Leftist Heaps

CMSC 420: Lecture 10

Priority Queue ADT

- Efficiently support the following operations on a set of keys:
  - *findmin*: return the smallest key
  - *deletemin*: return the smallest key & delete it
  - *insert*: add a new key to the set
  - *delete*: delete an arbitrary key

- All the balanced-tree dictionary implementations we've seen support these in O(log n) time.

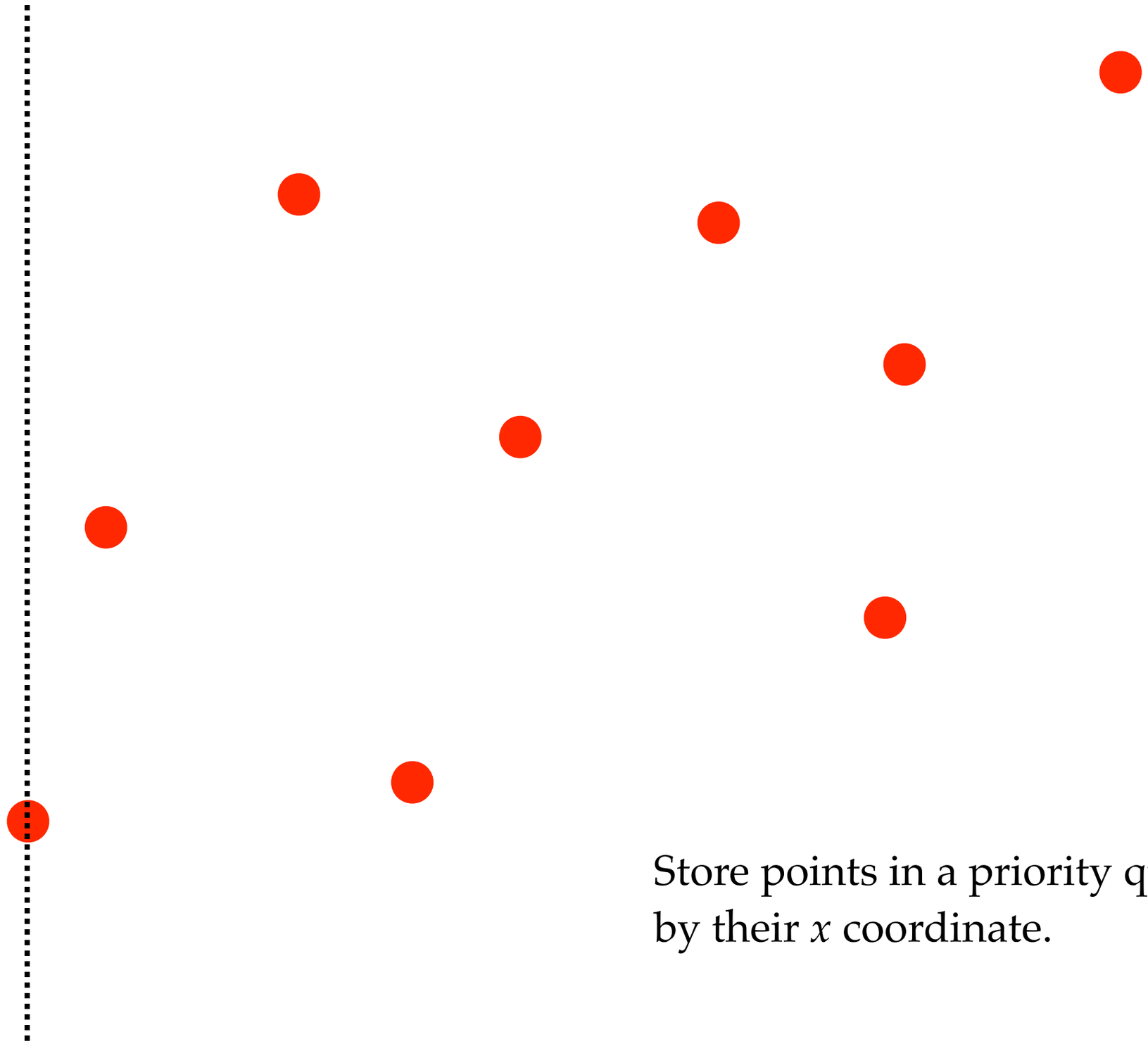
- Would like to be able to do *findmin* faster (say O(1)).

Job Scheduling: UNIX process priorities

```
PRI COMM
 14 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker
 31 -bash
 31 /Applications/iTunes.app/Contents/Resources/iTunesHelper.app/Contents/MacOS/iTunesHelper
 31 /System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
 31 /System/Library/CoreServices/FileSyncAgent.app/Contents/MacOS/FileSyncAgent
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/MacOS/AppleVNCServer
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/Support/RFBRegisterMDNS
 31 /System/Library/CoreServices/RemoteManagement/AppleVNCServer.bundle/Contents/Support/VNCPrivilegeProxy
 31 /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
 31 /System/Library/CoreServices/coreservicesd
...
 31 /System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/Resources/usbmuxd
 31 /System/Library/Services/AppleSpell.service/Contents/MacOS/AppleSpell
 31 /sbin/launchd
 31 /sbin/launchd
 31 /usr/bin/ssh-agent
 31 /usr/libexec/ApplicationFirewall/socketfilterfw
 31 /usr/libexec/hidd
 31 /usr/libexec/kextd
...
 31 /usr/sbin/mDNSResponder
 31 /usr/sbin/notifyd
 31 /usr/sbin/ntpd
 31 /usr/sbin/pboard
 31 /usr/sbin/racoon
 31 /usr/sbin/securityd
 31 /usr/sbin/syslogd
 31 /usr/sbin/update
 31 autofsd
 31 login
 31 ps
 31 sort
 46 /Applications/Preview.app/Contents/MacOS/Preview
 46 /Applications/iCal.app/Contents/MacOS/iCal
 47 /Applications/Utilities/Terminal.app/Contents/MacOS/Terminal
 50 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds
 50 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/CarbonCore.framework/Versions/A/Support/fseventsd
 62 /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
 63 /Applications/Safari.app/Contents/MacOS/Safari
 63 /Applications/iWork '08/Keynote.app/Contents/MacOS/Keynote
 63 /System/Library/CoreServices/Dock.app/Contents/Resources/DashboardClient.app/Contents/MacOS/DashboardClient
 63 /System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer
 63 /System/Library/CoreServices/loginwindow.app/Contents/MacOS/loginwindow
 63 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/CoreGraphics.framework/Resources/WindowServer
 63 /sbin/dynamic_pager
 63 /usr/sbin/UserEventAgent
 63 /usr/sbin/coreaudiod
```

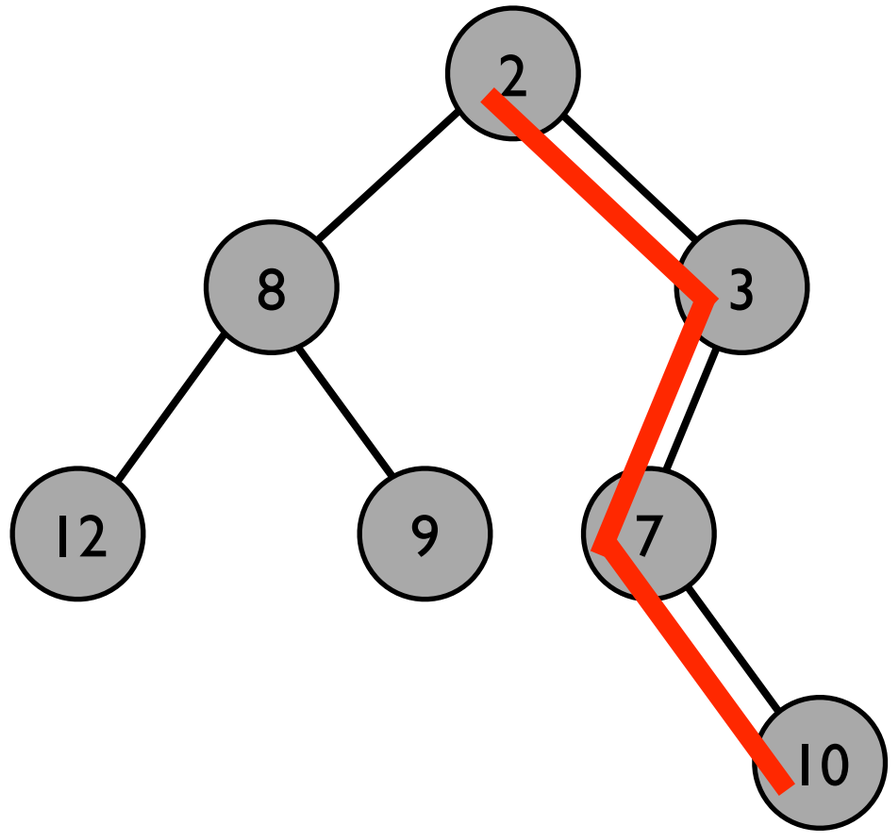
When scheduler asks "What should I run next?" it could *findmin(H)*.

Plane Sweep: Process points left to right:



Store points in a priority queue, ordered by their x coordinate.

