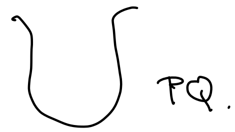
Priority Queues

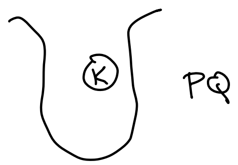
A **Priority Queue** is an ADT, like a regular queue but each element has a priority associated with it. Typically the PQ will remove items with the highest(or lowest) priority first.

Assume min-PQ. The **Operations** are:

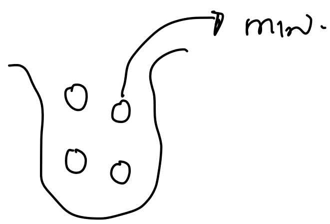
MAKENULL(PQ)



INSERT(k,PQ)

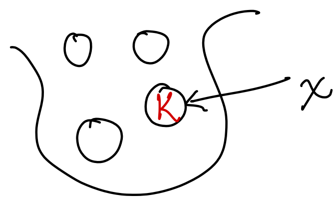


DELETEMIN(PQ)

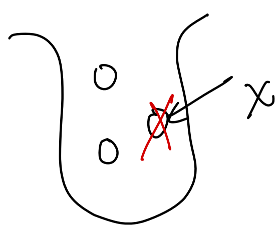


DECREASEKEY(x,k,PQ)

location new key.



DELETE(x,PQ)



The Priority Queue can be implemented with a variety of implementations:

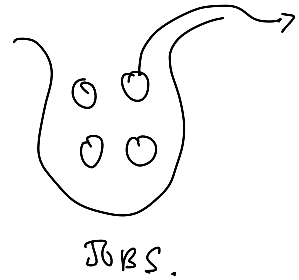
The obvious choice is the heap...

	Insert	Deletemin.
• Sorted Linked List	n	1
• Unsorted L.L.	1	n
• Balanced Search tree	$\log_2 n$	$\log_2 n$
• Tournament Tree	"	"
• BEAP	$\sqrt{n}$	$\sqrt{n}$
• Fibonacci Heap	1^*	$\log_2 n^*$

*: Amortized (discussed).

Applications:

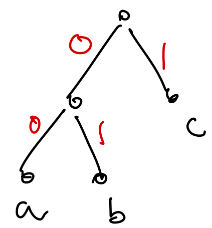
1. Operation Systems: scheduling of jobs



Remove JOB with highest priority.

2. Construction of huffman tree

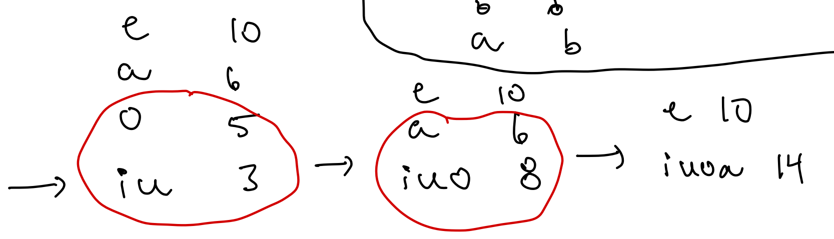
ex.

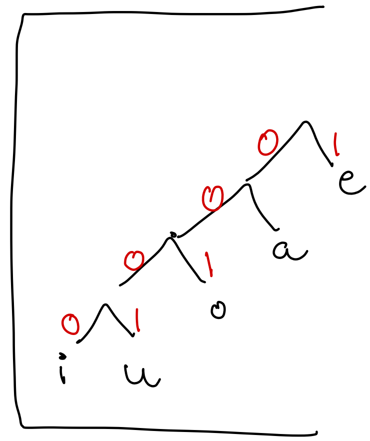


a: 00
b: 01
c: 1

PQ.

Sym.	freq.
e	10
a	6
o	5
i	2
u	1





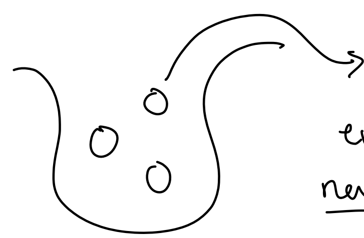
* Store frequencies in PQ

* Remove 2 mins, sum freq. to make new node...

Code:
e: 1
u: 0001

3. Discrete event simulation

PQ: future events.
Keys: time.

