Data Structures for Minimum Spanning Trees: Graphs & Heaps

Slides by Carl Kingsford

Jan. 17, 2014

KT 2.5,3.1

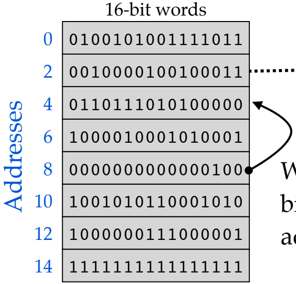
Recall Prim's Algorithm

```
# distToT[u] is distance from current tree to u
for u $\in$ V do distToT[u] $\leftarrow \infty$
u $\leftarrow$ s
while u $\neq$ null do
    # We put u in the tree, so distance is $-\infty$
    distToT[u] $\leftarrow -\infty$
    # Each of u's neighbors v are now incident to the current tree
    for v $\in$ Neighbors(u) do
        # If the distance is smaller than before, we have to update
        if d(u,v) $<$ distToT[v] then
            distToT[v] $\leftarrow$ d(u,v)
            parent[v] $\leftarrow$ u
    u $\leftarrow$ ClosestVertex(distToT)
return parent
```

RAM = Symbols + Pointers

(for our purposes)

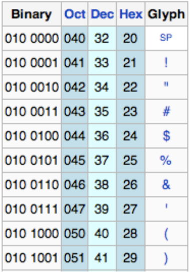
RAM is an array of bits, broken up into words, where each word has an address.



Addresses	16-bit words
0	0100101001111011
2	0010000100100011
4	0110111010100000
6	1000010001010001
8	0000000000000100
10	1001010110001010
12	1000000111000001
14	1111111111111111

We may interpret bits as a memory address (pointer)

We may interpret the bits as numbers, or letters, or other symbols

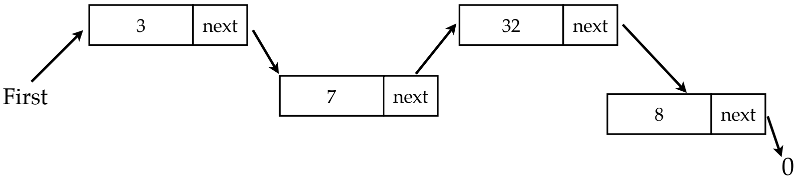


Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	SP
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)

We can store and manipulate arbitrary symbols (like letters) and associations between them.

Storing a list

Store a list of numbers such as 3, 7, 32, 8:



- Records located anywhere in memory
- Don't have to know the size of data at the start
- Pointers let us express relationships between pieces of information

What is a data structure anyway?

It's an agreement about:

- how to store a collection of objects in memory,
- what operations we can perform on that data,
- the algorithms for those operations, and
- how time and space efficient those algorithms are.

Data structures → Data structurING:

How do we organize information in memory so that we can find, update, add, and delete portions of it efficiently?

Often, the key to a fast algorithm is an efficient data structure.

Abstract data types (ADT)

ADT specifies permitted operations as well as time and space guarantees.

Example Graph ADT (without time/space guarantees):

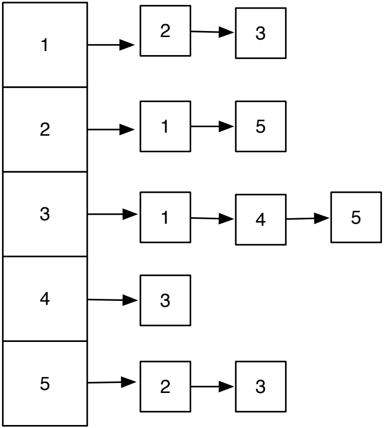
- G.add_vertex(u) — adds a vertex to graph G.
- G.add_edge(u, v, d) — adds an edge of weight d between vertices u and v.
- G.has_edge(u, v) — returns True iff edge {u,v} exists in G.
- G.neighbors(u) — gives a list of vertices adjacent to u in G.
- G.weight(u,v) — gives the weight of edge u and v

Representing Graphs

Adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency list:



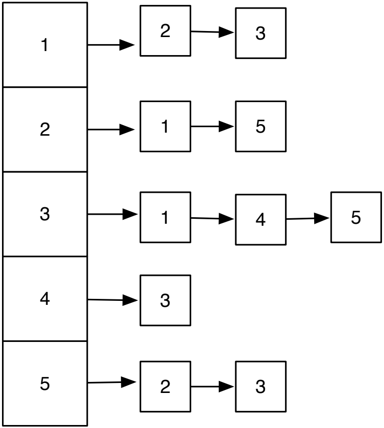
Representing Graphs

Adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

How long does G.neighbors(u) take?

Adjacency list:



Representing Graphs

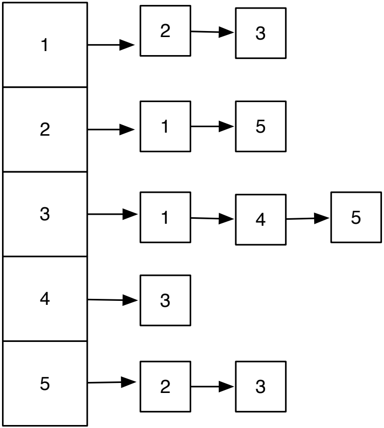
Adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

How long does G.neighbors(u) take?

How would you implement G.weight(u,v)?

Adjacency list:



Implementing ClosestVertex:

Priority Queue ADT & d-Heaps

Priority Queue ADT

A *priority queue*, also called a *heap*, holds items i that have keys $key(i)$.

They support the following operations:

- *H.insert(i)* — put item i into heap H
- *H.deletemin()* — return item with smallest key in H and delete it from H
- *H.makeheap(S)* — create a heap from a set S of items
- *H.findmin()* — return item with smallest key in H
- *H.delete(i)* — remove item i from H

ClosestVertex via Heaps

Items: vertices that are on the “frontier” (incident to the current tree)

Key(v) = distToT[v]

ClosestVertex: return H.deletemin()

How can we efficiently implement the heap ADT?

Priority Queue ADT

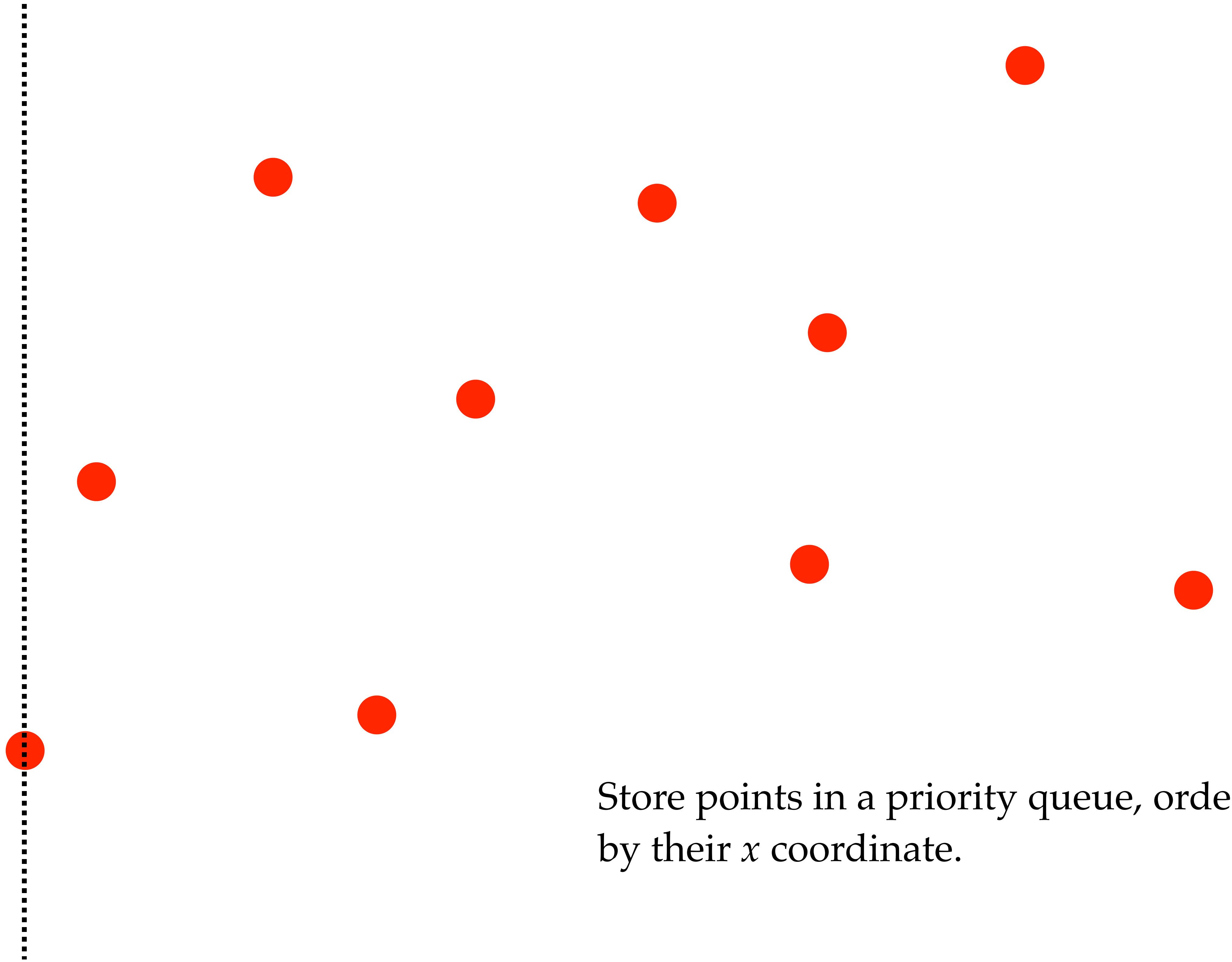
- Efficiently support the following operations on a set of **keys**:
  - *findmin*: return the smallest key
  - *deletemin*: return the smallest key & delete it
  - *insert*: add a new key to the set
  - *delete*: delete an arbitrary key
- Would like to be able to do *findmin* faster (say in time independent of the # of items in the set).