Heap-Ordered Trees

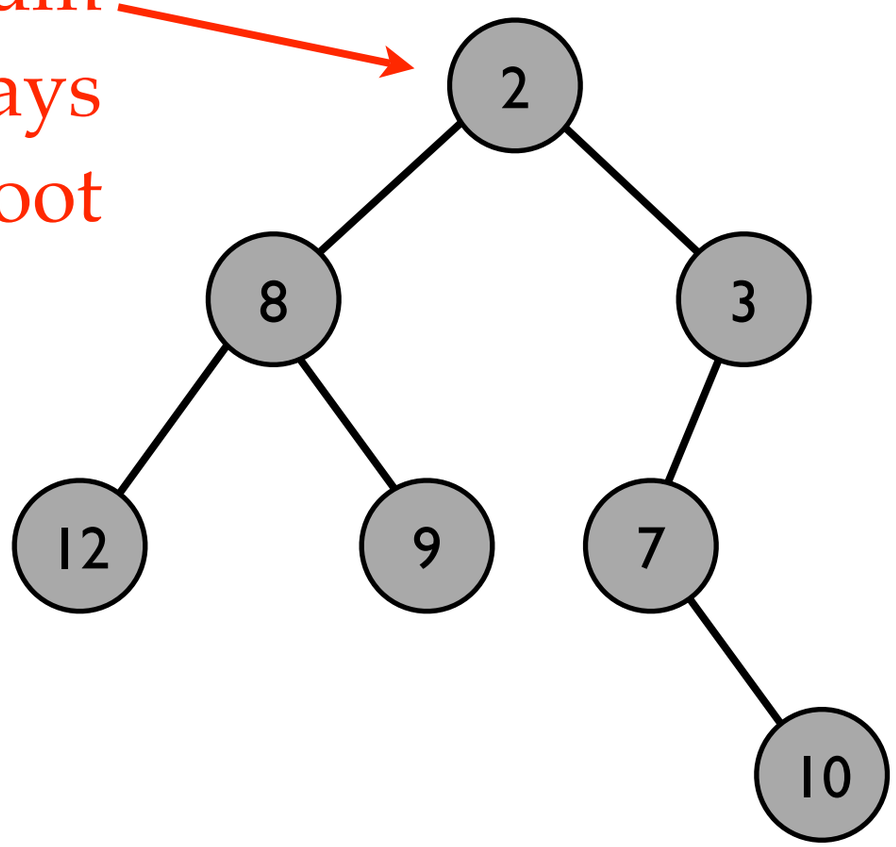


Along each path
keys are monotonically
non-decreasing

- The keys of the children of u are ≥ the key(u), for all nodes u.
- (This "heap" has nothing to do with the "heap" part of computer memory.)
- [Symmetric max-ordered version where keys are monotonically non-increasing]

Heap – Find min

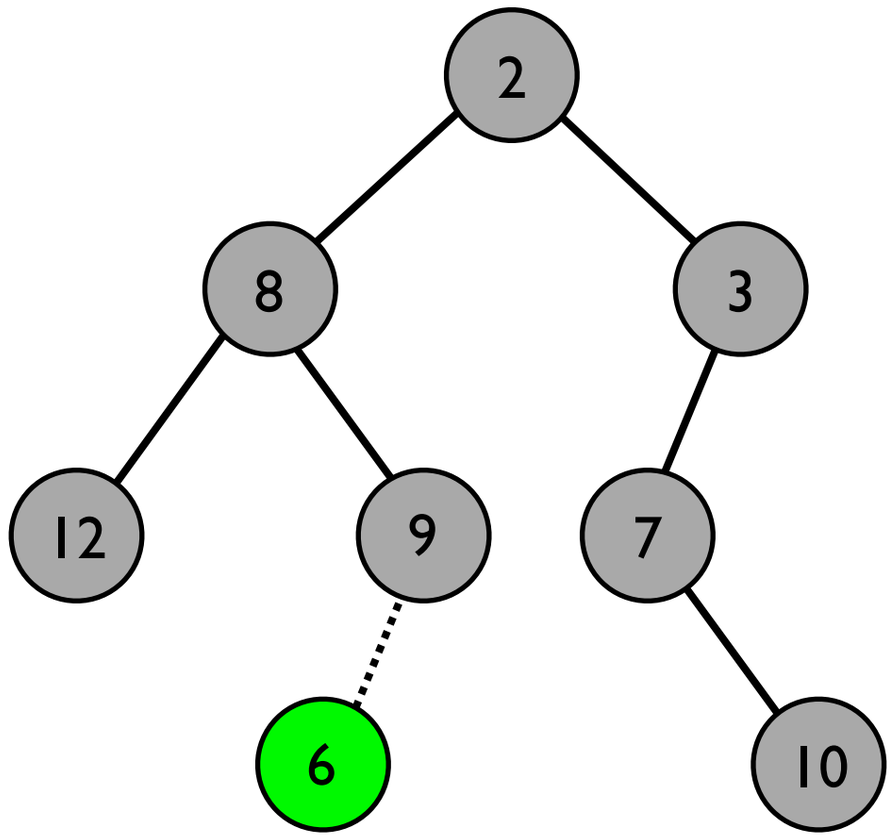
The minimum element is always the root



Heap – Insert

1. Add node as a leaf
(we'll see where later)

2. "*sift up:*" while current node is > its parent, swap them.



Heap – Delete(*i*)

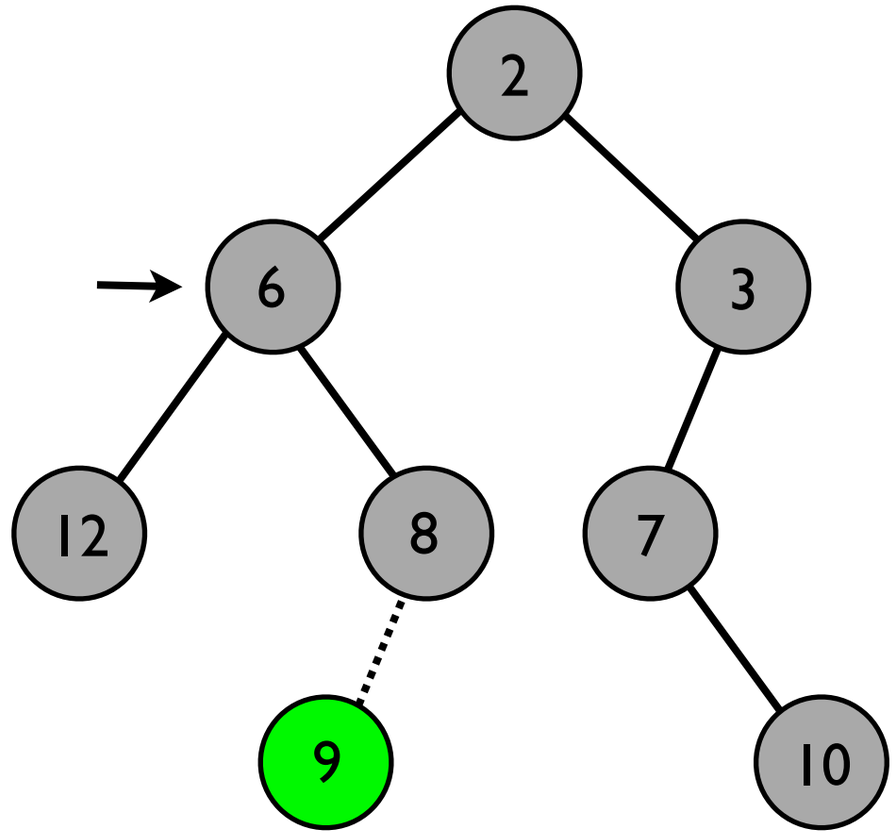
1. need a pointer to
node containing key *i*

2. replace key to delete *i* with key
j at a leaf node
(we'll see how to find a leaf soon)

3. Delete leaf

4. If $i < j$ then sift up, moving *j* up
the tree.

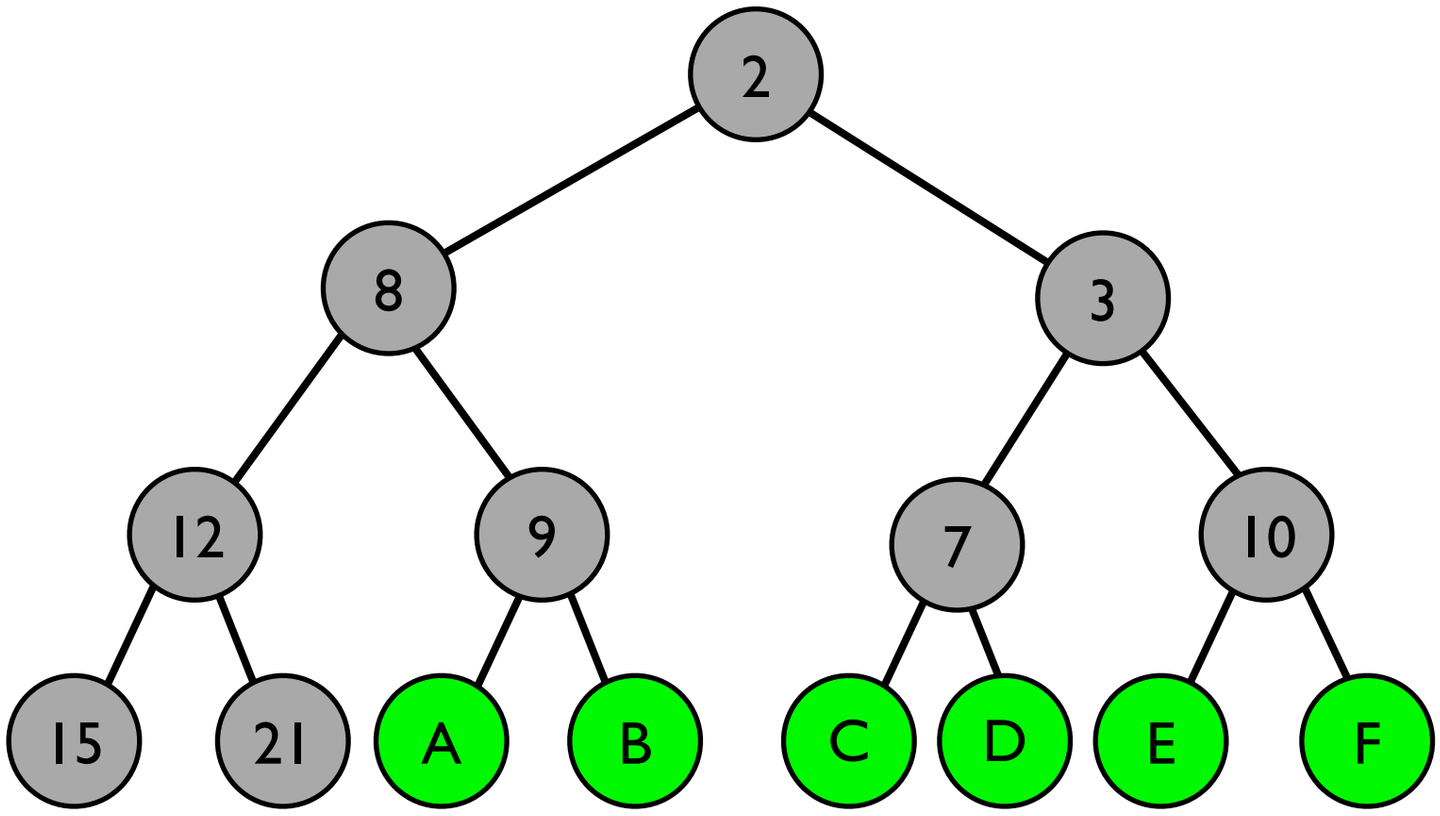
If $i > j$ then *"sift down"*: swap
current node with **smallest of children** until its bigger than all
of its children.



