HEAPS

Today talk about
data structure based on
BINARY TREE

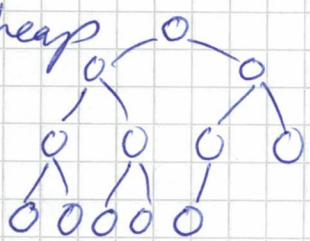
Kind of partially ordered
(namely: along any path in tree)

Used for (among other things):

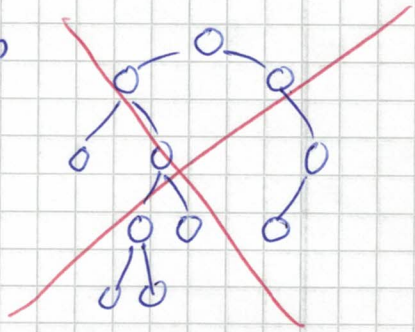
- SORTING (HEAPSORT)
- PRIORITY QUEUES
  (no longer first-in-first-out, but
  queue elements taken care of
  in order of priority

HEAP Nearly complete binary tree
Completely filled at all levels
except possibly lowest, which is filled
from the left

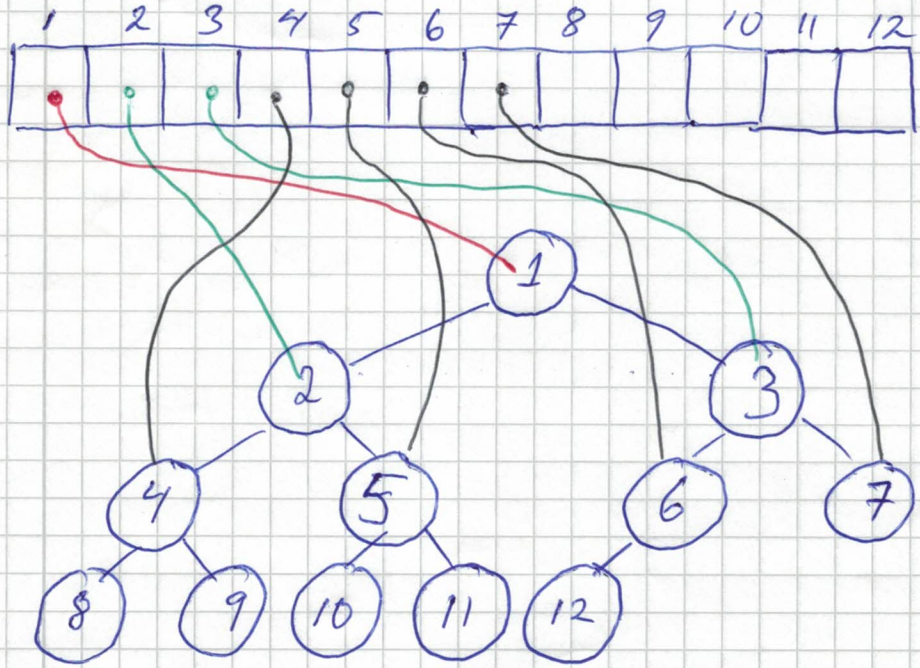
Ex heap



NOT heap



Represent the heap as an array A
(which we will assume is large enough)



These are INDICES for now — not elements stored in the array

PARENT (i) = $\lfloor i/2 \rfloor$

LEFT (i) = $2i$

RIGHT (i) = $2i + 1$

A[i] denotes heap element in position i

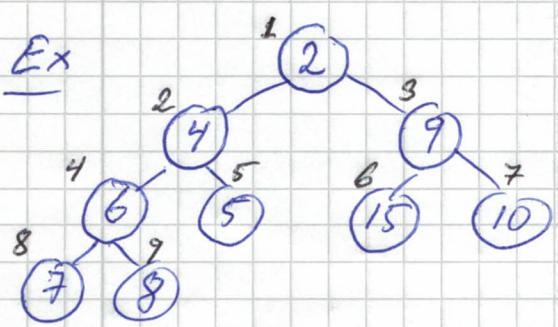
Two kinds of binary heaps:
- min-heap
- max-heap

We will focus on min-heaps.
Just change "$\leq$" to "$\geq$" everywhere and you get a max-heap instead.