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
```

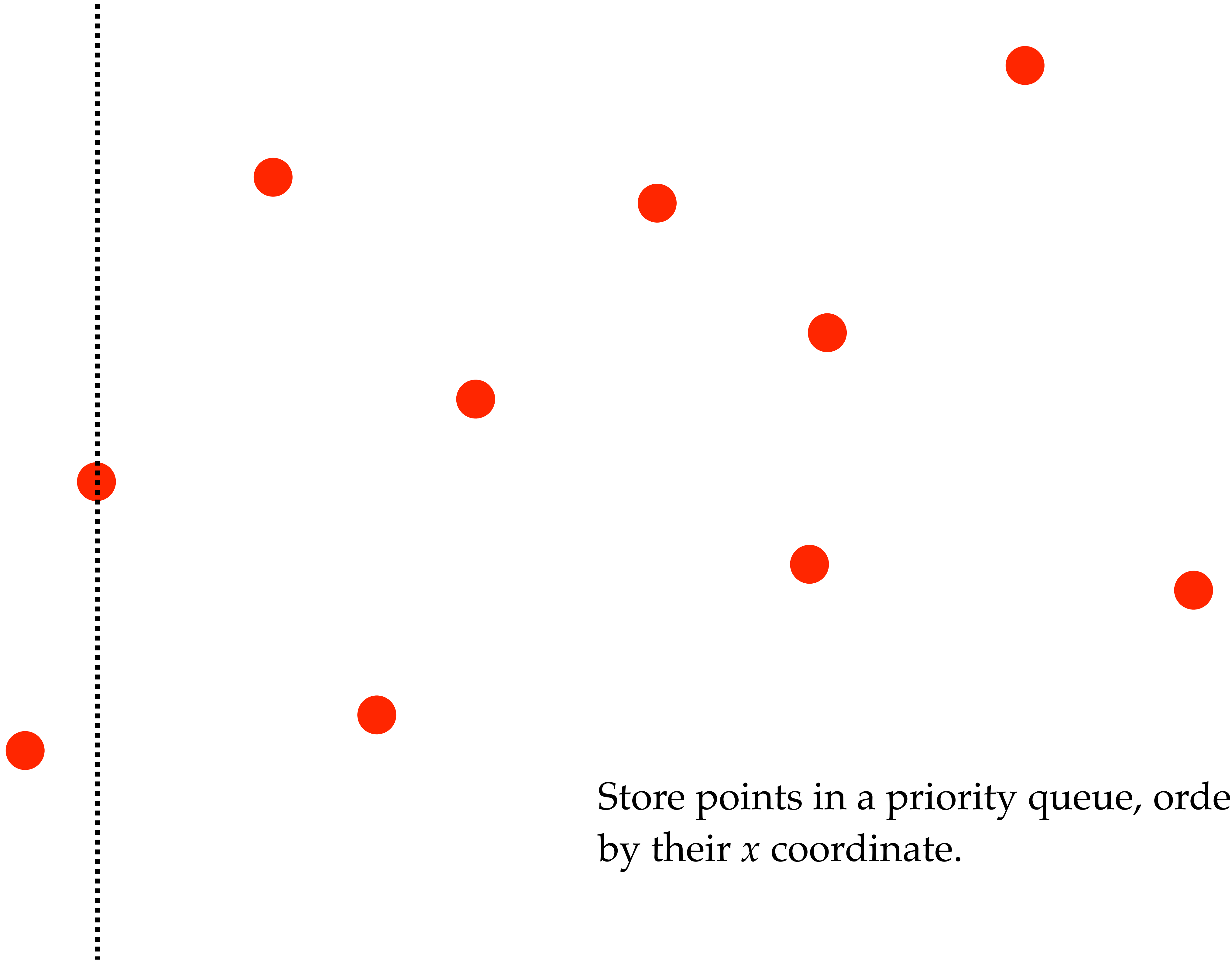
When scheduler asks “What should I run next?” it could *findmin(H)*.

Plane Sweep: Process points left to right:



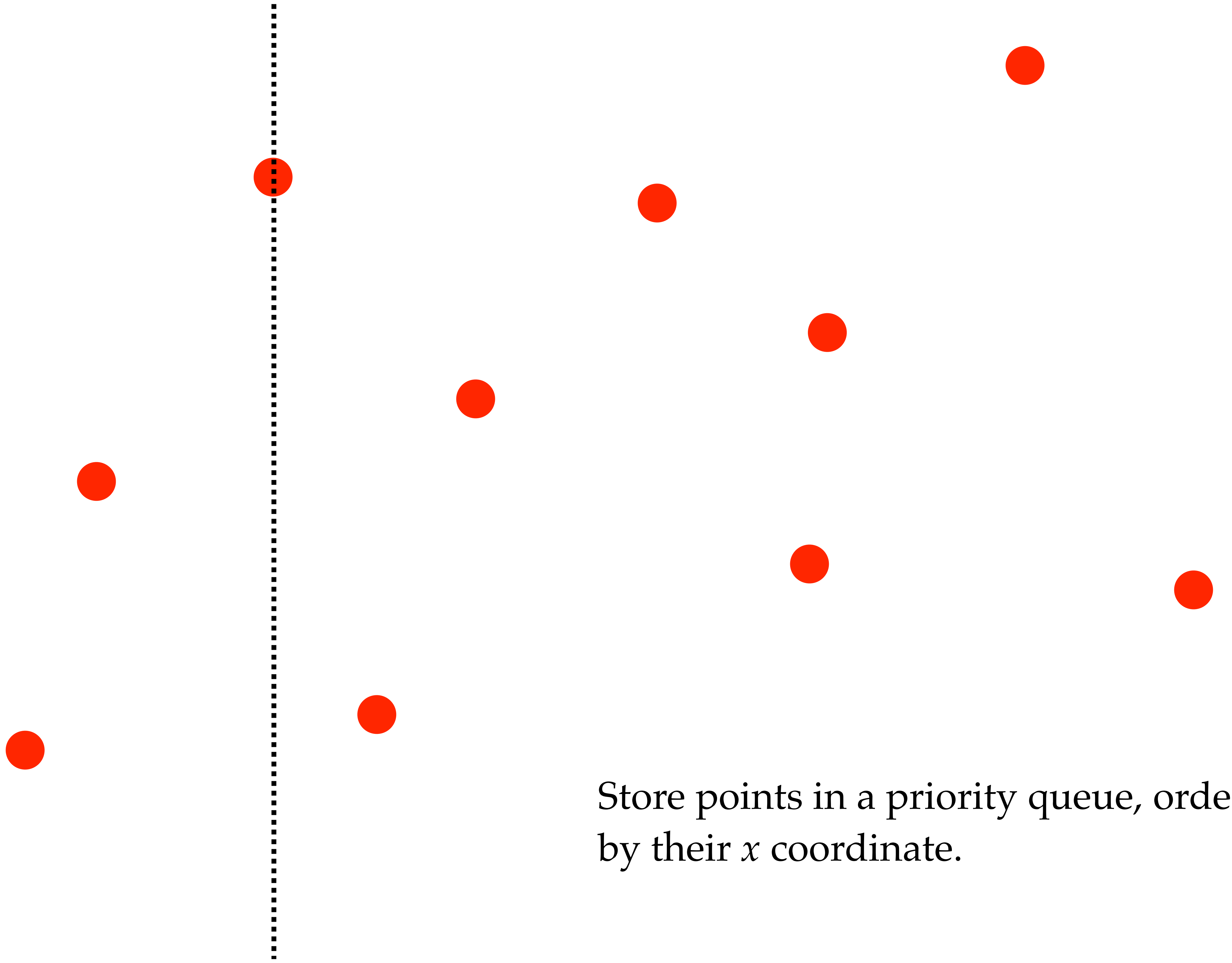
Store points in a priority queue, ordered by their x coordinate.

Plane Sweep: Process points left to right:



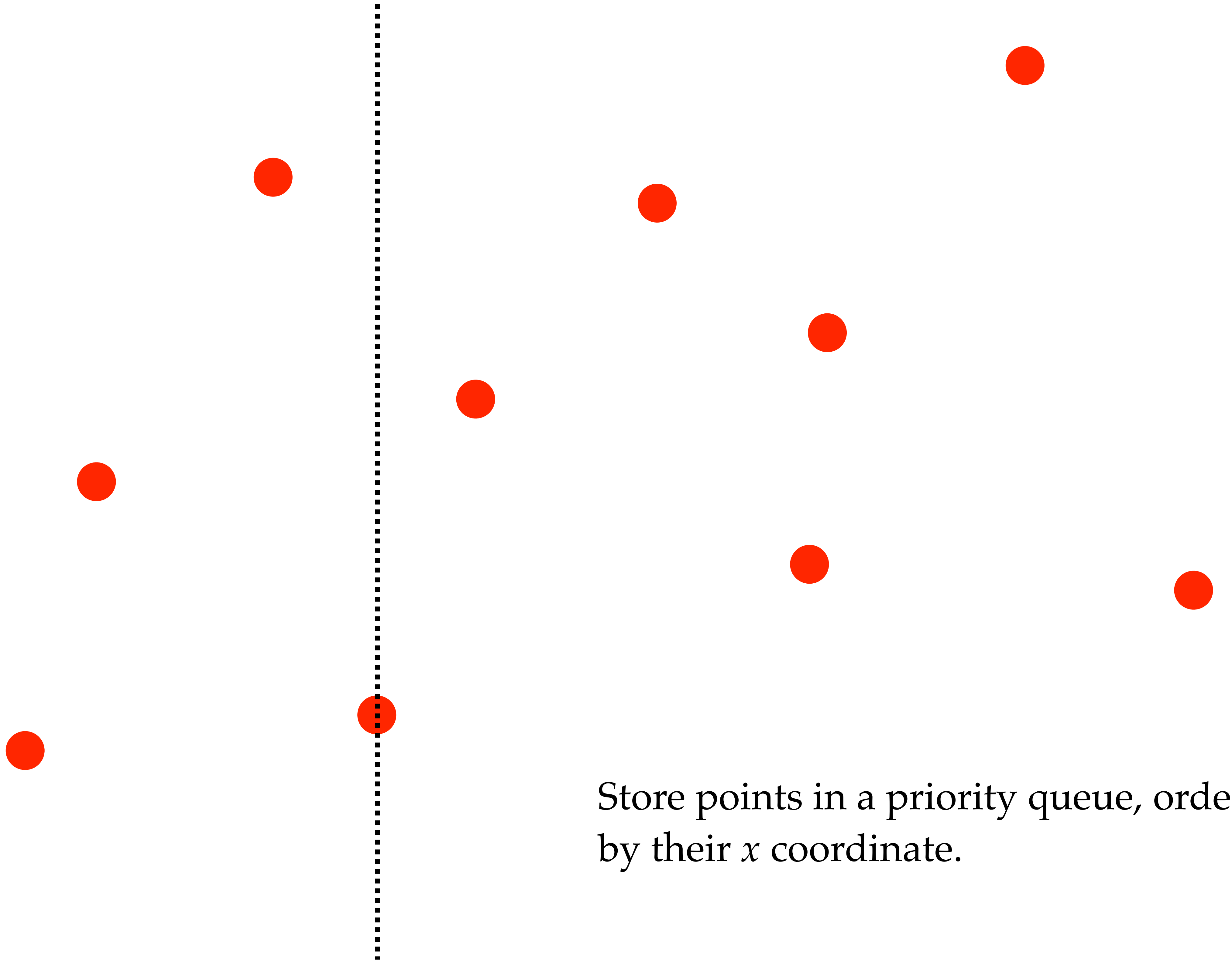
Store points in a priority queue, ordered by their *x* coordinate.

Plane Sweep: Process points left to right:



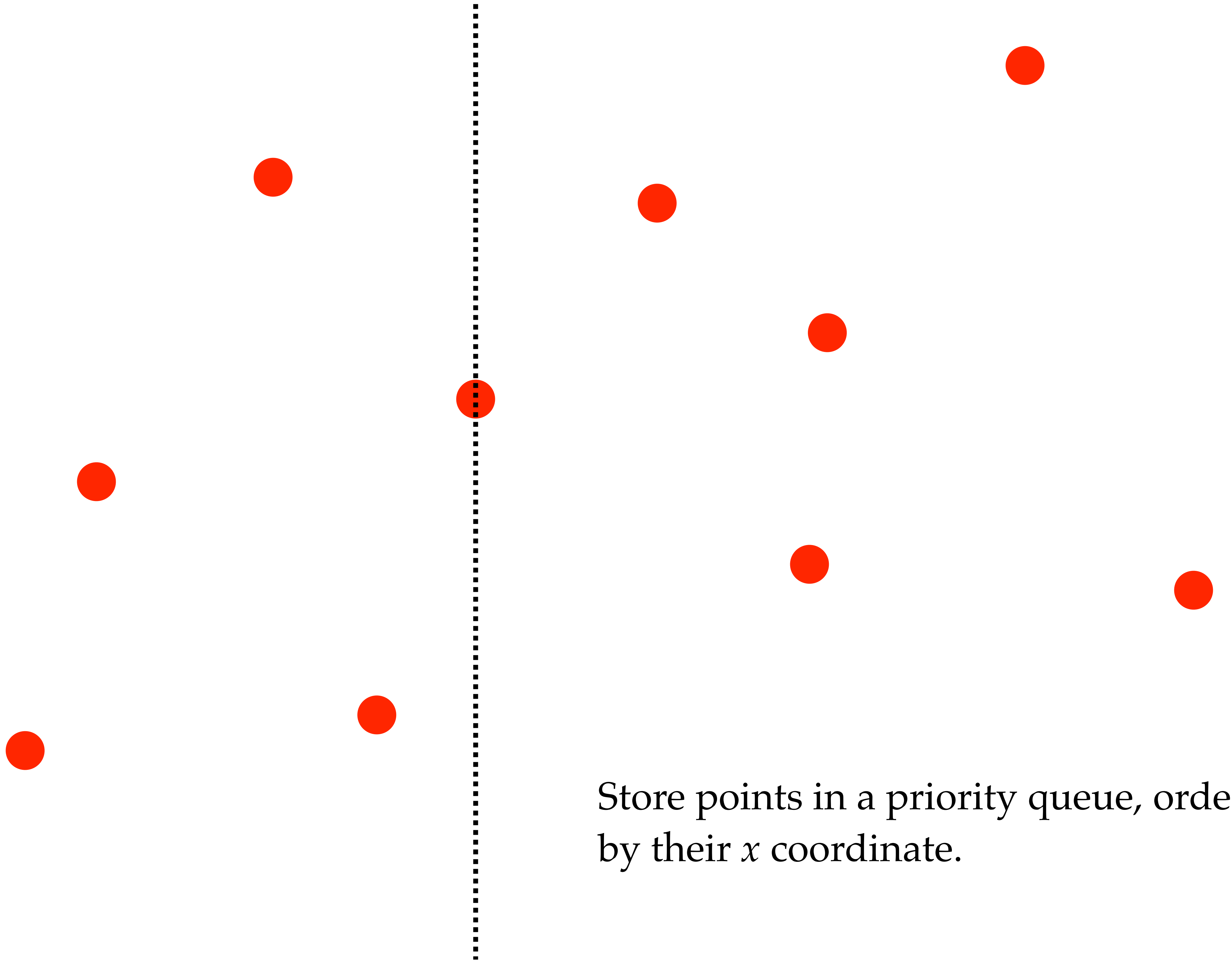
Store points in a priority queue, ordered by their x coordinate.

Plane Sweep: Process points left to right:



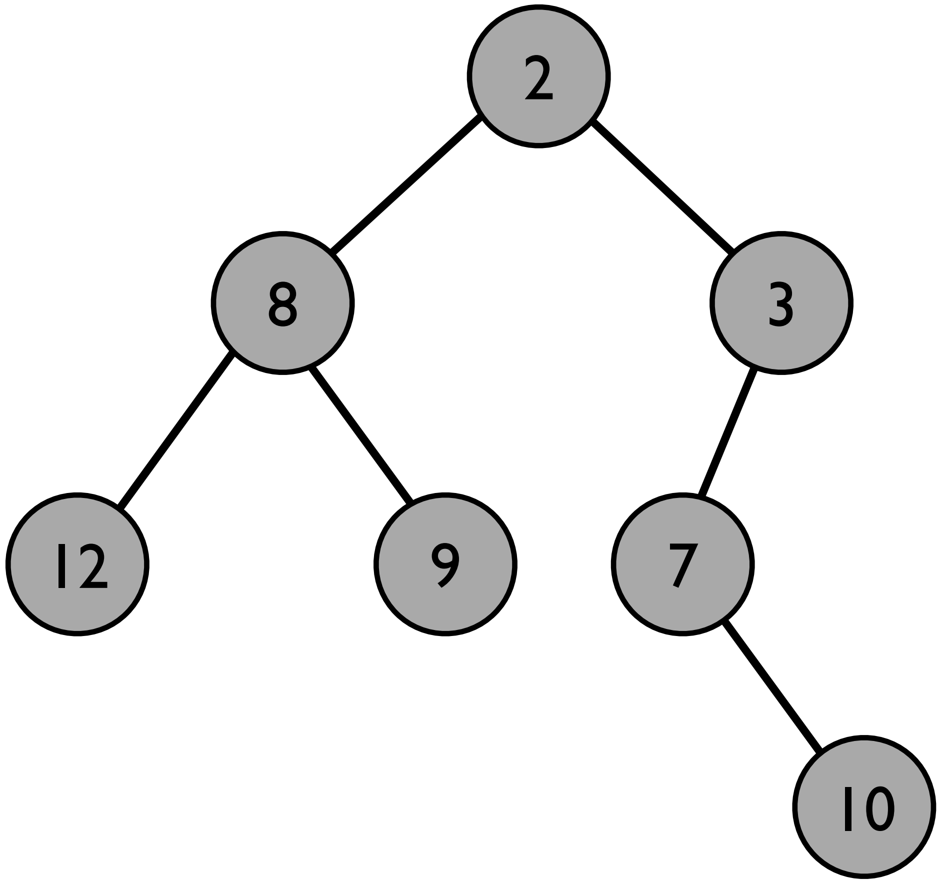
Store points in a priority queue, ordered by their x coordinate.

Plane Sweep: Process points left to right:



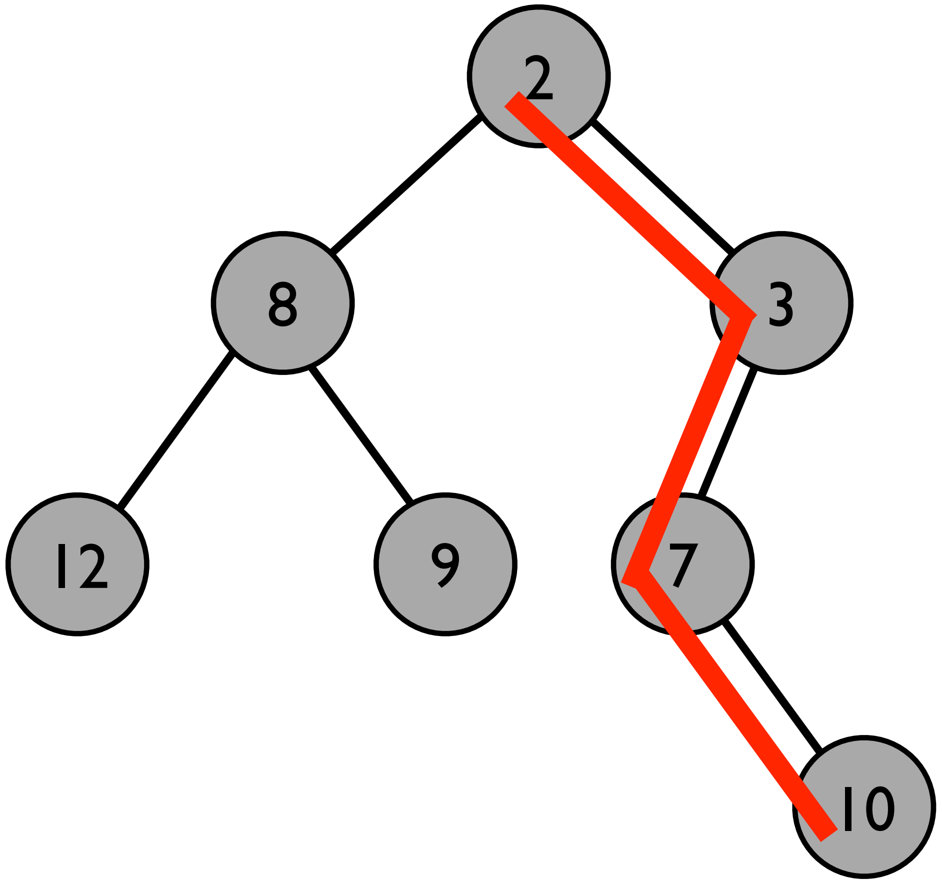
Store points in a priority queue, ordered by their x coordinate.

Heap-Ordered Trees



- 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.)

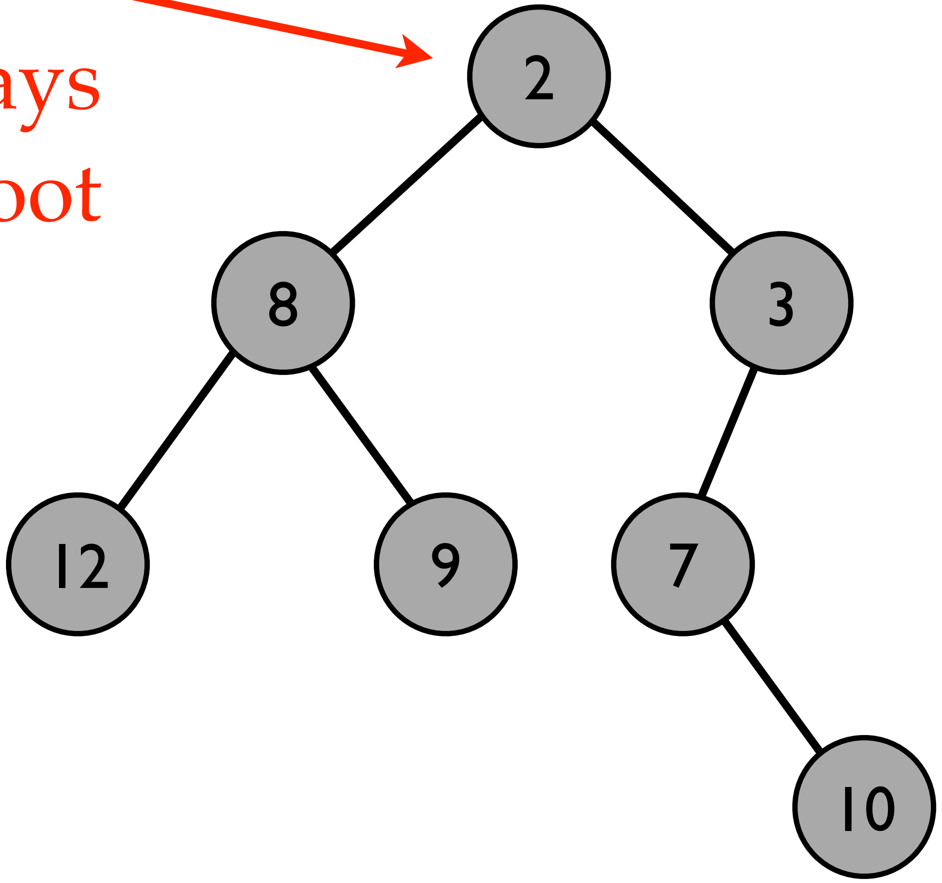
Heap-Ordered Trees



Along each path keys are monotonically non-decreasing

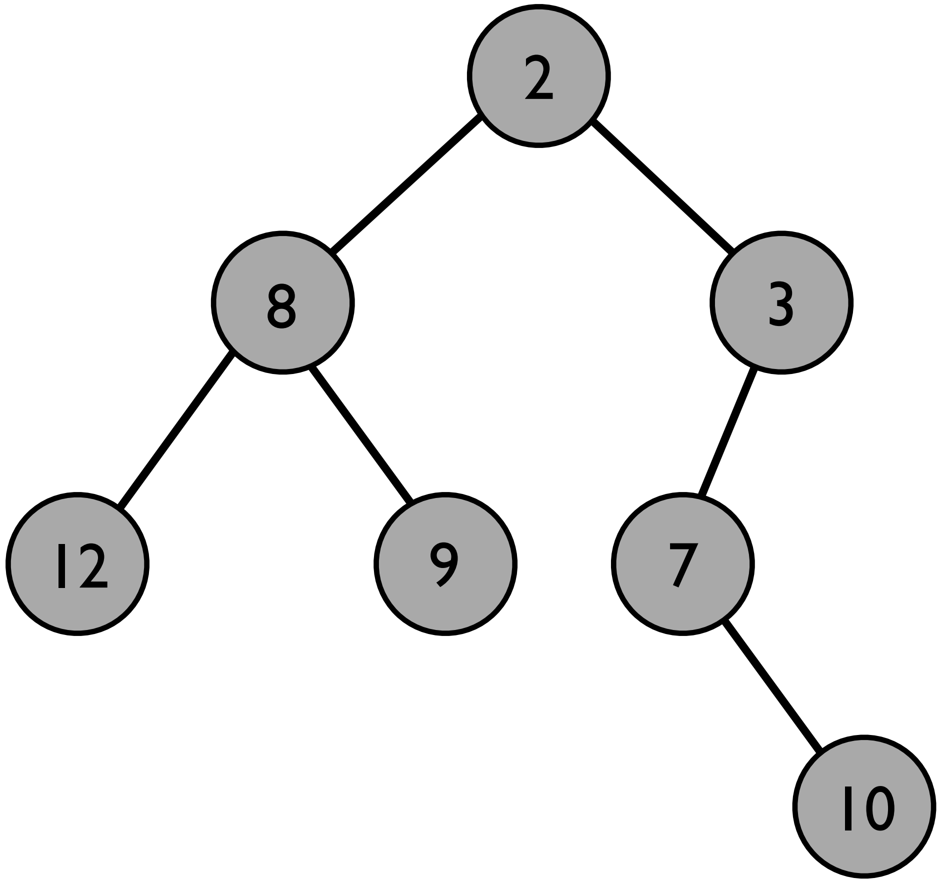
- The keys of the children of *u* are ≥ the key(*u*), for all nodes *u*.
- (This "heap" has nothing to do with the "heap" part of computer memory.)

Heap – Find min



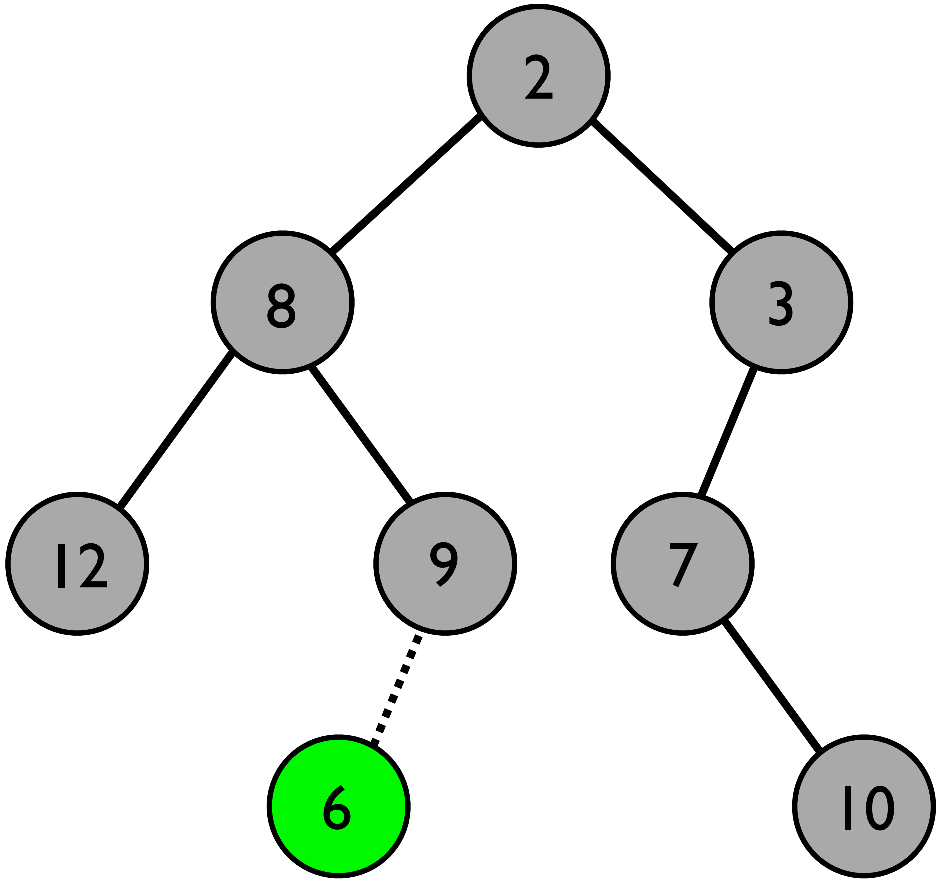
The minimum element is always the root

Heap – Insert



Heap – Insert

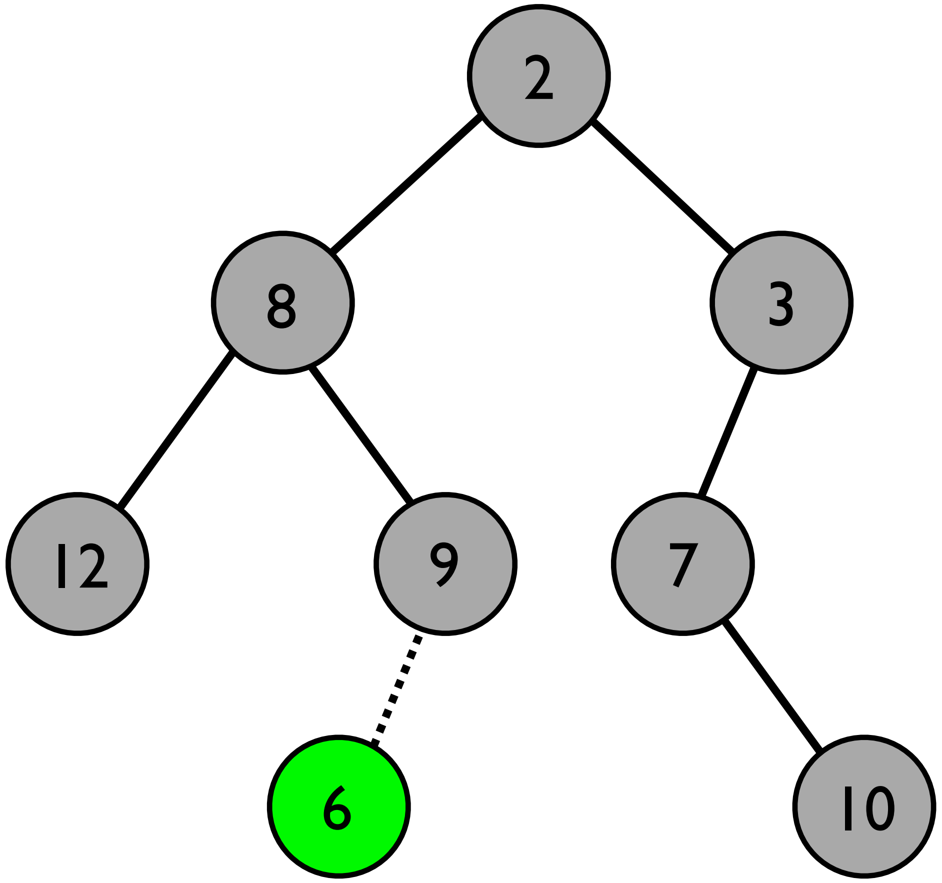
1. Add node as a leaf
(we’ll see where later)