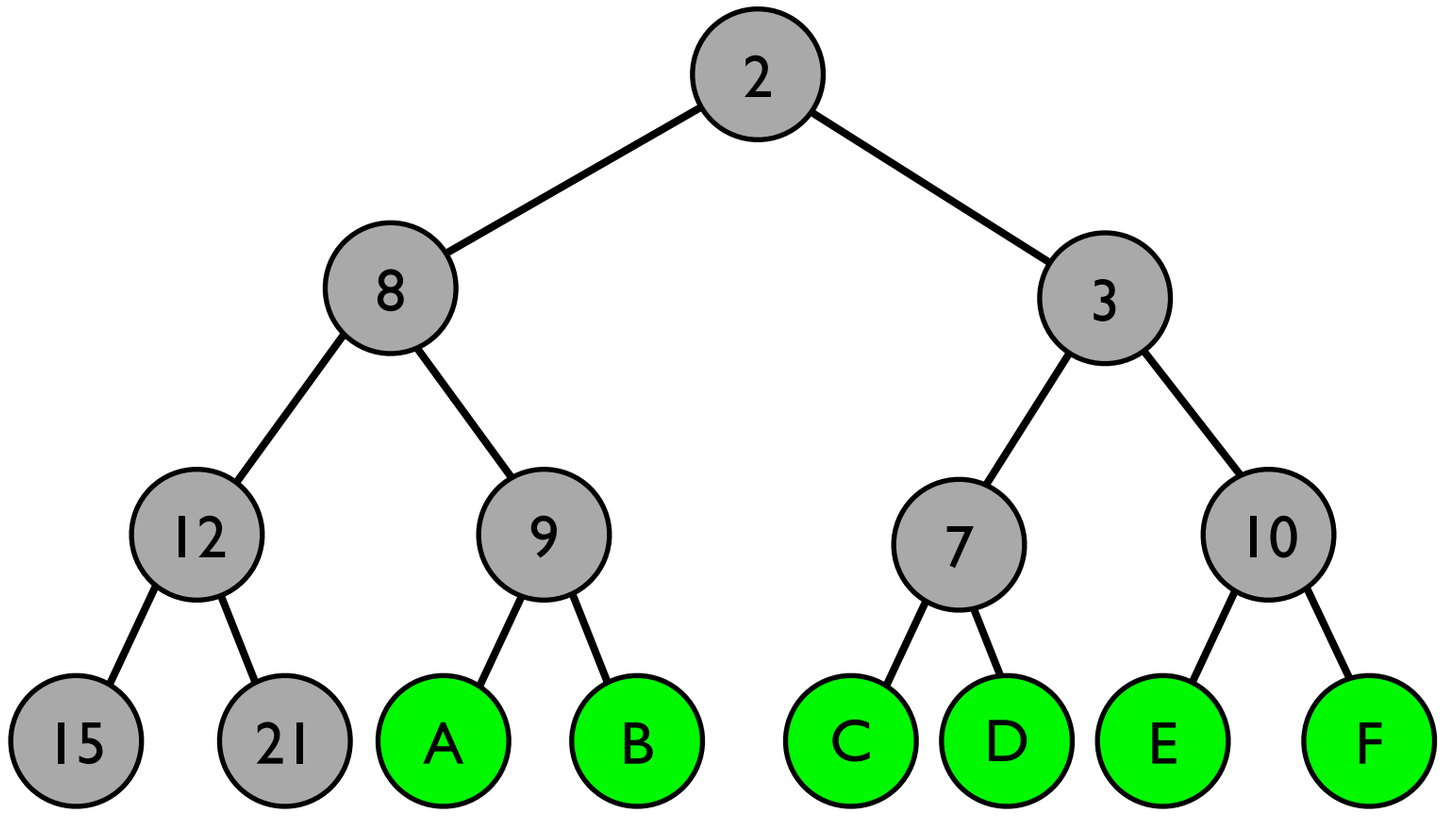
Time Complexity

- *findmin* takes O(1) time
- *insert*, *delete* take time O(tree height) plus the time to find the leaves.
- *deletemin*: same as delete
- But how do we find leaves used in *insert* and *delete*?
  - *delete*: use the last inserted node.
  - *insert*: choose node so tree remains complete.

Store Heap in a Complete Tree



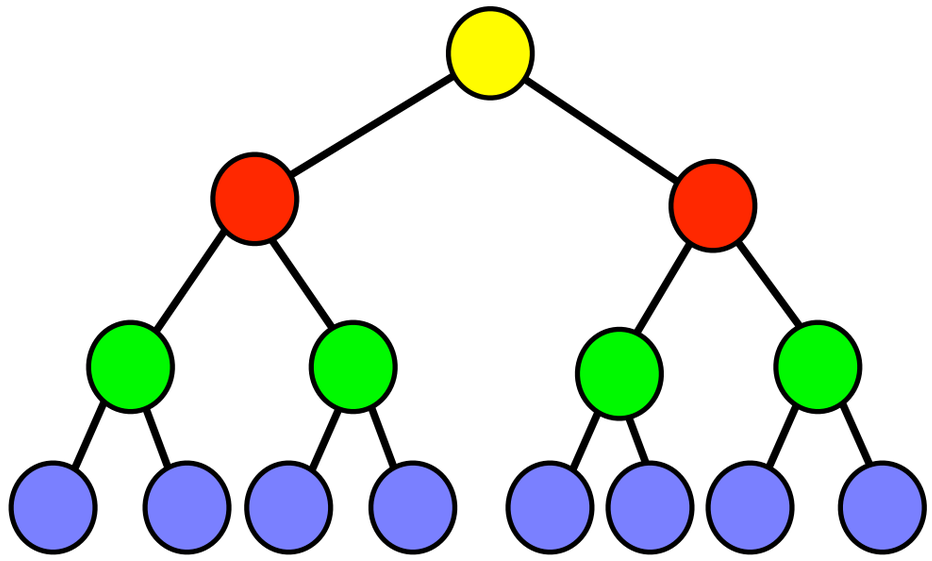
Store Heap in a Complete Tree



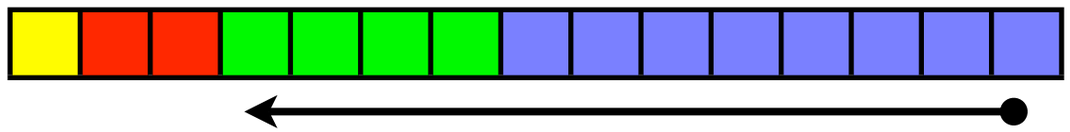
2	8	3	12	9	7	10	15	21	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

left(i): $2i$ **if** $2i \leq n$ **otherwise** 0
right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** 0
parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** 0

Make Heap

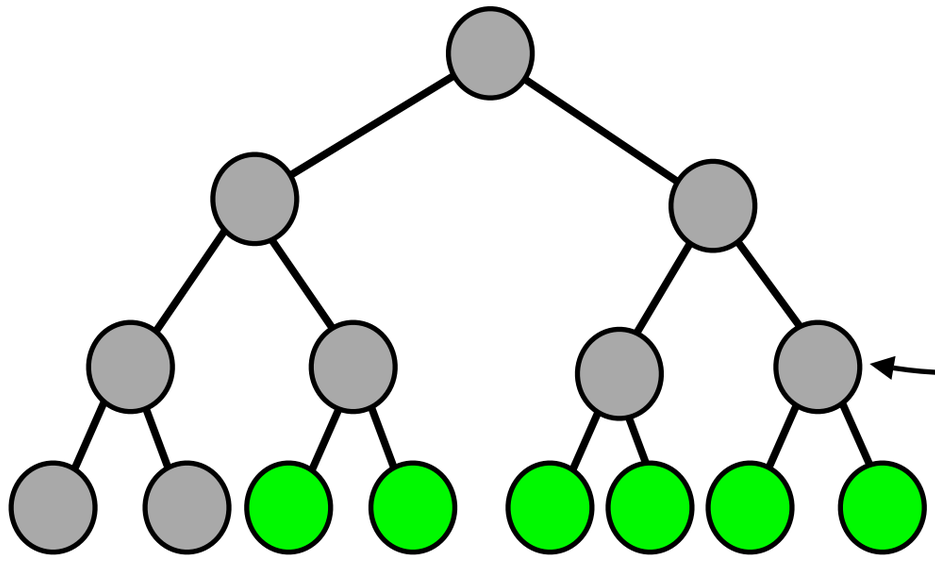


- n inserts gives a $O(n \log n)$ time bound.
- Better:
  - put items into array arbitrarily.
  - **for** $i = n \ldots 1$, *siftdown*(i).
- Each element trickles down to its correct place.



By the time you sift level i, all levels $i + 1$ and greater are already heap ordered.

Make Heap – Time Bound



There are at most $n / 2^h$ items at height h.

Siftdown for all height h nodes is $O(h \bullet n / 2^h)$ time

Total time

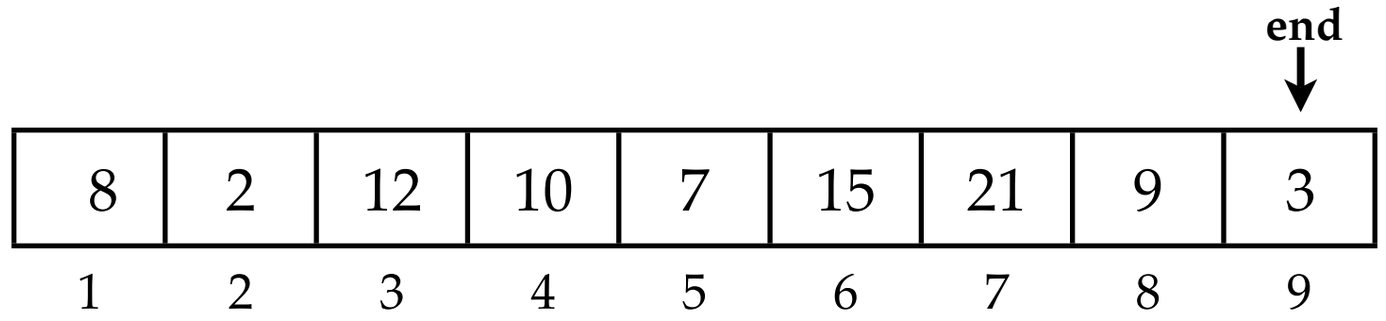
$= O(\sum_h \boldsymbol{h} \bullet \boldsymbol{n} / 2^h)$ [sum of time for each height]

$= O(\boldsymbol{n} \sum_h (\boldsymbol{h} \ / \ 2^h))$ [factor out the n]