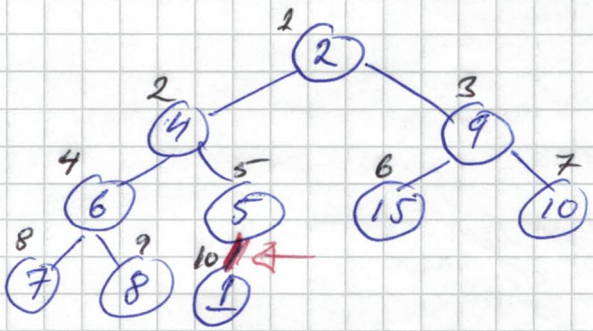
MIN-HEAP PROPERTY

For all $i > 1$ $\quad A[\text{PARENT}(i)] \leq A[i]$

Ex



Min heap



NOT min-heap

HEIGHT of heap = height of the tree

For n elements, height is $\Theta(\log n)$

OPERATIONS

MIN-HEAPIFY	Maintain/restore heap property locally
BUILD-MIN-HEAP	Build min-heap from unsorted array
HEAPSORT	Sort an array (in place, without extra storage)
MIN-HEAP-INSERT	Used for priority queues
HEAP-EXTRACT-MIN	
HEAP-DECREASE-KEY	

MIN-HEAPIFY (A, i)

PARENT(i) = $\lfloor i/2 \rfloor$
LEFT(i) = $2i$
RIGHT(i) = $2i+1$

H III

Precondition: Trees rooted at LEFT(i) and RIGHT(i) are min-heaps

Postcondition: Tree rooted at i is min-heap.

```
l := LEFT(i)
r := RIGHT(i)
if ( l ≤ A.heapsize and A[l] < A[i])
    smallest := l
else
    smallest := i
if ( r ≤ A.heapsize and A[r] < A[smallest])
    smallest := r
if ( smallest ≠ i)
    tmp := A[i]
    A[i] := A[smallest]
    A[smallest] := tmp
    MIN-HEAPIFY ( A, smallest)
```

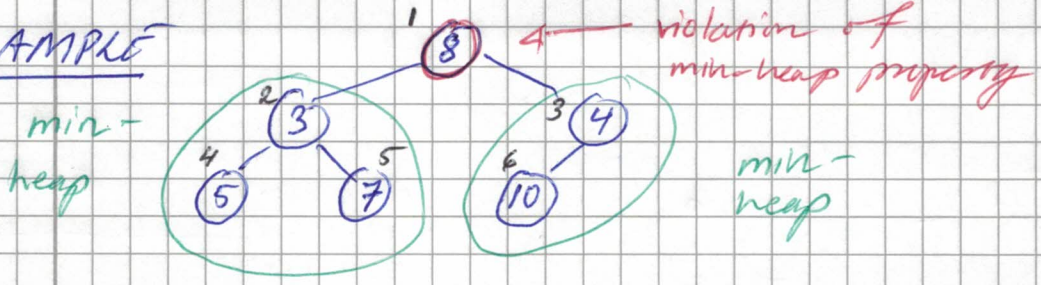
If A[i] is smaller than children, then min-heap property holds.

Otherwise, swapping A[i] with A[smallest] restores heap property locally for children of i.

