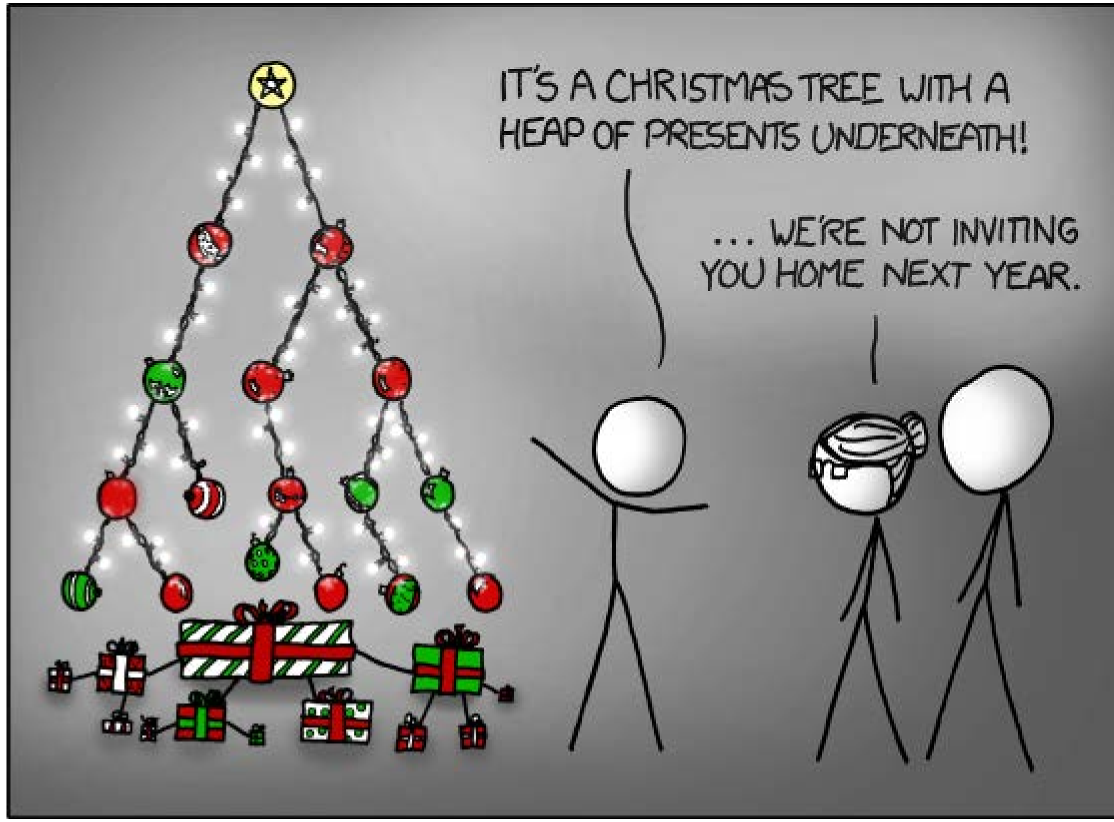


http://xkcd.com/835/

Lecture 14
Applications of priority queues

EECS-214

Priority queues

- Like normal queues
  - Objects wait in line to be processed
- However, items have an associated **numeric priority**
  - Priority specified when added to queue
  - Objects removed from queue in order of priority

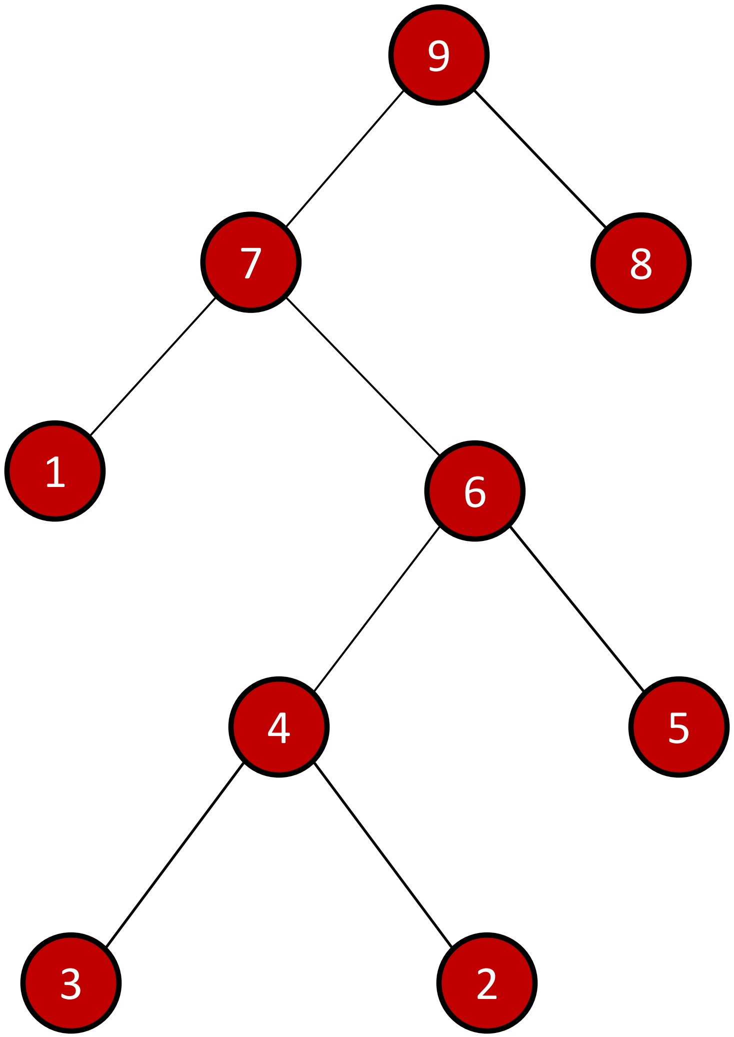
- Slightly different API
  - **Insert**(object, priority)
    - Adds object with specified priority
  - **ExtractMax**()
    - Returns highest priority object

Priority queues

- Like normal queues
  - Objects wait in line to be processed
- However, items have an associated numeric priority
  - Priority specified when added to queue
  - Objects removed from queue in order of priority

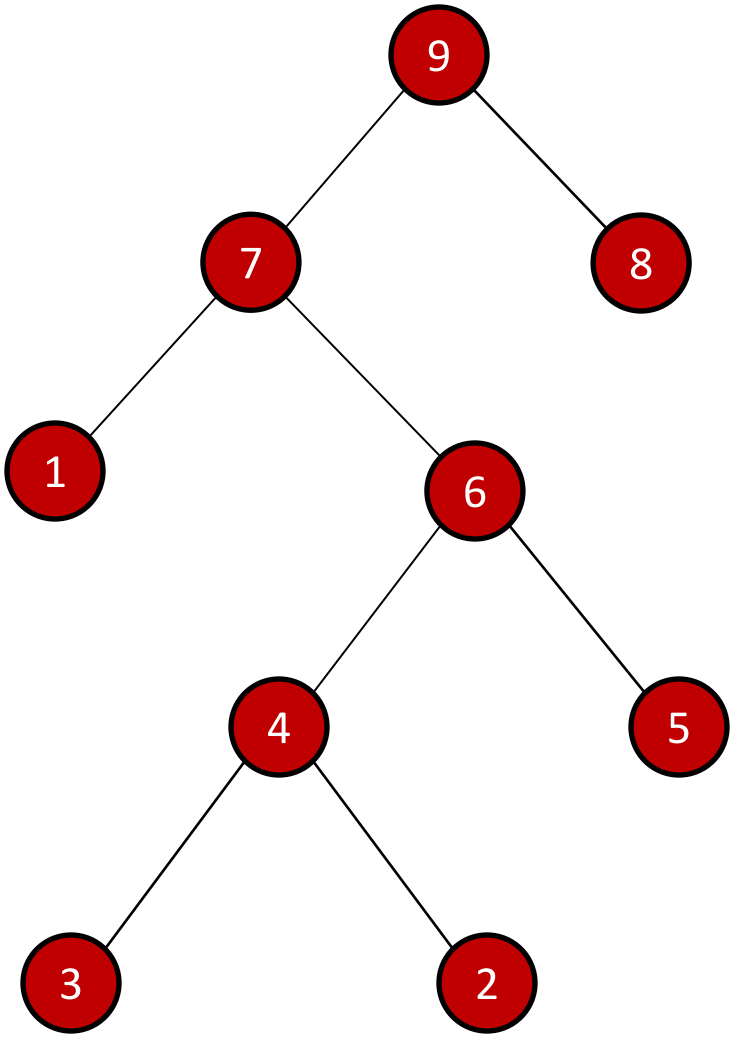
- Slightly different API
  - Insert(object, priority)
    - Adds object with specified priority
  - Extract**Min**()
    - "**Min priority queue**"
    - Returns **lowest** priority object
    - Except that in most applications, we usually want the lower "priorities" first

Heaps



- Heaps are a simple **tree structure** for implementing priority queues
- Rather than requiring their in-order traveral to be sorted
  - We just require that **parent nodes be larger than** their child nodes
- There are lots of exotic types of heaps
  - We'll focus on binary heaps
  - Which are **complete binary trees** with the heap property
  - We'll get to the completeness thing in a minute…

Heaps



Proposition: the largest element of a heap is always its root

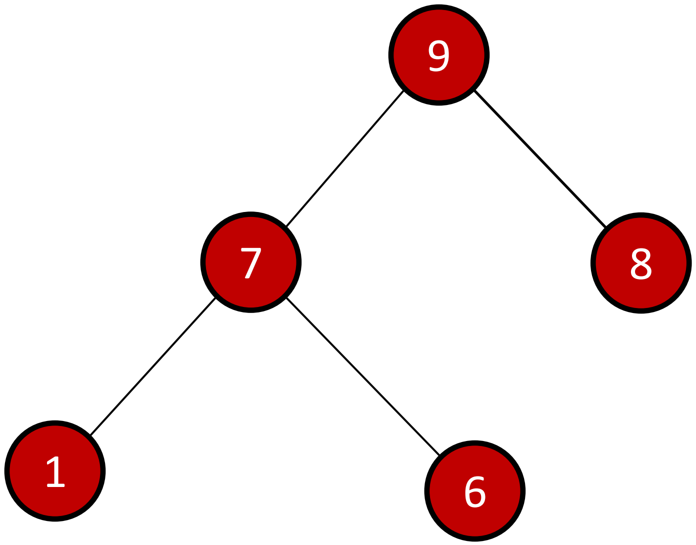
Proof:

- Suppose some other element is the largest element
- Since it isn't the root, it must have a parent
- Since it's the largest element, it must be larger than its parent
- But that contradicts the definition of a heap
- So the largest element must be the root

Binary heaps

- A **binary heap** is a
  - **Complete** binary tree
  - That satisfies the heap property

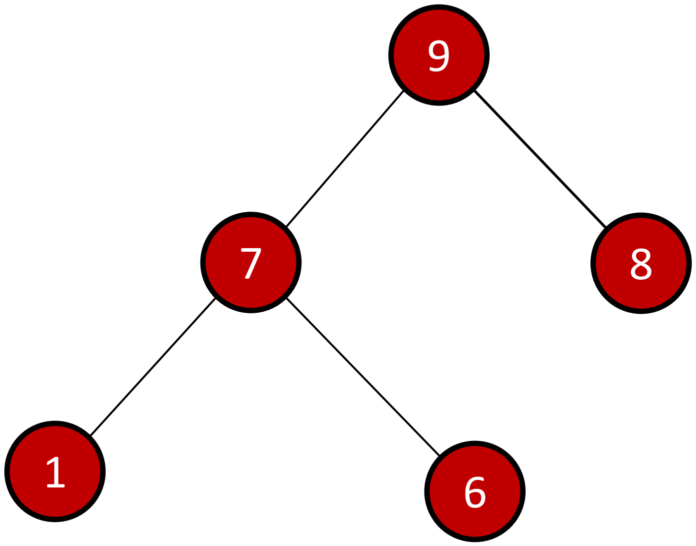
- Great!
- How do we ensure that the heap is a complete binary tree?



Embedding in an array

- It turns out that any complete binary tree can be **embedded an array** in a particularly cleaver way

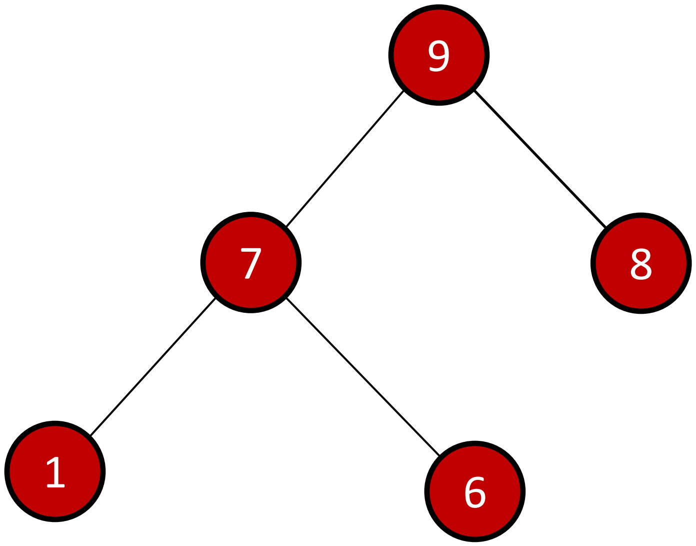
- We can compute
  - The position of its parent in the array,
  - and the positions of its children,
  - directly from its own position

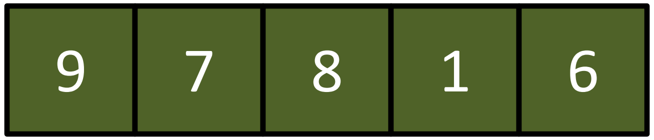


Embedding in an array

- Store the **root** in the first element (element 0)
- For any node
  - Let i be its position in the array (for the root, $i = 0$)
  - Store its **left child** at position $2i + 1$
  - Store its **right child** at position $2i + 2$
  - Its **parent** can be found at position $\lfloor (i - 1)/2 \rfloor$

- Trust me that this works :-)





Heap insertion using the array representation

HeapInsert(A, value)

A.size = A.size + 1

i = A.size

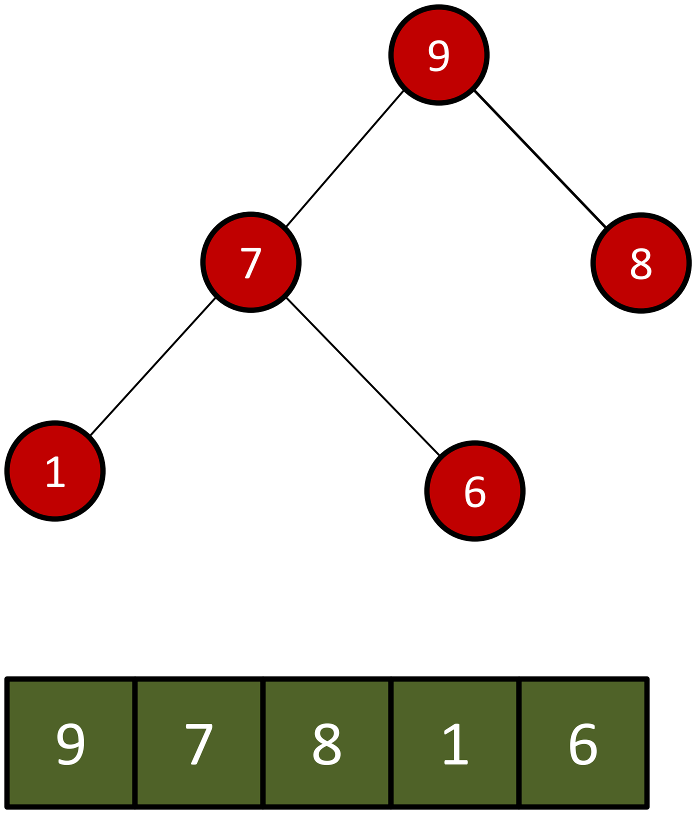
while i>0 and

A[Parent(i)] < key

A[i] = A[Parent(i)]

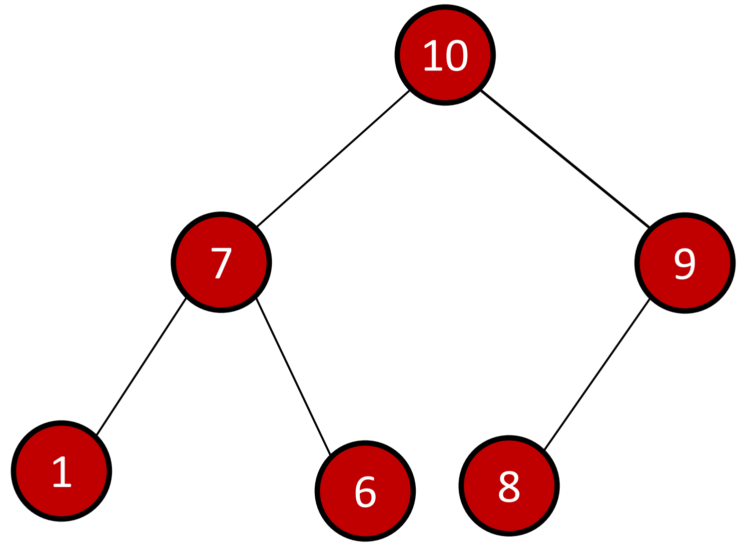
i = Parent(i)

A[i] = key



Extracting an element

```
HeapExtractMax(A, value)
   max = A[0]
   A[0] = A[A.size]
   A.size--
   Heapify(A,0)
   return max
```



10	7	9	1	6	8

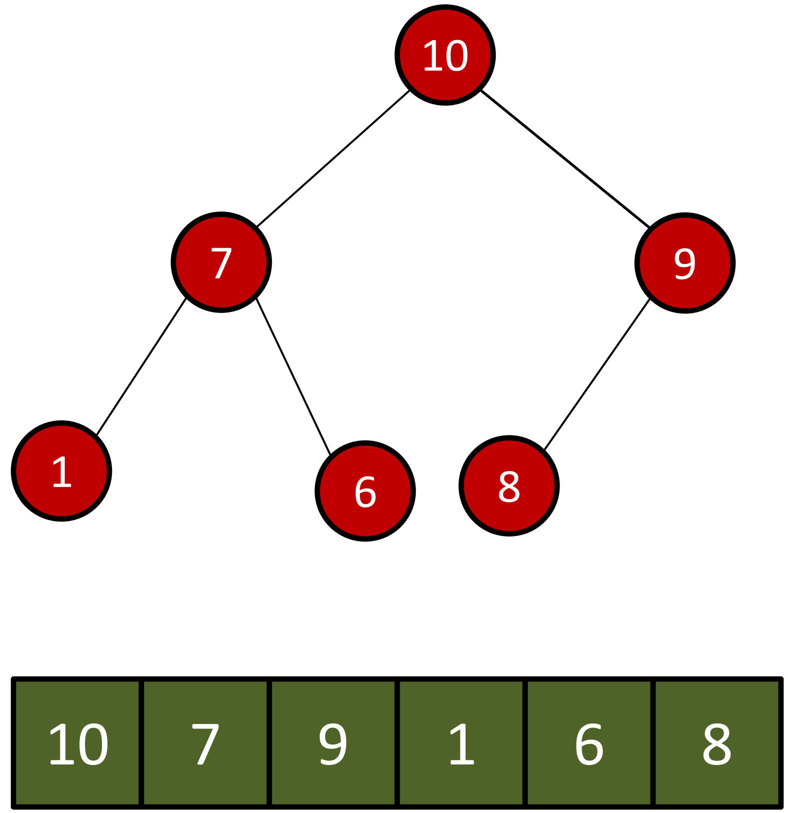
Extracting an element

```
Heapify(A, i)
  l = Left(i)
  r = Right(i)

  if l≤A.size and A[l]>A[i]
    largest = l
  else
    largest = i

  if r≤A.size and A[r]>A[largest]
    largest = r

  if largest≠i
    swap A[i] and A[largest]
    Heapify(A, largest)
```



Sorting

[Pretend I had found some awesome clipart about sorting here]

- Priority queues can be used for **sort algorithms**
  - **Add all items** to the queue
  - Repeatedly **extract max**
    - Or min, if it's a min queue
  - **Write them in order**

Version 1

- Here's the **straightforward** way of sorting using a heap
  - We **make the heap**
  - **Insert** all the **elements** into it
  - Pull them out one at a time, and **write them back** into A

- However, we're **copying** all the data
  - From one array, A
  - To the heap, H, which is also an array
  - $O(n)$ **space**

```
Heapsort(A)
  H = new, empty heap
  for each e in A
    Insert(H, e)
  for i = A.Length-1 to 0
     A[i] = ExtractMax(H)
```

In-place heapsort

- It would be cooler if we could **build the heap inside A itself**
- How do we do that?

```
HeapSort(A)
  H = new, empty heap
  for each e in A
    Insert(H, e)
  for i = A.Length-1 to 0
    A[i] = ExtractMax(H)
```

Sketch of in-place heap construction

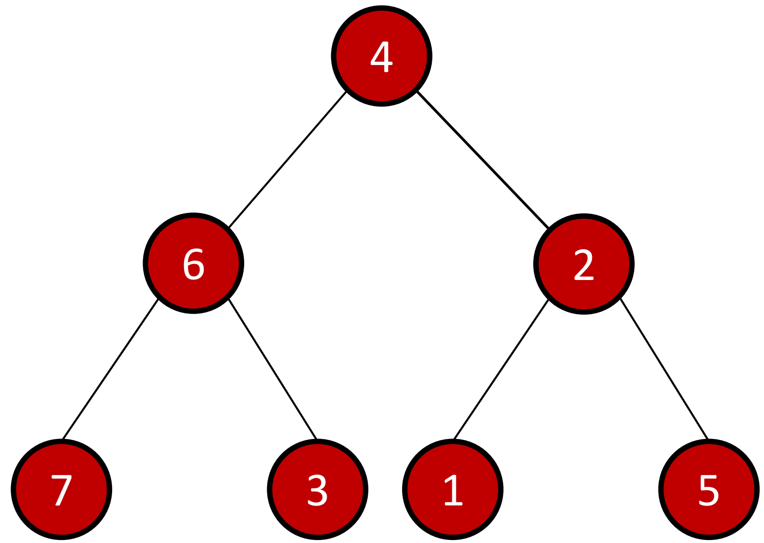
Sketch of in-place heap construction

- Start with the array

4	6	2	7	3	1	5

Sketch of in-place heap construction

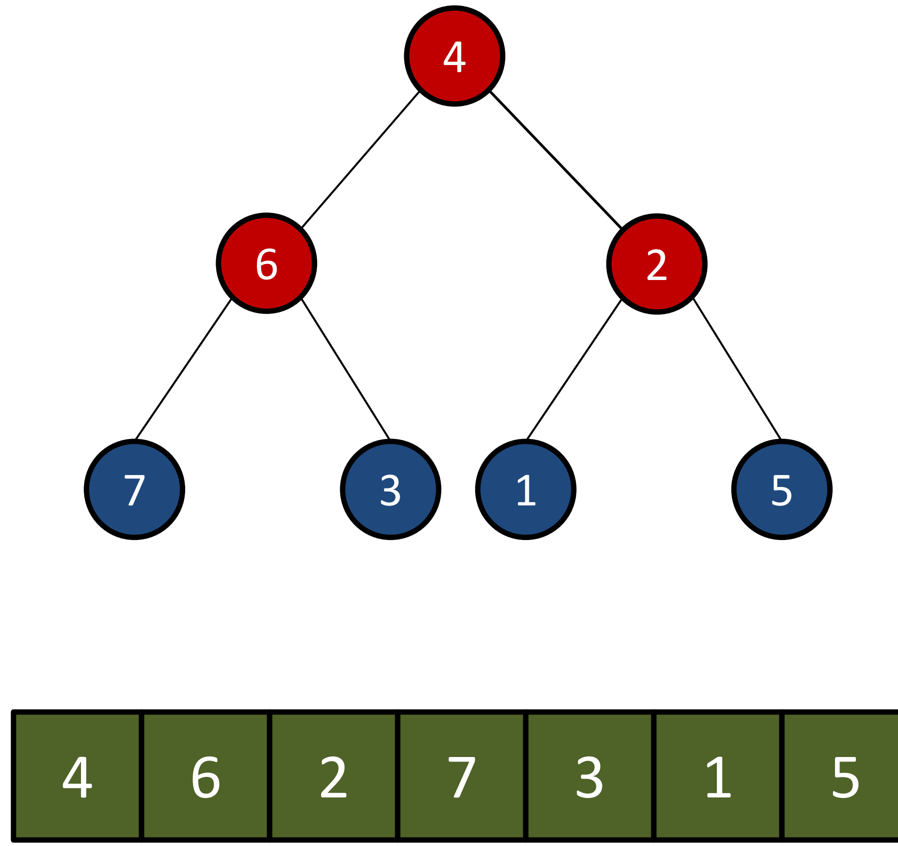
- Start with the array
- Pretend that it's a binary tree
- It probably **doesn't satisfy** the heap property
  - i.e. there are probably nodes that are larger than their parents