$= O(\boldsymbol{n})$ [sum bounded by const]

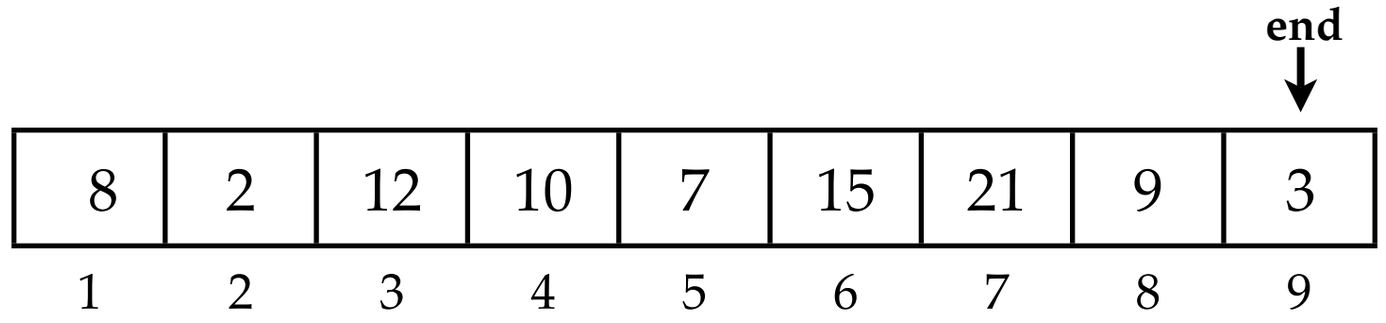
Heapsort – Another application of Heaps

Given unsorted array of integers

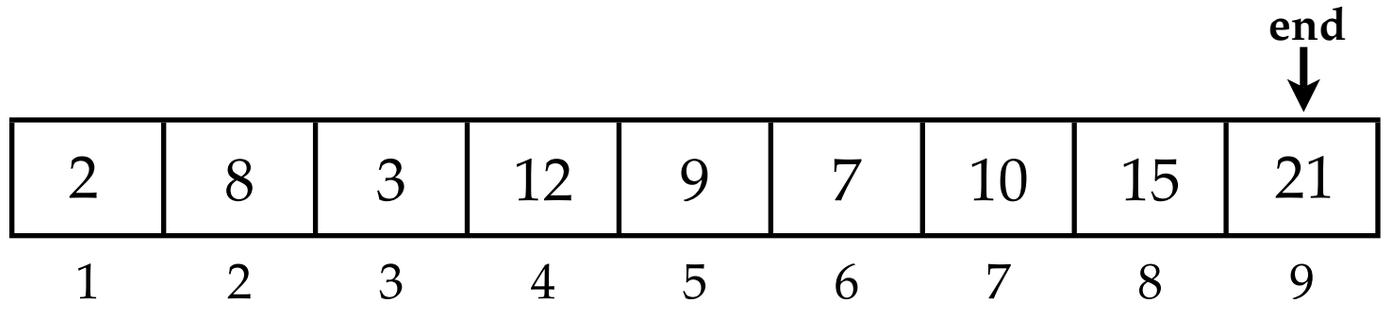


Heapsort – Another application of Heaps

Given unsorted array of integers



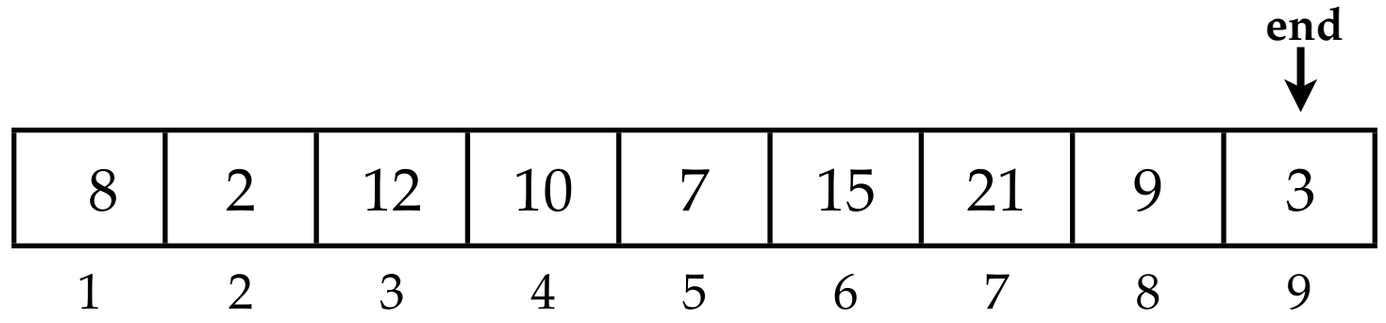
makeheap – O(n)
Now first position has smallest item.



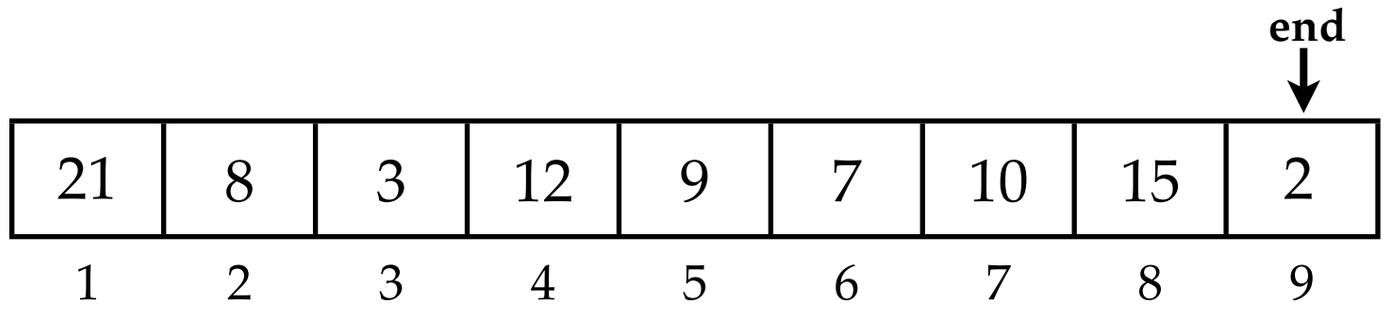
Swap first & last items.

Heapsort – Another application of Heaps

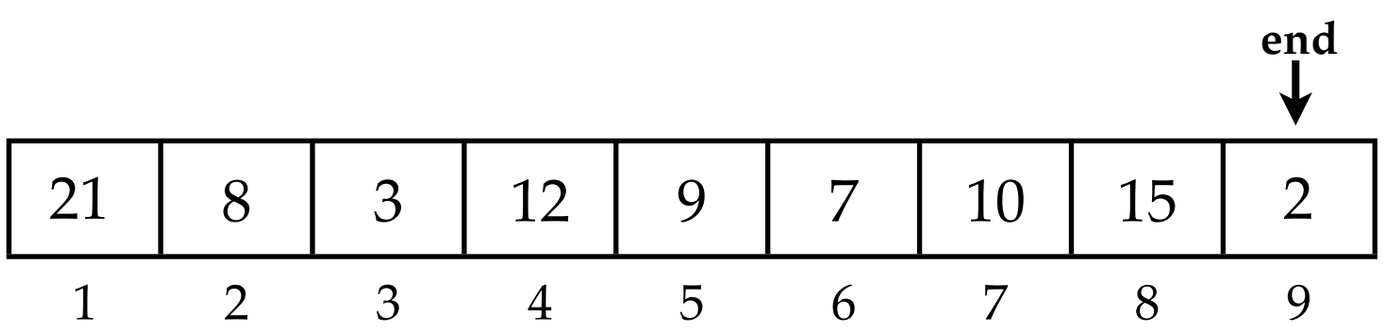
Given unsorted array of integers



makeheap – O(n)
Now first position has smallest item.

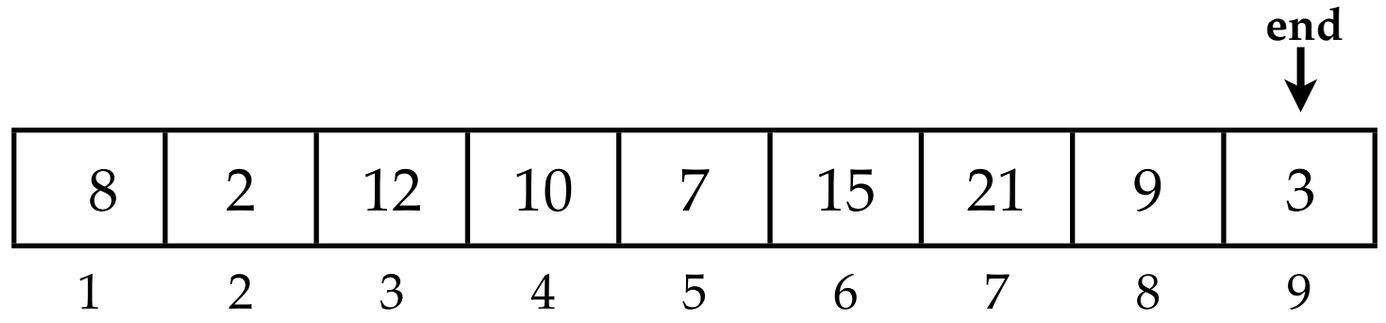


Delete last item from heap.