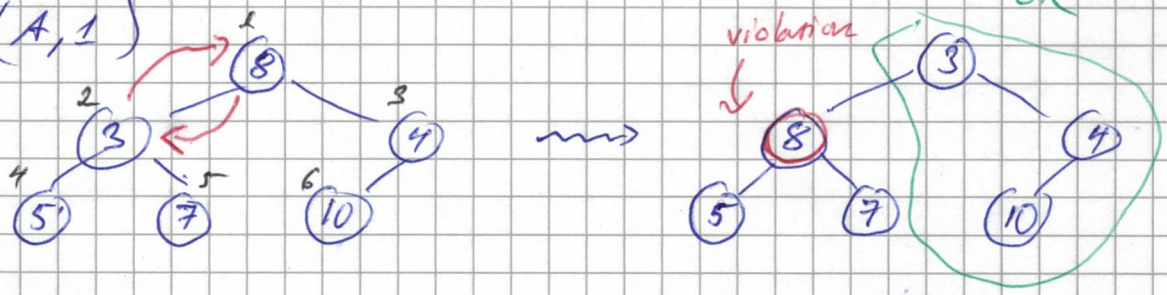
But new element in position A[smallest] can be too large, so restore property for that subtree. Note that other subtree is OK.

HW

EXAMPLE

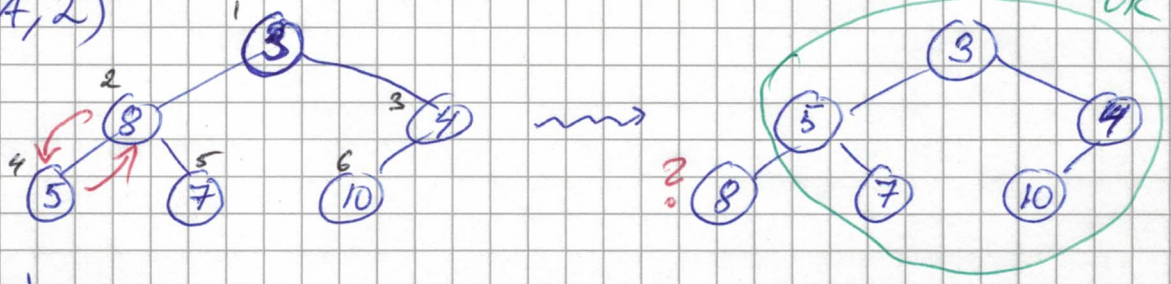


Heapify (A, 1)

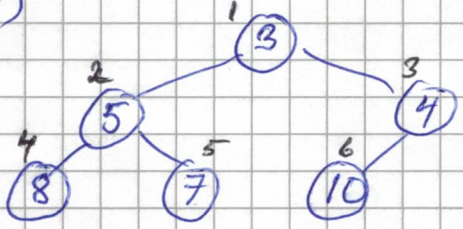


Heapify (A, 2)

Recursive call



Heapify (A, 4)



Nothing happens, since (8) is vacuously a min-heap

Proof of correctness: By induction over height of heap.

BASE MIN-HEAPIFY is correct on heaps of height 0 (size 1).

INDUCTION STEP Left and right subtrees are min-heaps by assumption. Suppose without loss of generality (w.l.o.g.) i swapped with LEFT(i) — if no swap, then clearly OK.
Right subtree was OK before — still OK
Right subtree root ~~smaller~~ larger than new root by construction.
After recursive call, left subtree is min-heap.
And left tree root has to be larger than root, because LEFT(i) was smallest in left subtree before swap
So by induction principle, MIN-HEAPIFY is correct ☐

