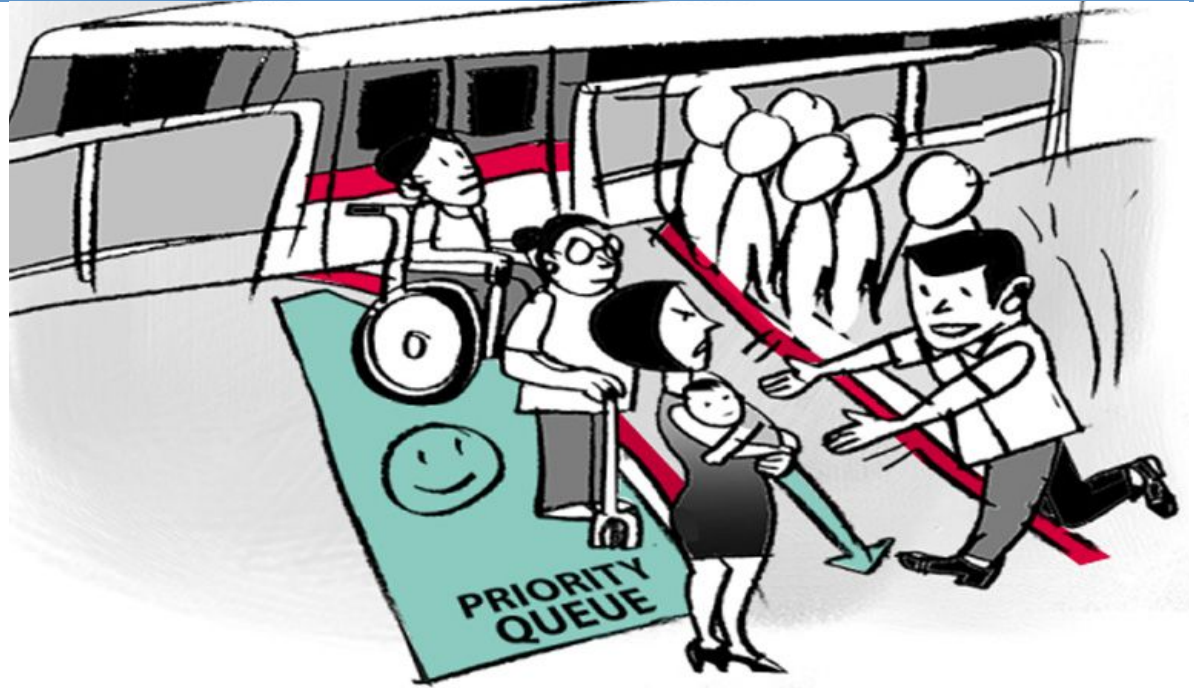


COMP20003 Workshop Week 9

- | | |
|----------|--|
| 1 | Reviewing MST: how to get 10/10 |
| 2 | Heaps & Binary Heaps |
| 3 | Heap Sort |
| 4 | Q 8.1 |
| 5 | Programming 8.1: Natural Merge Sort |
| 6 | Programming Extra: Binary Heap Implementation & Timing |

Yet Another ADT: Priority Queue

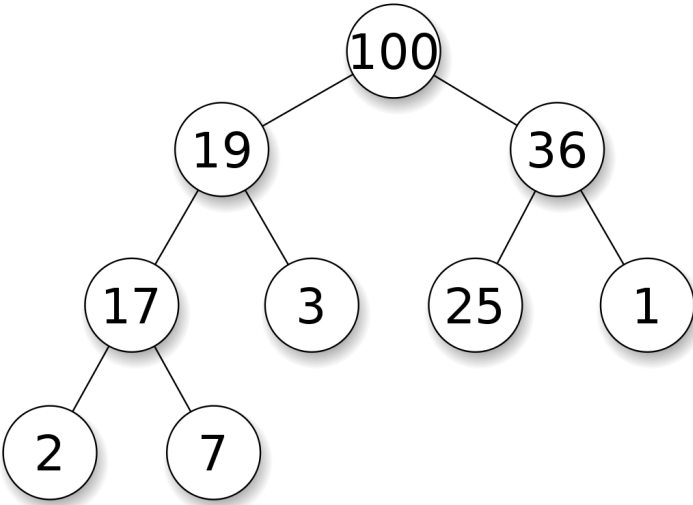


'Can I borrow your baby?...