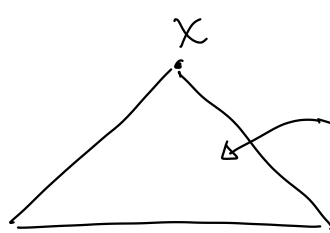


Remove the event that has the next time an event should be executed.

4. Sorting: heapsort

BINARY HEAP: (min PQ)

- Complete Binary Tree
- Each node has the heap property



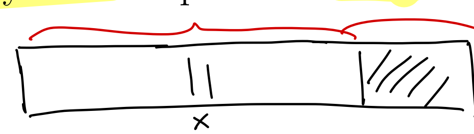
Key[x] < Key[y]

∀ y in subtree

- Height: Recall... $\lfloor \log_2 n \rfloor$

The implementation could be array-based or pointer-based. Assume for the following operations it is array-based.

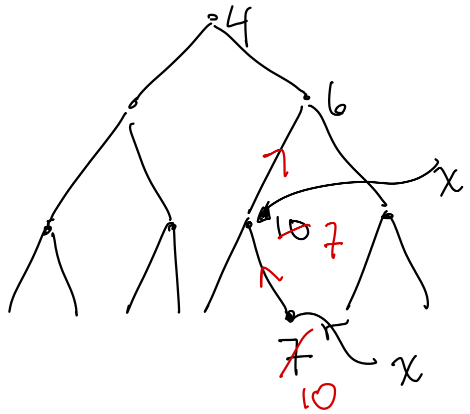
H:



Heapsize(H).

Operations:

1. Siftup(x): Moves element at position x up the heap until it respects the heap property.



While $x > 1$ and $\underbrace{H[parent[x]] > H[x]}_{\text{move up if parent is bigger.}}$

- swap Keys of parent[x] and x
- x = parent[x].

2. Heapify(x) Element at position x moves down the ~~array~~ tree until the heap~~ify~~ property is maintained.

while $x <$ Heapsize(H) do:

$y = \text{argmin}(H[x], H(left[x]), H(right[x]))$

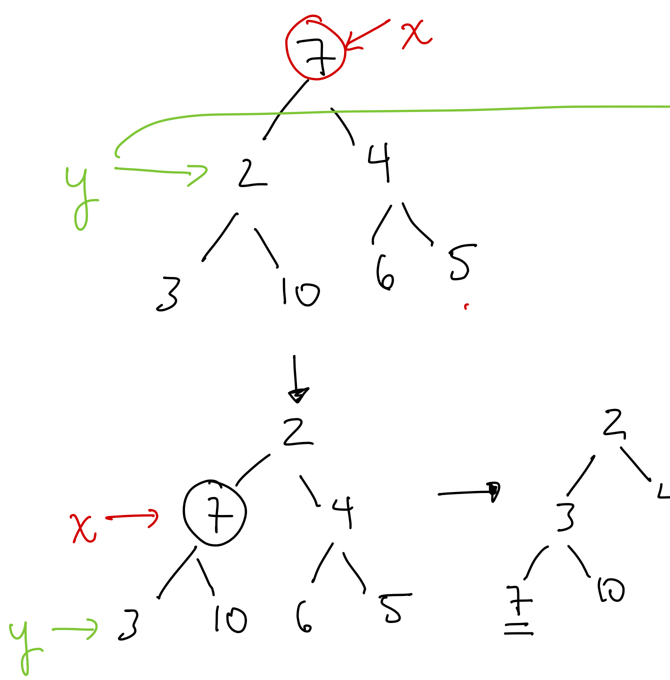
if $y = x$ halt

else

swap Keys y, x

$x = y$.

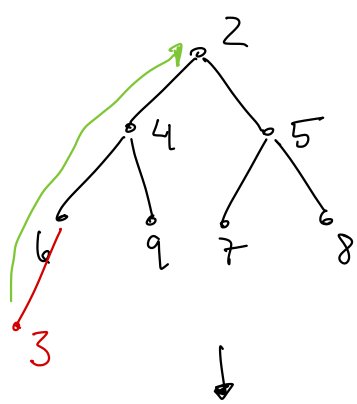
NOTE that the argmin op takes 2 comparisons!



