Data Structures

Lecture 7
Heaps: Motivation, Binary, and Binomial

Shiri Chechik, Or Zamir
Winter semester 2025-6

Heaps / Priority Queues

ADT:

- Insert, Find-Min, Delete-Min, Decrease-Key

Two notes:

- Delete can be implemented using Decrease-key + Delete-min
- Hopeless challenge: why can’t we have all in O(1)?
- We would eventually want a “meld” or “join” operation, which we don’t have in search trees

Motivation

- More restricted operations would lead to better amortized time for some of them

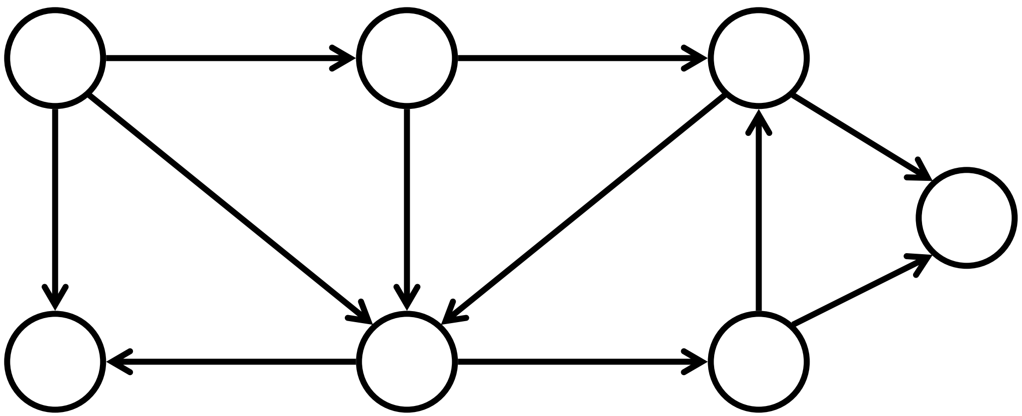
- Examples (algorithms course):
  - Dijkstra’s algorithm for single source shortest path
  - Prim’s algorithm for minimum spanning trees

Dijkstra’s algorithm
for
single source shortest paths

Single source shortest path

- Want to find the shortest route from New York to San Francisco
- Or the shortest neuro-chemical pathway from neuron X to neuron Y in the brain
- Or the shortest internet route from my PC to Amazon…
- Etc.
- All can be model with a graph

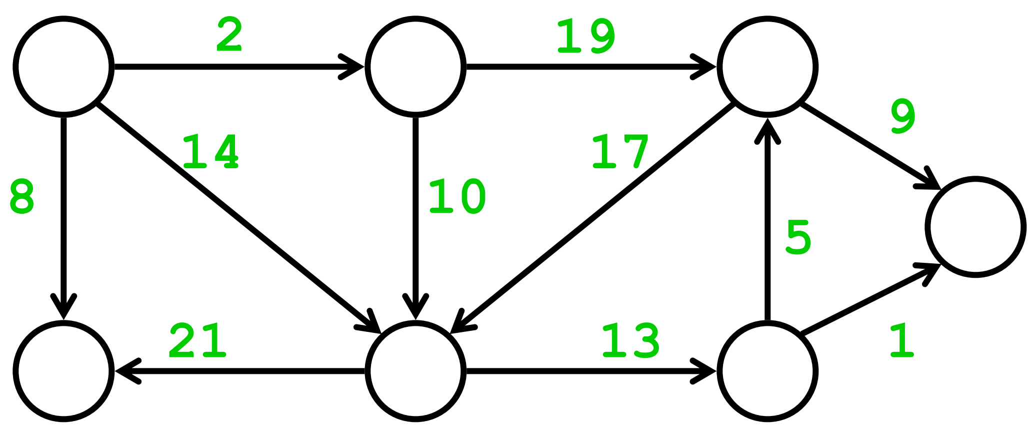
A Graph G=(V,E)



V is a set of vertices

E is a set of edges (pairs of vertices)

Model distances by edges weights

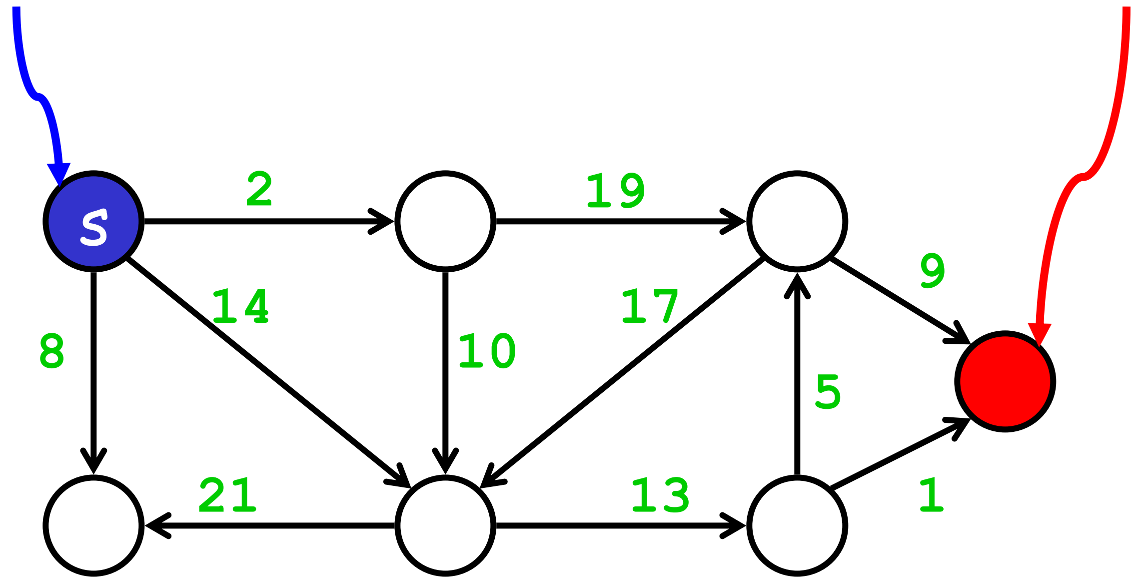


V is a set of vertices

E is a set of edges (pairs of vertices)

Assume all weights are non-negative

Source and destinations



- Want to find the shortest path from some fixed source vertex s to every other vertex

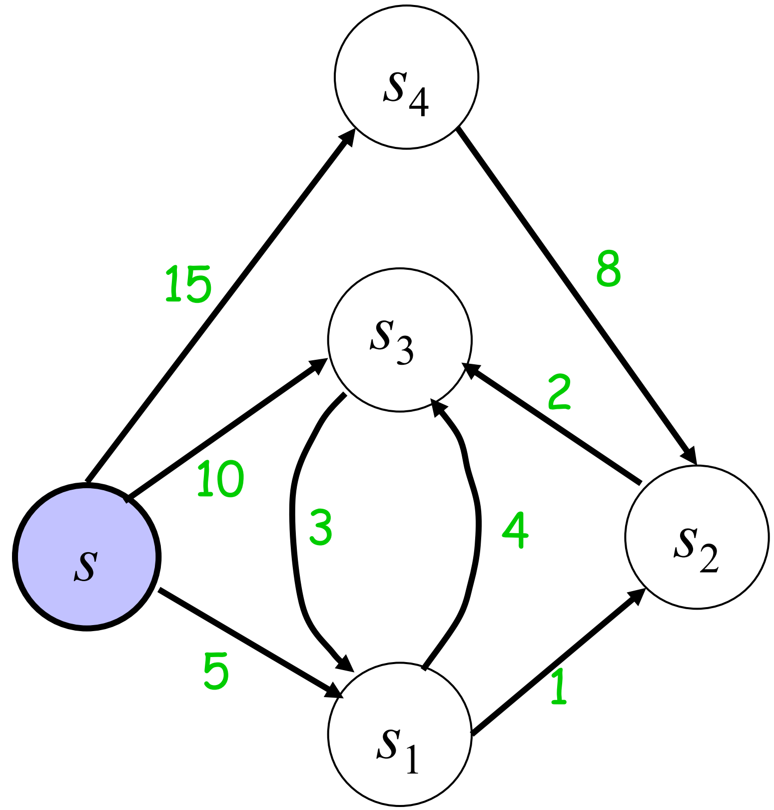
Dijkstra's algorithm: Simulation

- Input:

$$G = (V, E),$$
$$s \in V$$

- Output:

$\forall v \in V$ its distance from s and a path of that length

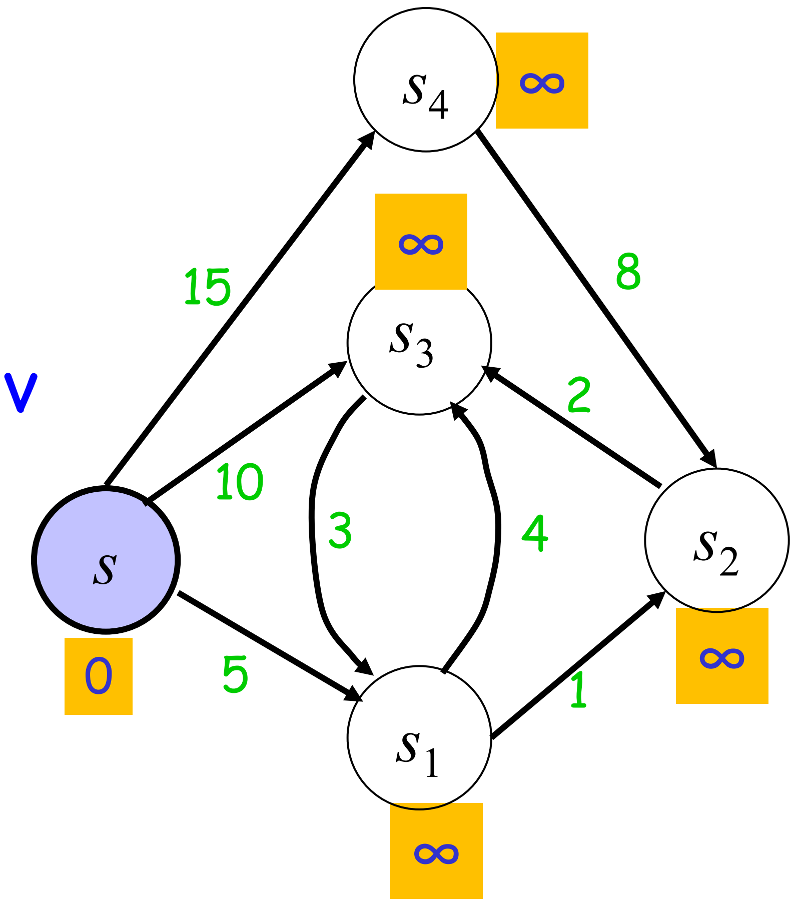


Simulation

For every node v,

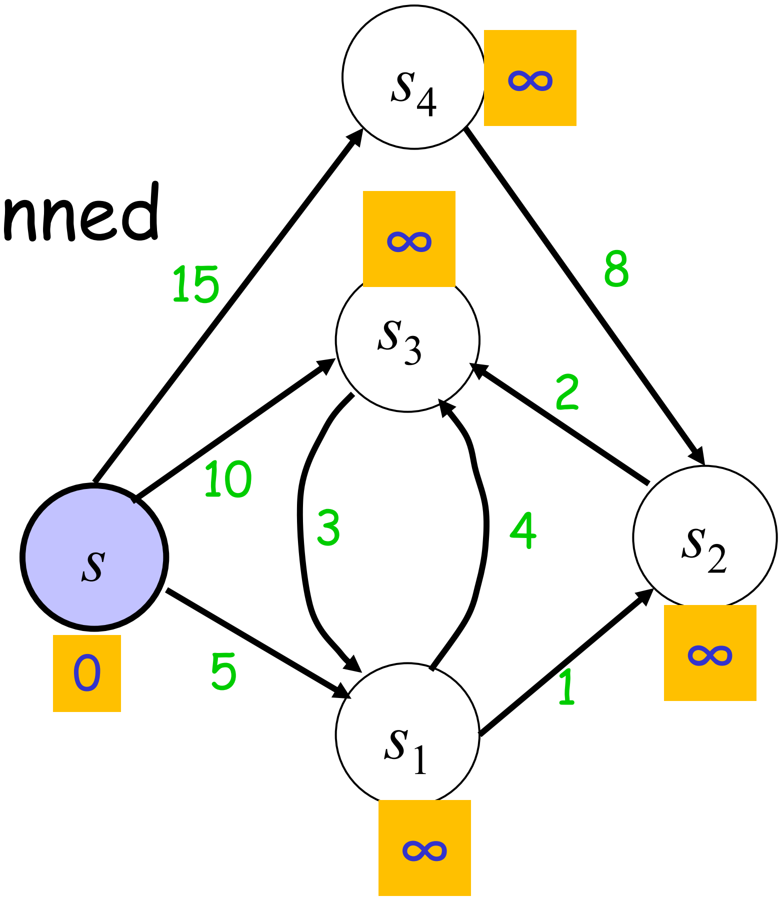
maintain d(v) -

an upper bound on the shortest path to from s to v



Maintain 2 sets:

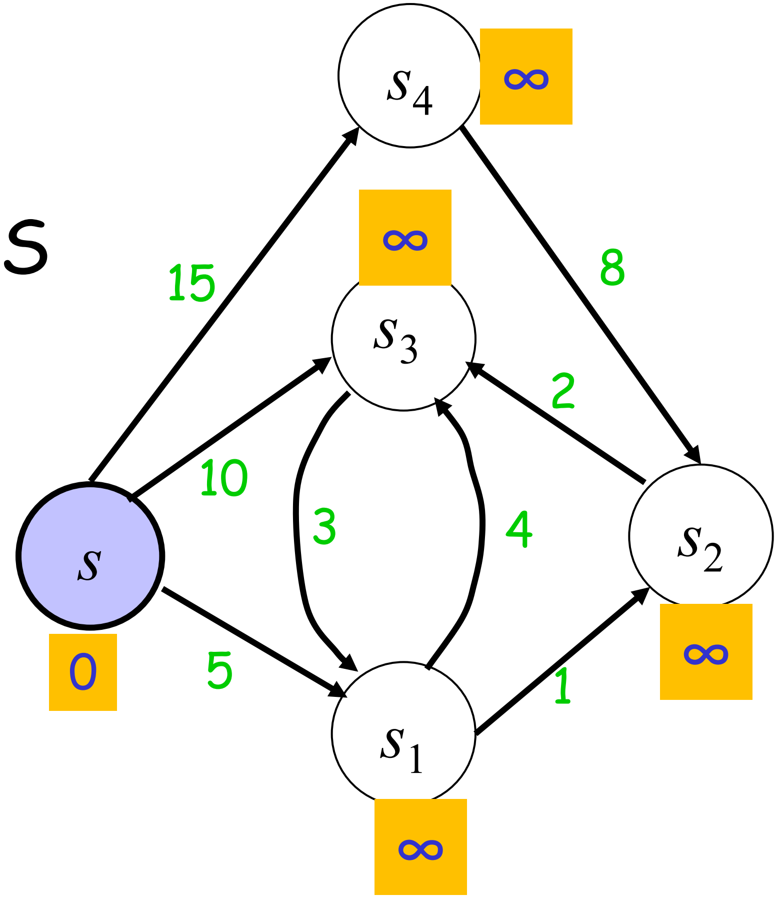
- S - vertices already scanned
  - Initially $S = \varnothing$
- Q - vertices yet to be scanned
  - Initially $Q = V$



Initially $S = \emptyset$ and $Q = V$

Pick a vertex $v \in Q$ with minimal $d(v)$ and move it to S

Now $S=\{s\}$

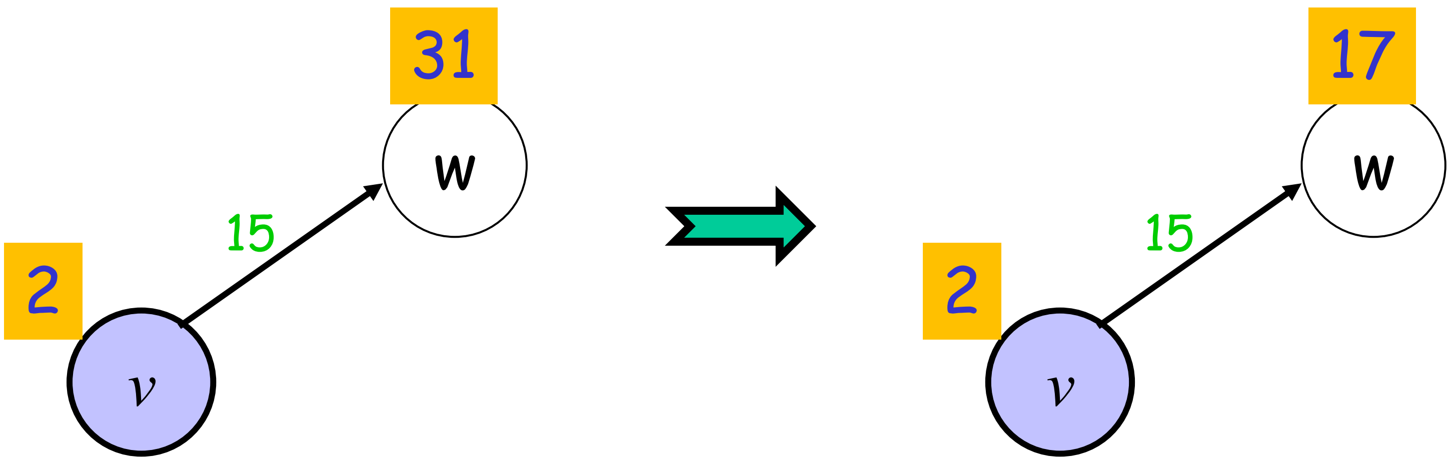


For every edge (v,w) where w in Q relax(v,w)

Relax(v,w)

If $d(w) > d(v) + w(v,w)$ then

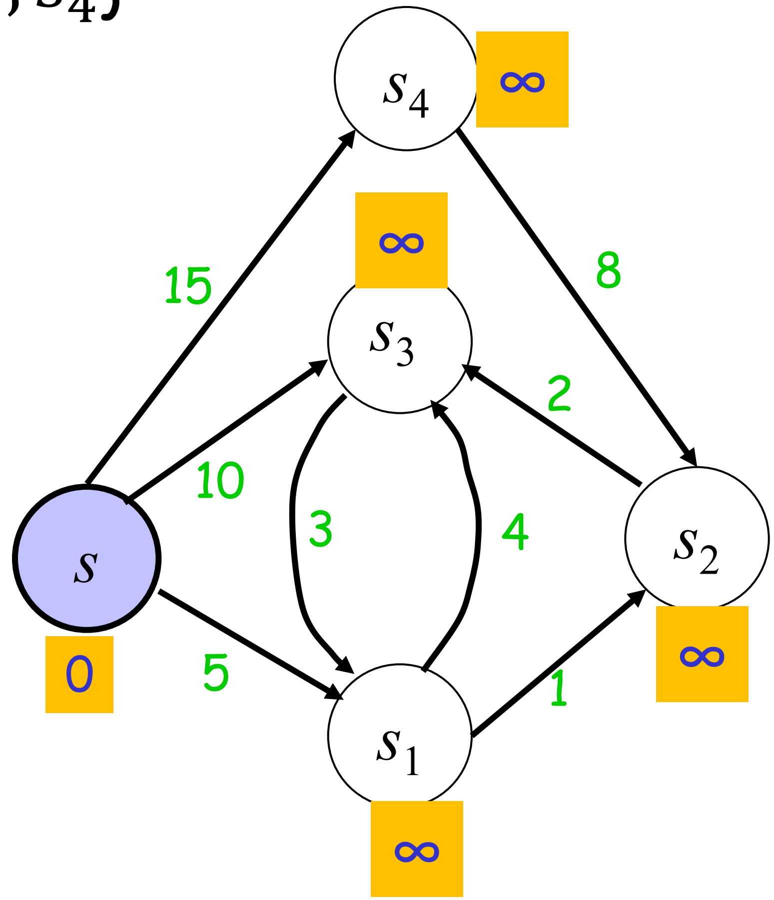
$d(w) \leftarrow d(v) + w(v,w)$



$S = \{s\}$

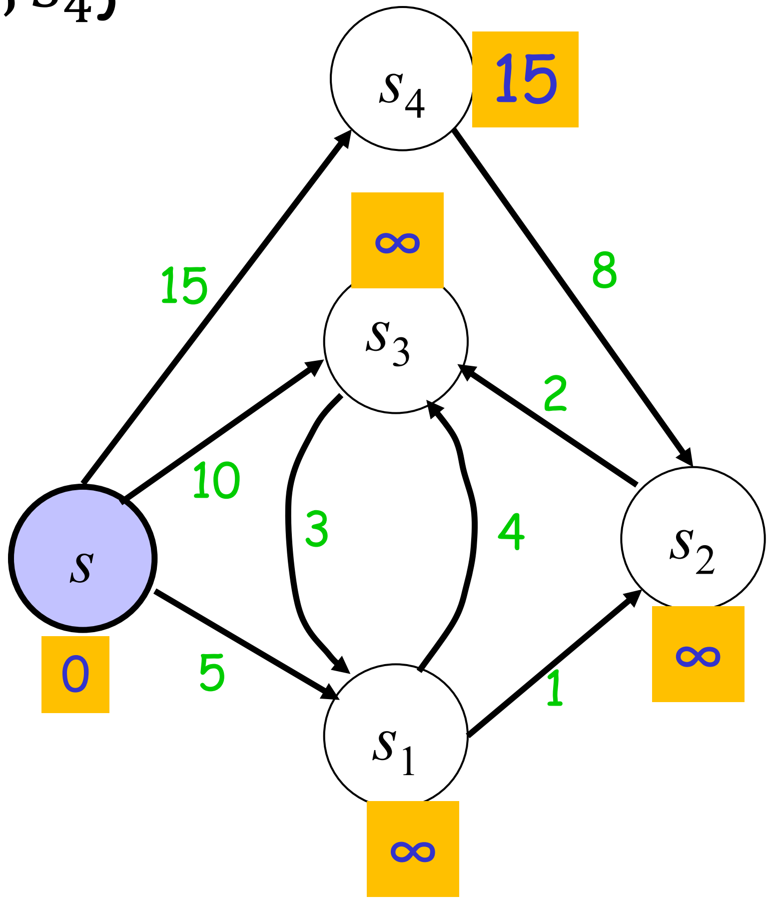
$Q = \{s_1, s_2, s_3, s_4\}$

Relax(s,s_4)



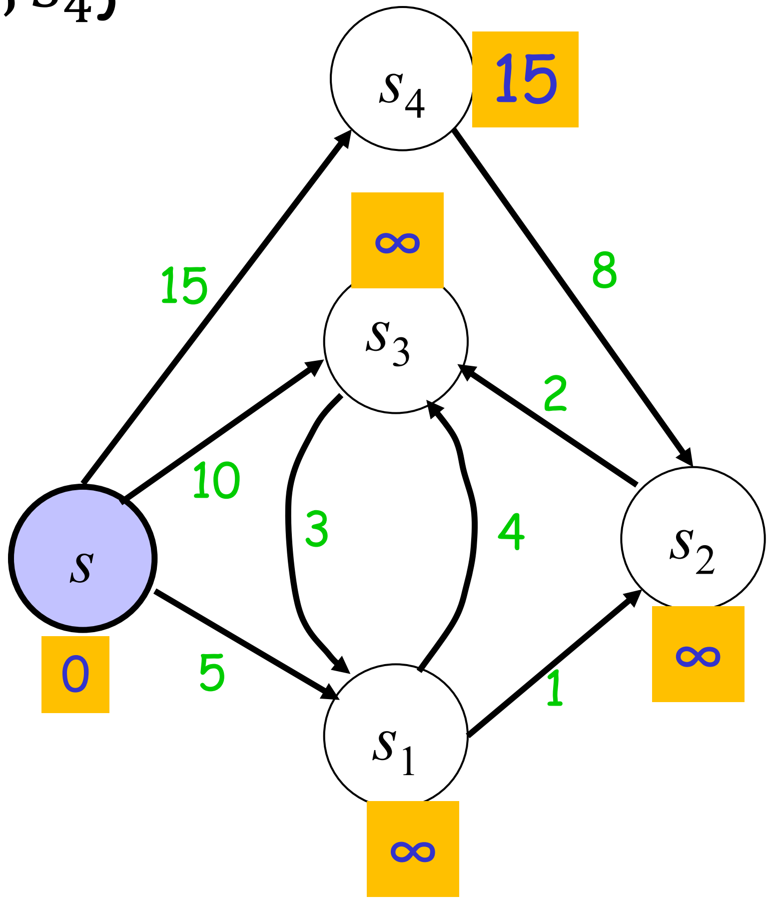
$S = \{s\}$ $Q = \{s_1, s_2, s_3, s_4\}$

Relax(s,s_4)

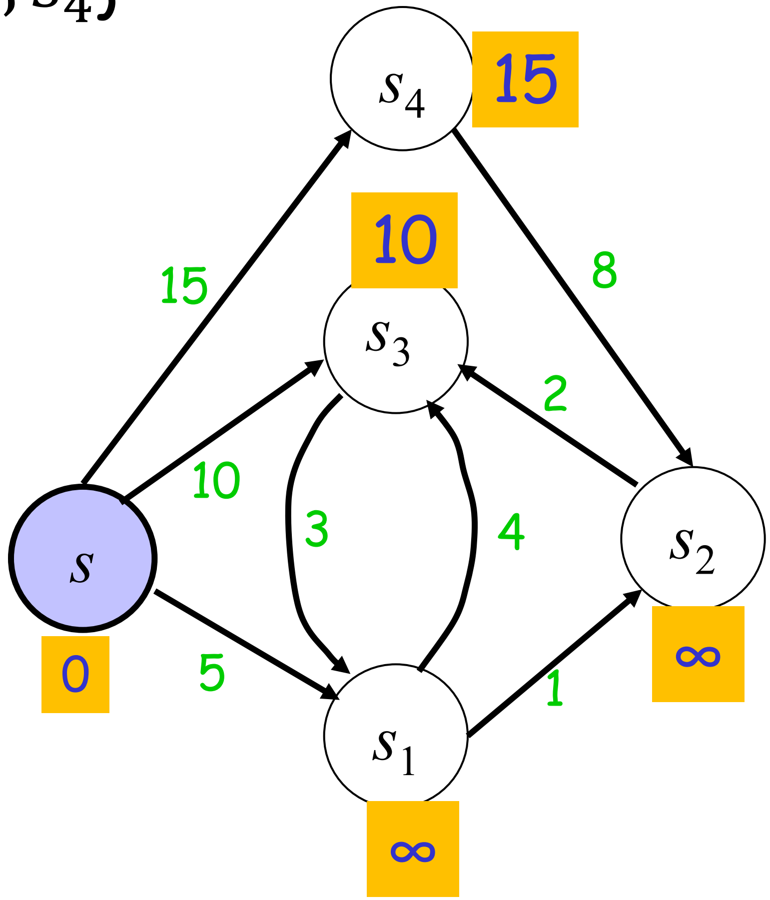


$S = \{s\}$ $Q=\{s_1, s_2, s_3, s_4\}$

Relax(s,s_4)
Relax(s,s_3)

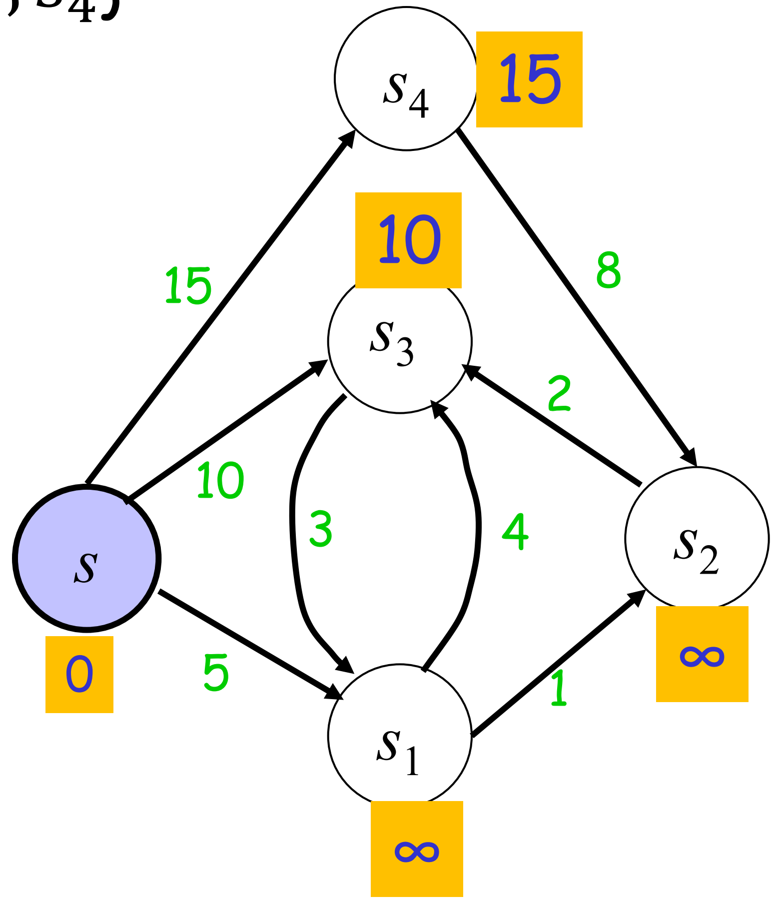


$S = \{s\}$ $Q=\{s_1, s_2, s_3, s_4\}$



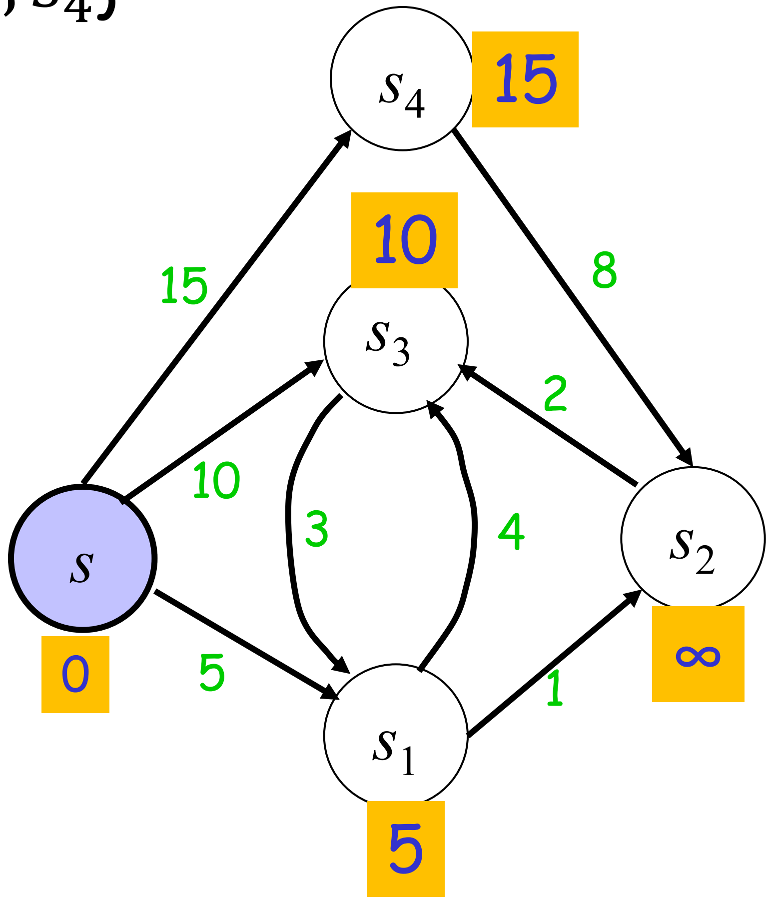
Relax(s,s_4)
Relax(s,s_3)

$S = \{s\}$ $Q=\{s_1, s_2, s_3, s_4\}$



Relax(s,s_4)
Relax(s,s_3)
Relax(s,s_1)

$S = \{s\}$ $Q=\{s_1, s_2, s_3, s_4\}$

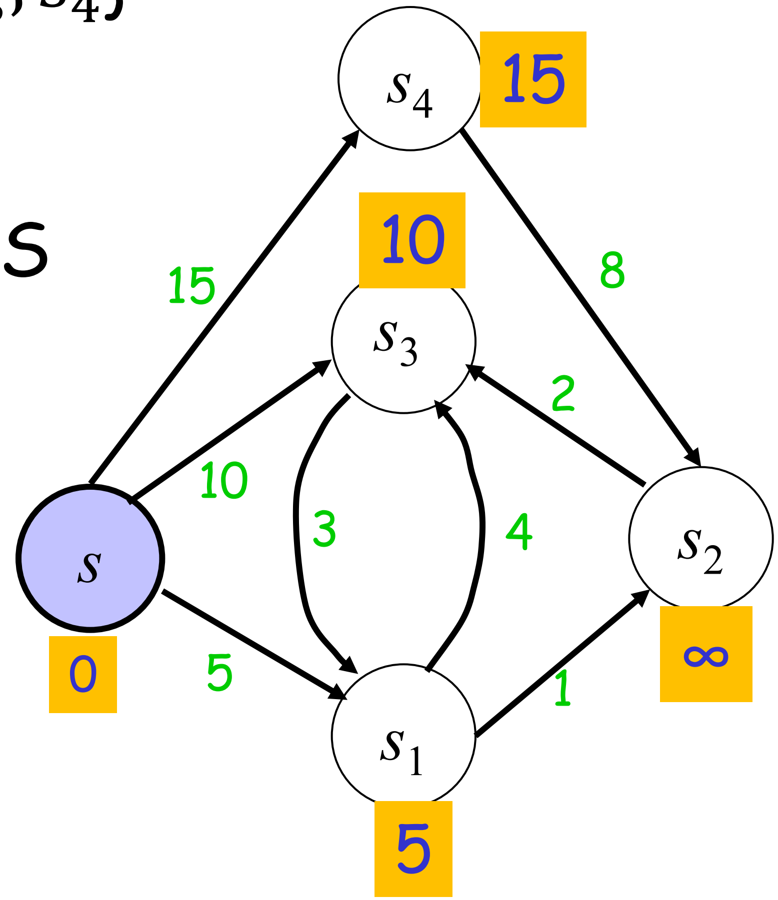


Relax(s,s_4)
Relax(s,s_3)
Relax(s,s_1)

S = {s}

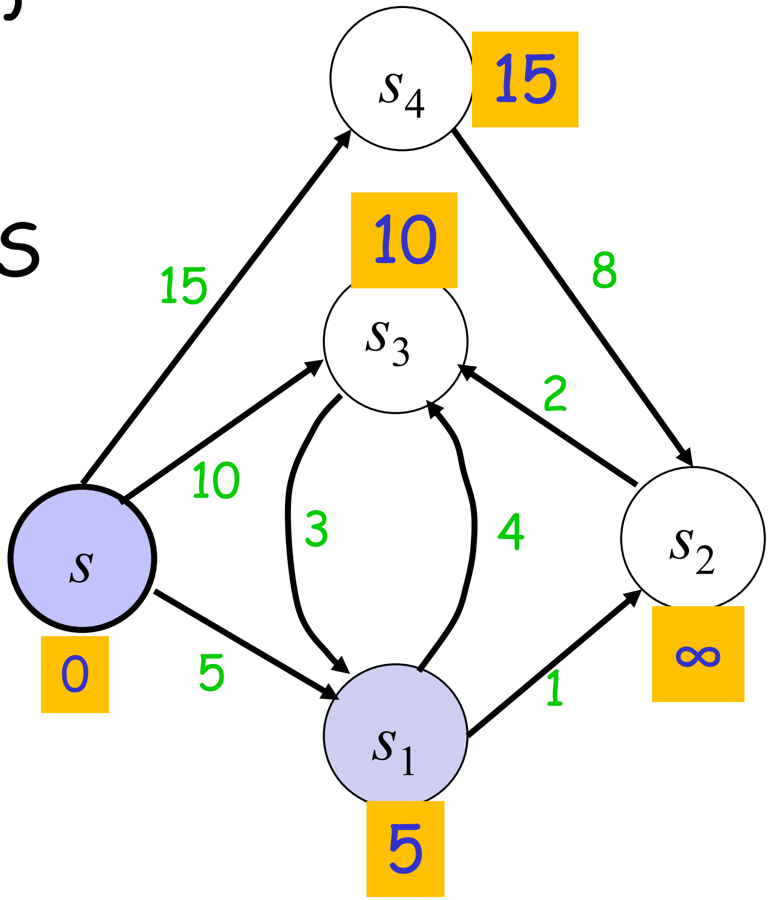
Q={s_1, s_2, s_3, s_4}

Pick a vertex v∈Q with minimal d(v) and move it to S



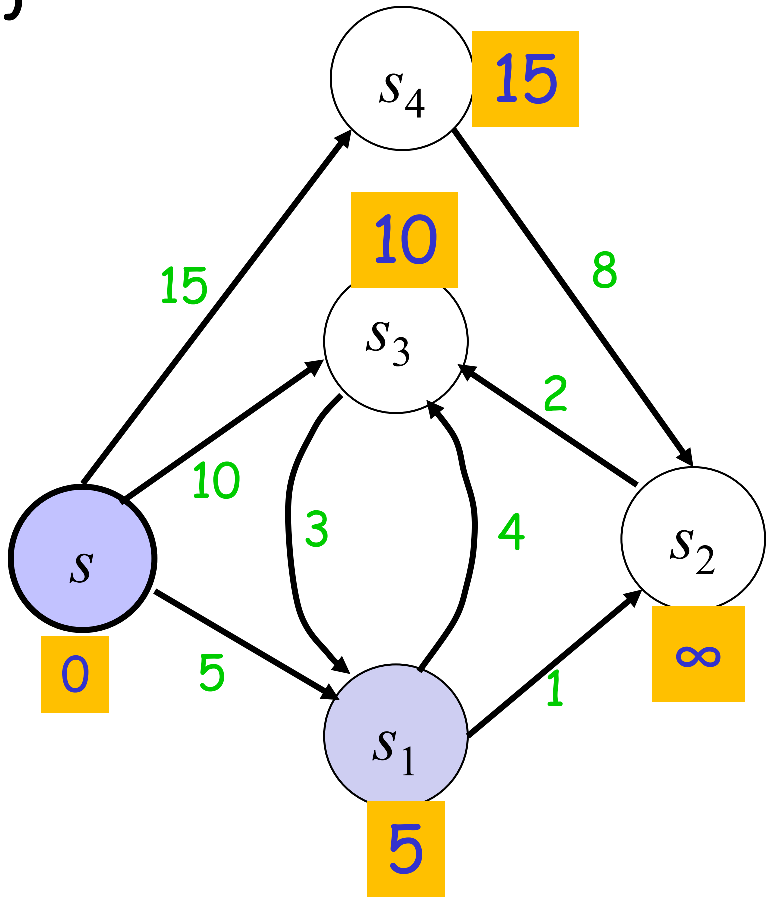
$S = \{s, s_1\}$ $Q=\{s_2, s_3, s_4\}$

Pick a vertex $v \in Q$ with **minimal d(v)** and move it to S



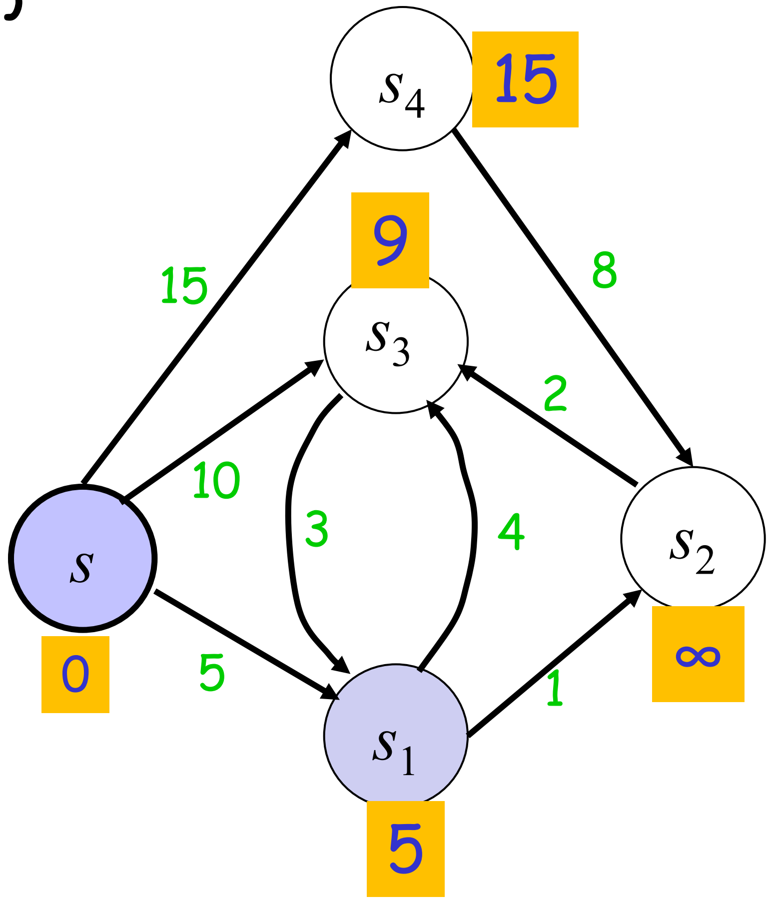
S = {s,s_1}　　　Q={s_2, s_3, s_4}

Relax(s_1,s_3)



$S = \{s, s_1\}$ $Q=\{s_2, s_3, s_4\}$

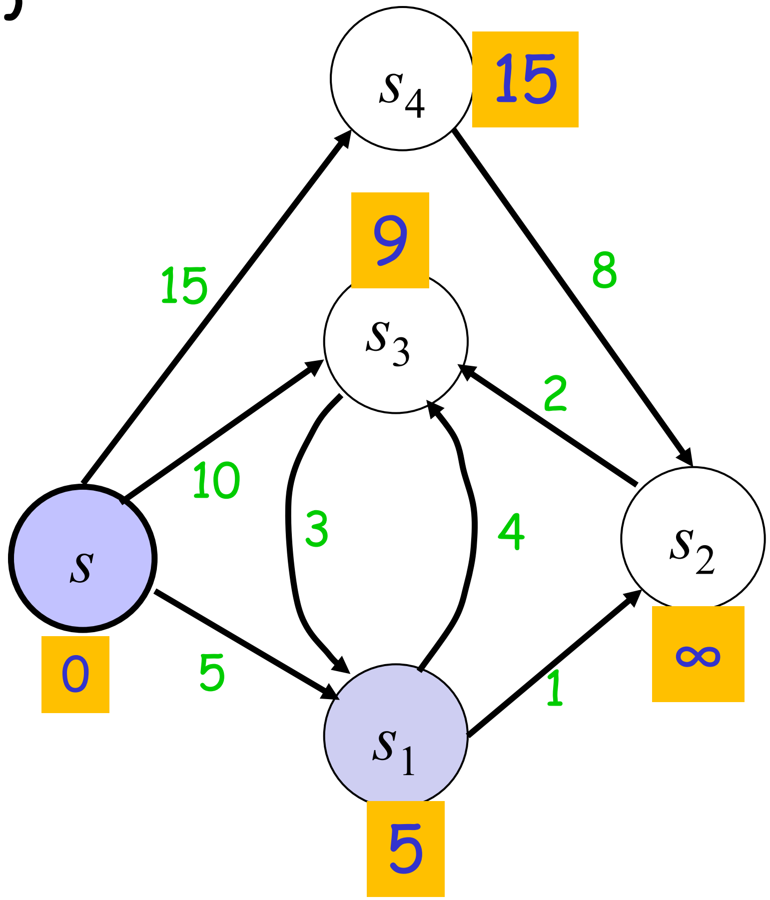
Relax(s_1,s_3)