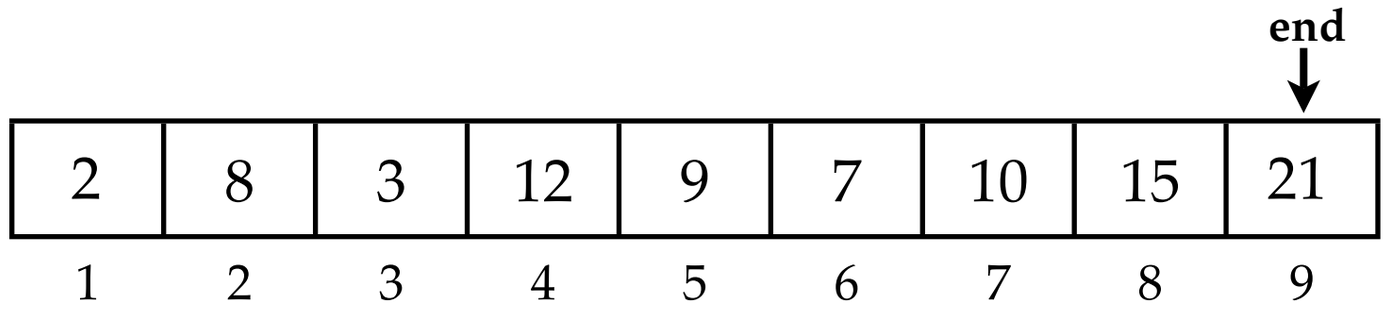


Heapsort – Another application of Heaps

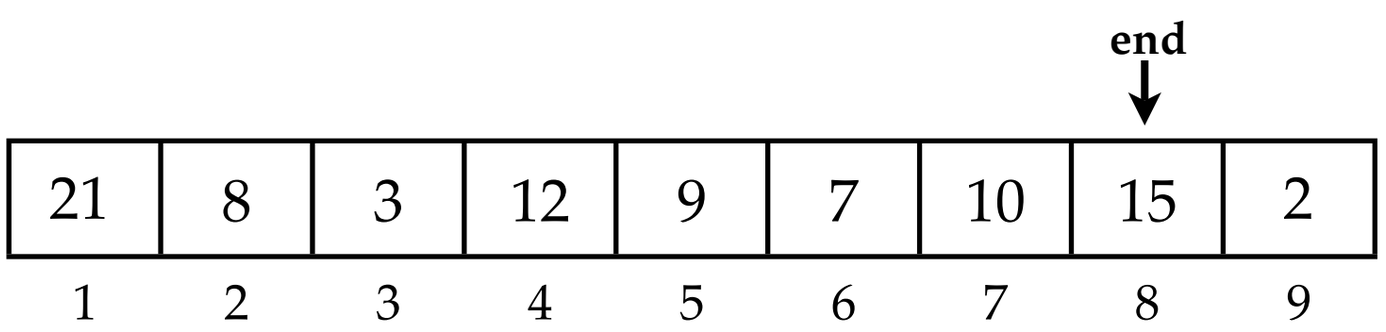
Given unsorted array of integers



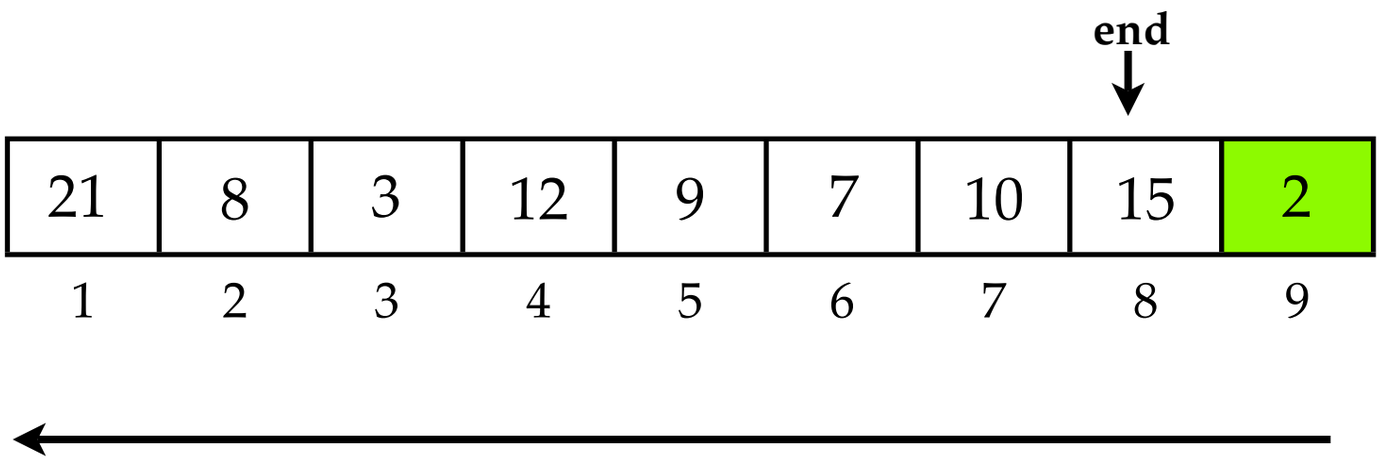
makeheap – O(n)
Now first position has smallest item.



Delete last item from heap.



siftdown new root key down



d-Heaps

- What about complete non-binary trees (e.g. every node has d children)?
  - *insert* takes $O(\log_d n)$ [because height $O(\log_d n)$]
  - *delete* takes $O(d \log_d n)$ [why?]

- Can still store in an array.

- If you have few deletions, make d bigger so that tree is shorter.
- Can tune d to fit the relative proportions of inserts / deletes.

Find(i) ? How would you do it?

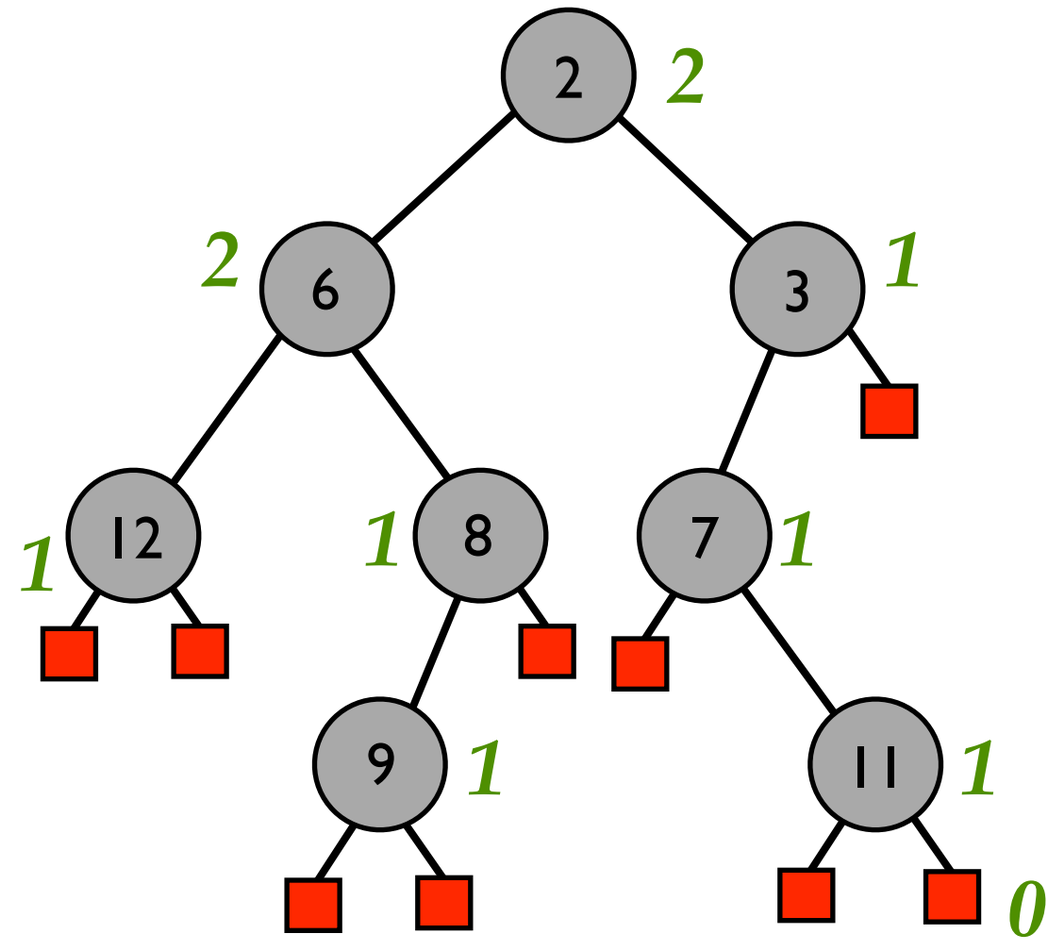
Leftist Heaps

- Often want to merge heaps:
  - *meld*(H_1, H_2): return new heap with the keys from H_1 and H_2, destroying heaps H_1 and H_2.
  - Hard to do with the complete tree implementation of heaps above.

- Idea: use ***imbalance*** to make melds fast.

Null path length

$$\text{npl}(u) = \begin{cases} 0 & u \text{ is an external node} \\ 1 + \min\{\text{npl}(\text{left}(u)), \text{npl}(\text{right}(u))\} & o.w. \end{cases}$$



Null Path Length / Rank / Balance

- A theme we've seen several times: associate a value with each node describing a property of its subtrees.

- *balance* - AVL trees - difference between right and left heights.

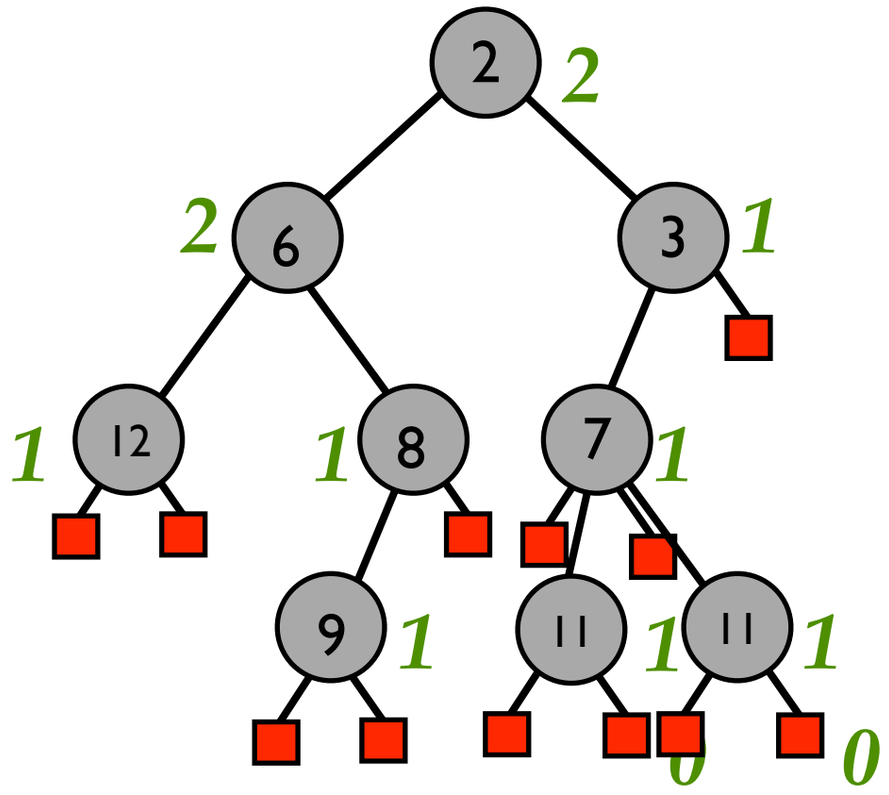
- *rank* - splay trees = floor(log #descendants)
  (used for the analysis only!)