3. Insert(k,H):

2	4	5	6	9	7	8	3	

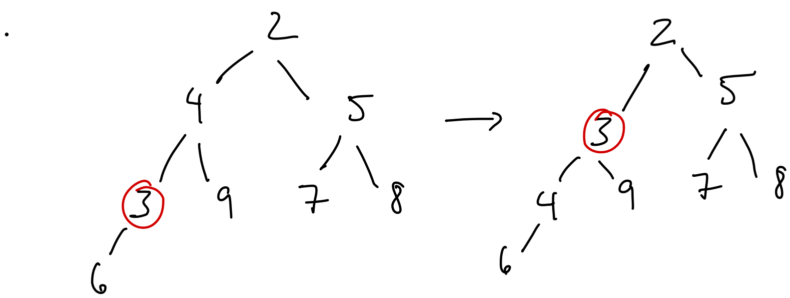
insert Key K = 3.



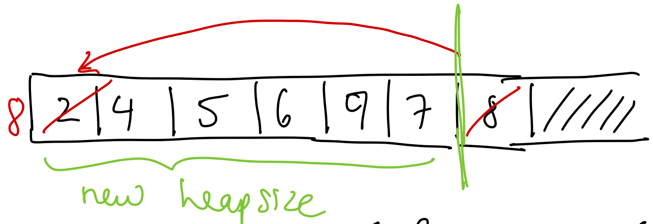
Heapsize(H) += 1

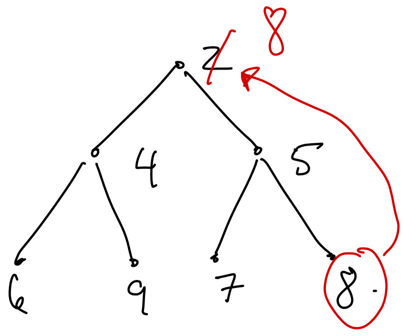
Key[Heapsize(H)] = K

Siftup(Heapsize(H))



4. Deletemin(H)





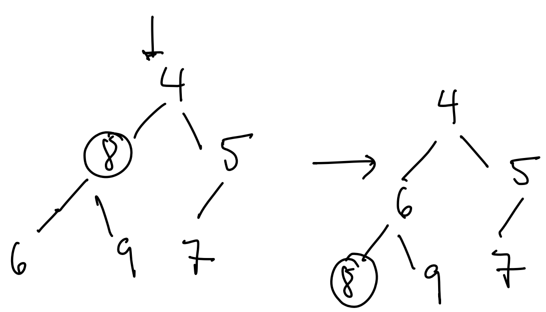
alg:

Return Key[1]

Key[1] = Key[Heapsize(H)]

Heap Size (H) -= 1

Heapify(1).

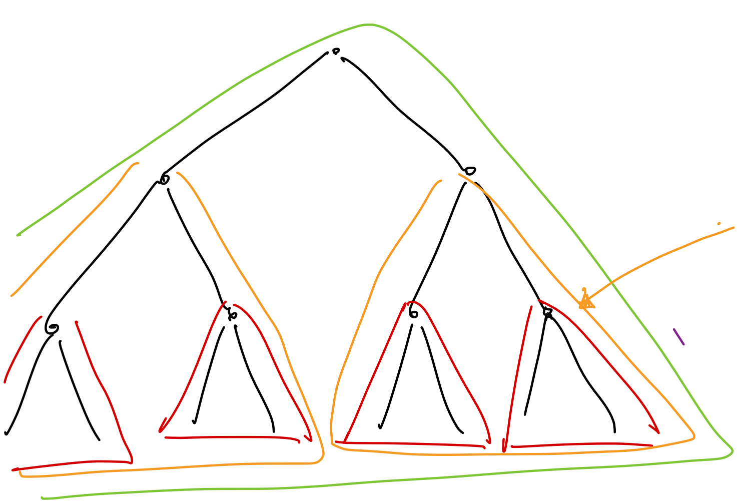


Compleixty: # comparisons ...

- Insert: $\leq \lfloor \log_2 n \rfloor$ } <u>one</u> comp. for each tree depth.
- Deltemin: $\leq 2 \lfloor \log_2 n \rfloor$. ← old size.

uses heapify, which needs 2 comp. as it calls argmin.

Buildheap(H) Given an array H of keys, turn H into a heap.



Naive # comparisons = $\underbrace{O(n\log n)}_{\text{heapify each node.}}$

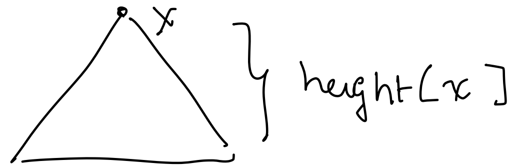
cost for heapify(x) $\leq$ 2 height(x).