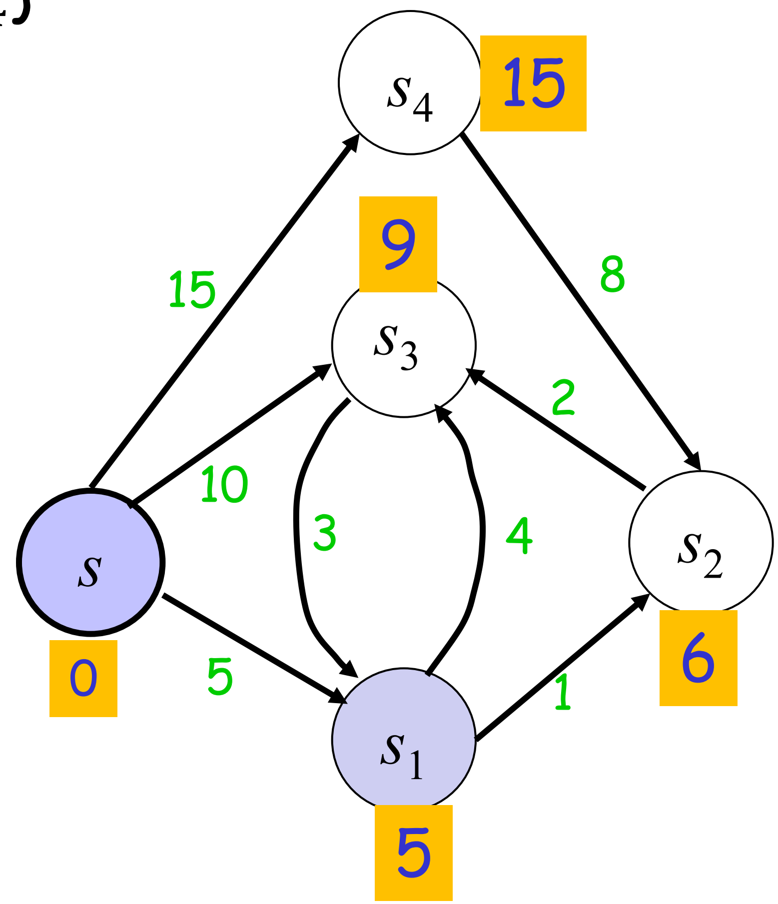
$S = \{s, s_1\}$ $Q=\{s_2, s_3, s_4\}$

Relax(s_1,s_3)

Relax(s_1,s_2)



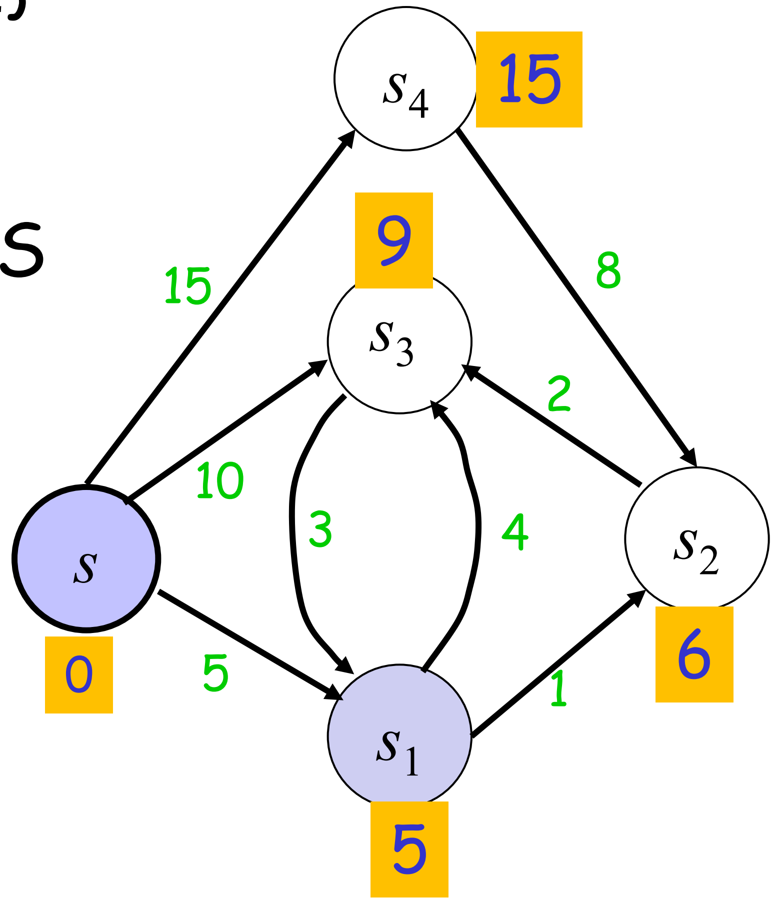
$S = \{s, s_1\}$ $Q=\{s_2, s_3, s_4\}$



Relax(s_1,s_3)
Relax(s_1,s_2)

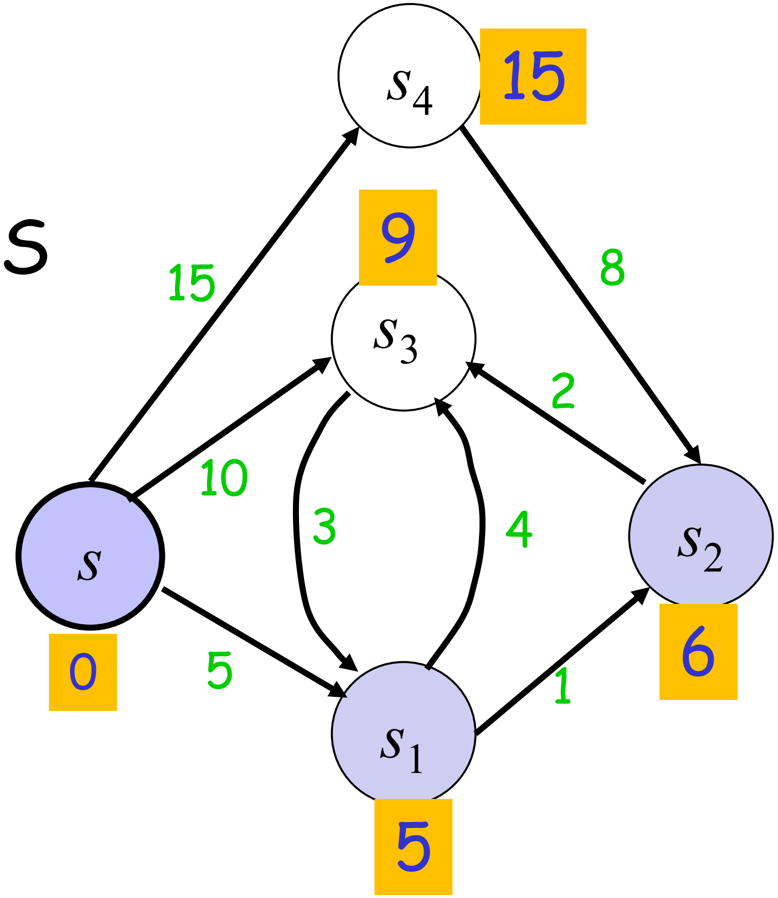
$S = \{s, s_1\}$ $Q=\{s_2, s_3, s_4\}$

Pick a vertex v∈Q with minimal d(v) and move it to S



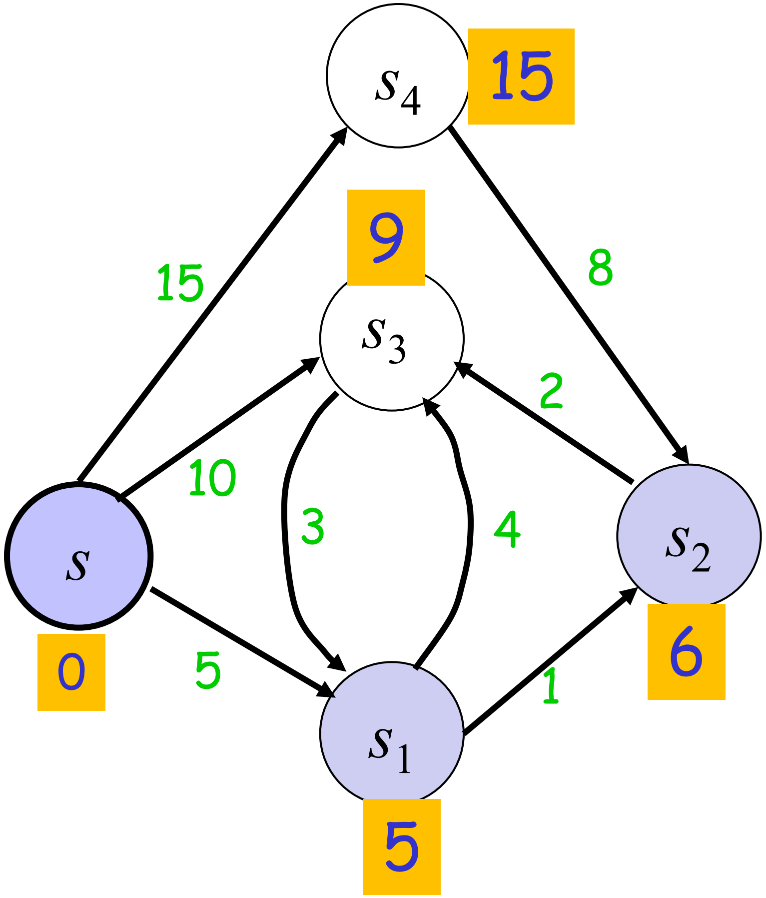
$S = \{s, s_1, s_2\}$ $Q=\{s_3, s_4\}$

Pick a vertex v∈Q with minimal d(v) and move it to S



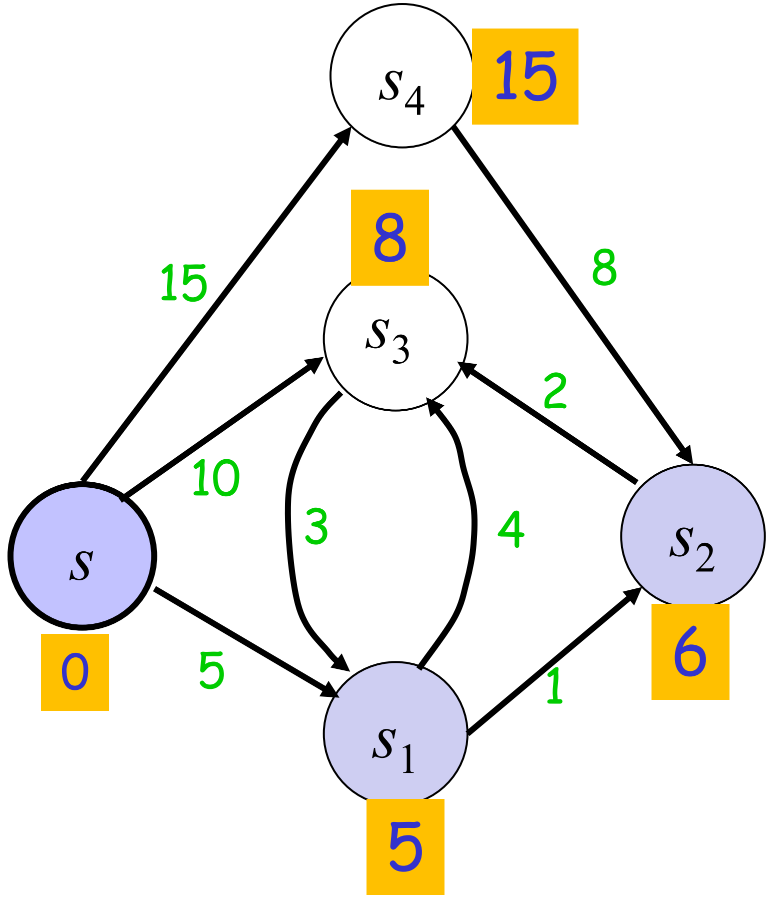
$S = \{s, s_1, s_2\}$ $Q=\{s_3, s_4\}$

Relax(s_2,s_3)



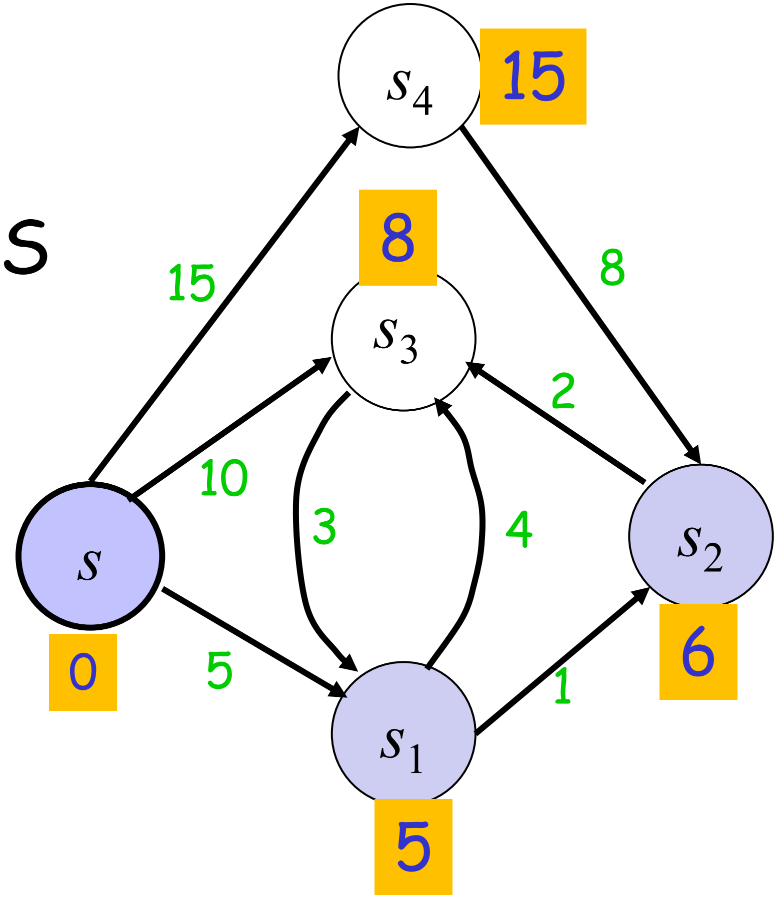
$S = \{s, s_1, s_2\}$ $Q = \{s_3, s_4\}$

Relax(s_2,s_3)



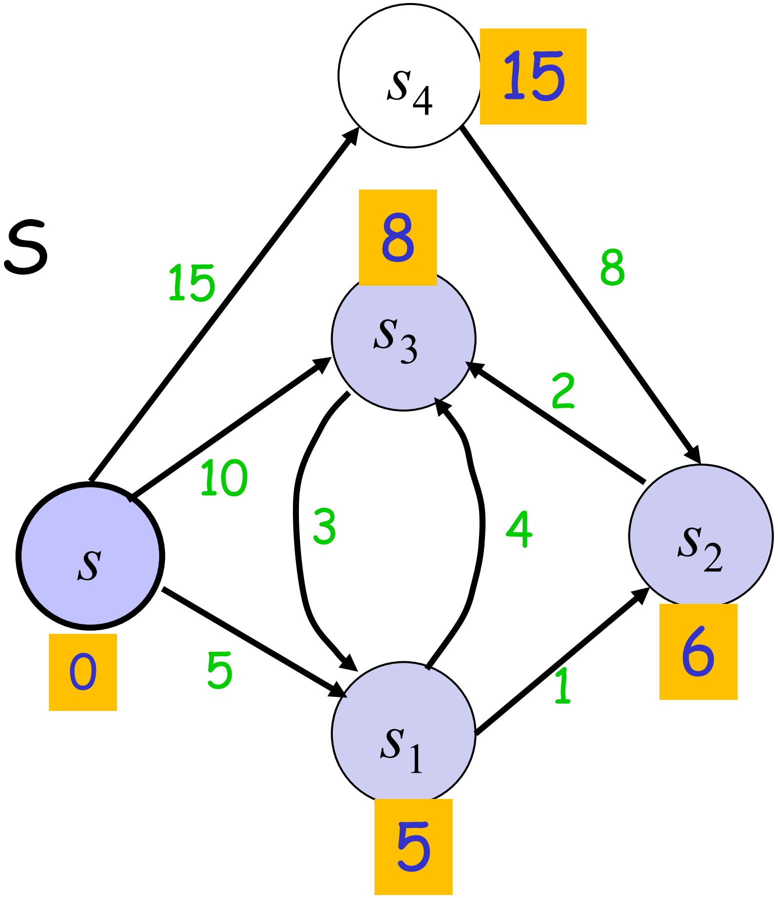
$S = \{s, s_1, s_2\}$ $Q = \{s_3, s_4\}$

Pick a vertex v∈Q with minimal d(v) and move it to S



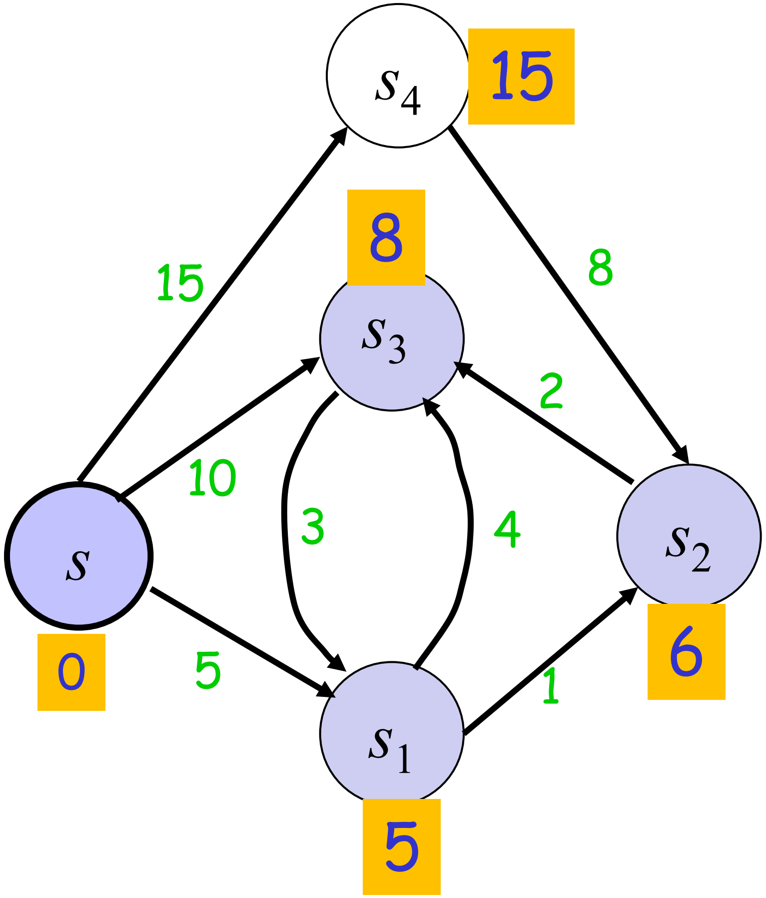
$S = \{s, s_1, s_2, s_3\}$ $Q=\{s_4\}$

Pick a vertex v∈Q with minimal d(v) and move it to S



$S = \{s, s_1, s_2, s_3\}$ $Q=\{s_4\}$

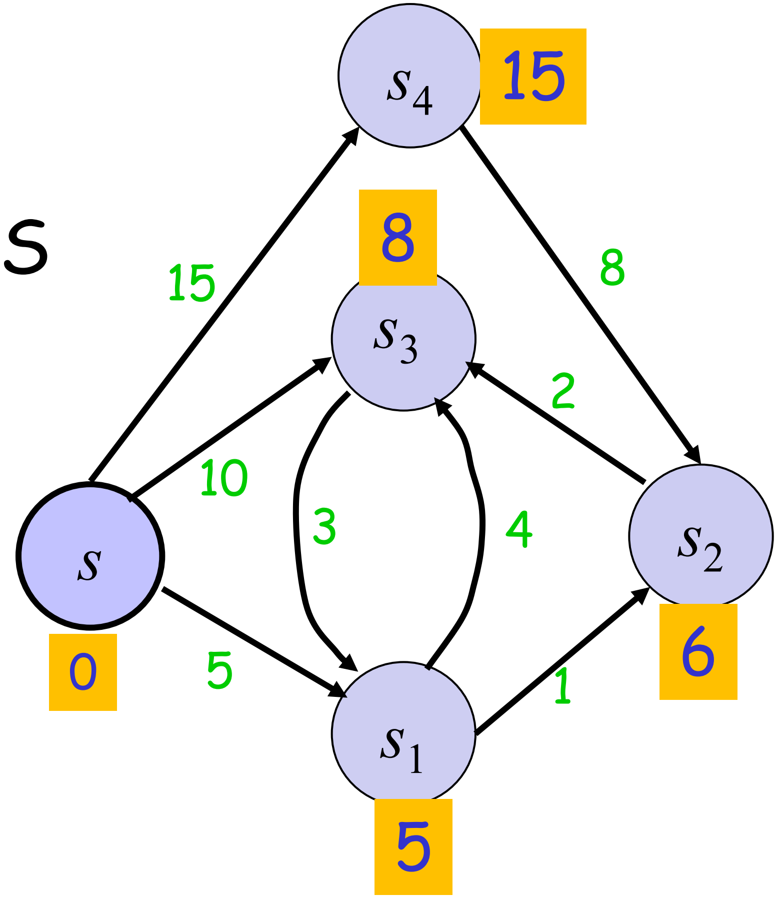
No edges to relax



$S = \{s, s_1, s_2, s_3, s_4\}$ $Q = \varnothing$

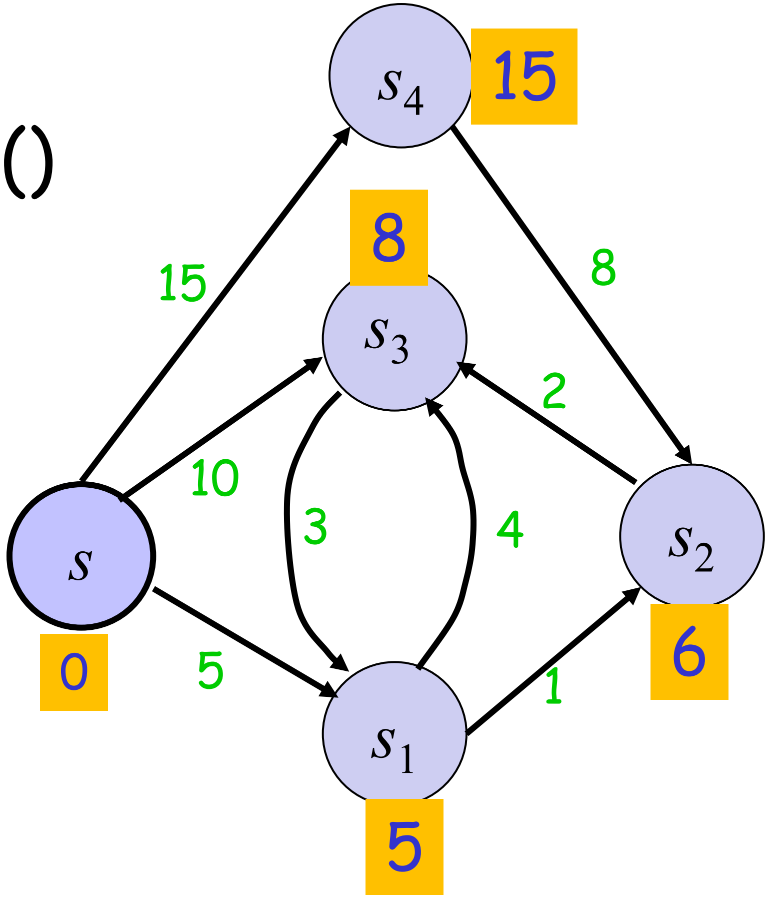
Pick a vertex $v \in Q$ with minimal d(v) and move it to S

No edges to relax

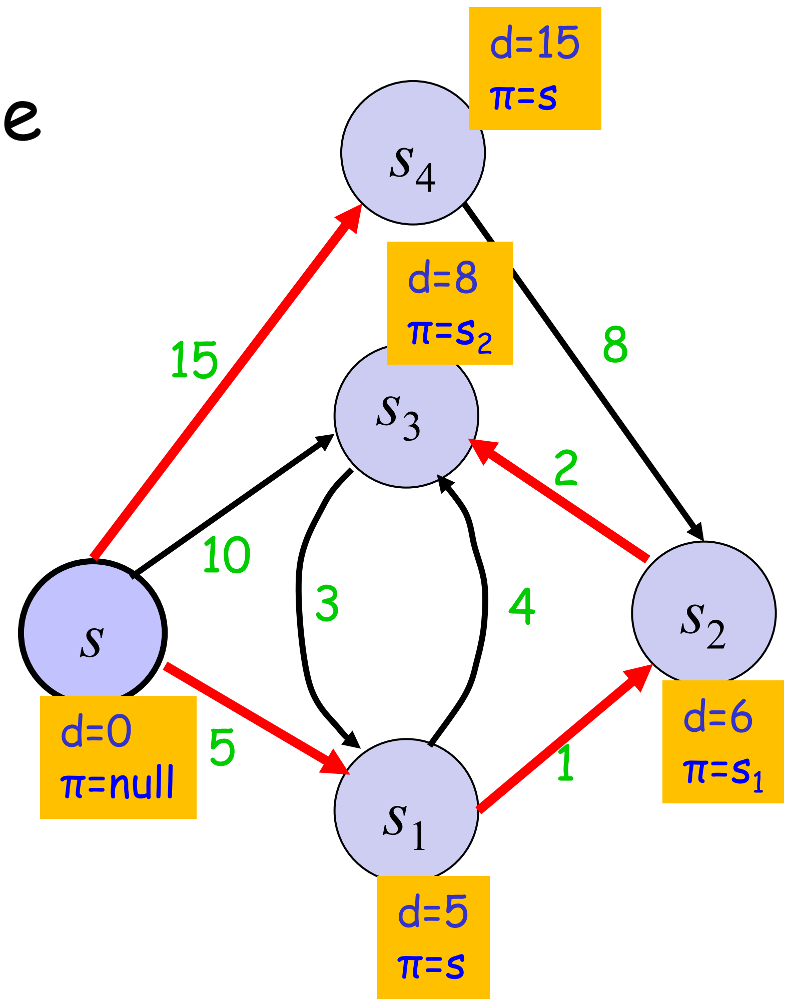


$S = \{s, s_1, s_2, s_3, s_4\}$ $Q = \varnothing$

At the end $Q = \varnothing$ and the d() values are the distances from s



- To reconstruct the shortest-paths, we also maintain for each node, the last node that relaxed it



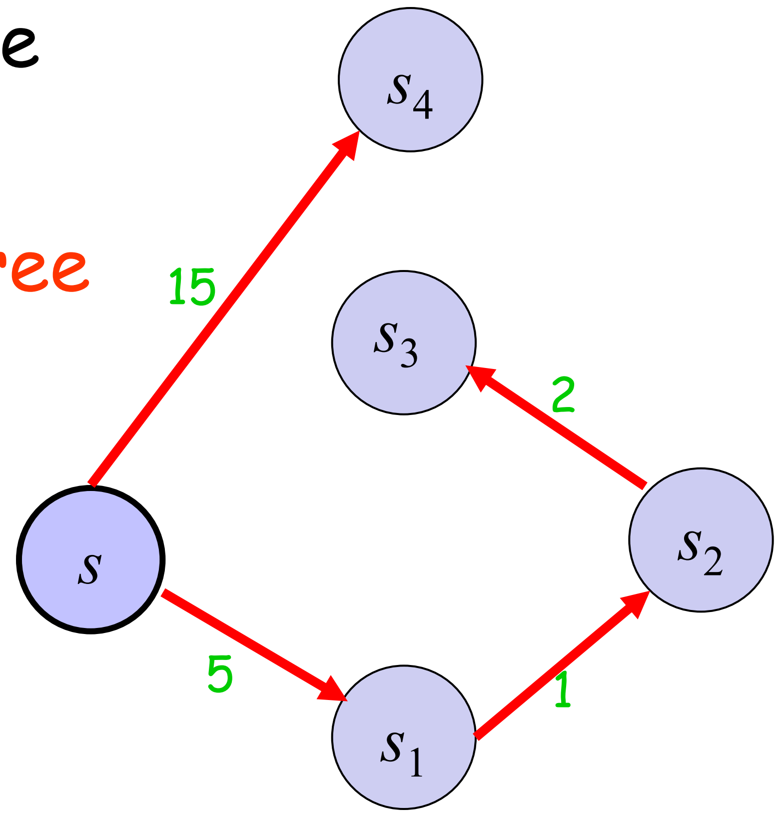
Relax(v,w)

If $d(w) > d(v) + w(v,w)$ then

$d(w) \leftarrow d(v) + w(v,w)$

$\pi(w) \leftarrow v$

- To reconstruct the shortest-paths, we also maintain for each node, the last node that relaxed it
- This is a shortest paths tree



Implementation of Dijkstra's algorithm

- We need to find the vertex with minimum d() in Q and remove it from Q
- In Relax, we need to decrease the d() values of vertices in Q

Required ADT

- Maintain items with keys subject to
  - $Q \leftarrow \text{Init}(V,E)$ — Once
  - Delete-min(Q) — $|V|=n$ times
  - Decrease-key$(x,Q,\Delta>0)$ — $\leq|E|=m$ times

Implementation using (W)AVL?
Implementation using binary heaps?

Both $O((n+m)\log n)$

Motivation

- Can we get rid of the mlogn?
- Yes, if we can improve Decrease-key to O(1) time
- Doesn't have to be worst case.
  - O(1) amortized is good enough
- Then total running time: O(m + nlogn)

Roadmap

	Binary Heaps	**Binomial Heaps**	**Lazy Binomial Heaps**	**Fibonacci Heaps**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld	—	$O(\log n)$	$O(1)$	$O(1)$

Worst case | Amortized

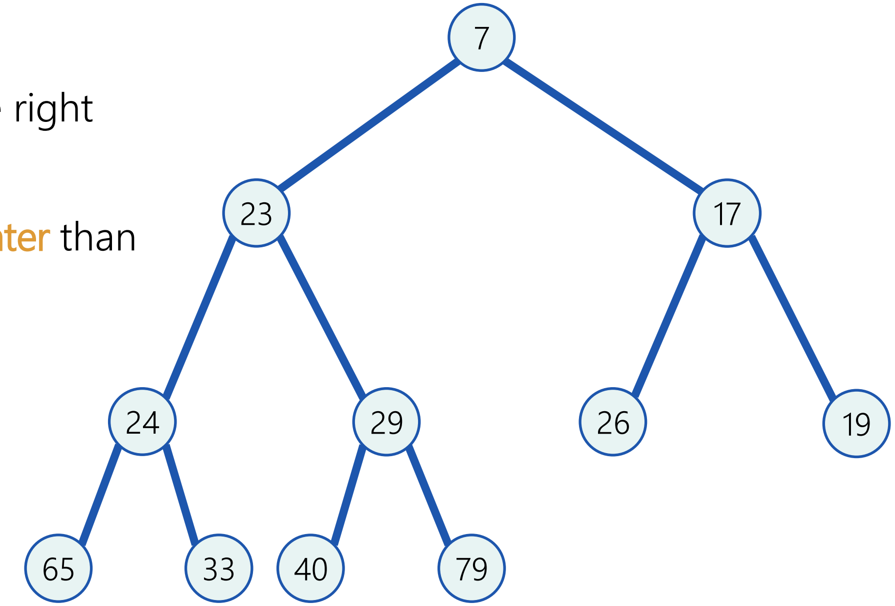
* Cheaper to build from an unsorted array.

Binary Heap

Description and Implementation

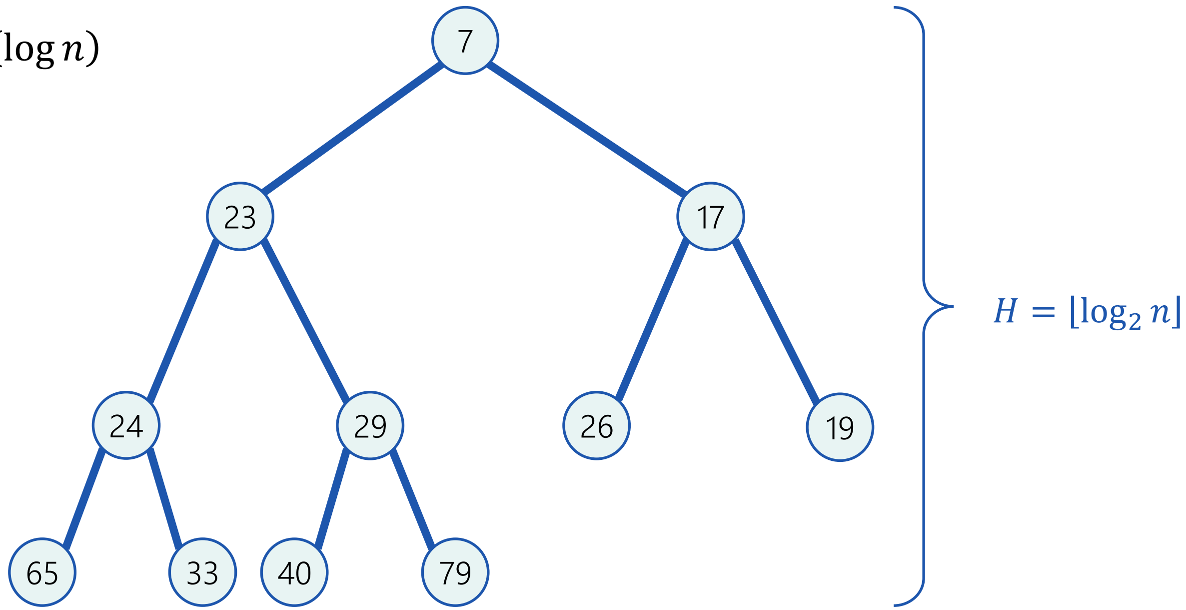
(Binary) Min-Heap

- Structure: an almost complete binary tree:
  all levels full, bottom level may lack nodes on the right
- Heap order: the keys at the children of v are greater than the one at v
- Binary Max-heap : similar, but keys at the children of v are smaller than the one at v



(Binary) Heap

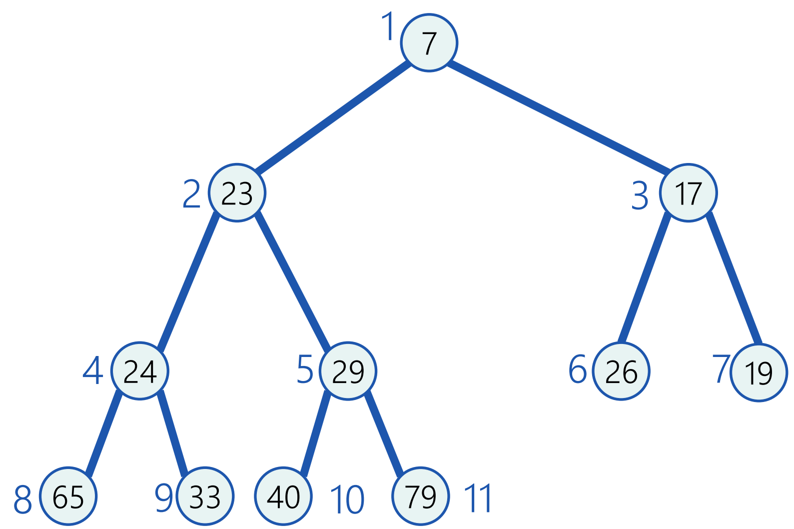
- Tree height is $O(\log n)$



Array Representation

Representing a heap with an array

- Efficient (less I/O operations)
- Get the elements from top to bottom, from left to right
- Root = minimum: Q[1]*
- Bottom right: last element in array



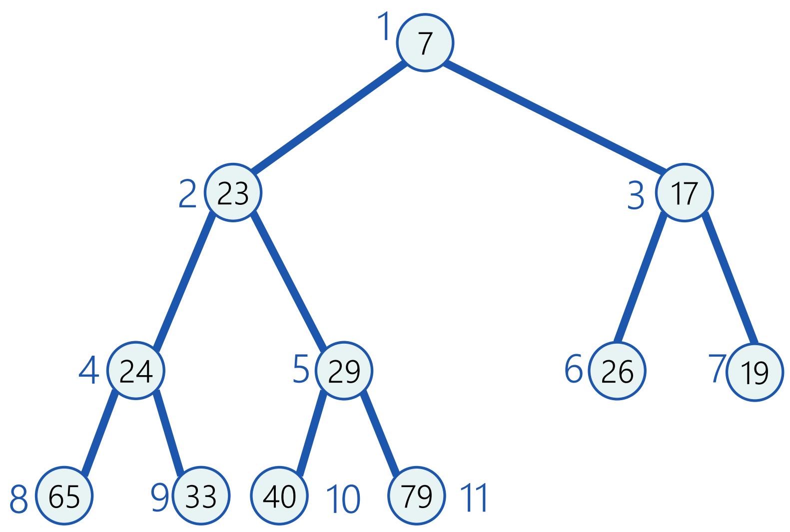
	1	2	3	4	5	6	7	8	9	10	11
Q	7	23	17	24	29	26	19	65	33	40	79

* We start from 1, not 0, for convenience (see soon why)

Array Representation

Representing a heap with an array

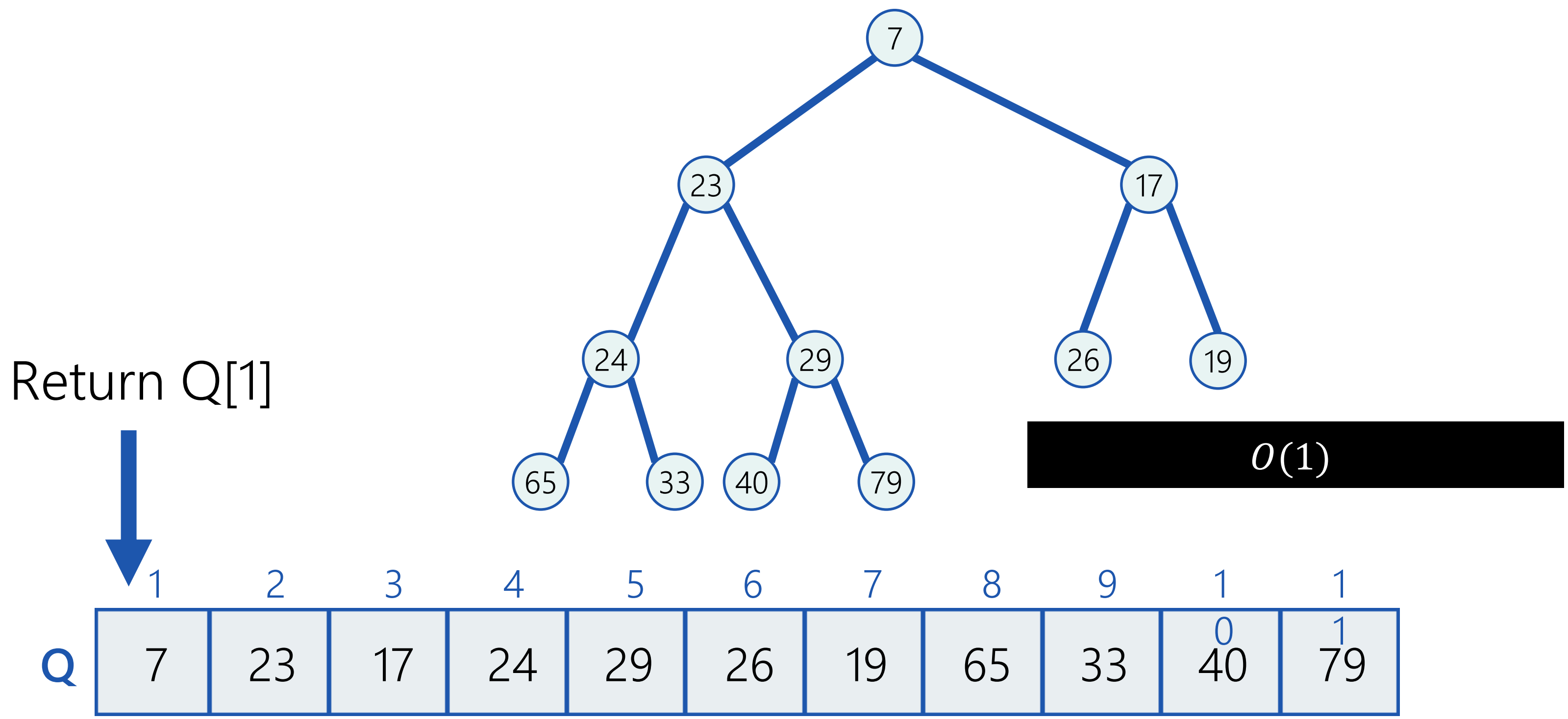
- Left(i): $2i$
- Right(i): $2i+1$
- Parent(i): $\left\lfloor \frac{i}{2} \right\rfloor$
- Concise, Fast



	1	2	3	4	5	6	7	8	9	10	11
Q	7	23	17	24	29	26	19	65	33	40	79

Finding the Minimum

Finding the Minimum



Applications and Required ADT

Maintain items (x) with keys (x.key)

Required Operations:

- Insert(x, Q)
- min(Q)
- Delete-min(Q)

$O(1)$

- Decrease-key(x, Q, Δ)
- Delete(x, Q)