Time complexity

In each call
- max 7 assignments
- max 5 comparisons

} $O(1)$

- max 1 recursive call on heap of height -1

Height $\Theta(\log n) \Rightarrow$ $O(\log n)$ recursive calls

$O(1)$ work in each call

Hence, MIN-HEAPIFY on a heap with n elements runs in time $O(\log n)$

BUILDING A HEAP

If A is heap of size n, then

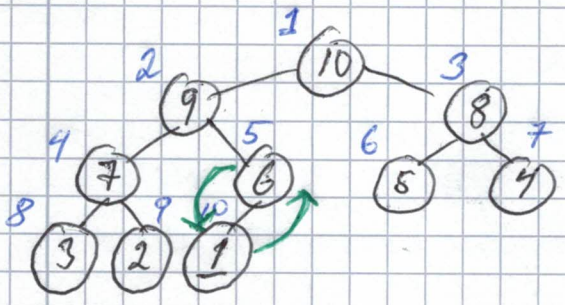
for $i \geq \lfloor n/2 \rfloor + 1$ $A[i]$ is a leaf (EXERCISE)
i.e., a min-heap of size 1 / height 0.
Can use MIN-HEAPIFY to build min-heap bottom-up, using that subheaps are already min-heaps.

BUILD-MIN-HEAP (A)

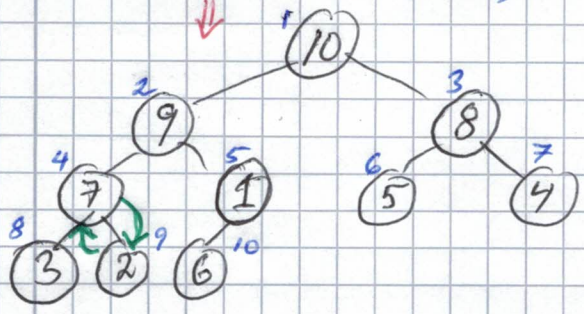
```
A.heapsize := A.length
for i := ⌊A.heapsize / 2⌋ downto 1
    MIN-HEAPIFY (A, i)
```

Ex $A =$

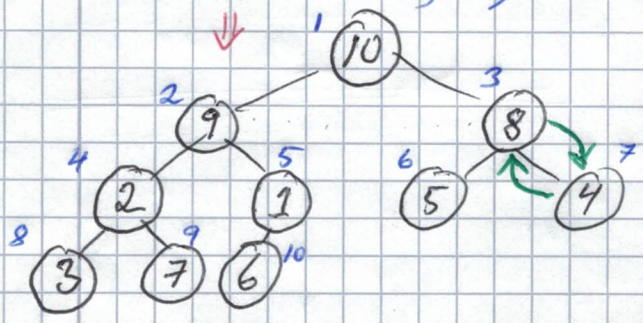
10	9	8	7	6	5	4	3	2	1



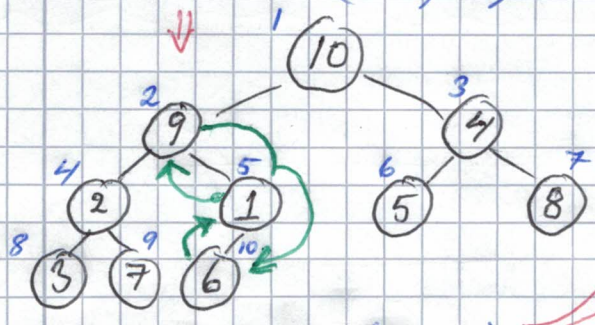
MIN-HEAPIFY (A, 5)



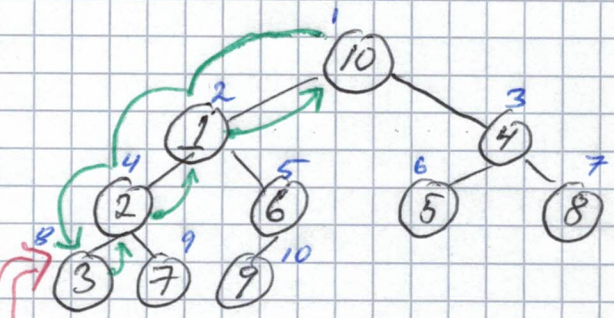
MIN-HEAPIFY (A, 4)