<u>How?</u>

Call heapify on subtrees of height 1, (red) subtrees of height 2(yellow), etc...

Cost for heapify[x]:

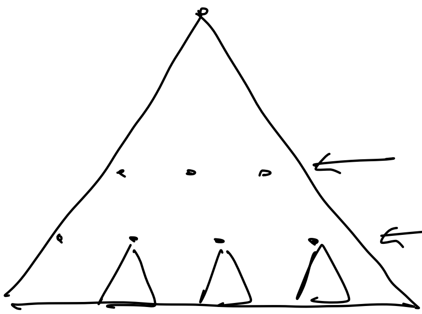
$$2\,\text{height}[x].$$



Overall cost: $\displaystyle\sum_{i=1}^{h} 2i \cdot (\#\text{ nodes at height } i)$ ← height of tree at x.

$$= \sum_{i=1}^{h} 2i\,(2^{h-i})$$

Time to heapify.



height(x) = 2, # nodes = 2^{h-2}.

height(x) = 1, # nodes = 2^{h-1}

$$= 2^{h+1}\sum_{i=1}^{h} \frac{i}{2^i}$$

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

diff.

$$\sum_{1} i x^{i-1} = \frac{1}{(1-x)^2}$$

$$\sum_{1} i x^{i} = \frac{x}{(1-x)^2}$$

$$x = 1/2.$$

$$\le 2^{h+1} \underbrace{\sum_{i=1}^{\infty} i/2^i}_{\le 2}$$

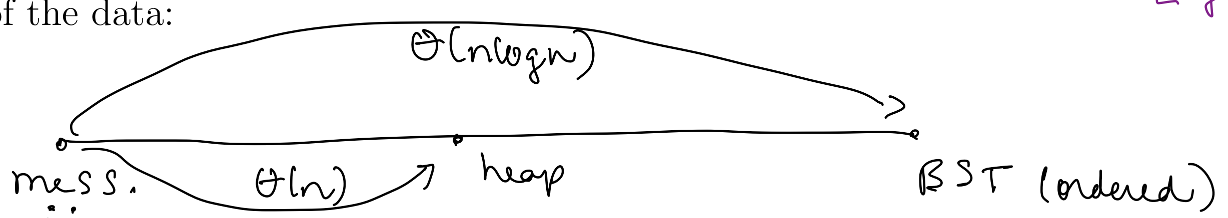
$$= 2^{h+1}\cdot 2$$

$$= 4\cdot 2^h \le 4n \in \Theta(n).$$

Recall $h \le \lfloor \log_2 n \rfloor$

Note: $\Theta(n)$ is much better than

Entropy of the data:



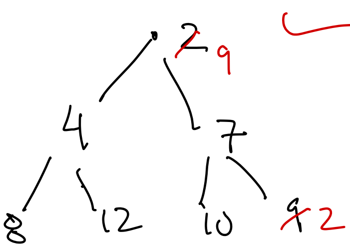
HeapSort: Given an array H of unsorted elements (n), sort them *in place*:

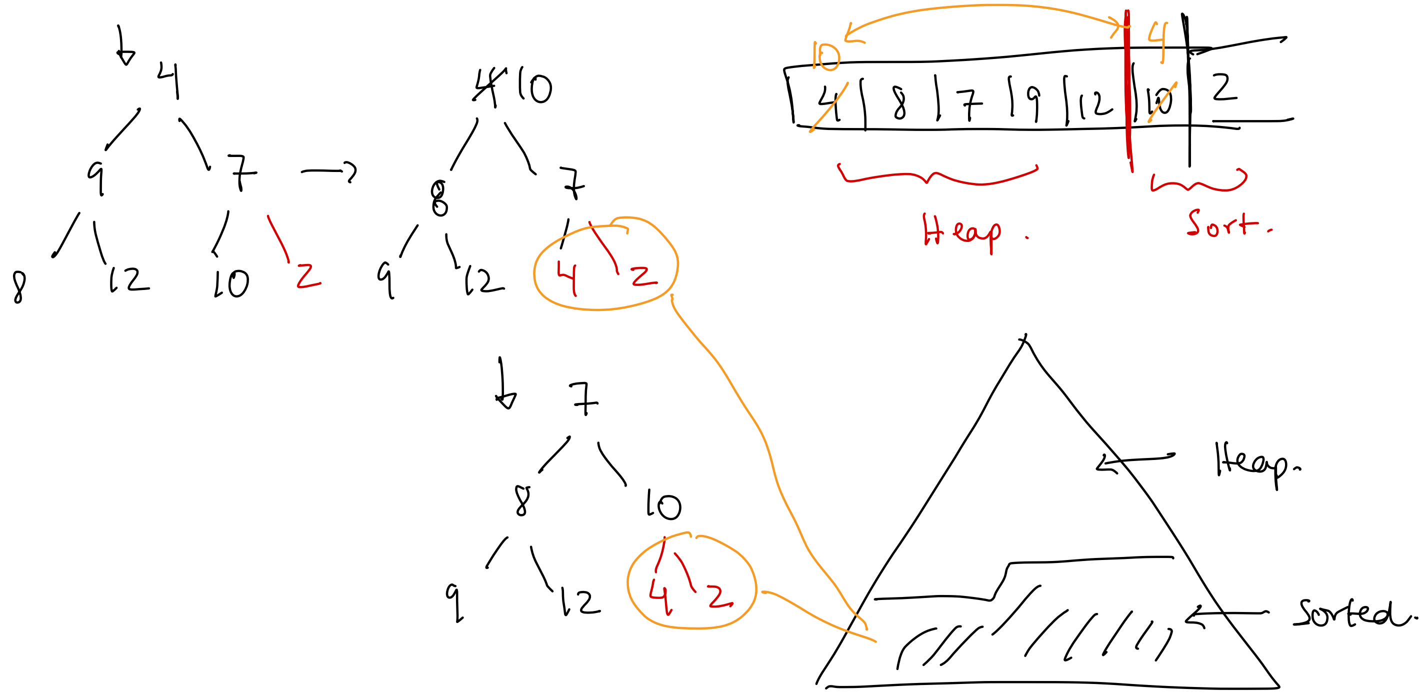
Ex.

9 ~~2~~	4	7	8	12	10	2 ~~9~~	////

heap | Sorted.

Switch min with last element.





Algorithm: Build heap (H)

Heapsize (H) = n.

For x = n downto 2 do

swap Key[x] and Key[1]

Heapsize (H) -= 1

Heapify(1)

* Note that the "Sorted" elements are still in the array (back) but are no longer part of heap.

Number of comparisons:

$$\underbrace{\text{Build heap cost}}_{4n.} + \sum \text{heapify cost}.$$

$$4n. + 2\cdot\sum_{i=1}^{n-1} \log_2 i$$

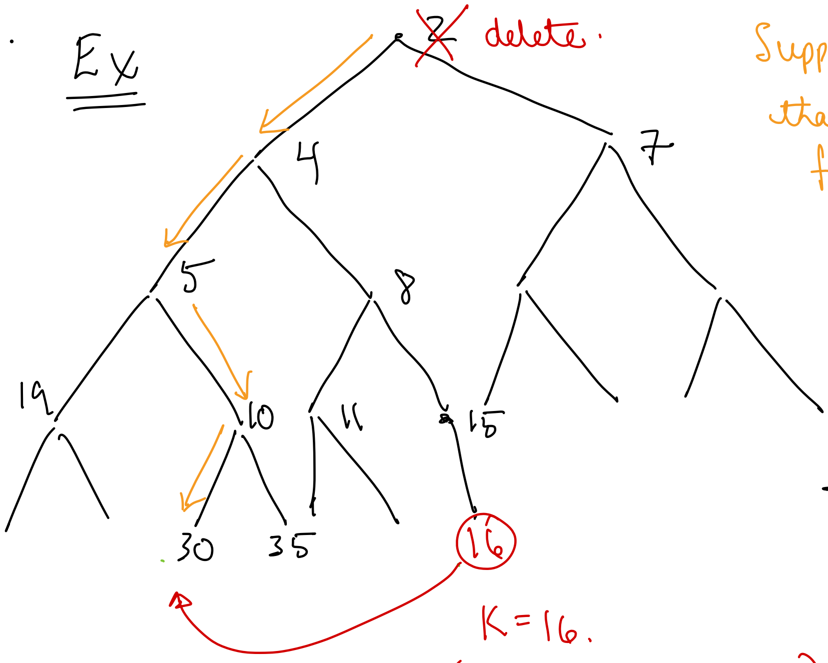
Cost of heapify is $\leq 2\log_2 i$ for a heap of size i