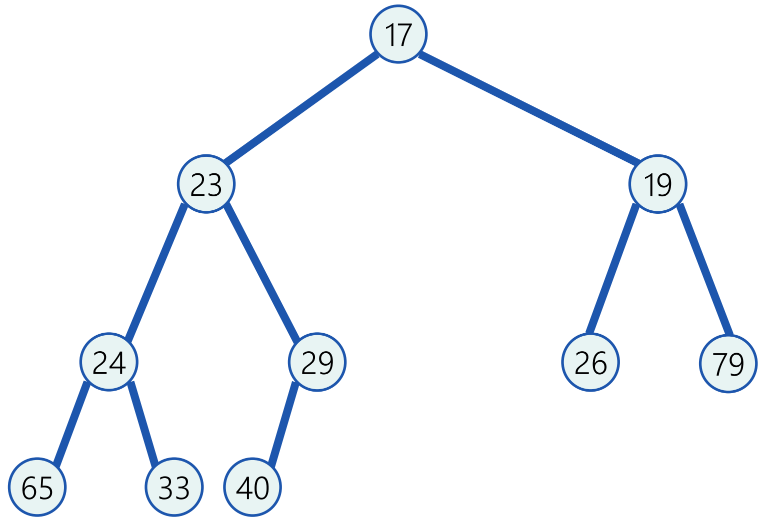
Non-Required Operations:

- Find(x, Q)

Insertion

Inserting an Item

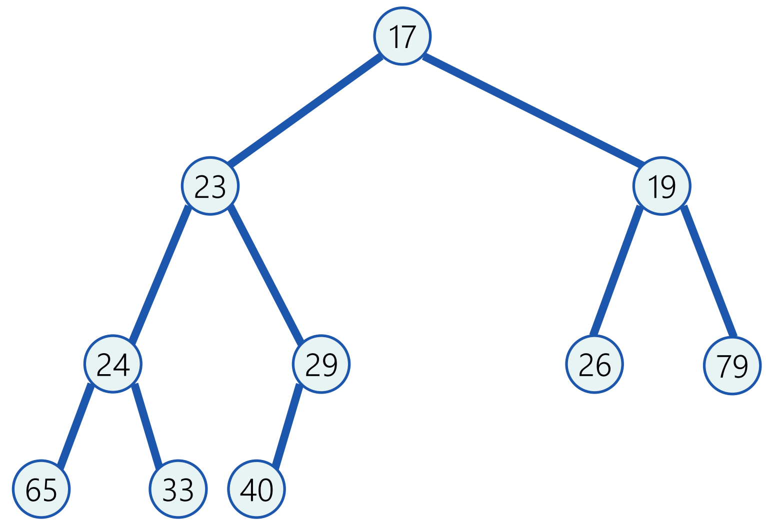
Function Insert (x ,Q)
size(Q) ← size(Q) + 1
Q[size(Q)] ← x.key and a pointer to x
Heapify-up(Q, size(Q))



	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40			

Inserting an Item

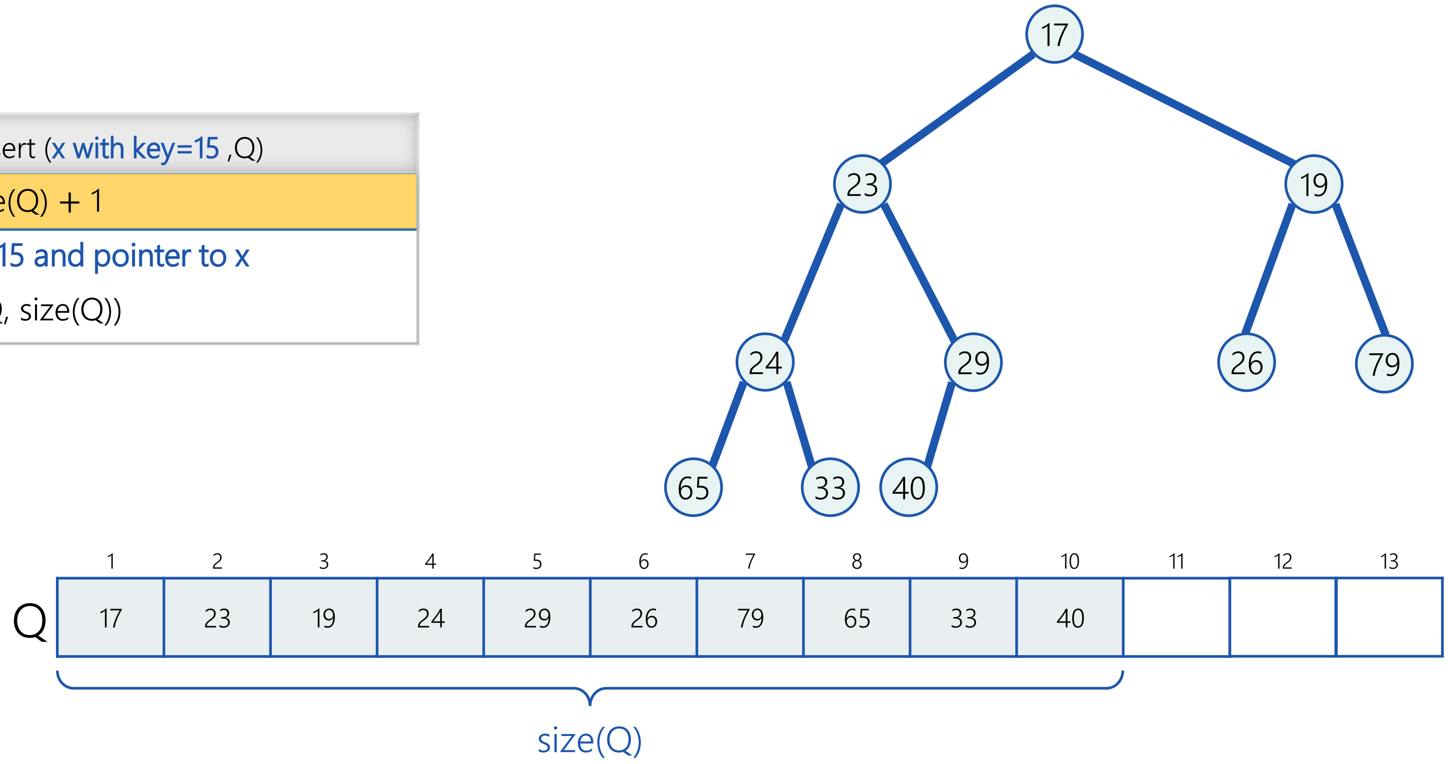
Function Insert (x with key=15 ,Q)
size(Q) ← size(Q) + 1
Q[size(Q)] ← 15 and pointer to x
Heapify-up(Q, size(Q))



	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40			

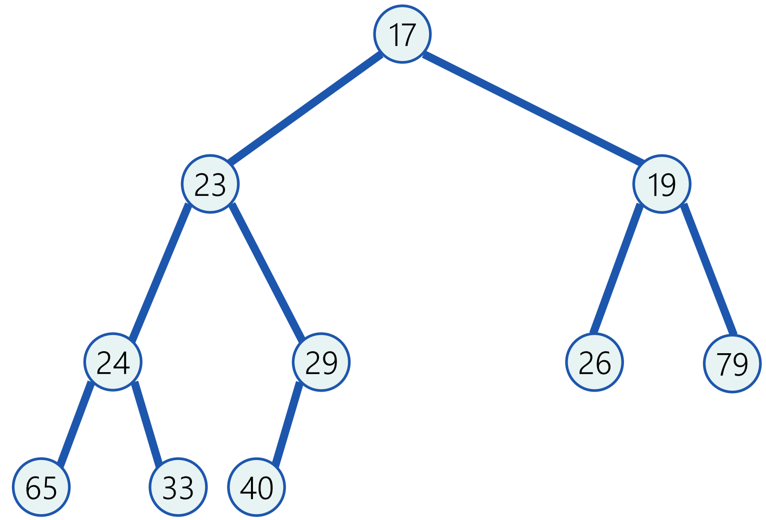
Inserting an Item

Function Insert (x with key=15 ,Q)
size(Q) ← size(Q) + 1
Q[size(Q)] ← 15 and pointer to x
Heapify-up(Q, size(Q))



Inserting an Item

Function Insert (x with key=15 ,Q)
size(Q) ← size(Q) + 1
Q[size(Q)] ← 15 and pointer to x
Heapify-up(Q, size(Q))

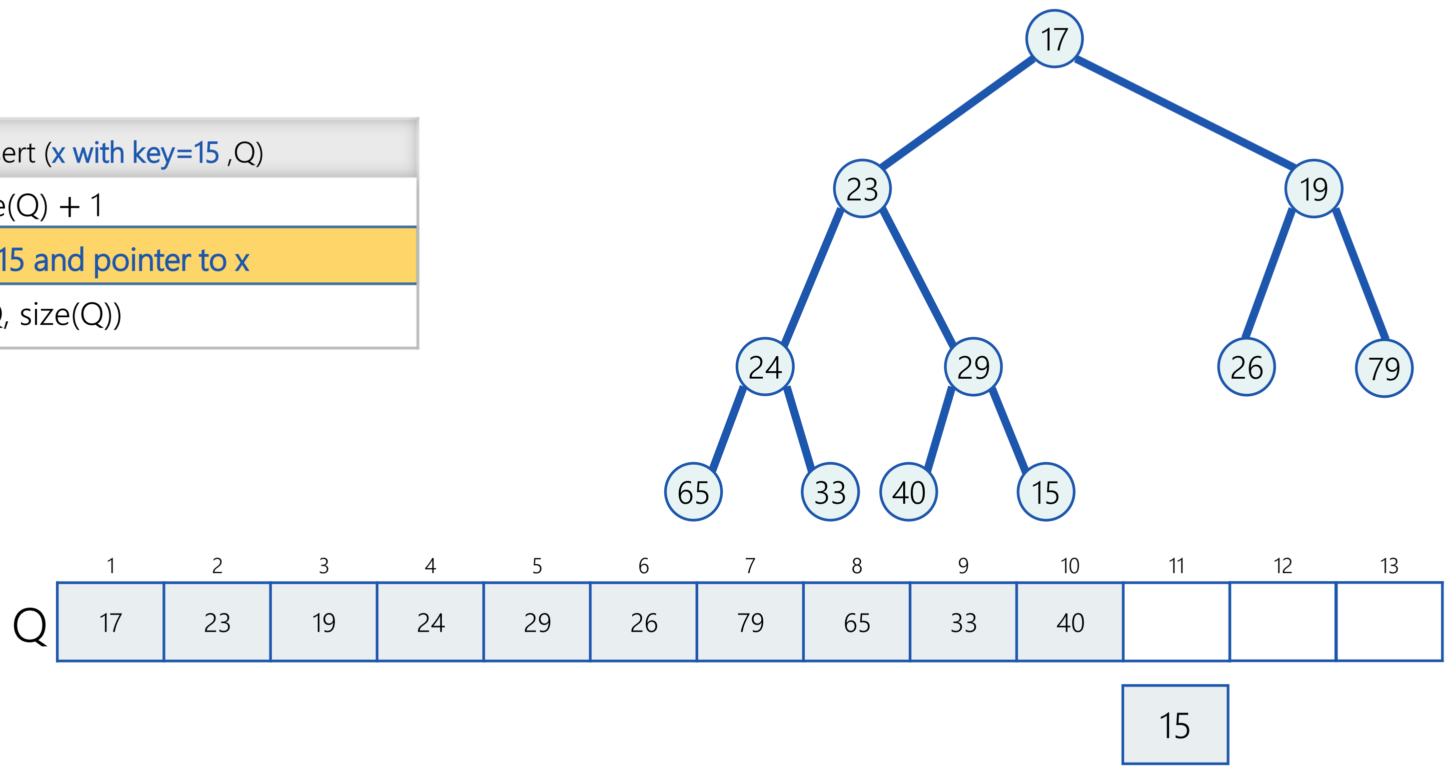


	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40			

size(Q)

Inserting an Item

Function Insert (x with key=15 ,Q)
size(Q) ← size(Q) + 1
Q[size(Q)] ← 15 and pointer to x
Heapify-up(Q, size(Q))

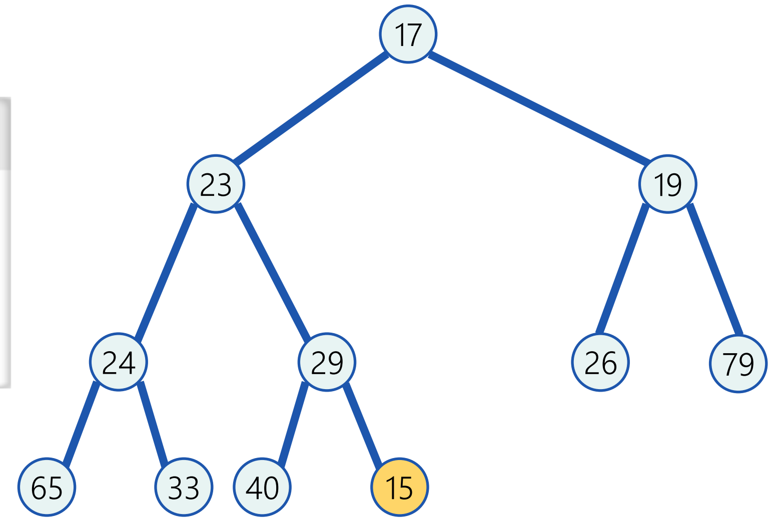


Inserting an Item

Heapify-up

Function Heapify-up(Q, i)

```
while i > 1 and Q[i] < Q[parent(i)] do
    Q[i] ↔ Q[parent(i)]
    i ← parent(i)
```



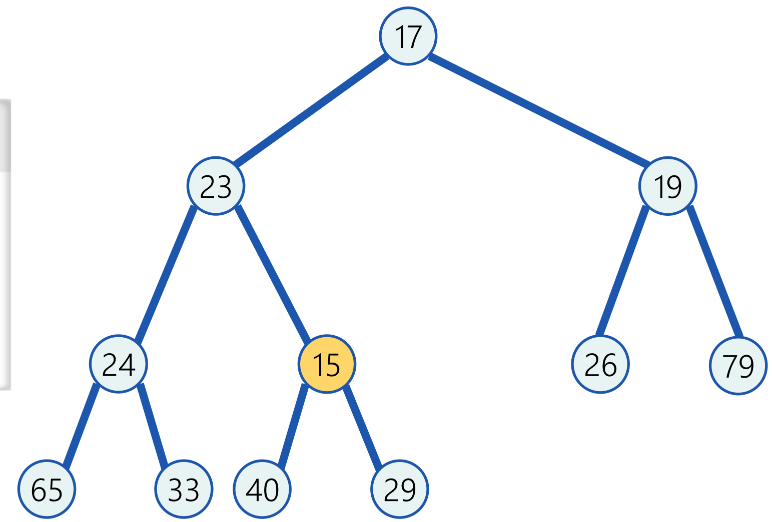
	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40	15		

Inserting an Item

Heapify-up

Function Heapify-up(Q, i)

```
while i > 1 and Q[i] < Q[parent(i)] do
    Q[i] ↔ Q[parent(i)]
    i ← parent(i)
```



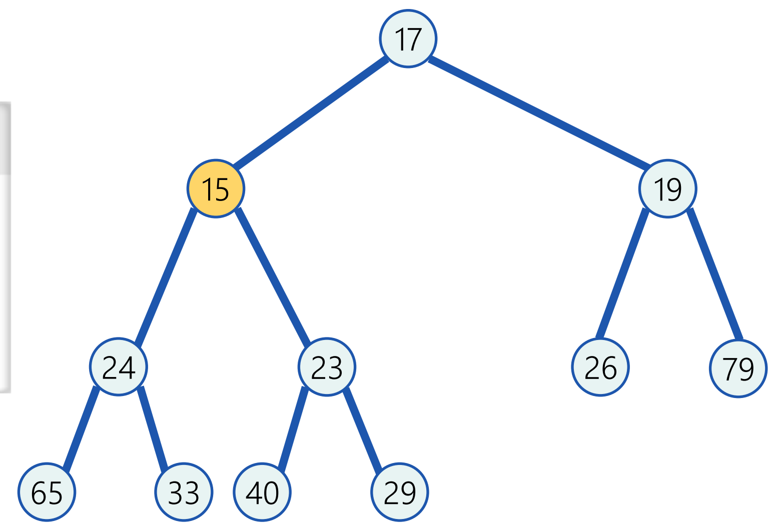
	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	15	26	79	65	33	40	29		

Inserting an Item

Heapify-up

Function Heapify-up(Q, i)

```
while i > 1 and Q[i] < Q[parent(i)] do
    Q[i] ↔ Q[parent(i)]
    i ← parent(i)
```



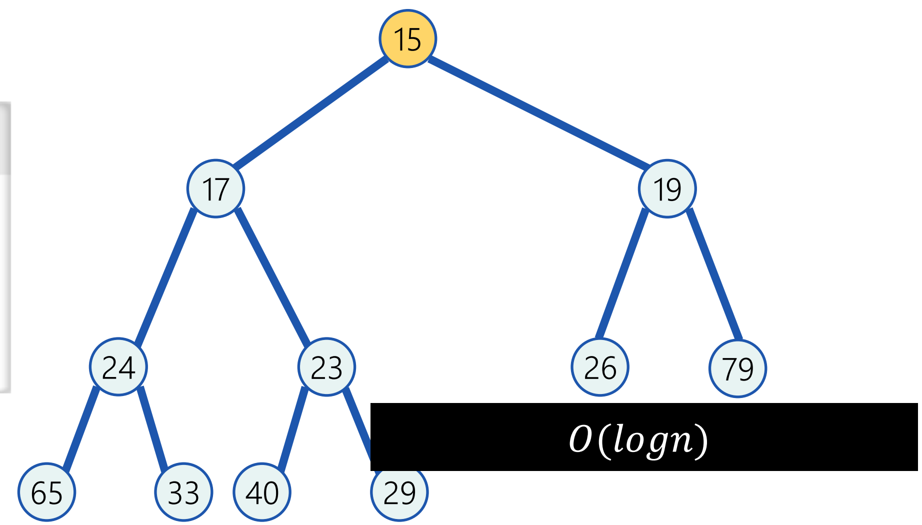
	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	15	19	24	23	26	79	65	33	40	29		

Inserting an Item

Heapify-up

Function Heapify-up(Q, i)

```
while i > 1 and Q[i] < Q[parent(i)] do
    Q[i] ↔ Q[parent(i)]
    i ← parent(i)
```



	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	15	17	19	24	23	26	79	65	33	40	29		

Applications and Required ADT

Maintain items (x) with keys (x.key)

Required Operations:

- Insert(x, Q) $O(logn)$
- min(Q) $O(1)$
- Delete-min(Q)

- Decrease-key(x, Q, Δ)
- Delete(x, Q)

Non-Required Operations:

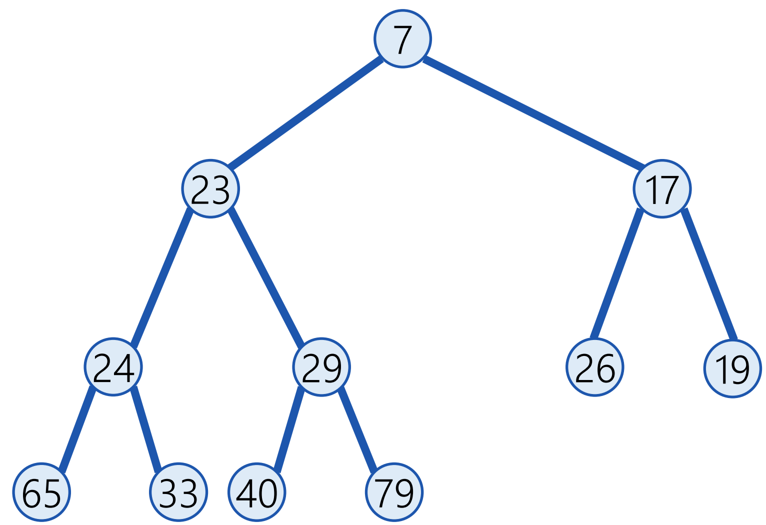
- Find(x, Q)

Deletion of the Minimum

Delete the minimum

Place last element at root:

Q[1] ← Q[size(Q)]
size(Q) ← size(Q)-1

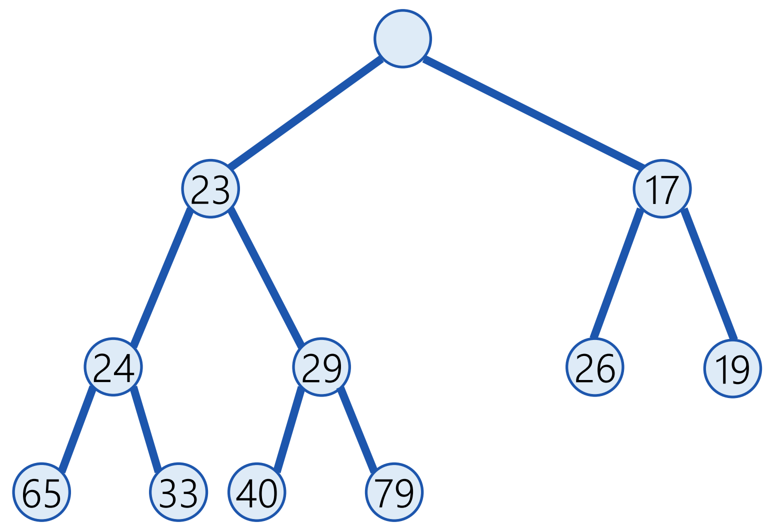


	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	7	23	17	24	29	26	19	65	33	40	79		

Delete the minimum

Place last element at root:

Q[1] ← Q[size(Q)]
size(Q) ← size(Q)-1



	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	7	23	17	24	29	26	19	65	33	40	79		

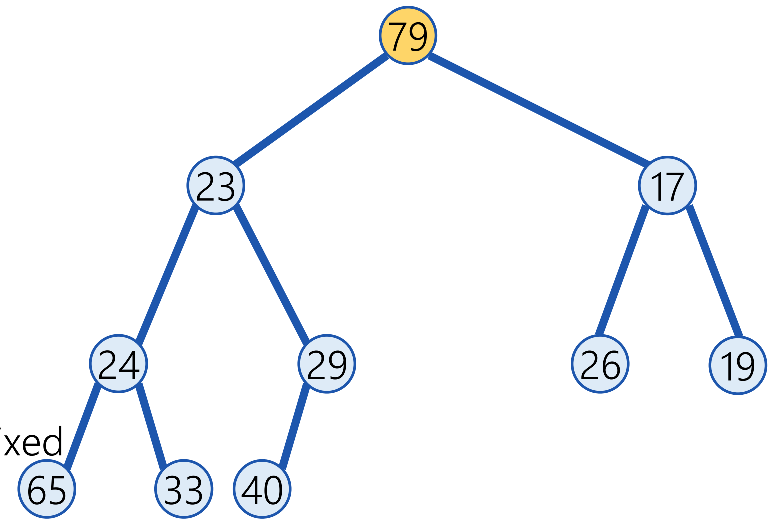
Delete the minimum

Place last element at root:

Q[1] ← Q[size(Q)]
size(Q) ← size(Q)-1

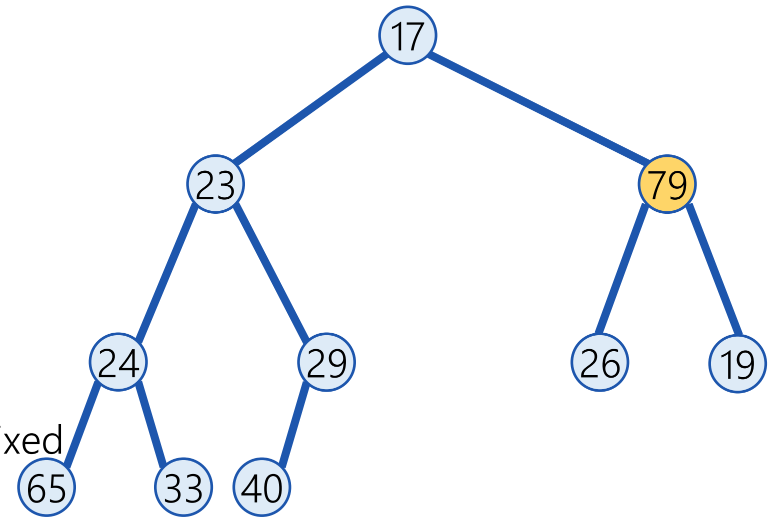
Fix: **Heapify-Down**

exchange with smallest child until fixed



	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	79	23	17	24	29	26	19	65	33	40			

Delete the minimum
Heapify-Down

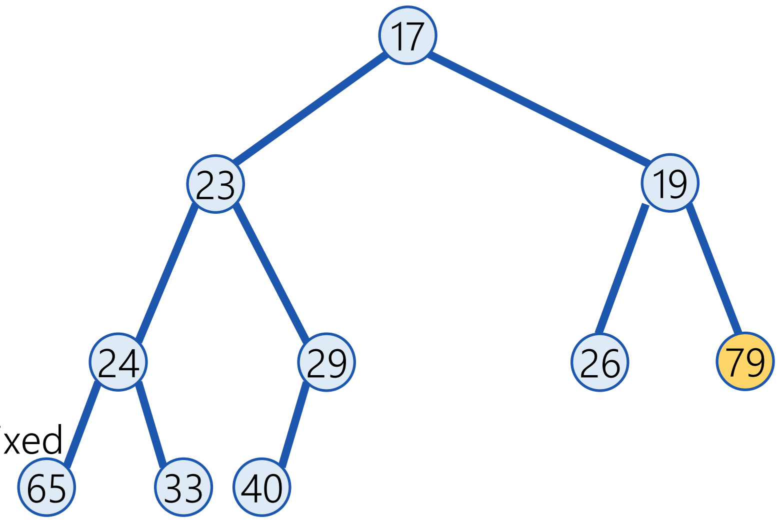


Fix: Heapify-Down

exchange with smallest child until fixed

	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	79	24	29	26	19	65	33	40			

Delete the minimum
Heapify-Down



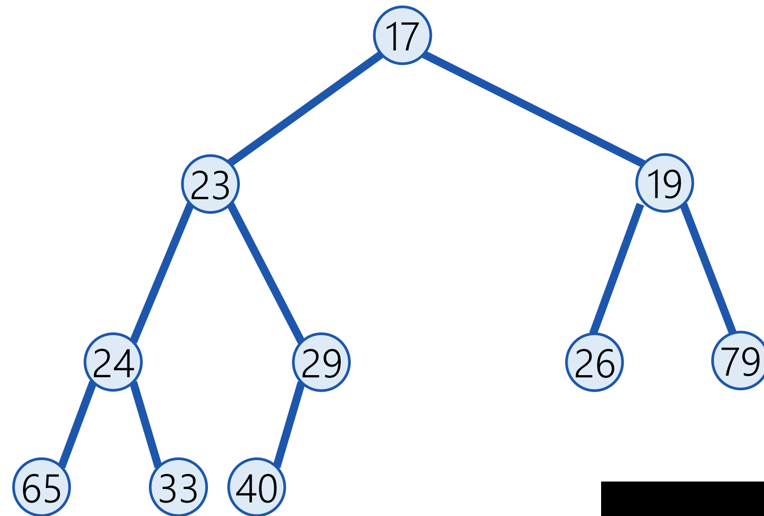
Fix: **Heapify-Down**

exchange with smallest child until fixed

	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40			

Delete the minimum

Heapify-Down



$O(logn)$

	1	2	3	4	5	6	7	8	9	10	11	12	13
Q	17	23	19	24	29	26	79	65	33	40			

Delete-min
Heapify-Down

```
Function Heapify-down(Q, i )
    l ← left(i)
    r ← right(i)
    smallest ← i
    if l < size(Q) and Q[l] < Q[smallest]
        then smallest ← l
    if r < size(Q) and Q[r] < Q[smallest]
        then smallest ← r
    if smallest > i then
        Q[i] ↔ Q[smallest]
        Heapify-down(Q, smallest)
```

Applications and Required ADT

Maintain items (x) with keys (x.key)

Required Operations:

• Insert(x, Q)	$O(logn)$
• min(Q)	$O(1)$
• Delete-min(Q)	$O(logn)$

- Decrease-key(x, Q, Δ)
- Delete(x, Q)

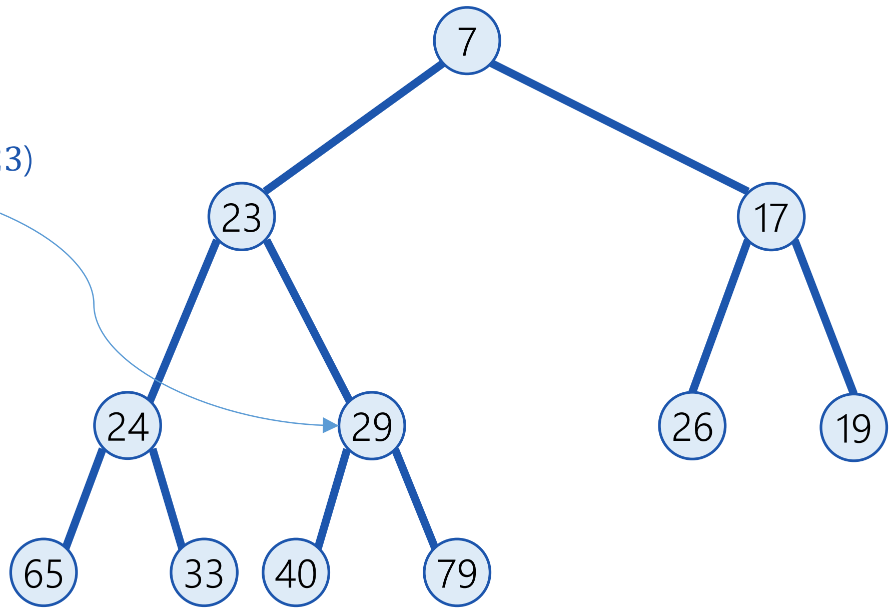
Non-Required Operations:

- Find(x, Q)

Decrease Key

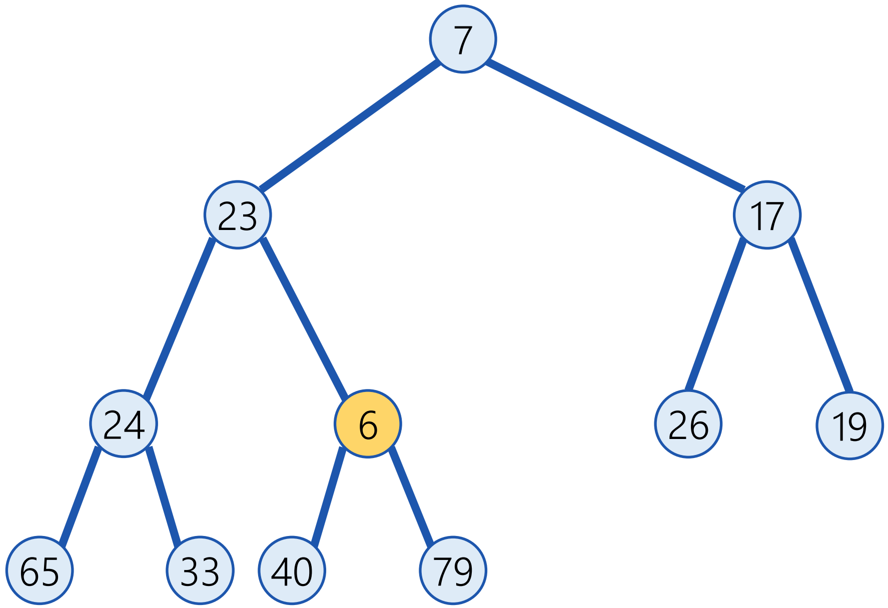
Decrease key

Decrease-key(x, Q, $\Delta = 23$)



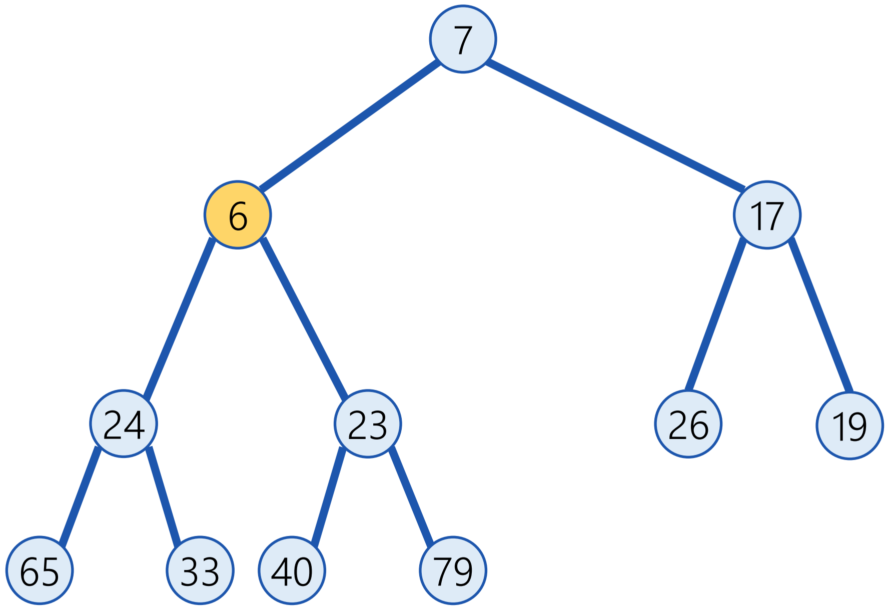
Decrease key

Now Heapify-Up



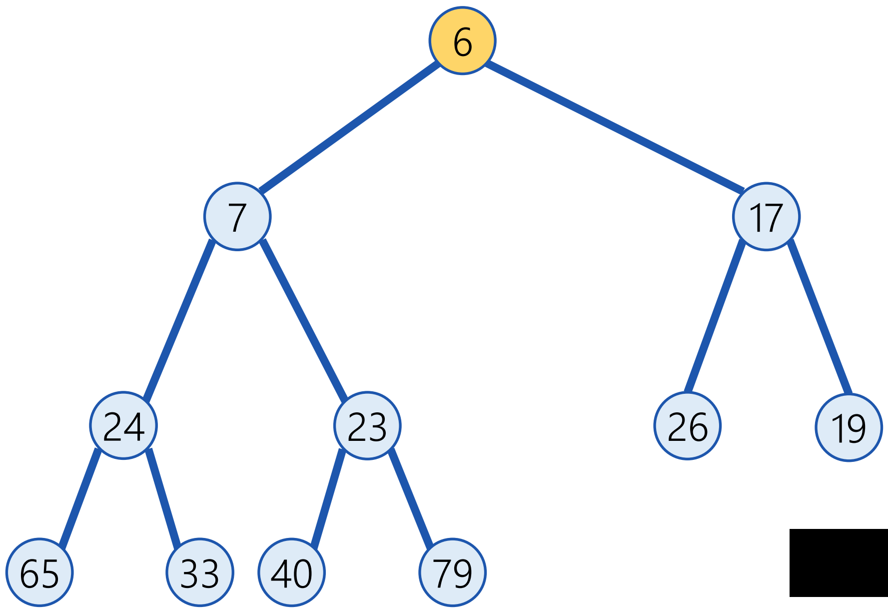
Decrease key

Now Heapify-Up



Decrease key

Now Heapify-Up



$O(logn)$

Applications and Required ADT

Maintain items (x) with keys (x.key)

Required Operations:

- Insert(x, Q) — $O(logn)$
- min(Q) — $O(1)$
- Delete-min(Q) — $O(logn)$

- Decrease-key(x, Q, Δ) — $O(logn)$
- Delete(x, Q)

Non-Required Operations:

- Find(x, Q)

Applications and Required ADT

Summary

Maintain items (x) with keys (x.key)

Required Operations:

• Insert(x, Q)	$O(log n)$
• min(Q)	$O(1)$
• Delete-min(Q)	$O(log n)$
• Decrease-key(x, Q, Δ)	$O(log n)$
• Delete(x, Q)	$O(log n)$

Non-Required Operations:

- Find(x, Q)

Creating a Heap

How Can We Do It Efficiently?

Turn an array into a heap

Naïve way: n inserts

$O(nlogn)$ worst case

We can do better!

79	65	26	24	19	15	29	23	33	40	7

Insert(79, Q)

Insert(65, Q)

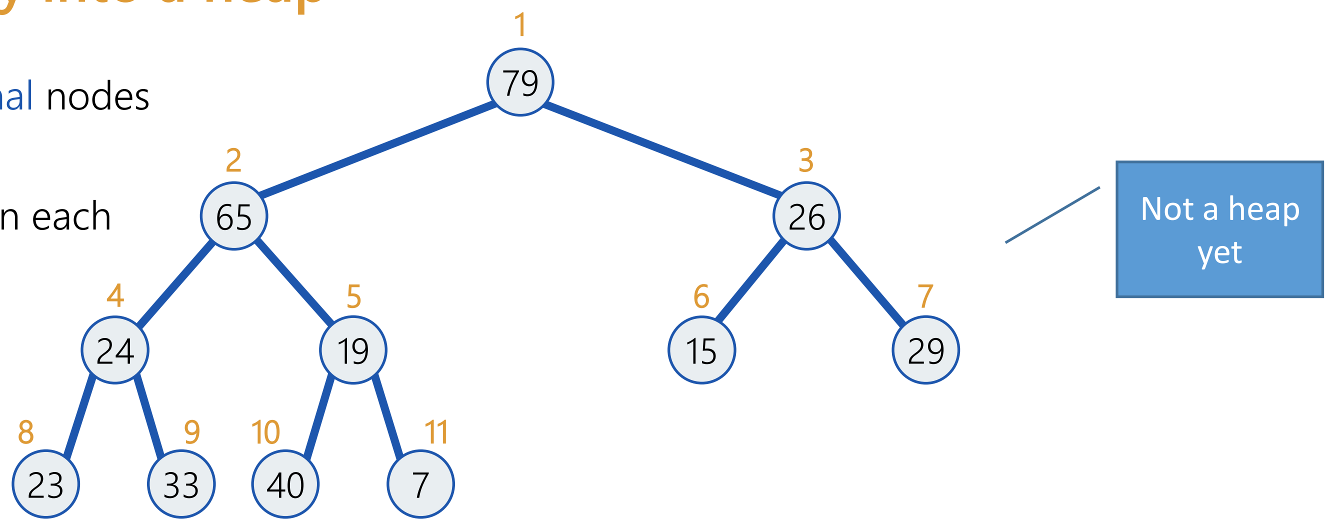
…

$$time = \sum_{i=2}^{n} O(\log i) = O(nlogn)$$

Turn an array into a heap

Iterate over internal nodes bottom up,

and Heapify-Down each



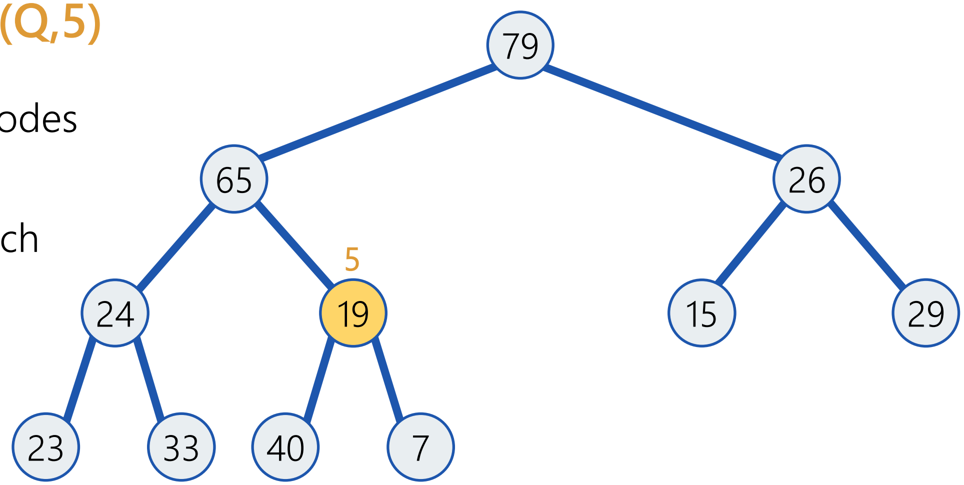
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	24	19	15	29	23	33	40	7

Turn an array into a heap

Heapify-down(Q,5)

Iterate over internal nodes bottom up,

and Heapify-Down each



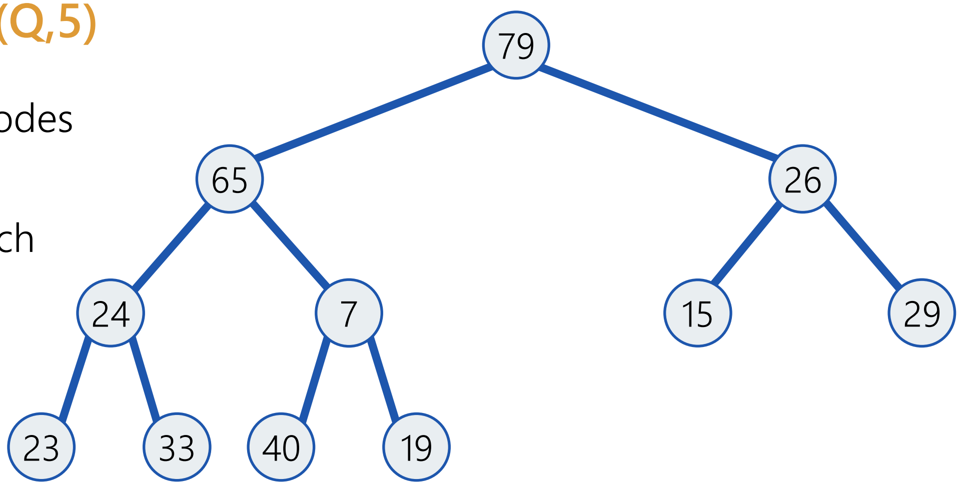
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	24	19	15	29	23	33	40	7

Turn an array into a heap

Heapify-down(Q,5)

Iterate over internal nodes bottom up,

and Heapify-Down each



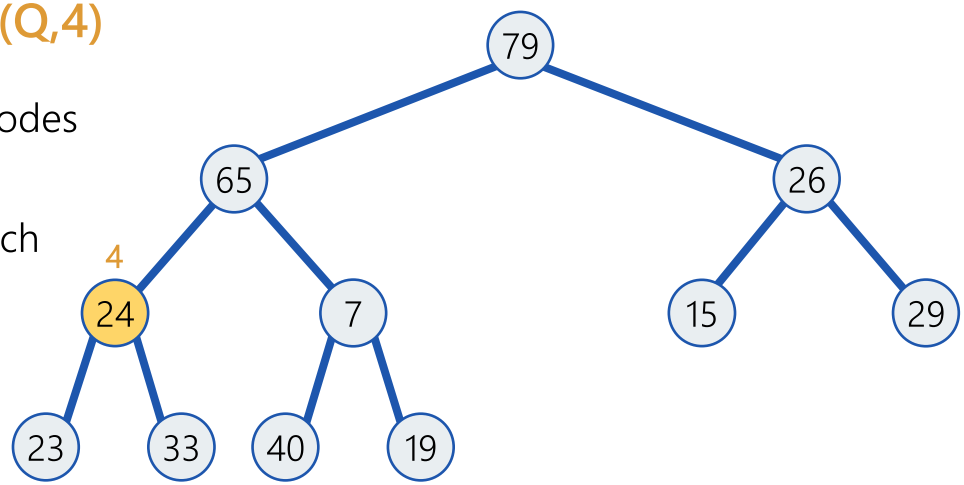
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	24	7	15	29	23	33	40	19