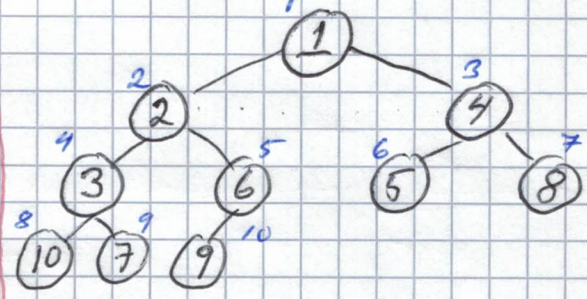
MIN-HEAPIFY (A, 3)



MIN-HEAPIFY (A, 2)



MIN-HEAPIFY (A, 1)



FINAL RESULT

Time complexity:

Suppose array contains n elements

$n/2 = O(n)$ calls to MIN-HEAPIFY

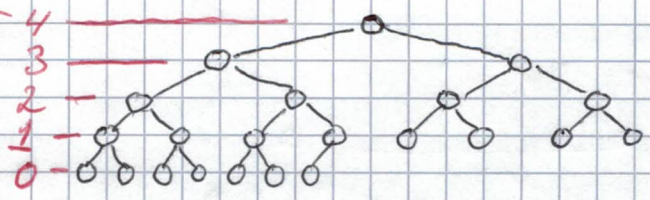
Each call takes time $O(\log n)$

So time complexity $O(n \log n)$

Actually, can do better!

Fact 1 n-element heap has height $\lfloor \log n \rfloor$

Fact 2 At most $\lceil n / 2^{h+1} \rceil$ nodes of height h in heap



[Exercises in CLRS]

MIN-HEAPIFY (A, i) takes time $O(\text{height of } i)$

So time complexity can be bounded by

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) =$$

$$O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) =$$

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = (*)$$

DETOUR (to Appendix A of CLRS)

GEOMETRIC SERIES $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$

(assuming $x \neq 1$)

Letting n go to infinity, get

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

if $|x| < 1$

Can differentiate this series termwise (not obvious, but true)

$$\sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{-1}{(1-x)^2} \cdot (-1) = \frac{1}{(1-x)^2}$$

Multiply by x to get useful summation formula

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \qquad (\dagger)$$

Plug in $x = 1/2$ in $(\dagger)$ to get $(*)$

$$O\left(n \cdot \frac{1/2}{(1-1/2)^2}\right) = O(n \cdot 2) = O(n)$$

So BUILD-MIN-HEAP actually runs in linear time!

Correctness proof

Loop invariant: At start of for loop with value i, every node $i+1, i+2, \ldots, n$ is root of min heap

True when entering loop: All $i+1, i+2, \ldots,$ leaves

At ith iteration: MIN-HEAPIFY makes this true for i also

At termination of loop, true for $i=0$ (i.e. for all nodes including 1st, so whole heap is min-heap

ALL WE HAVE SAID SO FAR ALSO WORKS FOR MAX-HEAPS

MAX-HEAP PROPERTY

For all $i > 1$ $\quad A[\text{PARENT}(i)] \geq A[i]$

Algorithms: Just change "$\leq$" to "$\geq$" everywhere

HEAPSORT

Can sort array in increasing order by

1. Building a max-heap
2. Repeat for $i = n$ down to 2
   - Swap element i with root, which is nth largest element.
   - Decrease heap size by 1
   - Run heapify on root (to restore heap property)

HEAPSORT (A)

```
BUILD-MAX-HEAP(A)                          O(n)
for i := A.length downto 2                 n times
    O(1) { tmp := A[i]
           A[i] := A[1]
           A[1] := tmp
           A.heapsize := A.heapsize - 1
    O(log n) MAX-HEAPIFY(A, 1)
```

Time complexity

$O(n \log n)$

Correctness

CLRS Exercise 6.4-2

Question 1 For BUILD-MAX-HEAP we also have n calls to MAX-HEAPIFY, but we argued all of these calls take linear time $O(n)$ together. Why can't we use the same argument here?

Question 2 What happens if you run heapsort on a sorted array, say

1	2	3	4	5	6	7	8	9	10

?

What is the running time?