Heap – Insert

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

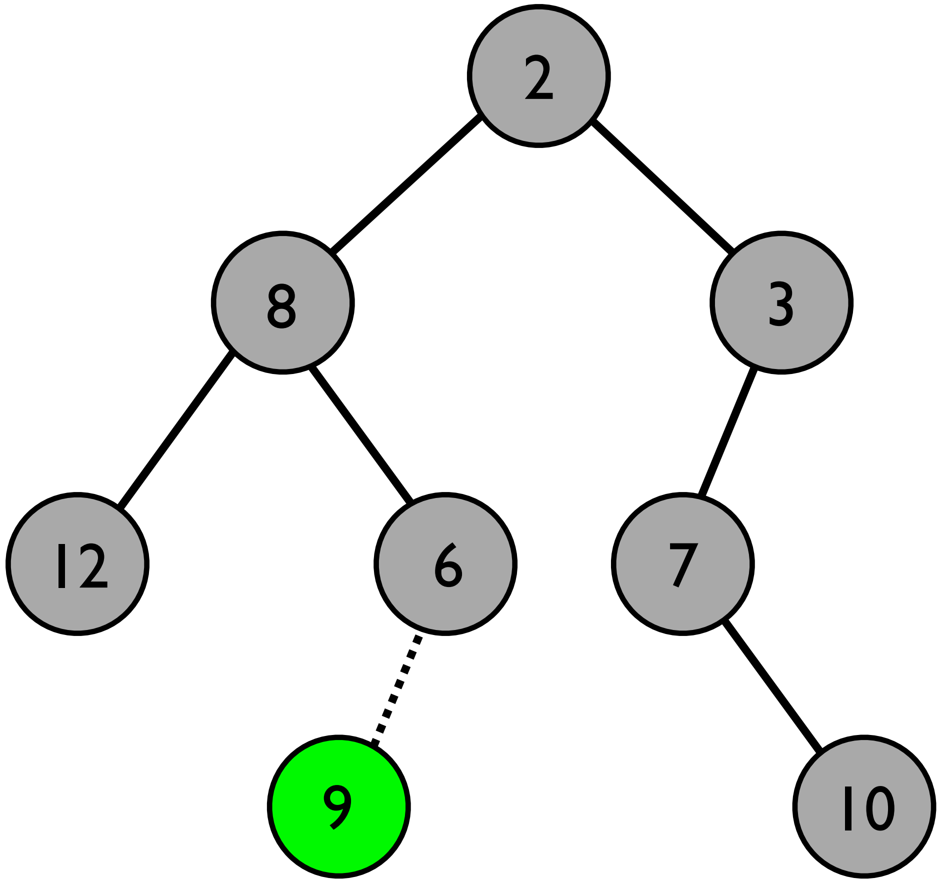
2. *"sift up:"* while current node is < its parent, swap them.



Heap – Insert

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

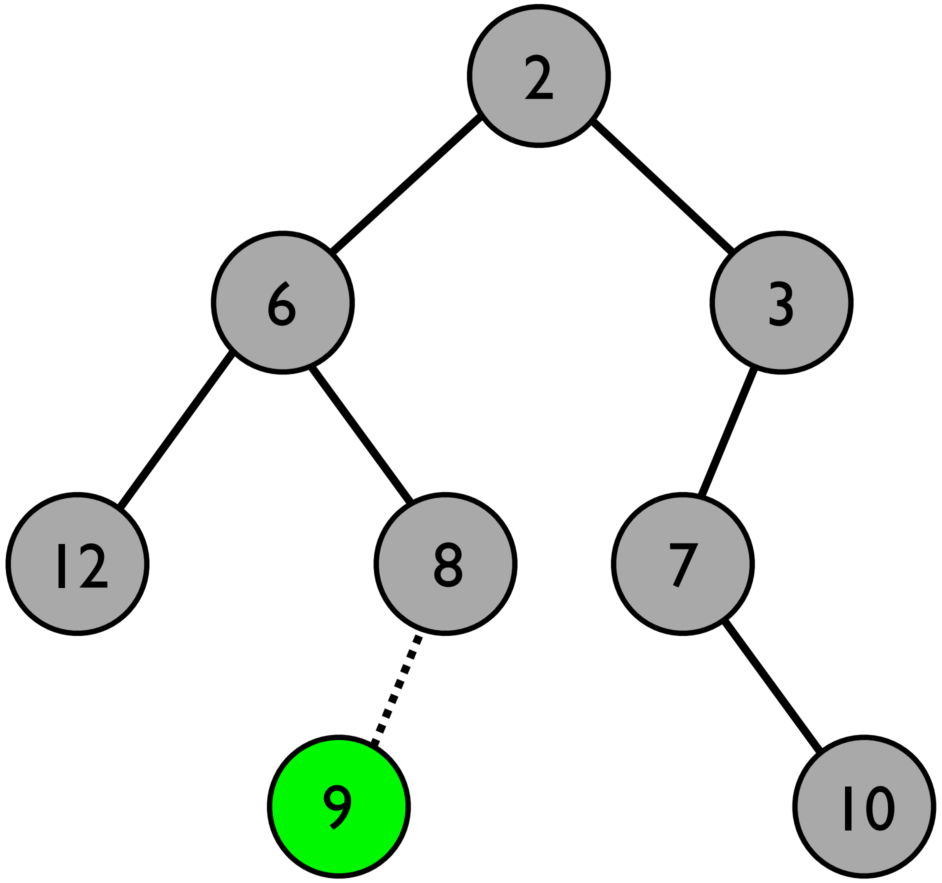
2. “*sift up:*” while current node is < its parent, swap them.



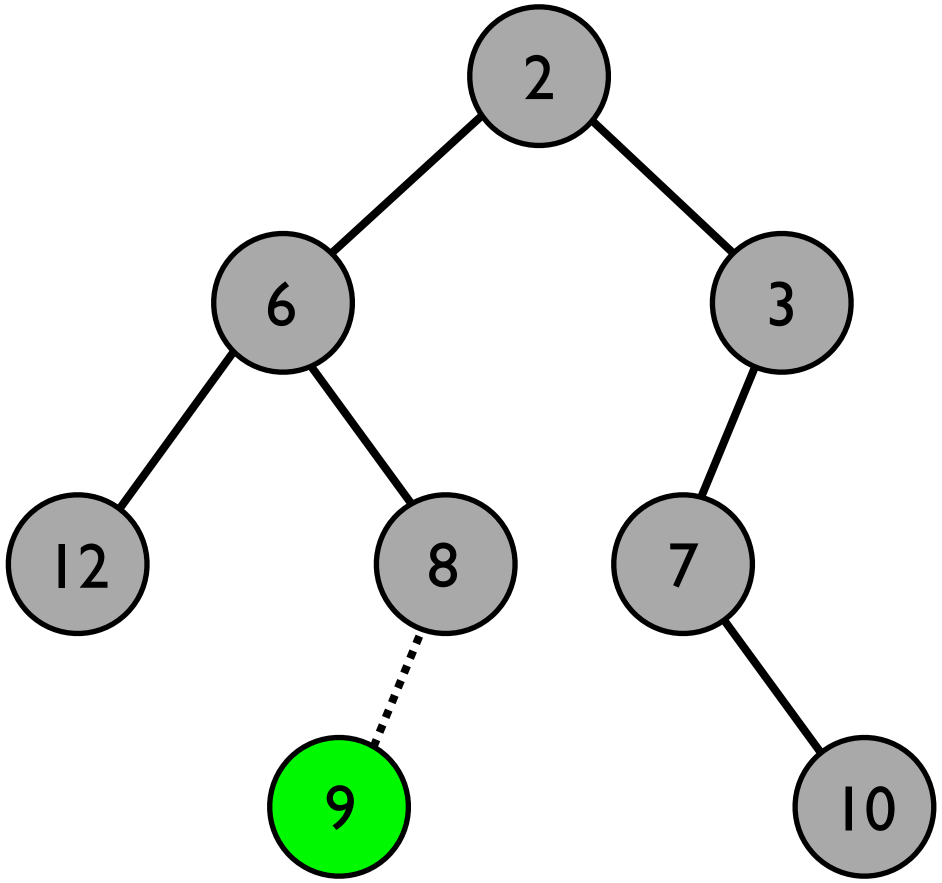
Heap – Insert

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

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

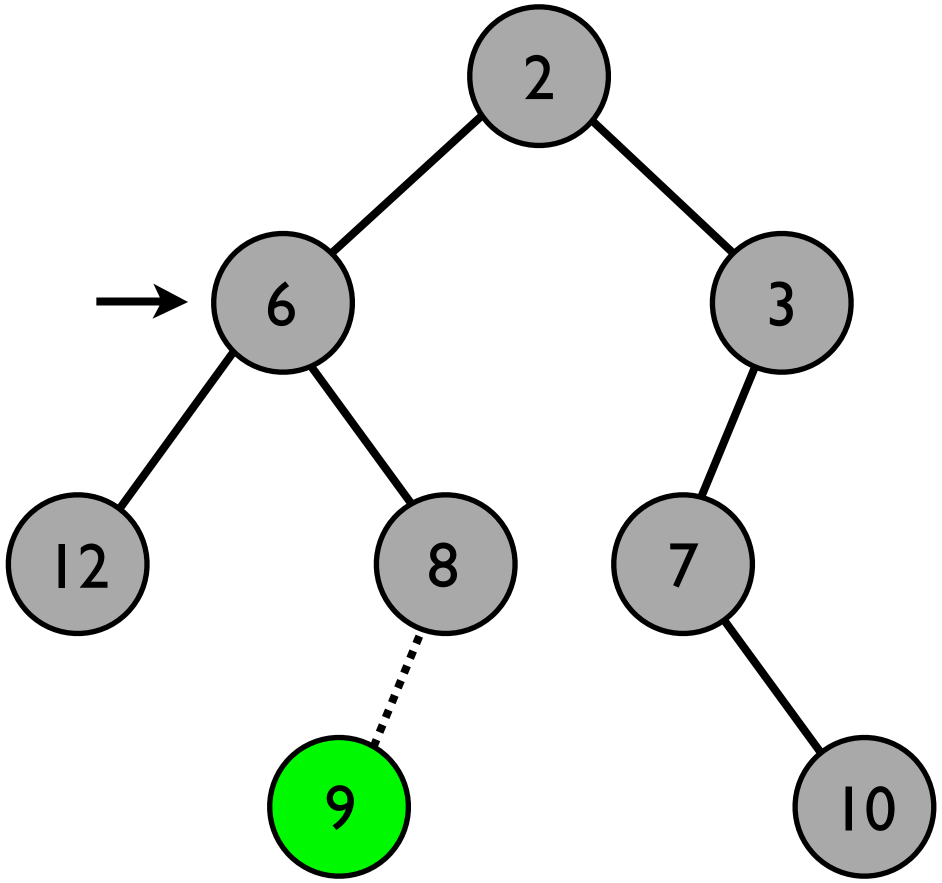


Heap – Delete(*i*)



Heap – Delete(*i*)

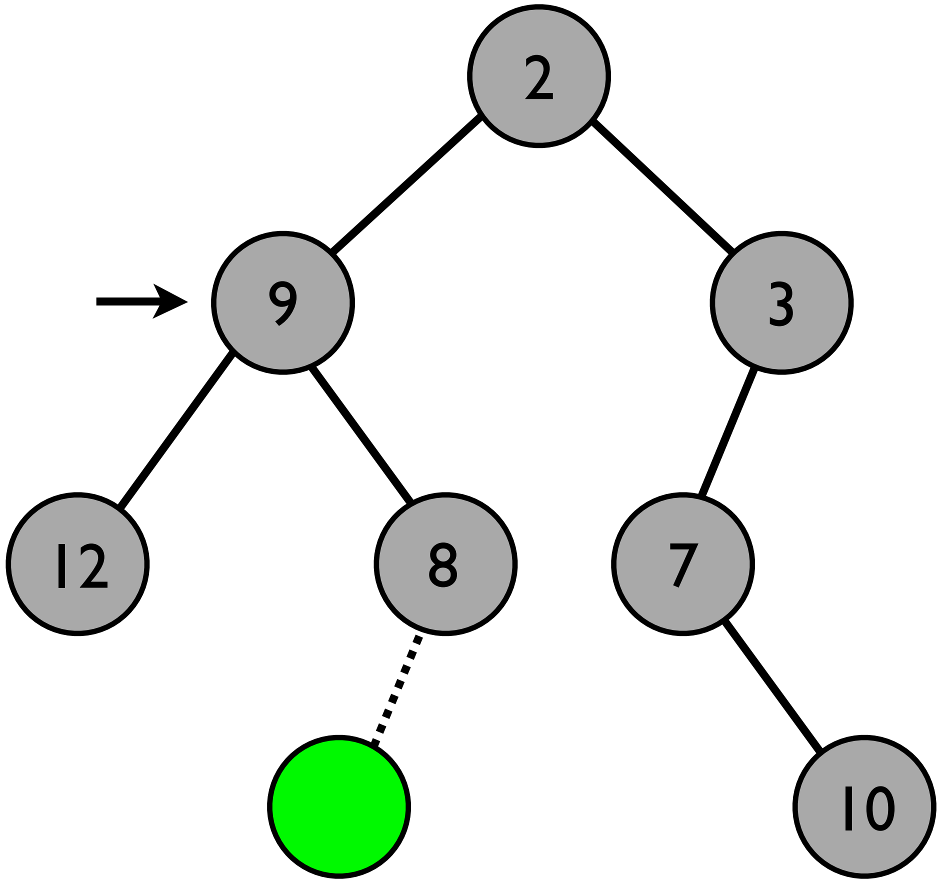
1. need a pointer to node containing key *i*



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)

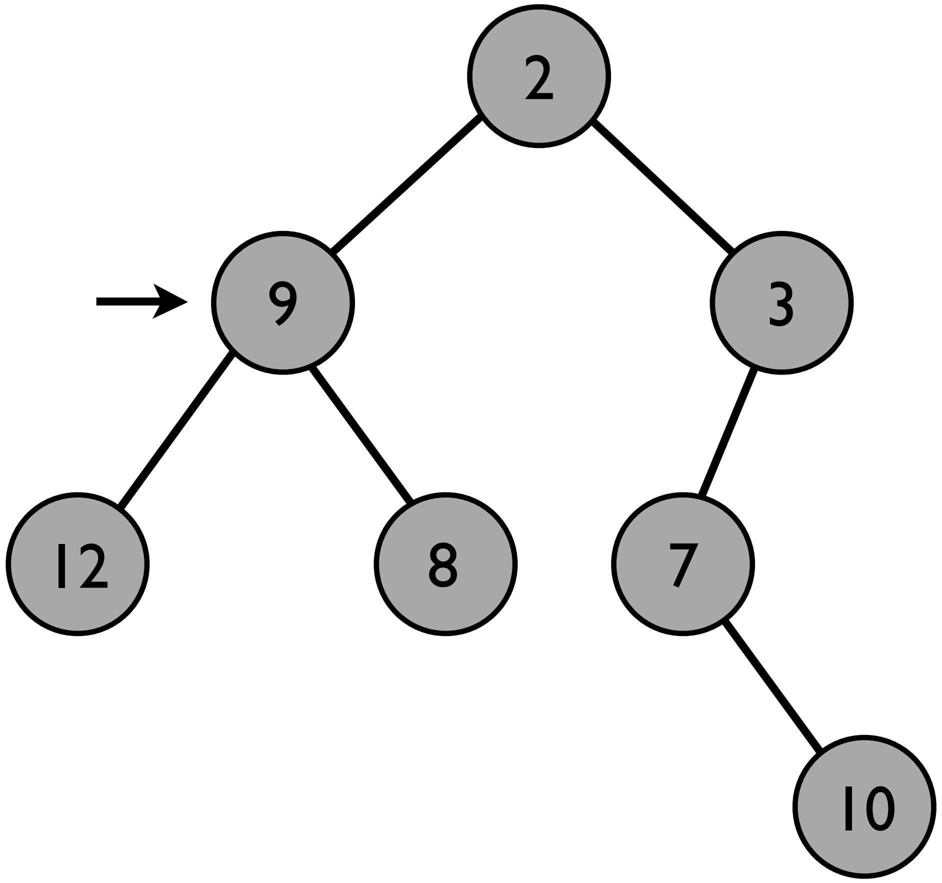


Heap – Delete(*i*)

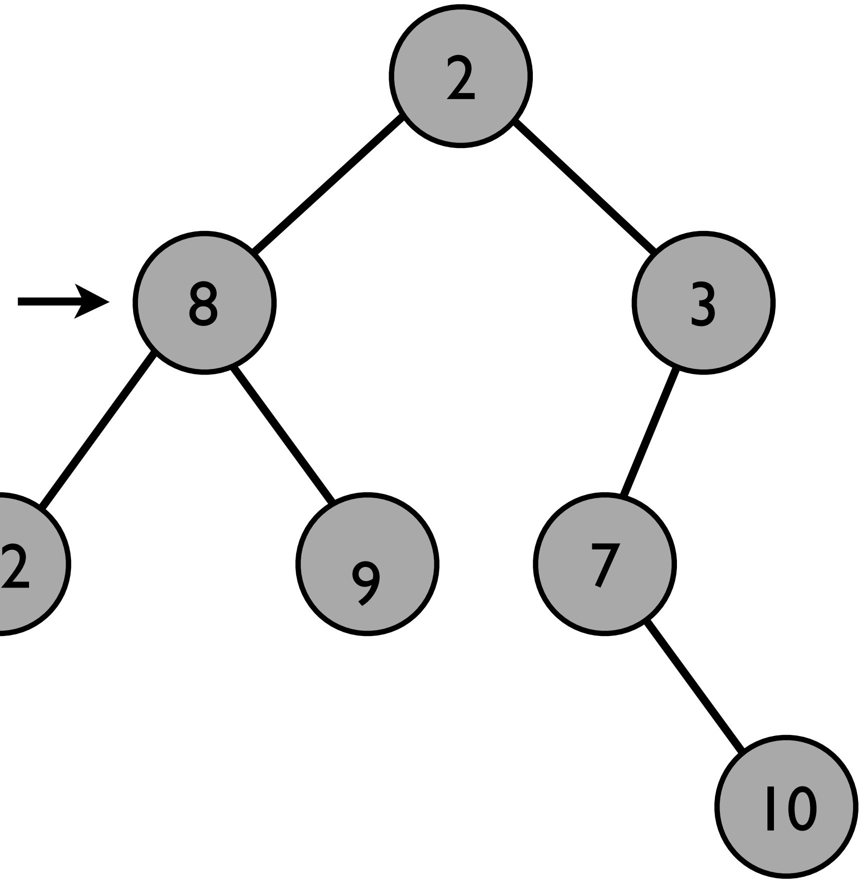
1. need a pointer to node containing key *i*

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



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

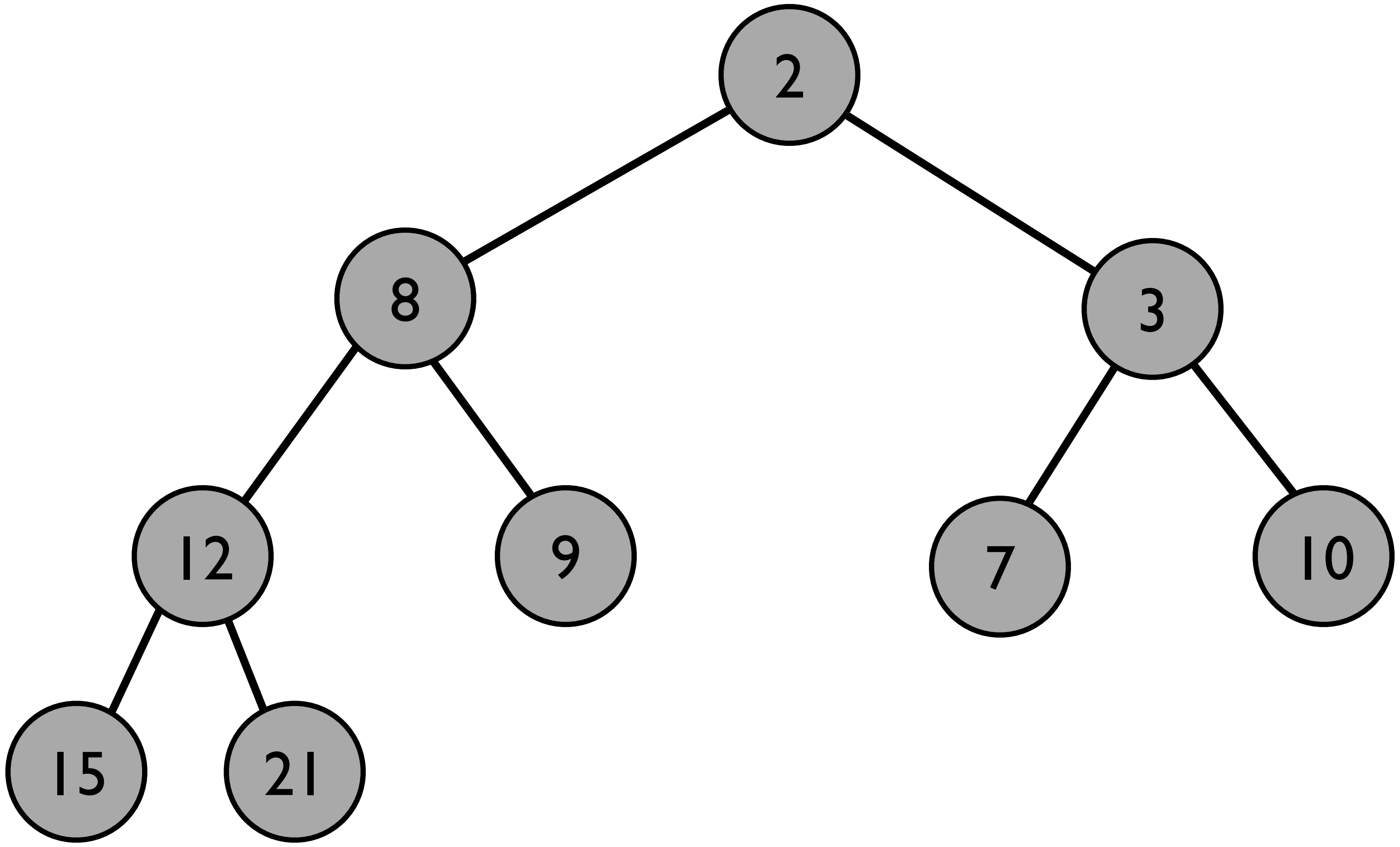
4. If *i* > *j* then sift up, moving *j* up the tree.

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

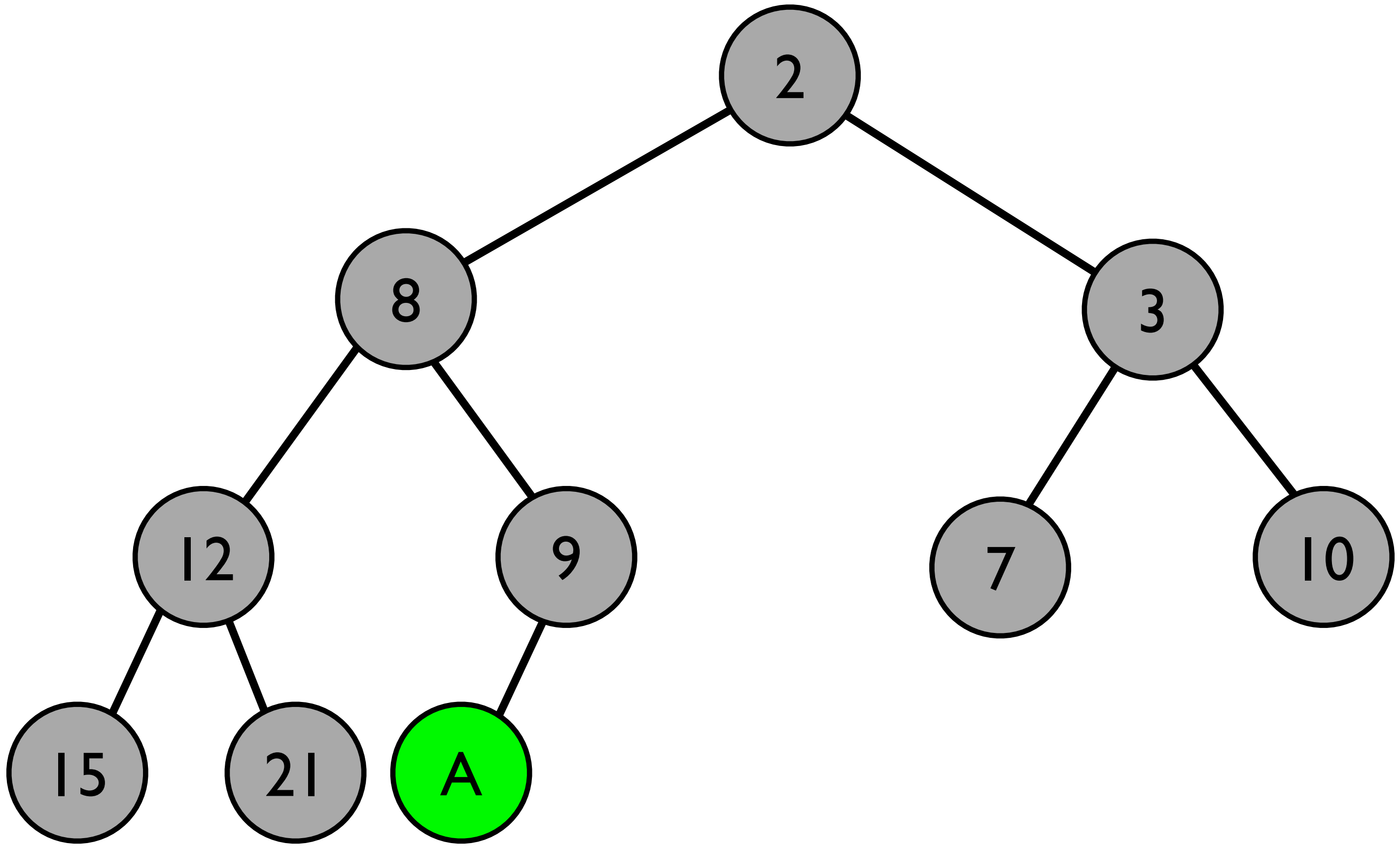
Running Times

- *findmin* takes constant time [O(1)]
- *insert*, *delete* take time $\propto$ tree height plus the time to find the leaves.
- *deletemin*: same as delete
- Q1: How do we find leaves used in *insert* and *delete*?
  - *delete*: use the last inserted node.
  - *insert*: choose node so tree remains complete.