Turn an array into a heap

Heapify-down(Q,4)

Iterate over internal nodes bottom up,

and Heapify-Down each



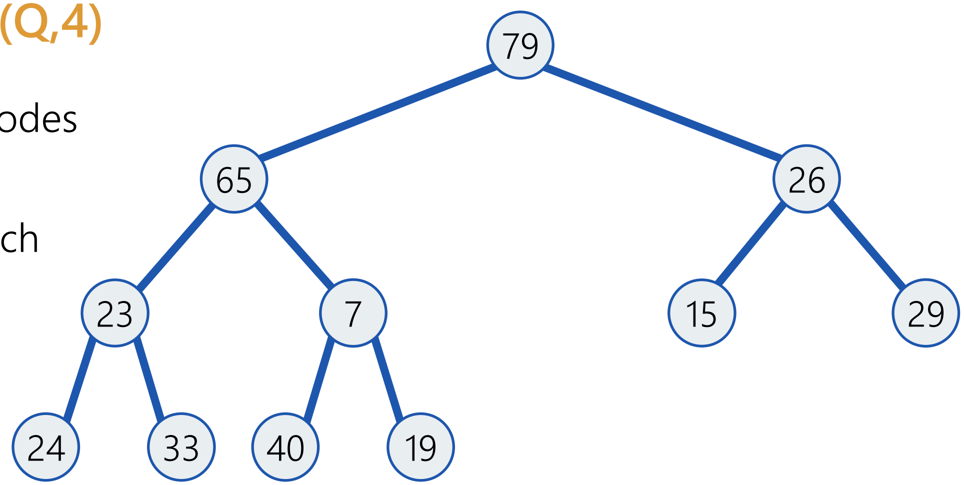
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	24	7	15	29	23	33	40	19

Turn an array into a heap

Heapify-down(Q,4)

Iterate over internal nodes bottom up,

and Heapify-Down each



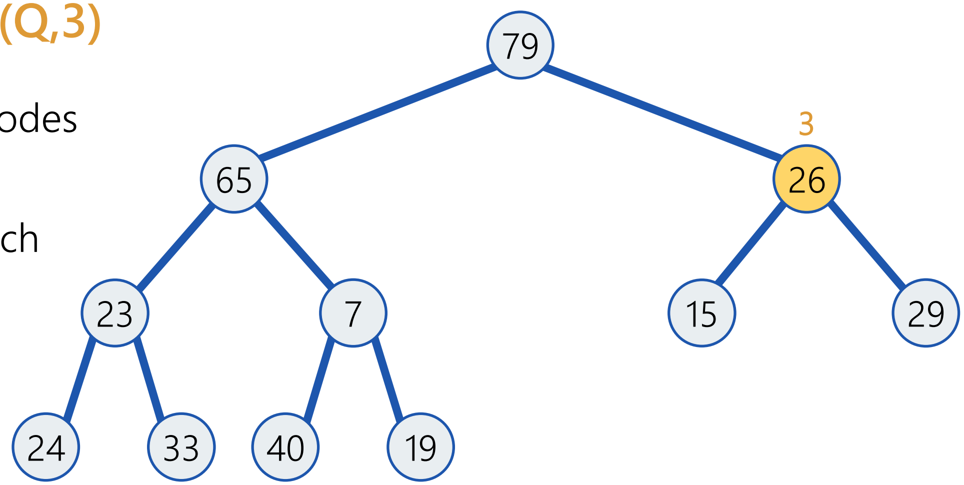
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	23	7	15	29	24	33	40	19

Turn an array into a heap

Heapify-down(Q,3)

Iterate over internal nodes bottom up,

and Heapify-Down each



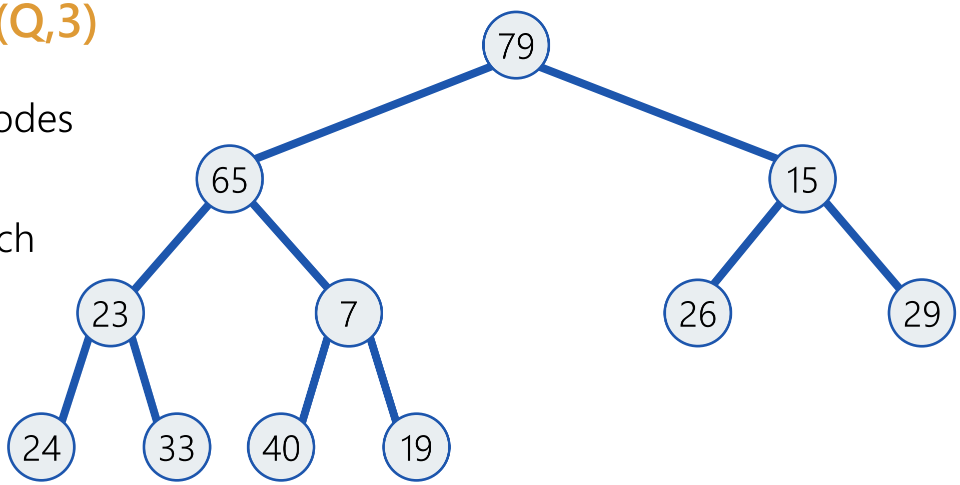
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	26	23	7	15	29	24	33	40	19

Turn an array into a heap

Heapify-down(Q,3)

Iterate over internal nodes bottom up,

and Heapify-Down each



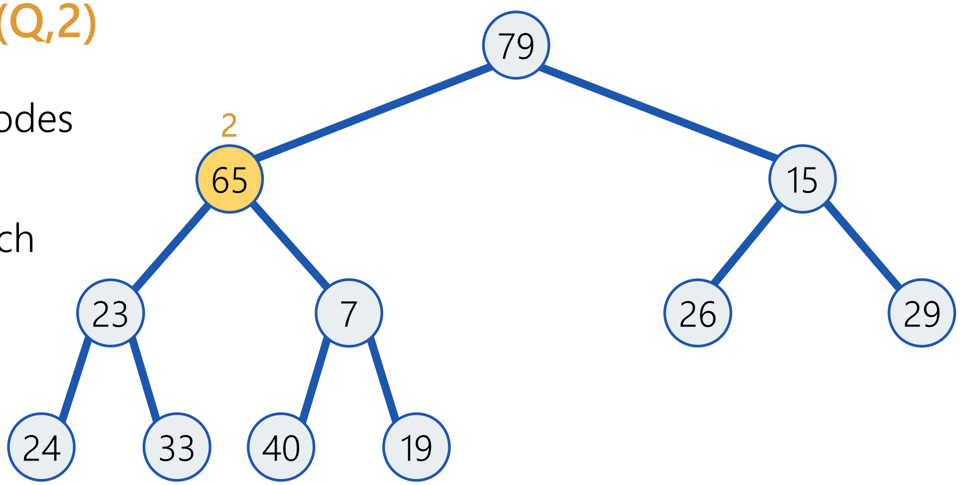
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	15	23	7	26	29	24	33	40	19

Turn an array into a heap

Heapify-down(Q,2)

Iterate over internal nodes bottom up,

and Heapify-Down each



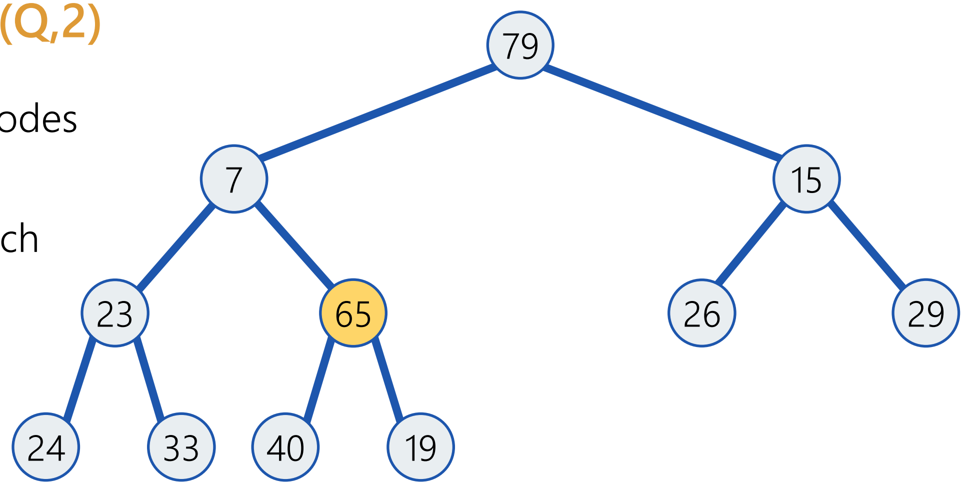
	1	2	3	4	5	6	7	8	9	10	11
Q	79	65	15	23	7	26	29	24	33	40	19

Turn an array into a heap

Heapify-down(Q,2)

Iterate over internal nodes bottom up,

and Heapify-Down each



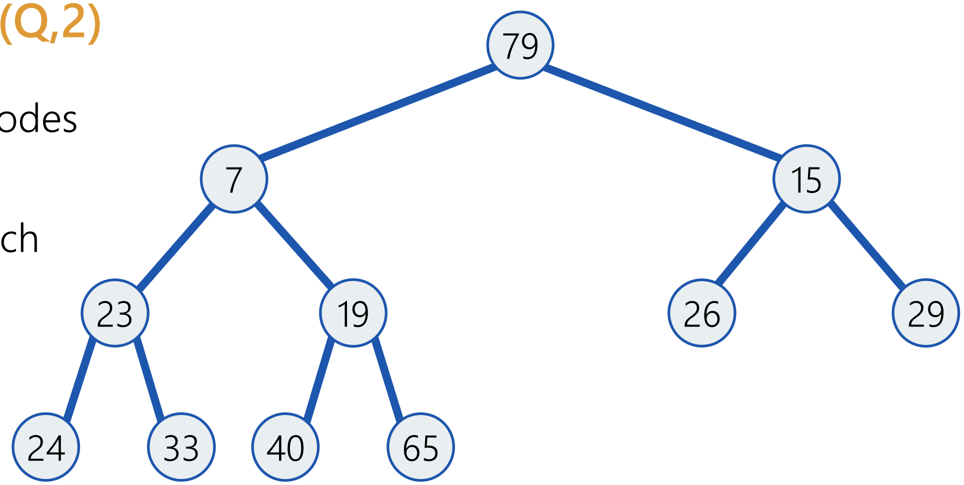
	1	2	3	4	5	6	7	8	9	10	11
Q	79	7	15	23	65	26	29	24	33	40	19

Turn an array into a heap

Heapify-down(Q,2)

Iterate over internal nodes bottom up,

and Heapify-Down each



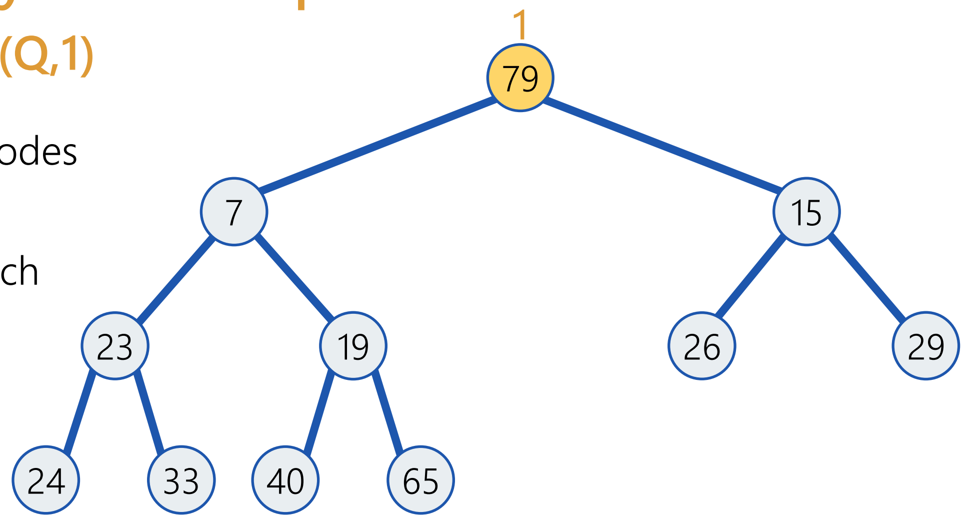
	1	2	3	4	5	6	7	8	9	10	11
Q	79	7	15	23	19	26	29	24	33	40	65

Turn an array into a heap

Heapify-down(Q,1)

Iterate over internal nodes bottom up,

and Heapify-Down each



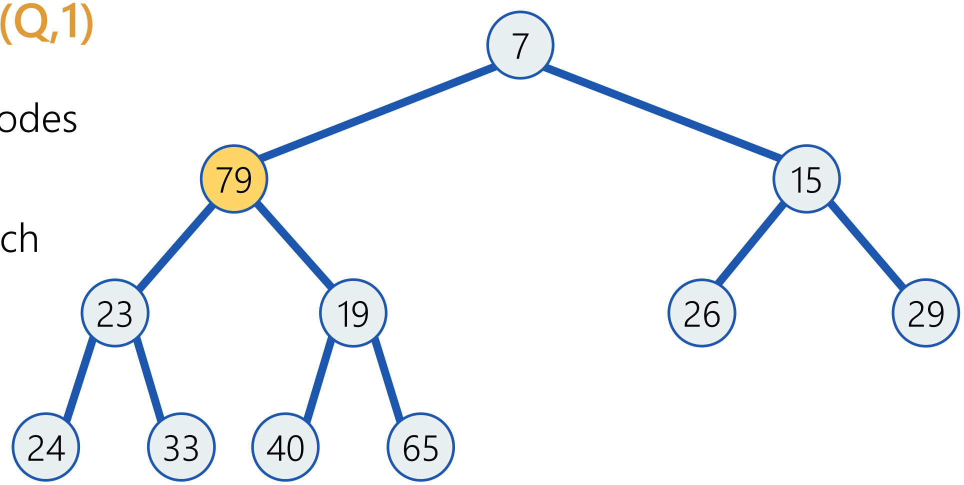
	1	2	3	4	5	6	7	8	9	10	11
Q	79	7	15	23	19	26	29	24	33	40	65

Turn an array into a heap

Heapify-down(Q,1)

Iterate over internal nodes bottom up,

and Heapify-Down each



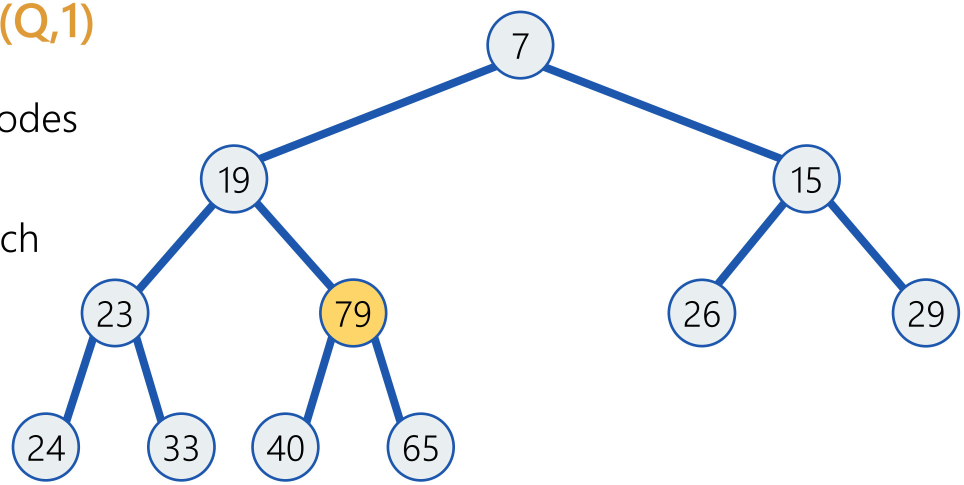
	1	2	3	4	5	6	7	8	9	10	11
Q	7	79	15	23	19	26	29	24	33	40	65

Turn an array into a heap

Heapify-down(Q,1)

Iterate over internal nodes bottom up,

and Heapify-Down each



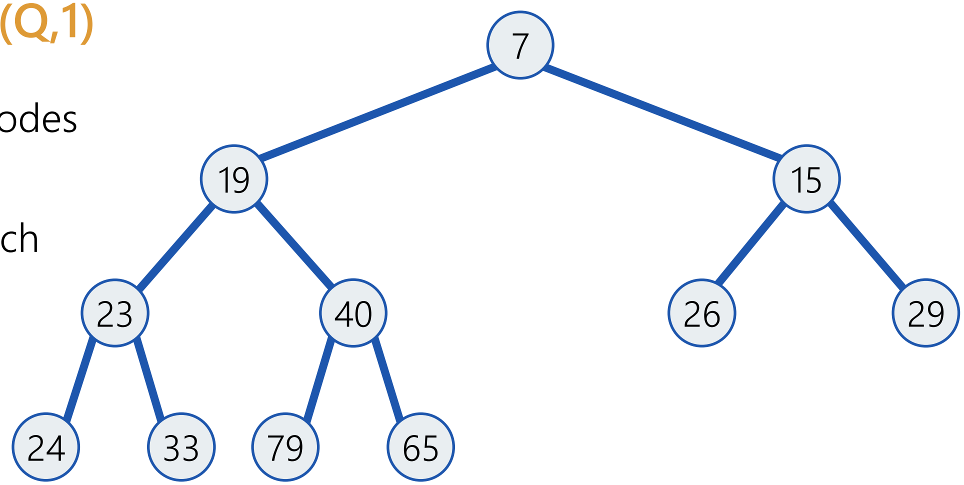
	1	2	3	4	5	6	7	8	9	10	11
Q	7	19	15	23	79	26	29	24	33	40	65

Turn an array into a heap

Heapify-down(Q,1)

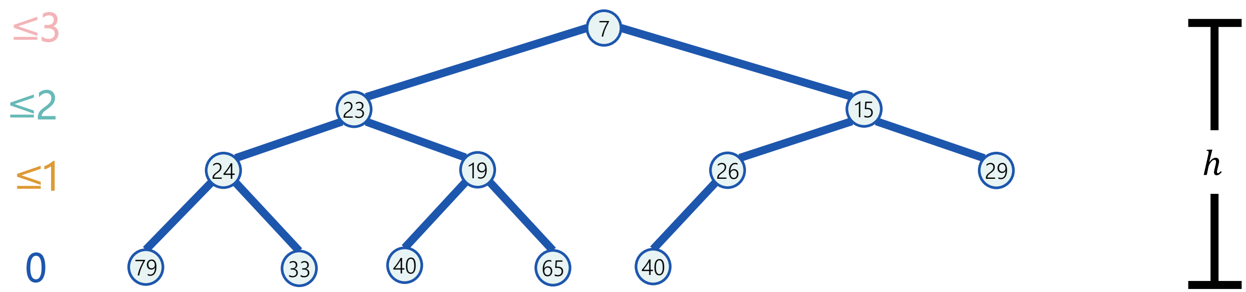
Iterate over internal nodes bottom up,

and Heapify-Down each



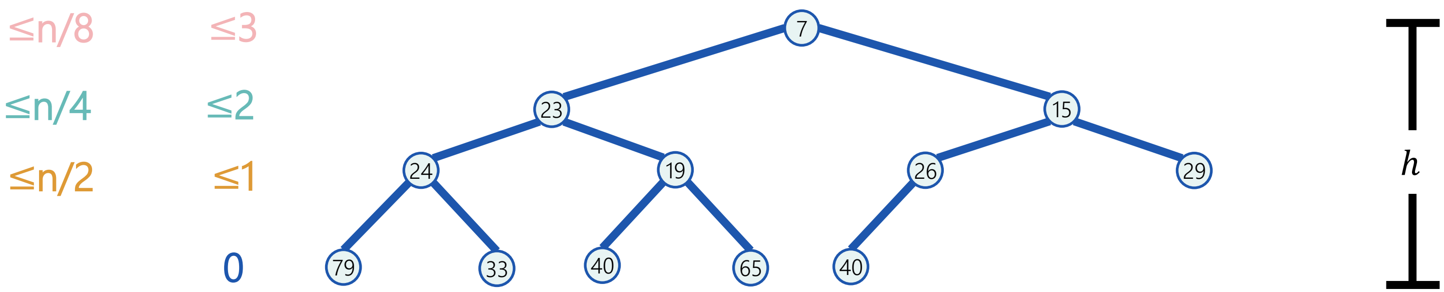
	1	2	3	4	5	6	7	8	9	10	11
Q	7	19	15	23	40	26	29	24	33	79	65

How much time does it take to build the heap this way ?



Cost of Heapifying a node bounded by its height

How much time does it take to build the heap this way ?



At most n/2 nodes heapified at height at most 1

At most n/4 nodes heapified at height at most 2

At most n/8 nodes heapified at height at most 3

$$Total\ time \leq 1\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \cdots + 1H = \sum_{h=1}^{H} h\frac{n}{2^h} < n\sum_{h=1}^{\infty}\frac{h}{2^h} = O(n)$$

Turn an array into a heap efficiently

“Iterate over internal nodes
bottom up,

and Heapify-Down each”

Why must we go bottom-up?

Heapify-Down assumes subtree of index i is legal, except maybe for its root (i).

Example where going top down would not work?

Turning an Array into a Heap - Summary

- **If done naively by insertions -** $O(nlogn)$ **time worst case**
- **If done by Heapifying-Down non-leaves bottom up -** $O(n)$
- **We will soon prove that binary search trees cannot be built in** $O(n)$**. Intuitively, they contain more order, thus need more time to build**

Heapsort: The Algorithm

How Does It Work?

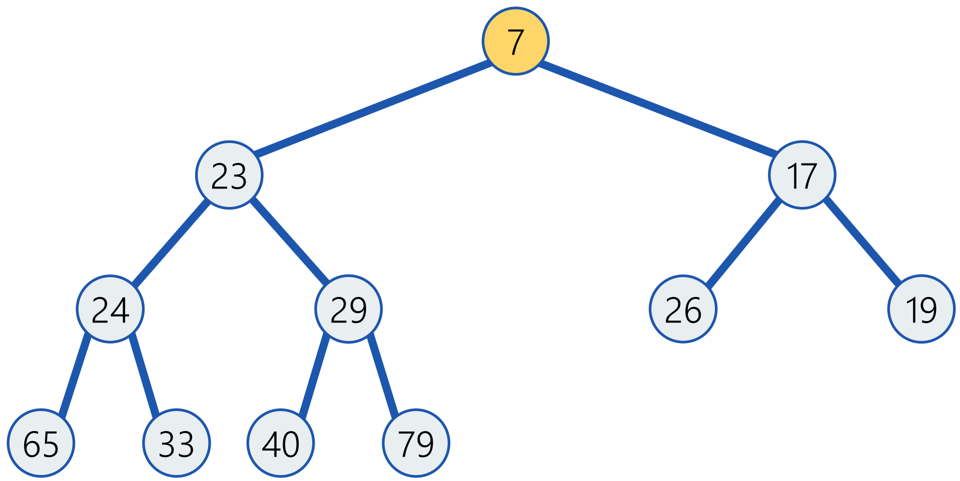
Heapsort (Williams, Floyd, 1964)

- Input: array with n elements
- Output: array sorted

- Algorithm:
  - Create min-heap from input
  - Do delete-min, and put the deleted element at the last position of the array.
    Repeat n times.
  - Reverse the array (or use a max-heap instead)

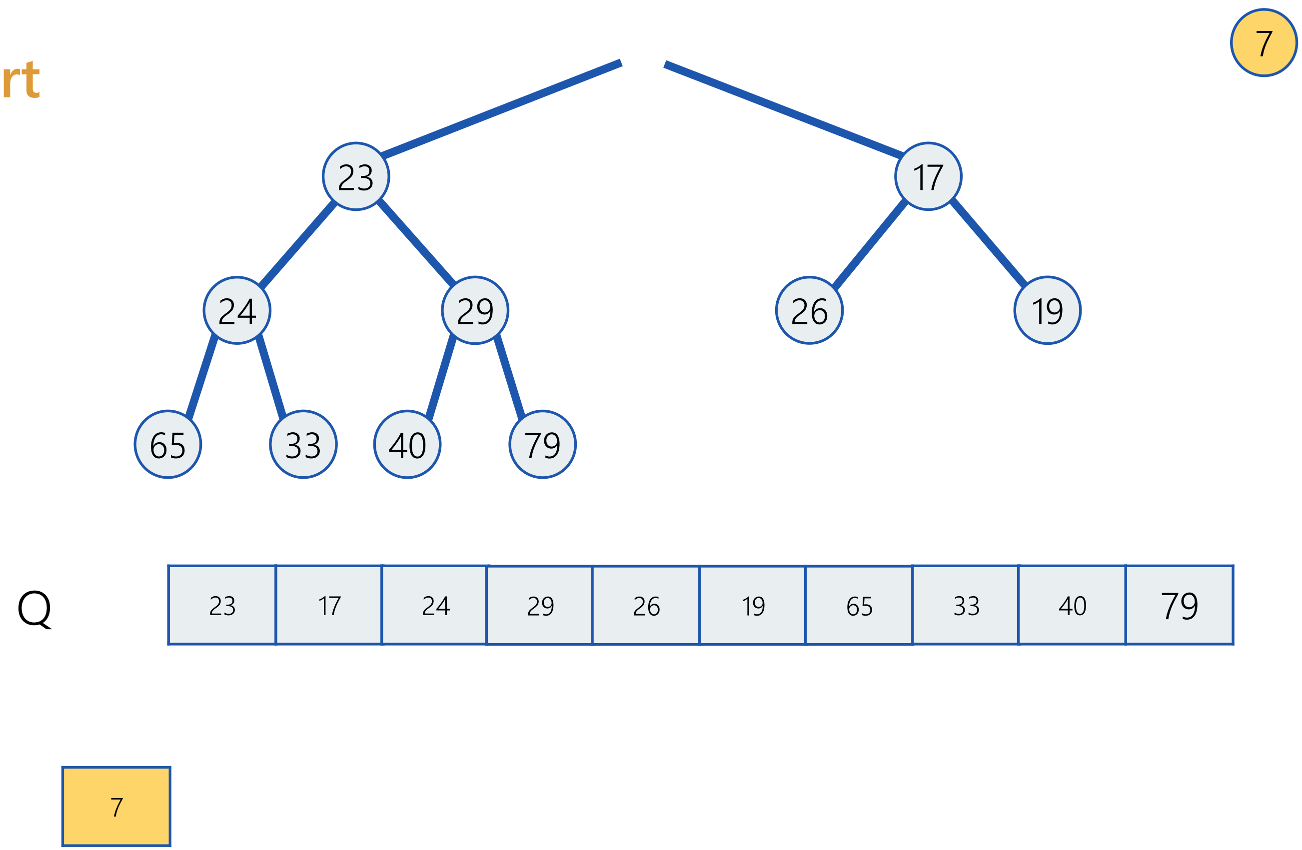
Insert(x, Q)	$O(logn)$
min(Q)	$O(1)$
Delete-min(Q)	$O(logn)$

Heapsort

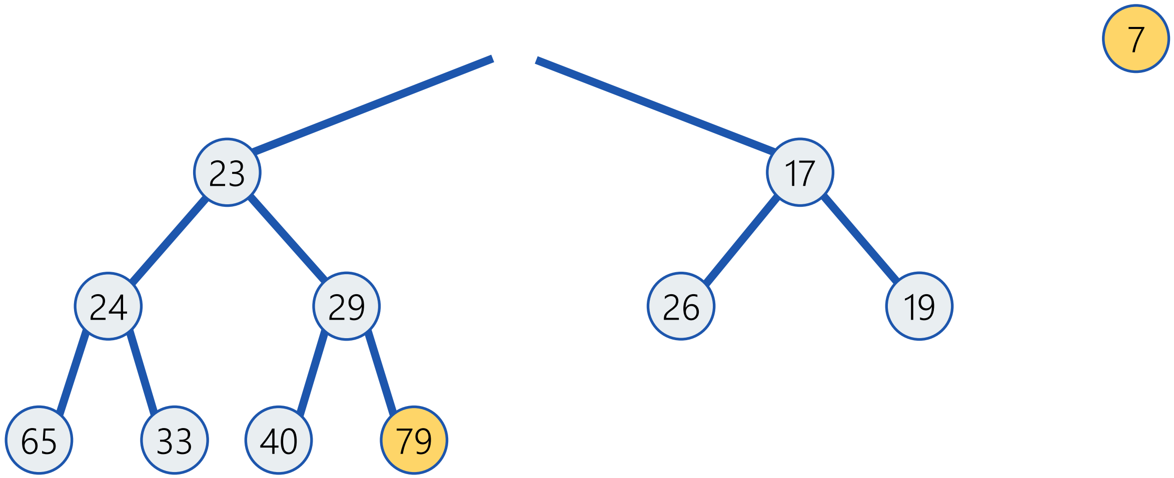


Q	7	23	17	24	29	26	19	65	33	40	79

Heapsort



Heapsort

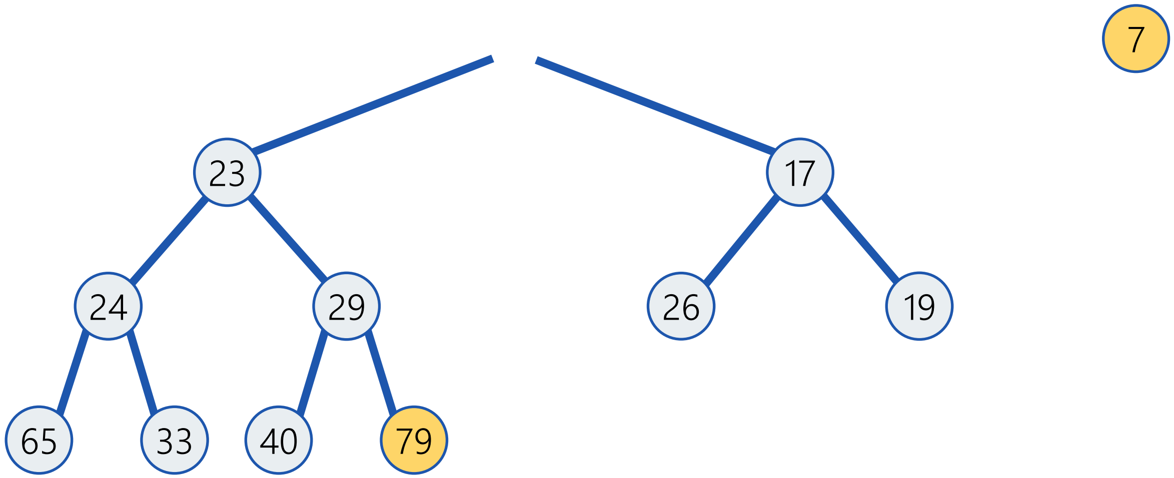


Q

23	17	24	29	26	19	65	33	40	79

7

Heapsort

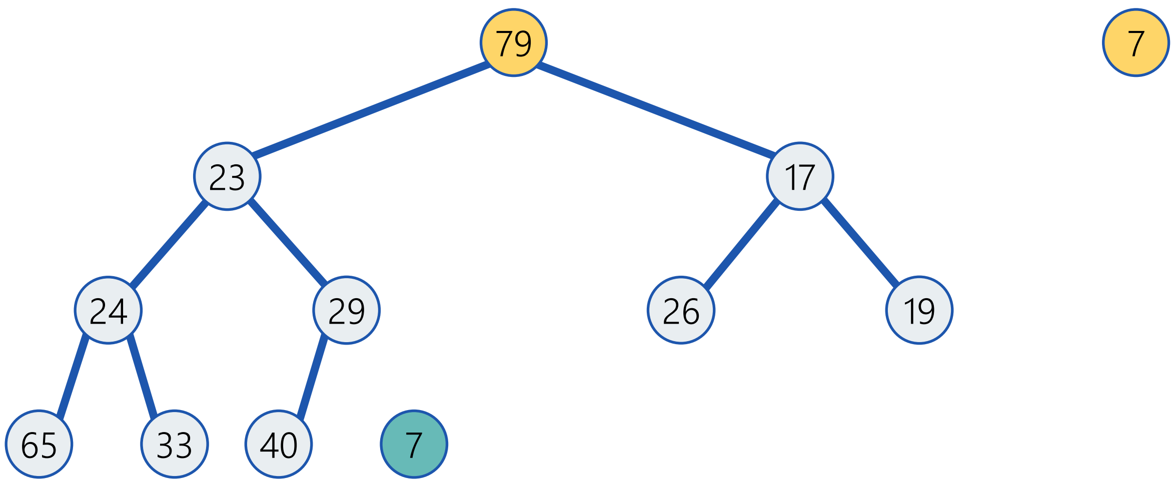


Q

23	17	24	29	26	19	65	33	40	79

7

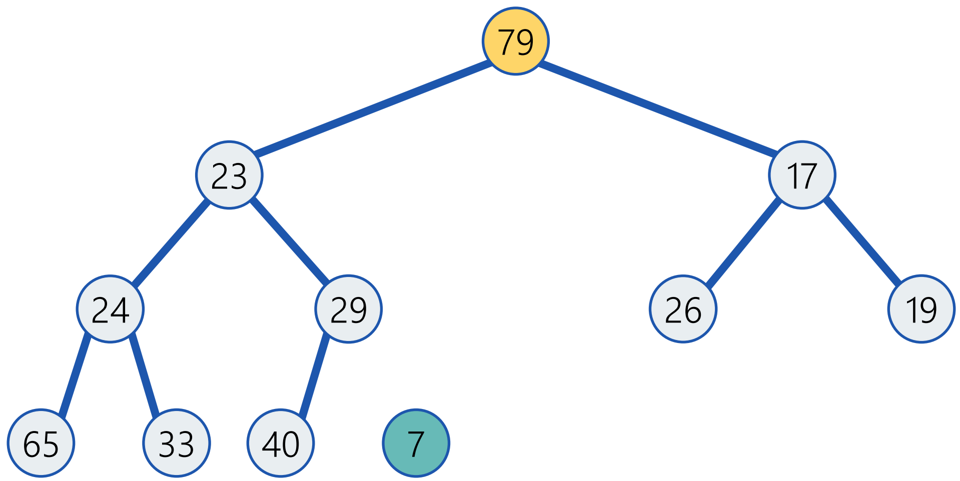
Heapsort



Q										
79	23	17	24	29	26	19	65	33	40	7

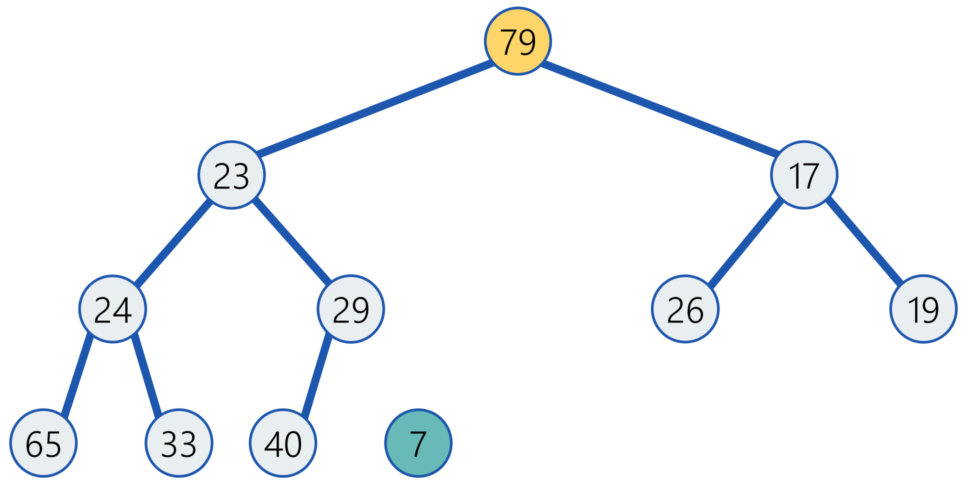
7

Heapsort



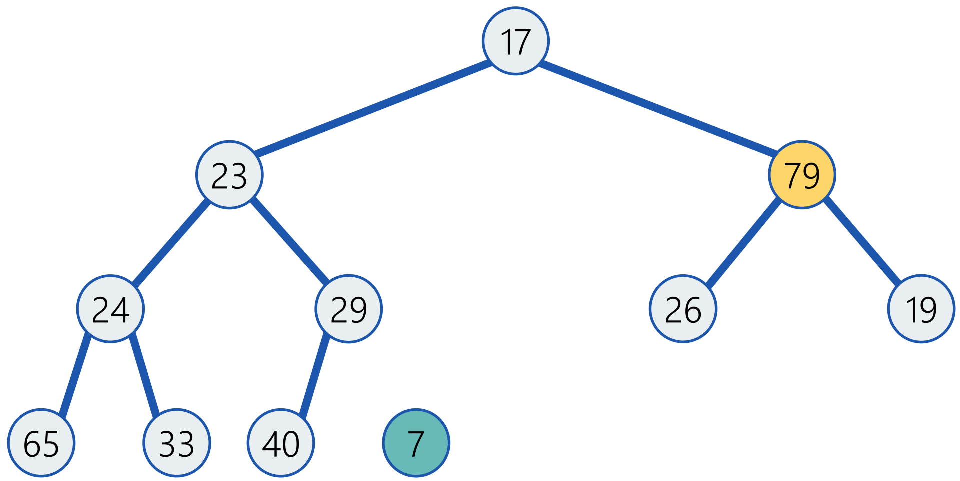
Q	79	23	17	24	29	26	19	65	33	40	7

Heapsort



Q	79	23	17	24	29	26	19	65	33	40	7

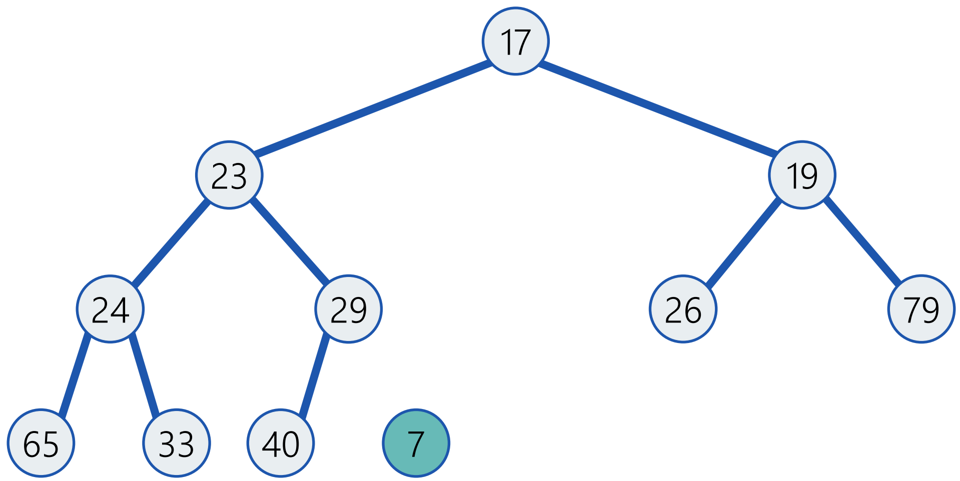
Heapsort



Q

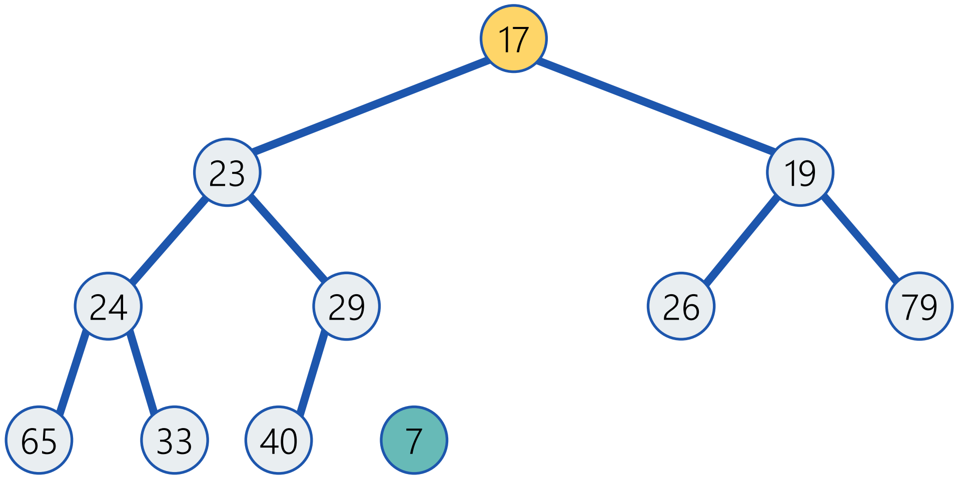
17	23	79	24	29	26	19	65	33	40	7

Heapsort



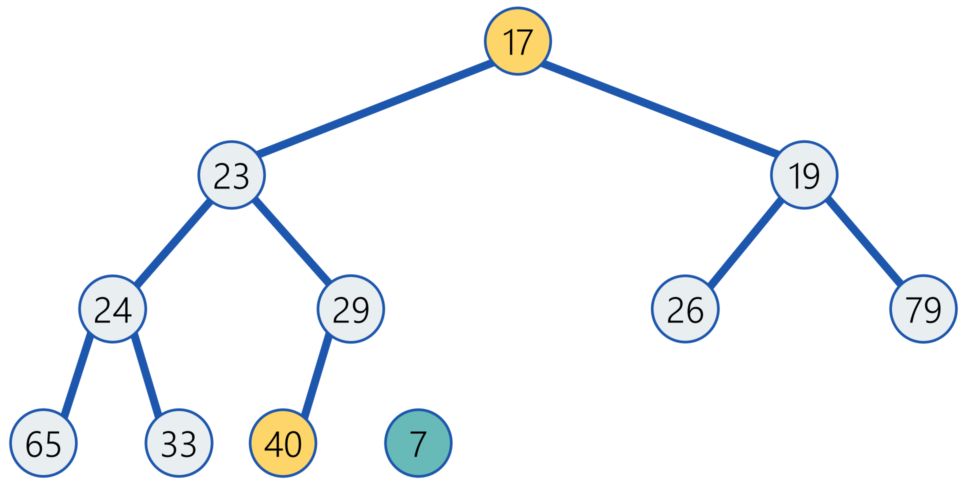
Q											
	17	23	19	24	29	26	79	65	33	40	7