4	6	2	7	3	1	5

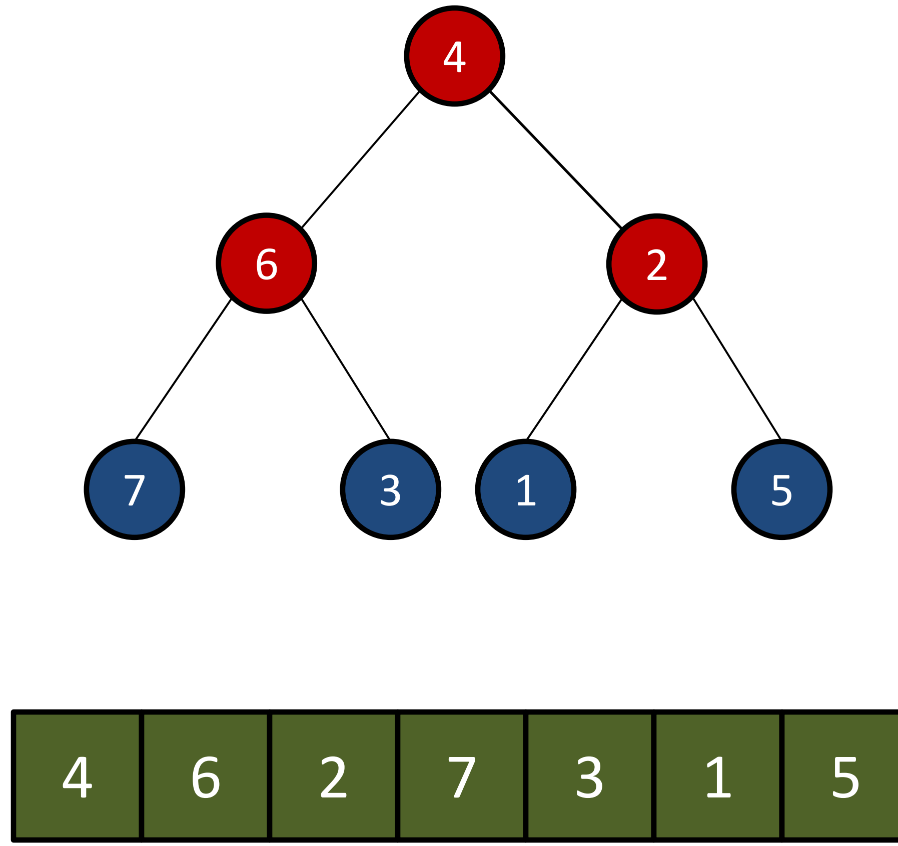
Sketch of in-place heap construction

- But we can think of **each leaf** as a little **1-element heap**
  - And they **automatically satisfy** the heap property
  - Because the only have one element



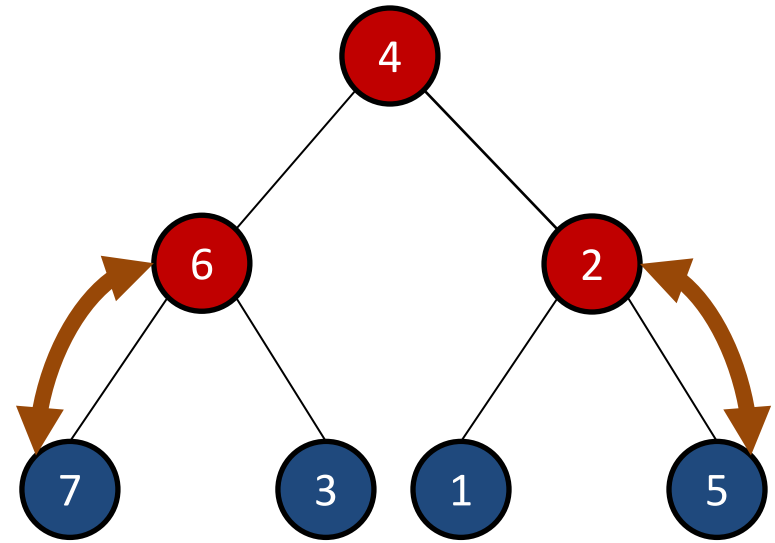
Sketch of in-place heap construction

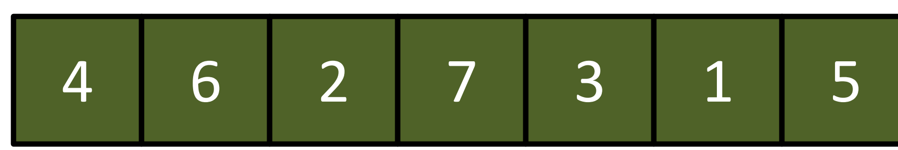
- Now run **heapify** on each of their parents
- Heapify
  - **Checks** if the **parent** is larger that both children



Sketch of in-place heap construction

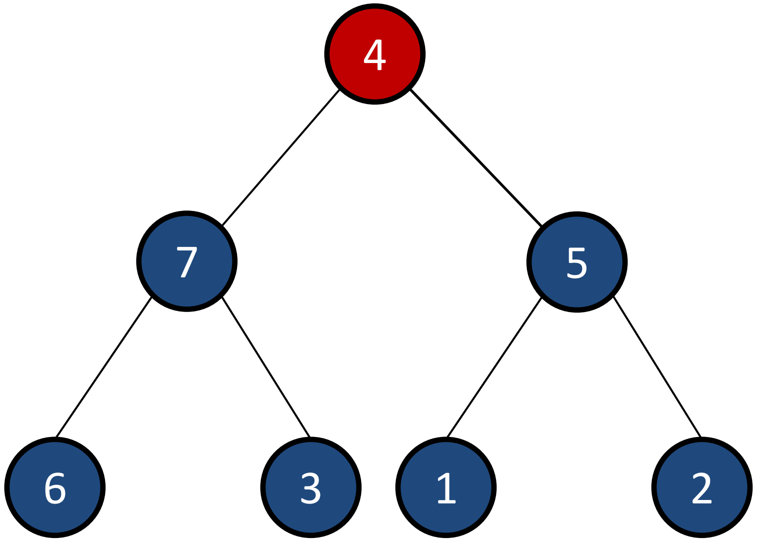
- Now run **heapify** on each of their parents
- Heapify
  - Checks if the **parent** is larger that both children
  - If not, it **swaps** it with the **larger child**





Sketch of in-place heap construction

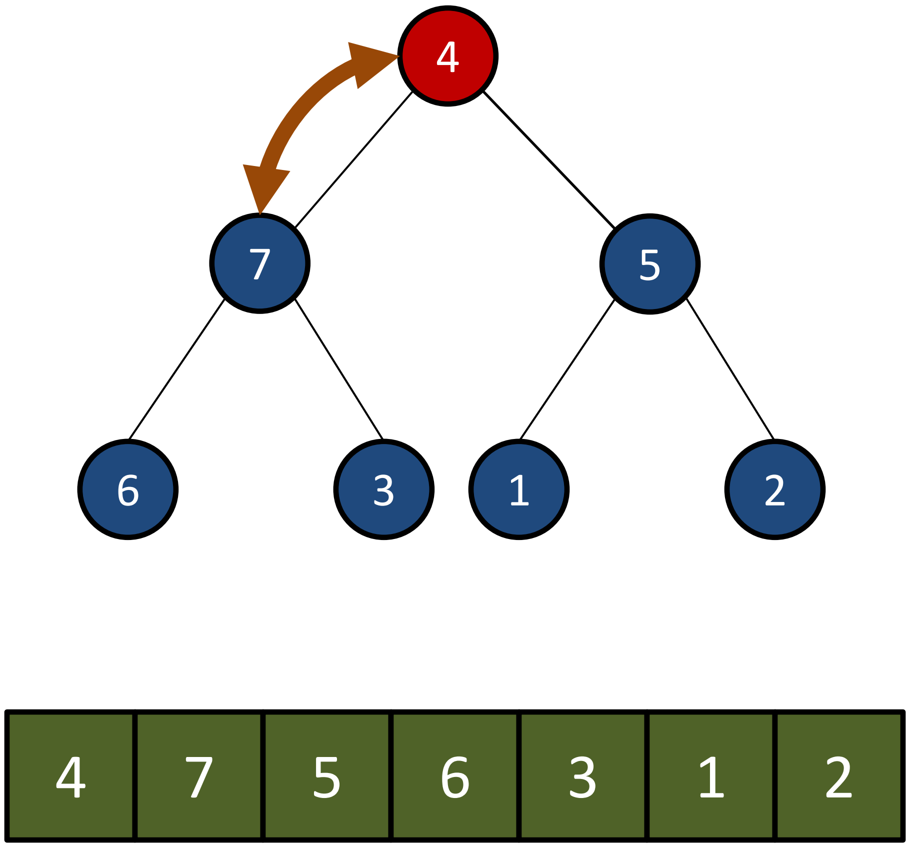
- Now we have
  - A bunch of **2-level subtrees** that
  - Each **satisfy the heap property**



4	7	5	6	3	1	2

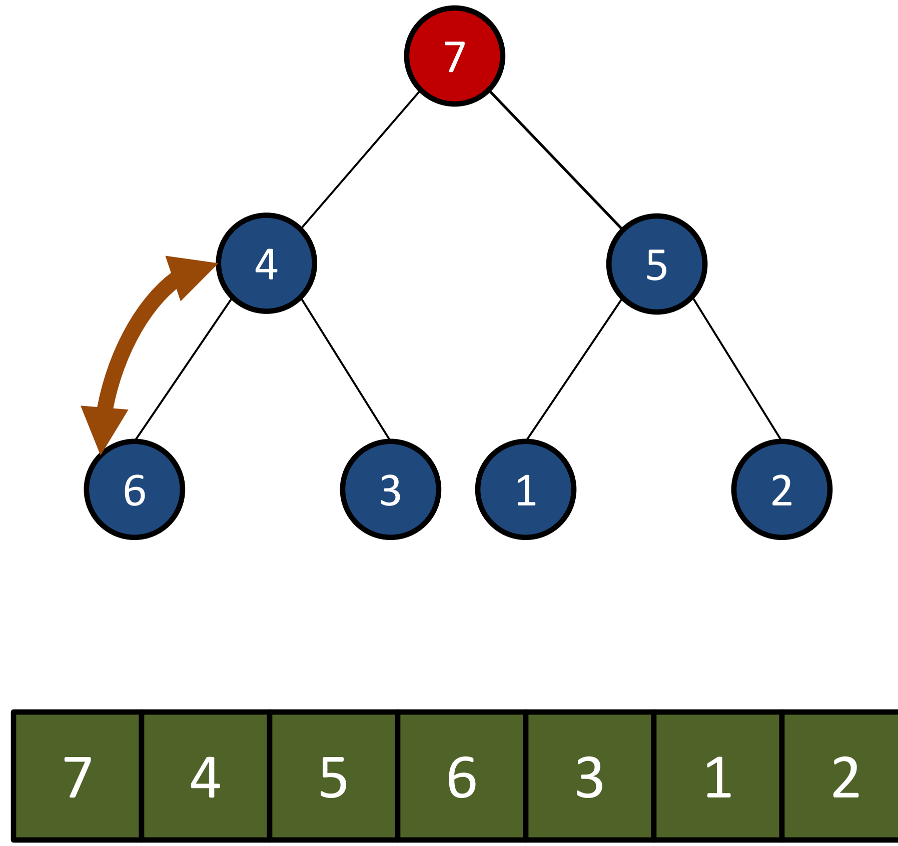
Sketch of in-place heap construction

- **Repeat** at the next level



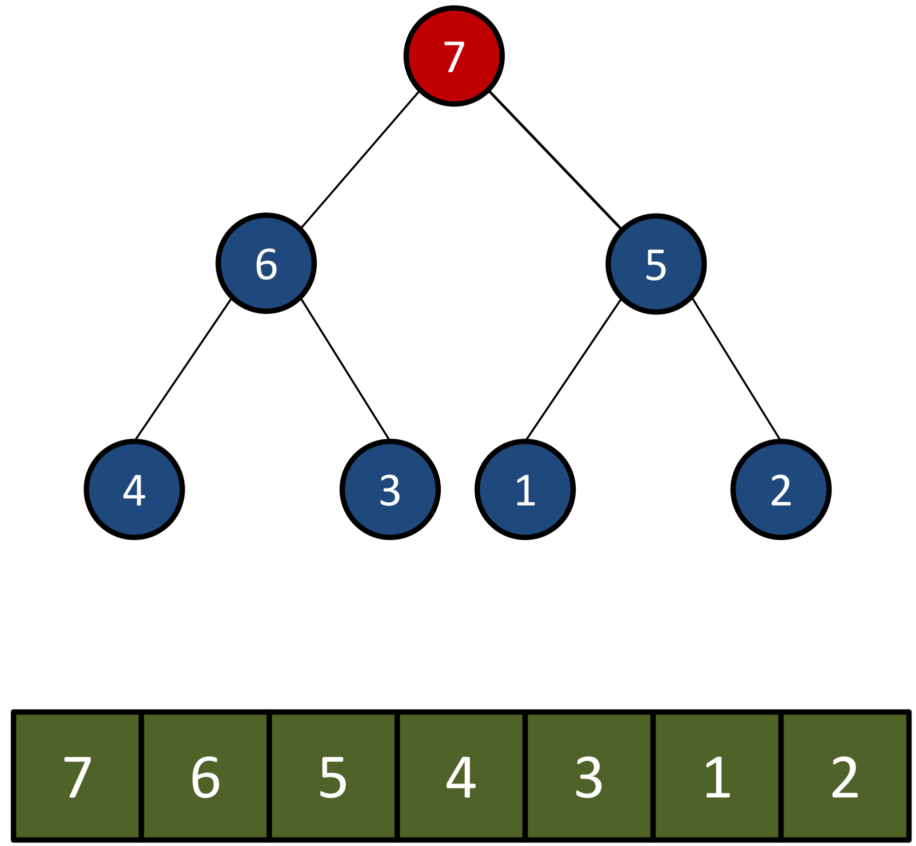
Sketch of in-place heap construction

- **Repeat** at the next level
- This may require Heapify to **recurse**



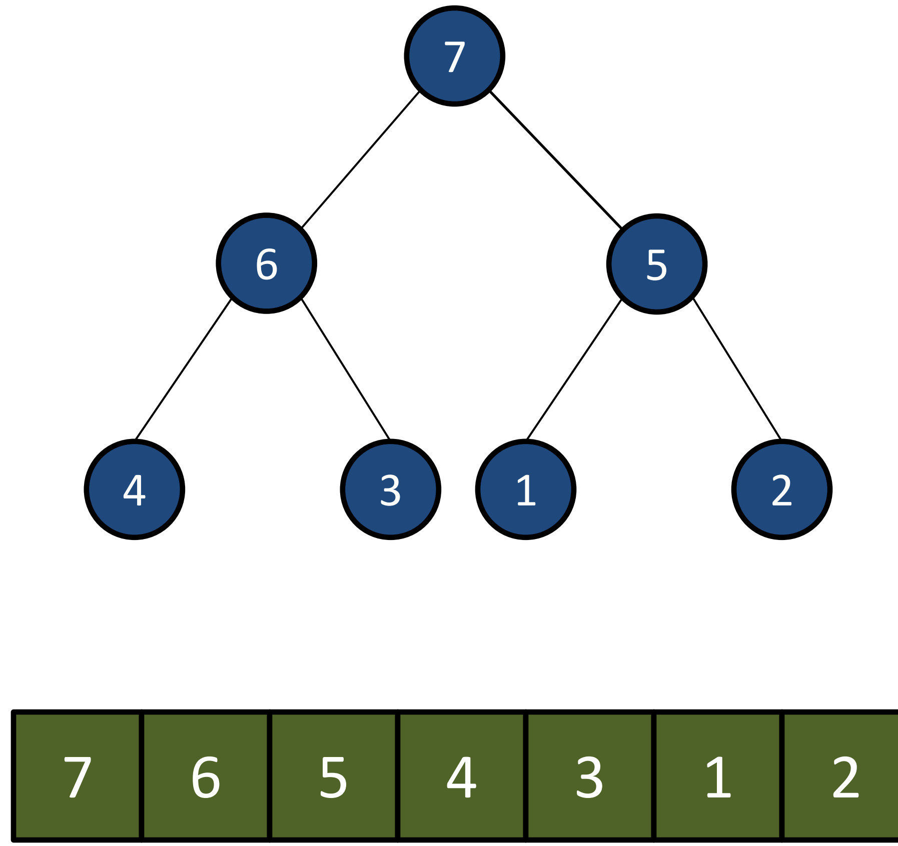
Sketch of in-place heap construction

- Repeat at the next level
- This may require Heapify to recurse
- But when we've **done every level**



Sketch of in-place heap construction

- Repeat at the next level
- This may require Heapify to recurse
- But when we've **done every level**
- We've transformed the data, **in-place**, into a **binary heap**



Algorithm for in-place heap construction

- The **last half** of the array is guaranteed to be **leaves**
  - So we don't have to do anything with it

BuildHeap(A)

  for i = A.Length/2 to 1

    Heapify(A, i, A.Length)

Algorithm for in-place heap construction

- We want to call **Heapify**
  - On every **non-leaf** node
  - **Starting at the bottom** of the tree
  - And moving upwards

- Since **lower parts** of the tree are **at the end** of the array,
  - All we have to do is **start halfway** through the array
  - and **move back**
  - Calling Heapify

```
BuildHeap(A)
  for i = A.Length/2 to 0
    Heapify(A, i, A.Length)
```

Version 2

Now this is really kind of **cool**

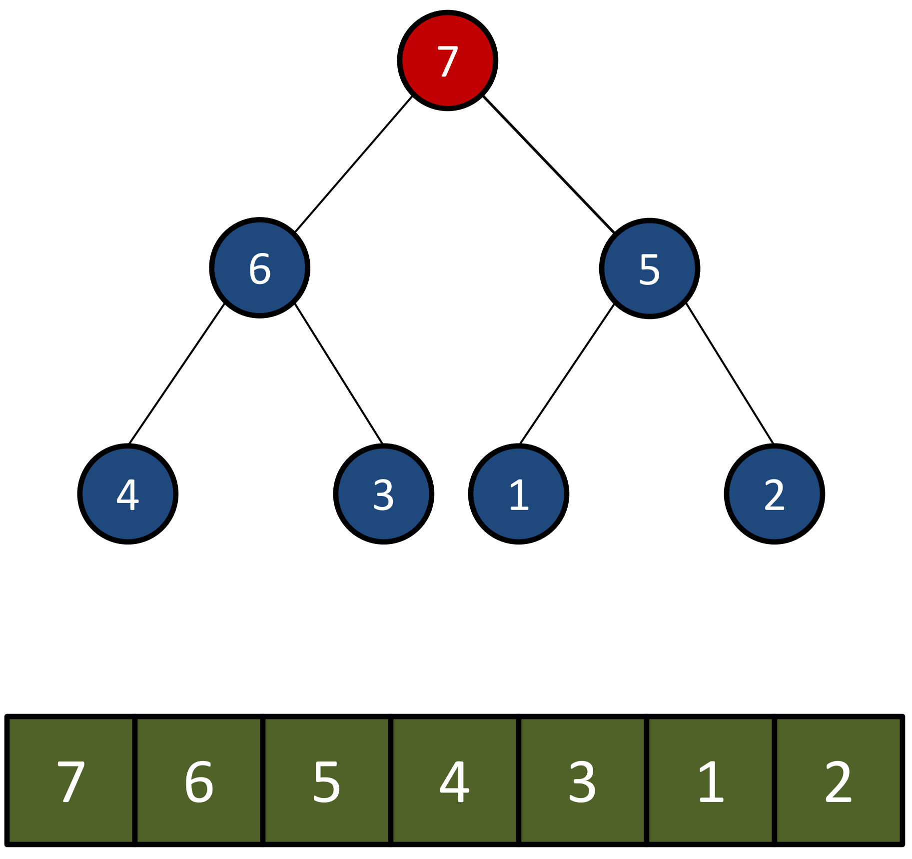
- We **build** the heap **in place**
- Then we **loop**
  - Each iteration **removes** the **maximal** element
  - That **shrinks** the heap
  - **Making room** at the end of the array
  - Which is **where** we would want to **put** the maximal element, anyway!

```
Heapsort(A)
  BuildHeap(A)
  for i = A.Length-1 to 1
     A[i] = ExtractMax(H)
```

Completing the sort

ExtractMax

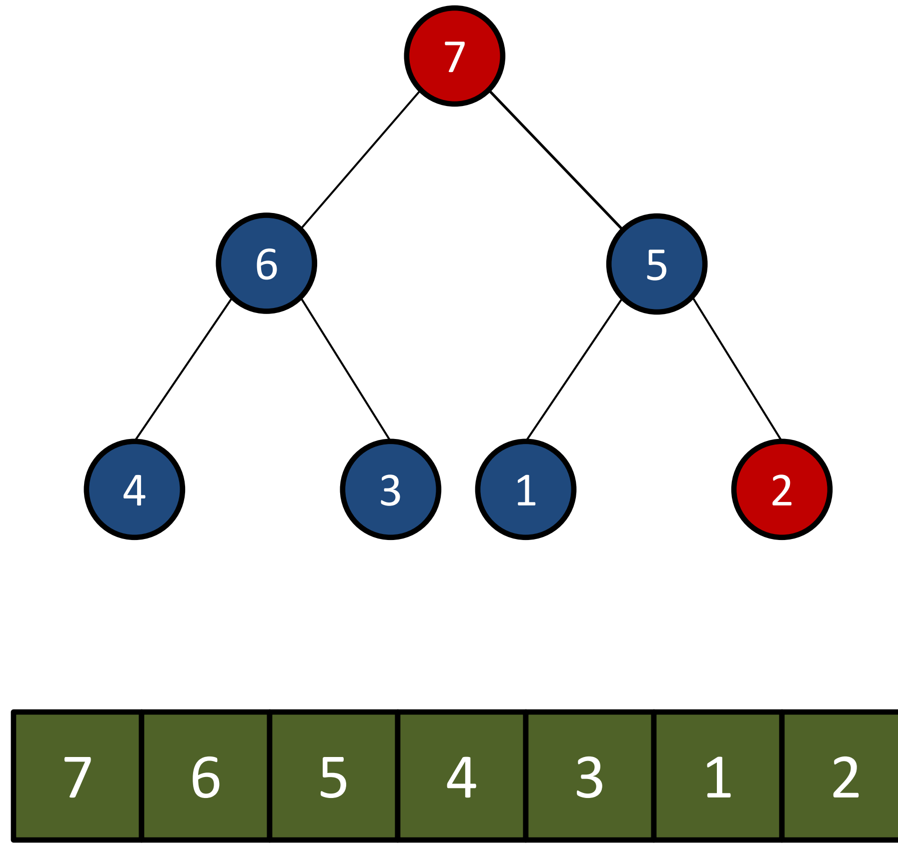
- Grabs the maximal element, 7



Completing the sort

ExtractMax

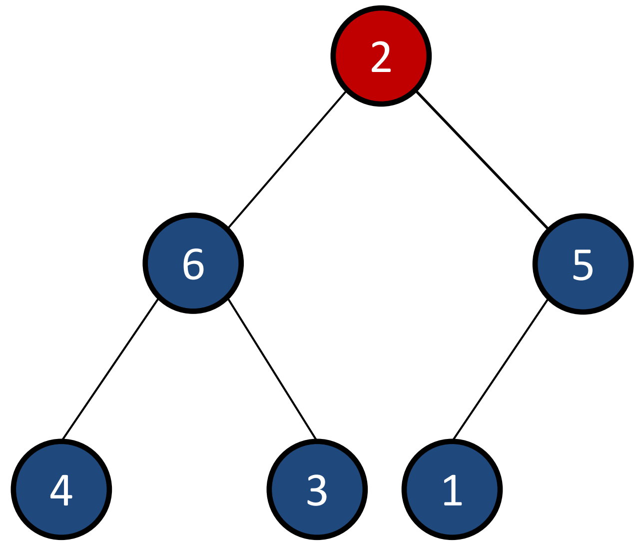
- Grabs the maximal element, 7
- Replaces it with the last leaf, 2