- Q2: How do we ensure the tree has low height?

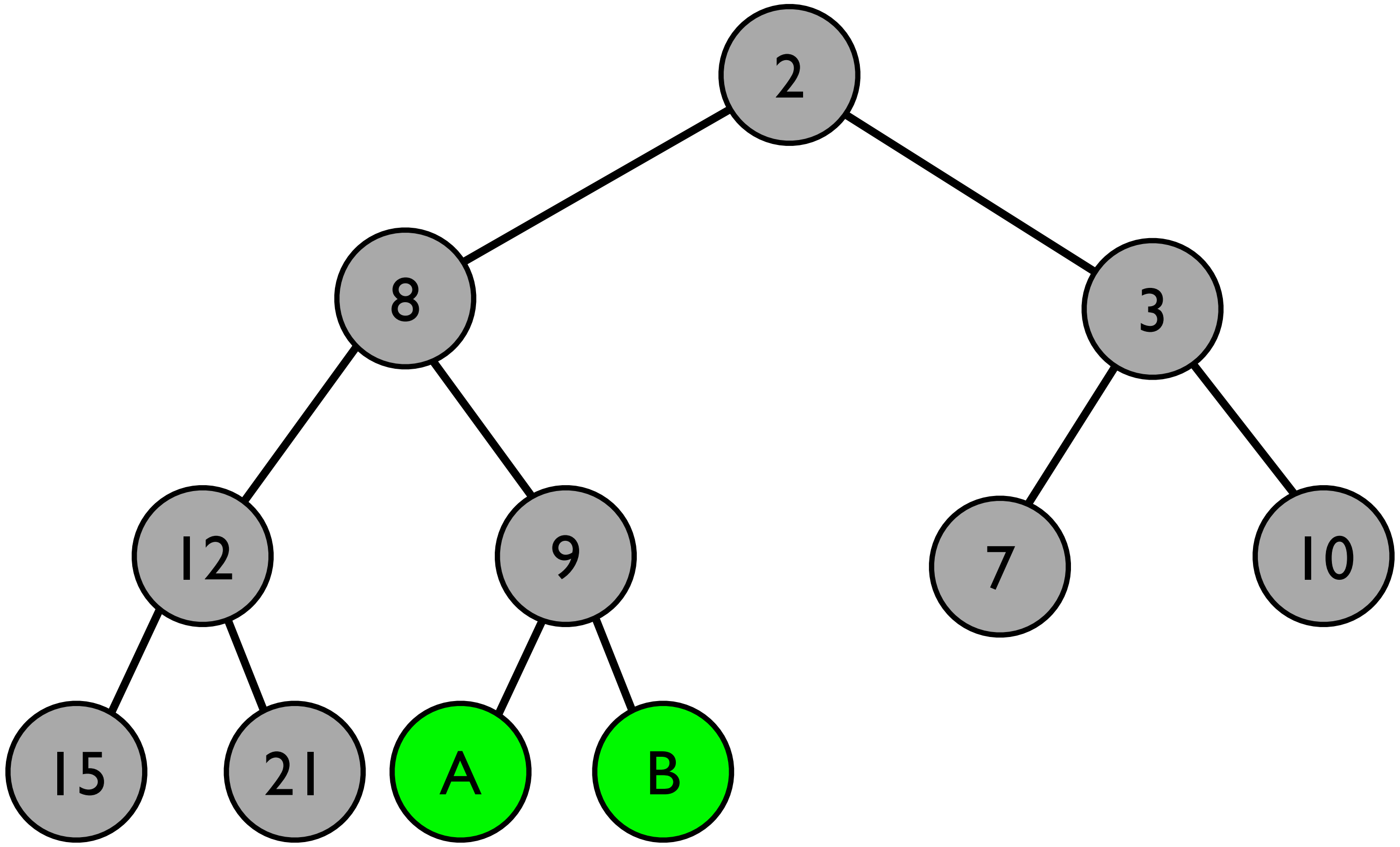
Store Heap in a Complete Tree



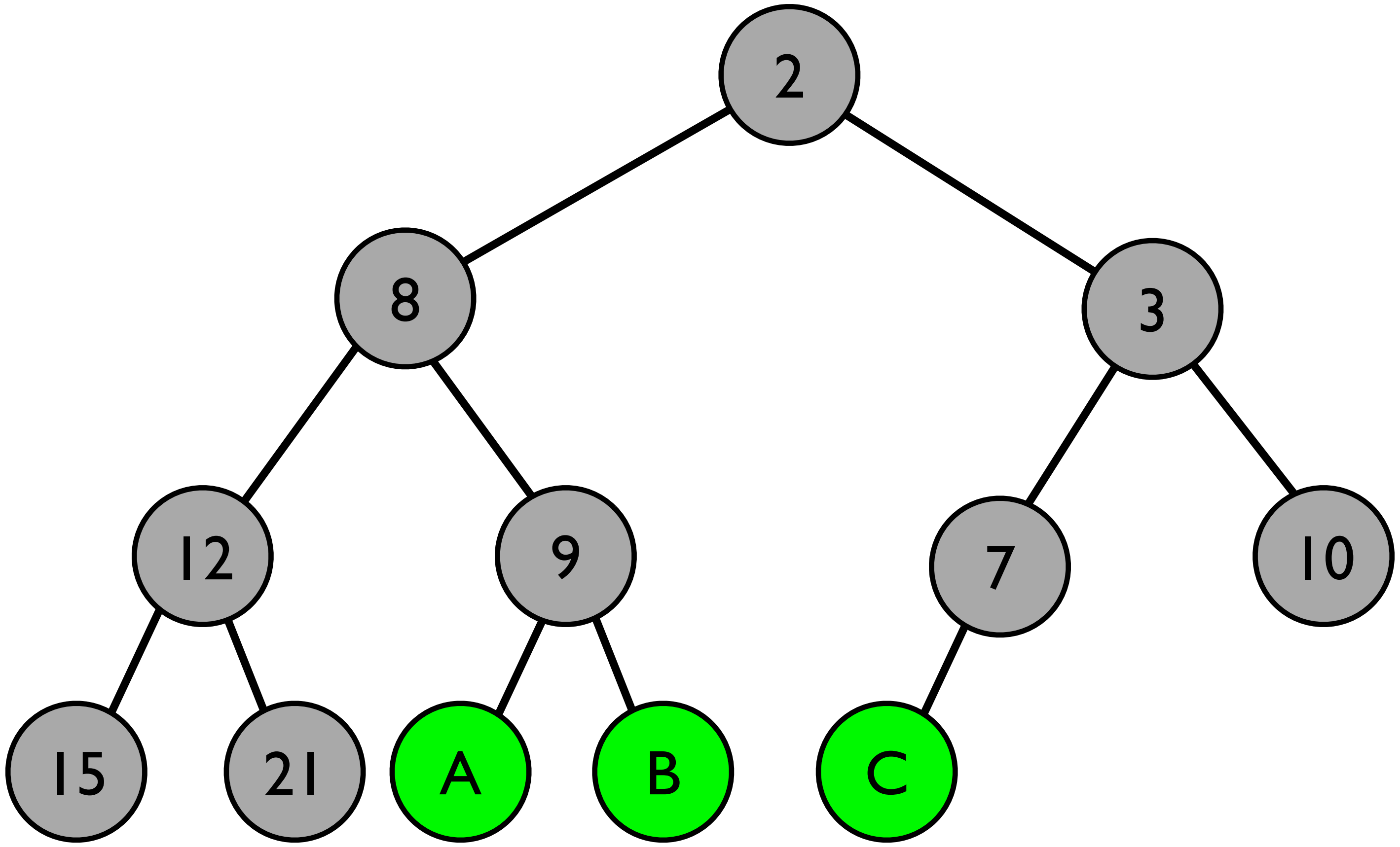
Store Heap in a Complete Tree



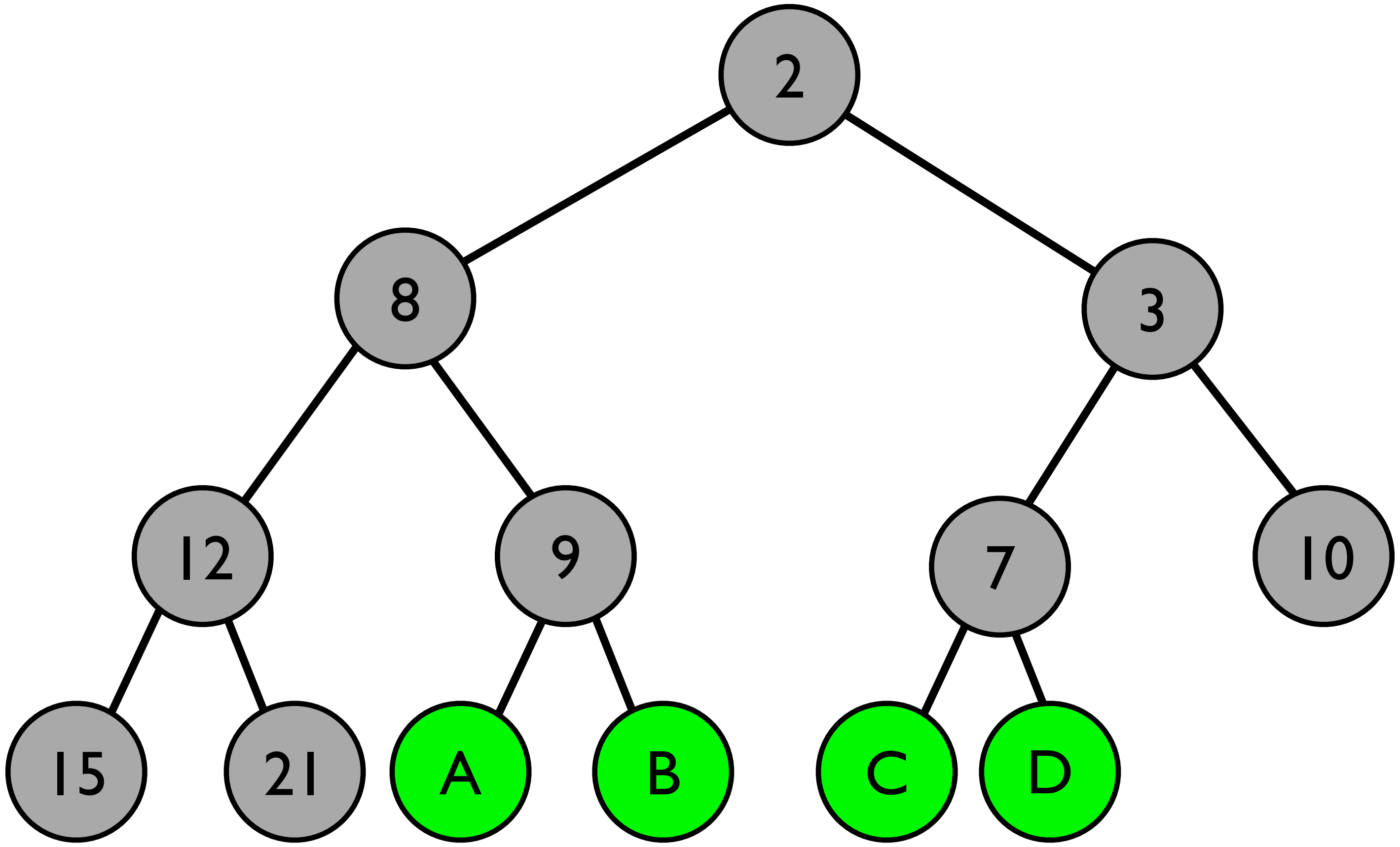
Store Heap in a Complete Tree



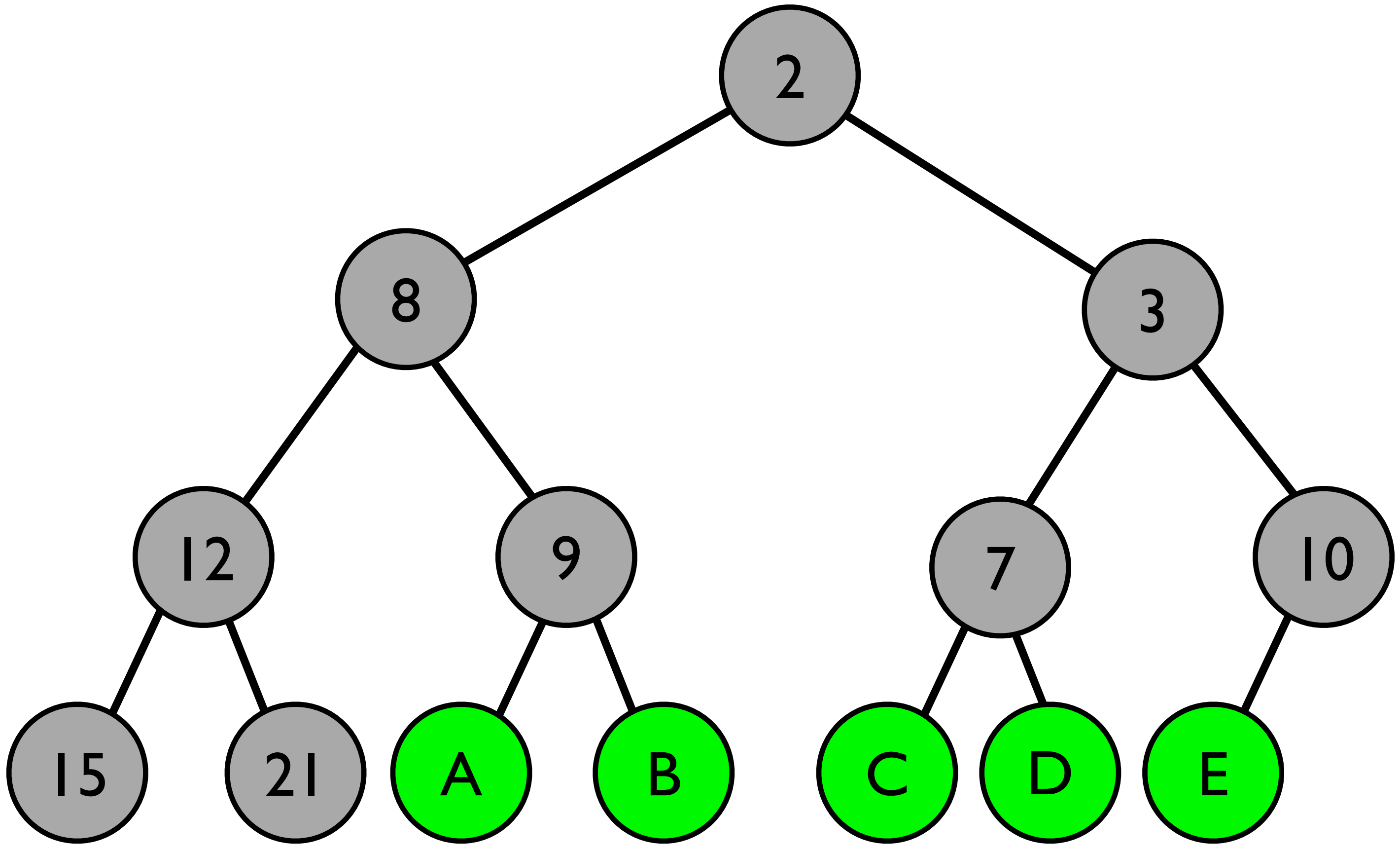
Store Heap in a Complete Tree



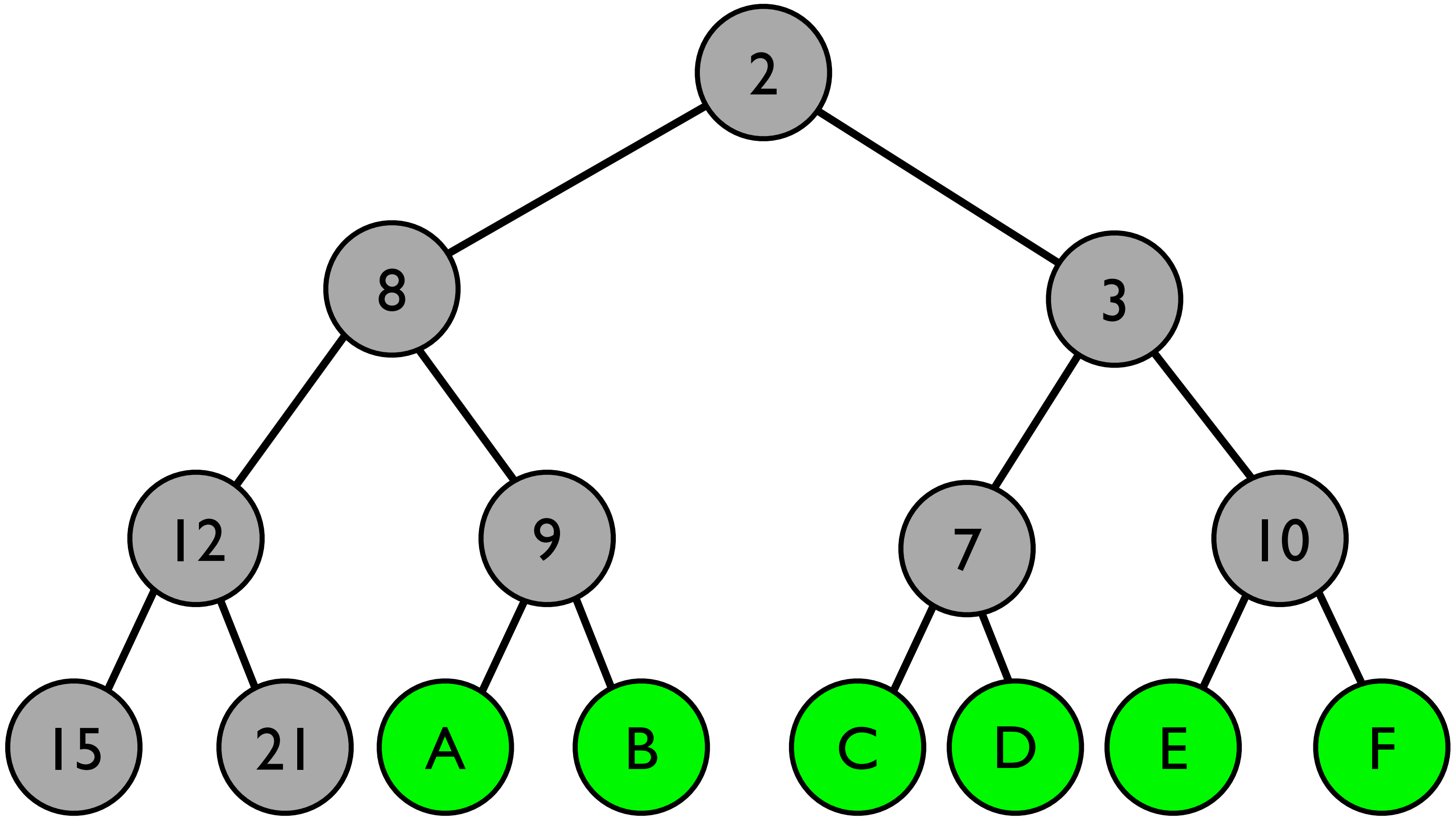
Store Heap in a Complete Tree



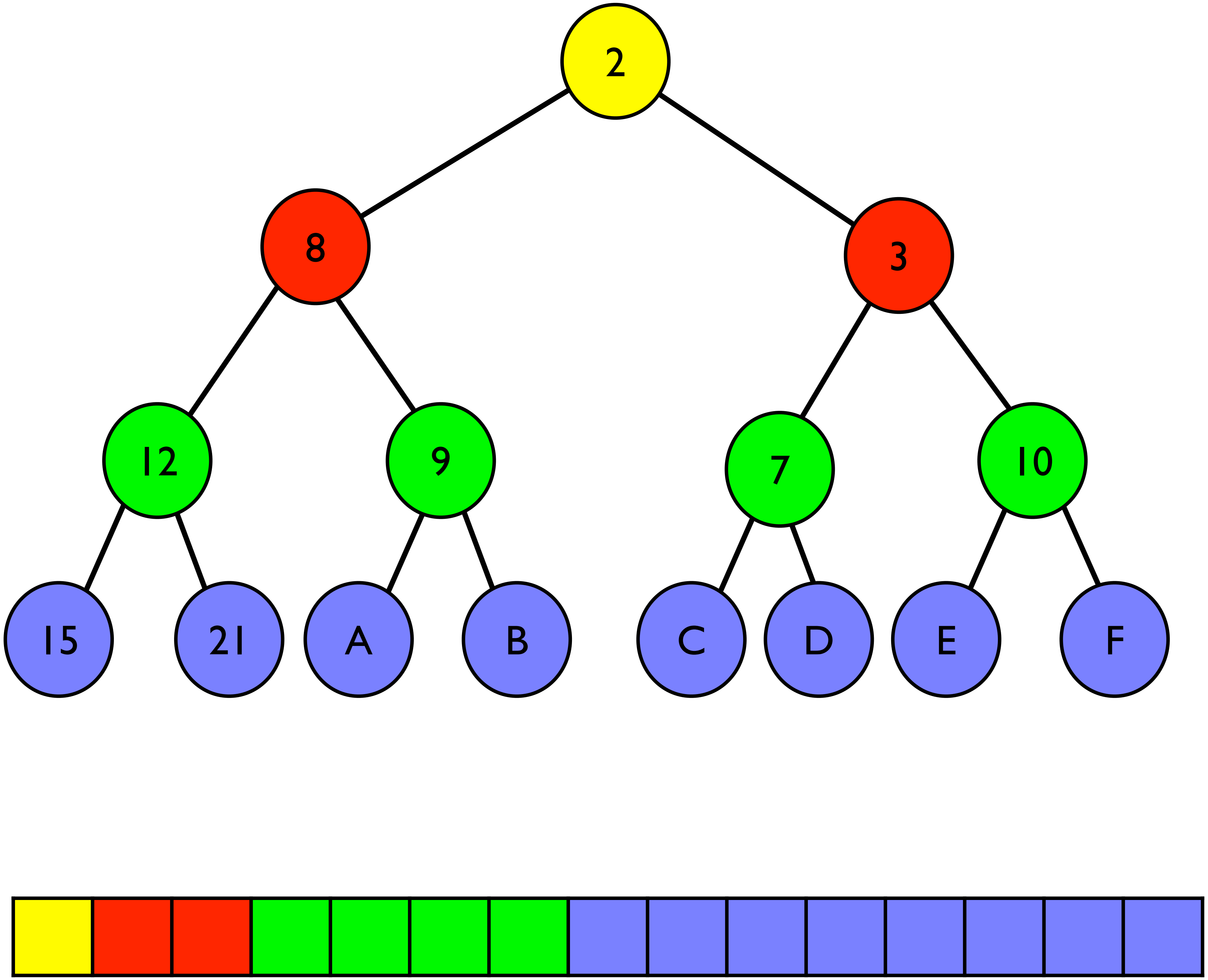
Store Heap in a Complete Tree



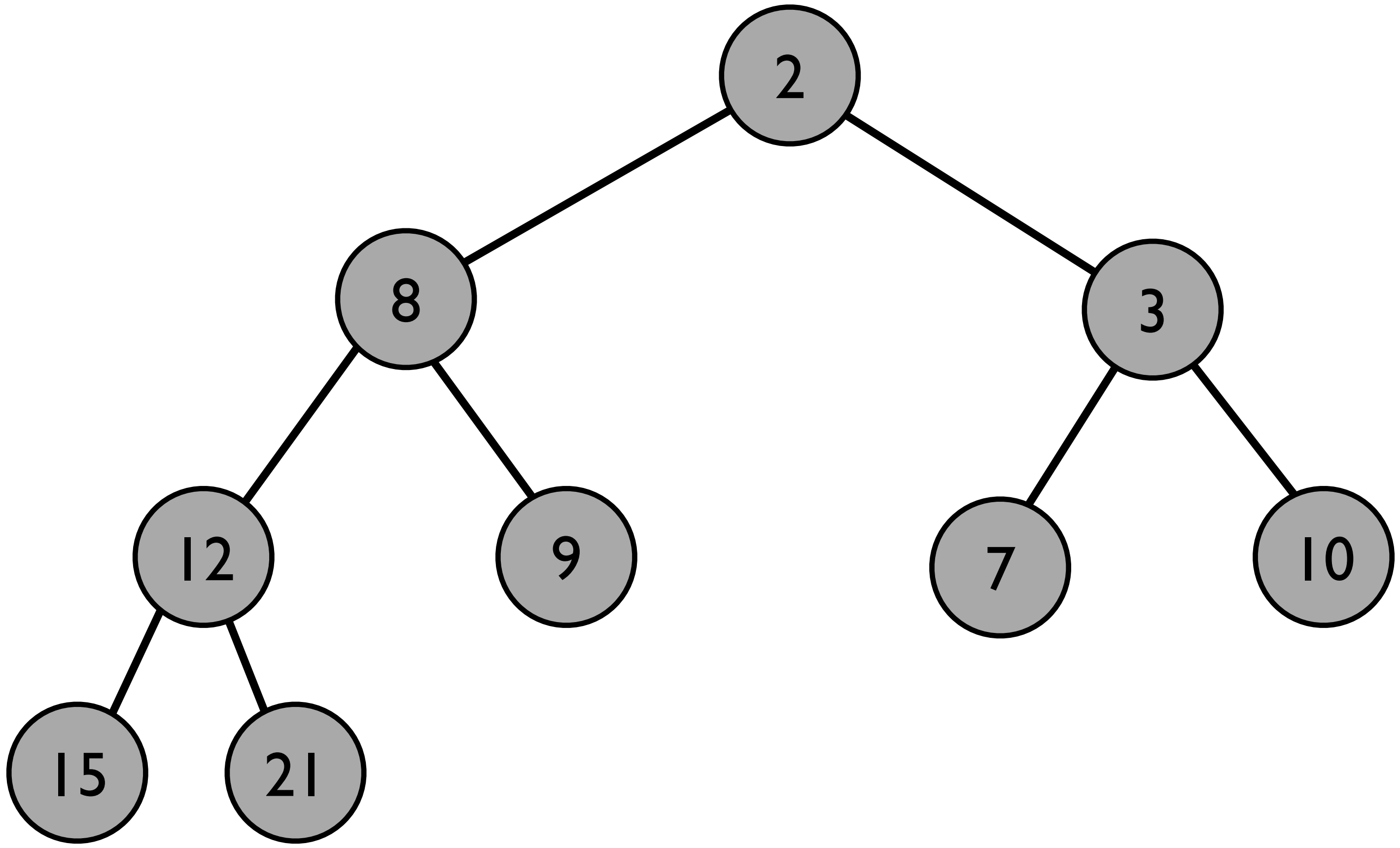
Store Heap in a Complete Tree



Arrays ⇔ Complete Binary Trees



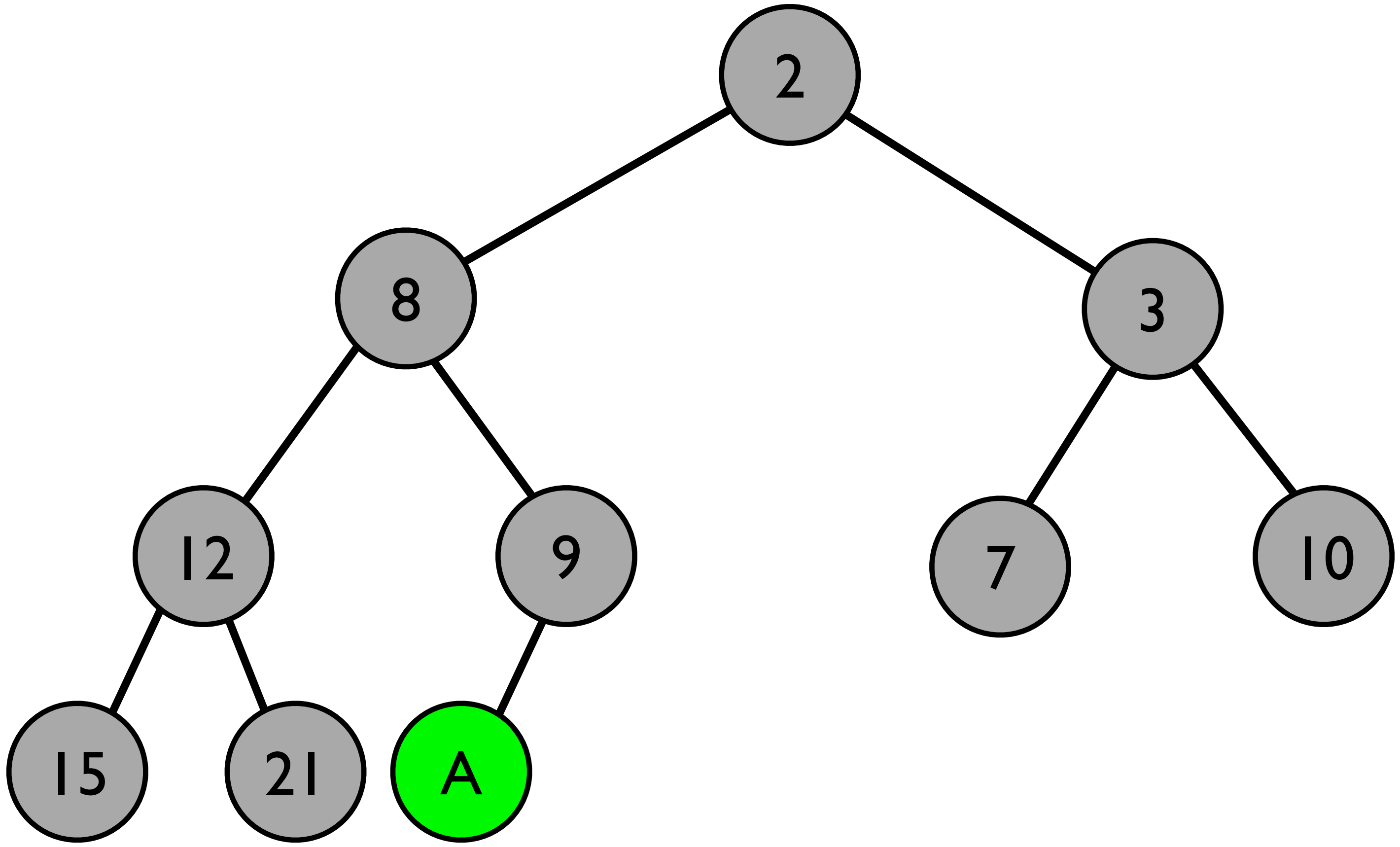
Store Heap in a Complete Tree



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** $2i \le n$ **otherwise** None
right(i): $(2i + 1)$ **if** $2i + 1 \le n$ **otherwise** None
parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

Store Heap in a Complete Tree



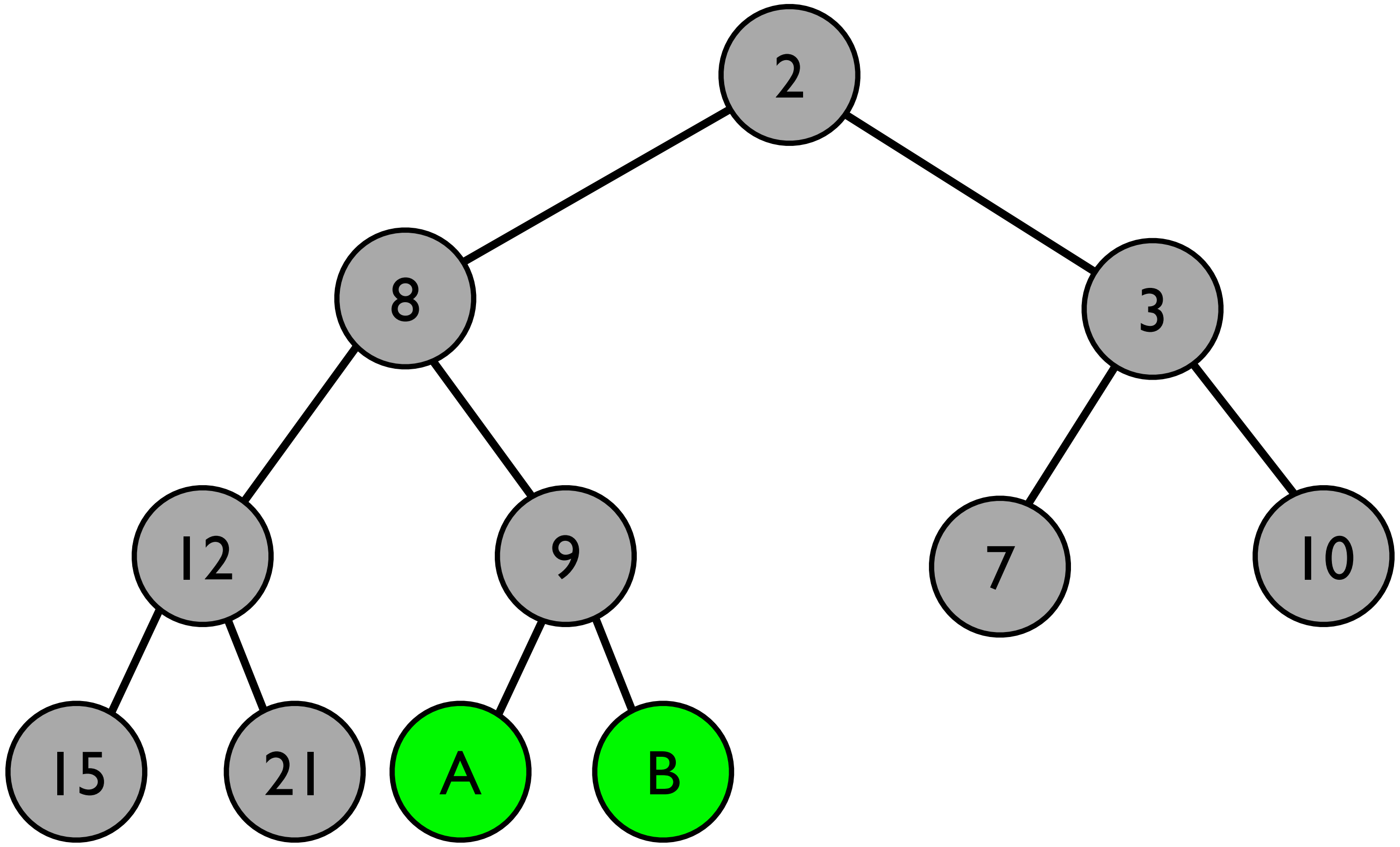
2	8	3	12	9	7	10	15	21	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

left(i): $2i$ **if** $2i \leq n$ **otherwise** None

right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** None

parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

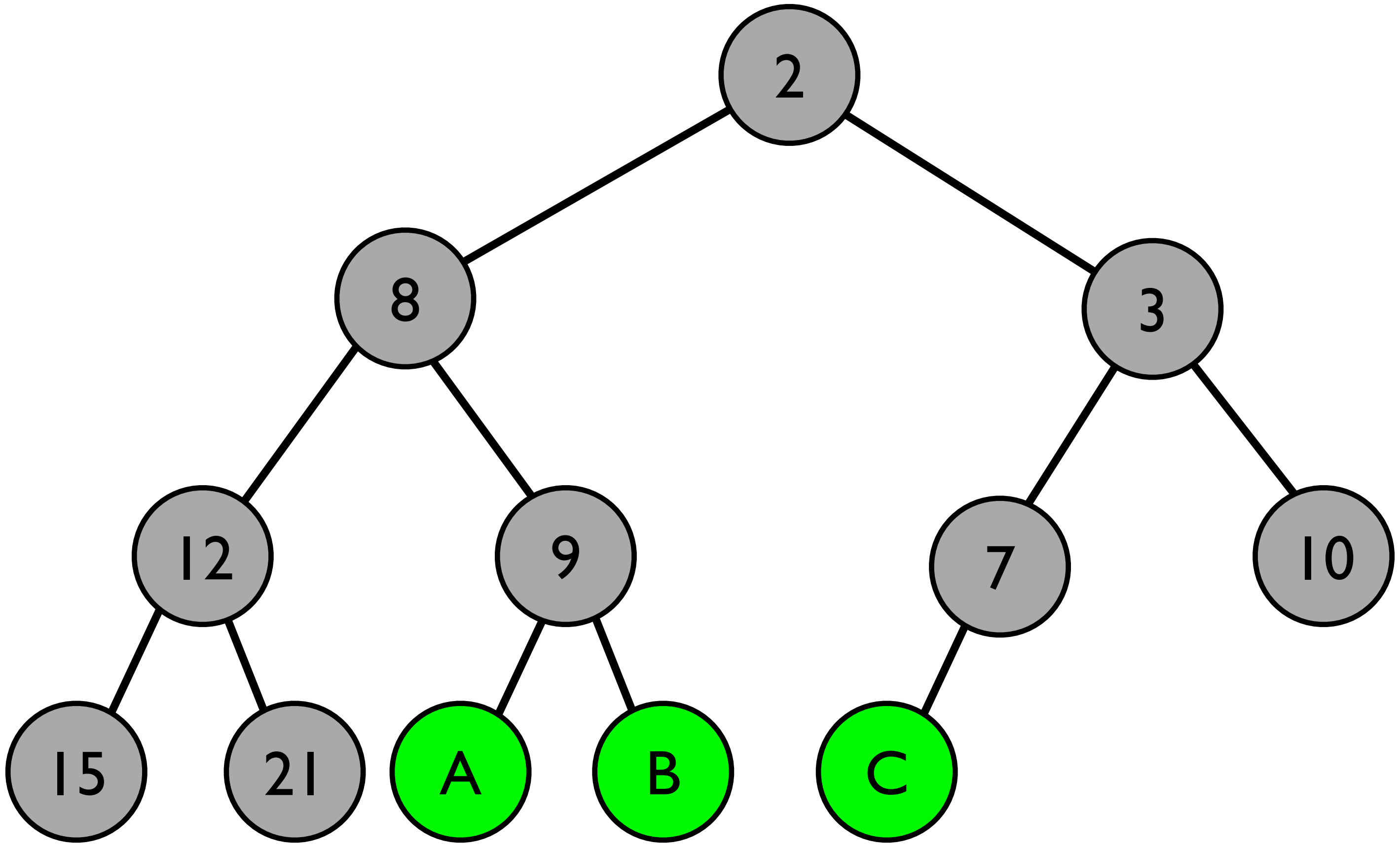
Store Heap in a Complete Tree



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** None
right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** None
parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

Store Heap in a Complete Tree



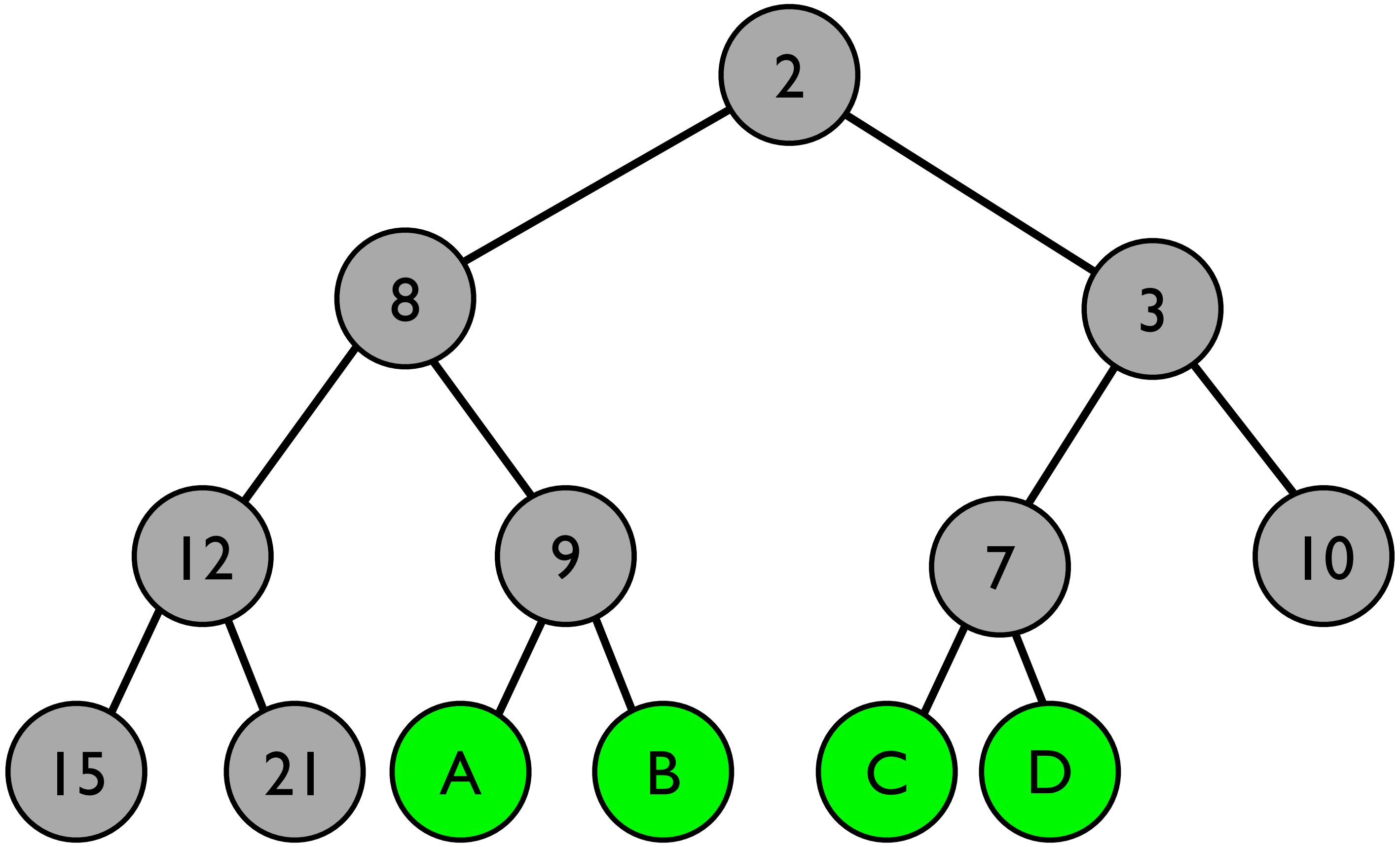