Heapsort



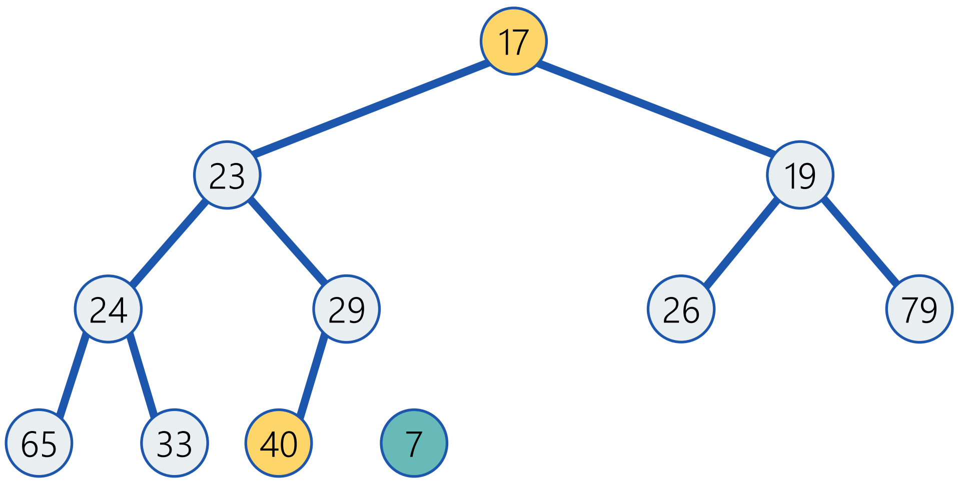
Q										
17	23	19	24	29	26	79	65	33	40	7

Heapsort



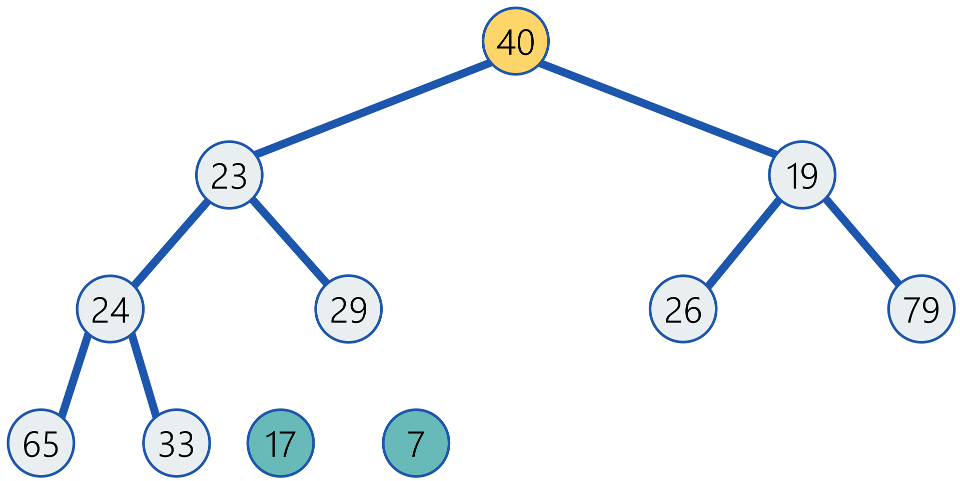
Q										
17	23	19	24	29	26	79	65	33	40	7

Heapsort



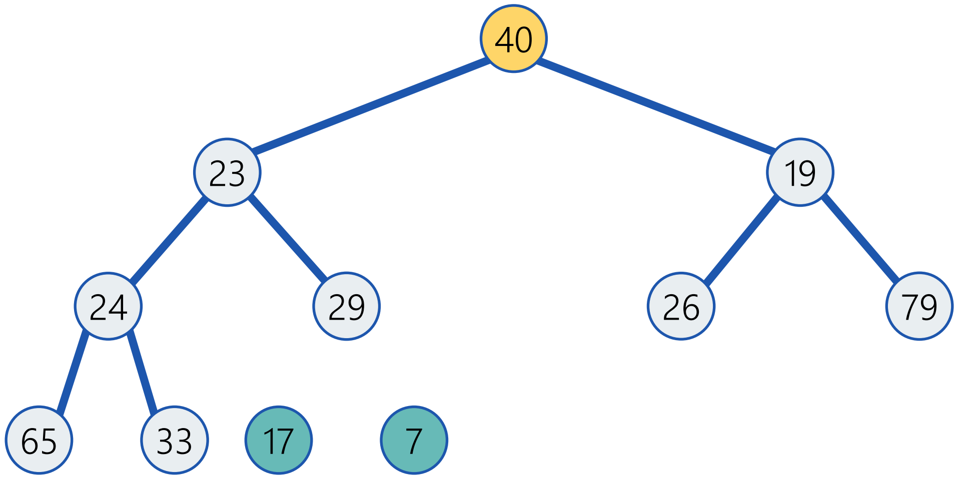
Q										
17	23	19	24	29	26	79	65	33	40	7

Heapsort



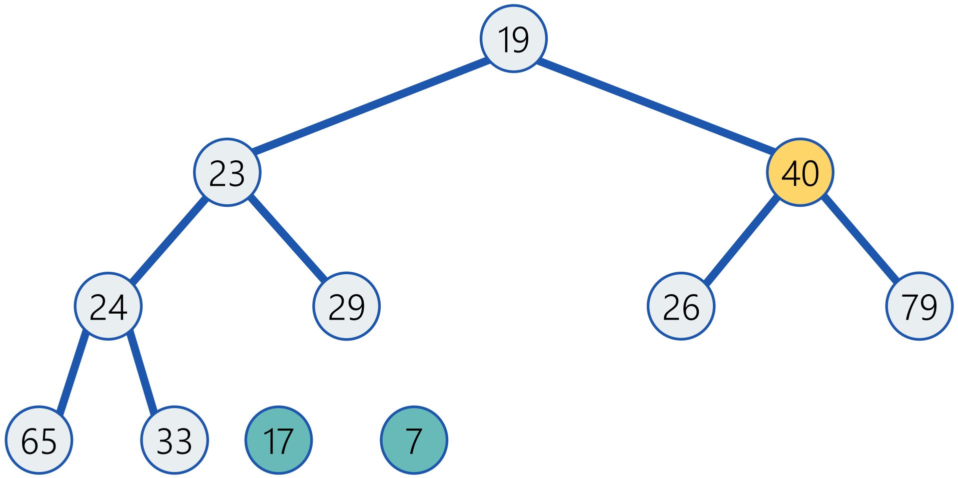
Q	40	23	19	24	29	26	79	65	33	17	7

Heapsort



Q										
40	23	19	24	29	26	79	65	33	17	7

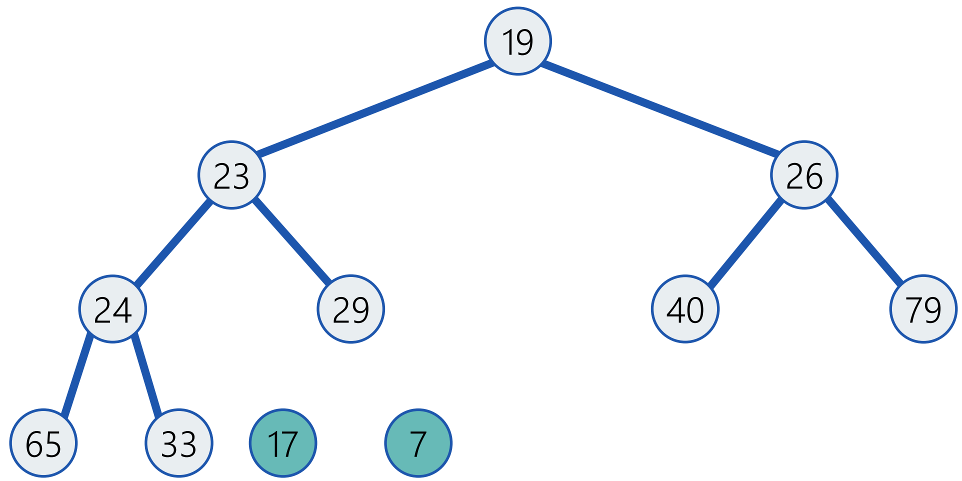
Heapsort



Q

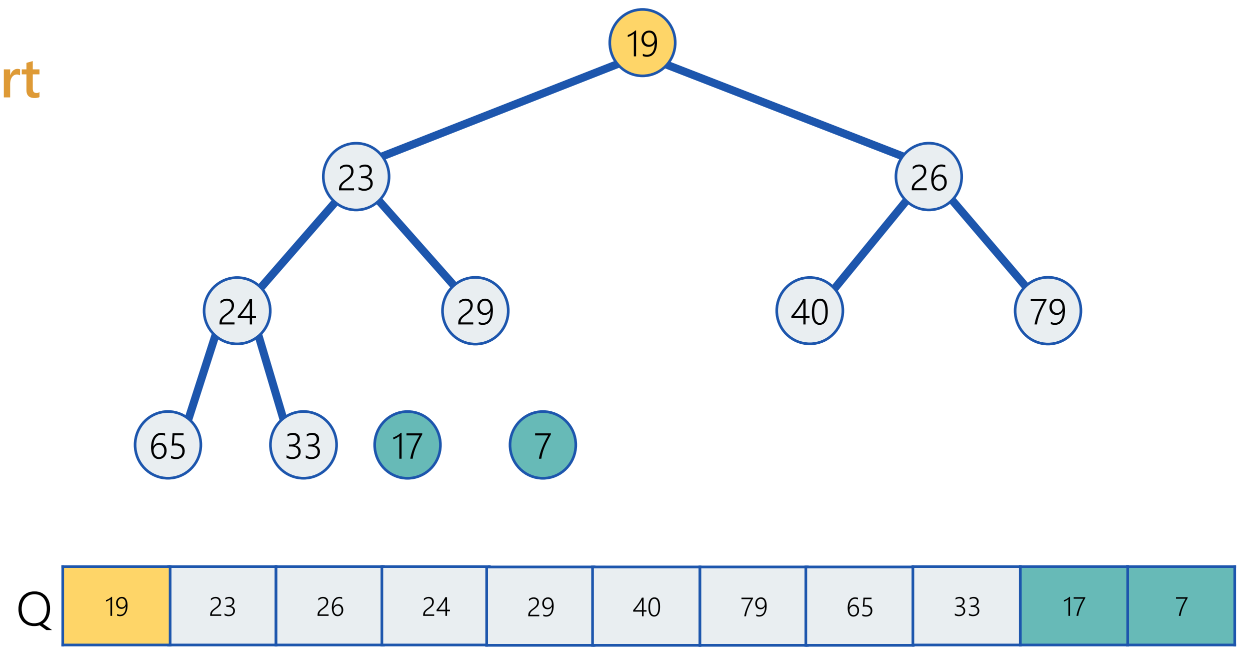
19	23	40	24	29	26	79	65	33	17	7

Heapsort

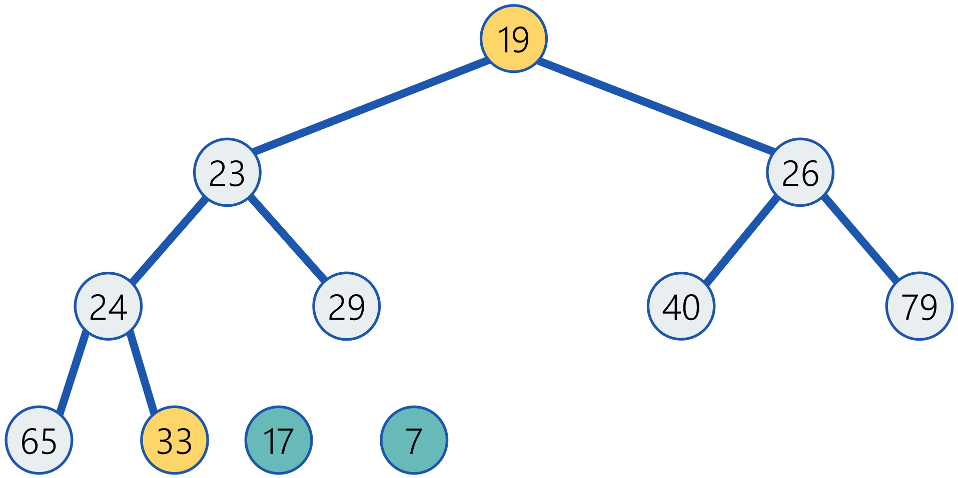


Q										
19	23	26	24	29	40	79	65	33	17	7

Heapsort



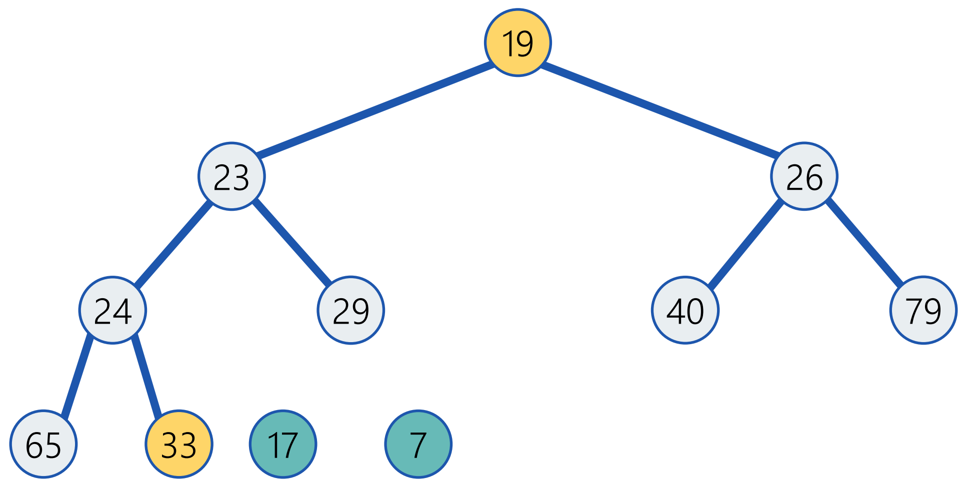
Heapsort



Q

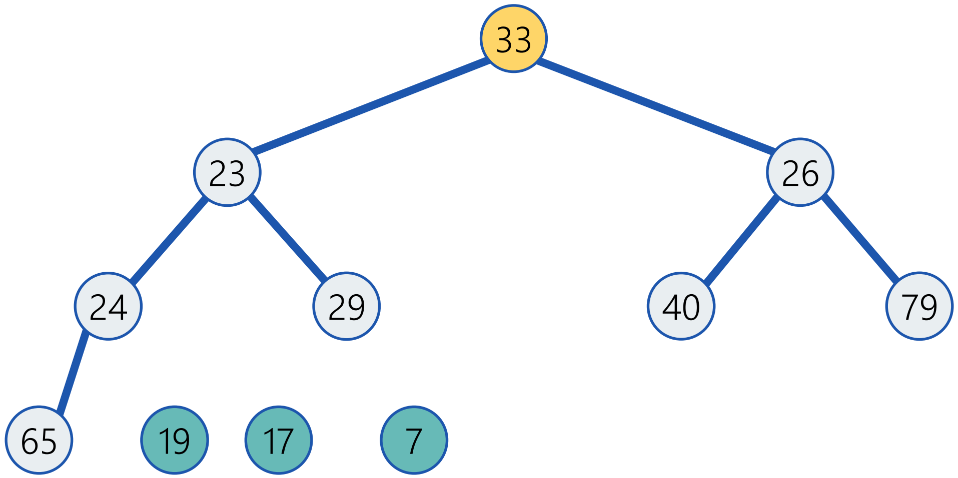
19	23	26	24	29	40	79	65	33	17	7

Heapsort



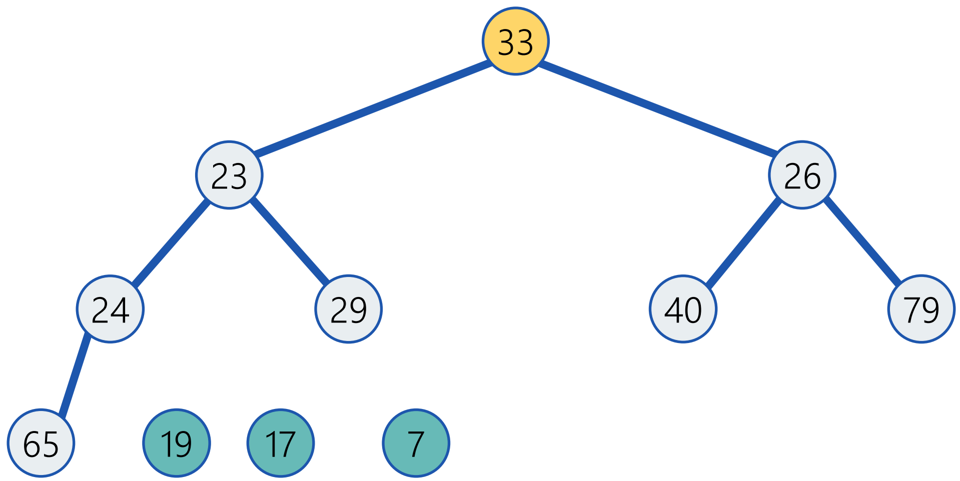
Q										
19	23	26	24	29	40	79	65	33	17	7

Heapsort



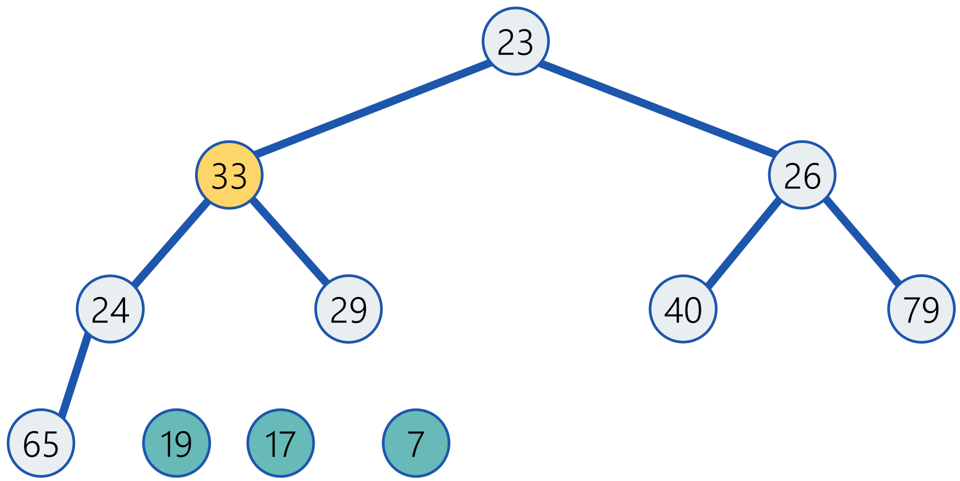
Q										
33	23	26	24	29	40	79	65	19	17	7