- *null path length* - shortest distance to get to a null pointer.

Leftist Trees

A tree is a *leftist tree* if npl(left(u)) ≥ npl(right(u))

A *leftist heap* is a leftist tree with keys in heap order.



Any non-leftist tree can be made leftist by swapping left & right children at node where leftist condition is violated.

Leftist trees have a short path

Thm. If rightmost path of leftist tree has r nodes, then whole tree has at least $2^r - 1$ nodes.

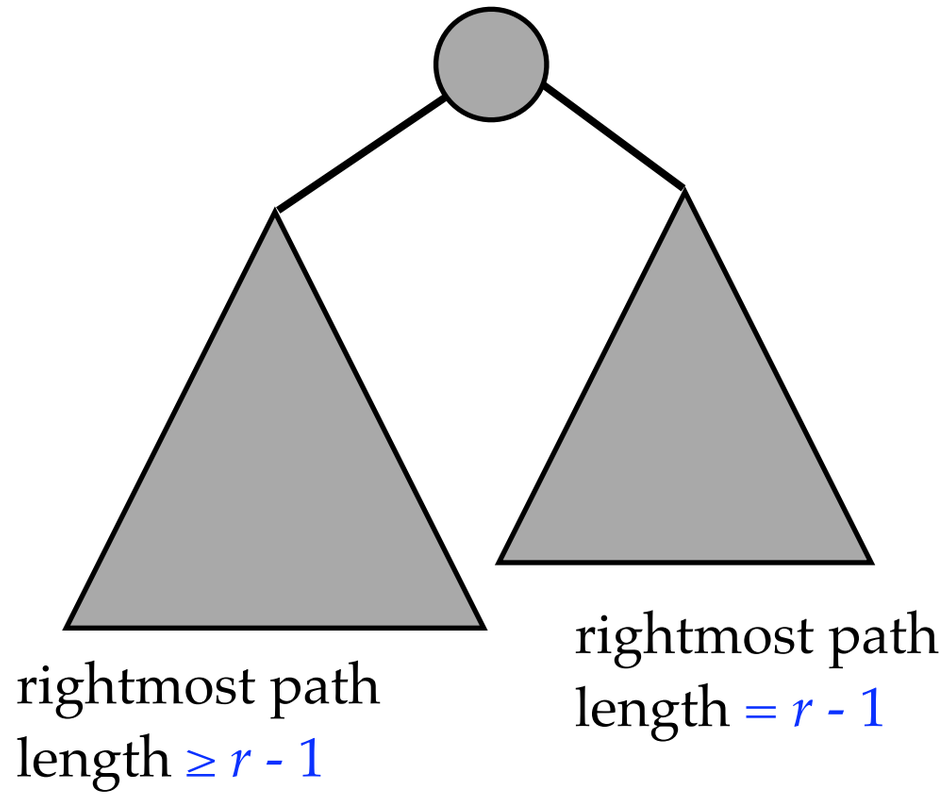
Proof.

Base Case: When $r = 1$, $2^1 - 1 = 1$ & tree has ≥ 1 node.

Induction hypothesis: Assume $N(i) \geq 2^i - 1$ for $i < r$.

Induction step: Left and right subtrees of the root have at least $2^{r-1} - 1$, nodes.

Thus, at least $2(2^{r-1} - 1) + 1 = 2^r - 1$ nodes in original tree. □



Therefore $n \geq 2^r - 1$, so r is $O(\log n)$

Meld is the fundamental operation

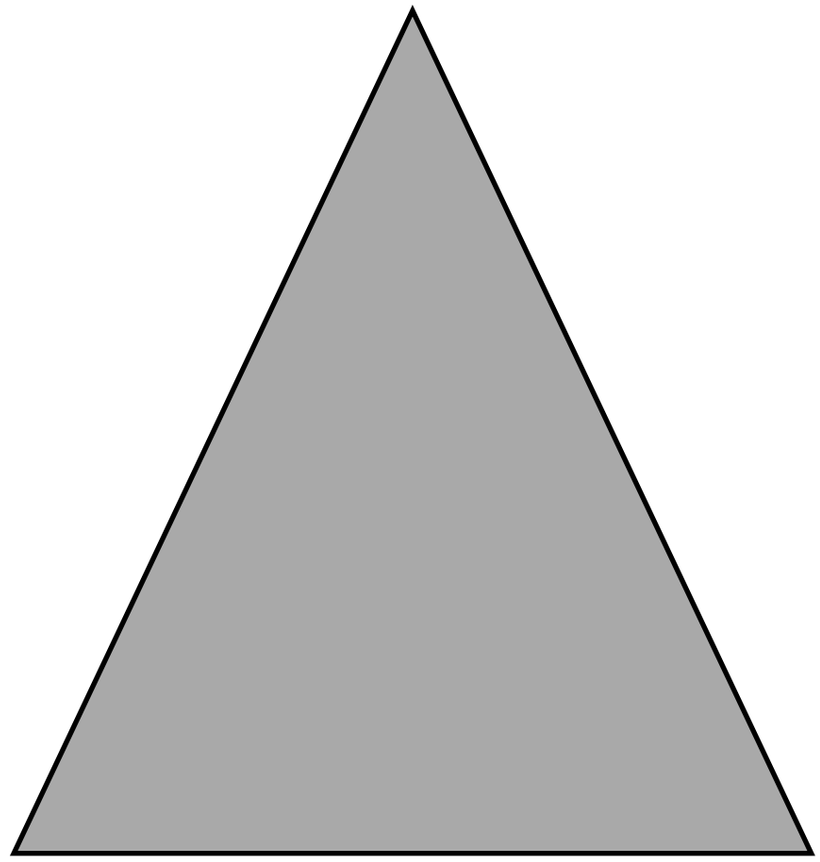
meld(H_1, H_2): return new heap with the keys from H_1 and H_2, destroying heaps H_1 and H_2.

As with *splay* in splay trees, *meld* is used to implement *insert, delete, deletemin*.

Insert Implemented with Meld

insert(H, 9) →

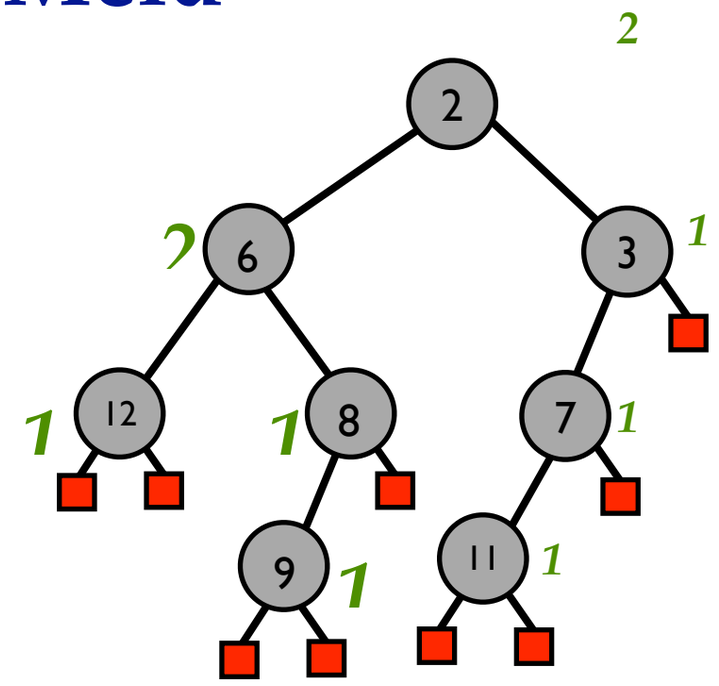
meld(

, )

Make a single-node heap

DeleteMin Implemented with Meld

deletemin(H) →



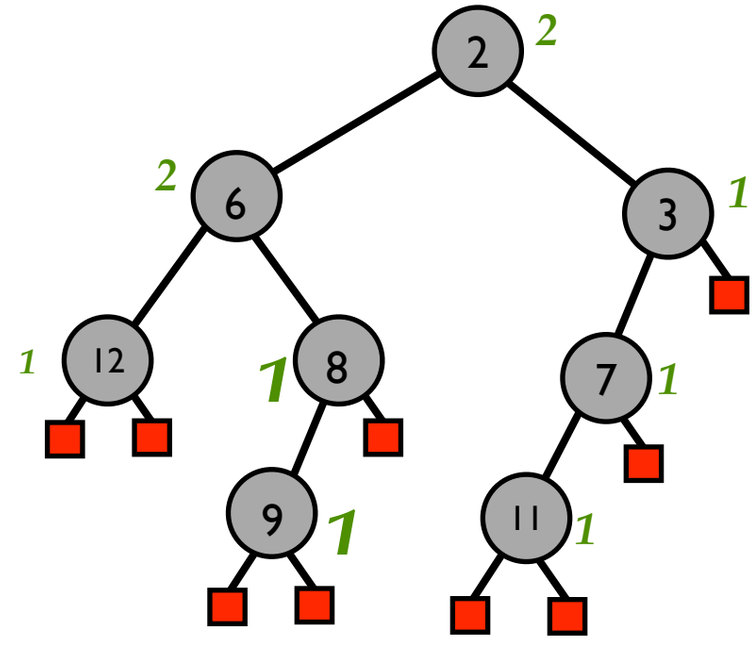
meld(,)

Are the npl values right in the subtrees?

Delete(*i*) Implemented with Meld

delete(H, 6) →

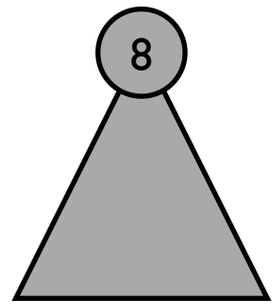
Again, assume we have a pointer to the node containing 6.



Are we done?