2	8	3	12	9	7	10	15	21	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

left(i): $2i$ **if** $2i \le n$ **otherwise** None

right(i): $(2i + 1)$ **if** $2i + 1 \le n$ **otherwise** None

parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

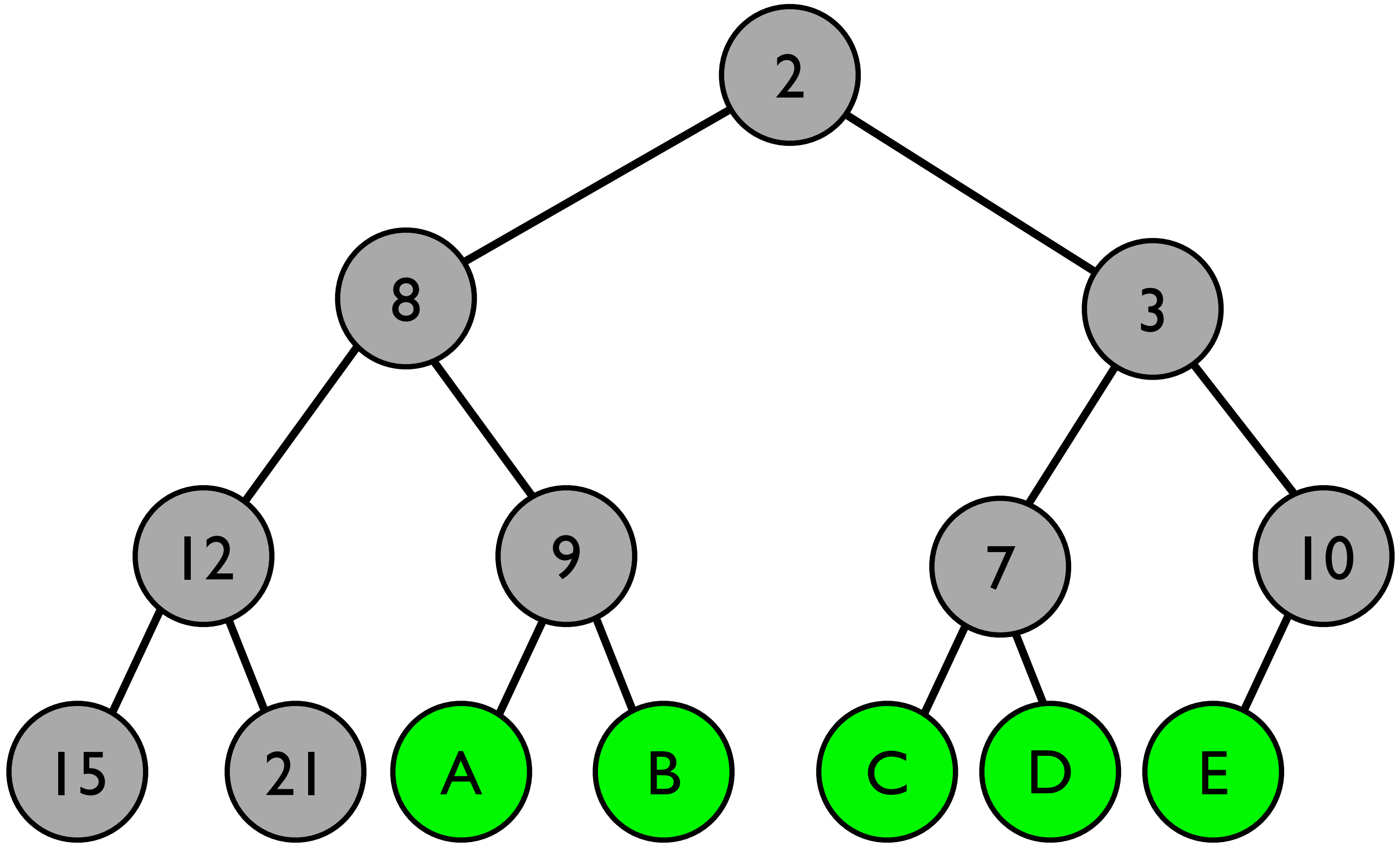
Store Heap in a Complete Tree



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** $2i \le n$ **otherwise** None
right(i): $(2i + 1)$ **if** $2i + 1 \le n$ **otherwise** None
parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

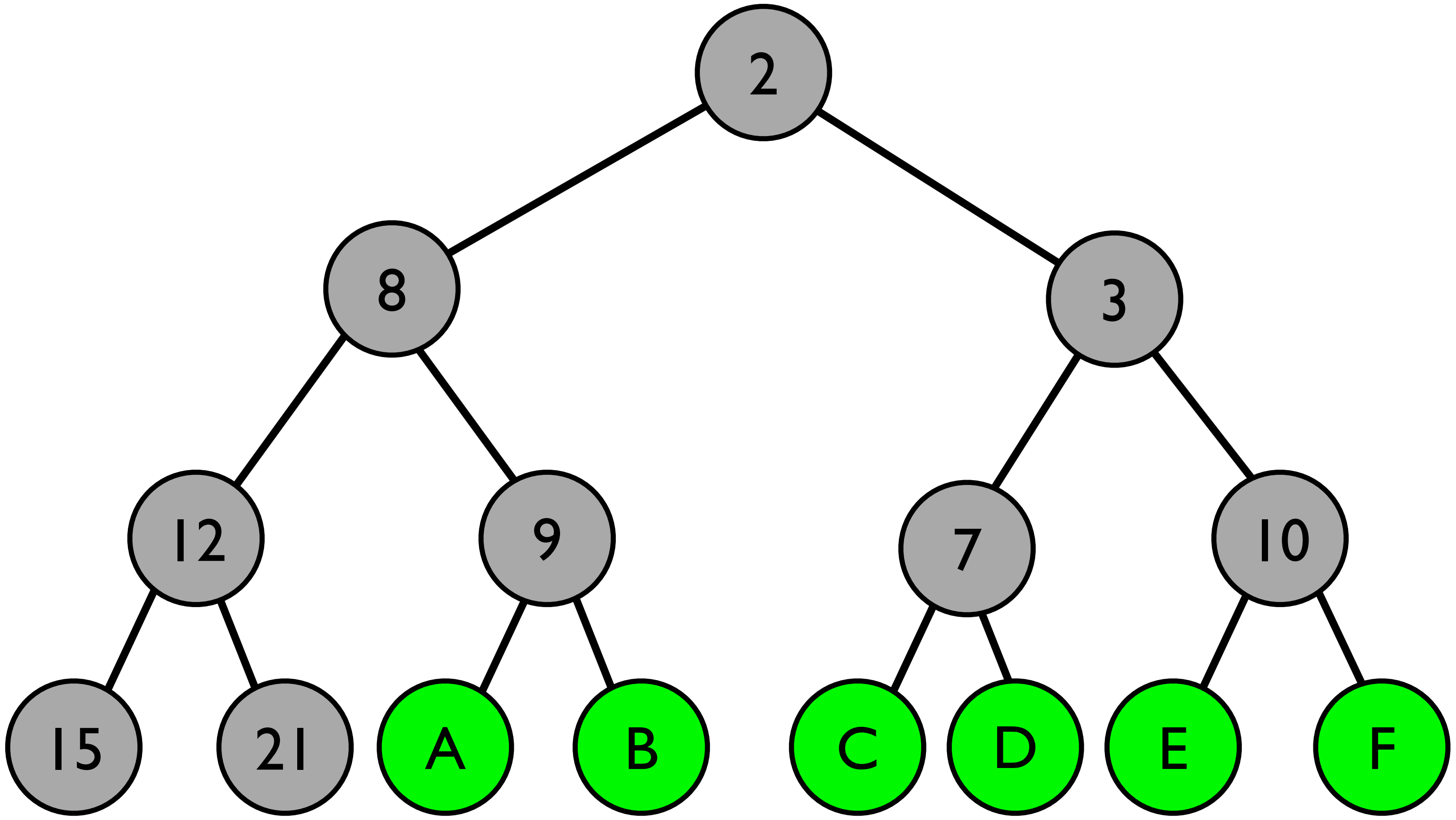
Store Heap in a Complete Tree



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** $2i \le n$ **otherwise** None
right(i): $(2i + 1)$ **if** $2i + 1 \le n$ **otherwise** None
parent(i): $\lfloor i/2 \rfloor$ **if** i $\ge$ 2 **otherwise** None

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** None

right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** None

parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** None

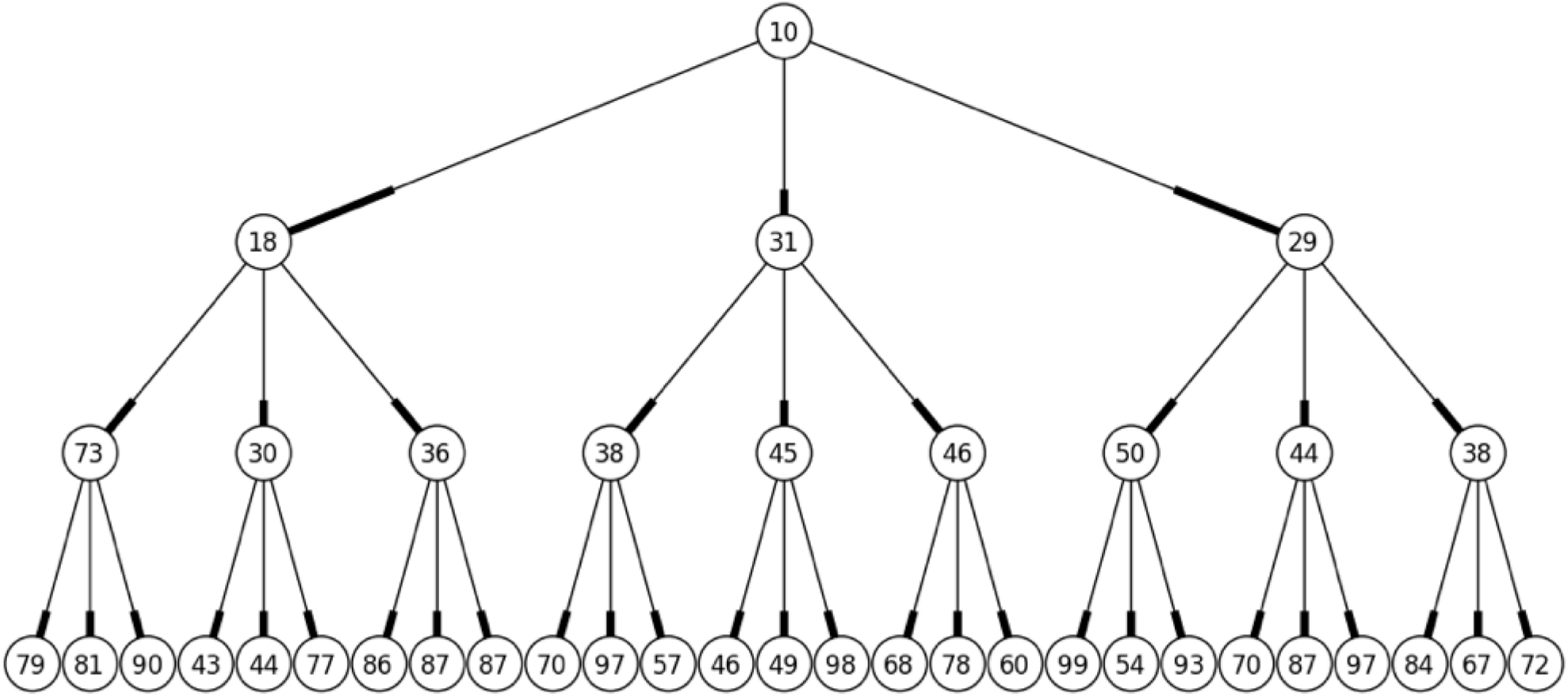
d-Heaps – Don't have to use binary trees

- 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 $O(\boldsymbol{d} \log_d n)$ [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.

3-Heap Example



d-Heap Runtime Summary

- *findmin* takes O(1) time
- *insert* takes $O(\log_d n)$ time
- *delete* takes $O(d \log_d n)$ time
- *deletemin* takes time $O(d \log_d n)$
- *makeheap* takes $O(n)$ time

Reason: height of a complete binary tree with n nodes is about $\log n$.