Heapsort



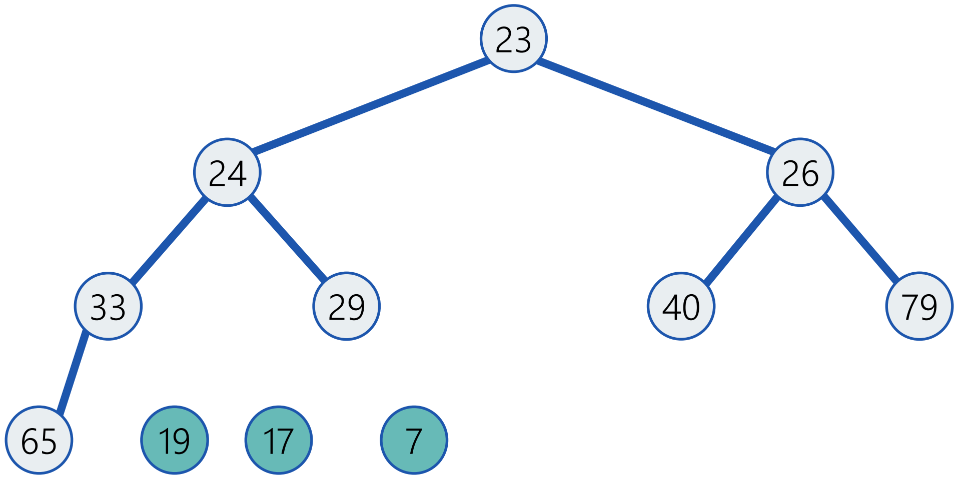
Q	33	23	26	24	29	40	79	65	19	17	7

Heapsort



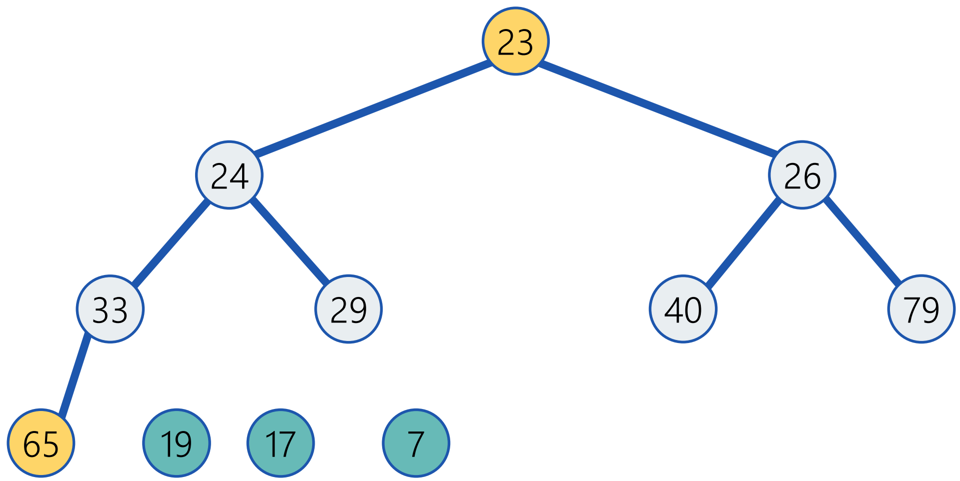
Q	23	33	26	24	29	40	79	65	19	17	7

Heapsort



Q	23	24	26	33	29	40	79	65	19	17	7

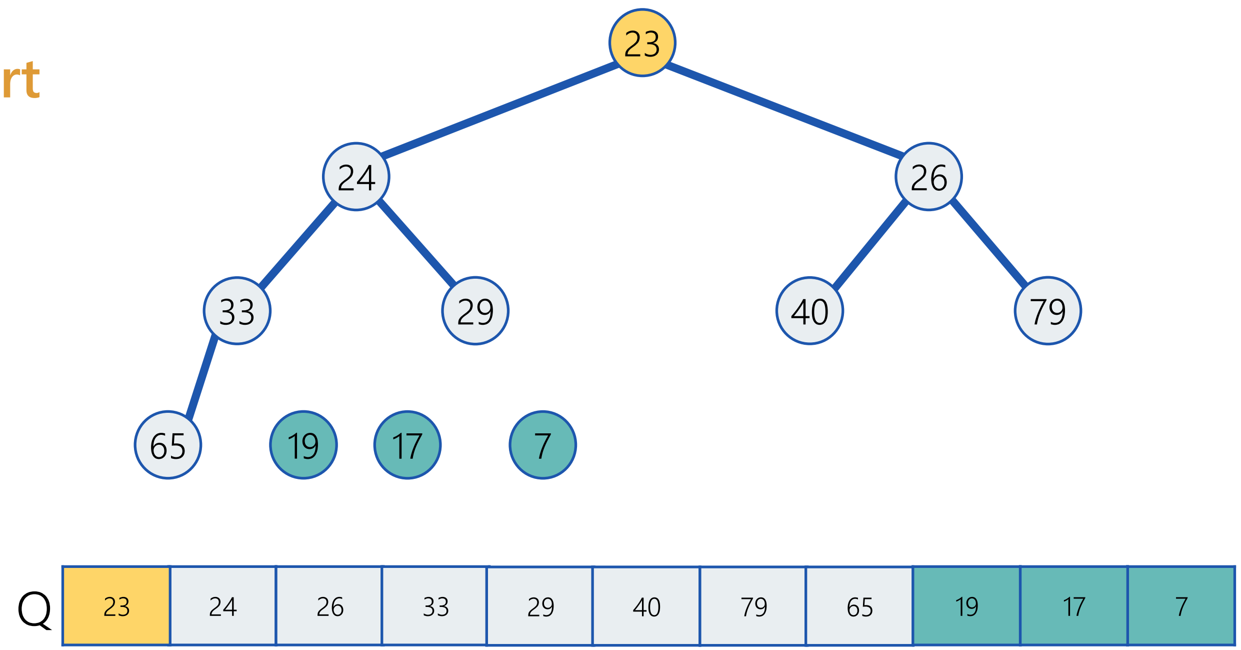
Heapsort



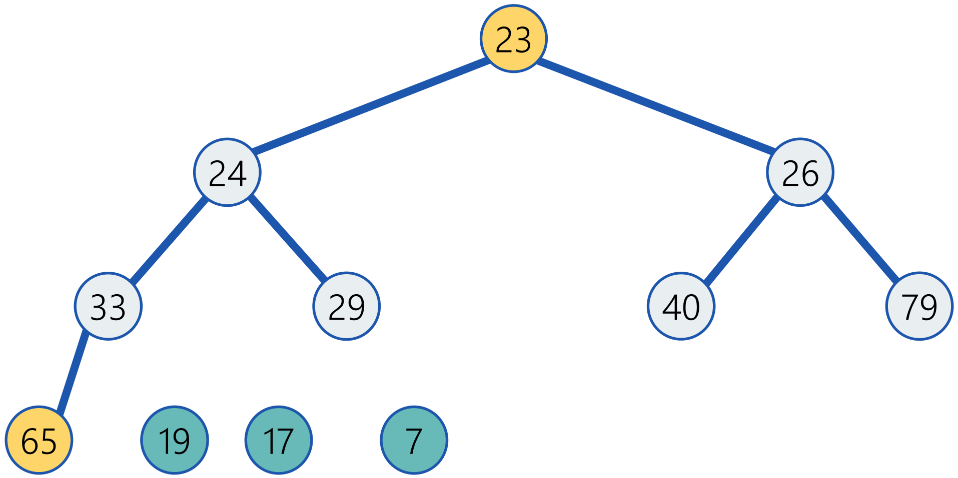
Q

23	24	26	33	29	40	79	65	19	17	7

Heapsort

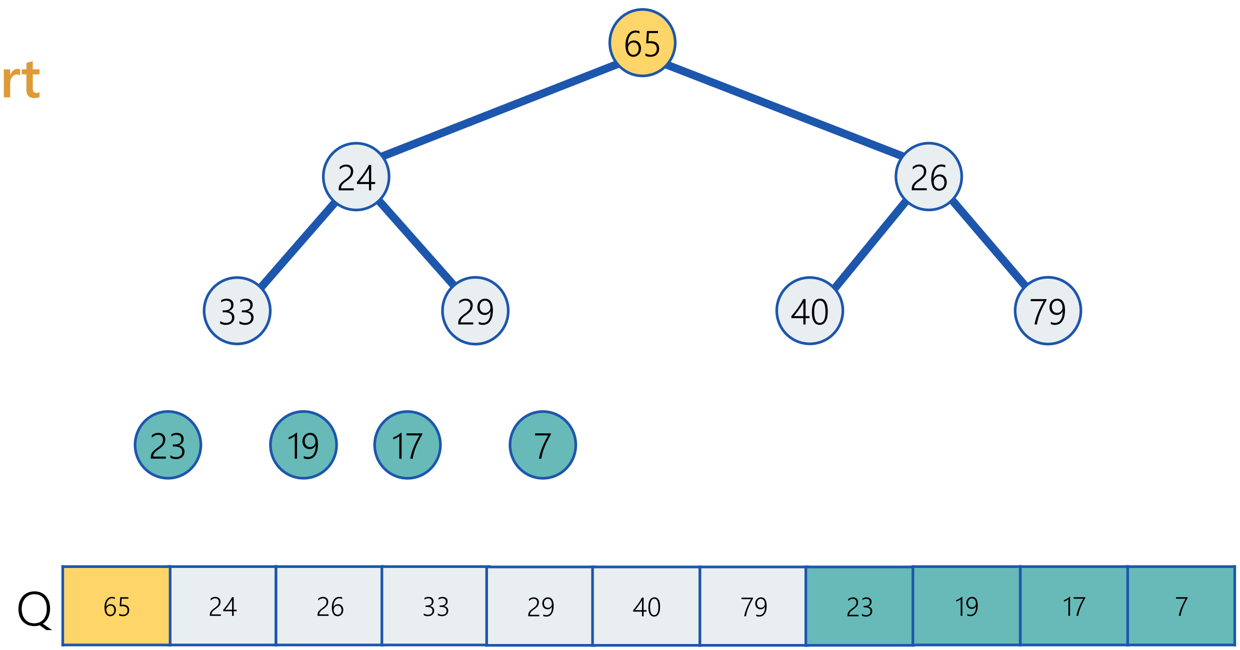


Heapsort

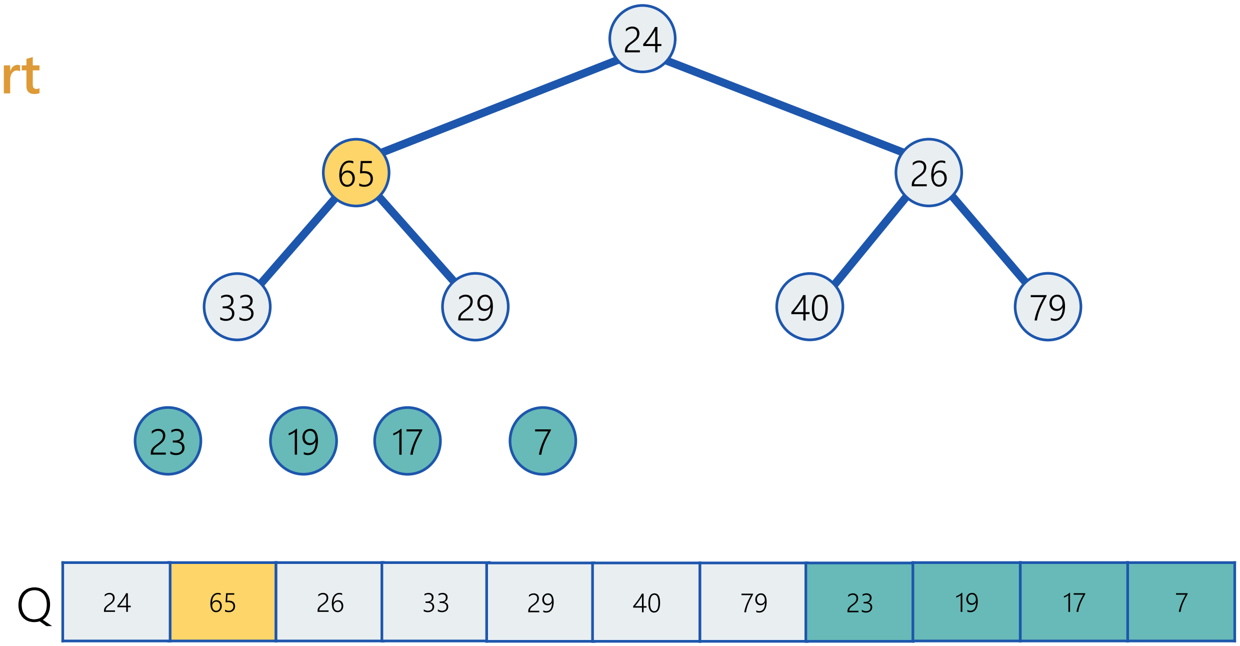


Q	23	24	26	33	29	40	79	65	19	17	7

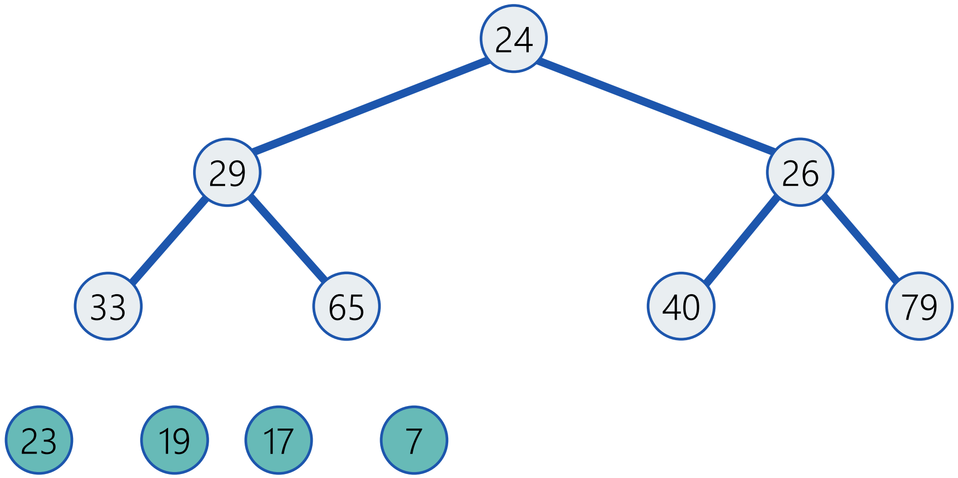
Heapsort



Heapsort



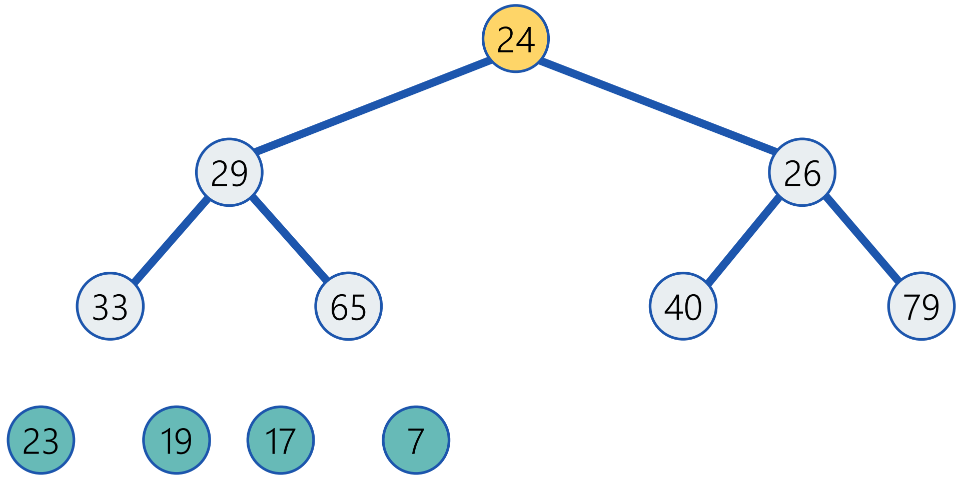
Heapsort



Q

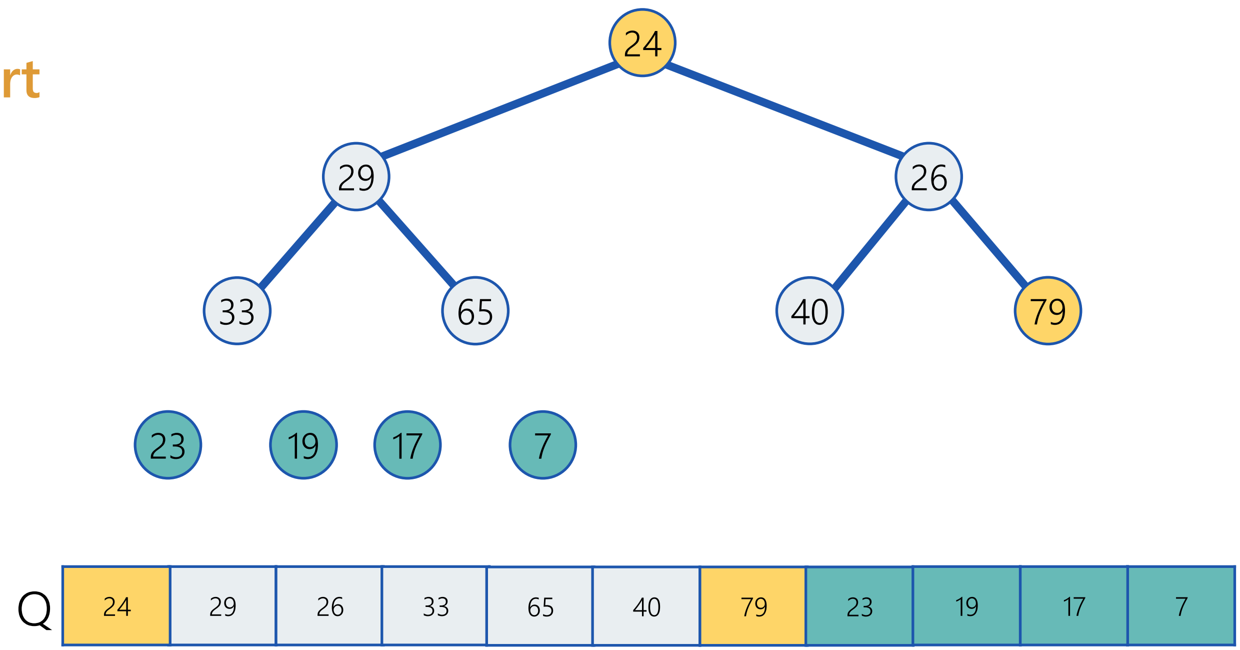
24	29	26	33	65	40	79	23	19	17	7

Heapsort

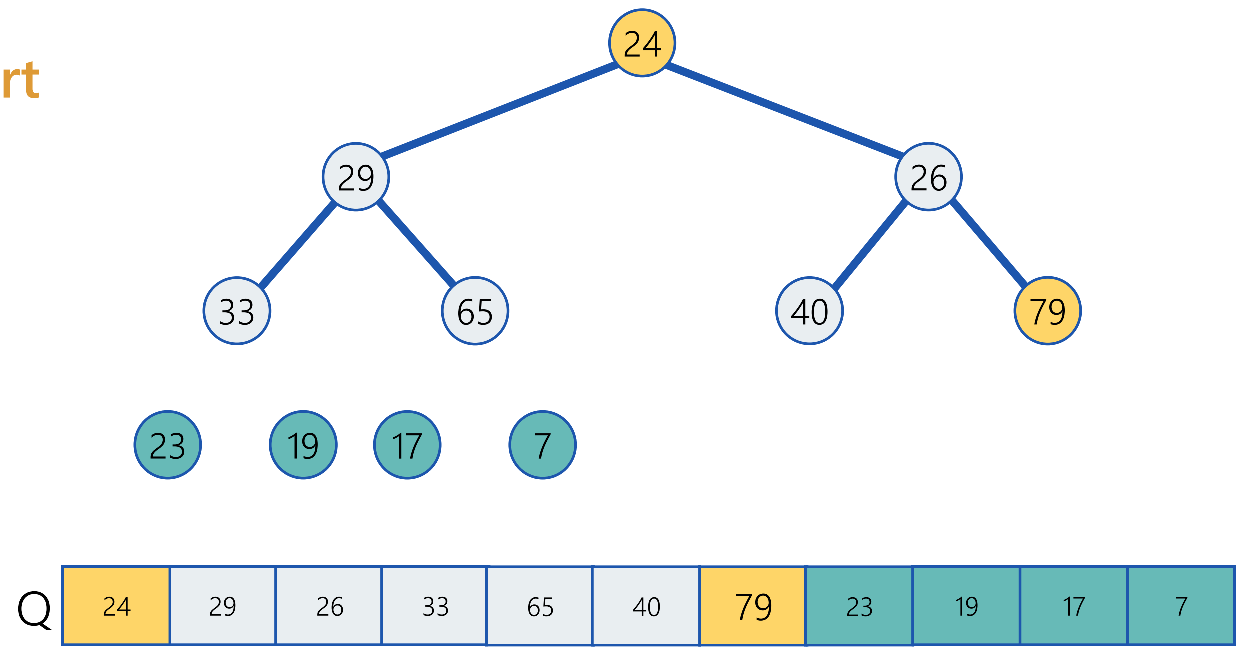


Q	24	29	26	33	65	40	79	23	19	17	7

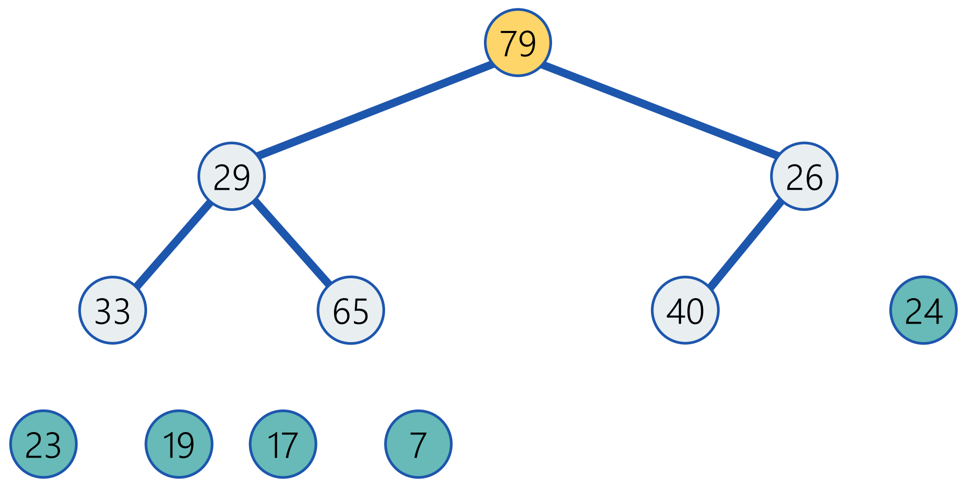
Heapsort



Heapsort

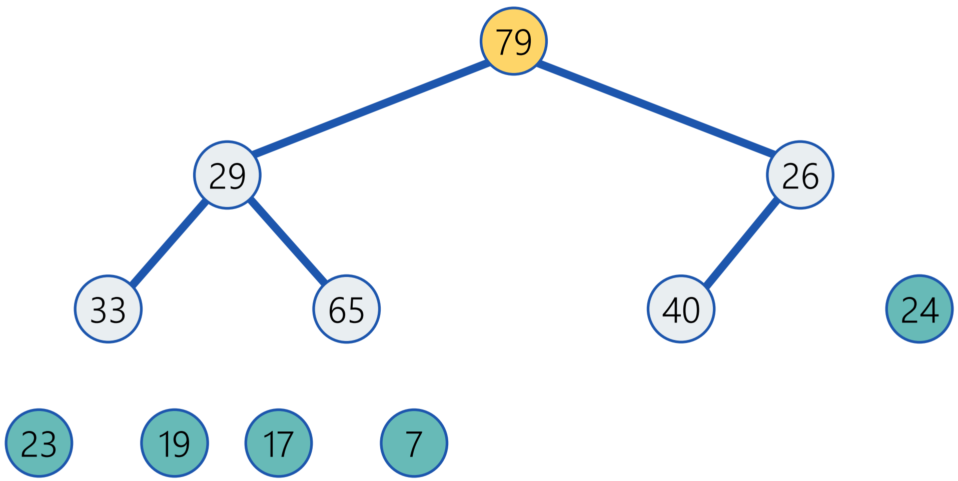


Heapsort



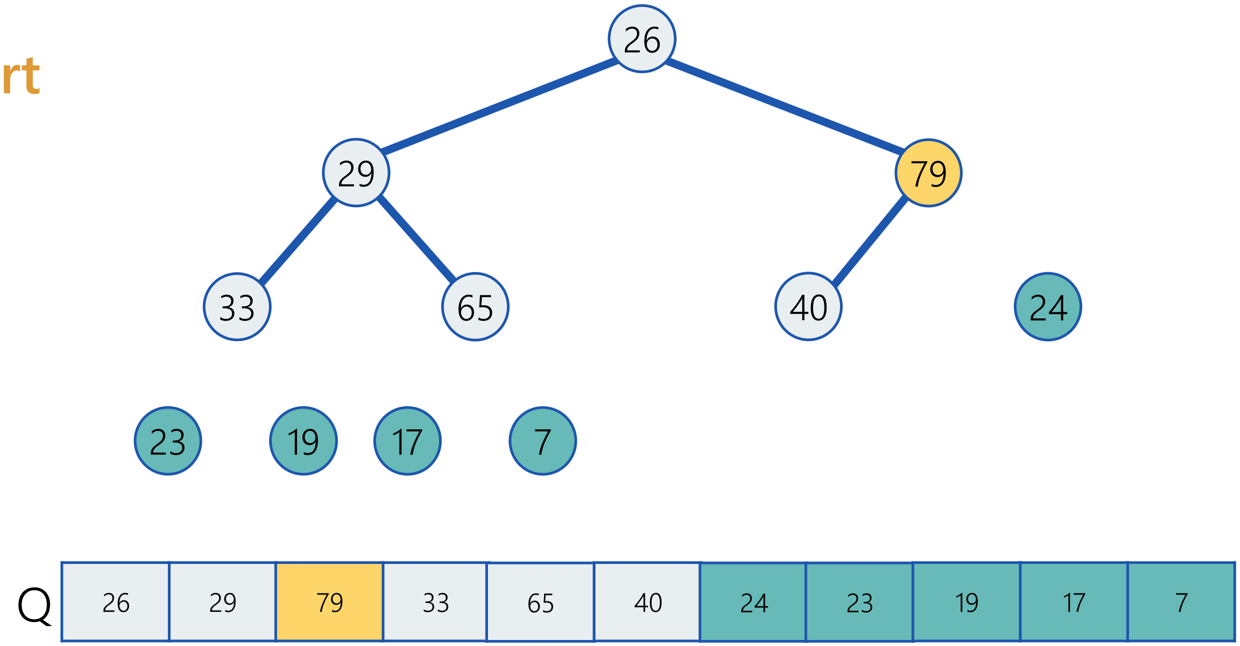
Q	79	29	26	33	65	40	24	23	19	17	7

Heapsort

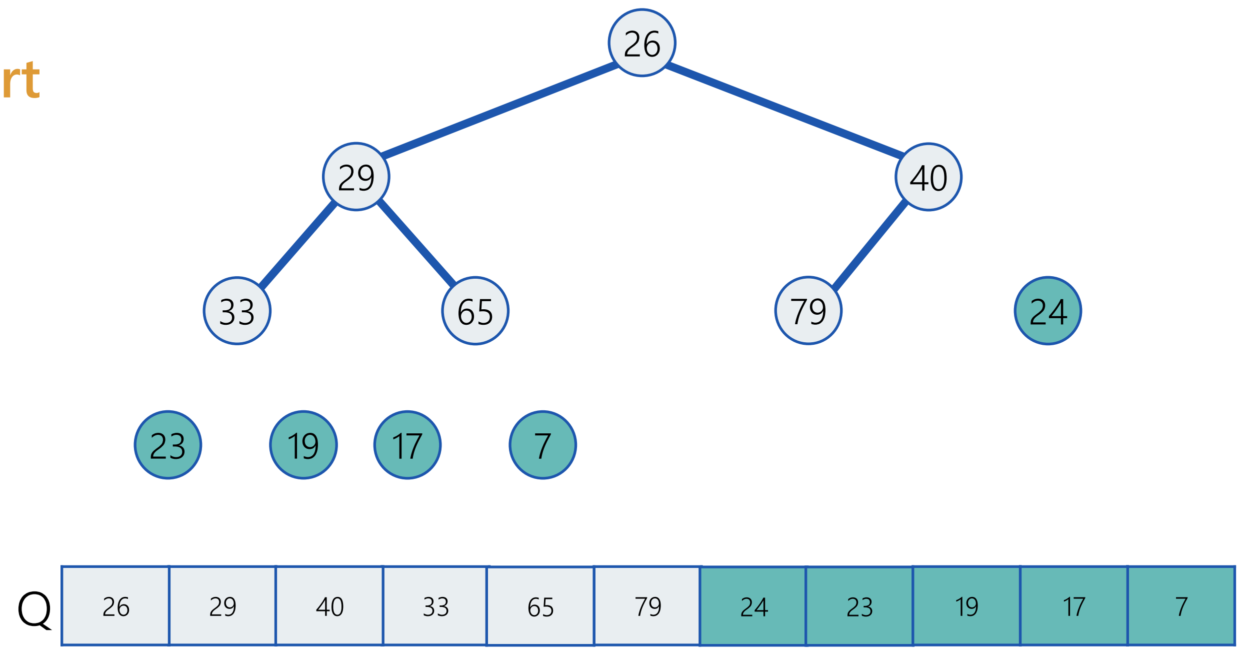


Q										
79	29	26	33	65	40	24	23	19	17	7

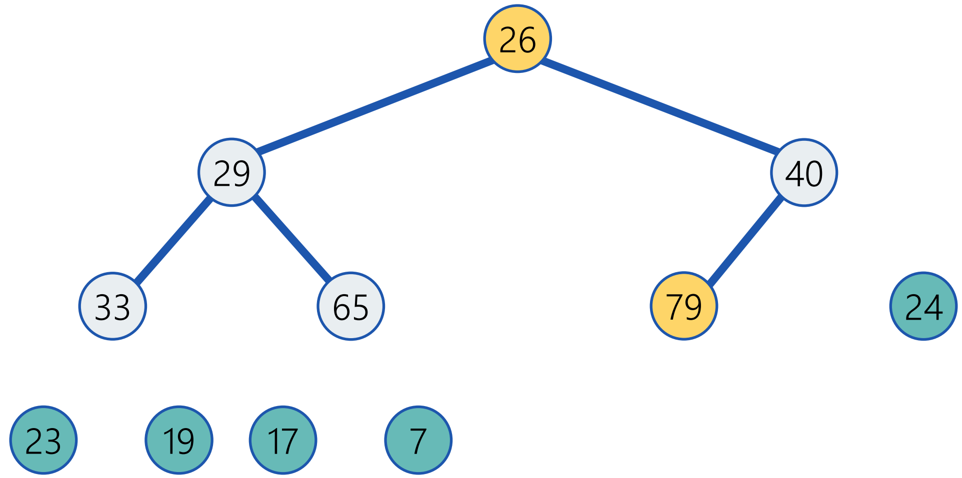
Heapsort



Heapsort

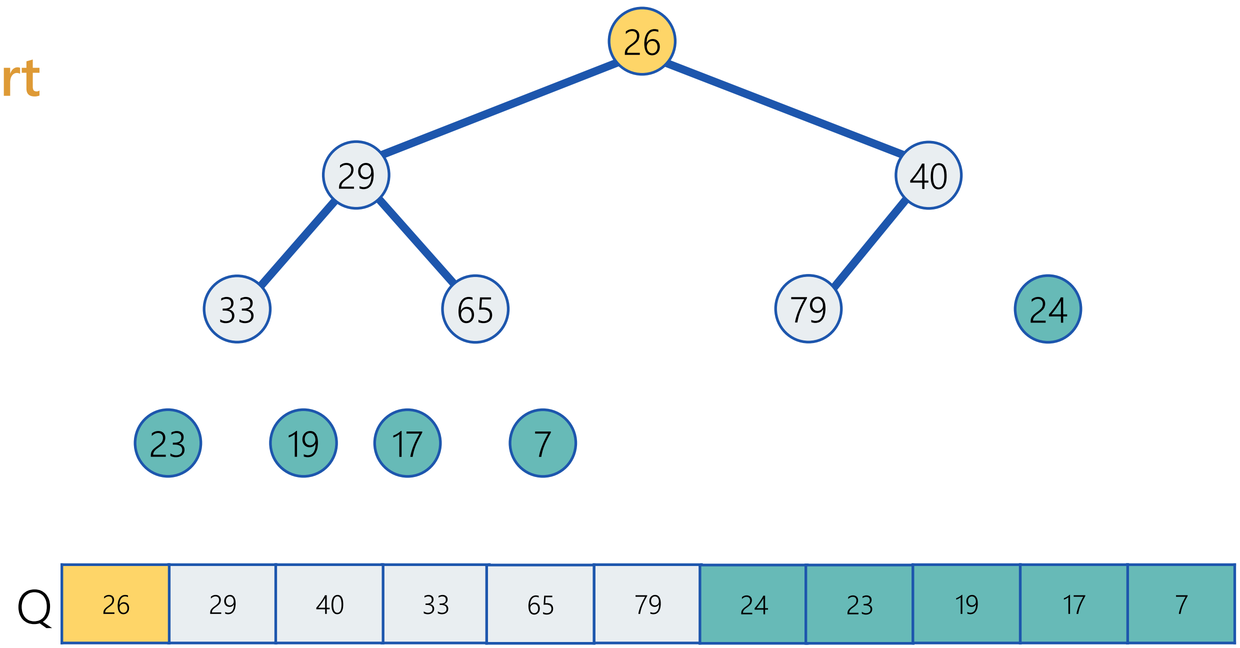


Heapsort

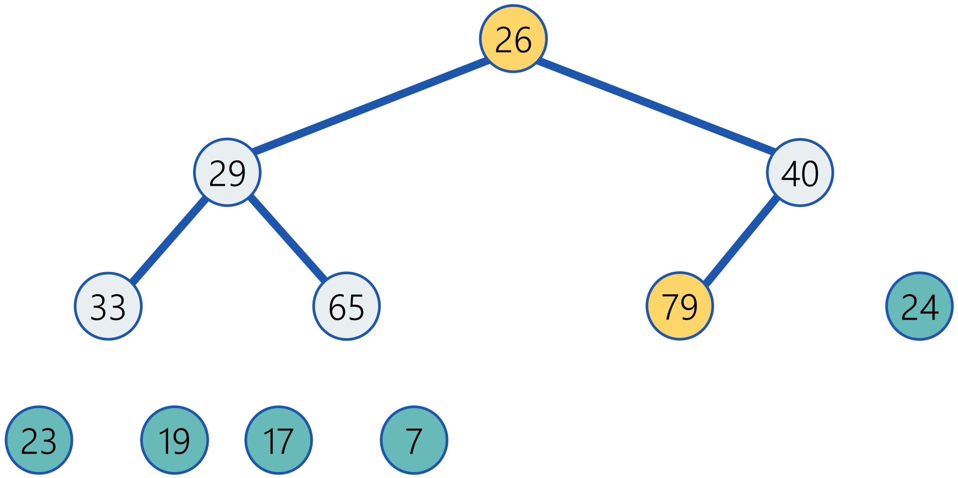


Q	26	29	40	33	65	79	24	23	19	17	7

Heapsort

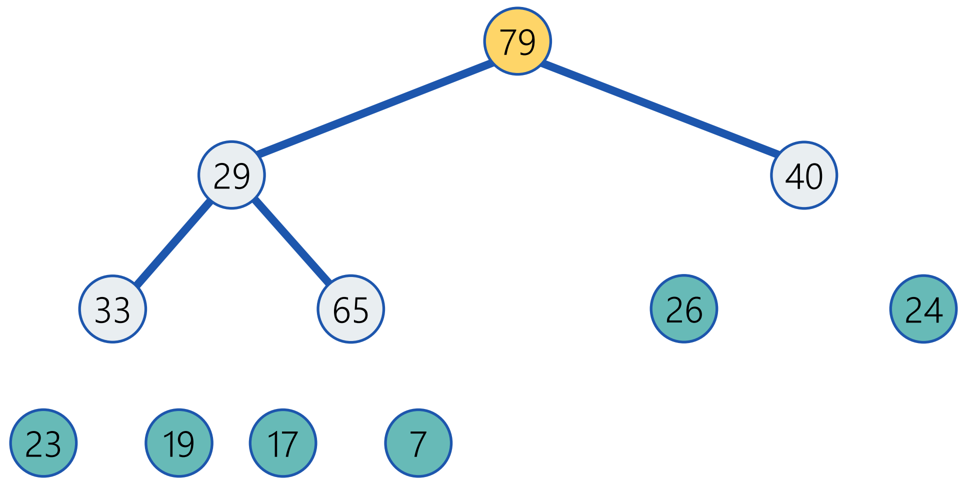


Heapsort



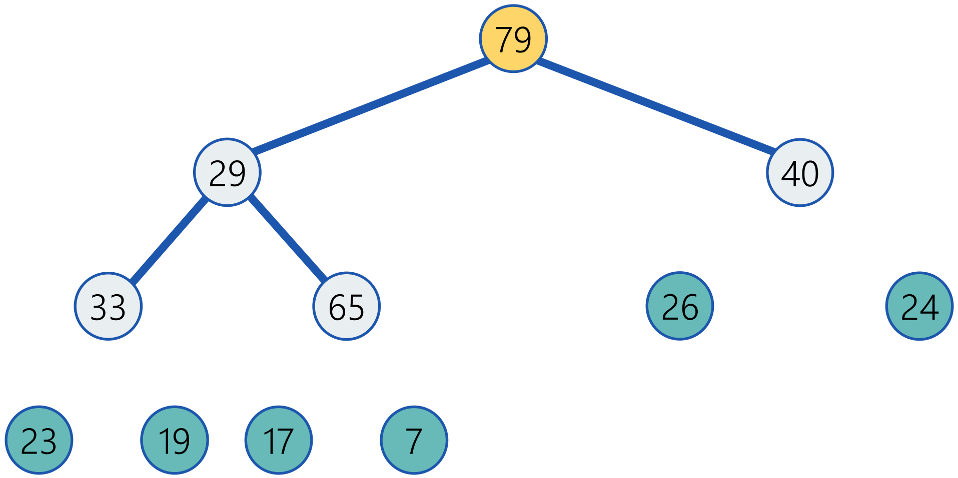
Q	26	29	40	33	65	79	24	23	19	17	7

Heapsort



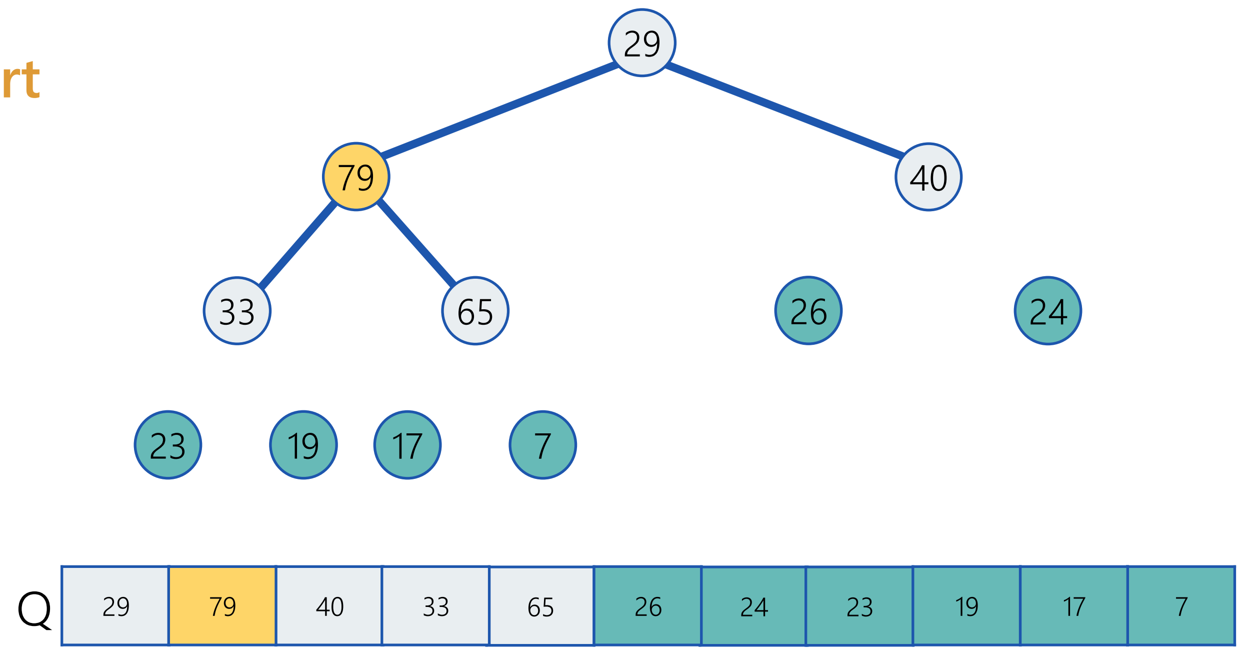
Q	79	29	40	33	65	26	24	23	19	17	7

Heapsort

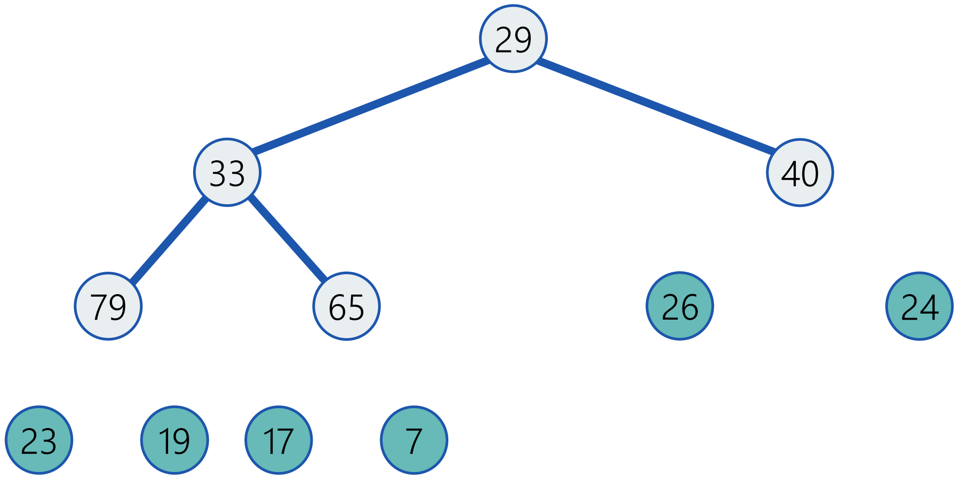


Q										
79	29	40	33	65	26	24	23	19	17	7

Heapsort

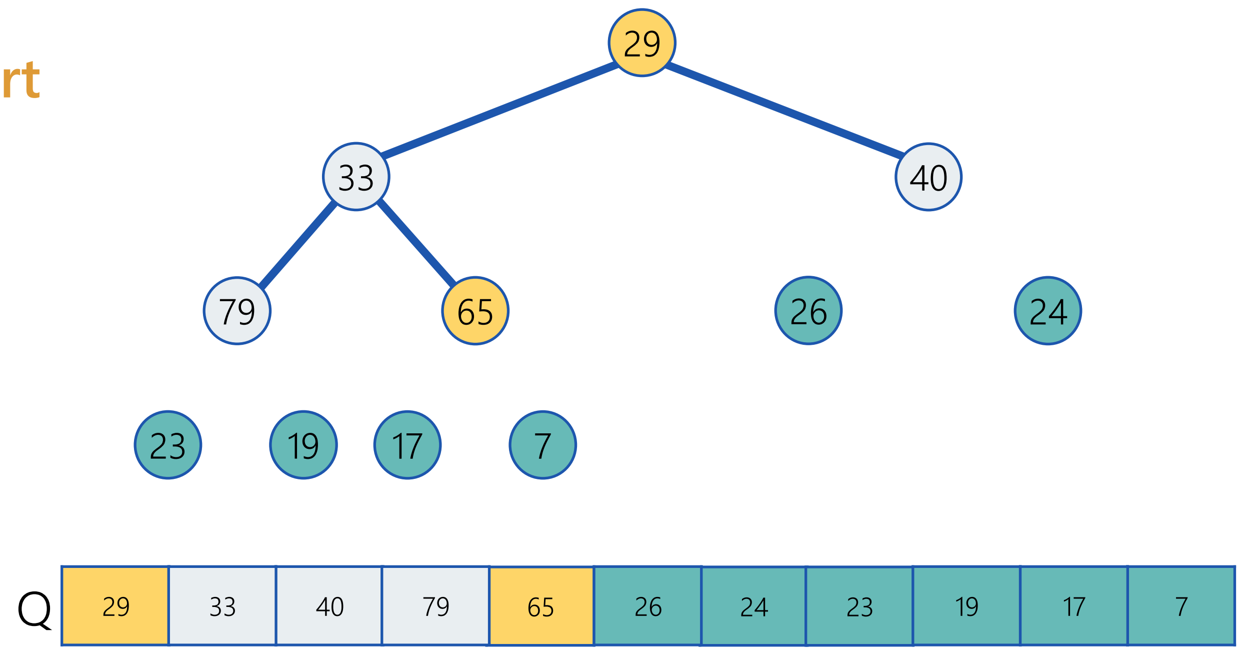


Heapsort

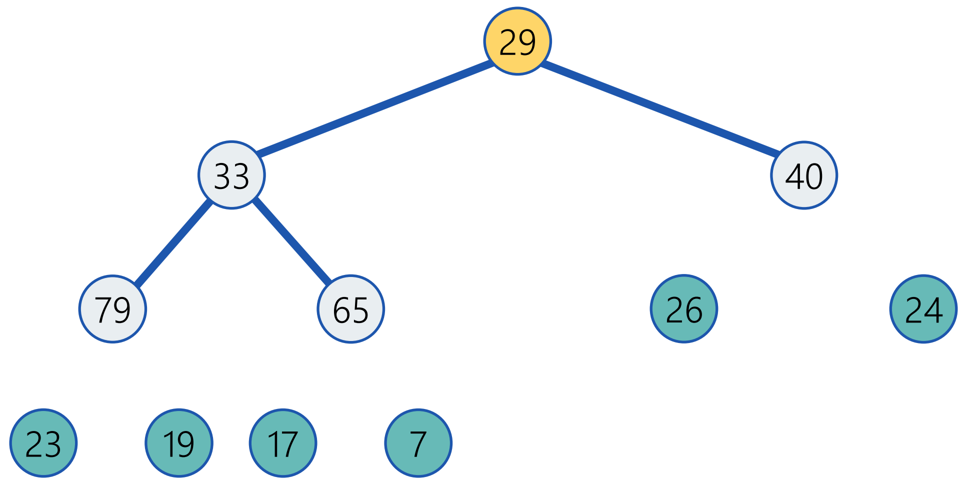


Q										
29	33	40	79	65	26	24	23	19	17	7

Heapsort

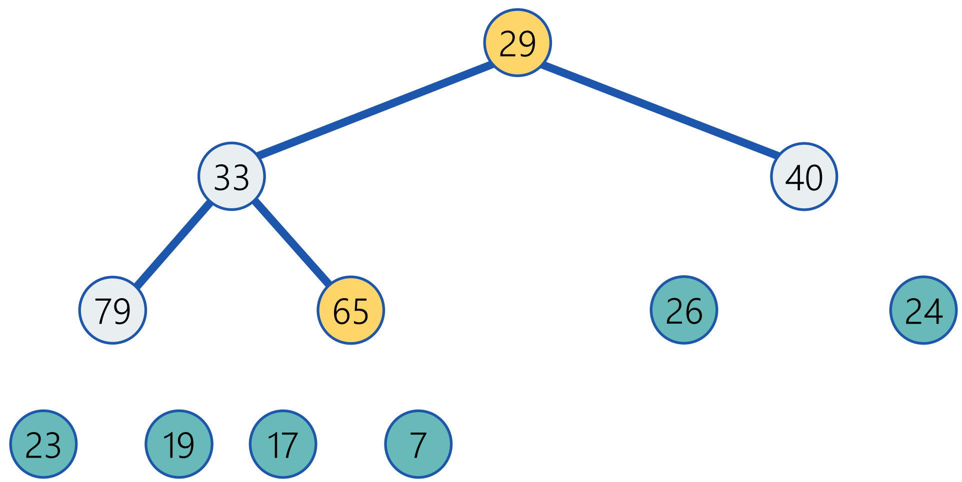


Heapsort



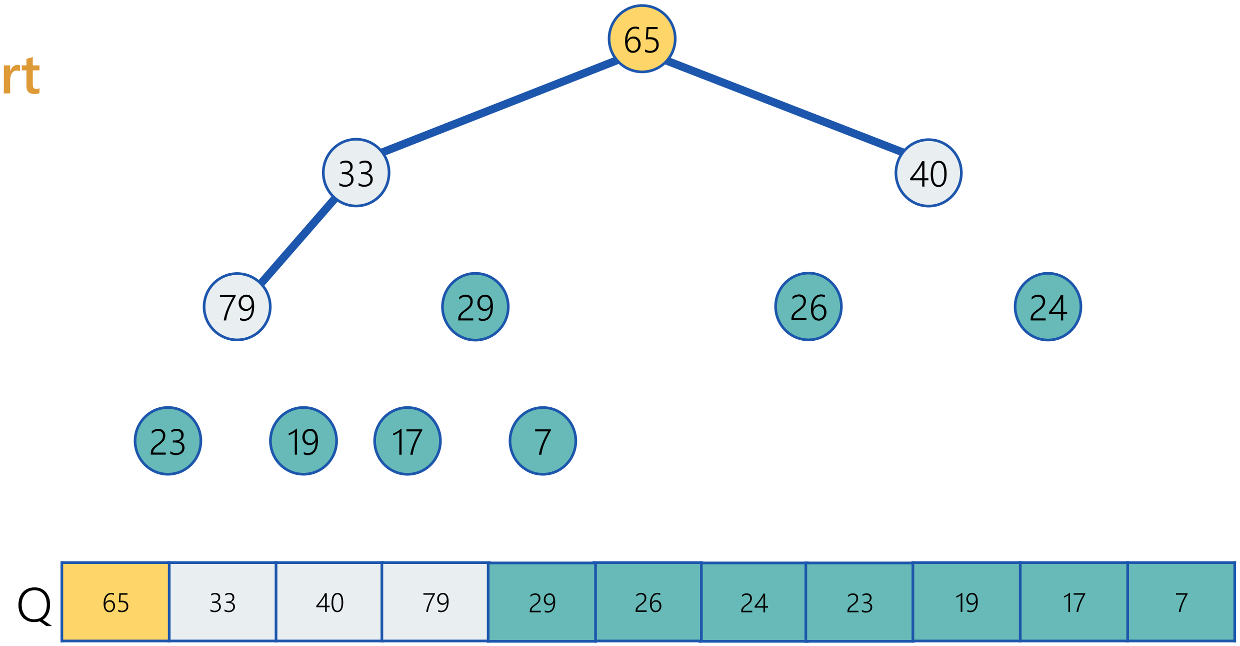
Q	29	33	40	79	65	26	24	23	19	17	7

Heapsort

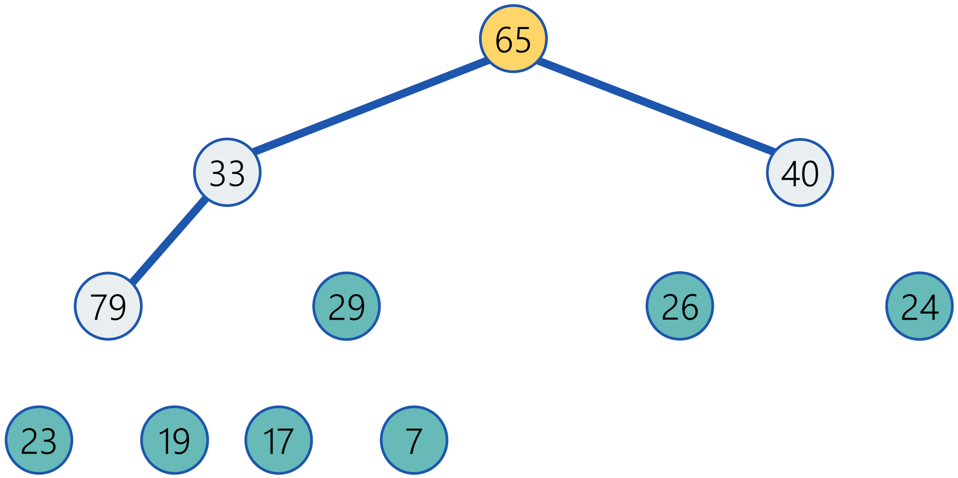


Q											
	29	33	40	79	65	26	24	23	19	17	7

Heapsort

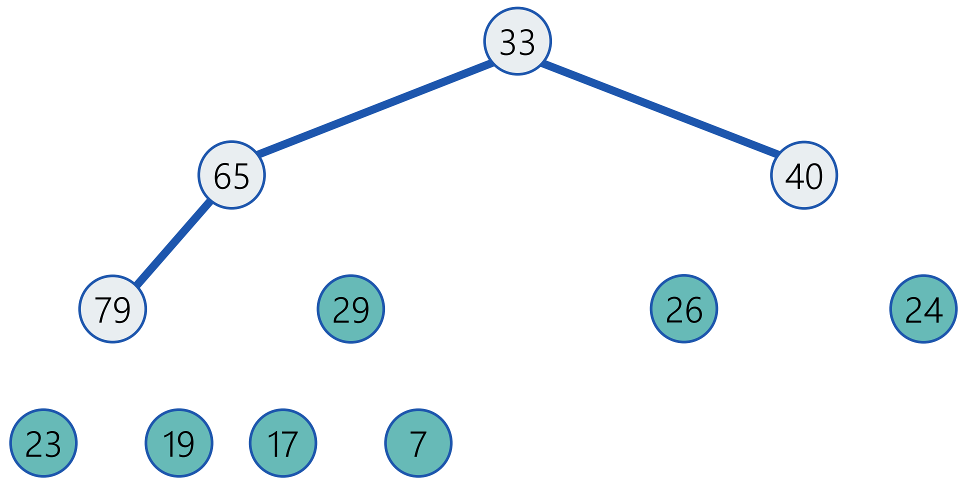


Heapsort



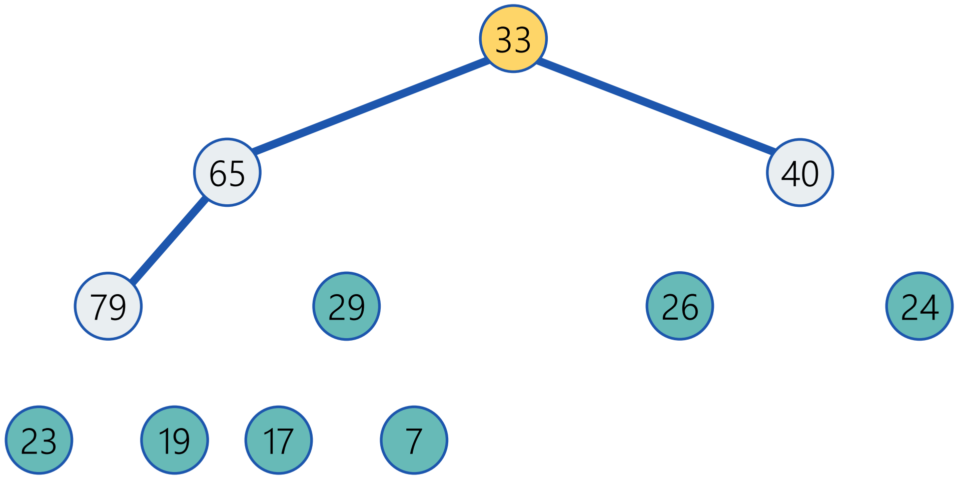
Q										
65	33	40	79	29	26	24	23	19	17	7

Heapsort



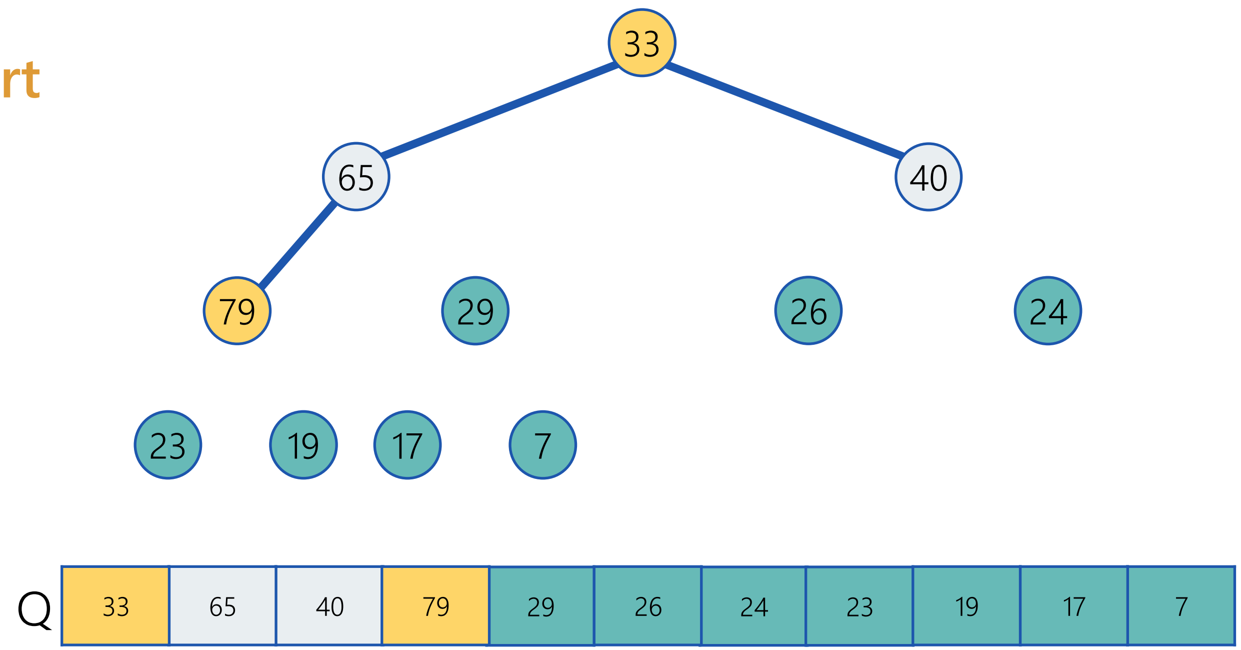
Q										
33	65	40	79	29	26	24	23	19	17	7

Heapsort

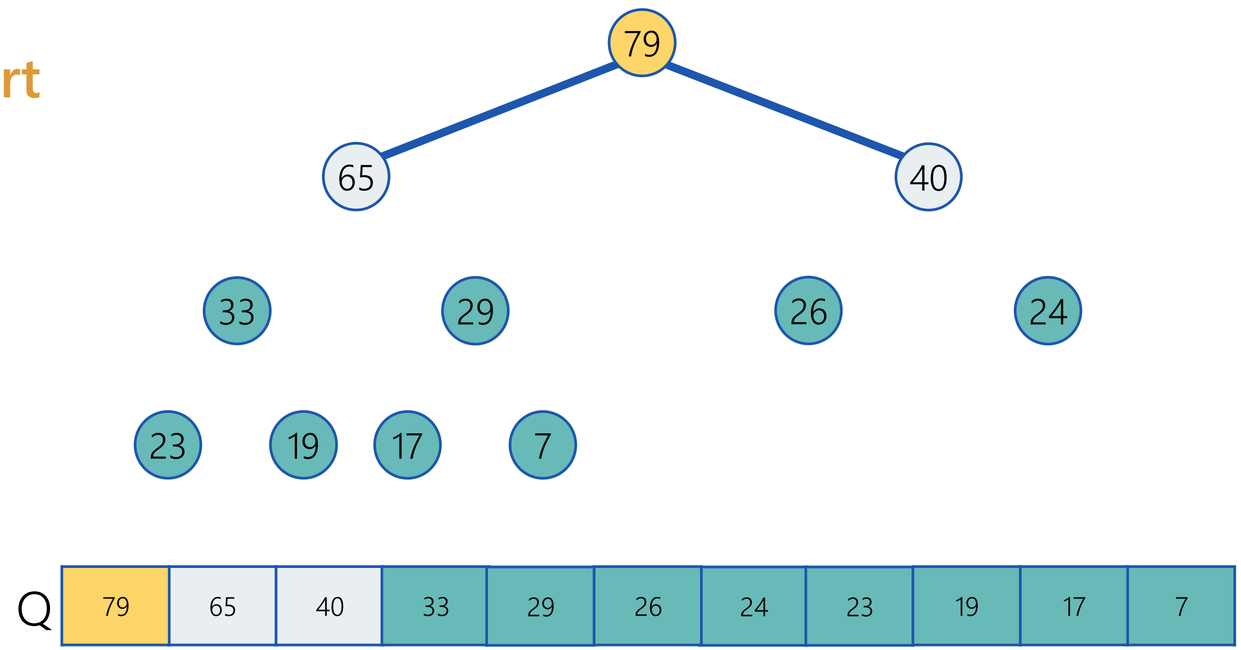


Q										
33	65	40	79	29	26	24	23	19	17	7

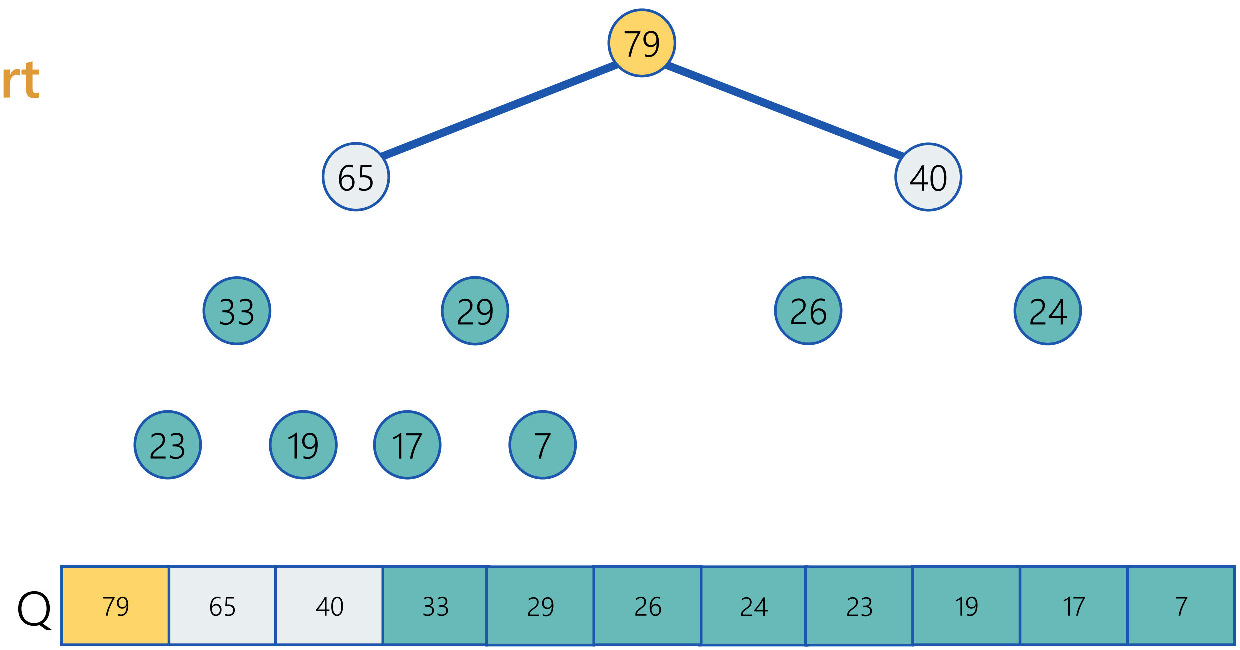
Heapsort



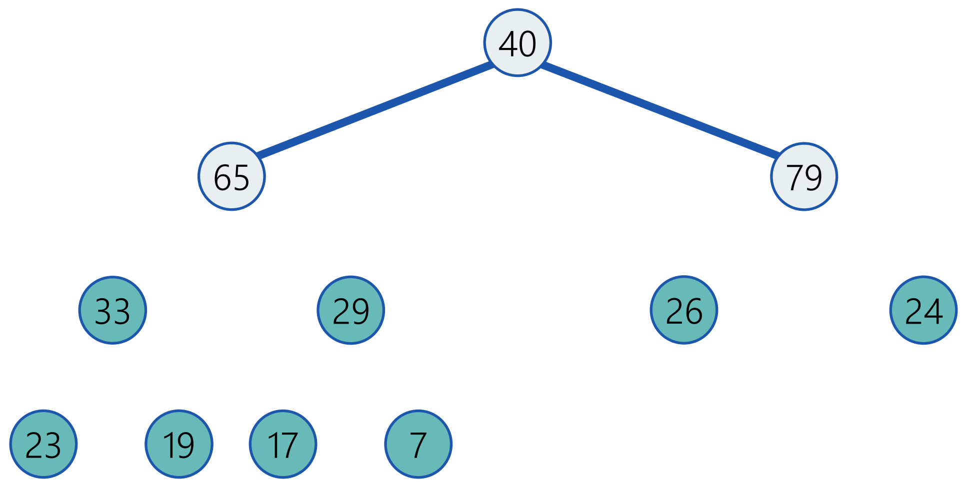
Heapsort



Heapsort

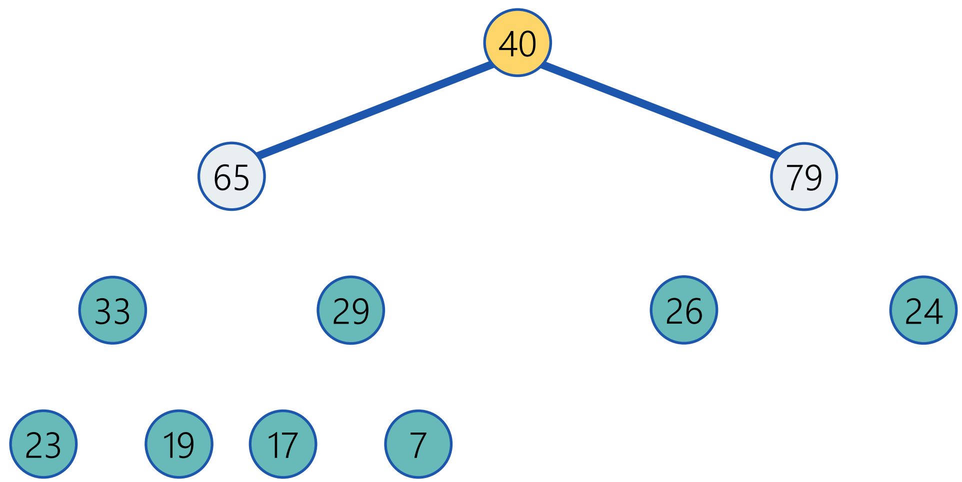


Heapsort



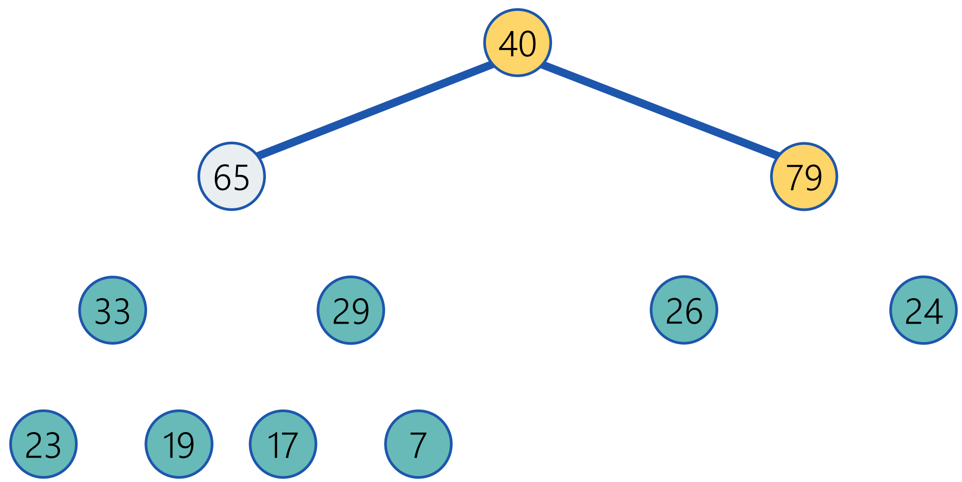
Q	40	65	79	33	29	26	24	23	19	17	7

Heapsort



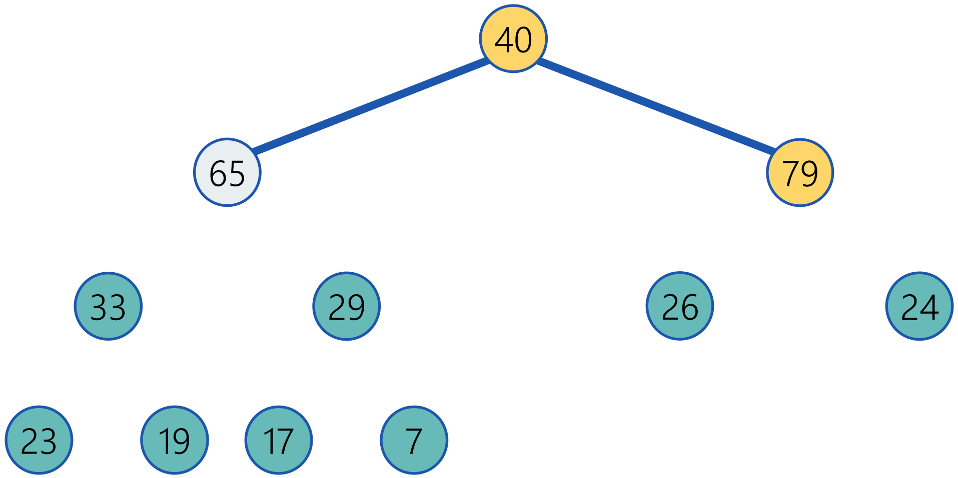
Q											
	40	65	79	33	29	26	24	23	19	17	7