No: must check to see if leftist property holds, and swap if not.

meld(,) →

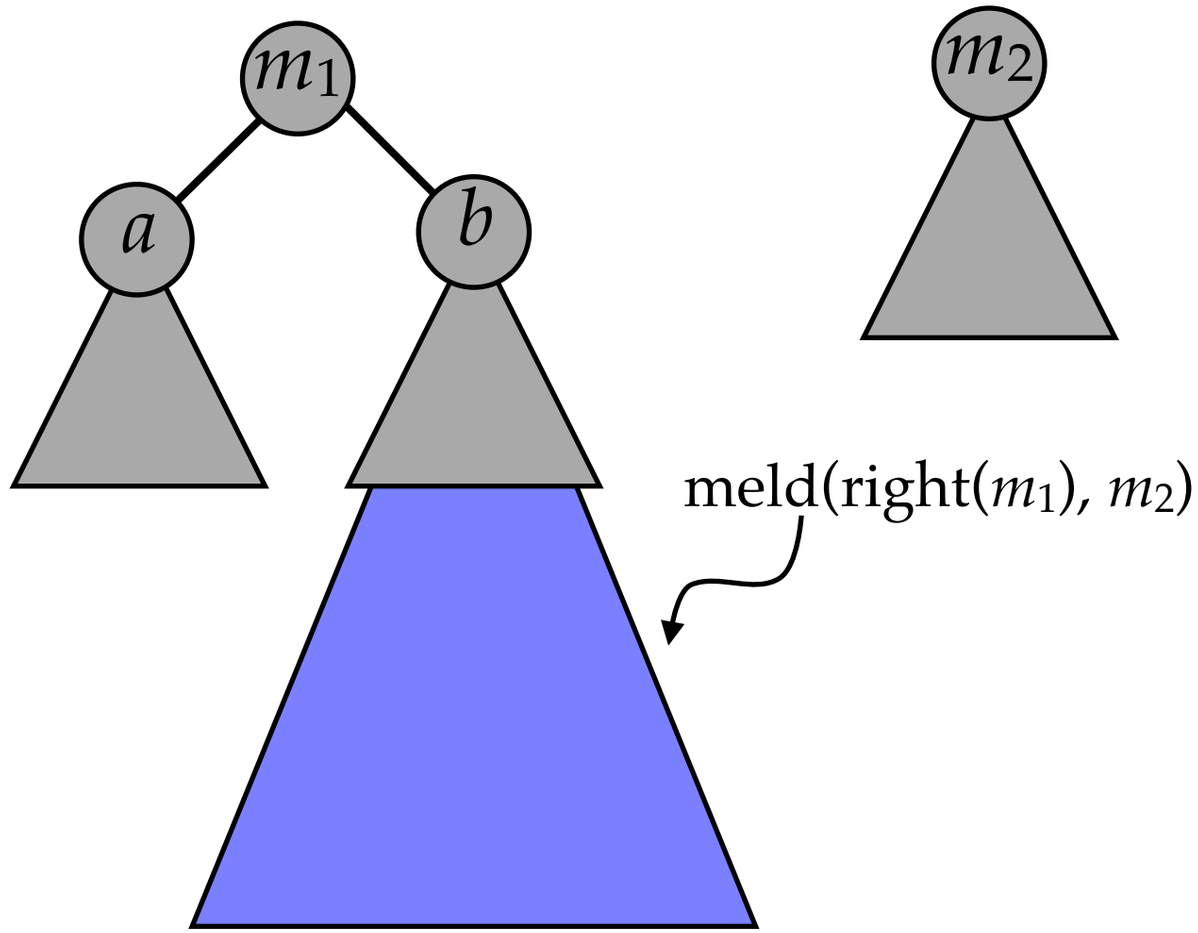


Meld – finally....

meld(null, null) = null

meld(null, H) = H
meld(H, null) = H

meld(H_1, H_2) =



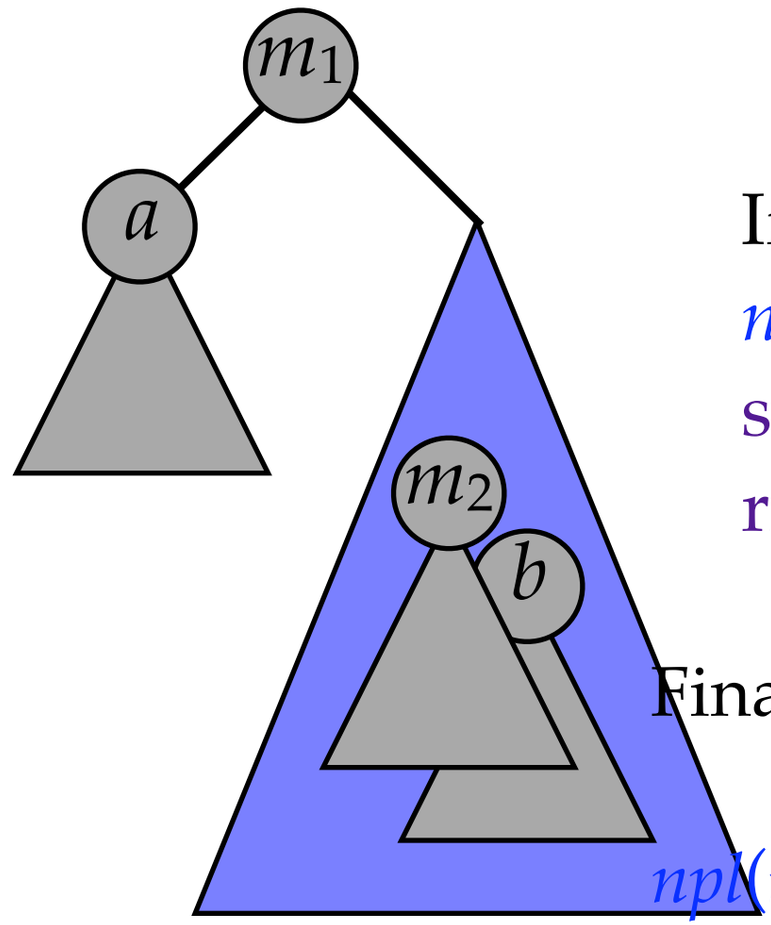
Meld – finally....

meld(null, null) = null

meld(null, H) = H
meld(H, null) = H

meld(H_1, H_2) =

Make the new tree leftist...



If $npl(\text{right}(m_1)) > npl(\text{left}(m_1))$,
swap the left &
right children.

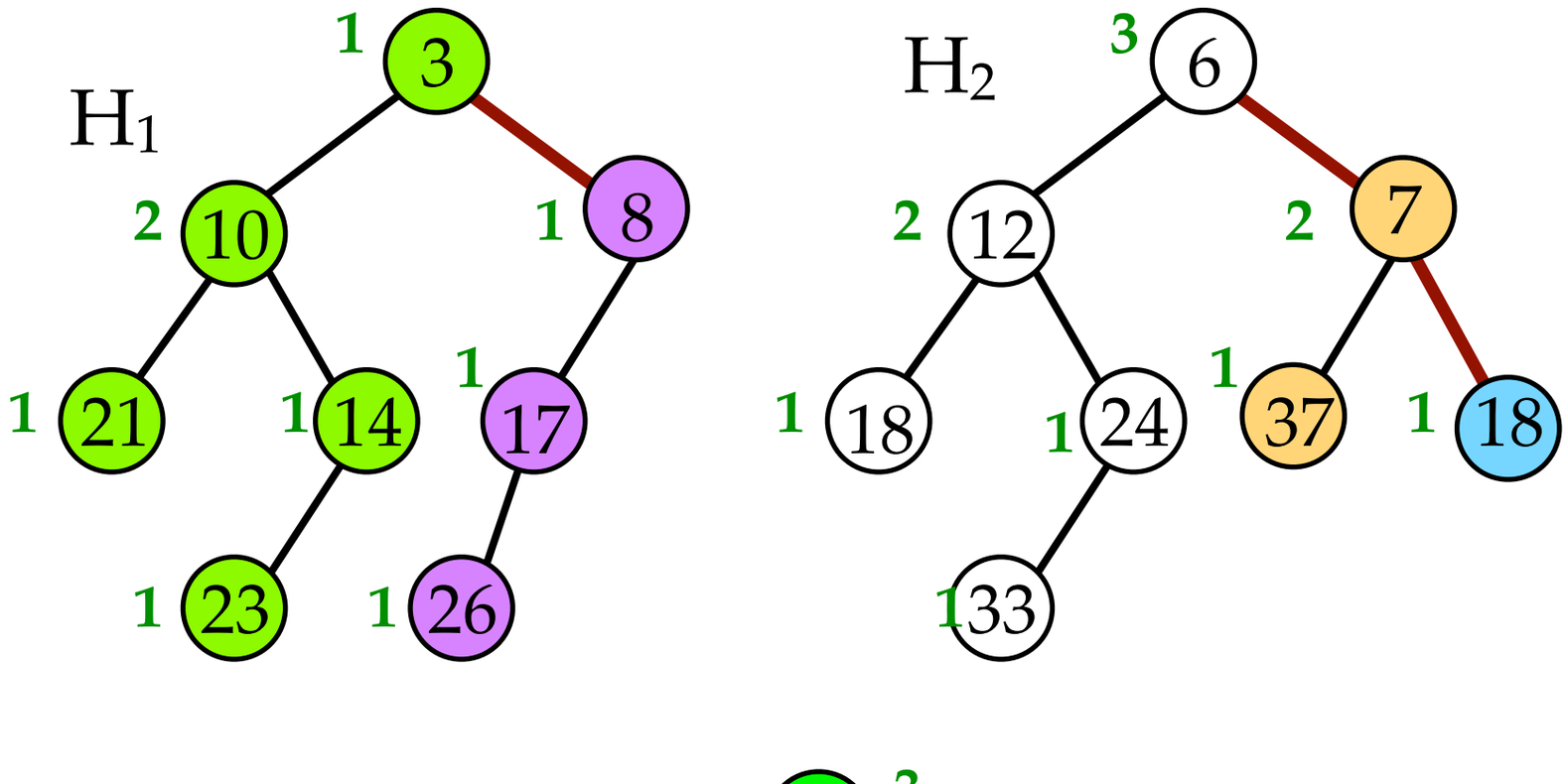
Finally, update rank of m_1:

$npl(m_1) = 1 + npl(\text{right}(m_1))$

Meld Code (Python)

```
def meld(H1, H2):
    # the base cases with one or more empty trees
    if H1 == None: return H2
    if H2 == None: return H1

  # make H1 the heap with the smaller root value
    if H1.key > H2.key:
        H1, H2 = H2, H1

    H1.right = meld(H1.right, H2)

    # swap left and right subtrees if needed
    if H1.left == None or H1.left.npl < H1.right.npl:
        H1.left, H1.right = H1.right, H1.left

    # the null path length is one more that right child
    H1.npl = H1.right.npl + 1

    return H1
```