Heapsort sorts IN PLACE, i.e., without needing extra storage for the sorting (which the standard implementation of merge sort does need)

PRIORITY QUEUES

This is why we started talking about heaps — we will use them for the priority queue in Dijkstra's algorithm and Prim's algorithm

A MIN-PRIORITY QUEUE Q supports the following operations

INSERT (x, k)	Insert element x with key k
MINIMUM	Return element with smallest key
EXTRACT-MIN	Remove and return minimum element
DECREASE-KEY(x,k)	Decrease key of x to k
EMPTY	Return true if queue empty

EMPTY

Actually, should check that Q is not empty; generate error if this is so

```
return (Q.heapsize == 0)
```

O(1) time

MINIMUM

```
return Q[1]
```

O(1) time

```
if (Q.EMPTY)
    error "Heap empty"
else
    return Q[1]
```

What should we do if we also want to remove minimum element Q[1] ?

- Heap size should shrink by 1
- Last leaf should disappear (to maintain heap structure)

Shift last leaf to root — now root is probably wrong, but rest of min-heap OK

Run MIN-HEAPIFY

EXTRACT-MIN

```
if Q.EMPTY
    error "Heap empty"
min := Q[1]
Q[1] := Q[Q.heapsize]
Q.heapsize := Q.heapsize - 1
MIN-HEAPIFY(Q, 1)
```

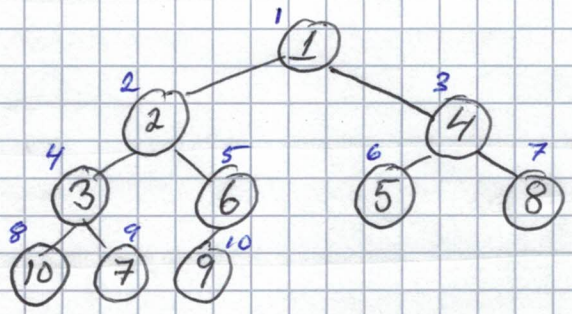
O(1) time

O(log n)

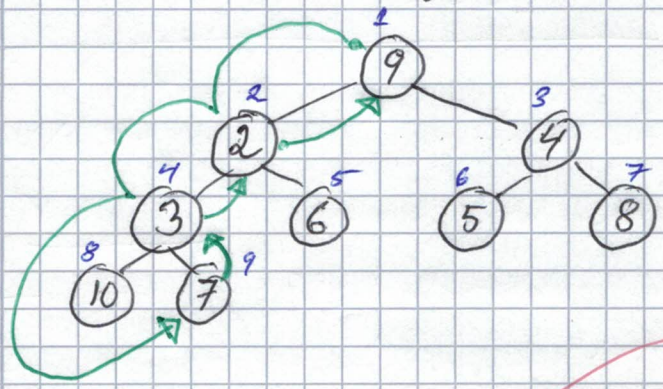
Time complexity: $O(\log n)$

Correctness: Exercise (analogous to what we have done before)

