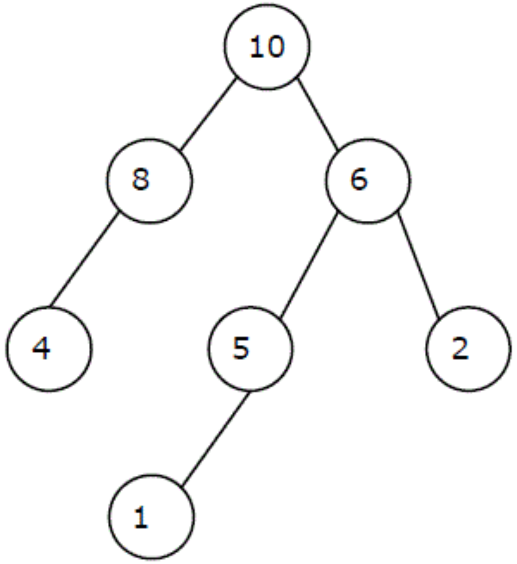
Heap: requirements

Example	**Conditions**
	1. The tree is complete. 2. *The heap property*: each node has a higher priority (here, is larger) than any of its descendants (or equivalently, just its children).

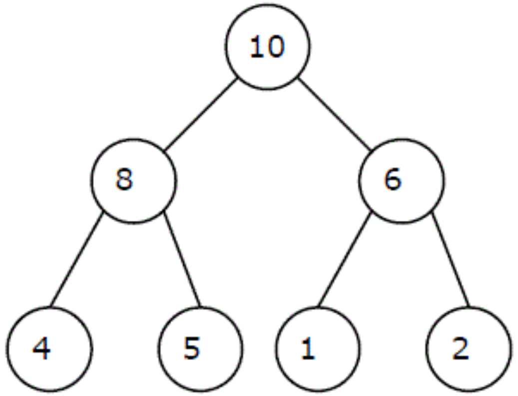


which one is a binary heap?

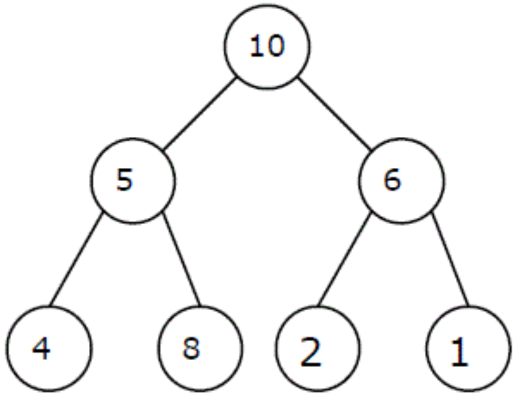
(A)



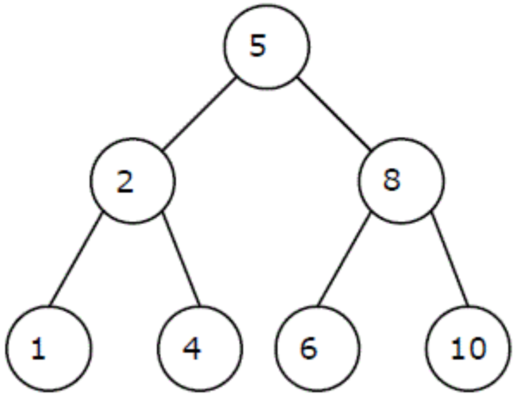
(B)



(C)