Completing the sort

ExtractMax

- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify

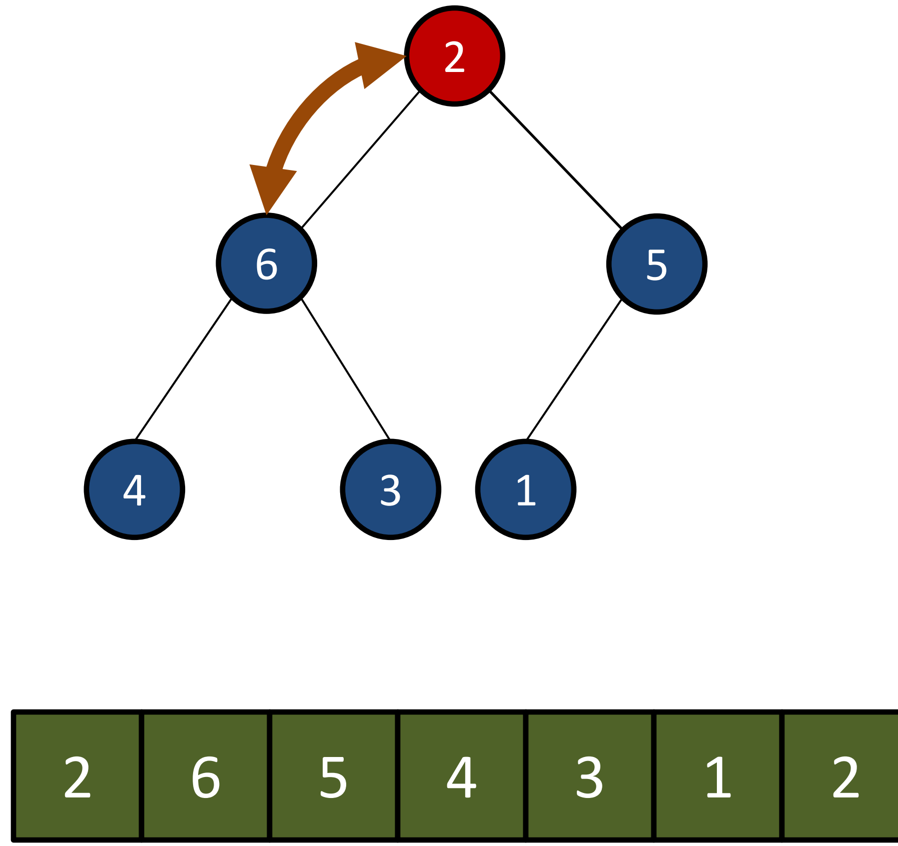


2	6	5	4	3	1	2

Completing the sort

ExtractMax

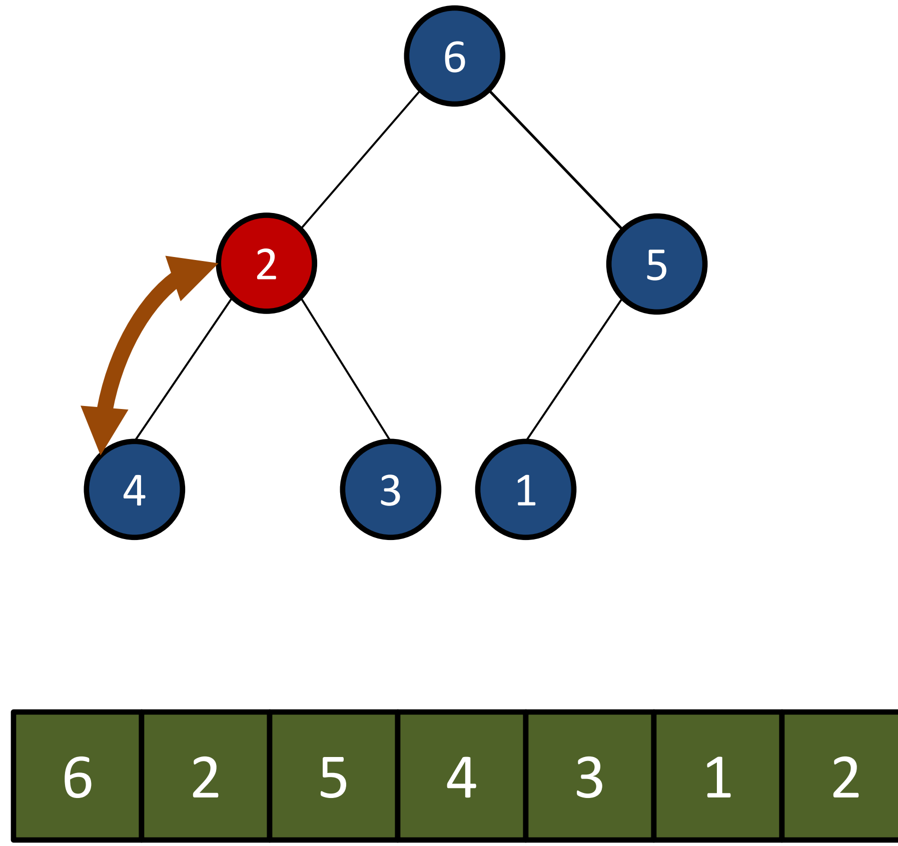
- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify



Completing the sort

ExtractMax

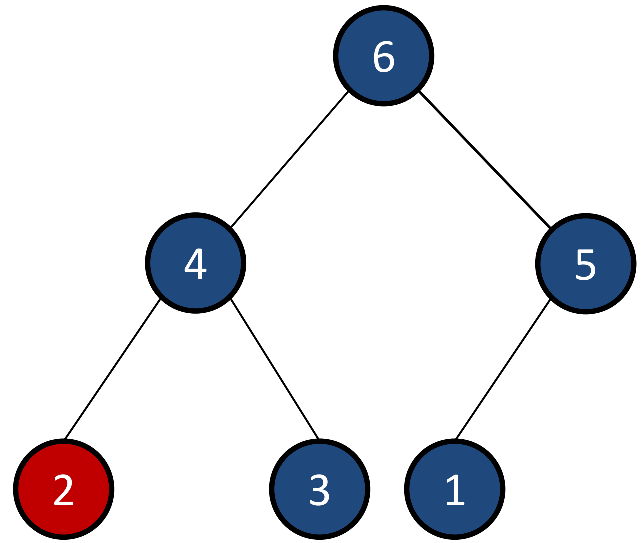
- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify



Completing the sort

ExtractMax

- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify

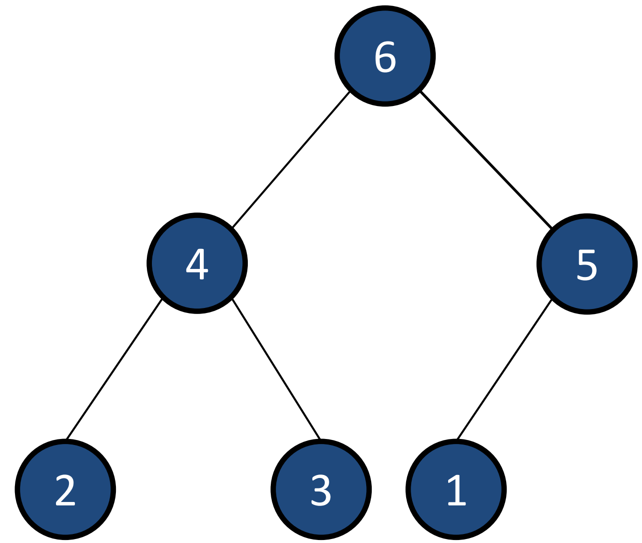


6	4	5	2	3	1	2

Completing the sort

ExtractMax

- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify
  - Thereby reestablishing the heap property

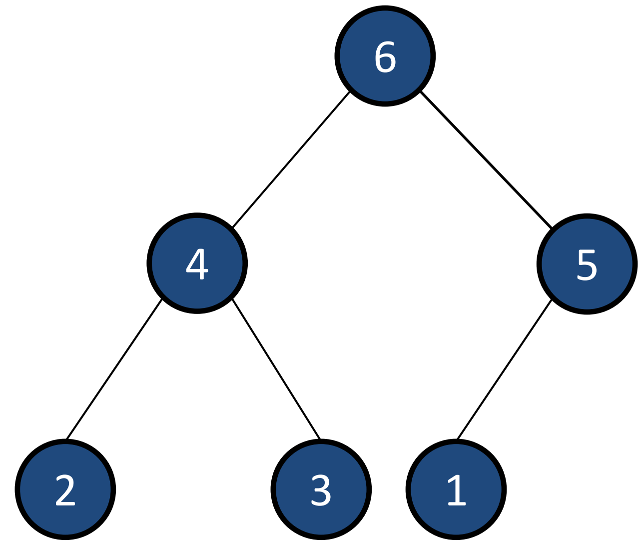


6	4	5	2	3	1	2

Completing the sort

ExtractMax

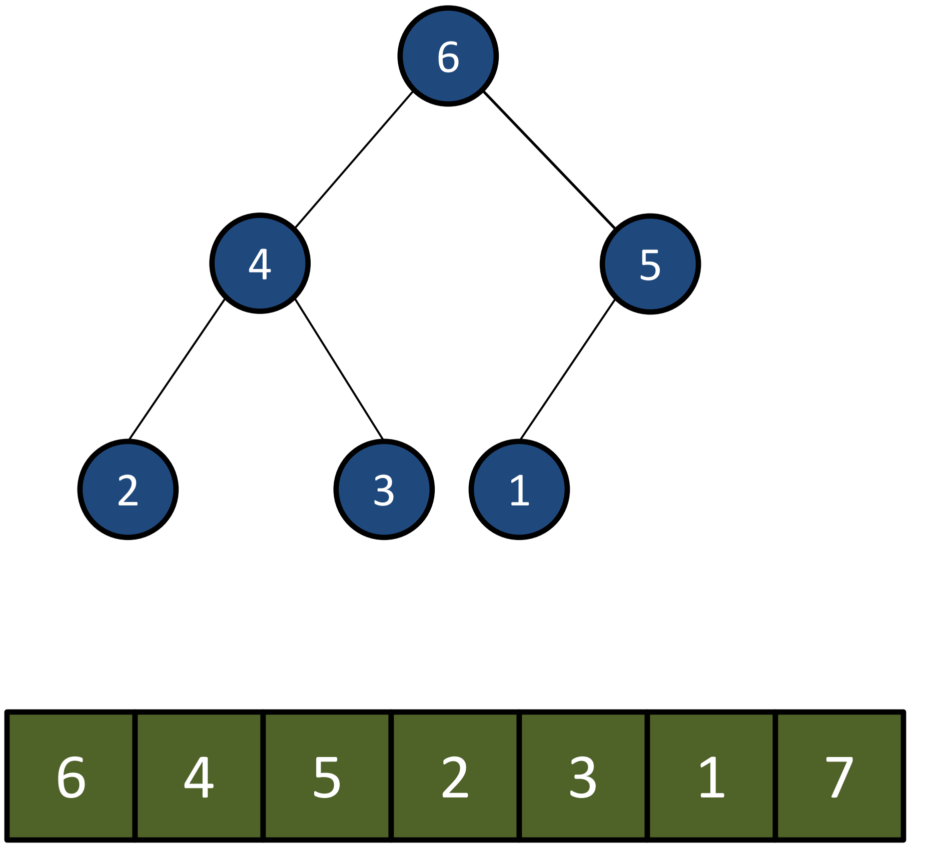
- Grabs the maximal element, 7
- Replaces it with the last leaf, 2
- And calls Heapify
  - Thereby reestablishing the heap property
- And now we store 7 (the max) at the end of the array)



6	4	5	2	3	1	7

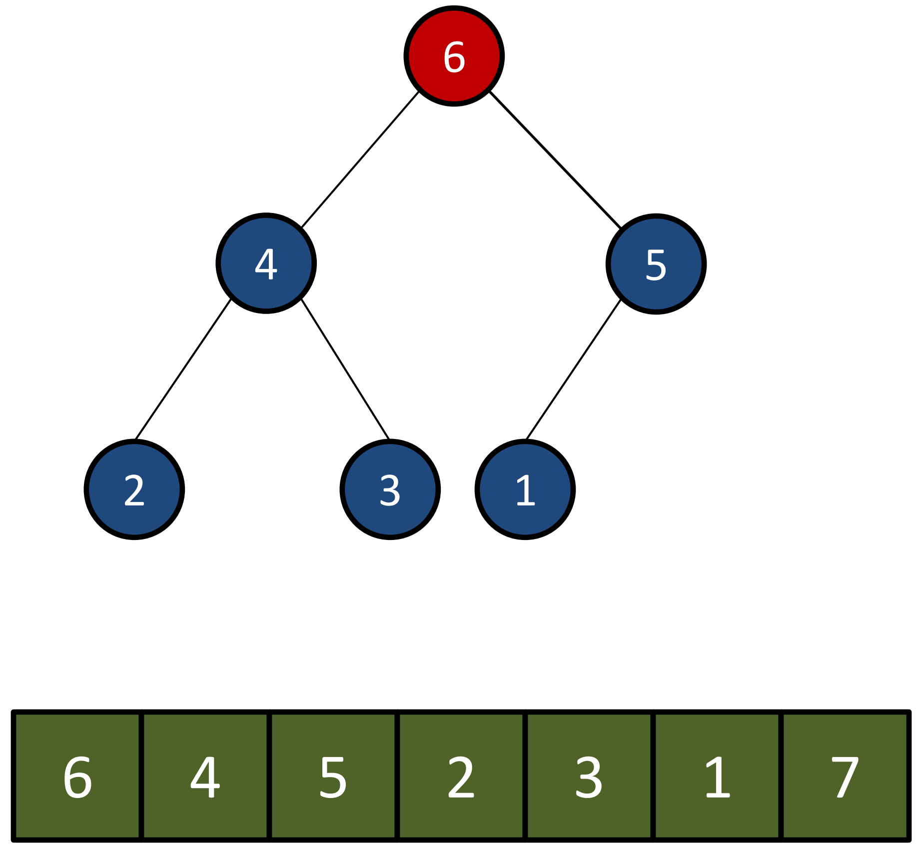
Completing the sort

Repeat



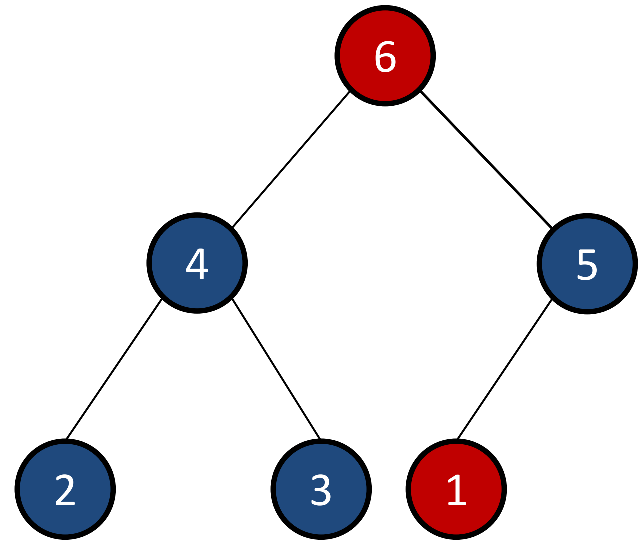
Completing the sort

- Grab the max, 6



Completing the sort

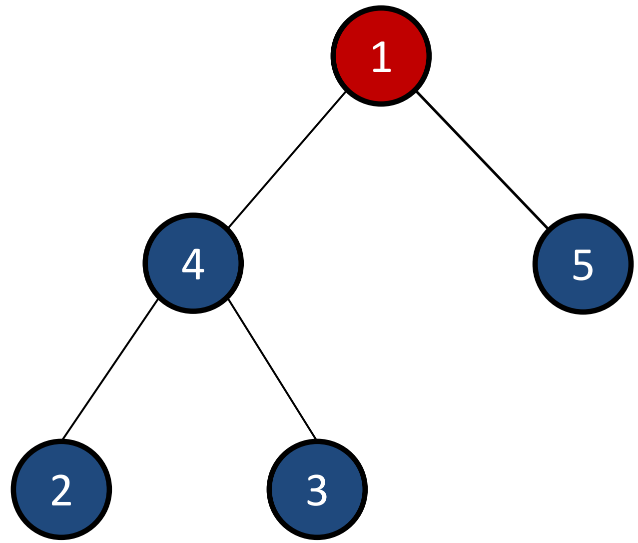
- Grab the max, 6
- Replace it with the last leaf, 1

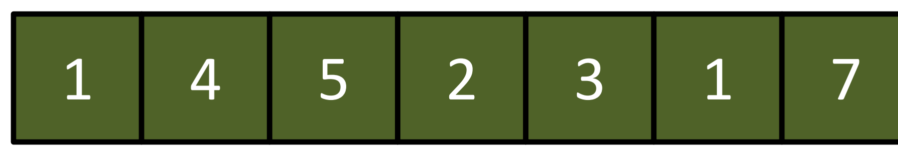


6	4	5	2	3	1	7

Completing the sort

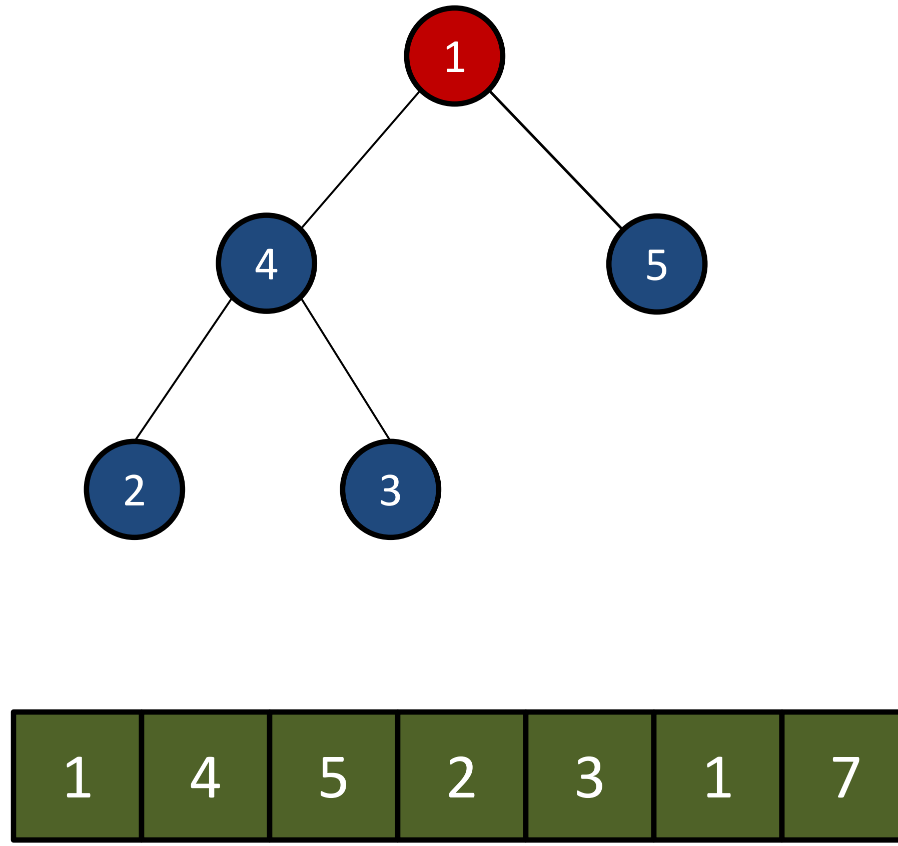
- Grab the max, 6
- Replace it with the last leaf, 1





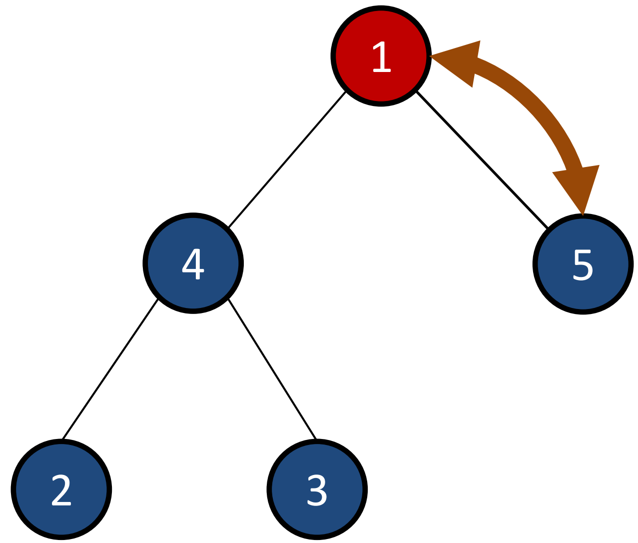
Completing the sort

- Grab the max, 6
- Replace it with the last leaf, 1
- Heapify



Completing the sort

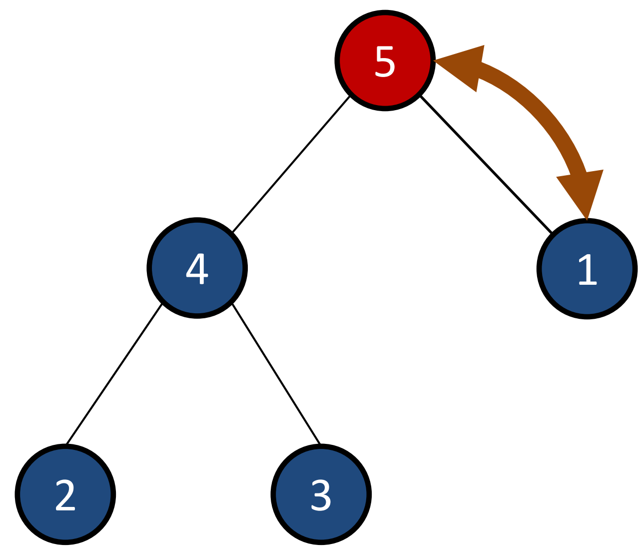
- Grab the max, 6
- Replace it with the last leaf, 1
- Heapify



1	4	5	2	3	1	7

Completing the sort

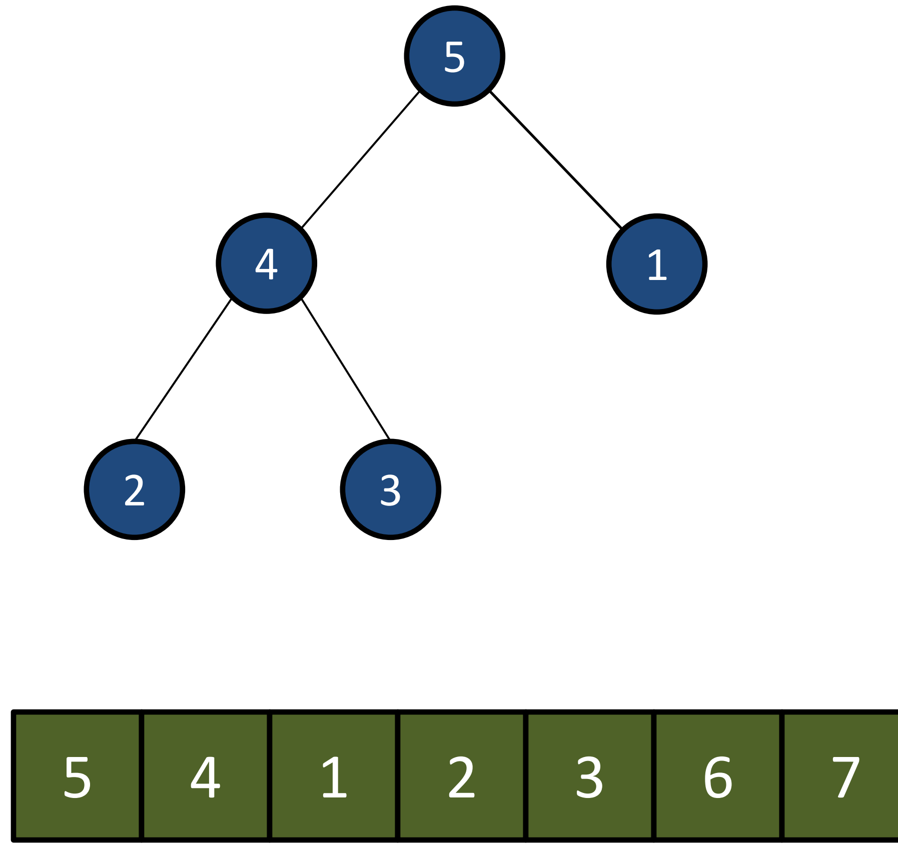
- Grab the max, 6
- Replace it with the last leaf, 1
- Heapify