$$\leq \Theta(n) + 2 n \log_2 n.$$

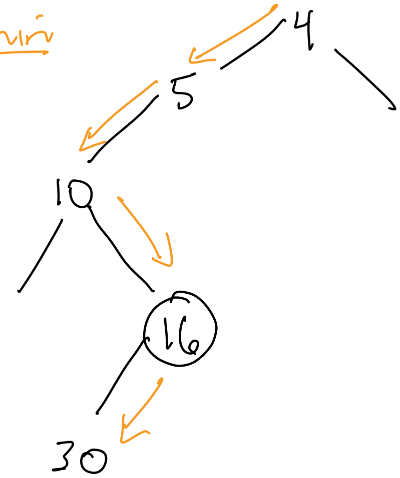
Bottom-up Heapsort: *Reduces the number of comparisons.

- Builds heap as before (Small subtrees → bigger subtrees)
- The delete min operation is performed *bottom-up*, which means there is *no call to Heapify*.

Ex



Suppose we found the yellow path that is the path that follows the min children.



instead of placing it at the root and calling heapify.

Above, we can see that the new delete-min will insert the last key k into the heap as follows:

- Step 1) Find the path from the root following *smallest* children $\lfloor \log_2 n \rfloor$
- Step 2) Search up from the bottom leaf of this path until you find the position to insert the key, k.

$\leq \lfloor \log_2 n \rfloor$.

Cost:

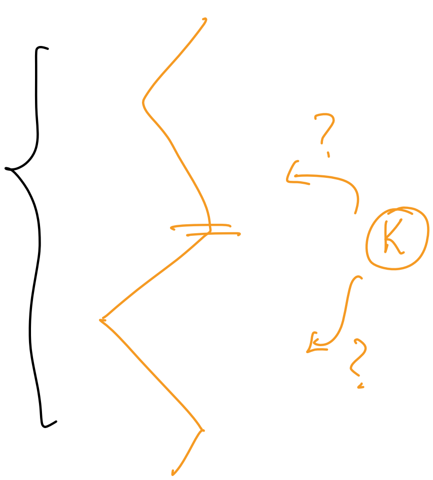
Total: $\leq 2 \lfloor \log_2 n \rfloor$.

We can do better!!
Simply find a faster way to insert K on the 'yellow' path.

Faster Variation of delete-min:

Notice that the path of smallest children is *sorted.*
We can find the insert position of key k using **Binary Search**!

height of path $\leq \lfloor \log_2 n \rfloor$



Binary Search time!

$\leq \log_2 (\underbrace{\lfloor \log_2 n \rfloor}_{\text{size}})$

Cost of delete min with this fast variation:

$$\underbrace{\lfloor \log_2 n \rfloor}_{\text{Finding the path of min. children.}} + \underbrace{\log_2(\log_2 n)}_{\text{Binary Search.}}$$

Comparisions in Bottom-up Heapsort:

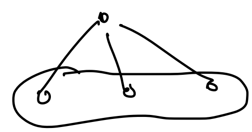
Buildheap + delete mins.

$$\le \Theta(n) + n(\log_2 n + \log_2(\log_2 n))$$

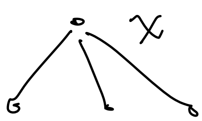
$n \log_2 n$ * Big improvements.

k-ary HEAPS:

Complete k-ary tree has height: $\lfloor \log_k n \rfloor = \dfrac{\log_2 n}{\log_2 K}$ $\Longrightarrow$



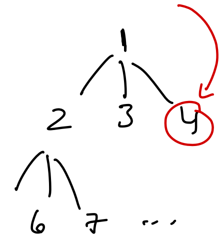
The positions in an array are given by:



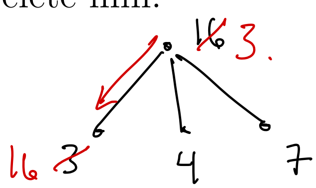
• $\text{parent}[x] = \left\lfloor \dfrac{x + K - 2}{K} \right\rfloor$

• Right child: $\underbrace{Kx + 1}$

• Left child $= \underbrace{Kx - (K-2)}$



Delete min:



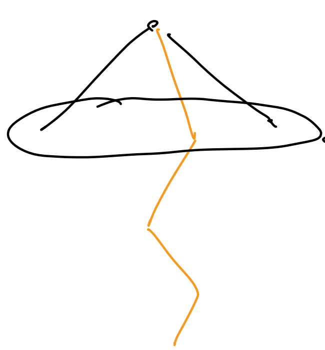
comparisons

$= K\lfloor \log_K n \rfloor$

min of K+1...

height

Bottom-up (fast) delete-min:



comparisons :

$$\Theta((\underbrace{K-1}) \lfloor \log_K n \rfloor)$$

time to find the path of smallest children: min of K children.