One more operation needed for Prim's

- H.decreaseKey(u,j): reduce the key for item u by j.

One more operation needed for Prim's

- H.decreaseKey(*u*,*j*): reduce the key for item *u* by *j*.

Why is this needed?

One more operation needed for Prim's

- H.decreaseKey(u,j): reduce the key for item u by j.

Why is this needed?

How can we implement it?

One more operation needed for Prim's

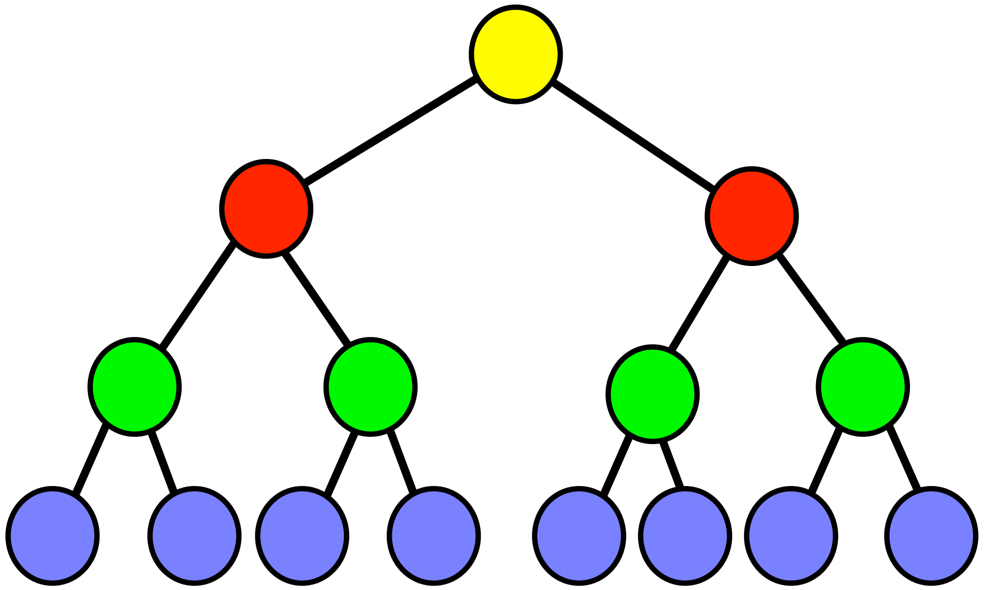
- H.decreaseKey(u,j): reduce the key for item u by j.

Why is this needed?

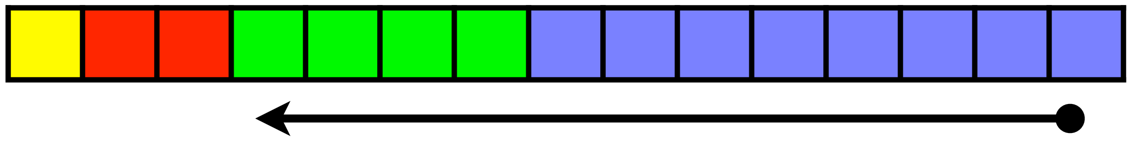
How can we implement it?

1. Reduce the key by j
2. *siftup* to put it in the right place.
3. Takes time proportional to the height of the tree ≈ log n

Make Heap – Create a heap from *n* items



- n inserts take time $\propto n \log n$.
- Better:
  - put items into array arbitrarily.
  - **for** $i = \mathrm{n} \ldots 1$, $\mathit{siftdown}(i)$.
- Each element trickles down to its correct place.



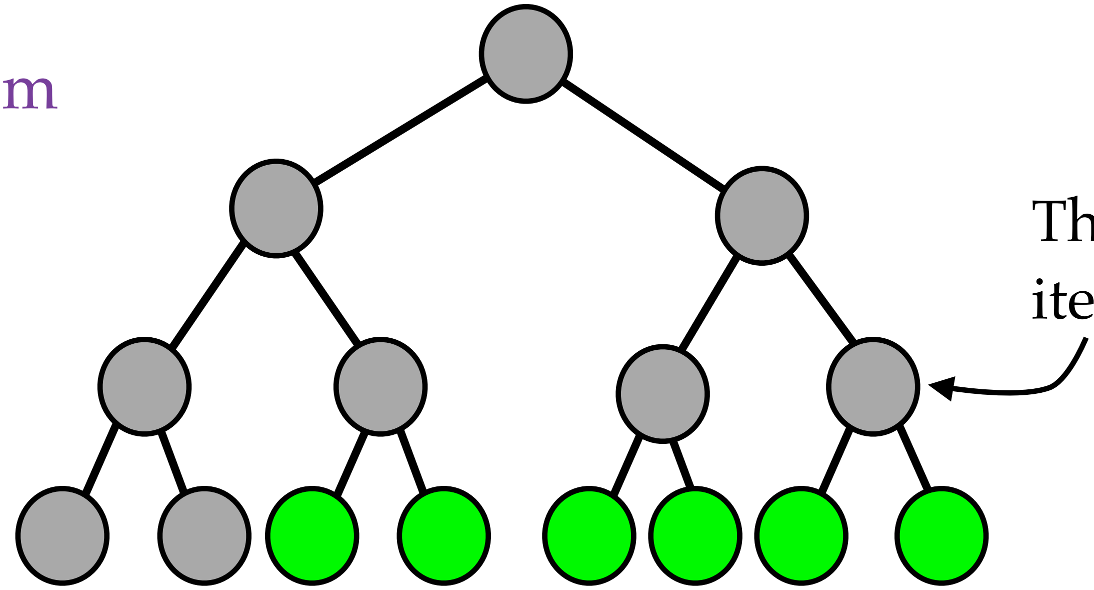
By the time you sift level i, all levels $i + 1$ and greater are already heap ordered.

Make Heap – Time Bound

Height counts from bottom
Level counts from top

2^{H-h} nodes at height h in a tree with total height $H = 2^H \ / \ 2^h$

$H \approx \log n$; $2^H = n$



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

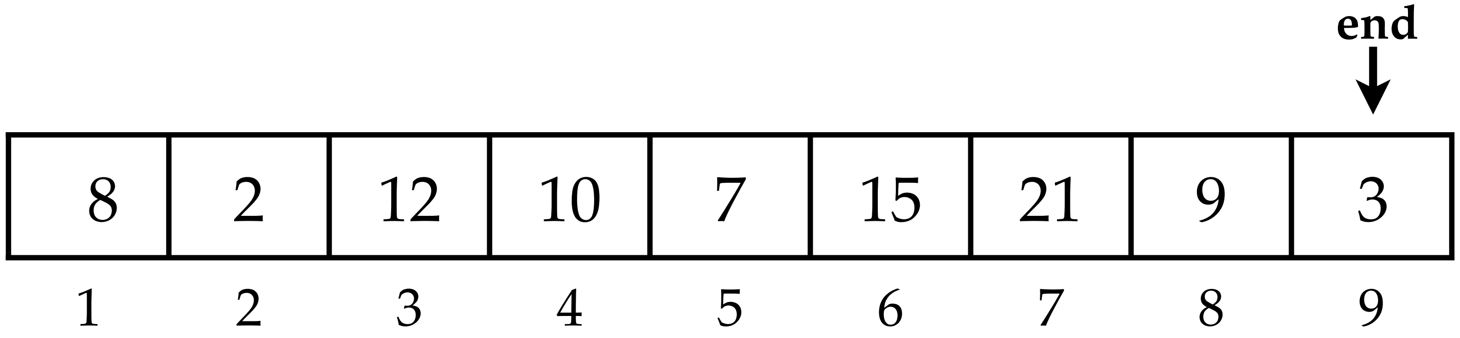
Siftdown for all height h nodes is $\approx hn/2^h$ time

Total time

$\approx \sum_h hn/2^h$	[sum of time for each height]
$= n \sum_h (h \ / \ 2^h)$	[factor out the n]
$\approx n$	[sum bounded by const]