5	4	1	2	3	1	7

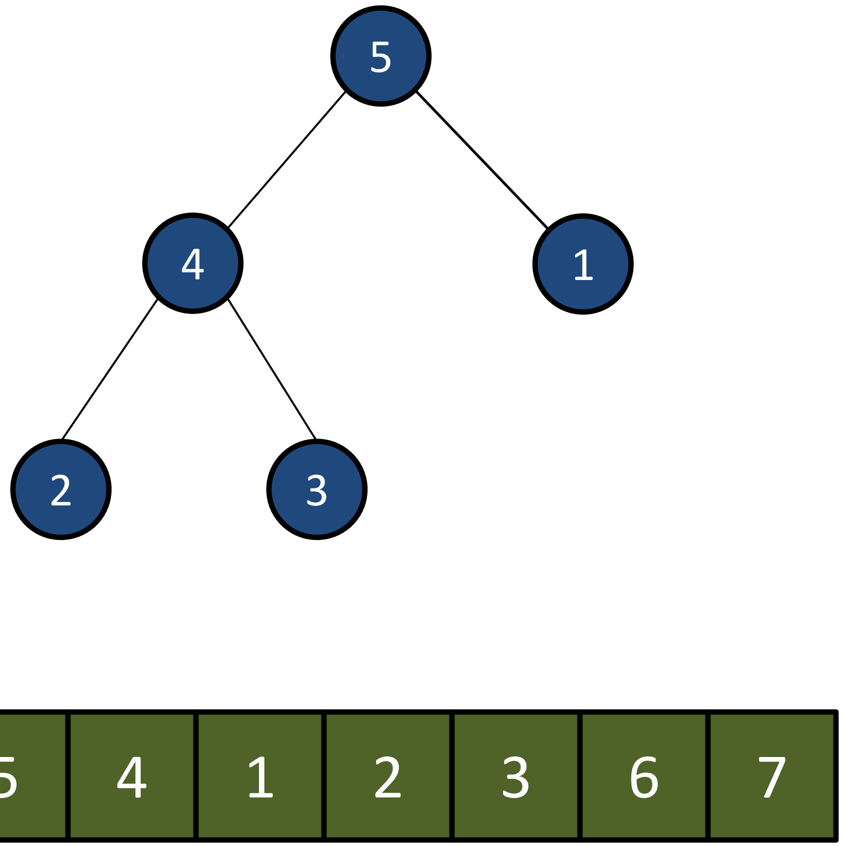
Completing the sort

- Grab the max, 6
- Replace it with the last leaf, 1
- Heapify
- Store the max, 6, in the next-to-last slot



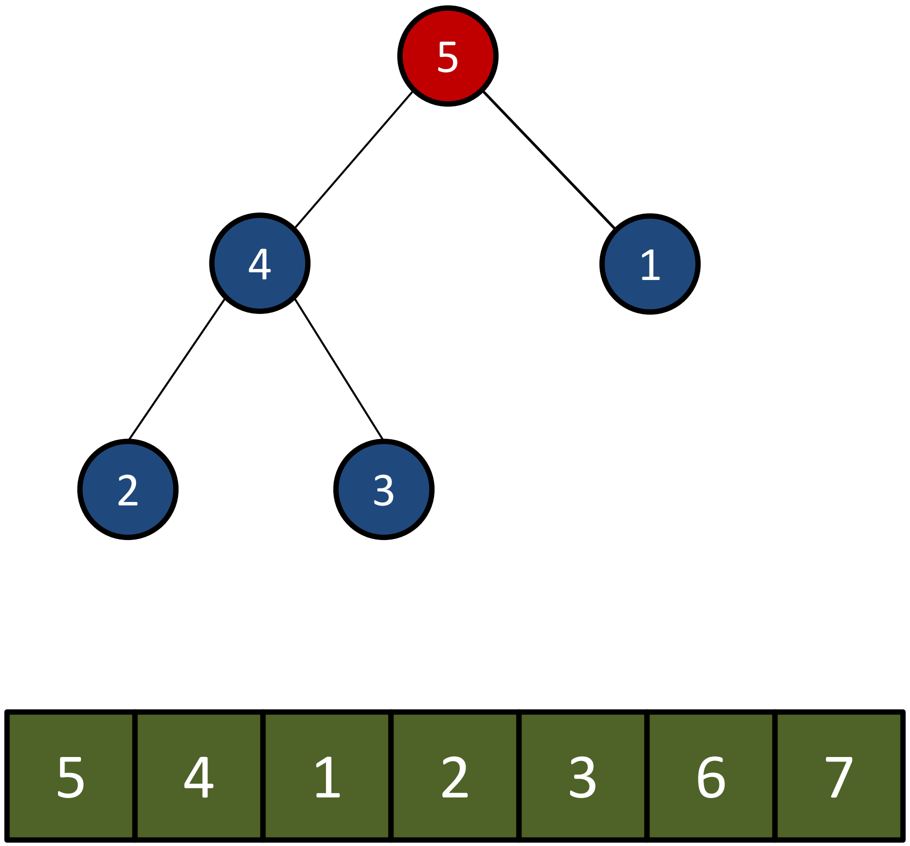
Completing the sort

Repeat



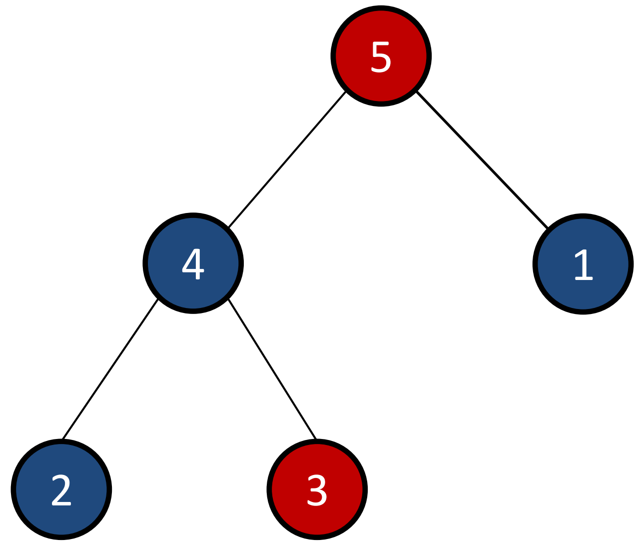
Completing the sort

- Grab the max, 5



Completing the sort

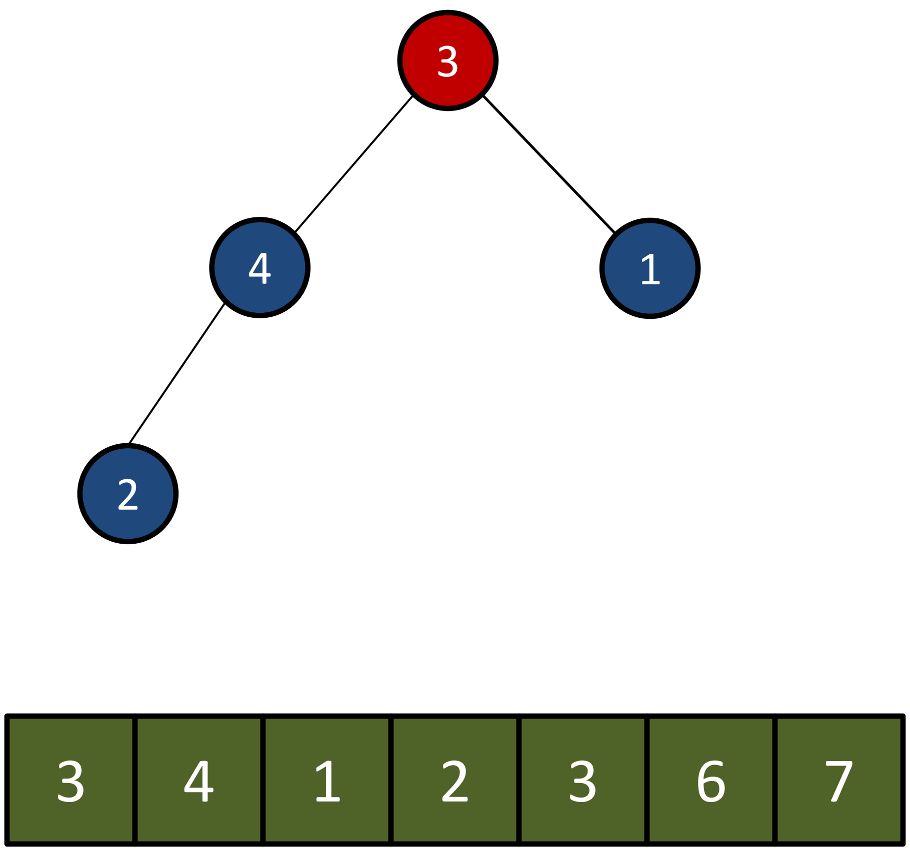
- Grab the max, 5
- Replace with the last leaf, 3



5	4	1	2	3	6	7

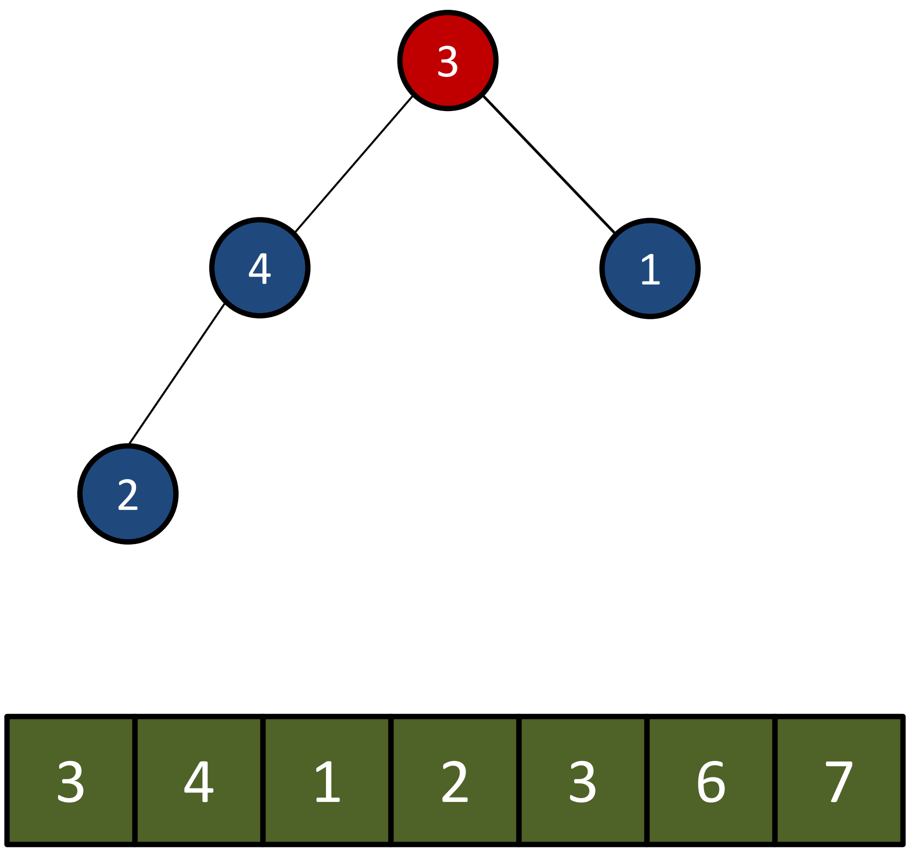
Completing the sort

- Grab the max, 5
- Replace with the last leaf, 3



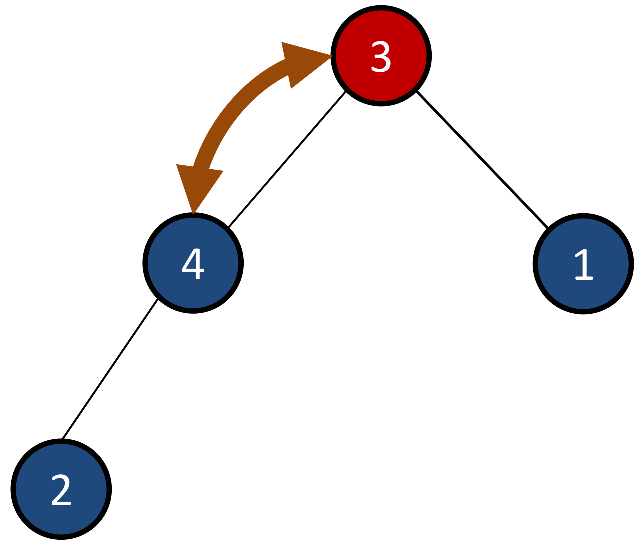
Completing the sort

- Grab the max, 5
- Replace with the last leaf, 3
- Heapify



Completing the sort

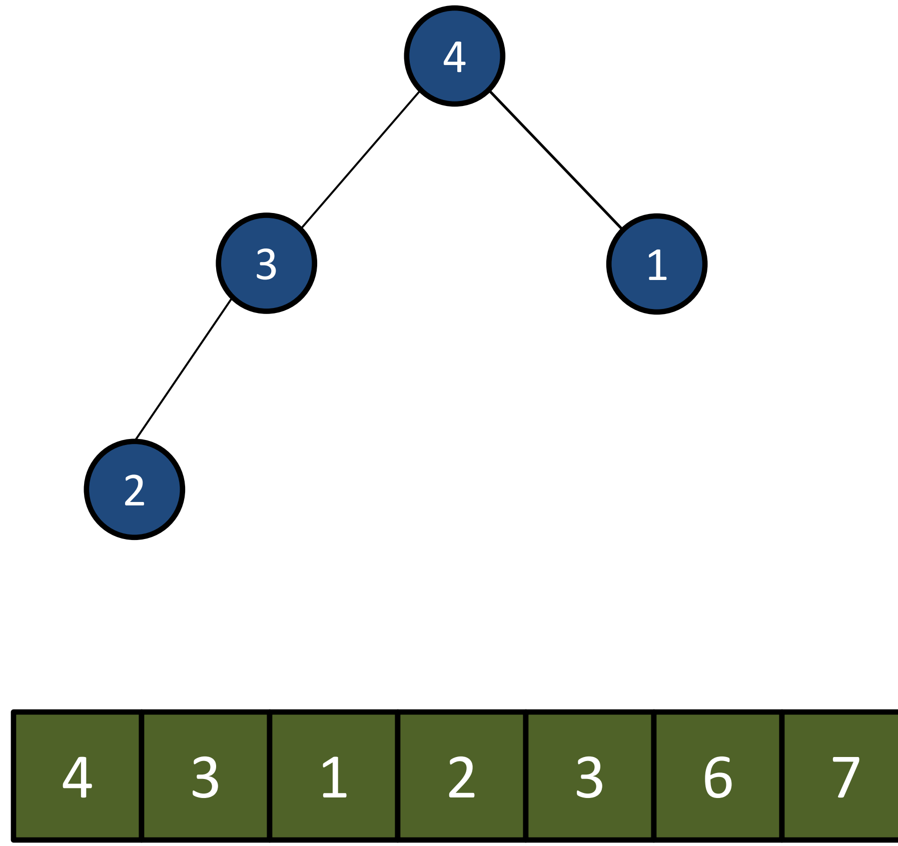
- Grab the max, 5
- Replace with the last leaf, 3
- Heapify



3	4	1	2	3	6	7

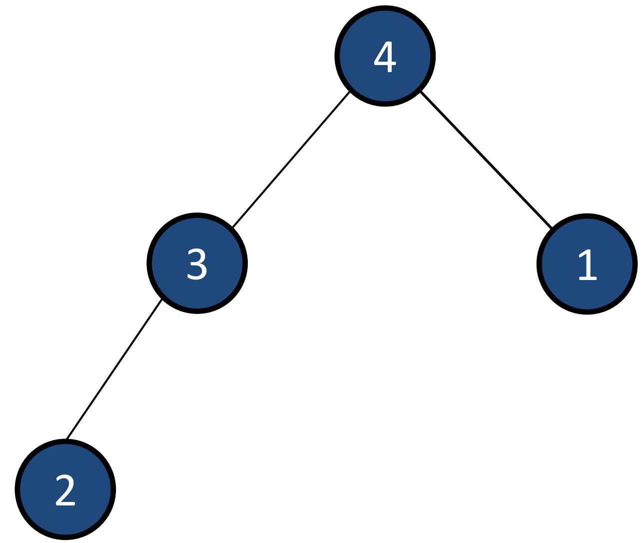
Completing the sort

- Grab the max, 5
- Replace with the last leaf, 3
- Heapify



Completing the sort

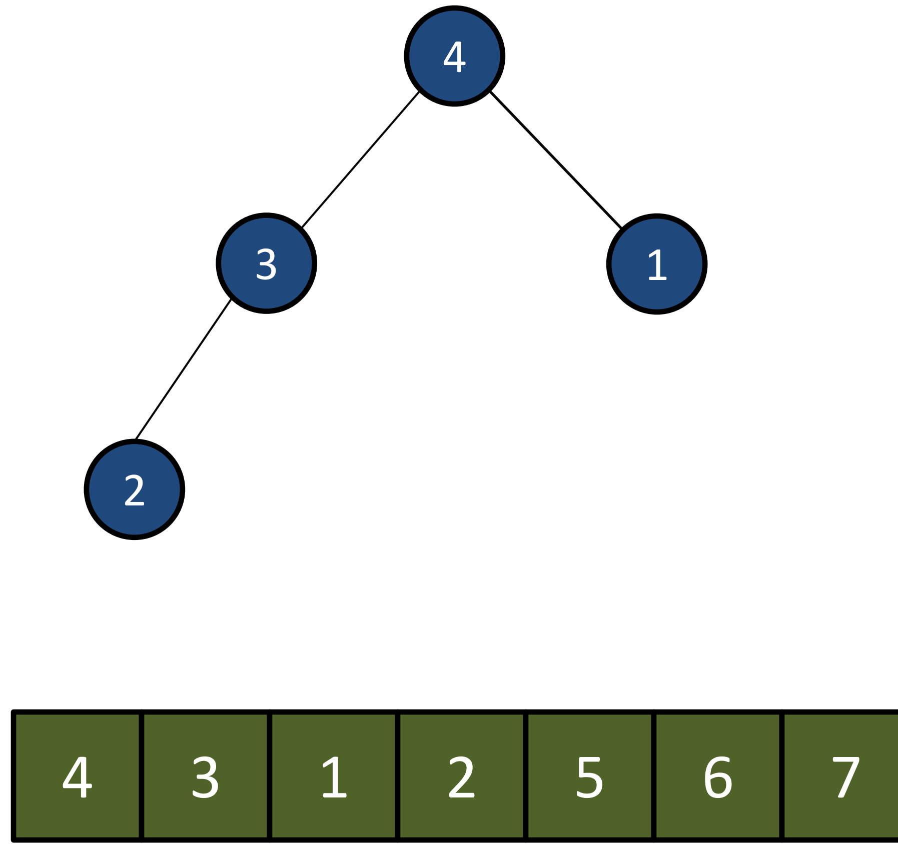
- Grab the max, 5
- Replace with the last leaf, 3
- Heapify
- Store the max



4	3	1	2	5	6	7

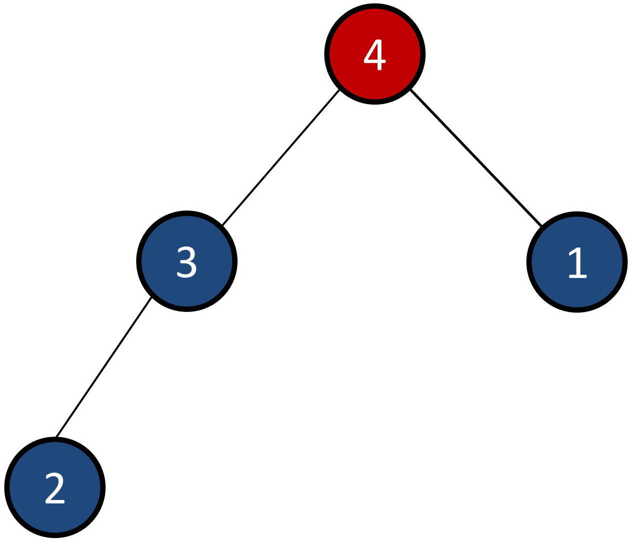
Completing the sort

Repeat



Completing the sort

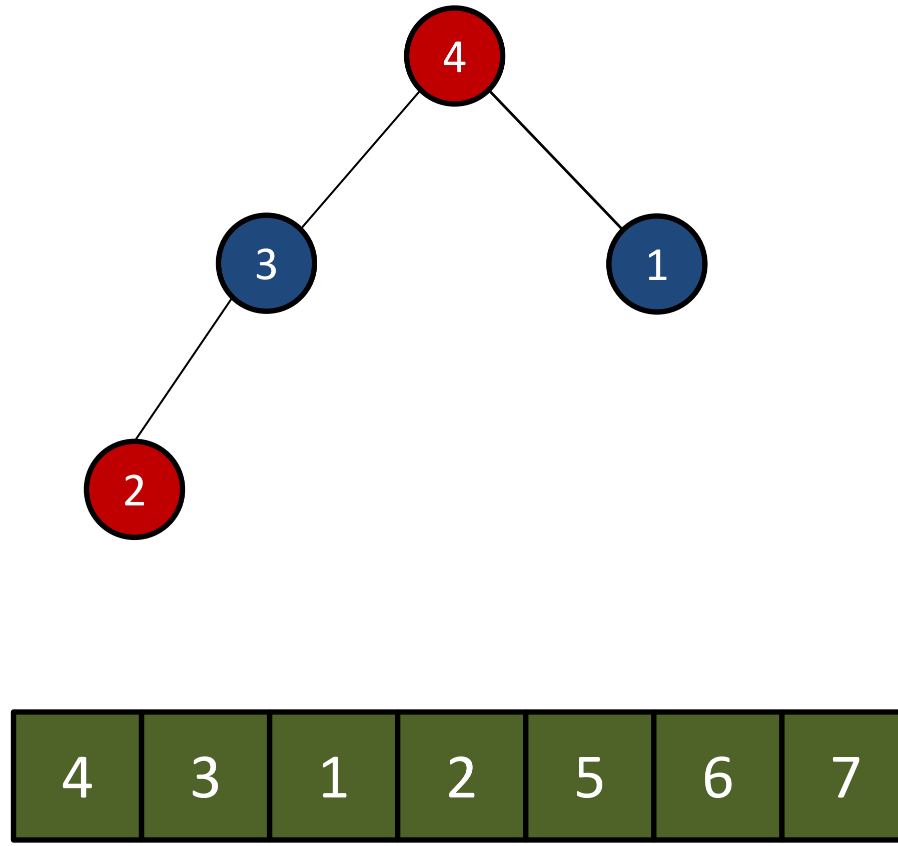
- Grab the max, 4



4	3	1	2	5	6	7

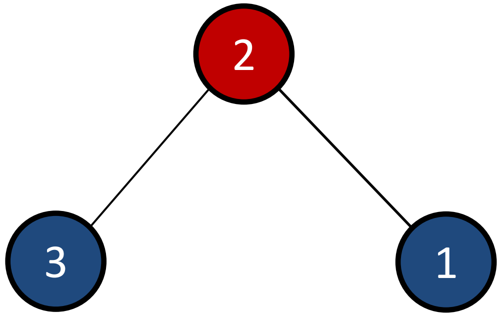
Completing the sort

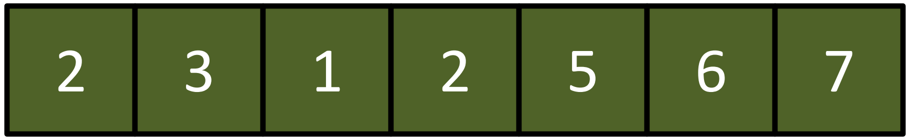
- Grab the max, 4
- Replace with the last leaf, 2



Completing the sort

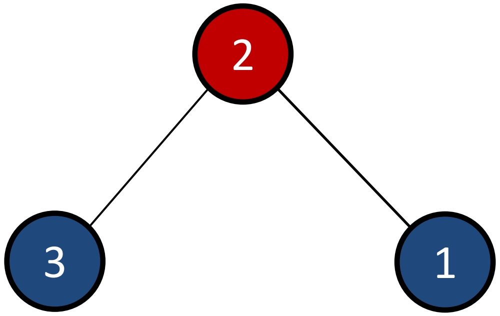
- Grab the max, 4
- Replace with the last leaf, 2

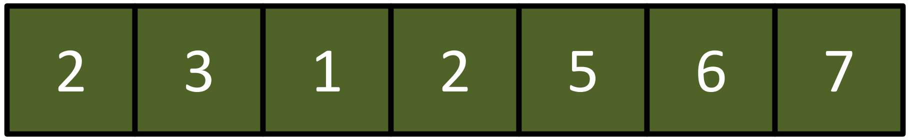




Completing the sort

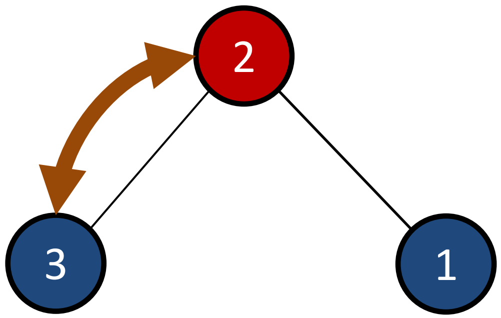
- Grab the max, 4
- Replace with the last leaf, 2
- Heapify