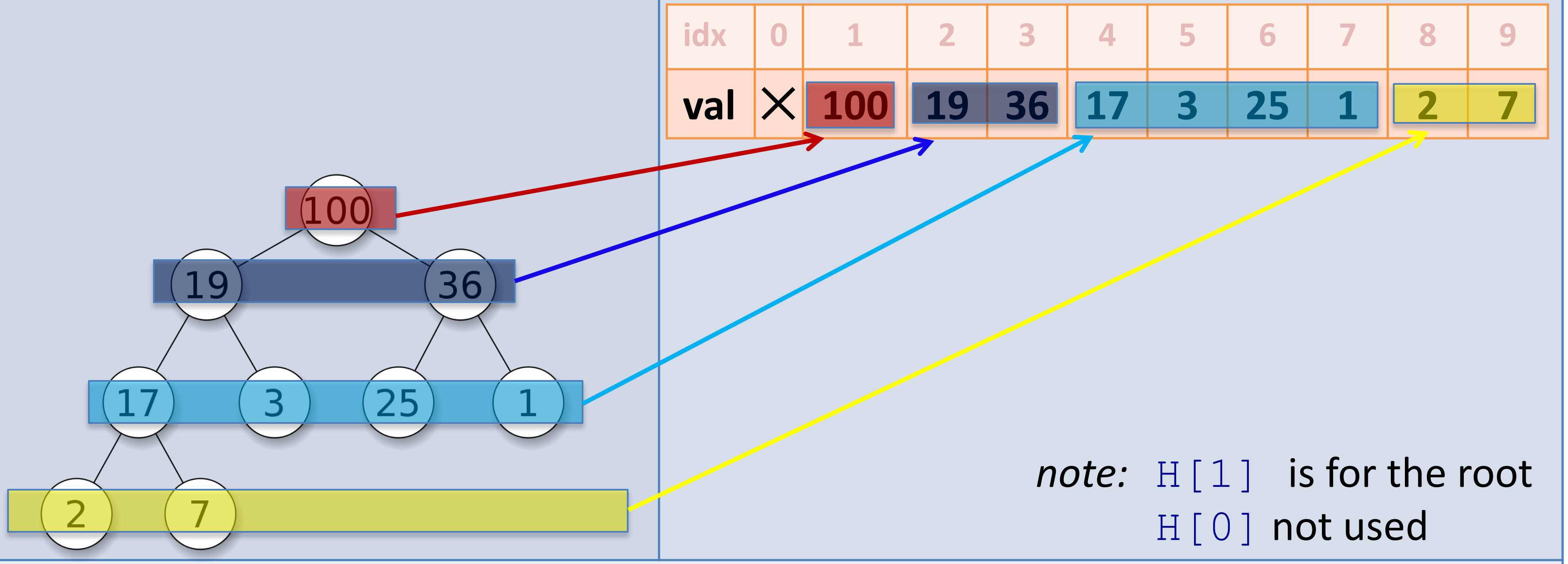


(D)



Binary Heap is implemented as an array!

Visualisation: as a complete binary tree	Implementation: using arrays



Heap is `H[1..n]`

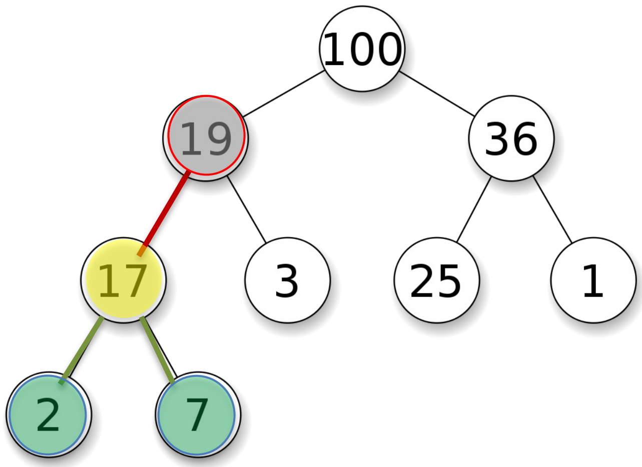
- level `i` occupies 2^i cells in array `H[1..n]` from idex 2^i to $2^{i+1}-1$