Heapsort – Another application of Heaps

Given unsorted array of integers



end

8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

Heapsort – Another application of Heaps

Given unsorted
array of integers

								end ↓
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

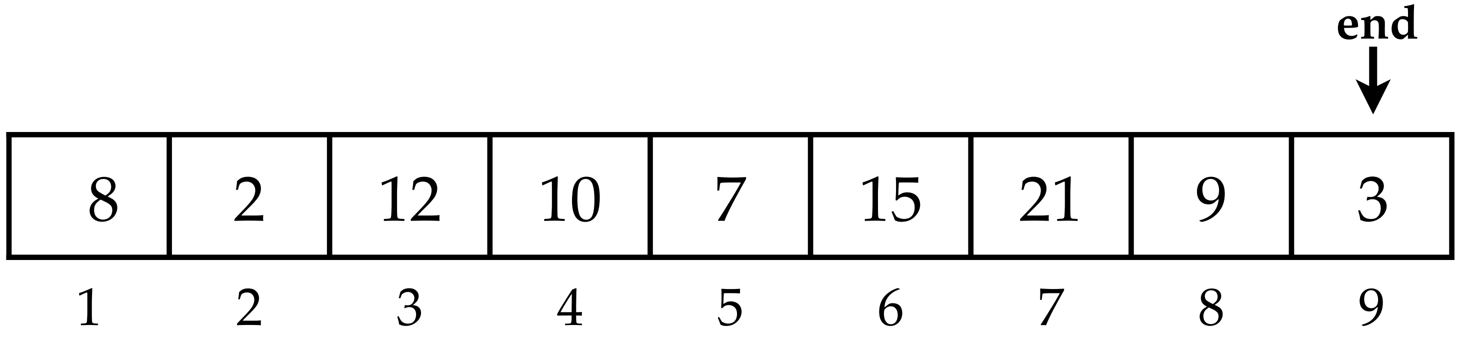
makeheap – O(n)
Now first position
has smallest item.

								end ↓
2	3	12	8	7	15	21	9	10
1	2	3	4	5	6	7	8	9

Swap first & last items.

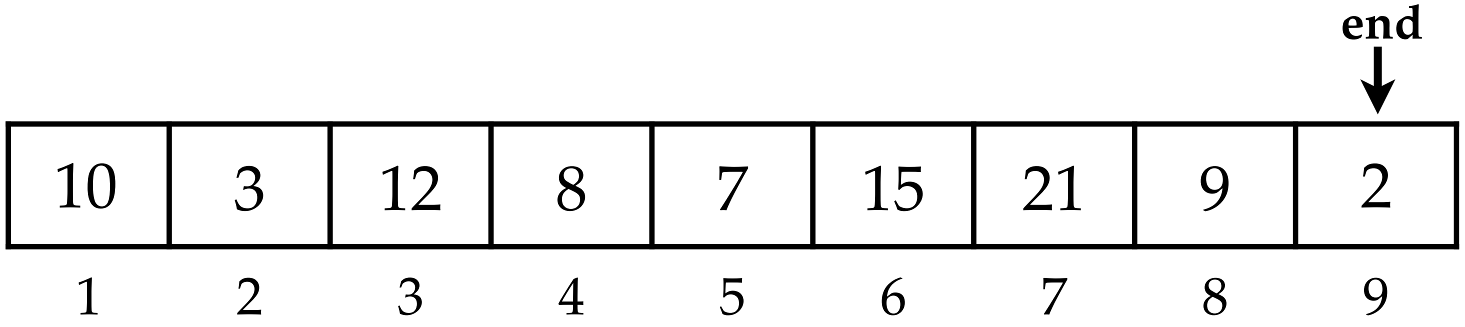
Heapsort – Another application of Heaps

Given unsorted
array of integers



8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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



10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Swap first & last items.

Heapsort – Another application of Heaps

Given unsorted
array of integers

end ↓

8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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

end ↓

10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Delete last item from heap.

end ↓

10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Heapsort – Another application of Heaps

Given unsorted
array of integers

								end ↓
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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

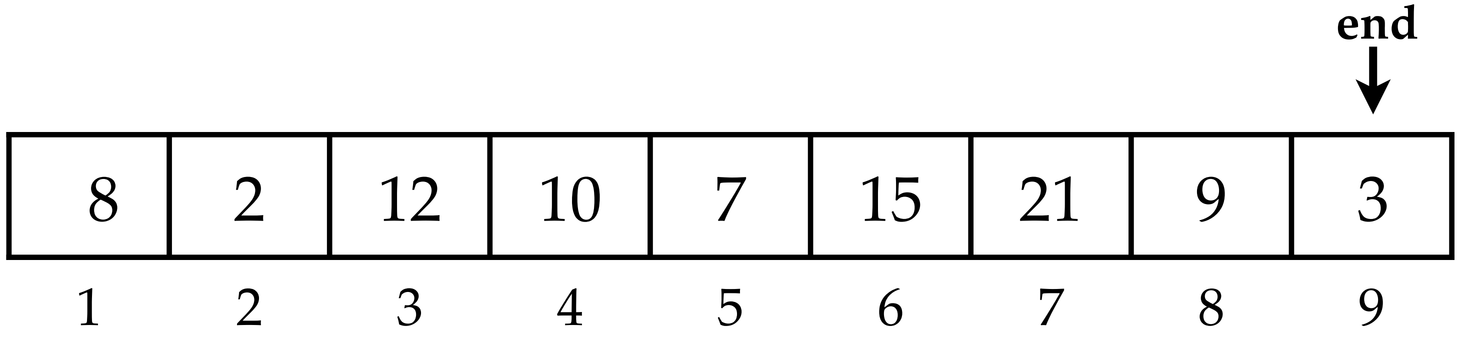
								end ↓
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Delete last item from heap.

							end ↓	
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