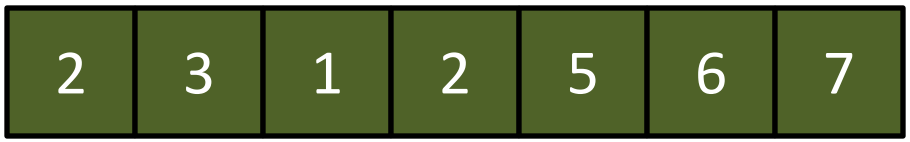




Completing the sort

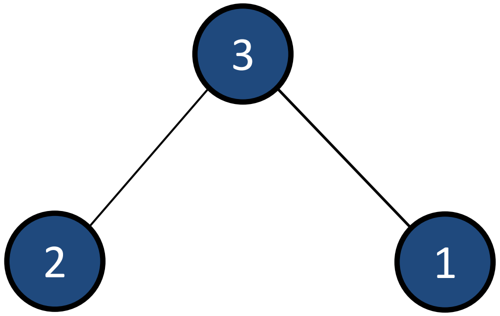
- Grab the max, 4
- Replace with the last leaf, 2
- Heapify

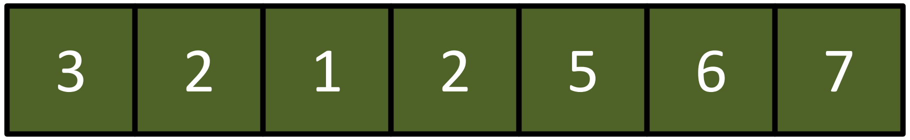




Completing the sort

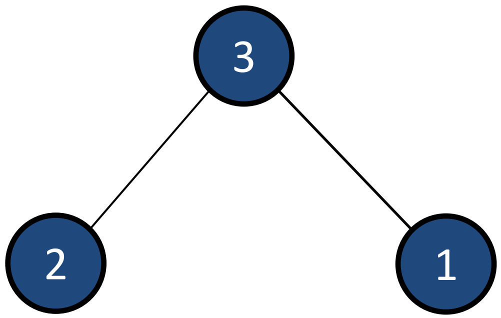
- Grab the max, 4
- Replace with the last leaf, 2
- Heapify





Completing the sort

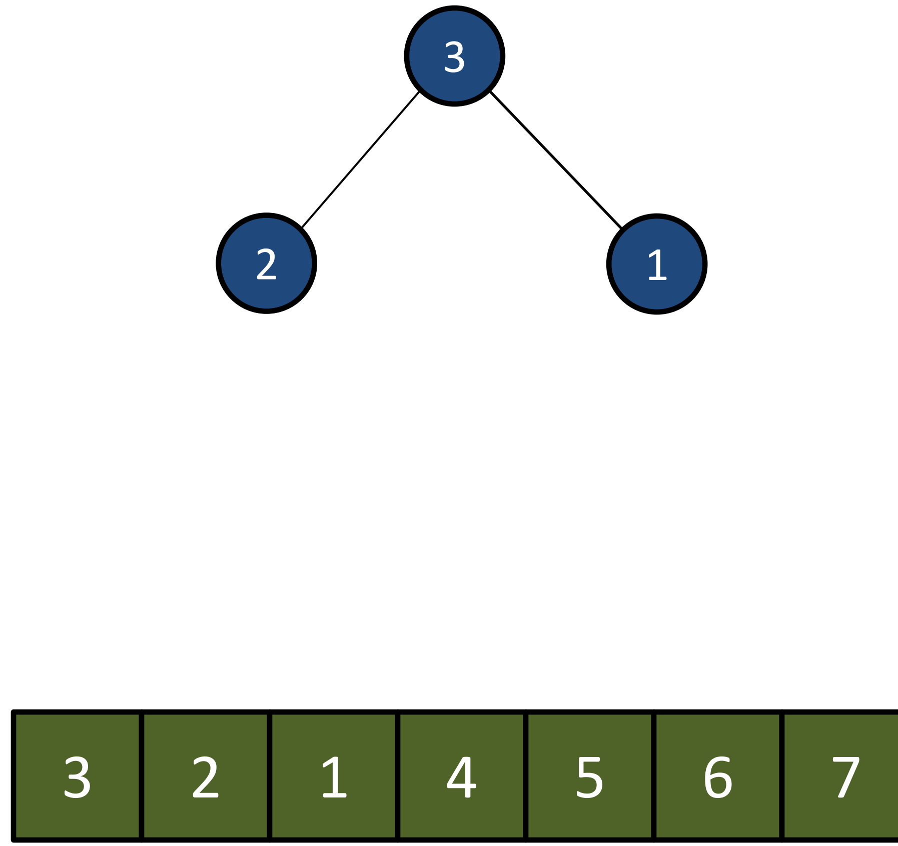
- Grab the max, 4
- Replace with the last leaf, 2
- Heapify
- Store the max



3	2	1	4	5	6	7

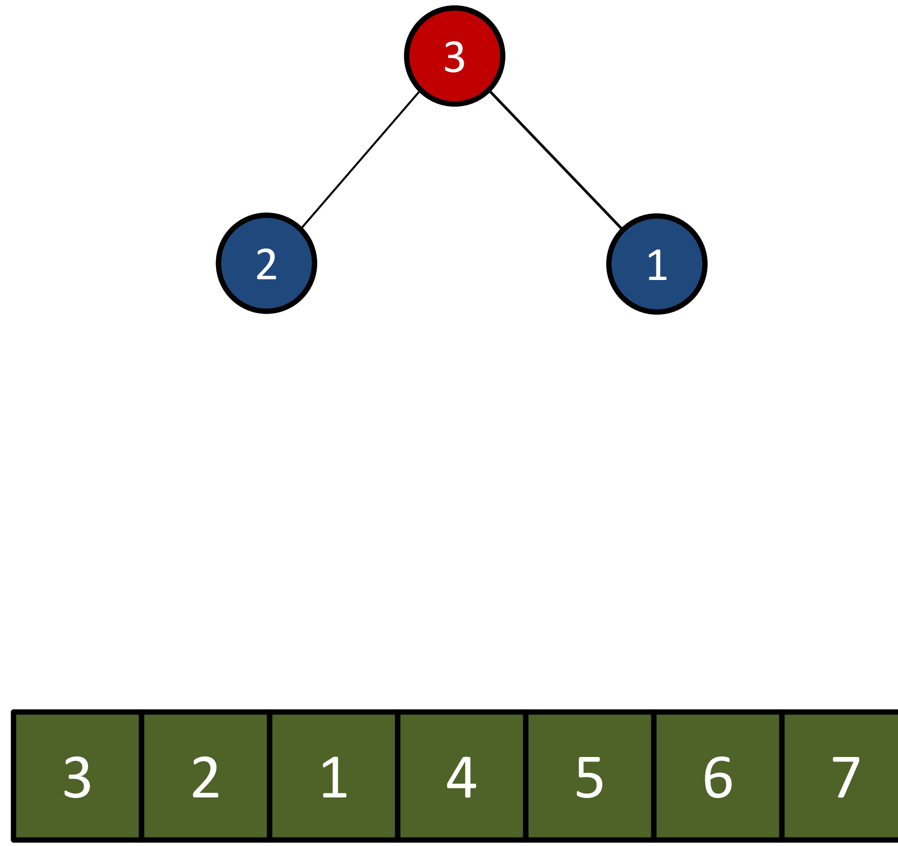
Completing the sort

Repeat



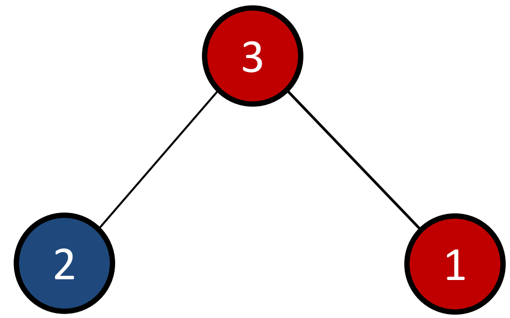
Completing the sort

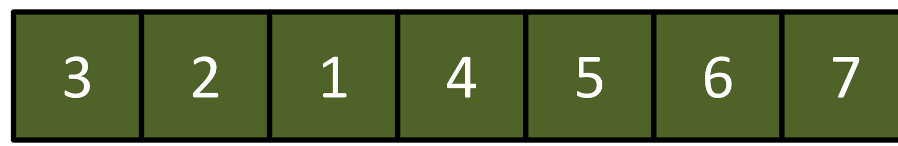
- Grab the max, 3



Completing the sort

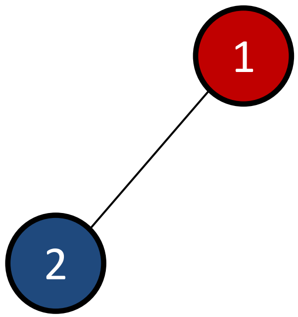
- Grab the max, 3
- Replace with the last leaf, 1





Completing the sort

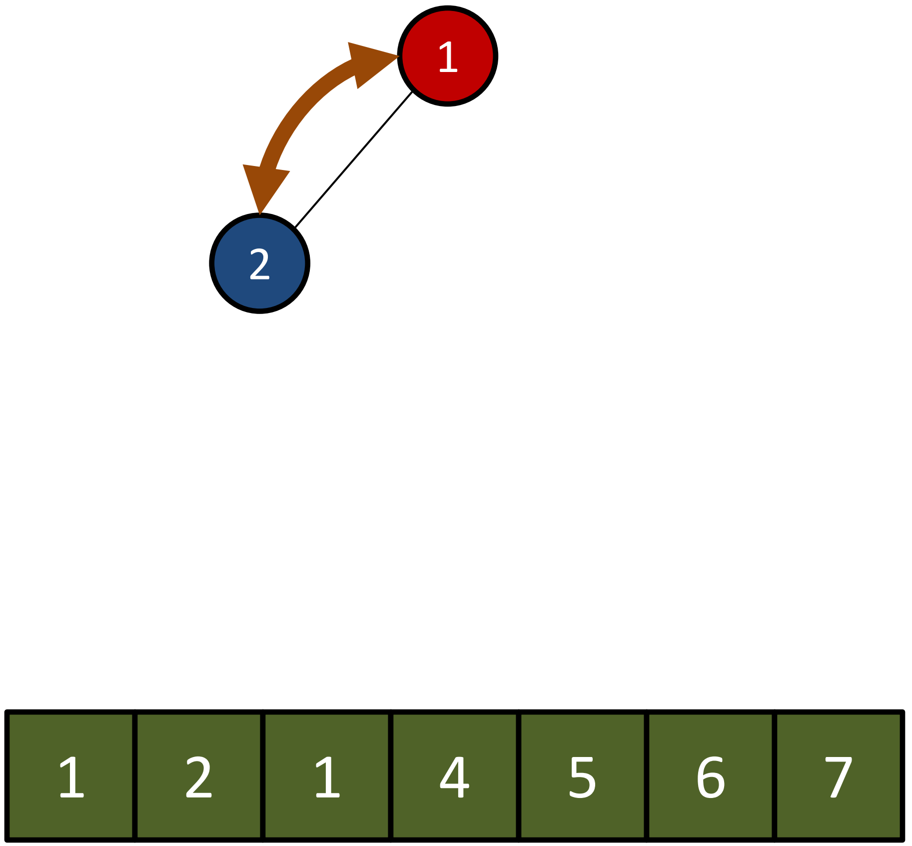
- Grab the max, 3
- Replace with the last leaf, 1



1	2	1	4	5	6	7

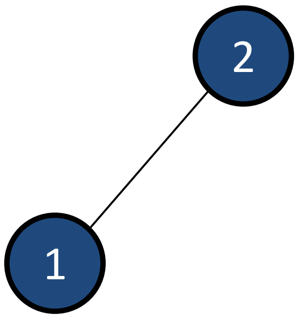
Completing the sort

- Grab the max, 3
- Replace with the last leaf, 1
- Heapify



Completing the sort

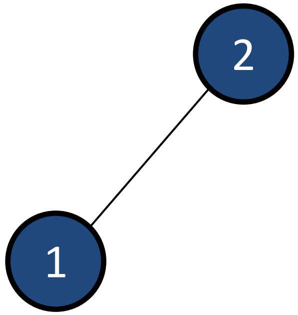
- Grab the max, 3
- Replace with the last leaf, 1
- Heapify



2	1	1	4	5	6	7

Completing the sort

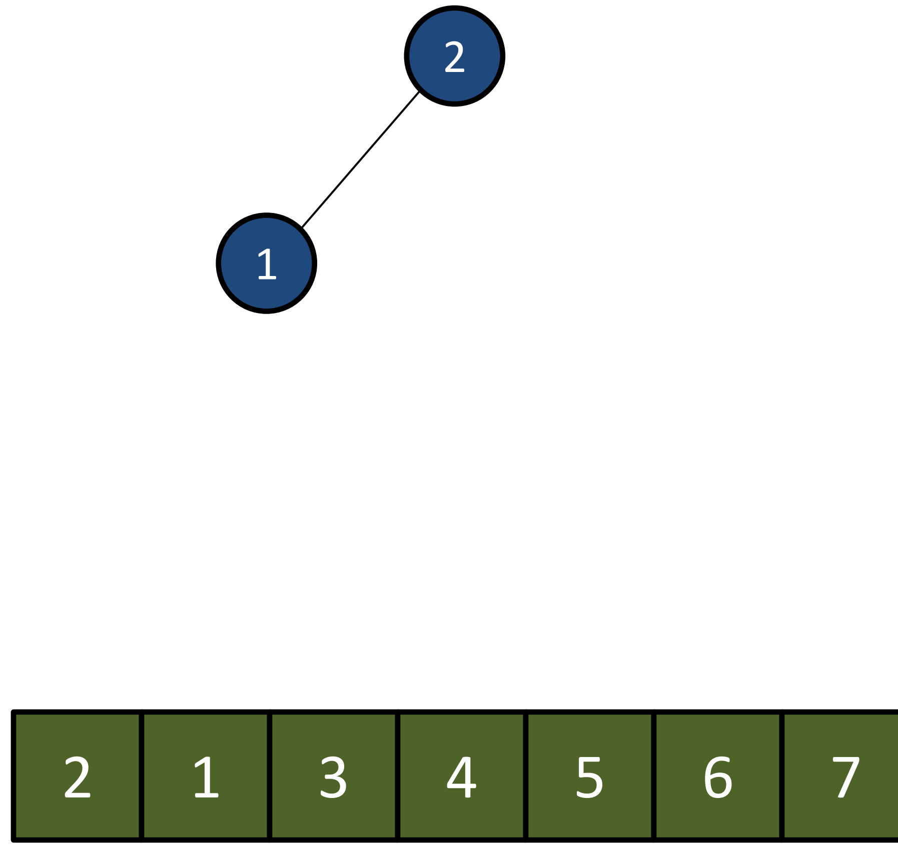
- Grab the max, 3
- Replace with the last leaf, 1
- Heapify
- Store the max



2	1	3	4	5	6	7

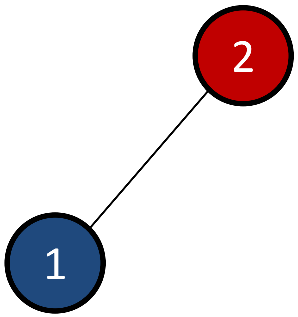
Completing the sort

Repeat



Completing the sort

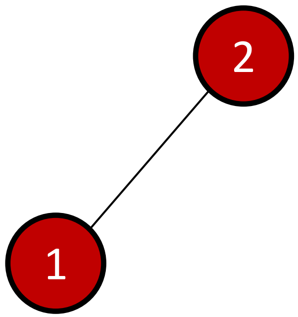
- Grab the max, 2



2	1	3	4	5	6	7

Completing the sort

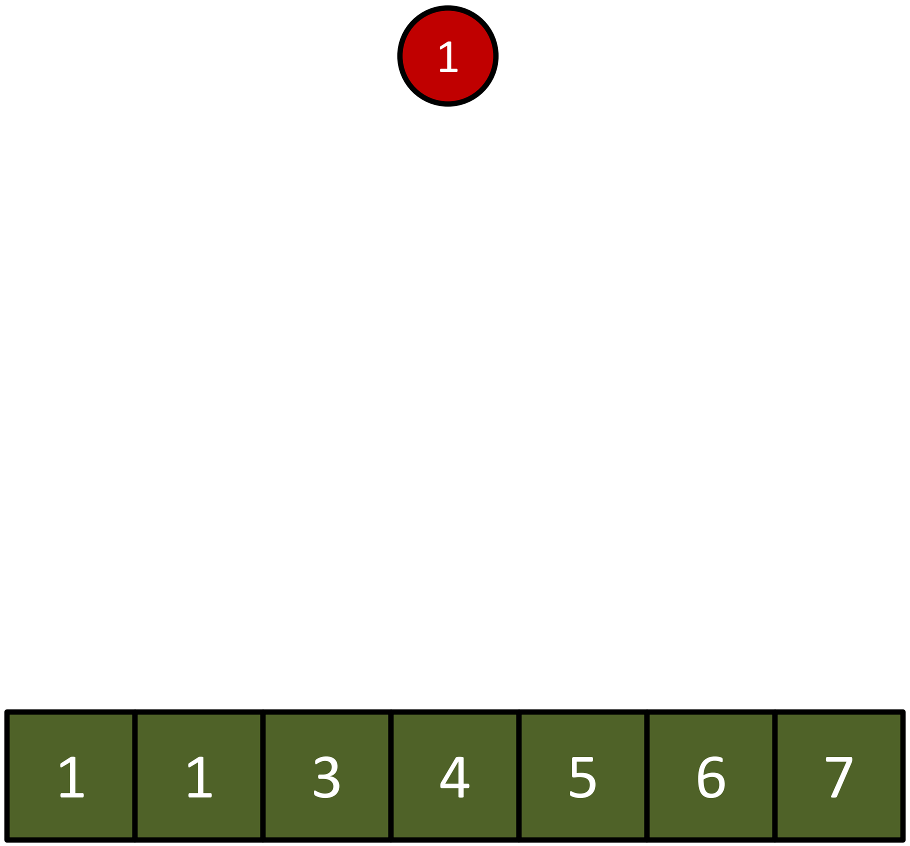
- Grab the max, 2
- Replace with the last leaf, 1



2	1	3	4	5	6	7

Completing the sort

- Grab the max, 2
- Replace with the last leaf, 1



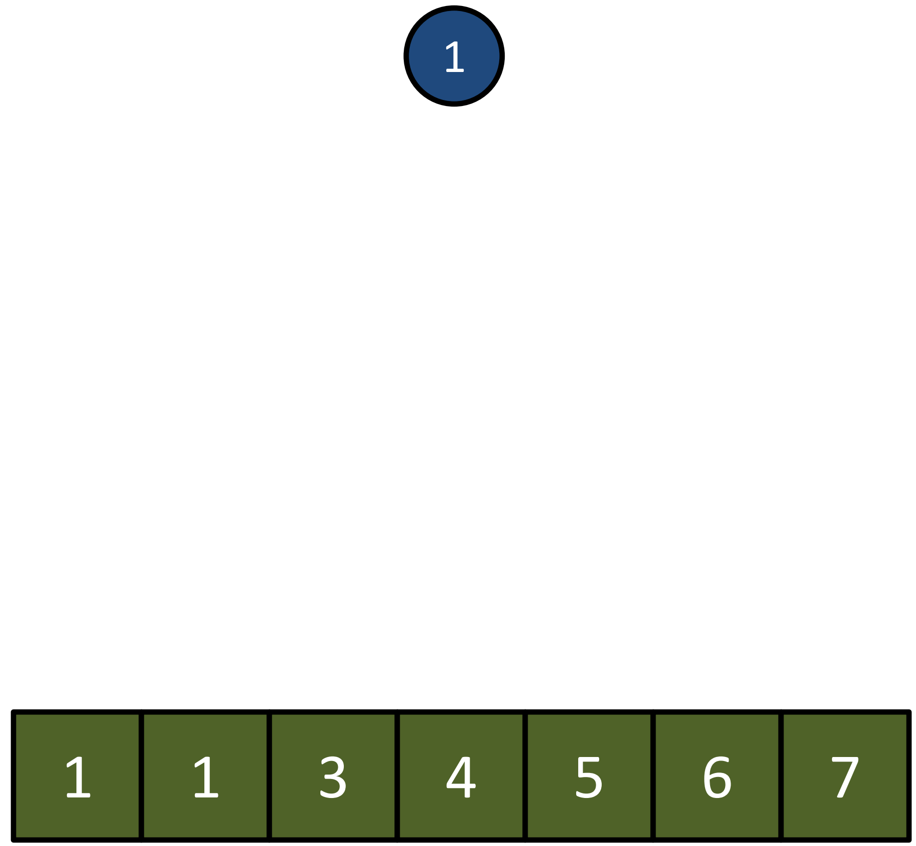
Completing the sort

- Grab the max, 2
- Replace with the last leaf, 1
- Heapify

1	1	3	4	5	6	7

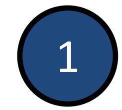
Completing the sort

- Grab the max, 2
- Replace with the last leaf, 1
- Heapify



Completing the sort

- Grab the max, 2
- Replace with the last leaf, 1
- Heapify
- Store the max

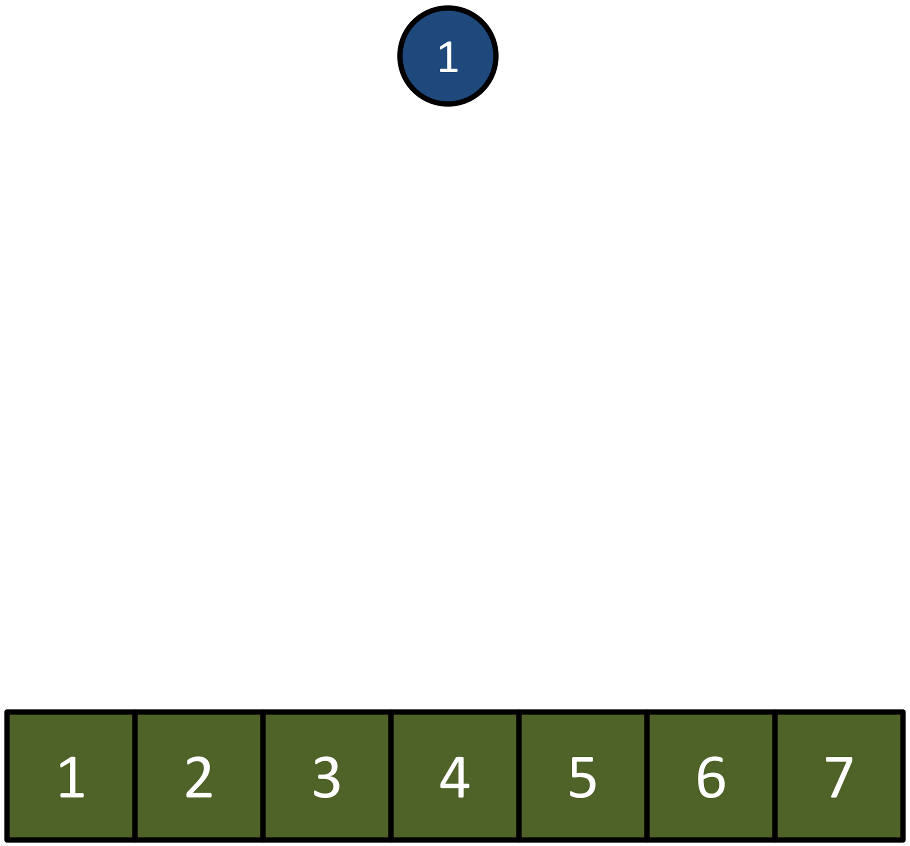


1	2	3	4	5	6	7

Completing the sort

- Only **one element left**
- So it has to be the **smallest** element
- And it has to be in **position 0**

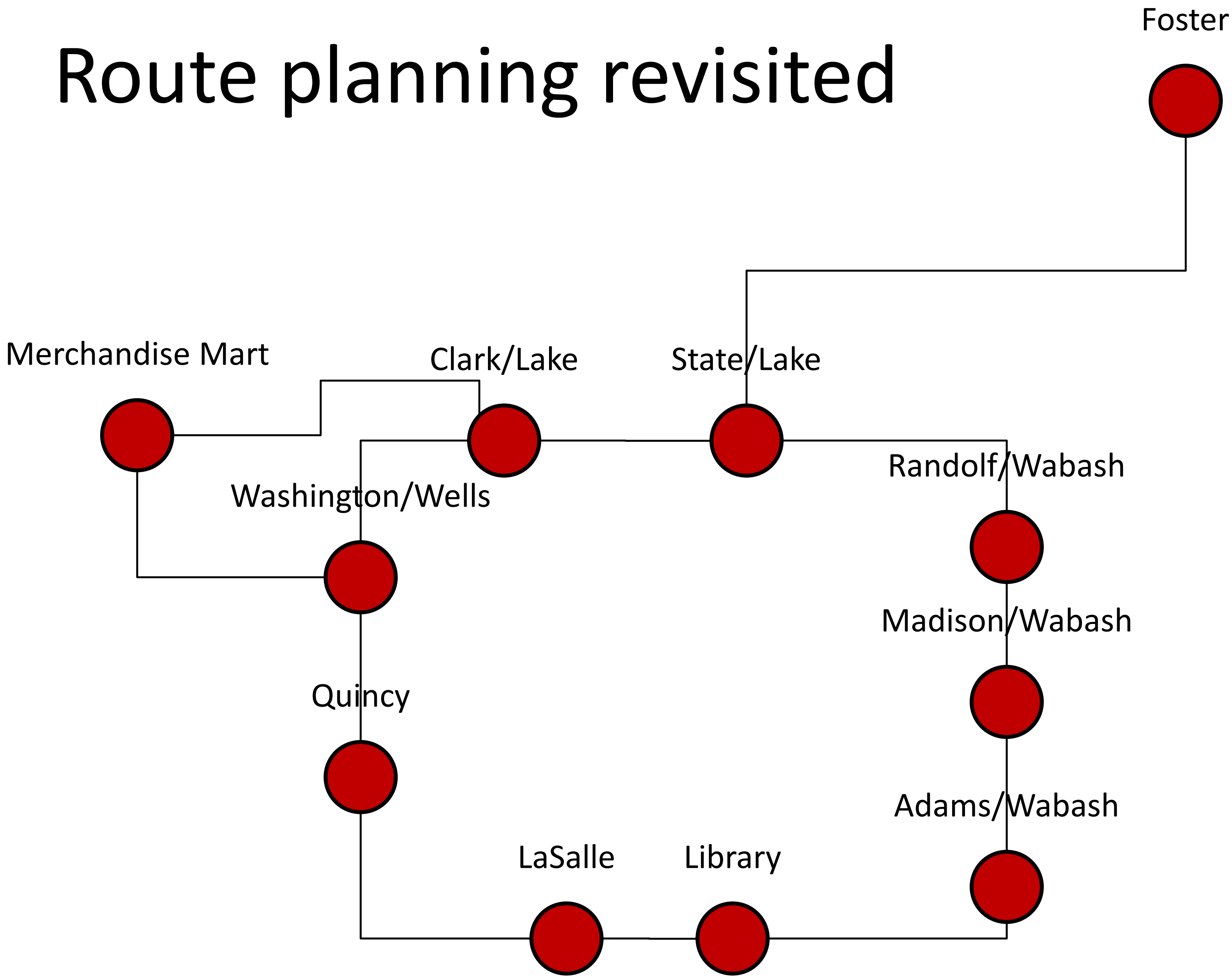
We're done!