Insert: $\lfloor \log_K n \rfloor$.

What is the best k-value?

* Trade-off between insert/delete! → insert + delete.

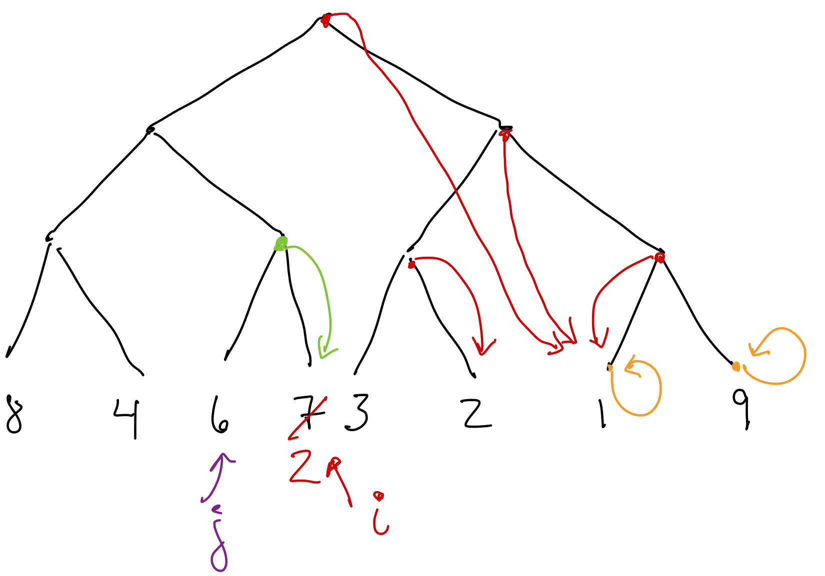
$$\log_K n + (K-1)\log_K n$$

$$\log_K n\,(K) \rightarrow \frac{K}{\log_2 K} \cdot \log_2 n$$

minimize.

Tournament Trees

- Complete Binary tree
- Data in leaves
- Represents outcomes of **tournament**
- Internal nodes point to smallest descendant leaf: pointer called σ.



leaf pointers point to themselves.

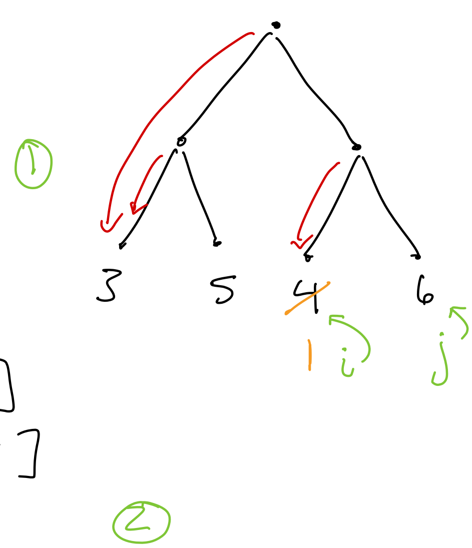
Number of nodes: n leaves, $n-1$ internal nodes. $\}$ $2n-1$ total.

OPERATIONS:

1. Update(i,k): Leaf i gets new key k:

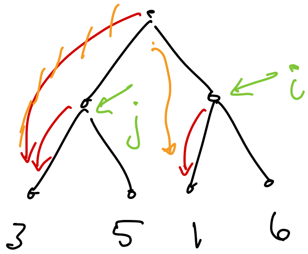
ex $K = 1$

```
Key[i] = K
σ[i] = i
while i ≠ 1 do
    j = sibling[i]
    if Key[σ[j]] < Key[σ[i]]
        σ[parent(i)] = σ[j]
    else
```