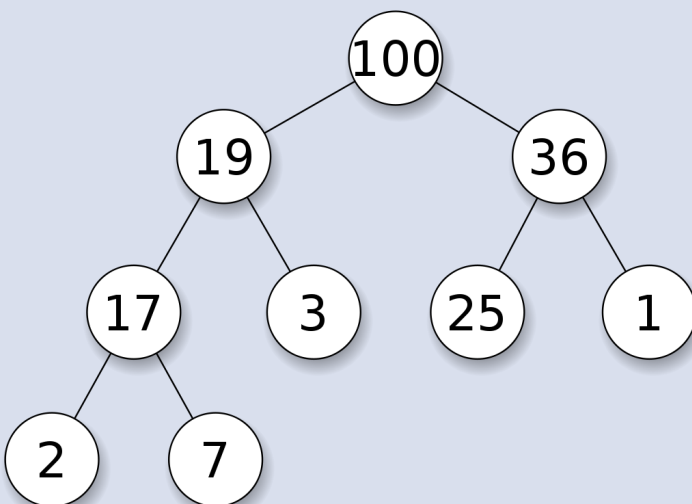
Main operations:

- creates an empty queue *‘Can I borrow your baby?...’*
- inserts an element into **Q** (**enPQ**)
- returns the highest/heaviest element, and removes it from **Q** (**dePQ**, or **deleteMax**, or **deleteMin**)
- **emptyPQ**
- **changeWeight**: change the weight of a particular element of a queue

Example of PQ: Binary Max Heap

Example	Conditions
 <pre>graph TD; 100((100)) --- 19((19)); 100 --- 36((36)); 19 --- 17((17)); 19 --- 3((3)); 17 --- 2((2)); 17 --- 7((7)); 36 --- 25((25)); 36 --- 1((1));</pre>	<ol style="list-style-type: none"><li data-bbox="1014 406 1661 464">1. The tree is complete.<li data-bbox="1014 556 1825 992">2. <i>The heap property</i>: each node has a higher priority (here, is larger) than any of its descendants (or equivalently, just its children).

Binary Heap is implemented as an array!

Visualisation	Implementation
	<pre>0 1 2 3 4 5 6 7 8 9 100 19 36 17 3 25 1 2 7 -----</pre> <p>heap h includes:</p> <ul style="list-style-type: none">- array $H[]$- n: number of elements in $H[]$
<ul style="list-style-type: none">- the tree is complete- a node A might have up to 1 parent- a node A might have up to 1 left child- a node A might have up to 1 right child	<p>Heap h is a pair $\{H[], n\}$, start from $H[1]$</p> <ul style="list-style-type: none">- there is no “hole” in array $H[]$- parent of $H[i]$ is $H[i/2]$ iif $i > 1$- left child of $H[i]$ is $H[2*i]$ iif $2*i < n$- right child of $H[i]$ is $H[2*i+1]$ iif $2*i+1 < n$




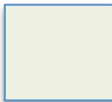
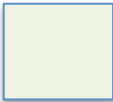
Insert a new elem into a heap

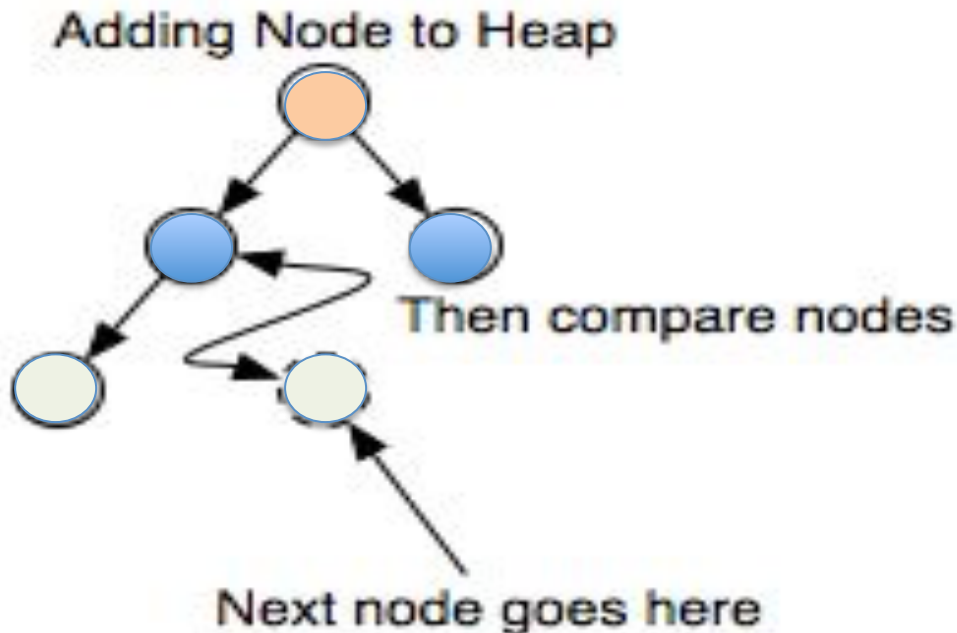
1. put new element to the very end of the heap's array (and hence change the number of elements in the heap), the new element will likely violate the heap condition, then
2. repair the heap using **upheap**.

upheap – basic operation for inserting into heaps

Problem: The last node of (e.g. just inserted into) a heap, and only it, might violate the heap property. Need to repair the heap.