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]`

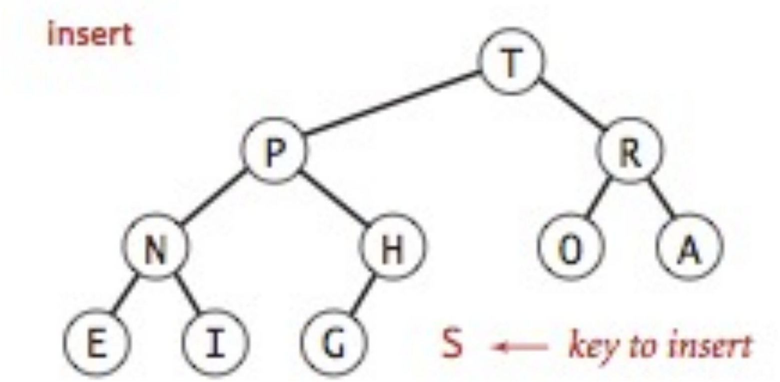
parent of `H[i]` is `H[i/2]`

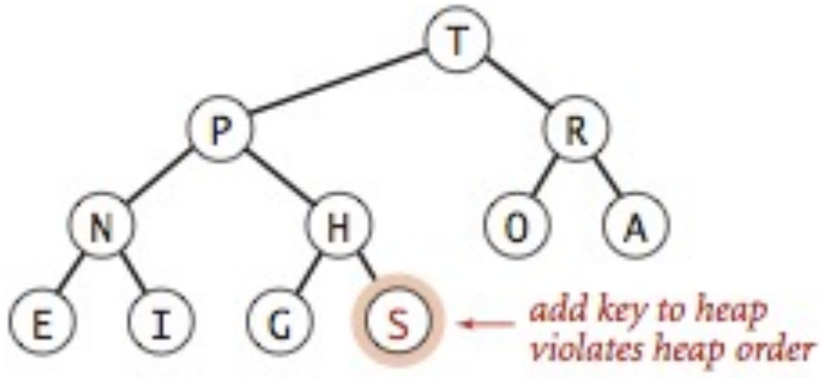
			4/2		i=4				4*2	4*2+1
idx	0	1	2	3	4	5	6	7	8	9
val	X	100	19	36	17	3	25	1	2	7



Insert into a heap.

tree visualisation





in the implemented array

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

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

`H` has 10 elements

Insert S

Just added `H[11]` = `S`

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

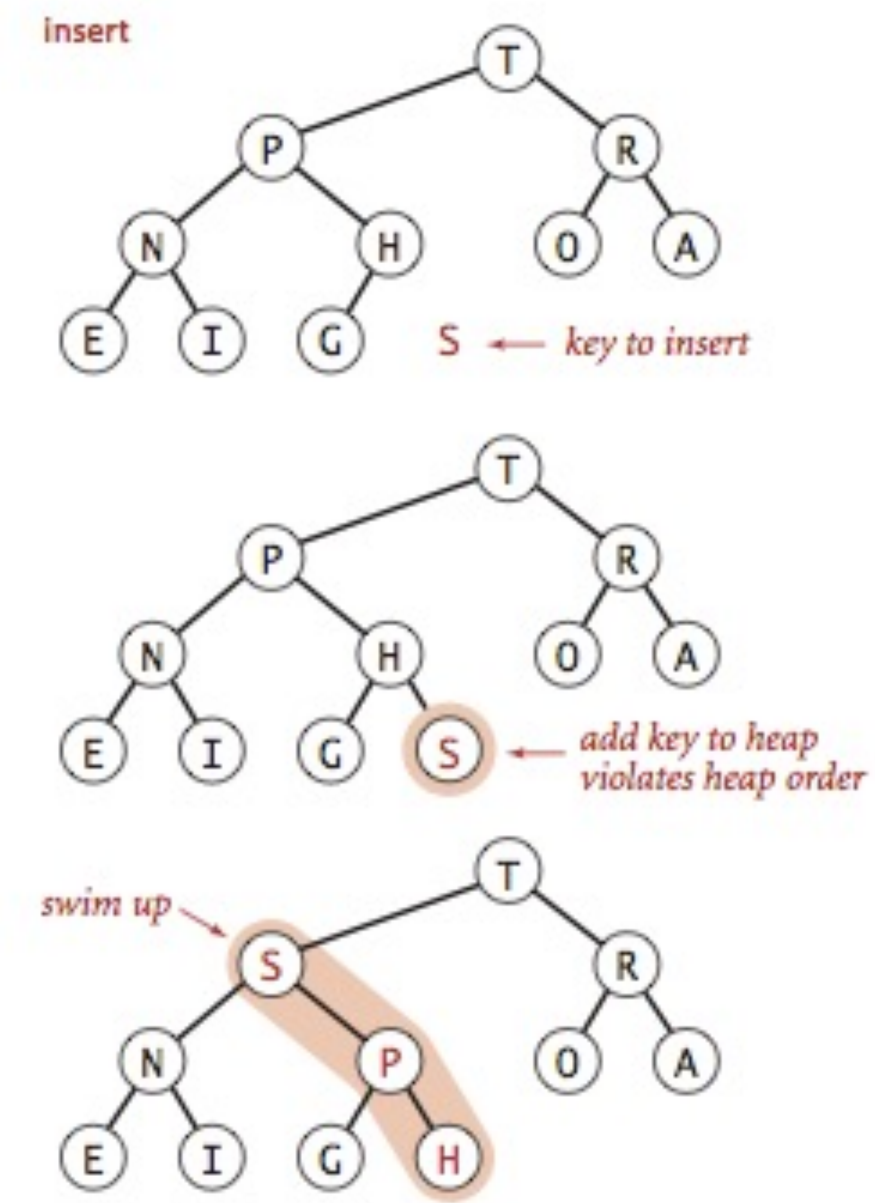
index 5=11/2 11

`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

Insert a new elem into a heap (enPQ). 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(?) , θ( )*