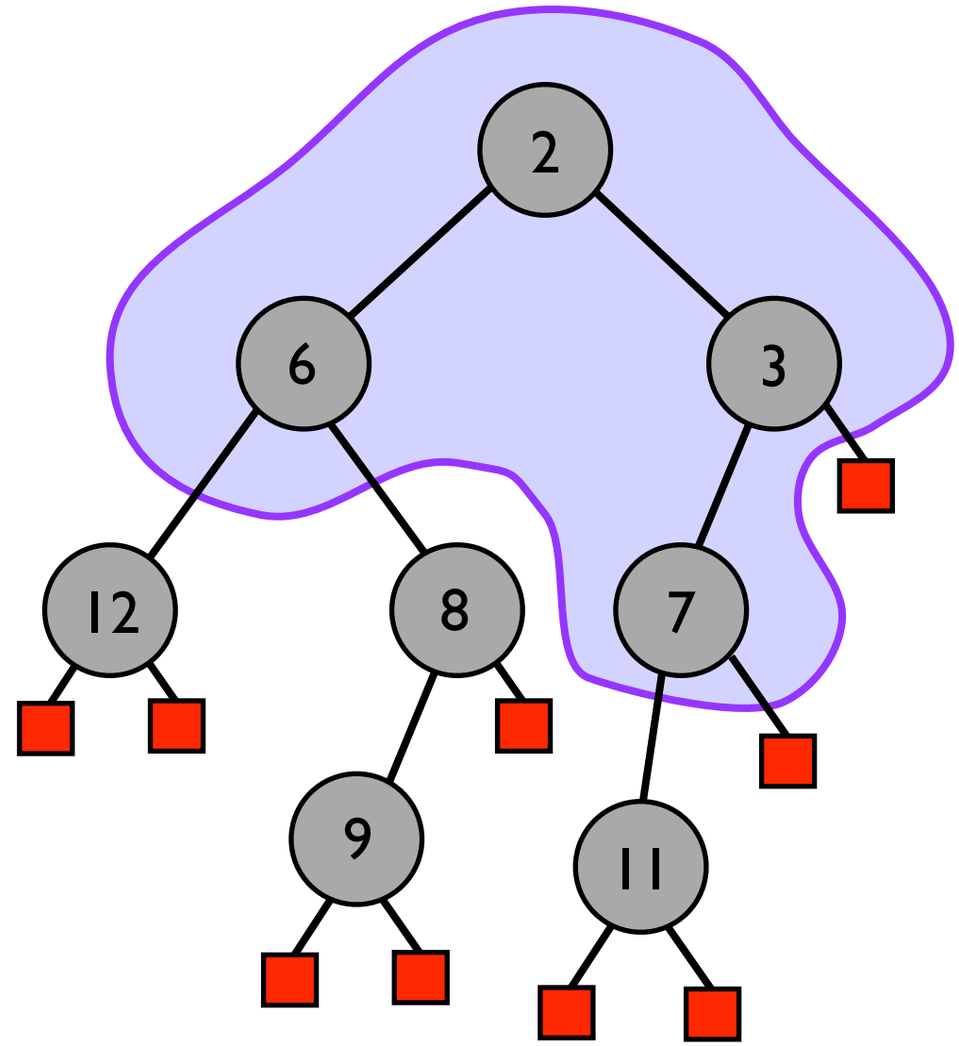
Meld Example



List Small Items

smallitems(H, *r*): return a list of keys < *r*

smallitems(H, 7.2) =



Preorder traversal, pruning trees with roots that are too large.

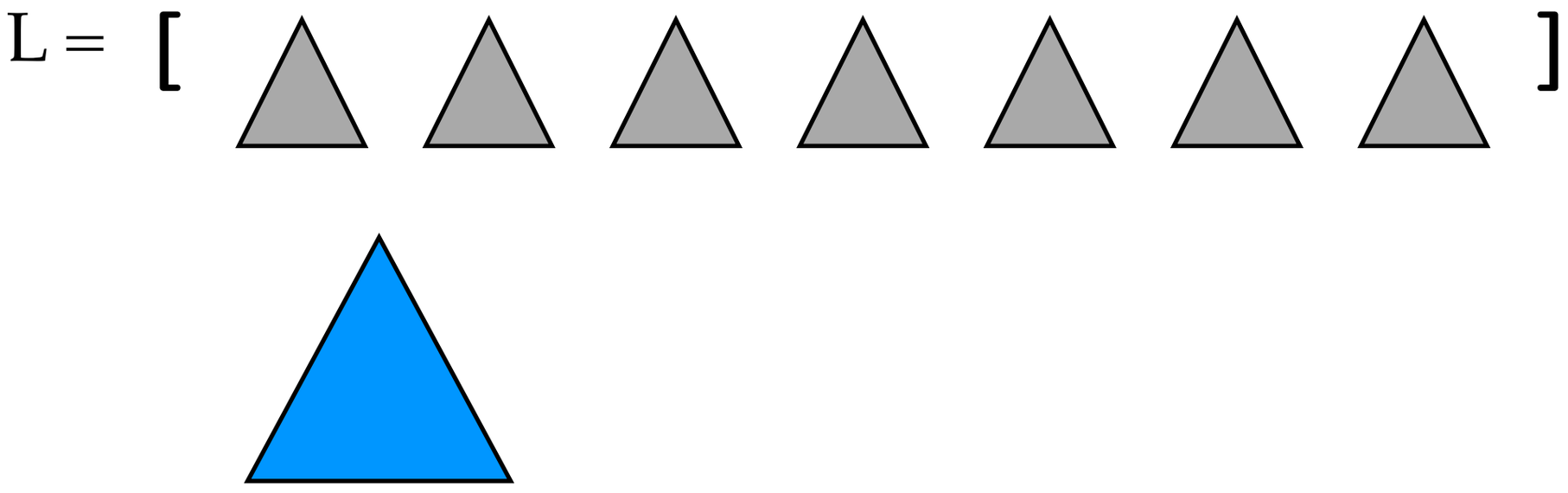
O(*m*) time, where *m* is the number of elements output.

Heapify

heapify(L): given a list of heaps H_1, H_2, ..., H_k, return a new heap that contains the union of keys in all of them.

(As usual, we're allowed to destroy each H_i and the list.)

Treat L as a queue
Repeat until only 1 heap left:
1. *meld* the front two items
2. *enqueue* the resulting heap:



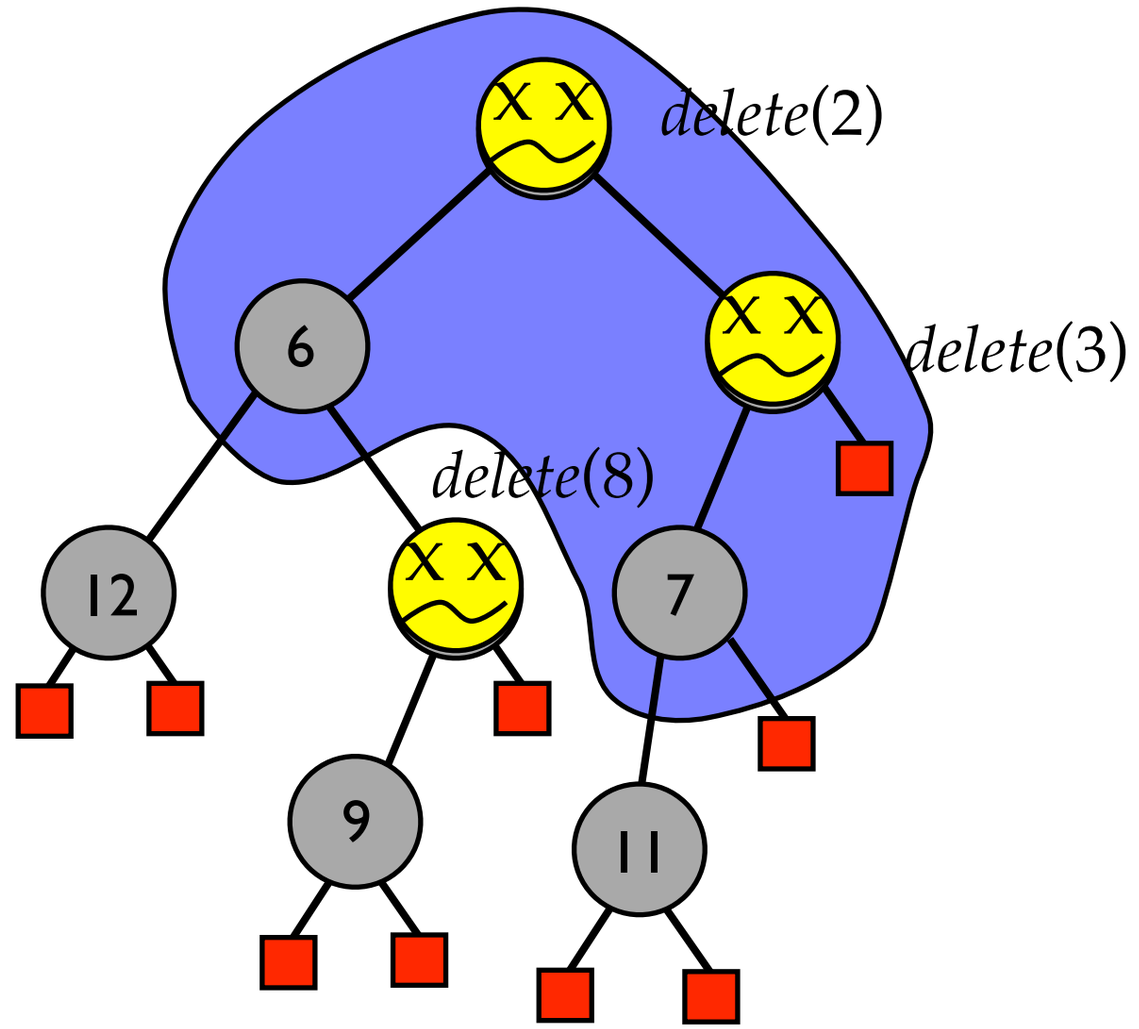
Lazy Deletion

Just mark nodes deleted;
don't actually change tree.

Now *delete*(*i*) and
deletemin() are O(1)

During *findmin*(), do
preorder traversal, making
a list L of subtrees for which
all ancestors are deleted.

Heapify(L)



L = [6 7]

Skew Heaps

- Self-adjusting version of leftist heaps
- Don't store *npl* (or any other auxiliary information at the nodes)
- **Difference**:
  - *always* swap the left & right subtrees at each step of meld
  - old rightmost path becomes new leftmost path
- Can show (beyond the scope of this class) that a series of *m insert*, *findmin*, *meld* operations take $O(m \log n)$ time.
  - like splay trees, each operation takes O(log n) amortized time.