Heapsort



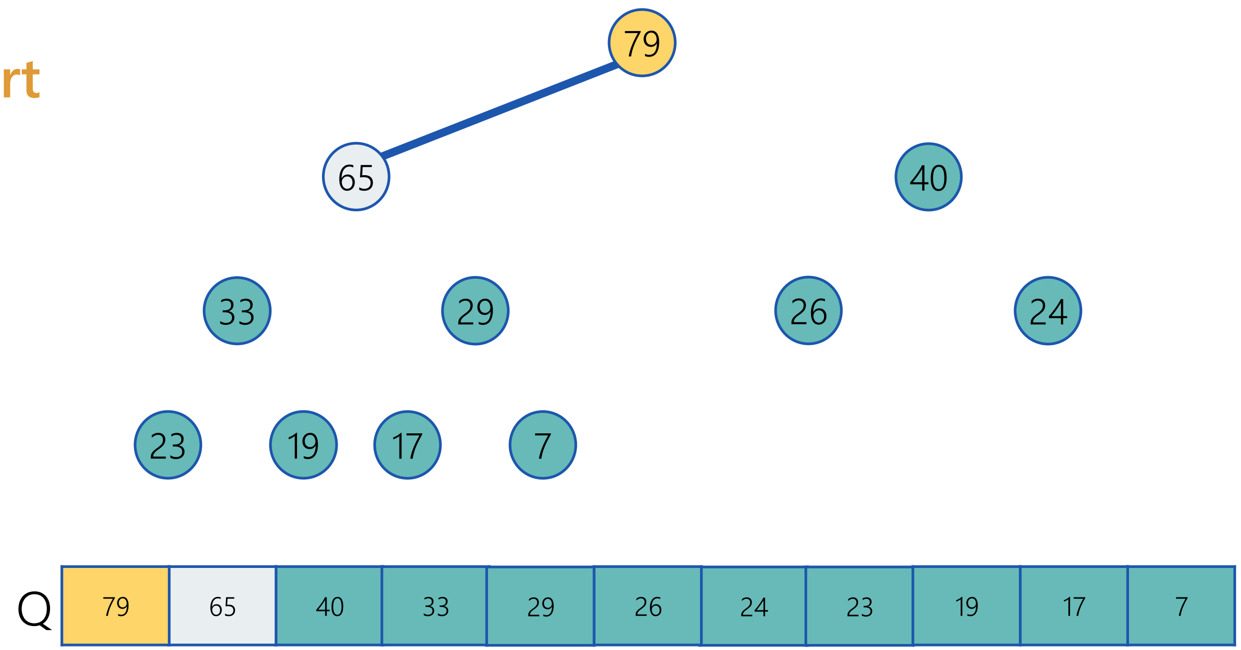
Q	40	65	79	33	29	26	24	23	19	17	7

Heapsort

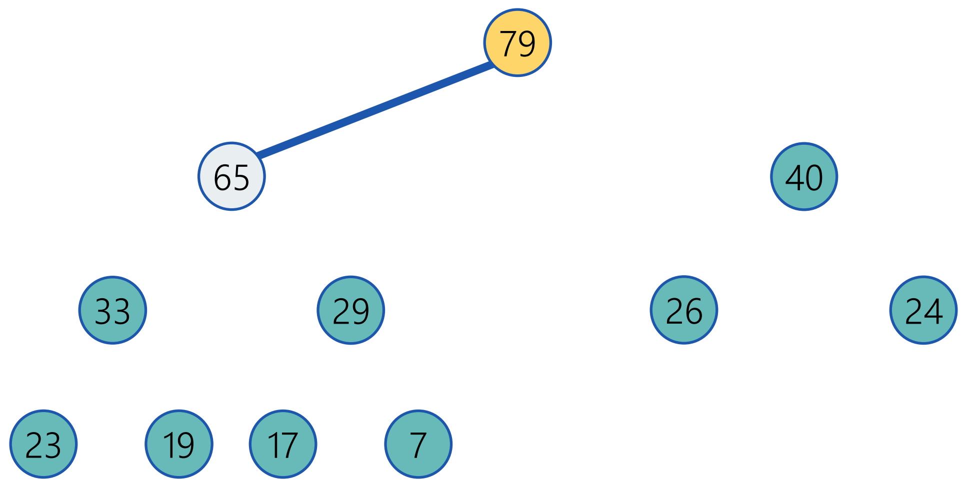


Q	40	65	79	33	29	26	24	23	19	17	7

Heapsort

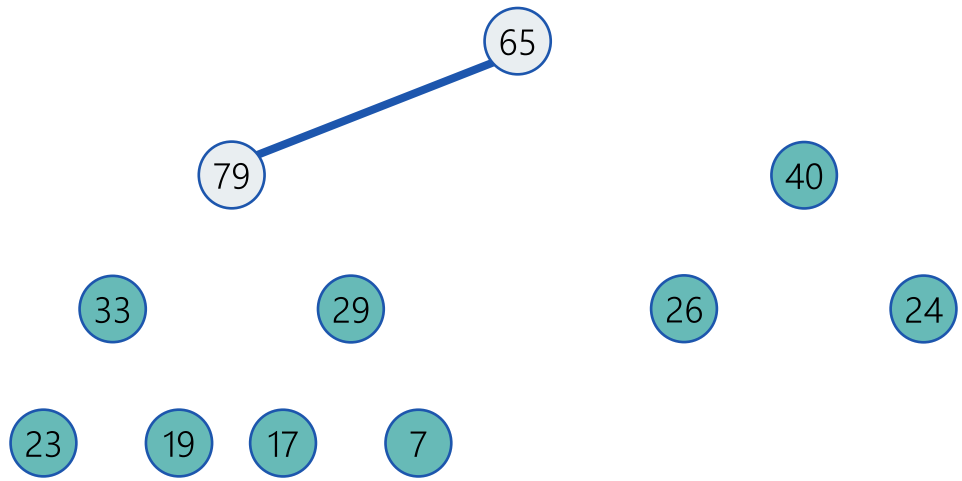


Heapsort



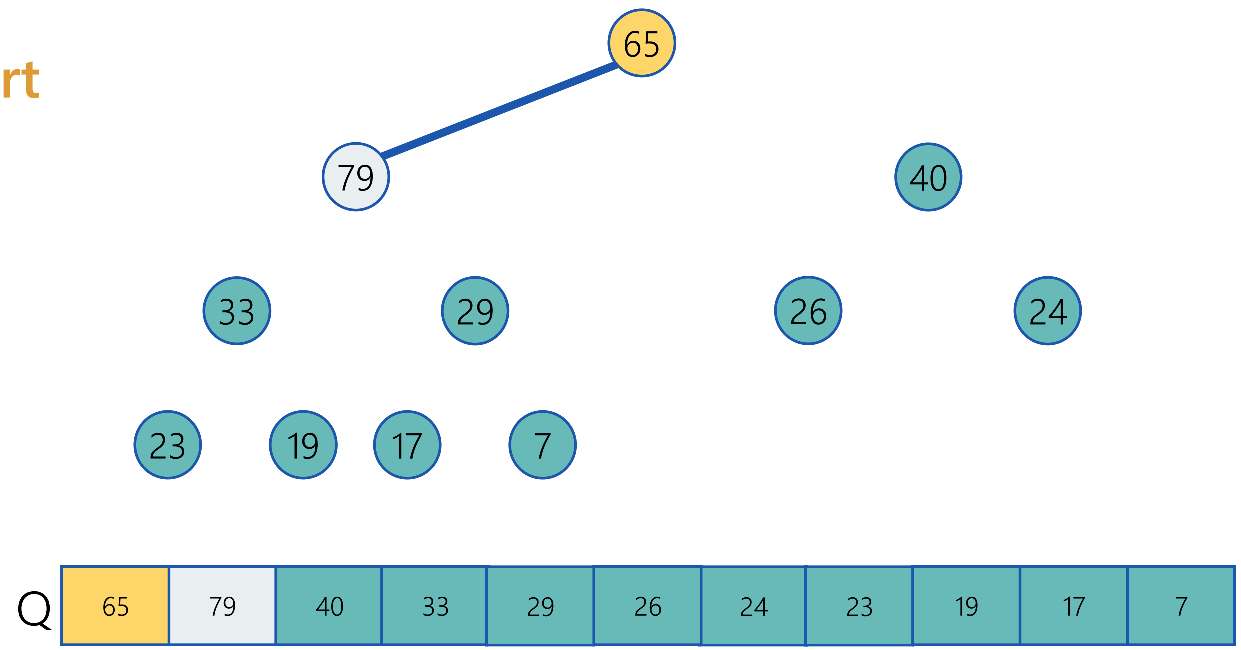
Q	79	65	40	33	29	26	24	23	19	17	7

Heapsort

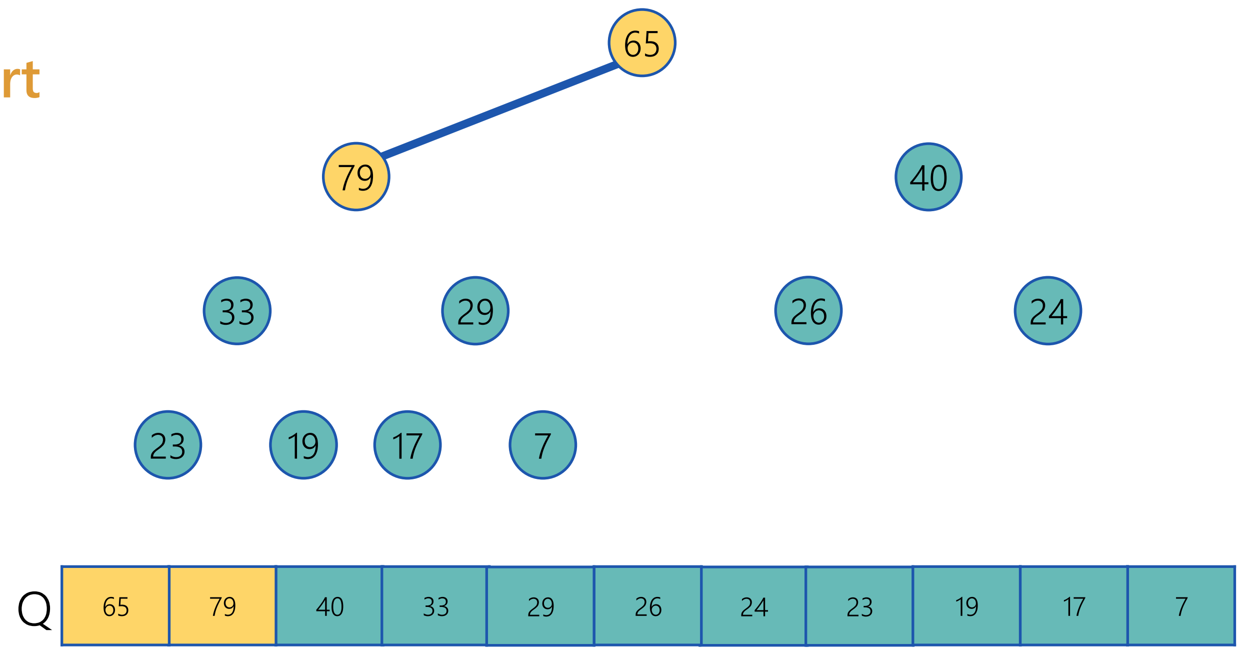


Q	65	79	40	33	29	26	24	23	19	17	7

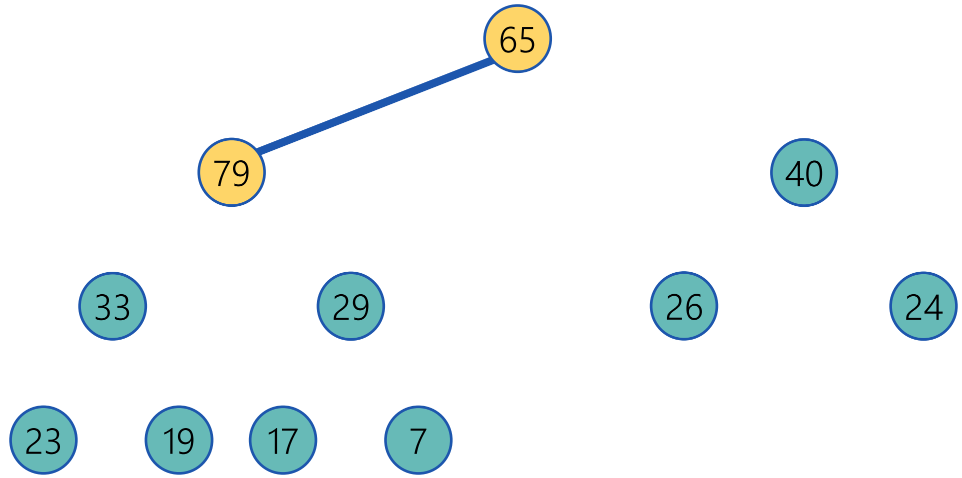
Heapsort



Heapsort

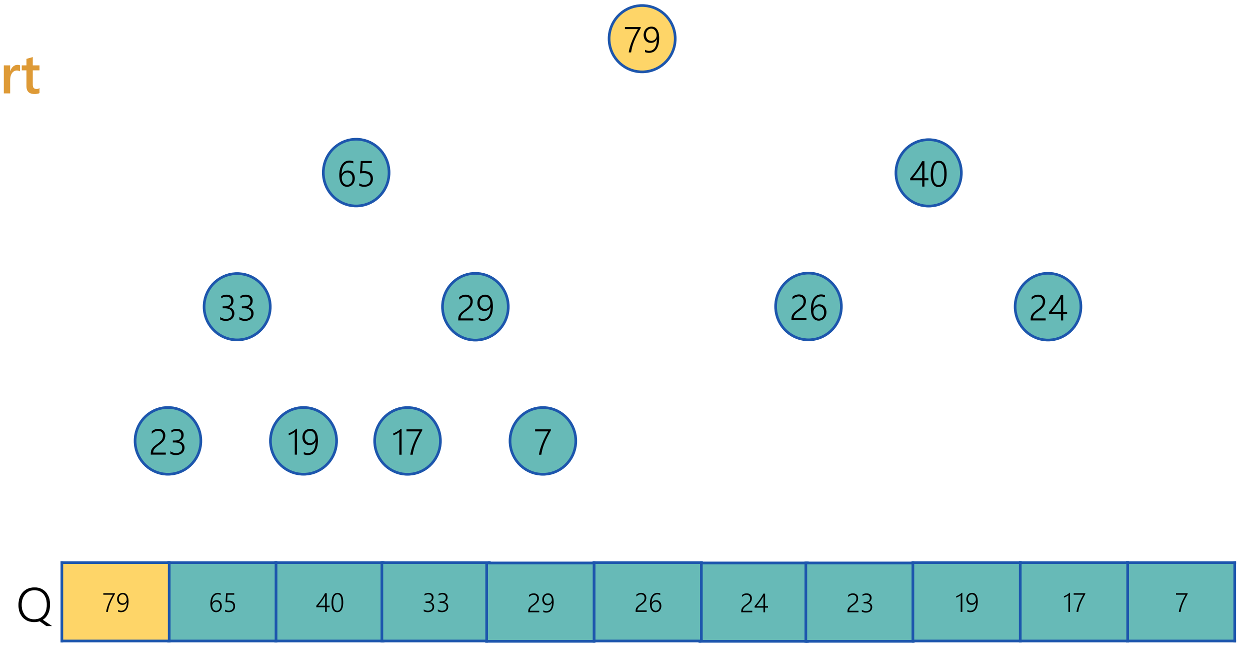


Heapsort

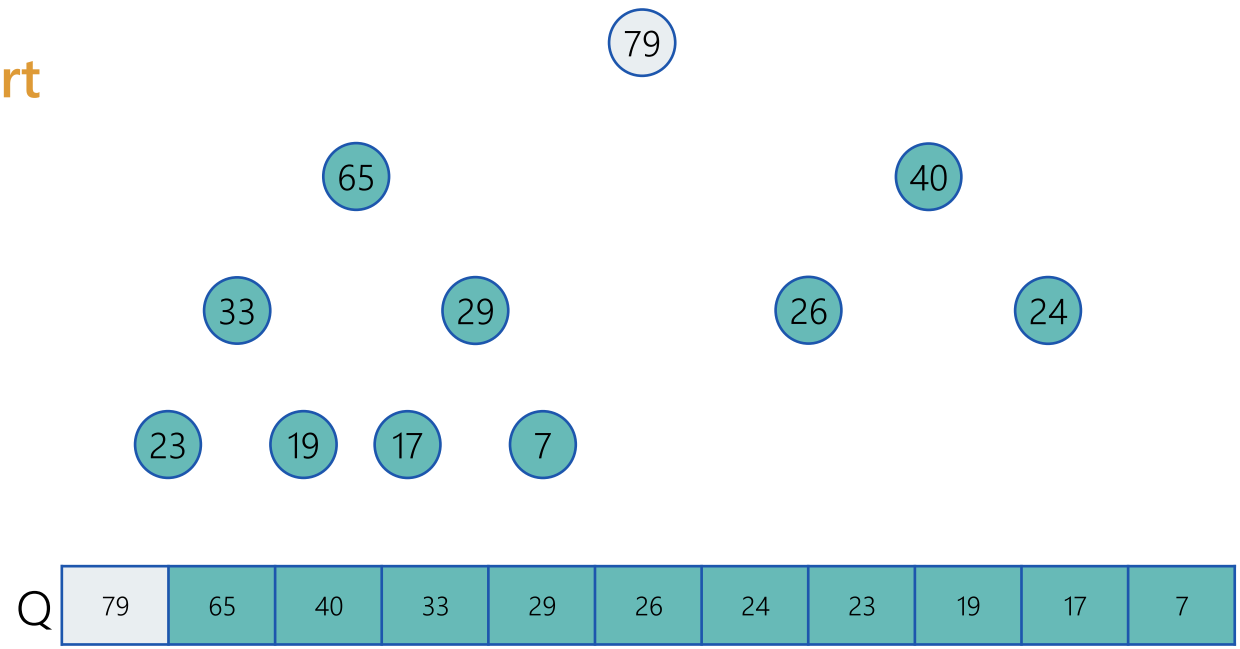


Q	65	79	40	33	29	26	24	23	19	17	7

Heapsort



Heapsort



Heapsort

Q	79	65	40	33	29	26	24	23	19	17	7

- Reverse the array:

Heapsort

Q										
79	65	40	33	29	26	24	23	19	17	7

- Reverse the array:

Q										
7	17	19	23	24	26	29	33	40	65	79

- You may also want to look at this animation which uses a max-heap:

  https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Heapsort Complexity

- Input: array with n elements
- Output: array sorted

- Algorithm:
  - Create min-heap from input $O(n)$
  - Do delete-min, and put the deleted element at the last position of the array. Repeat n times. $O(nlogn)$
  - Reverse the array (or use a max-heap instead) $O(n)$

- Overall time complexity $O(n \log n)$, additional memory $O(1)$

Application of Heaps

- Input: array with n elements, index k
- Output: The k smallest elements in the array

- Algorithm:
  - Create min-heap from input $O(n)$
  - Do delete-min, and put the deleted element at the last position of the array. Repeat k times. $O(klogn)$

- Overall time complexity $O(n + k \log n)$

Binomial Heaps

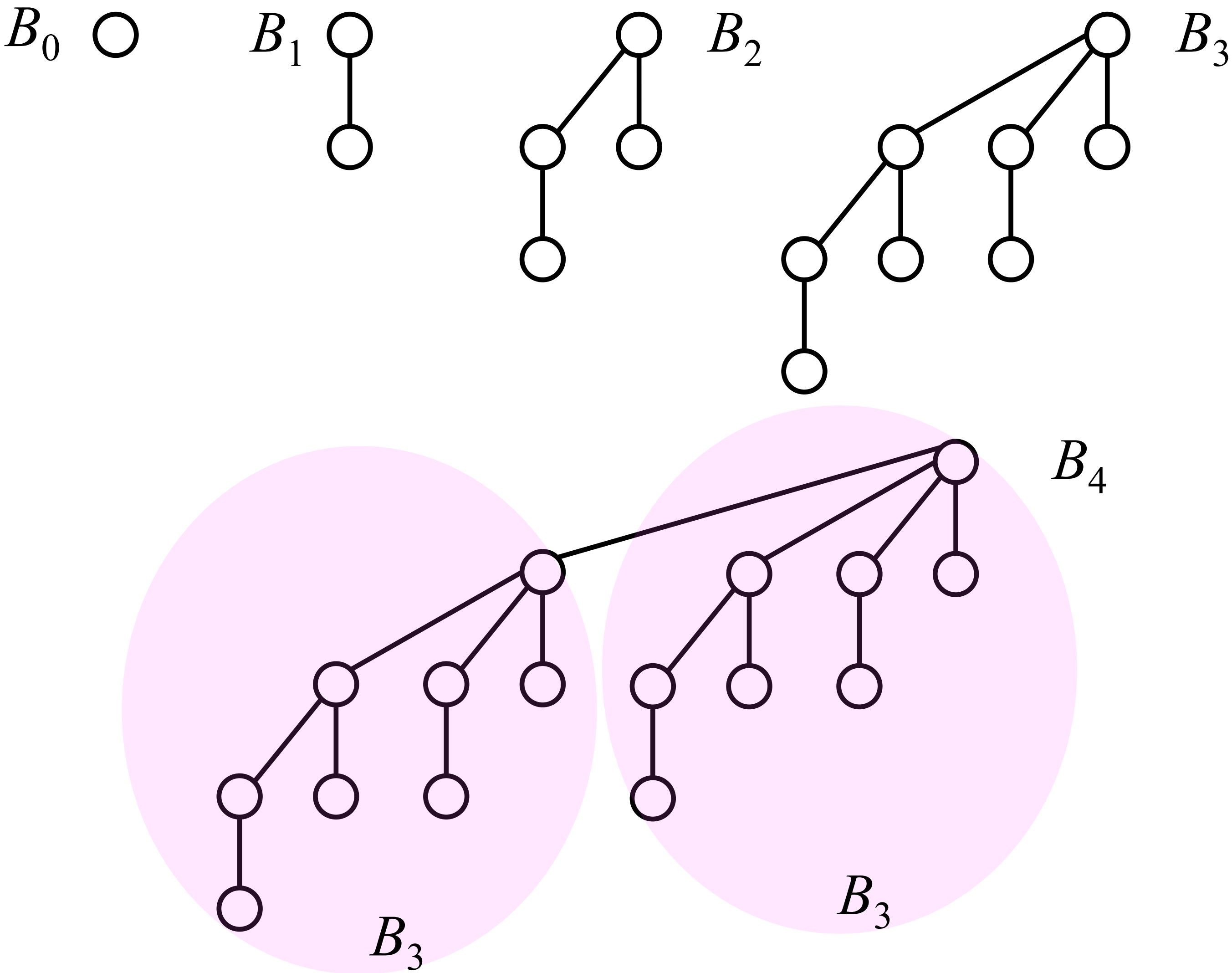
[Vuillemin (1978)]

Binomial Heaps

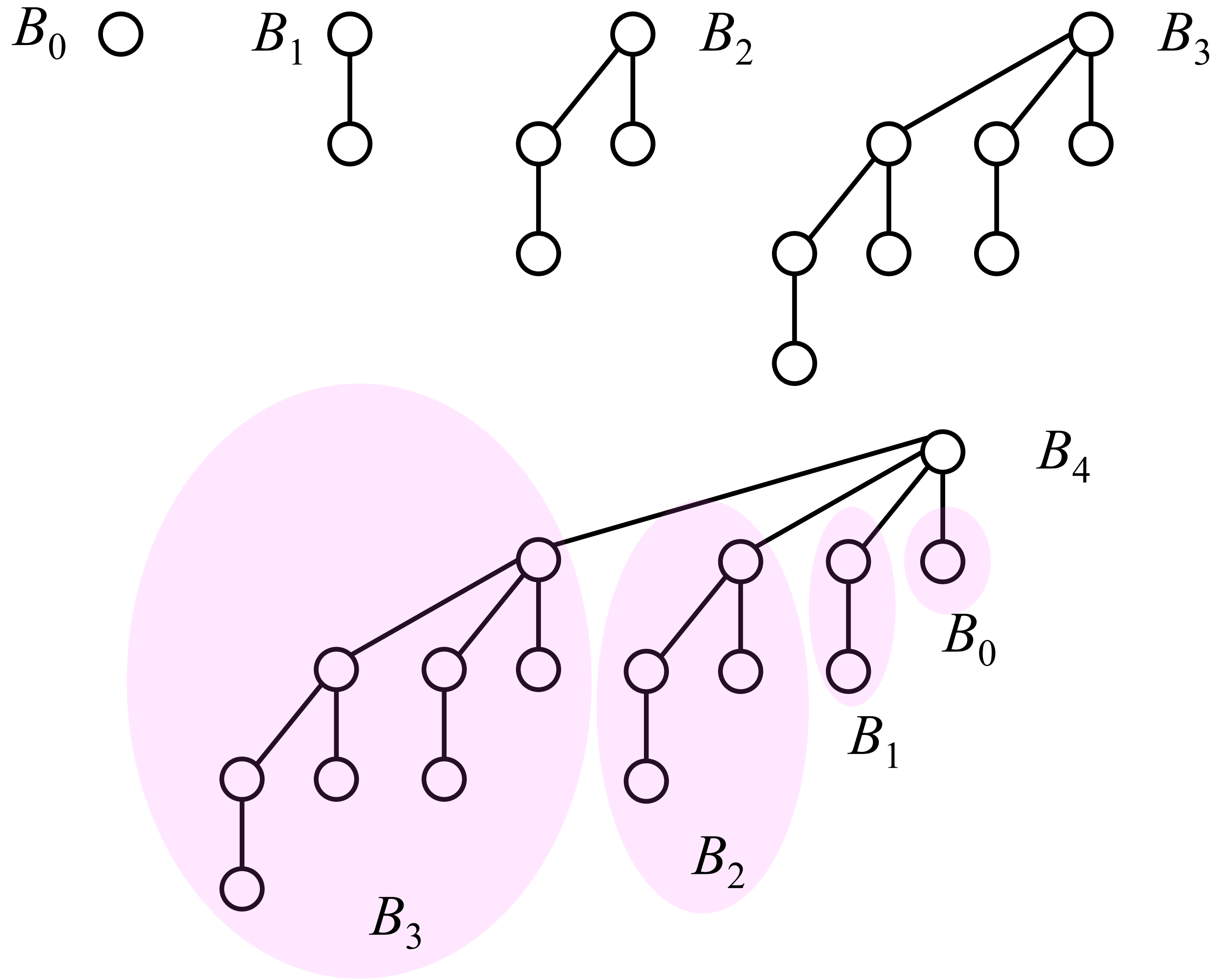
	Binary Heaps ➡	Binomial Heaps ➡	Lazy Binomial Heaps ➡	Fibonacci Heaps
Insert	$O(\log n)$	←		
Find-min	$O(1)$	←		
Delete-min	$O(\log n)$	←		
Decrease-key	$O(\log n)$	←		
Meld / Join	$O(n)$	**$O(\log n)$**		

Worst case — Amortized

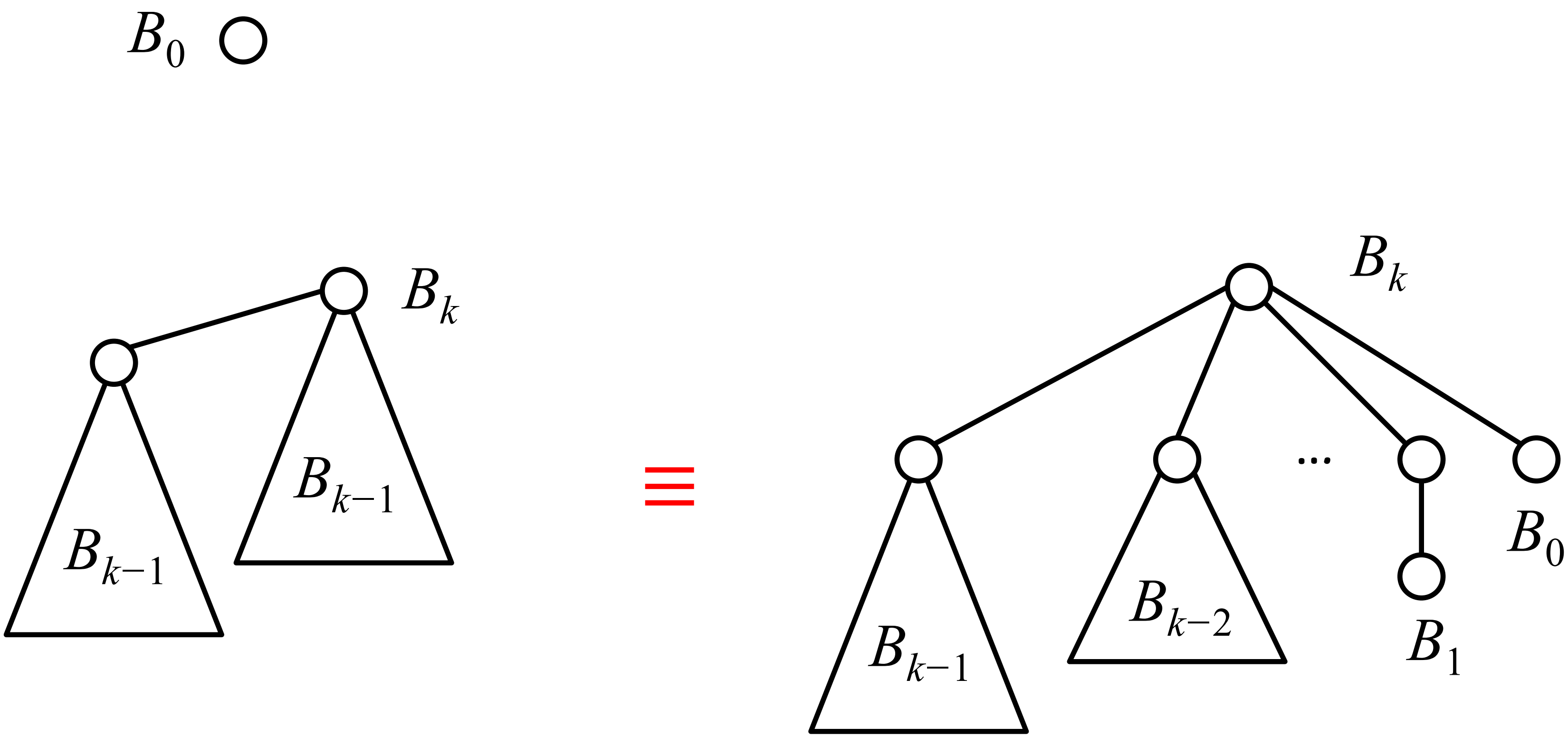
Binomial Trees - definition



Binomial Trees – another view

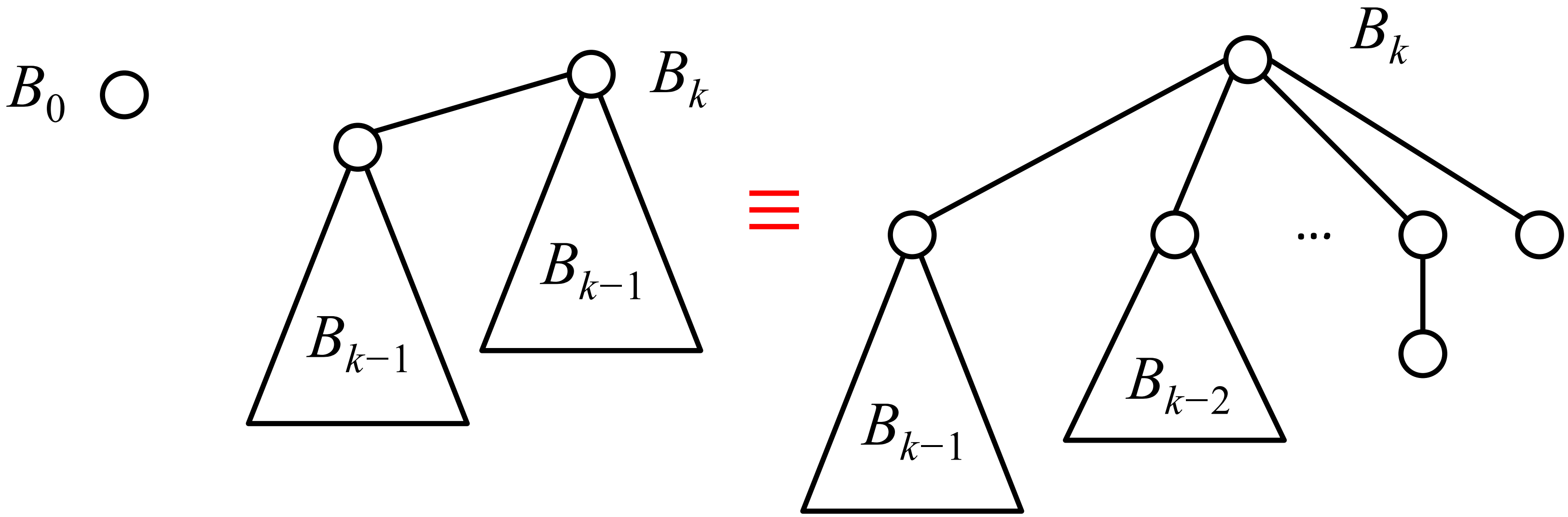


Binomial Trees - definition



We say that B_k is a binomial tree of degree k
(the root has k children)

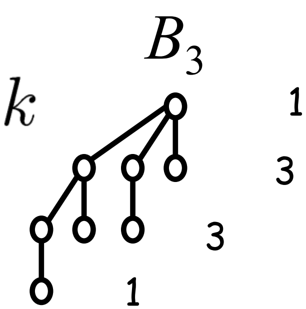
Binomial Trees - properties



1) The root of B_k has k children

2) B_k contains 2^k nodes and its depth is k

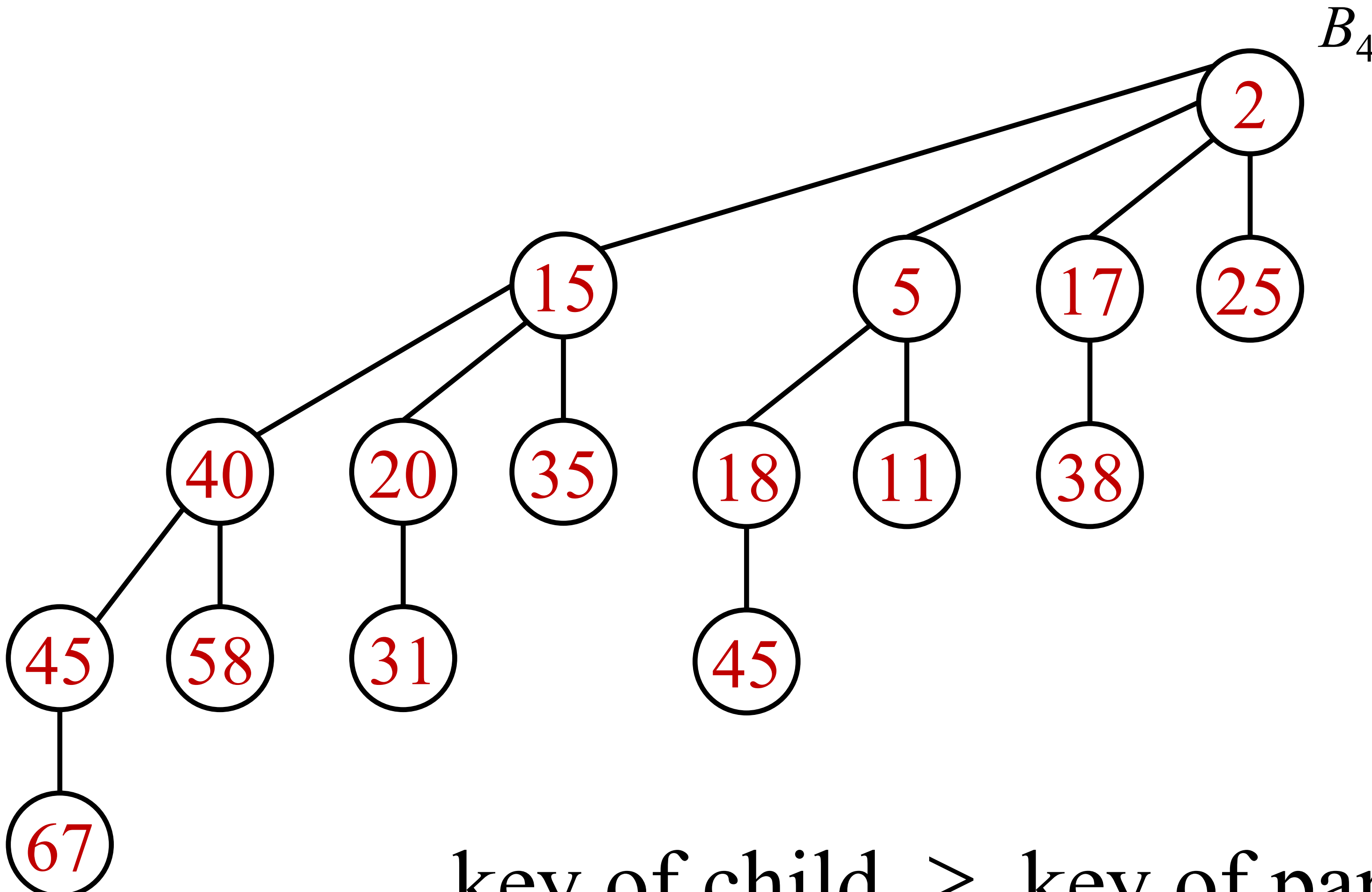
3) $\binom{k}{i}$ of the nodes of B_k are at level i



Exercise: prove all 3 properties

recall: $\sum_{i=0}^{k} \binom{k}{i} = 2^k$ $\qquad \binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$