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

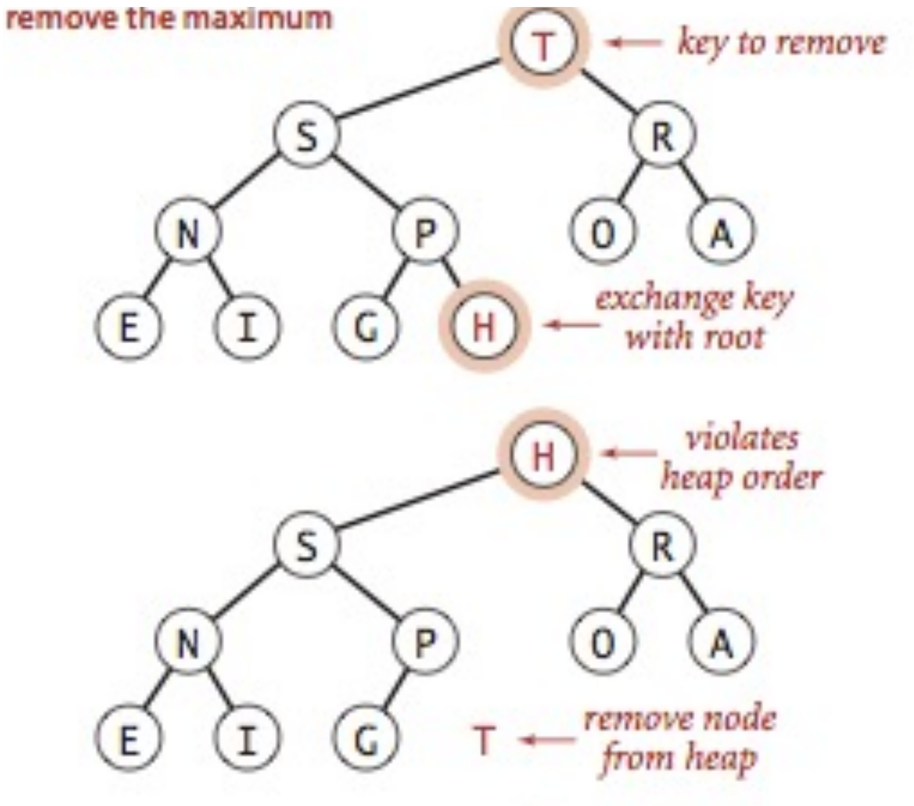
`[T,P,R,N,H,O,A,E,I,C,S]`

S   H

S   P

S

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

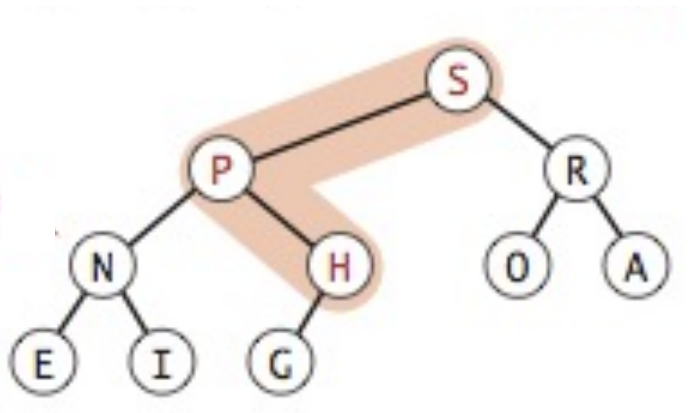


`Heap= [T,S,R,N,P,O,A,E,I,G,H]`

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)` for deletion. But 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.