Operation: `upheap(h, node)`

index=	1	2	3	4	5
					
	$(5/2)/2$	$5/2$			5



```
loop while(needed) {  
    swap(node, parent)  
}
```

```
needed =  
    having_parent  
    && parent < node
```

deletemax: delete (and returns) the highest-priority elem

The highest-priority elem is the root, ie. $H[1]$. To delete it:

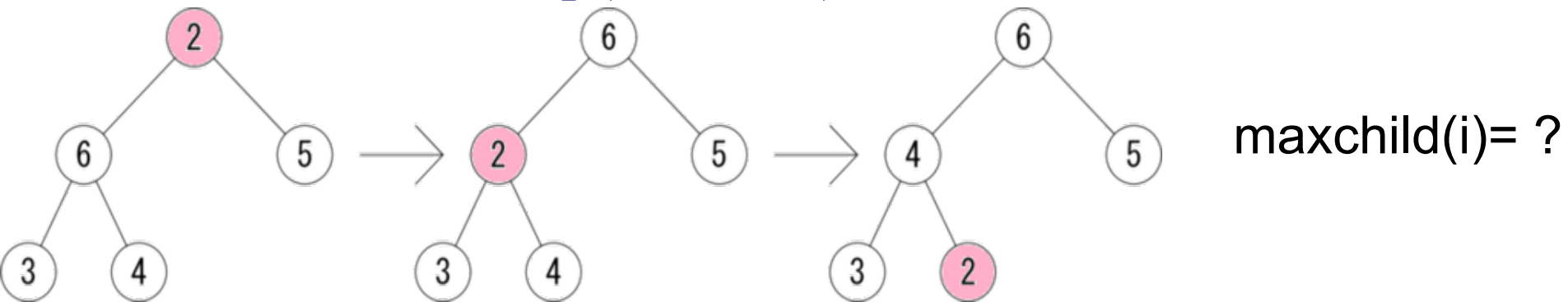
1. keep $tmp = H[1]$ for returning later,
2. replace $H[1]$ by the very last elem, and decrease the number of elements in heap n , then
3. since the new root might violate the heap condition, repair the heap using $downheap(h, 1)$, and finally
4. $return tmp$.

Note: Recall that heap h is a pair (H, n) where H is the heap's array, n is the current number of elements in heap.

downheap – basic operation deletion from heap

Problem: The root (and only the root) might violate the heap property. Need to repair the heap.

Operation: `downheap(h, node)`



```
loop while (needed) {  
    swap (node, maxchild)  
}
```



```
needed =    having_child  
         && node < maxchild
```


Heap Sort

Heap suggests a method for sorting:

- Construct a max heap of n elements.
- Swap root (max) with last element.
- Remove last element from further consideration, *i.e.* decrease size of heap by 1.
- Fix heap using....
... `downheap()`
- Repeat until finished.

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:

$$\log(1) + \log(2) + \dots + \log(n) = \Theta(n \log n)$$

How to efficiently build a heap with n elements?

- 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--) {  
    downheap(h, i);  
}
```

= $\Theta(n)$ (ask Google for a proof)

Heap & Heap Sort: Big-O Complexity

Heap operations:

- `upheap`:
- `downheap`:
- insert 1 element:
- `deleteMax`:
- turn an array to heap:
- `heapsort`:

Q 9.1

Construct a max binary heap from the following keys:

8 7 16 10 5 13 5 11 15 12 1 17

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).

What is the complexity of each method? Did the time it took you to do the exercise on paper correlate (roughly) with the theoretical complexity?

P 9.1: Programming Tasks

An *adaptive sorting algorithm* takes advantage of already existing order in a file. For example, insertion sort is an adaptive sort, since it exhibits linear behavior with sorted or nearly sorted files.

mergesort is not adaptive, but *natural mergesort* is.

In natural mergesort, existing “runs” in the data are used as the first units input into a bottom-up mergesort. For example, if the first ten items are already in order, and the next five items are already in order, then the first items to be dequeued and merged would be a “run” of size 10 and a “run” of size 5. No time would be wasted merging the first two elements, or the next two, etc. since they are already in order.

Write code for natural mergesort. Choose to:

- develop from last-week linked-list-based implementation, or.
- start a new implement a new array-based implementation.

Bottom-up adaptive merge sort with linked list

You can use last week's implementation of mergesort, available at:

github.com/anhvir/c203

where `queue.[c,h]` is a linked-list implementation of queue. Note that:

some changes have been inserted into `linked_list.c`.

Programming Extra: Binary Heap Implementation

- implement a simple version of minHeap
- apply for heap sort
- Timing: compare time of building heap using 2 methods