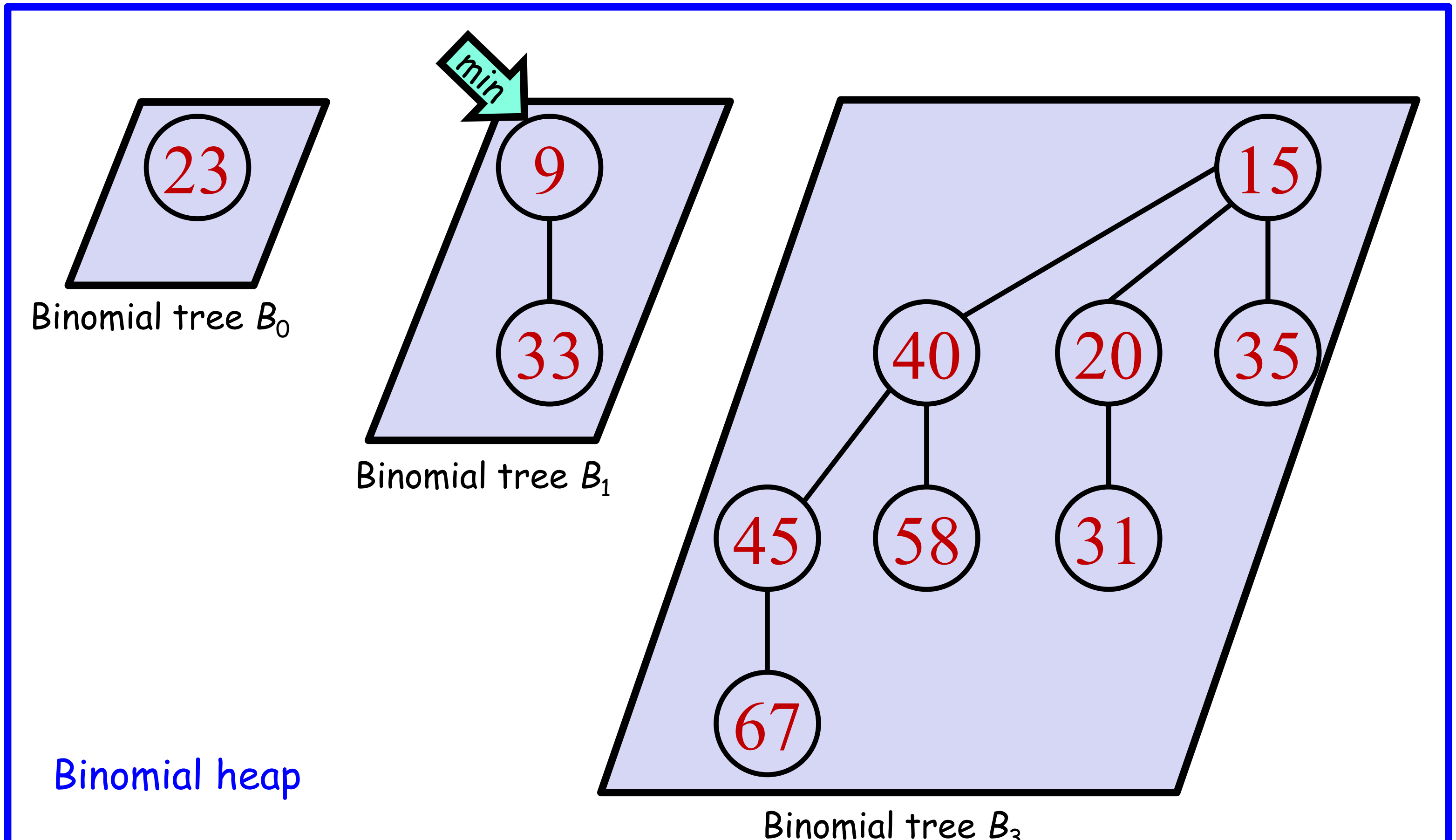
Min-heap Ordered Binomial Trees



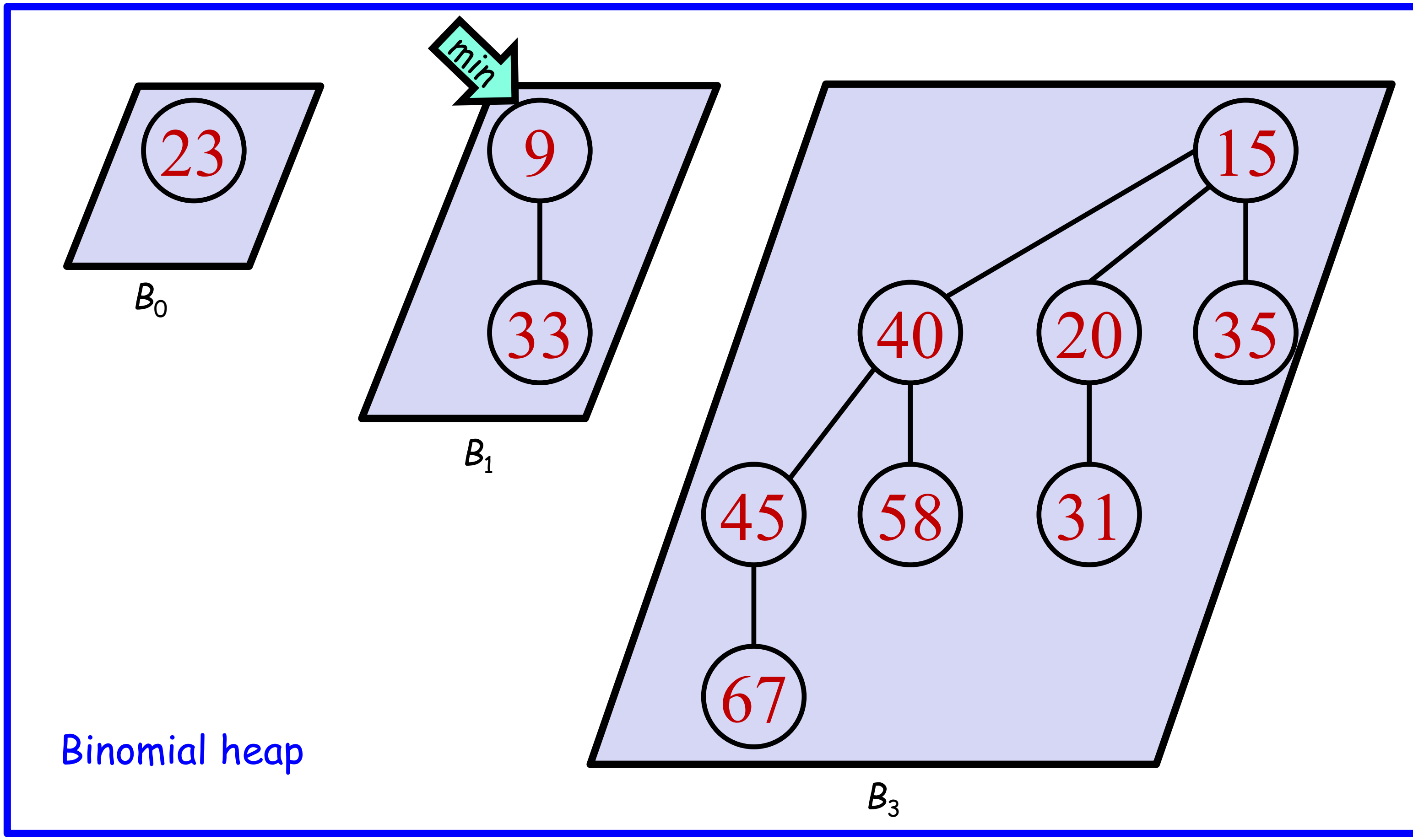
key of child $\geq$ key of parent

Binomial Heap - definition

A list of heap-ordered binomial trees,
at most one of each degree
+ pointer to root with minimal key



Binomial Heap - definition



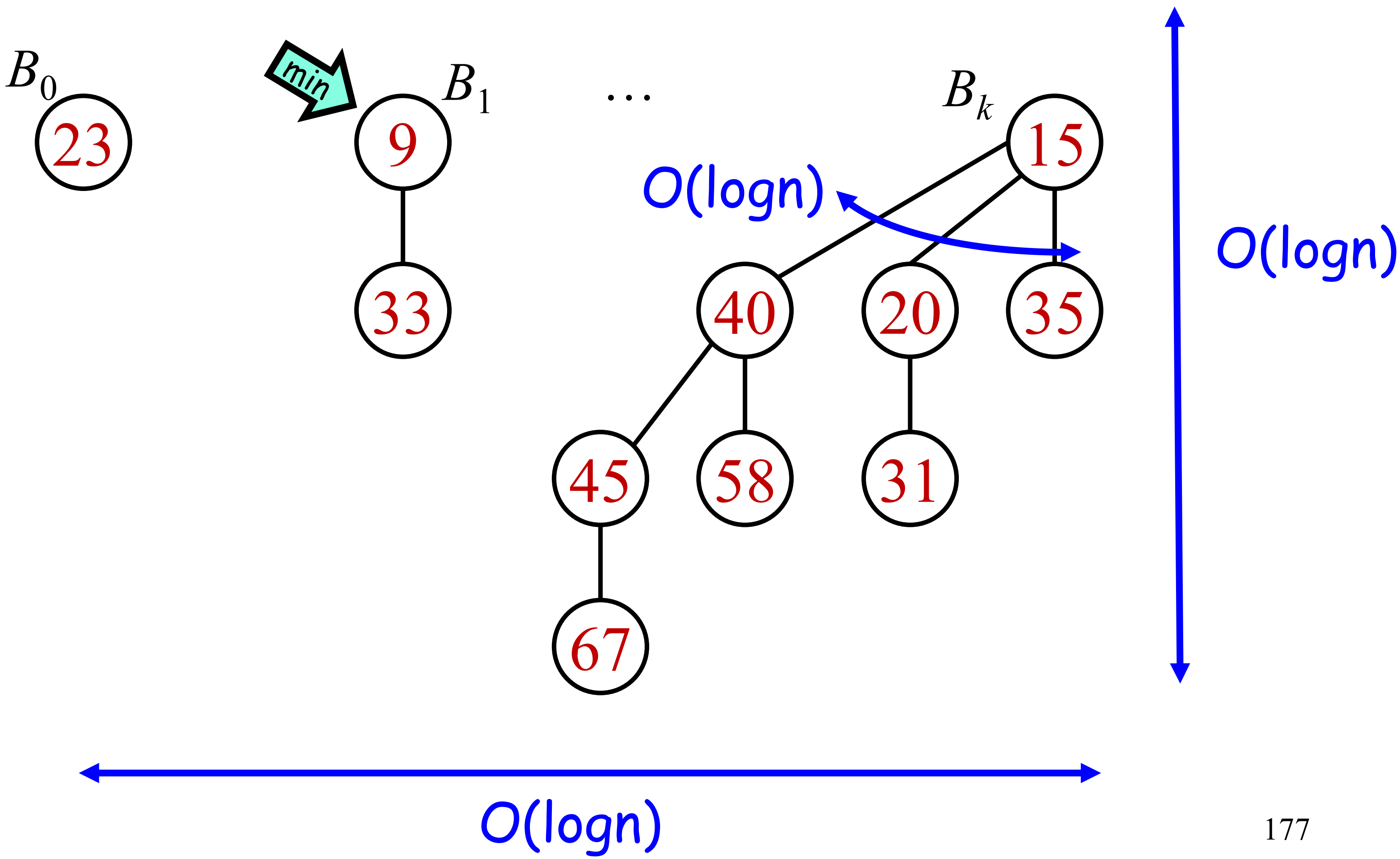
Each integer n can be written in a unique way as a sum of powers of 2:

$n = 11_{(10)} = 1011_{(2)} = 8{+}2{+}1$

At most $\lfloor \log_2 n \rfloor + 1$ trees

Binomial Heap - Intuition

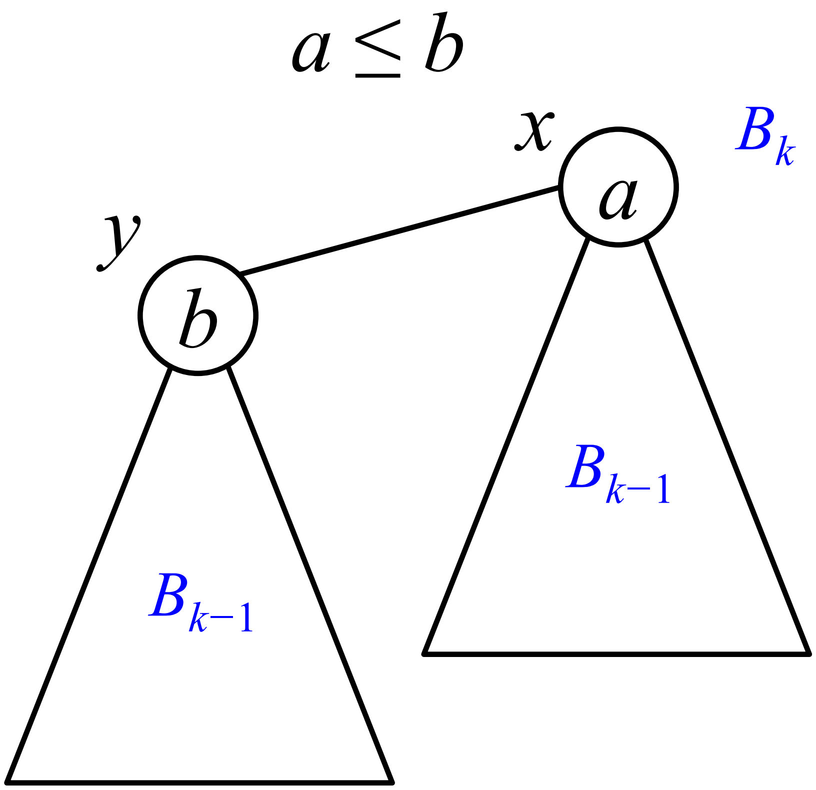
- A binomial heap is neither too wide nor too deep



Linking binomial trees of same degree

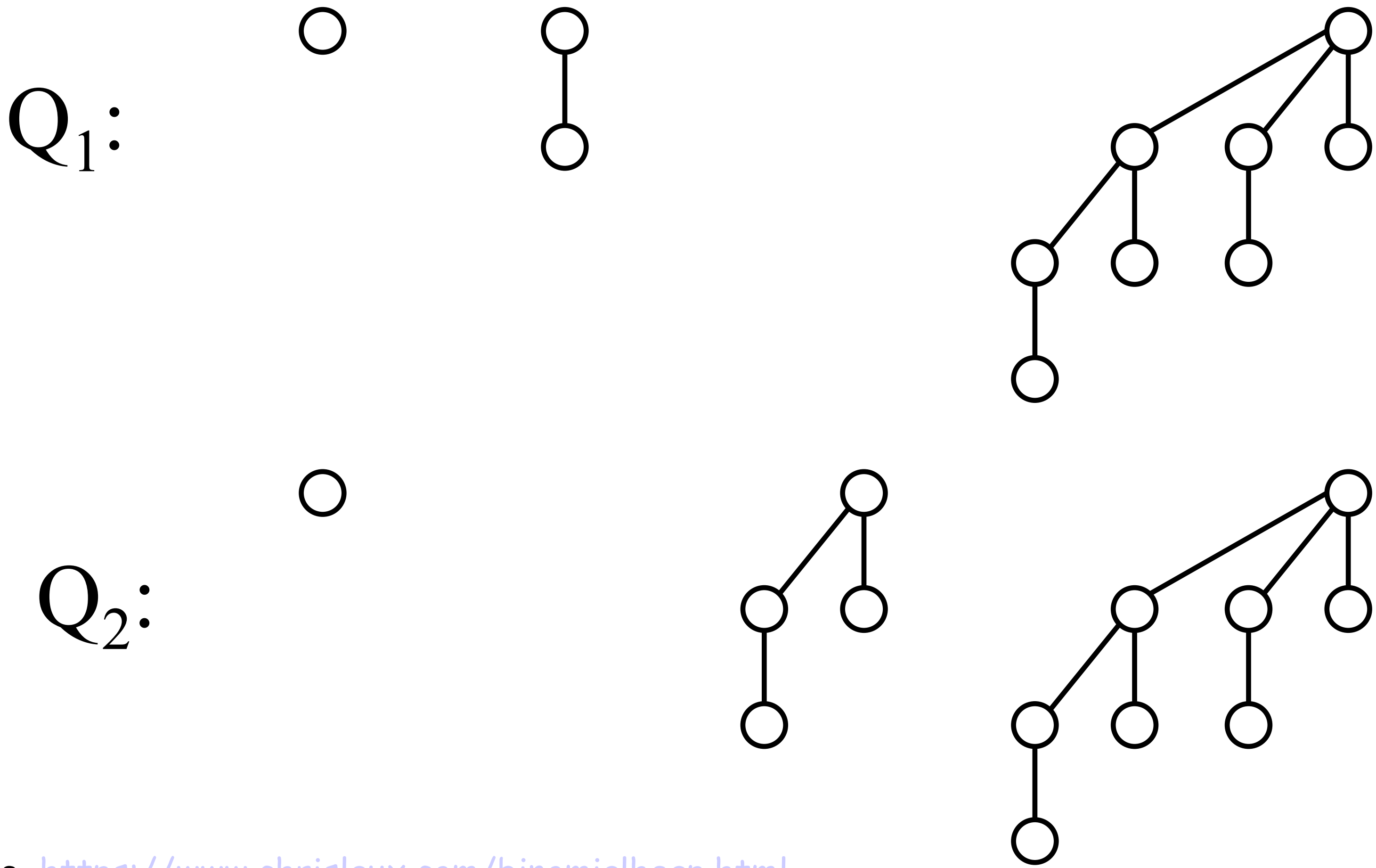
Function `link`(x, y)
if $x.key > y.key$ **then**
$\quad x \leftrightarrow y$
$y.next \leftarrow x.child$
$x.child \leftarrow y$
return x

O(1) time



Melding binomial heaps in $O(\log n)$

Link trees of same degree



see https://www.chrislaux.com/binomialheap.html

Melding binomial heaps in $O(\log n)$

Link trees of same degree

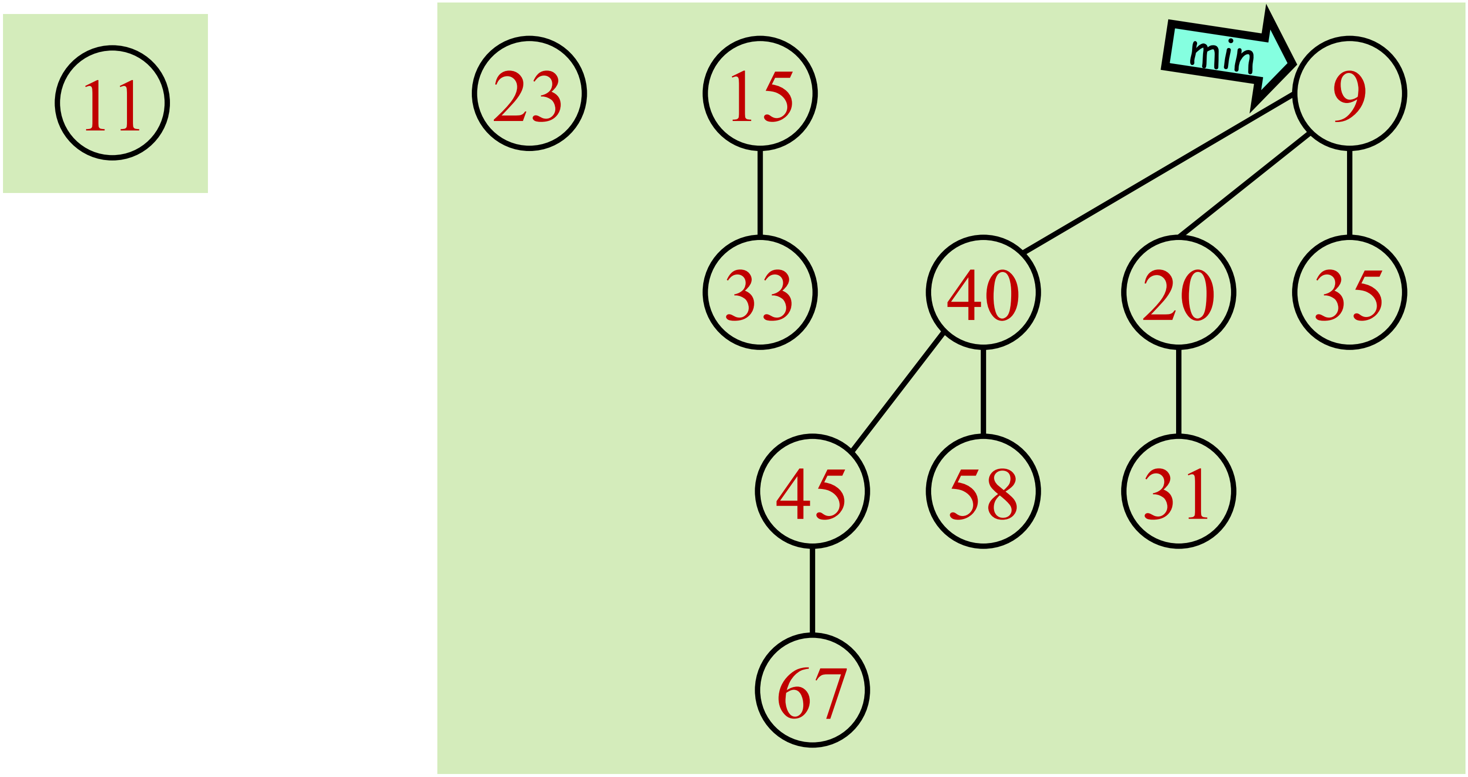
$$
\begin{array}{lccccc}
 & & B_1 & B_2 & B_3 & \\
\mathrm{Q}_1: & B_0 & B_1 & - & B_3 & \\
\mathrm{Q}_2: & B_0 & - & B_2 & B_3 & \\
\hline
 & - & - & - & B_3 & B_4
\end{array}
$$

Like adding binary numbers

Maintain a pointer to the minimum

$O(\log n)$ time

Insert

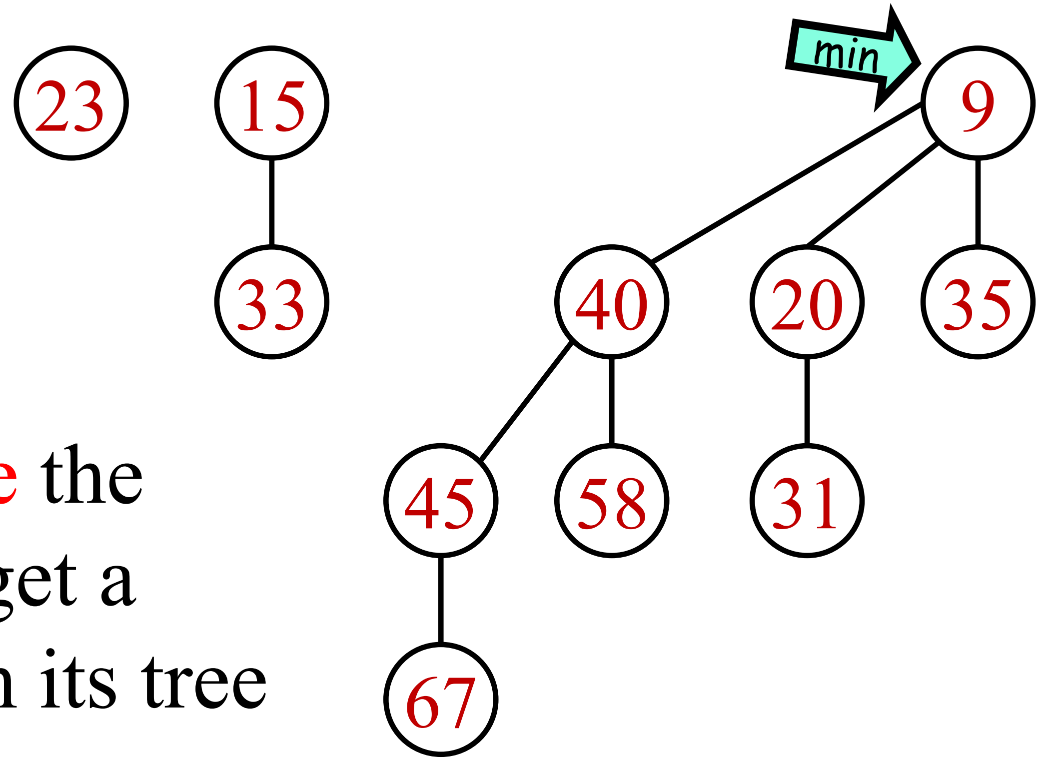


New item is a one-tree binomial heap (B_0)

Meld it to the original heap

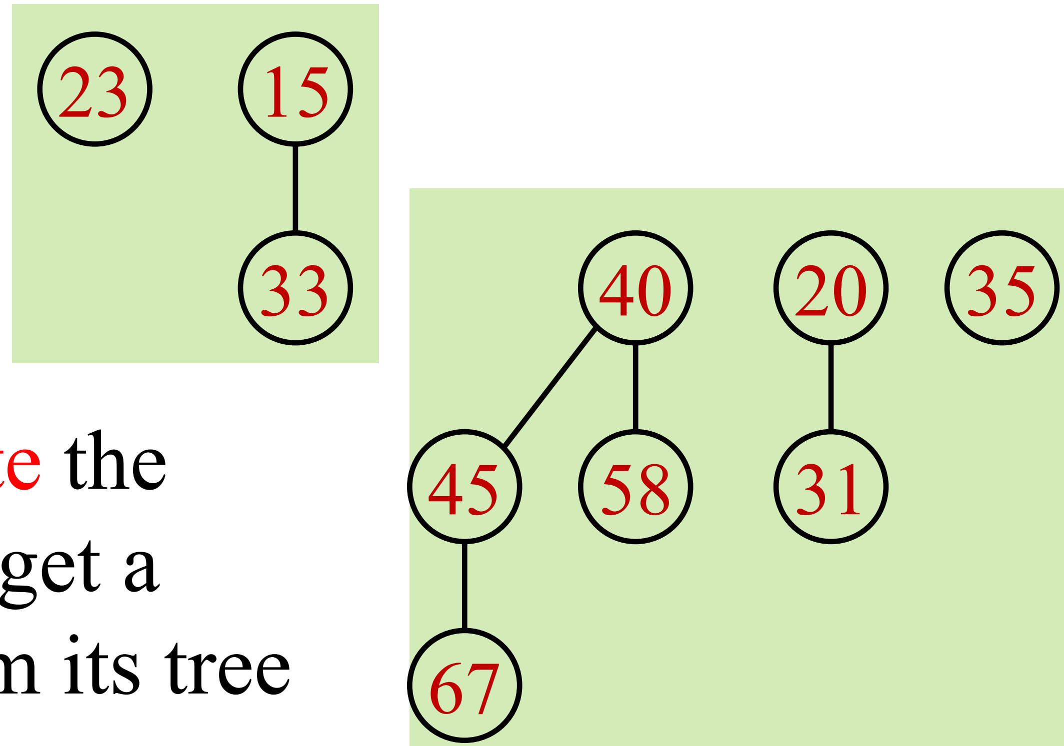
$O(\log n)$ time

Delete-min



When we delete the minimum, we get a binomial heap from its tree

Delete-min



When we delete the minimum, we get a binomial heap from its tree

Meld it to the original heap

$O(\log n)$ time

Linking binomial trees

Function `link`(x, y)

> **if** $x.key > y.key$ **then**
> $x \leftrightarrow y$
> $y.next \leftarrow x.child$
> $x.child \leftarrow y$
> **return** x

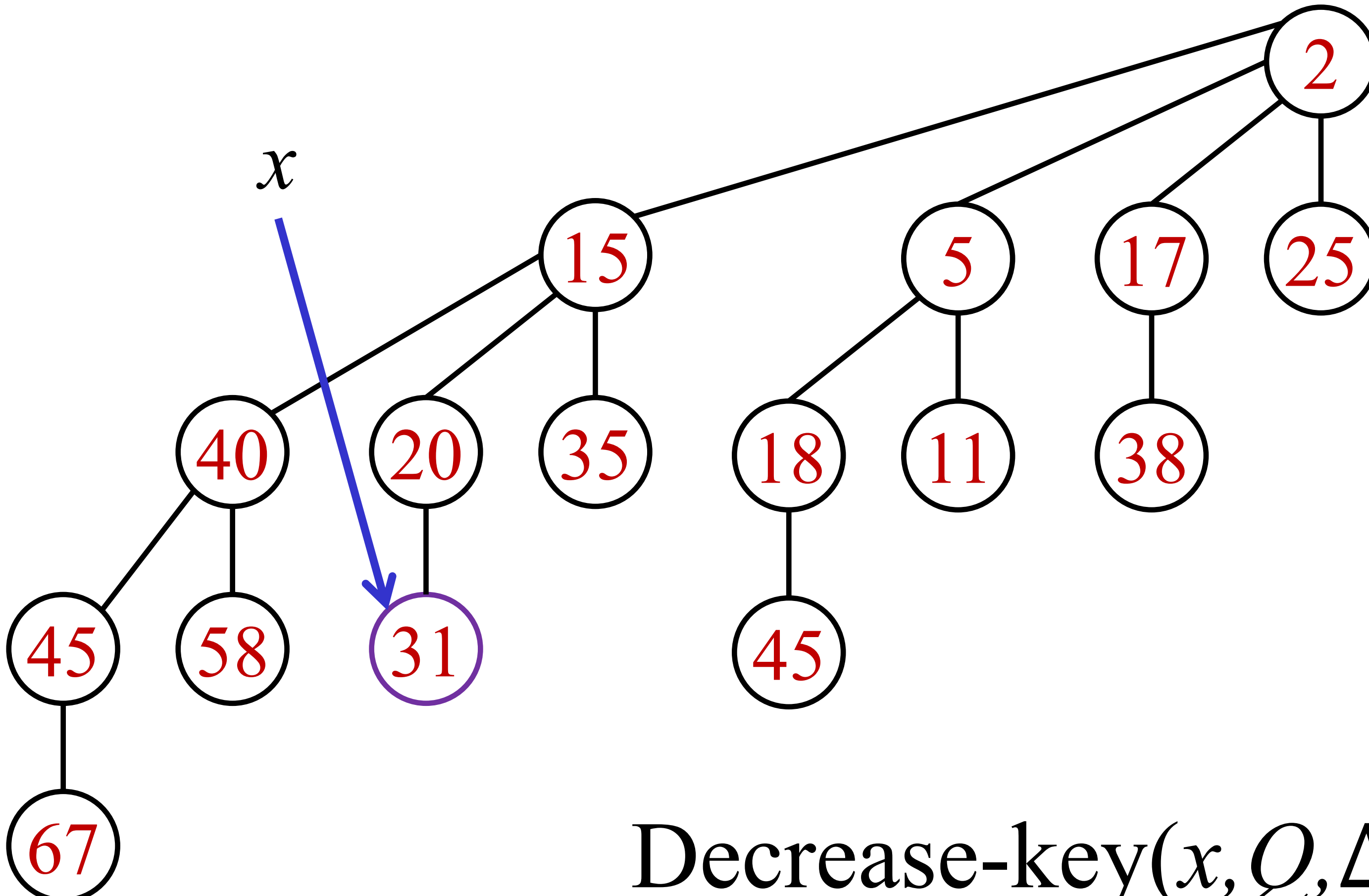
Linking in first representation

Function `link`(x, y)

> **if** $x.key > y.key$ **then**
> $x \leftrightarrow y$
> **if** $x.child = null$ **then**
> $y.next \leftarrow y$
> **else**
> $y.next \leftarrow x.child.next$
> $x.child.next \leftarrow y$
> $x.child \leftarrow y$
> **return** x

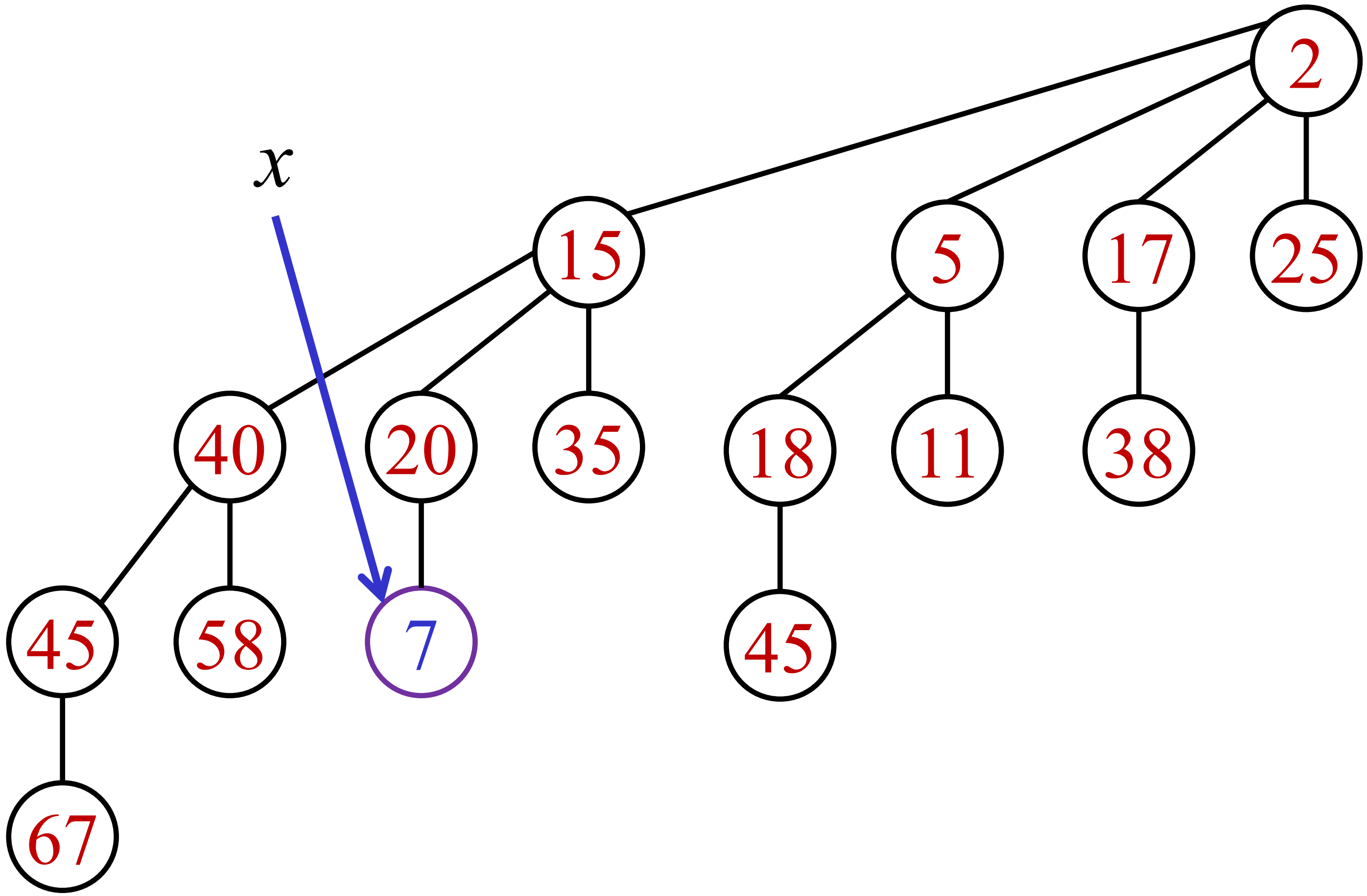
Linking in second representation

Decrease-key using "sift-up"

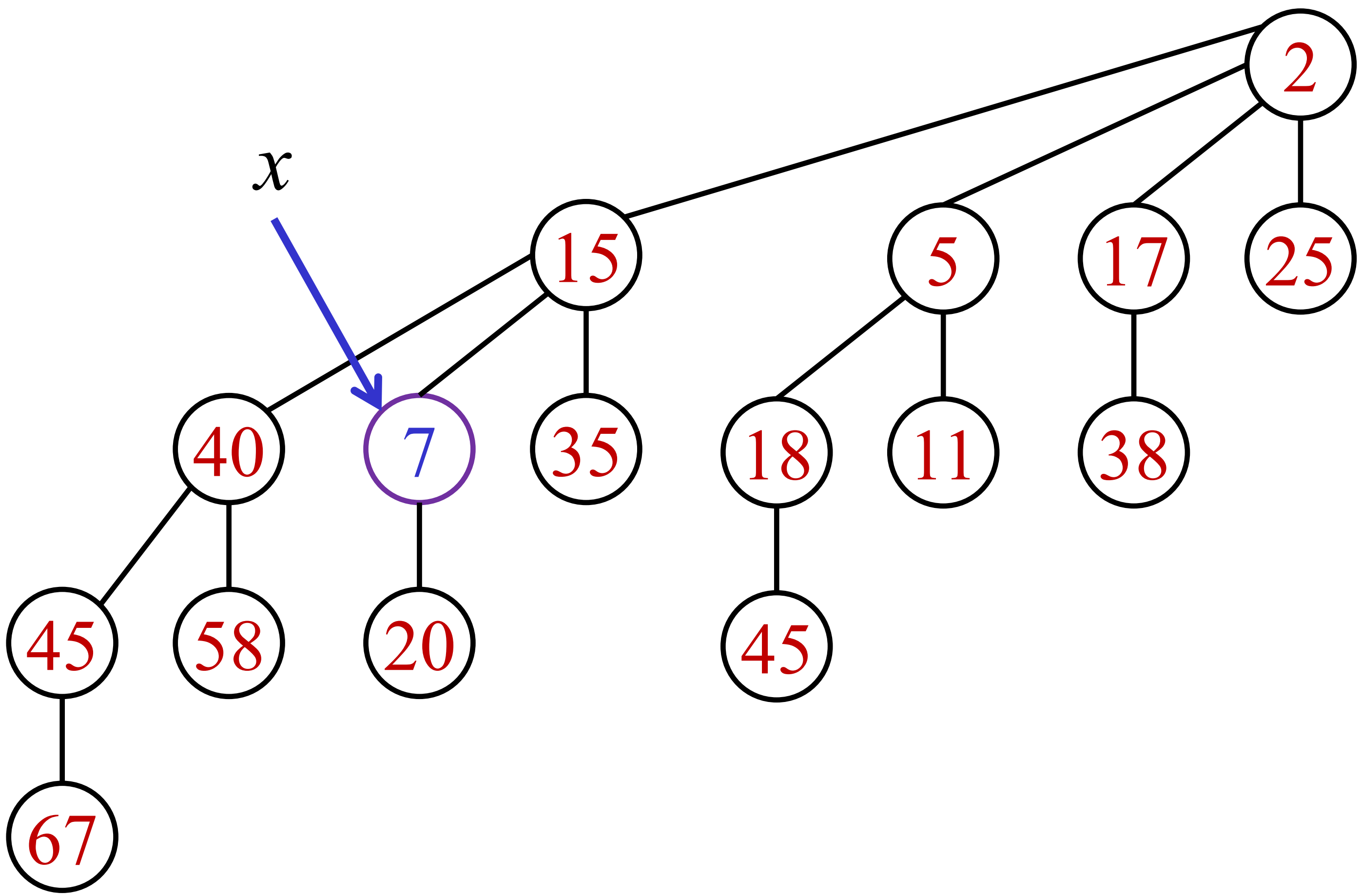


Decrease-key$(x, Q, \Delta = 24)$

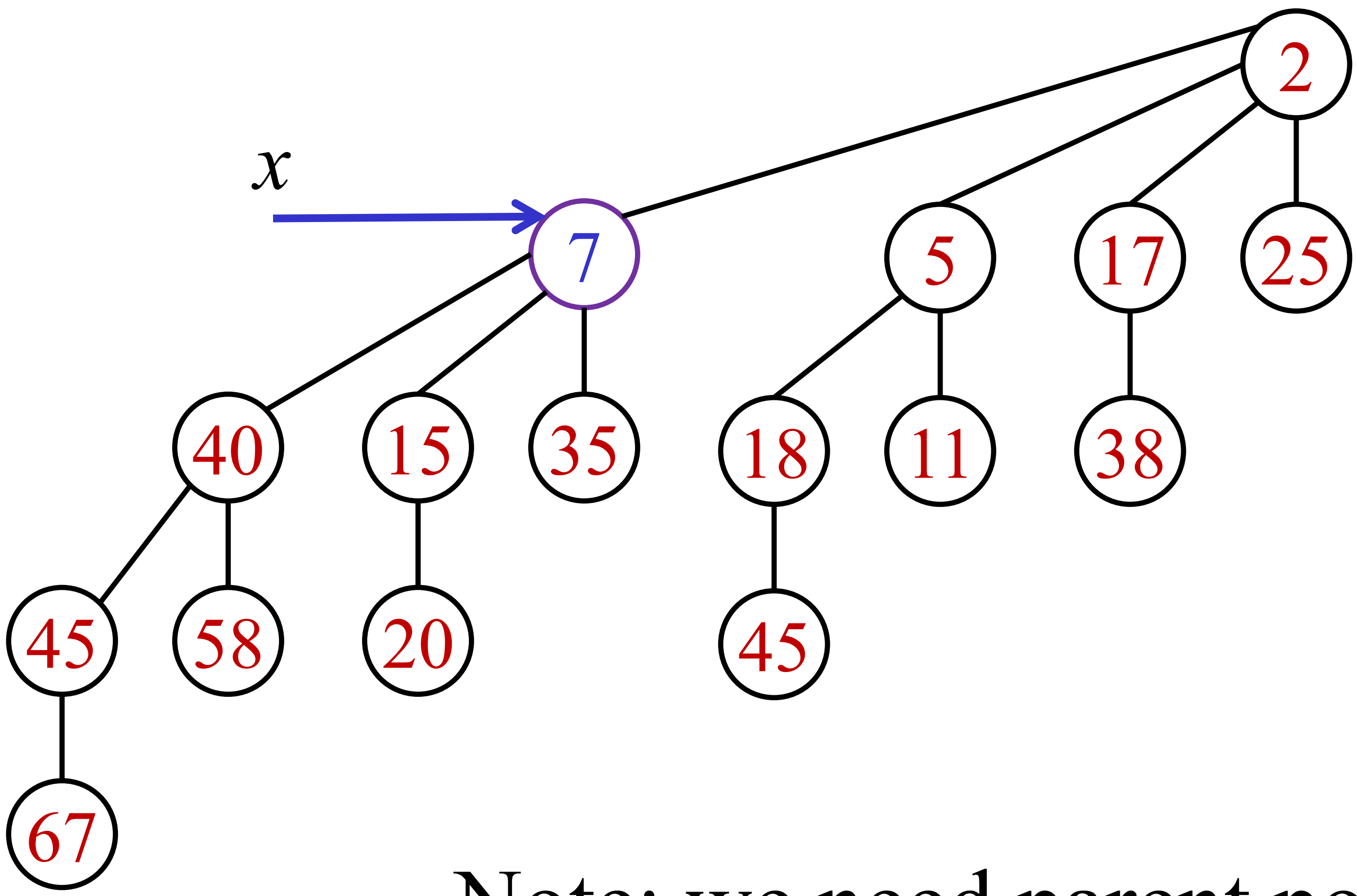
Decrease-key using "sift-up"



Decrease-key using "sift-up"



Decrease-key using “sift-up”



Note: we need parent pointers

Heaps / Priority queues

	Binary Heaps	**Binomial Heaps**	**Lazy Binomial Heaps**	**Fibonacci Heaps**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld	–	$O(\log n)$	$O(1)$	$O(1)$

Worst case

Amortized