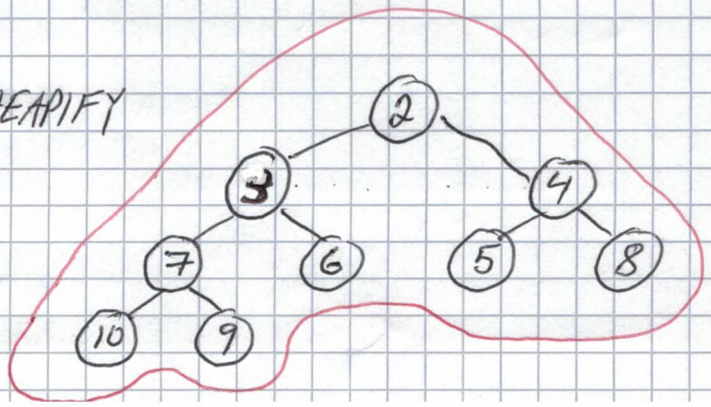
Ex Look at our heap from before



After dequeueing min element



After MIN-HEAPIFY

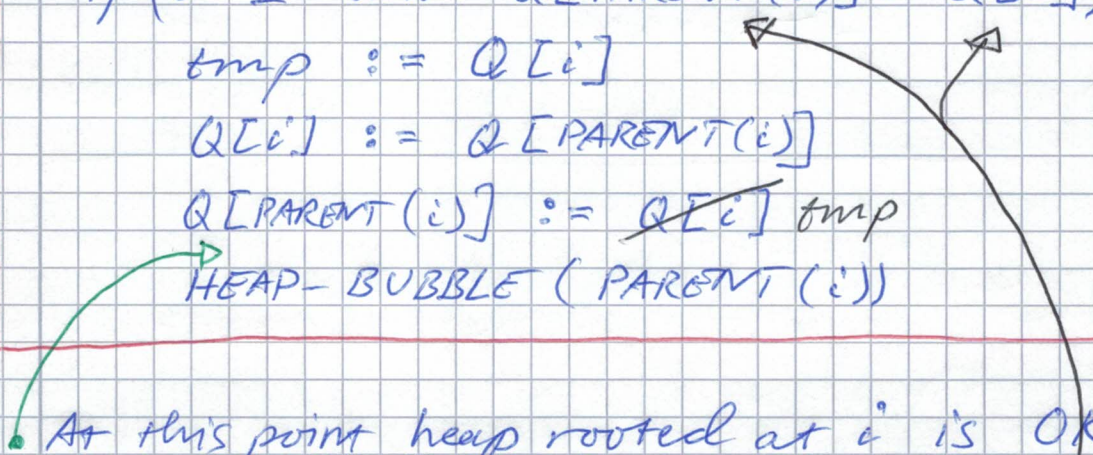


When an element key decreases, might need to shift element up rather than down

$$PARENT(i) = \lfloor i/2 \rfloor$$

HEAP-BUBBLE (i)

```
if (i > 1 and Q[PARENT(i)] > Q[i])
    tmp := Q[i]
    Q[i] := Q[PARENT(i)]
    Q[PARENT(i)] := Q[i] tmp
    HEAP-BUBBLE ( PARENT (i))
```



- At this point heap rooted at i is OK, because PARENT(i) must be smaller than LEFT(i) and RIGHT(i). Heap rooted at sibling of i also OK, since parent decreased
- Note that we should actually compare the keys of the elements here, so the comparison should be something like

```
if (i>1 and Q[PARENT(i)].key > Q[i].key)
```

but the pseudo-code in CLRS is a bit relaxed here.

For Dijkstra's algorithm:
- the ELEMENTS are the vertices
- the KEYS are the distance estimates

For Prim: KEY for v is cheapest edge to v

Time complexity of HEAP-BUBBLE

O(1) per recursive call
At most log n recursive calls
} O(log n)

HEAP-DECREASE-KEY (x, k)

```
O(1)   i := index of x in heap   [assume stored in x]
                                 [if x not in heap,  ]
                                 [then generate error]
       if (k > x.key)
          error "New key larger than old key"
       x.key := k
O(log n)  HEAP-BUBBLE (i)
```

Time complexity: $O(\log n)$

Correctness: As for HEAP-BUBBLE

INSERT (x, k)

```
O(1)   Q.heapsize := Q.heapsize + 1
       x.key := k
       Q[Q.heapsize] := x
O(log n)  HEAP-BUBBLE (Q, Q.heapsize)
```

Time complexity: $O(\log n)$

Correctness: As for HEAP-DECREASE-KEY

Pretend first that $x.key = \infty$

Then x inserted in admissible location.

Then key decreased to actual value

Note We are using the recursive method HEAP-BUBBLE here just to practice recursive algorithms and analysis of such algorithms. To get more efficient code in practice, use iterative methods as in CLRS.