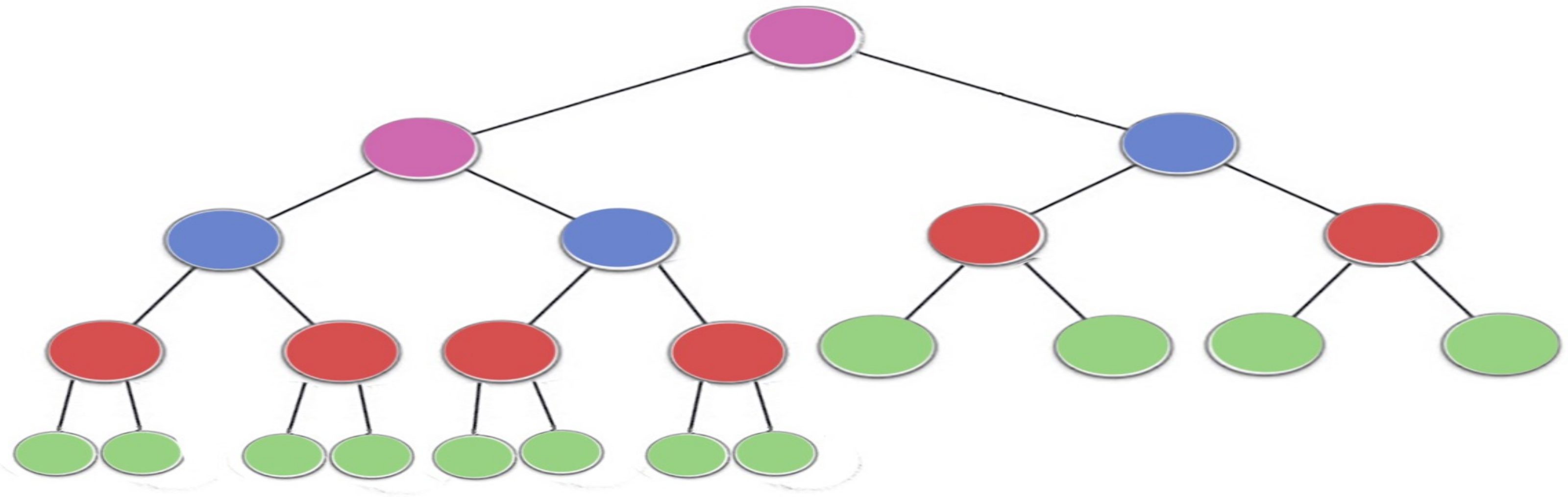
  Complexity: O(?)

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=n/2; i>0; i--) {
    // for i from last parent to first parent
   downheap(h, i);
```

= **Θ(n)** (see lectures and/or ask Google for a proof)
The operation is known as Heapify/Makeheap/ Bottom-Up Heap Construction



Heapsort= sorting using a heap

How? Complexity=?

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=
- = O(?)
- Questions:
  - What's the best case of heapsort?
  - Is heapsort stable?

Heap & Heap Sort: Complexity

Heap operations:

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

W9.2 (simplified)

Construct a max binary heap from the following keys:

8 7 16 8 10

a) Construct a max binary heap using the up-heap, inserting one number at a time.

b) Now construct a max binary heap from the same keys, using downheap (ie convert the original array into a heap).

c) What is the complexity of each method?

Peer Activity: when can we implement PQ using hash tables?

With some specific constraints, can a Priority Queue be efficiently implemented using a hash table:

A) Yes

B) No

W9.10.1

Consider the following **constraints:**

- keys
  - range is known ($0 \le \text{key} \le r$)
  - hashed with the function

    `hash(key) = key`
- hash table

Index	0	1	…	r - 1	r
Key	…		…		

…

W9.10.2

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	…		…		

…

W9.10.3: Finding the m-th smallest

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

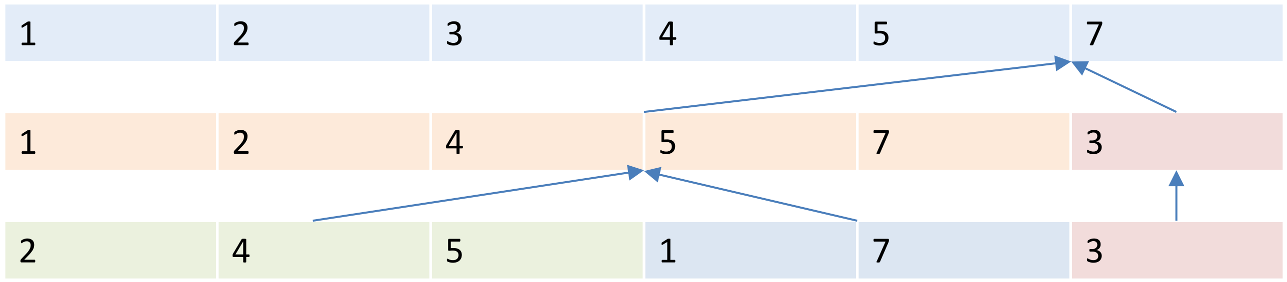
- Monotonic increasing runs already sorted
- Insert monotonic runs into queue instead of singletons

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

W9.3: Implementing Natural Merge Sort, given the implementation of:

- linked list of integers
- queue of linked lists