Analysis

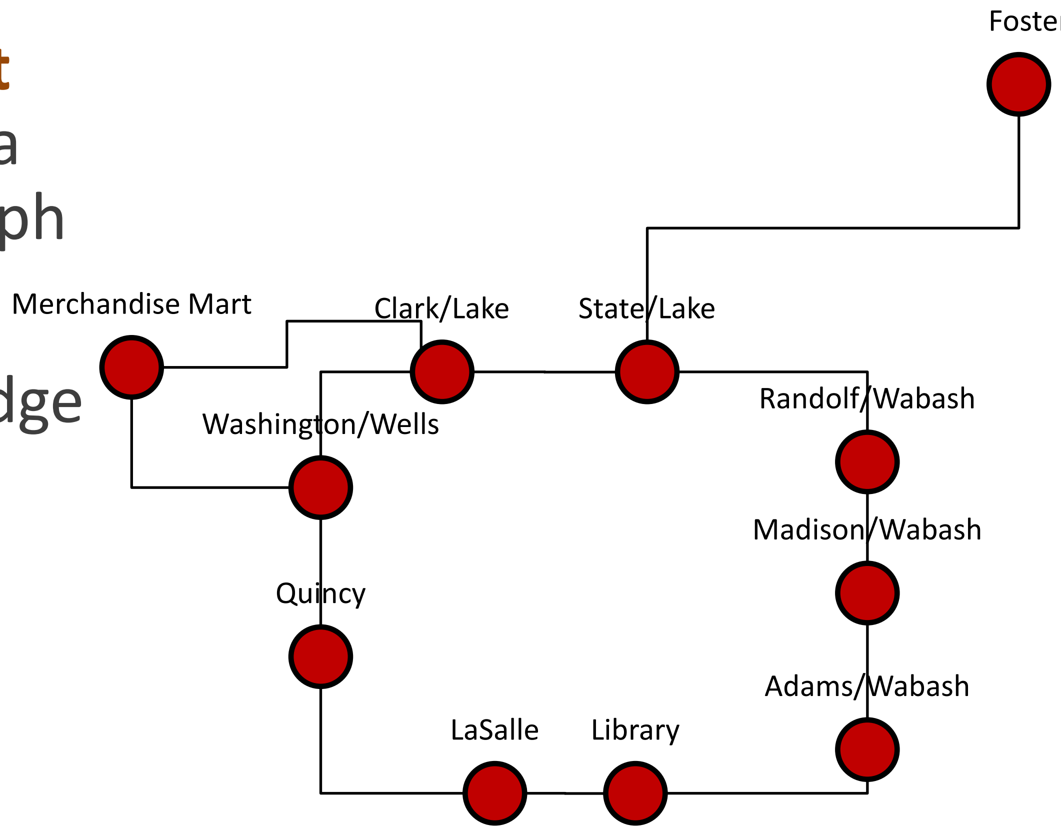
- Every **heap operation** is $\boldsymbol{O(log\, n)}$
- We do $\boldsymbol{n/2}$ calls to Heapify to build the heap
- We do $\boldsymbol{n}$ calls to ExtractMax to do the final sorting
- So execution time is $\boldsymbol{O(n\, log\, n)}$
- The nice thing about Heapsort is that it's $O(n \log n)$ **worst-case**, so it's **never quadratic**

Route planning revisited



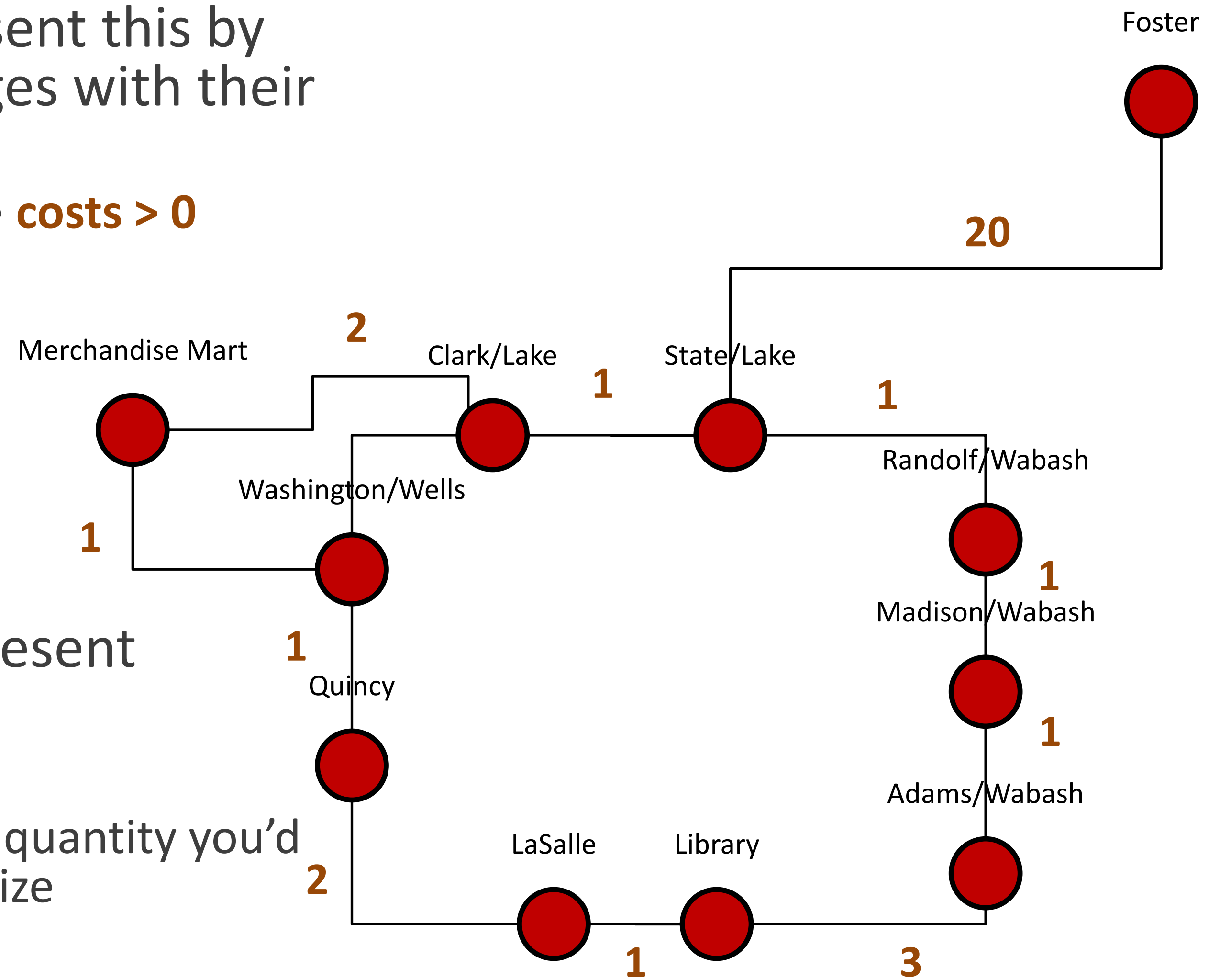
Route planning revisited

- We used **breadth-first** search before to find a **shortest path** in a graph
- But BFS treats each edge as being equal
- In route planning, different edges take **different amounts of time**



Route planning revisited

- We can represent this by **weighting** edges with their "**costs**"
  - We'll assume **costs > 0**



- Costs can represent
  - Time
  - Distance
  - Or any other quantity you'd like to minimize

Path finding with edge costs

- How do we make a version of BFS that searches nodes
  - In order of **increasing total path cost**
  - Rather than increasing **number of edges**

- Use a **priority queue**!
  - **Priority** of a node = **cost of path from start to that node**

Dijkstra’s least-cost path algorithm

```
Dijkstra(G, s, e)
  PQ = new priority queue
  Set all node costs to infinity
  s.cost = 0
  for each node n in G
     PQ.Insert(n, n.cost)
  while PQ not empty
     u = PQ.ExtractMin()
     if u == end then done!
     for each neighbor v of u
         w = weight of edge from u to v
         newCost = u.cost+w
         if newCost<v.cost
            PQ.DecreaseKey(v, newCost)
            v.cost = newCost
            v.predecessor = u
```

Wait a minute... decrease key?

- We have to add a new operation to our priority queue: **decreasing the priority** of an item already in the queue
  - Note that since we're using extract min, rather than extract max, decreasing the "priority" actually **moves it ahead** in the queue

- How could we implement decrease key?

Implementing DecreaseKey

- One way we could do it would be to:
  - **Remove** it (somehow)
  - **Reinsert** it

- But the insert algorithm
  - Adds it at the bottom of the heap
  - **Swaps it upward** until its priority is lower than its parent
    - At least for a min heap

```
HeapInsert(A, key)
    A.size = A.size + 1
    i = A.size
    while i>0 and
            A[Parent(i)] > key
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = key
```

Implementing DecreaseKey

So DecreaseKey is actually **easy**:

- Just **move** the node **up**
- Until its in the **right place**

- Just **copy the code for insert**
  - And **remove** the part that starts by inserting it at the end

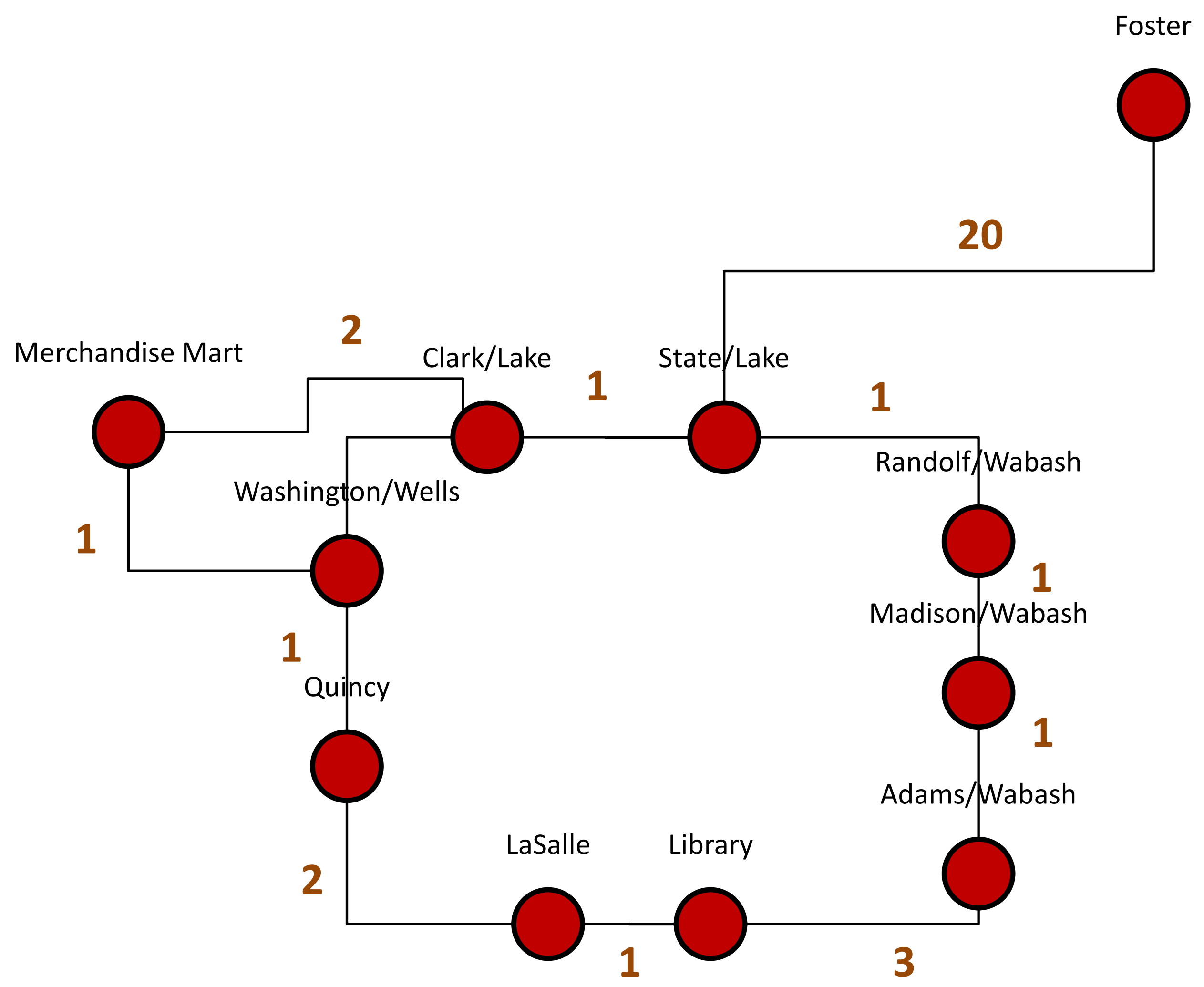
```
DecreaseKey(A, i, key)
  while i>0 and
          A[Parent(i)] > key
      A[i] = A[Parent(i)]
      i = Parent(i)
   A[i] = key
```

Implementing DecreaseKey

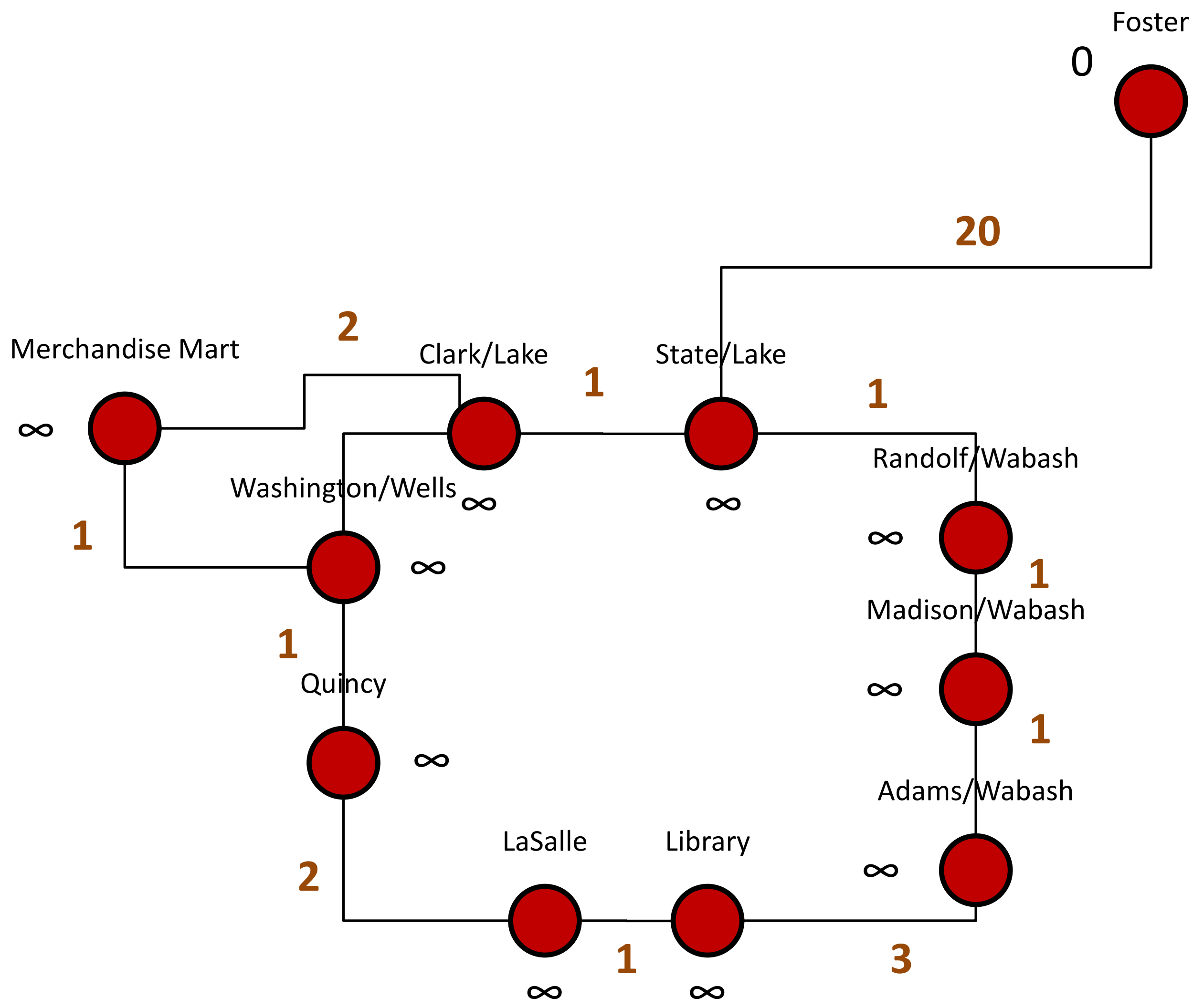
- Unfortunately, you need to know **where the node is** stored in the heap
  - i.e. its **index** in the stupid heap **array**
- Best done by storing the **index in the graph node** itself
- Have to remember to **update it** any time the node moves in the heap

```
DecreaseKey(A, i, key)
  while i>0 and
        A[Parent(i)] > key
     A[i] = A[Parent(i)]
     i = Parent(i)
   A[i] = key
```

Running Dijkstra's algorithm



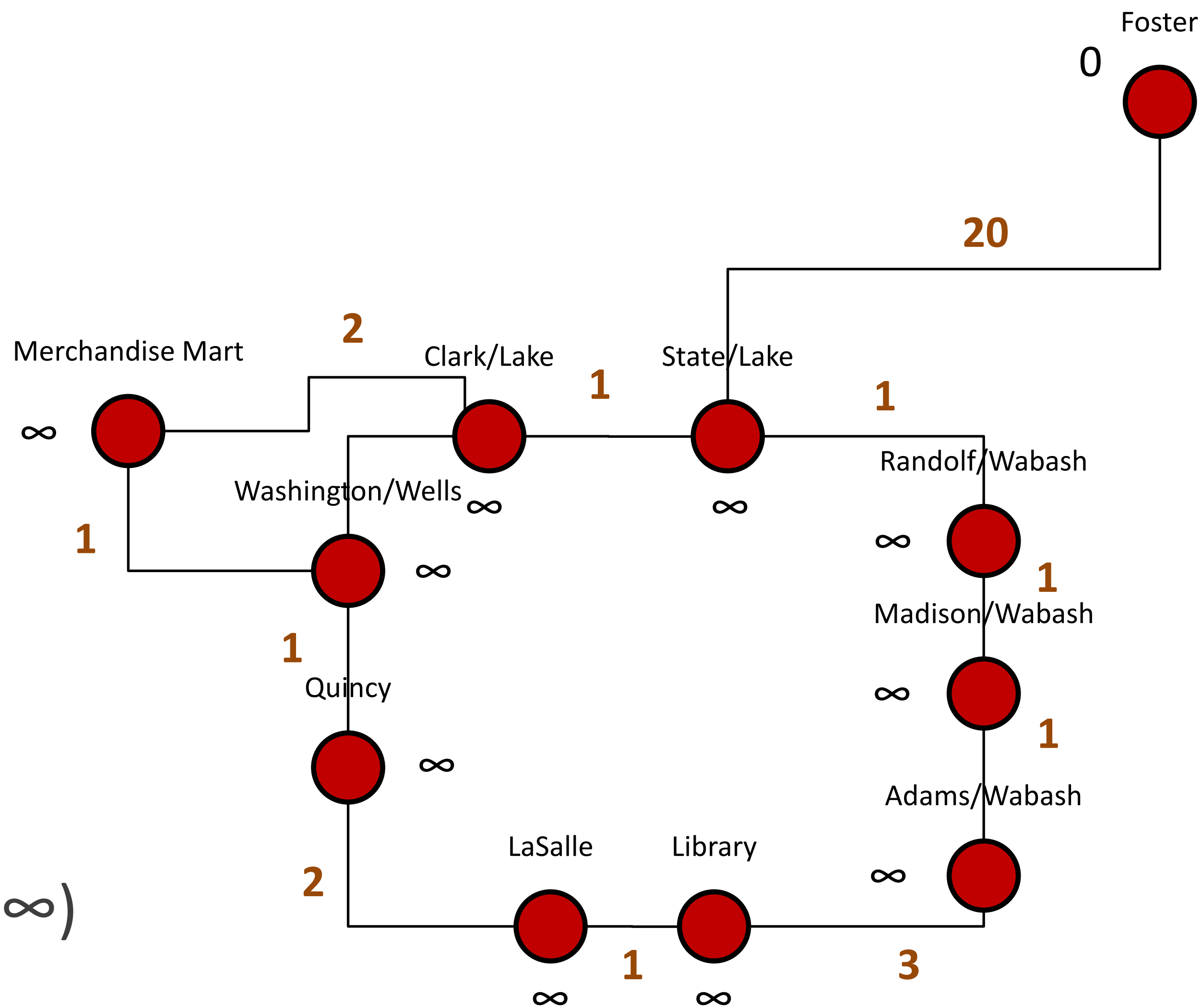
Initialize node costs



Initialize priority queue

Queue:

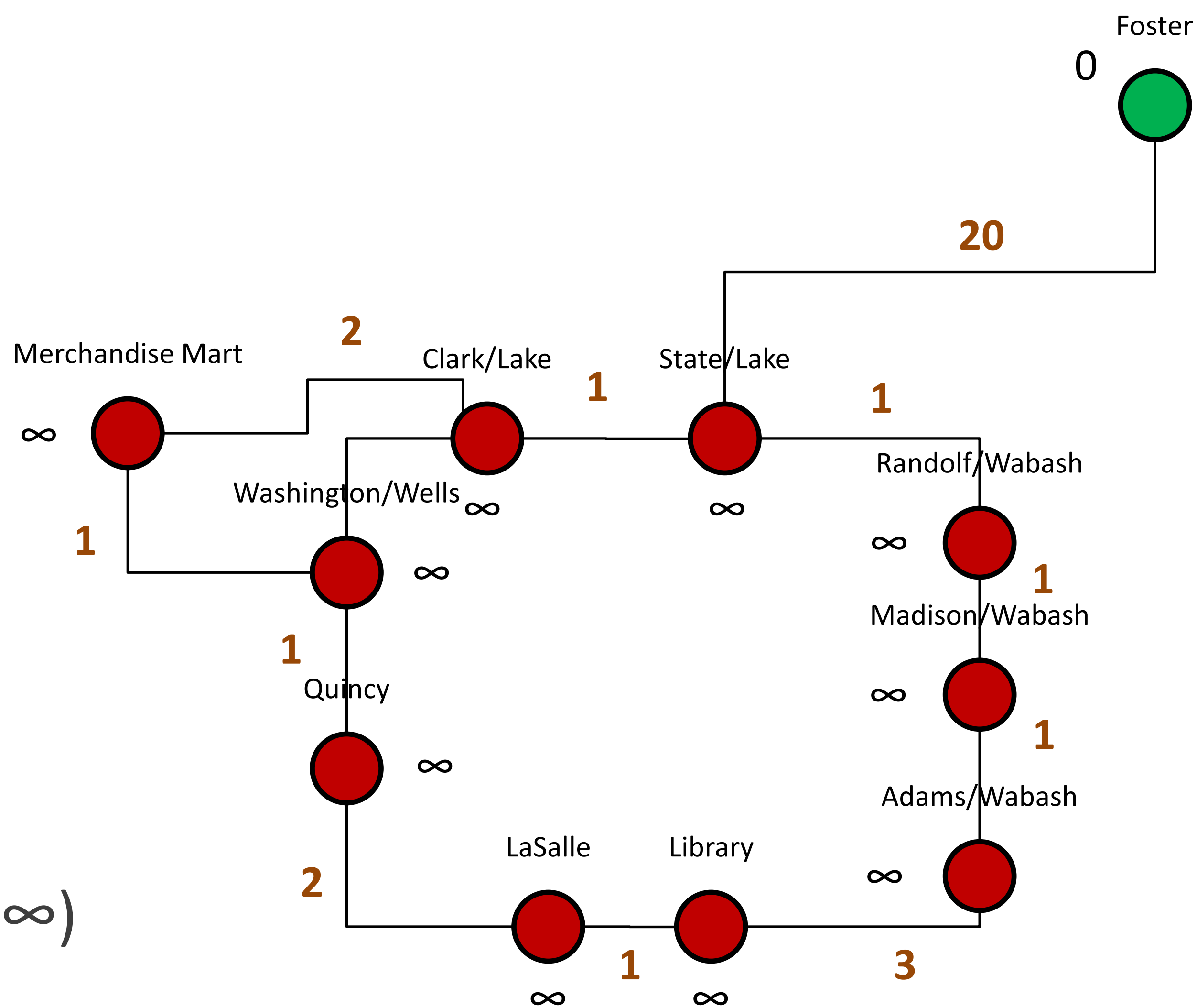
- Foster (0)
- Adams (∞)
- Clark (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Randolf (∞)
- State (∞)
- Washington (∞)



Extract min (Foster)

Queue:

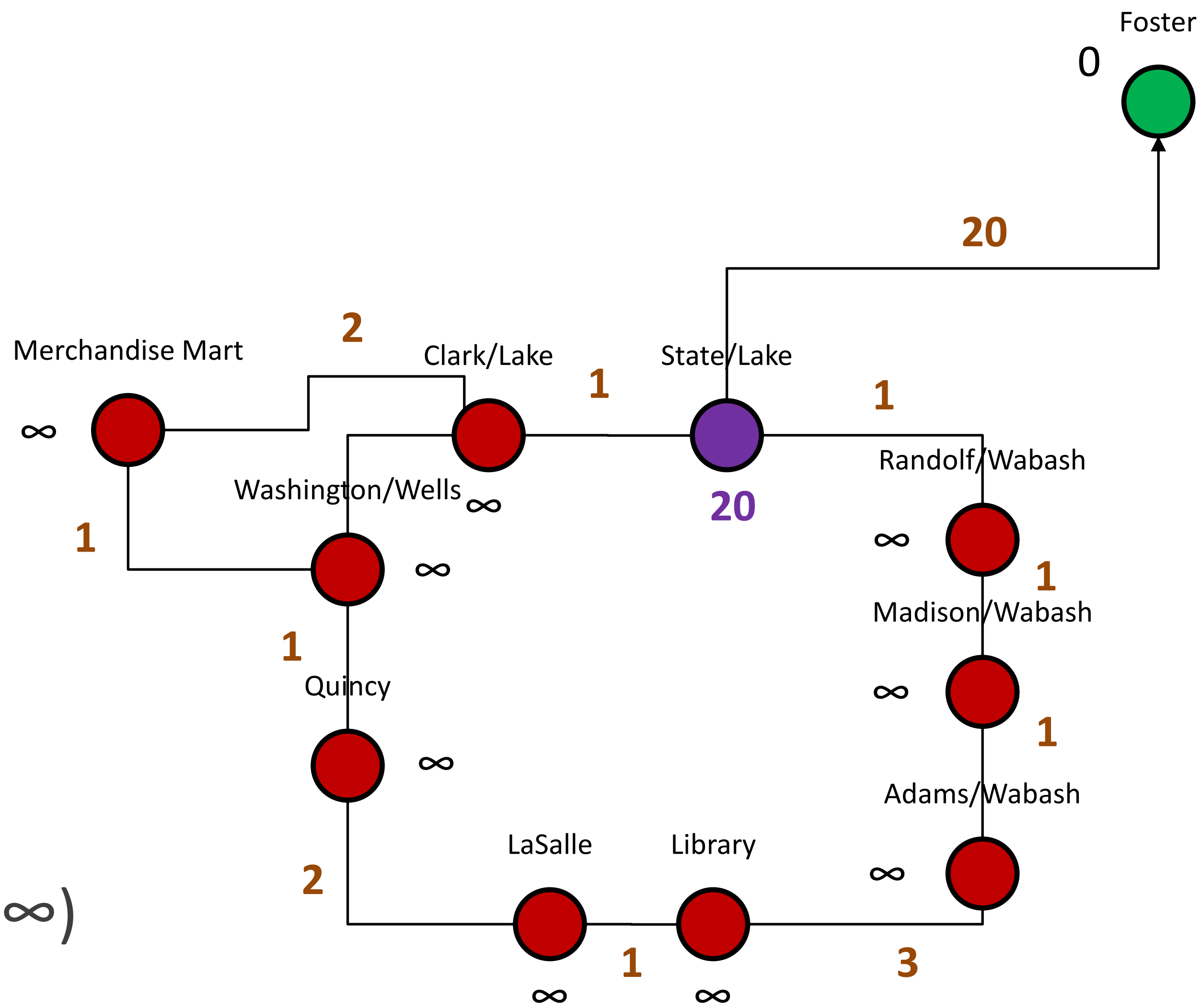
- Adams (∞)
- Clark (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Randolf (∞)
- State (∞)
- Washington (∞)



Update neighbors

Queue:

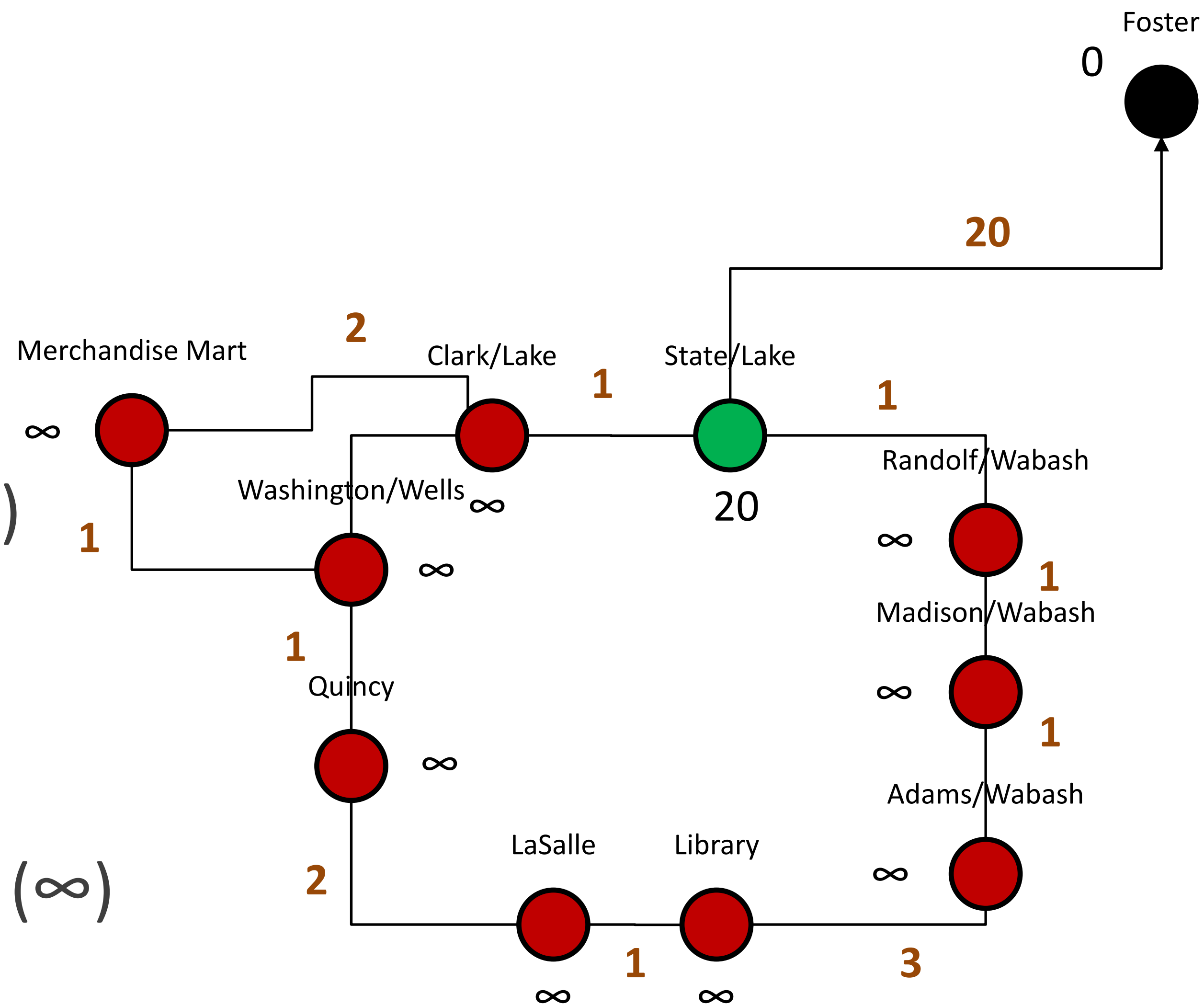
- **State (20)**
- Adams (∞)
- Clark (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Randolf (∞)
- Washington (∞)



Extract min (State)

Queue:

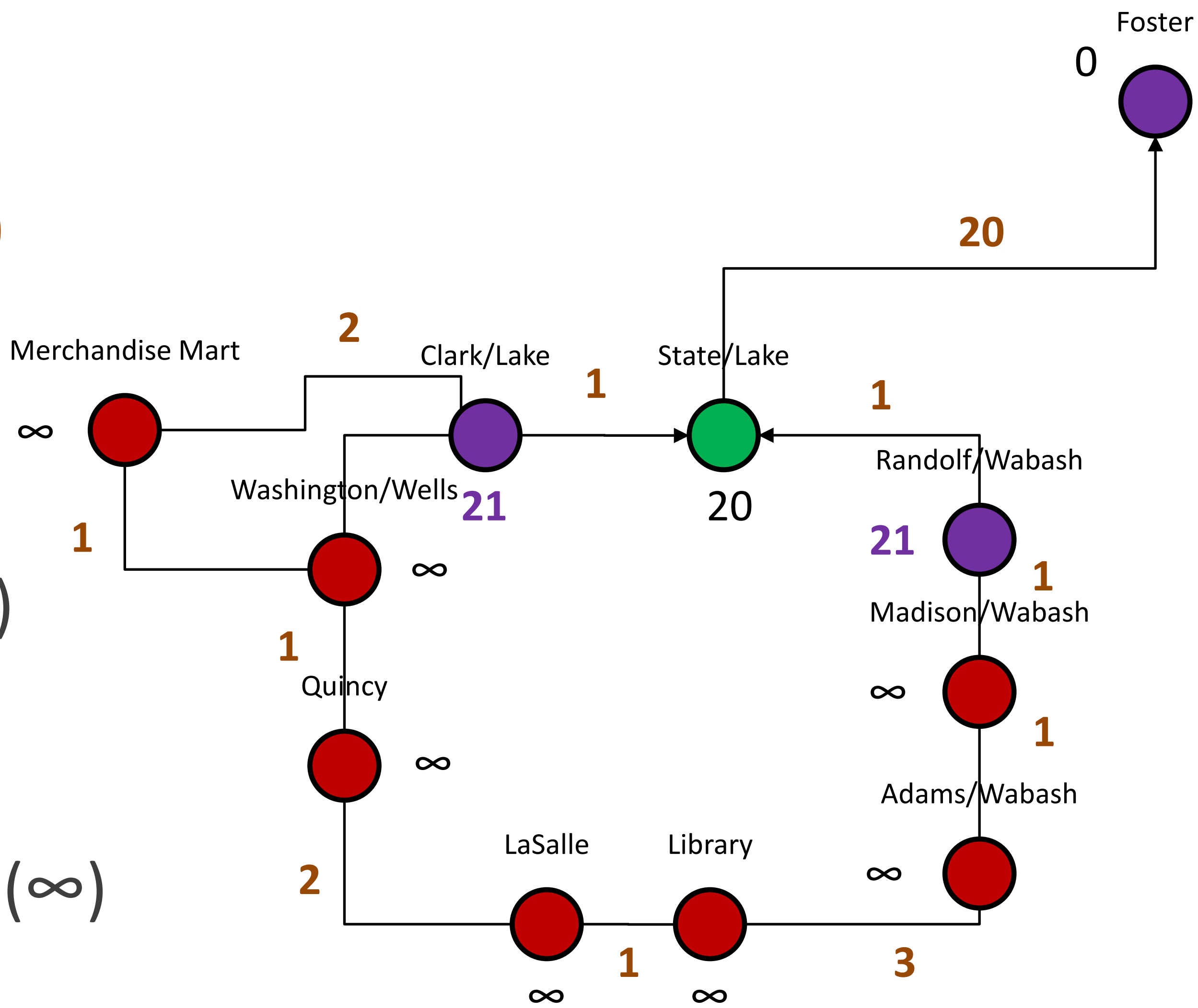
- Adams (∞)
- Clark (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Randolf (∞)
- Washington (∞)



Update neighbors

Queue:

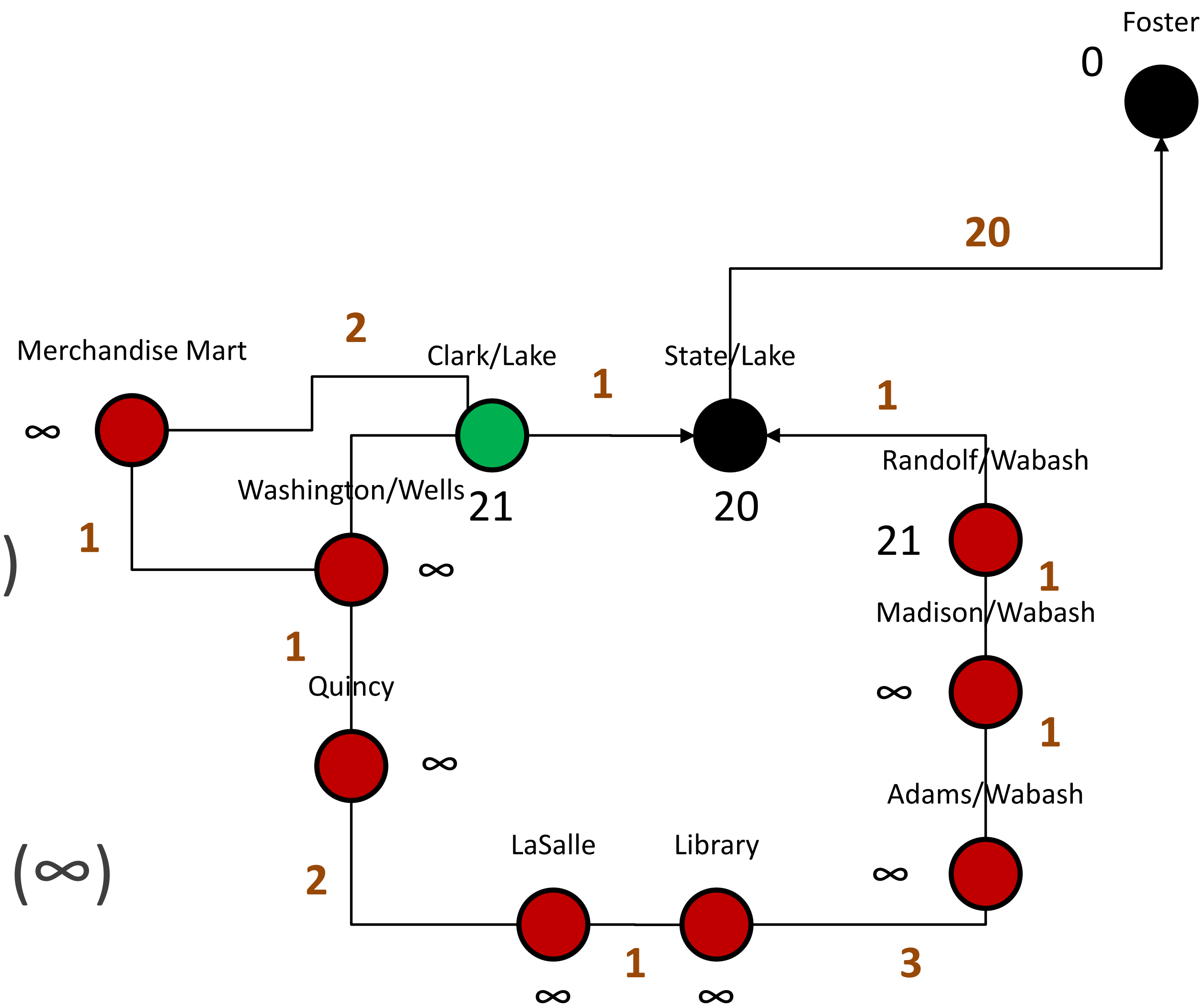
- **Clark (21)**
- **Randolf (21)**
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Washington (∞)



Extract min (Clark)

Queue:

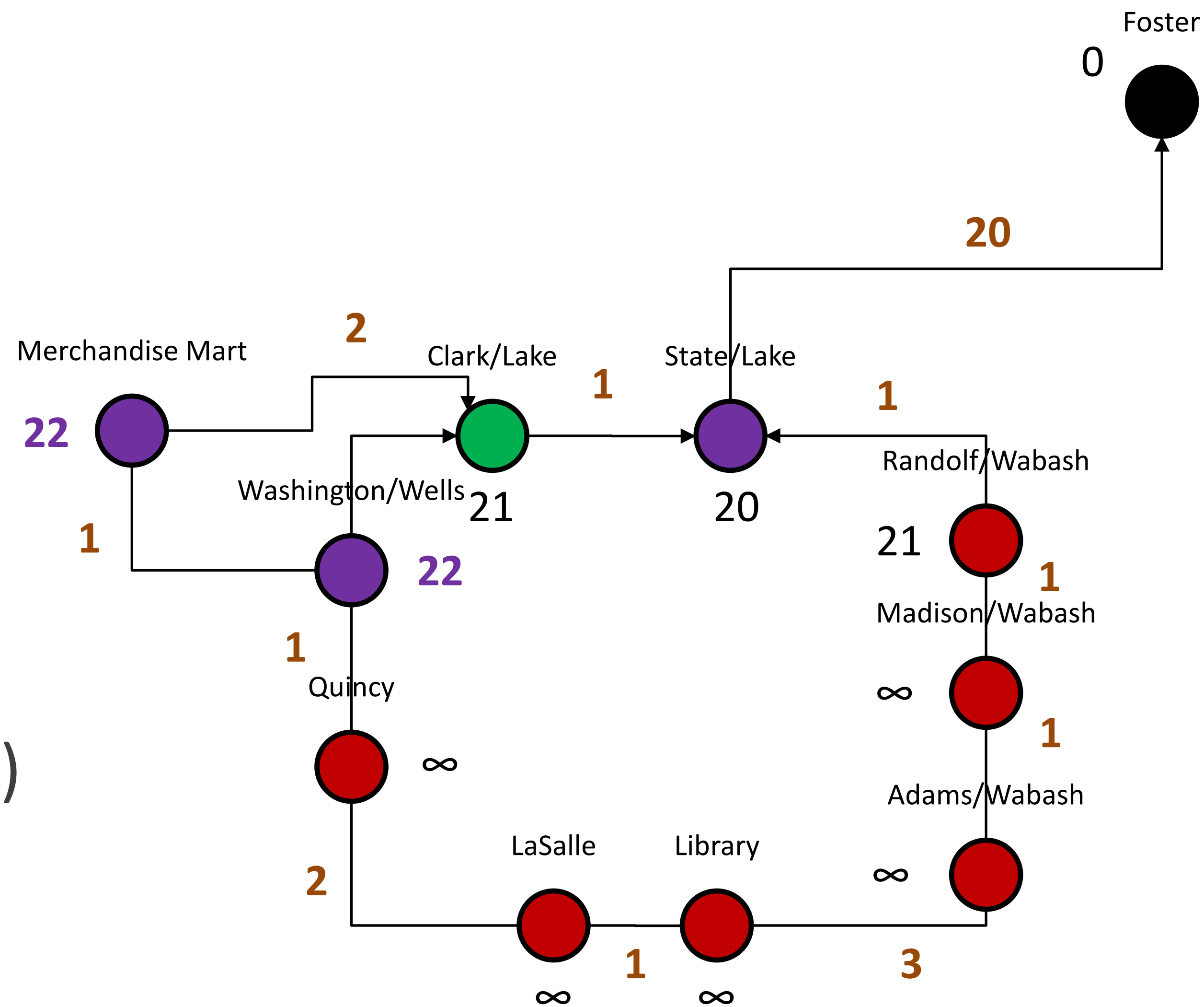
- Randolf (21)
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Mmart (∞)
- Quincy (∞)
- Washington (∞)



Update neighbors

Queue:

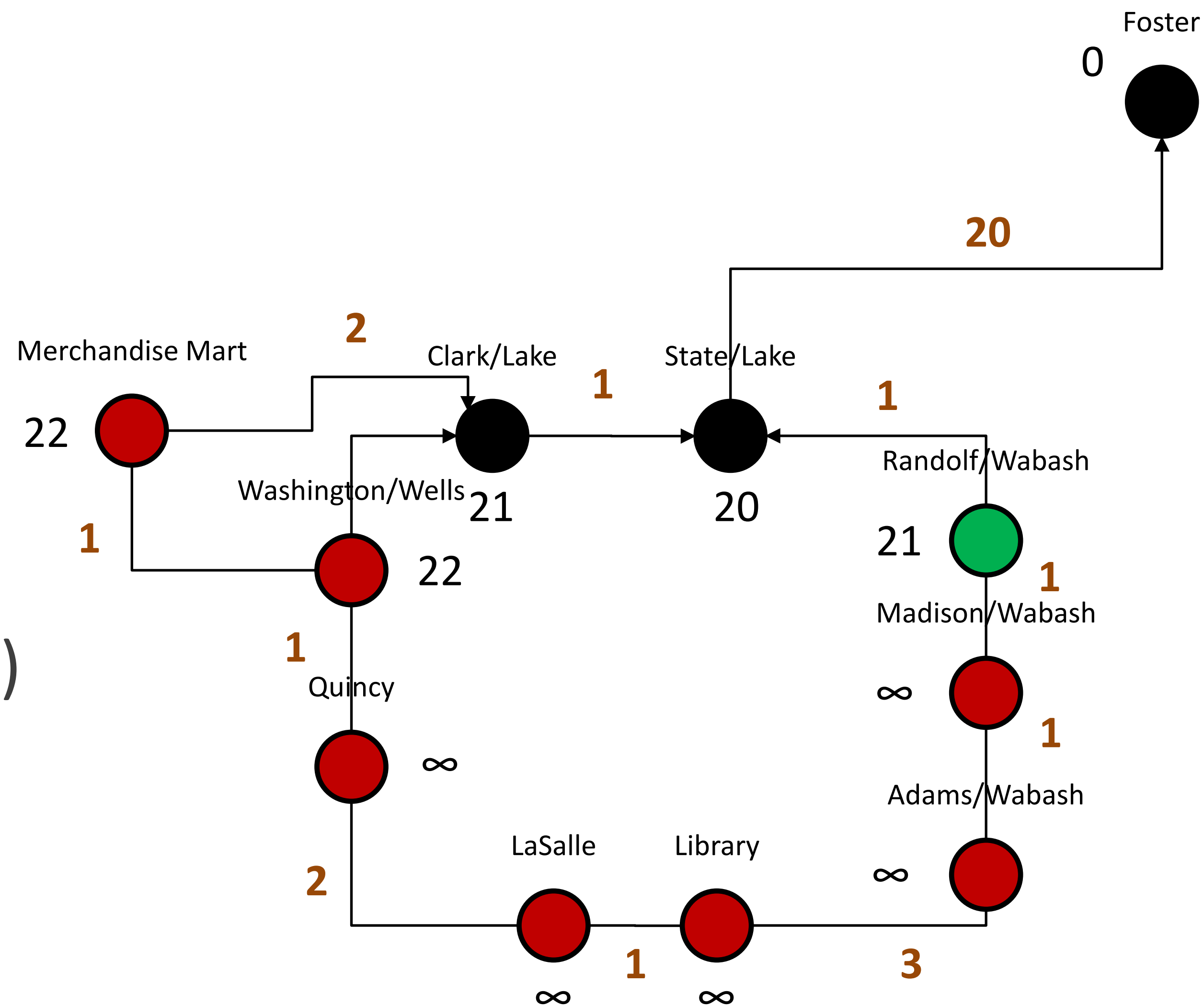
- Randolf (21)
- **Mmart (22)**
- **Wash (22)**
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Quincy (∞)