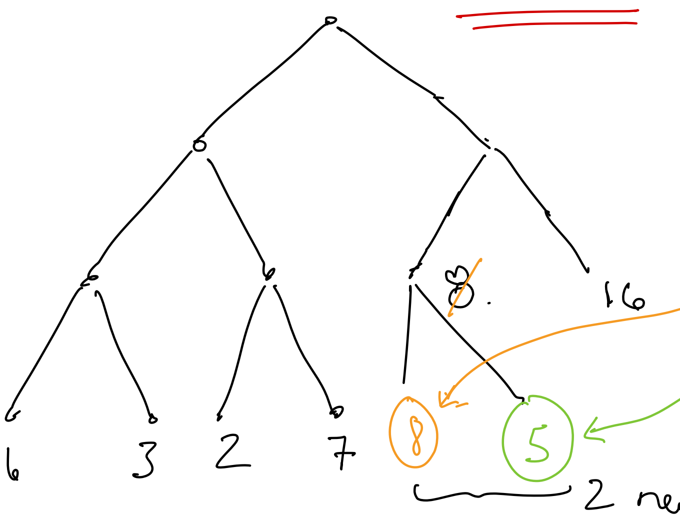
$\sigma(\text{parent}[i]) = \sigma[i]$.

$i = \text{parent}[i]$



2. Insert(k,n): insert key k into a tree with n leaves:

Ex. $K = 5$



$\text{Key}[2n+1] = K$. } * new key

$\text{Key}[2n] = \text{Key}[n]$ } copy internal node...

$\sigma[2n] = 2n$.

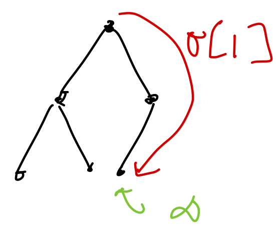
$\sigma[2n+1] = 2n+1$ }

update $(2n+1, K)$ } fix tree.

3. Deletemin:

Report $\sigma[1]$.

update $(\sigma[1], \infty)$

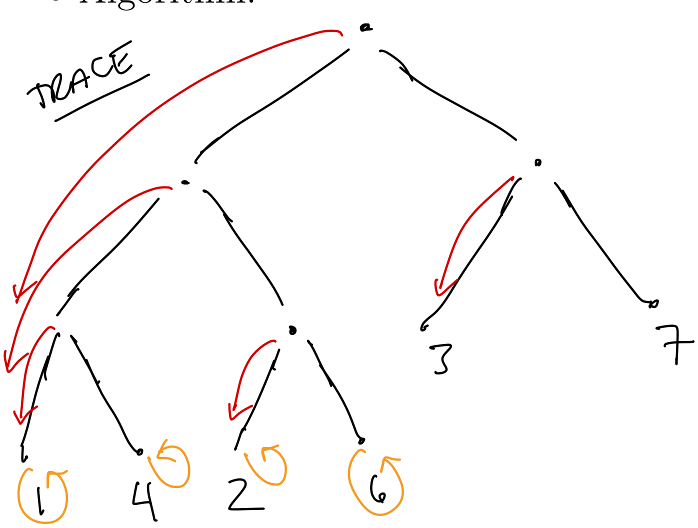


} updates tree to exclude key of $\sigma[1]$.

4. MakeTree(n): Create a tournament tree for n given keys.

- Create arrays of size $2n - 1$ for $\sigma[]$ and $key[]$.
- Algorithm:

TRACE



for $i = n$ to $2n-1$ — leaf pointers.

$\sigma[i] = i$

$\text{Key}[i] = \_\_\_\_$ } fill in Keys[ ] array w/ data.

For $i = n-1$ down to 1 :

min of children... { if $\text{Key}(\sigma[2i]) < \text{Key}[\sigma[2i+1]]$

$\sigma[i] = \sigma[2i]$.

else $\sigma[i] = \sigma[2i+1]$

Number of comparisons: $n - 1$ (# games in tournaments).

Tournament Tree Sort: As w/ heapsort, we can create a tree and then delete the min over and over to sort the data...

$$\# \text{comp.} = \underbrace{\text{maketree}}_{\substack{\text{Alg. above does 'n'} \\ \text{comparisons in the} \\ \text{for loop!}}} + n \cdot \underbrace{\text{deletemin}}_{\lfloor \log_2(\text{tree height}) \rfloor}.$$

\# nodes in tree.

$$= n + n \cdot \lfloor \log_2(2n-1) \rfloor$$