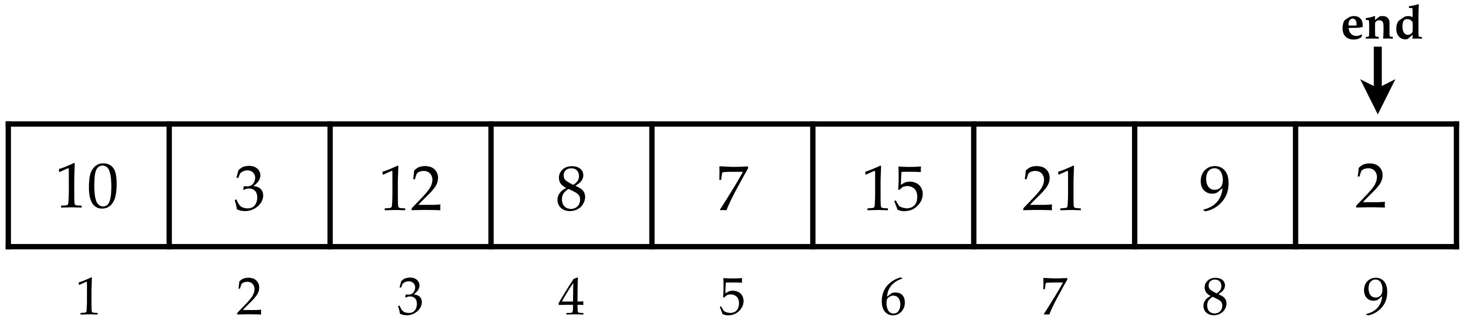
Heapsort – Another application of Heaps

Given unsorted
array of integers



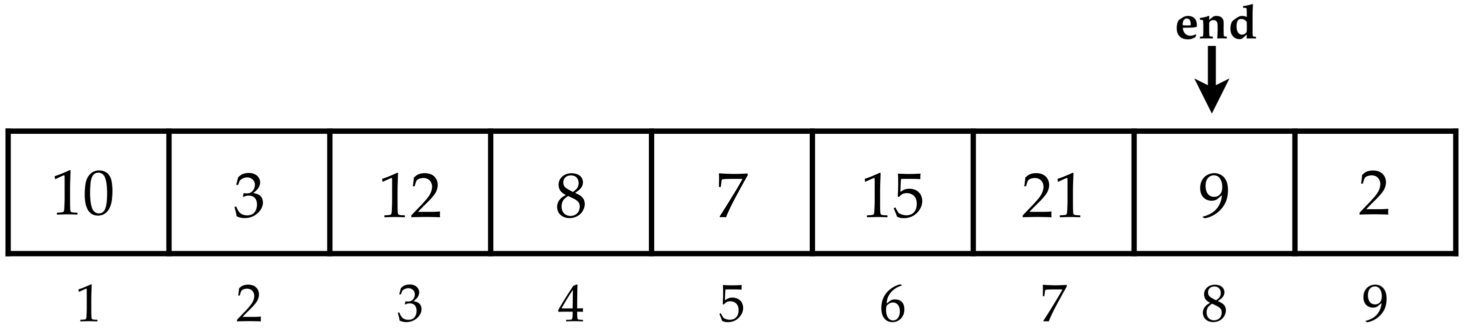
								end ↓
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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



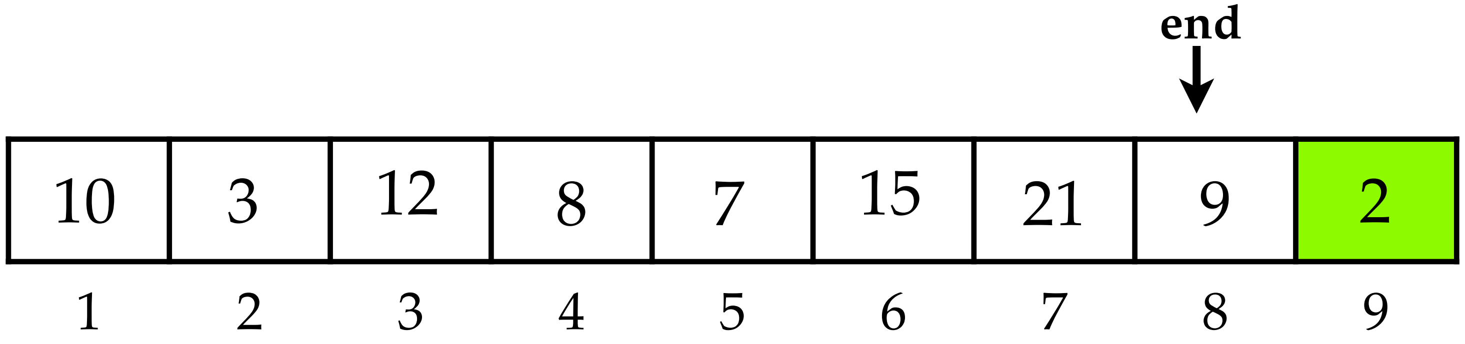
								end ↓
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Delete last item from heap.



							end ↓	
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

siftdown new root key
down



							end ↓	
10	3	12	8	7	15	21	9	🟩 2
1	2	3	4	5	6	7	8	9

Heapsort – Another application of Heaps

Given unsorted
array of integers

								end ↓
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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

								end ↓
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Delete last item from heap.

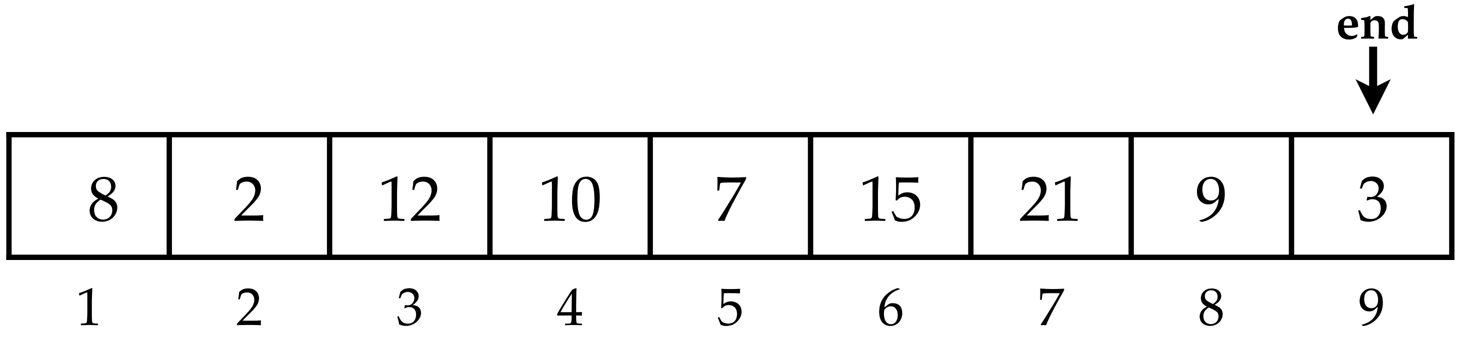
							end ↓	
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

siftdown new root key
down

							end ↓	
3	7	12	8	10	15	21	9	2
1	2	3	4	5	6	7	8	9

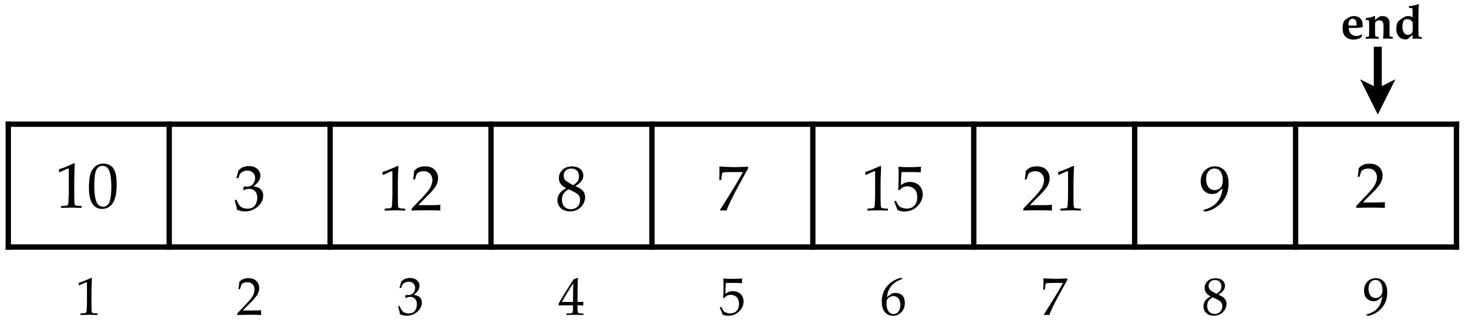
Heapsort – Another application of Heaps

Given unsorted
array of integers



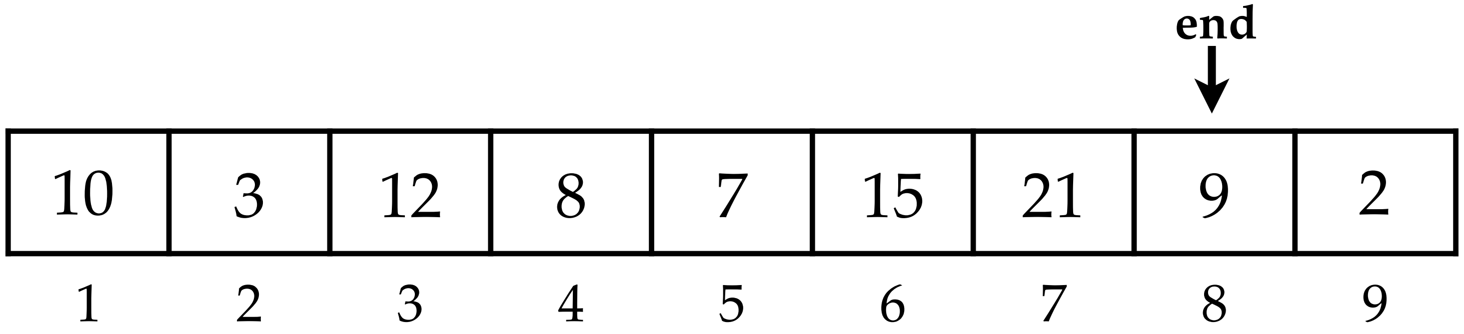
8	2	12	10	7	15	21	9	3
1	2	3	4	5	6	7	8	9

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



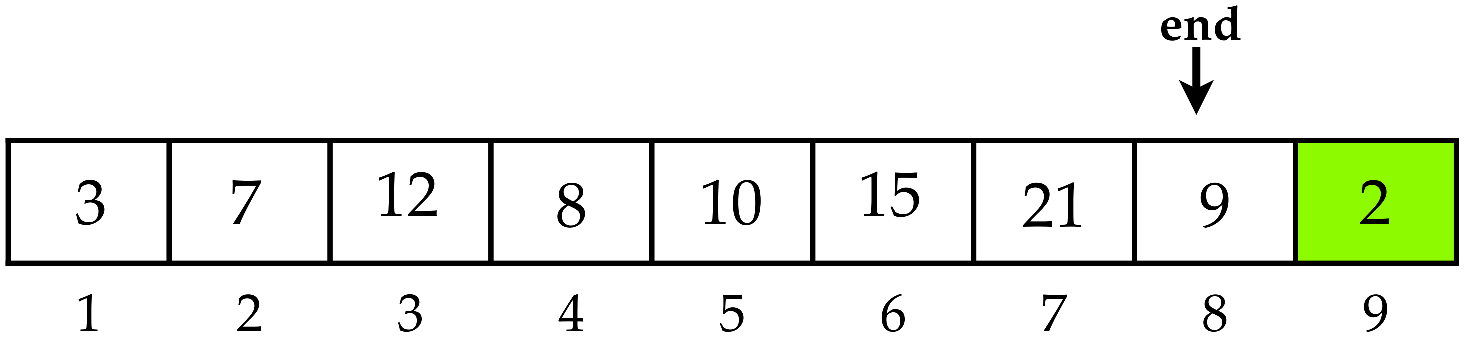
10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

Delete last item from heap.



10	3	12	8	7	15	21	9	2
1	2	3	4	5	6	7	8	9

siftdown new root key
down



3	7	12	8	10	15	21	9	2
1	2	3	4	5	6	7	8	9