Extract min (Randolf)

Queue:

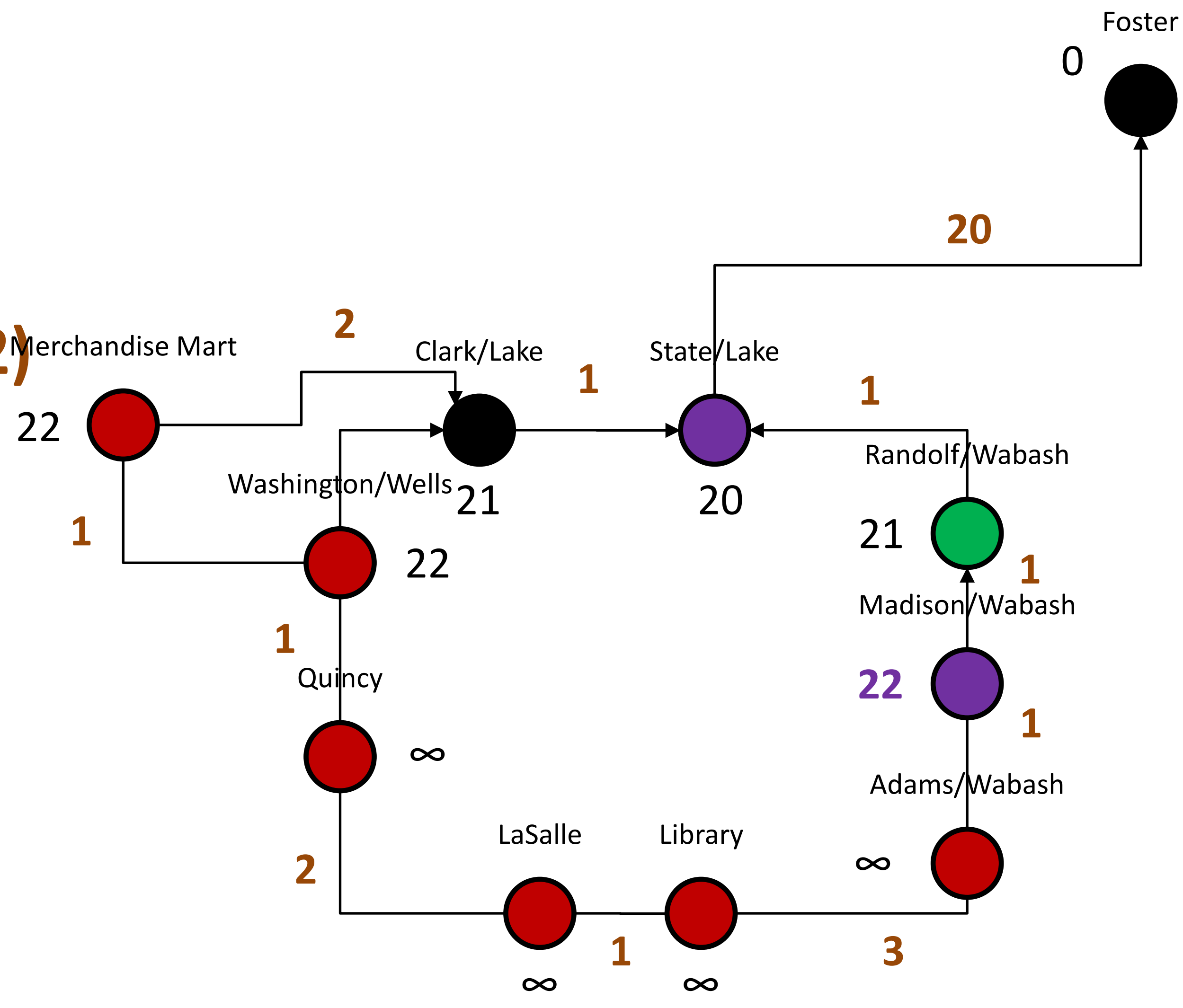
- Mmart (22)
- Wash (22)
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Madison (∞)
- Quincy (∞)



Update neighbors

Queue:

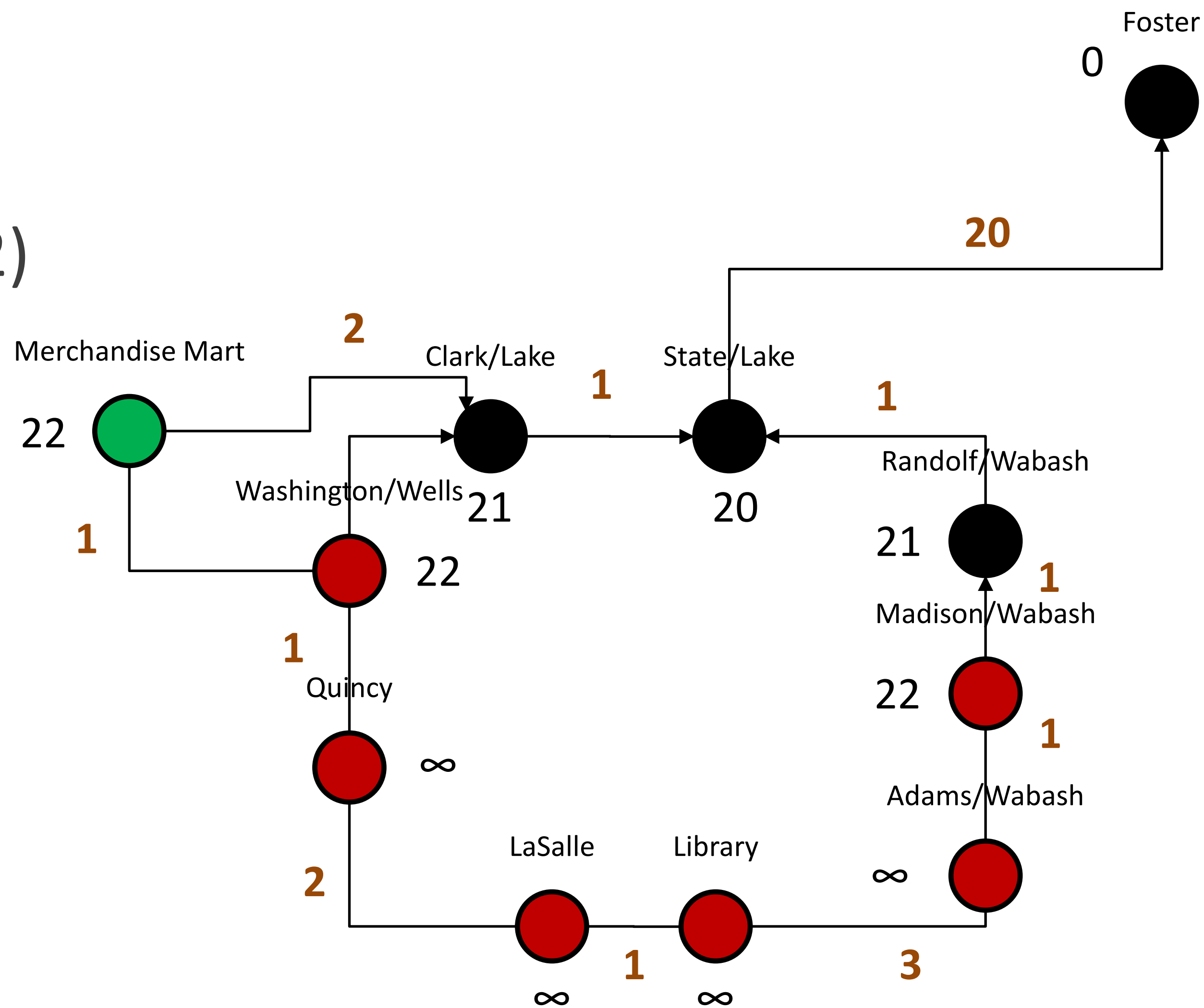
- Mmart (22)
- Wash (22)
- **Madison (22)**
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Quincy (∞)



Extract min (MMart)

Queue:

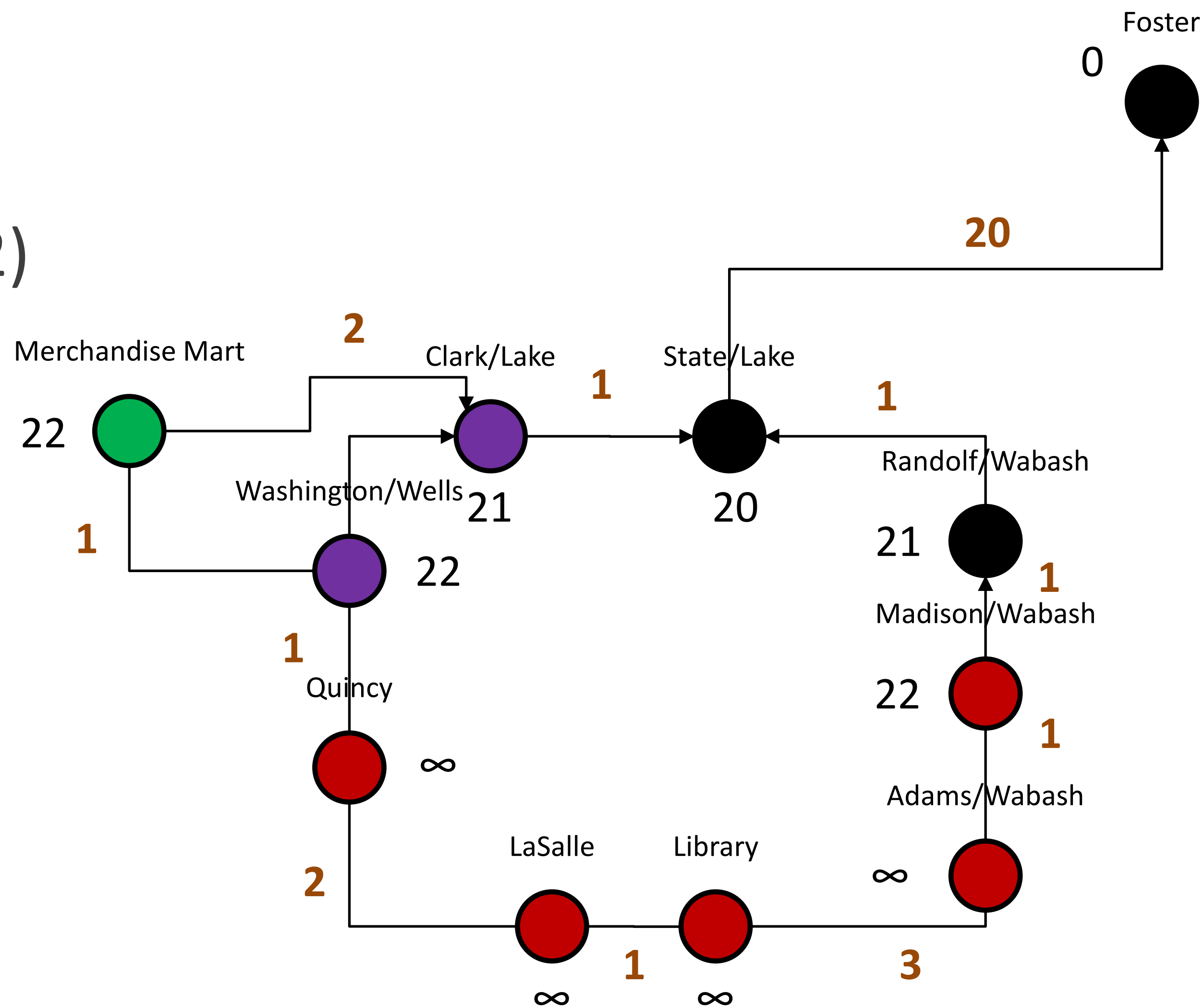
- Wash (22)
- Madison (22)
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Quincy (∞)



Update neighbors (but no distances changed)

Queue:

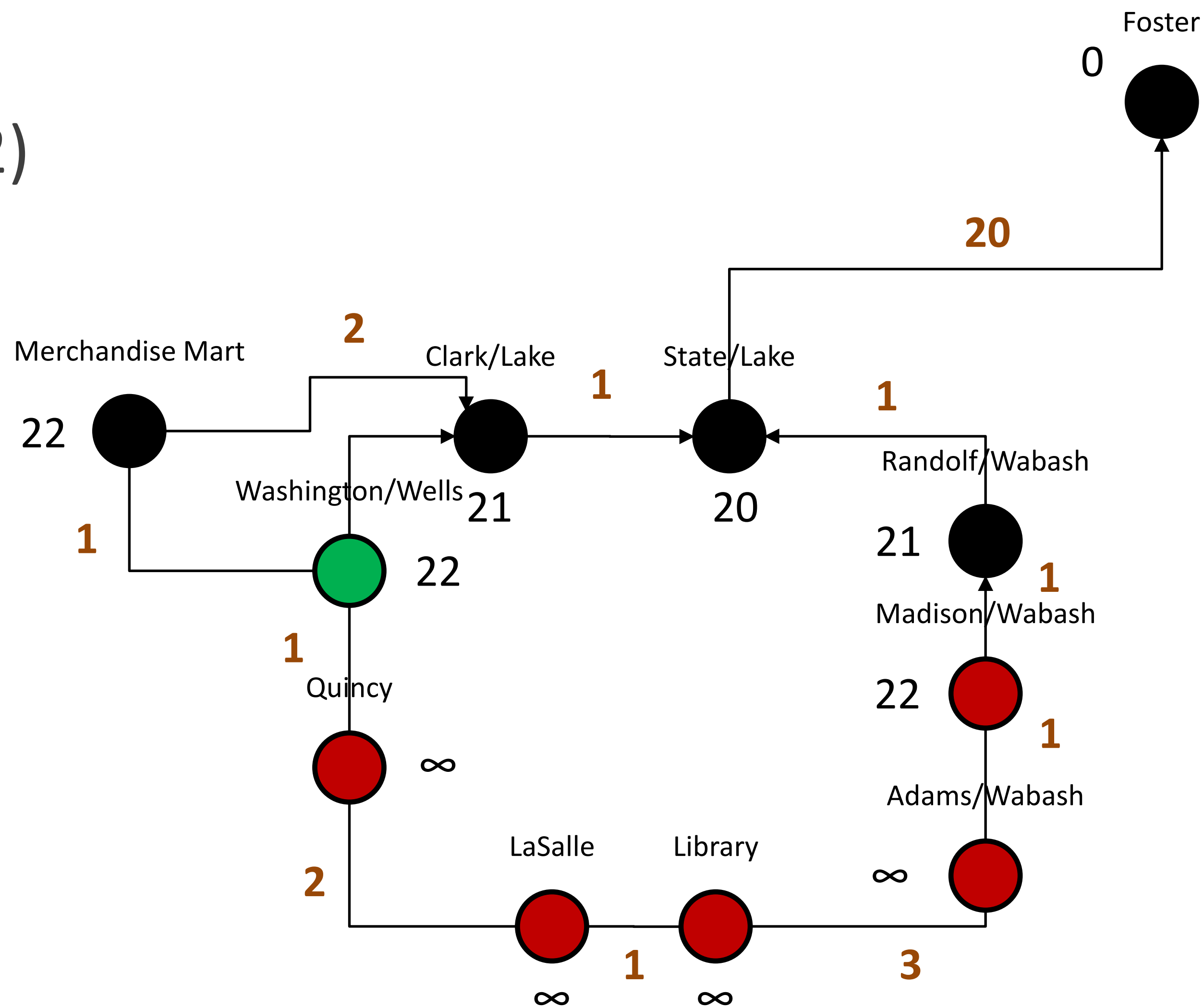
- Wash (22)
- Madison (22)
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Quincy (∞)



Extract min (Washington)

Queue:

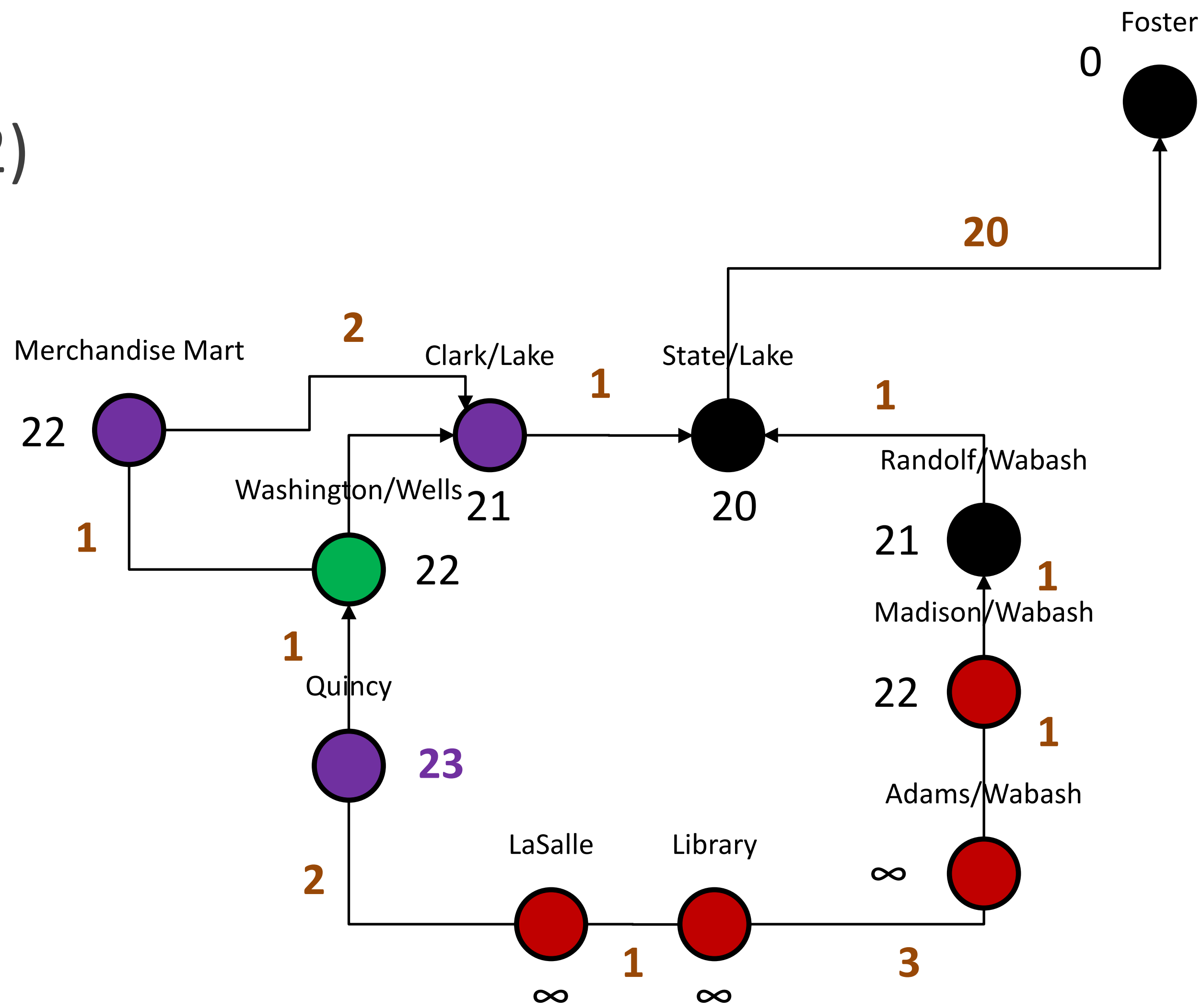
- Madison (22)
- Adams (∞)
- LaSalle (∞)
- Library (∞)
- Quincy (∞)



Update neighbors

Queue:

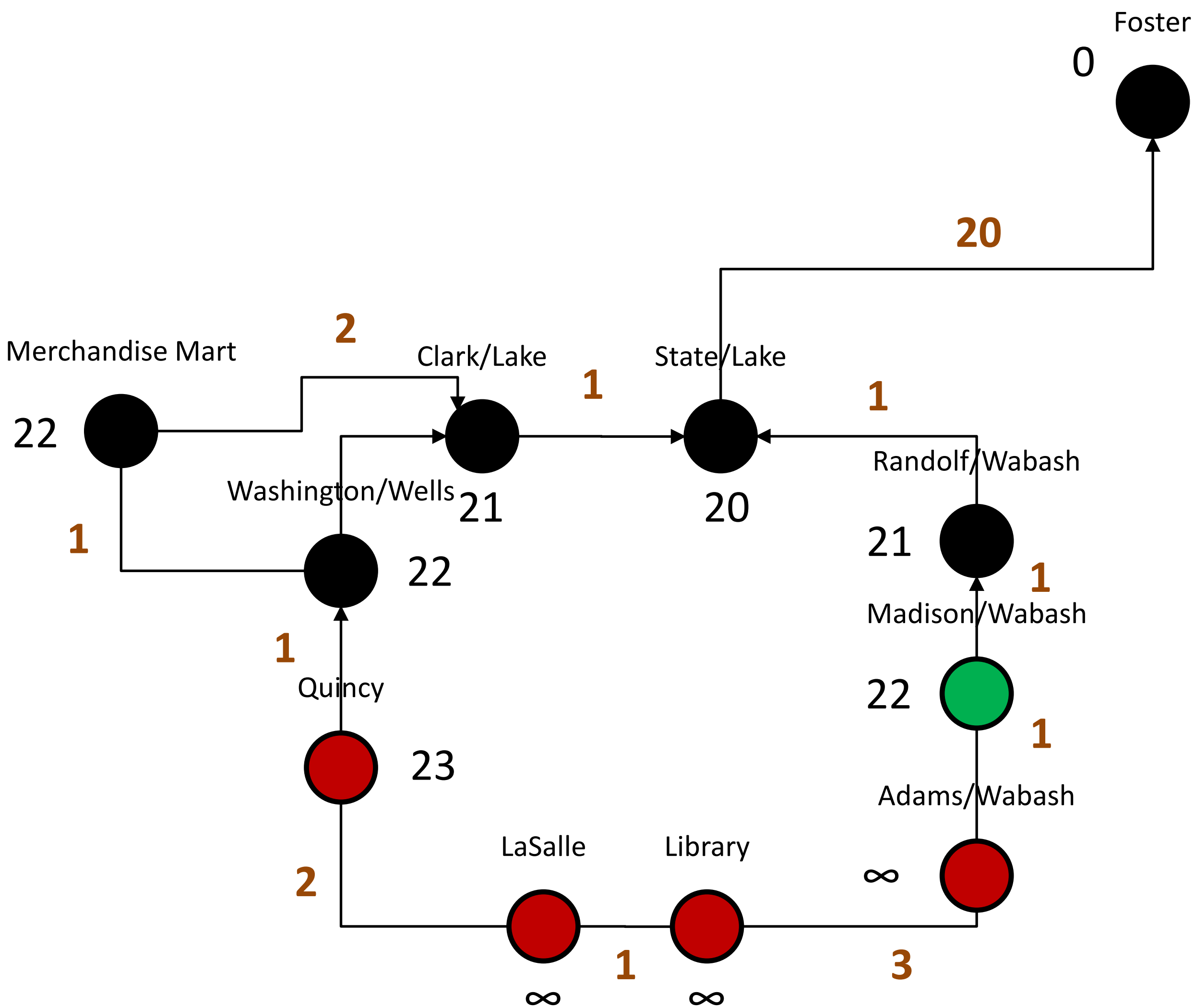
- Madison (22)
- **Quincy (23)**
- Adams (∞)
- LaSalle (∞)
- Library (∞)



Extract min (Madison)

Queue:

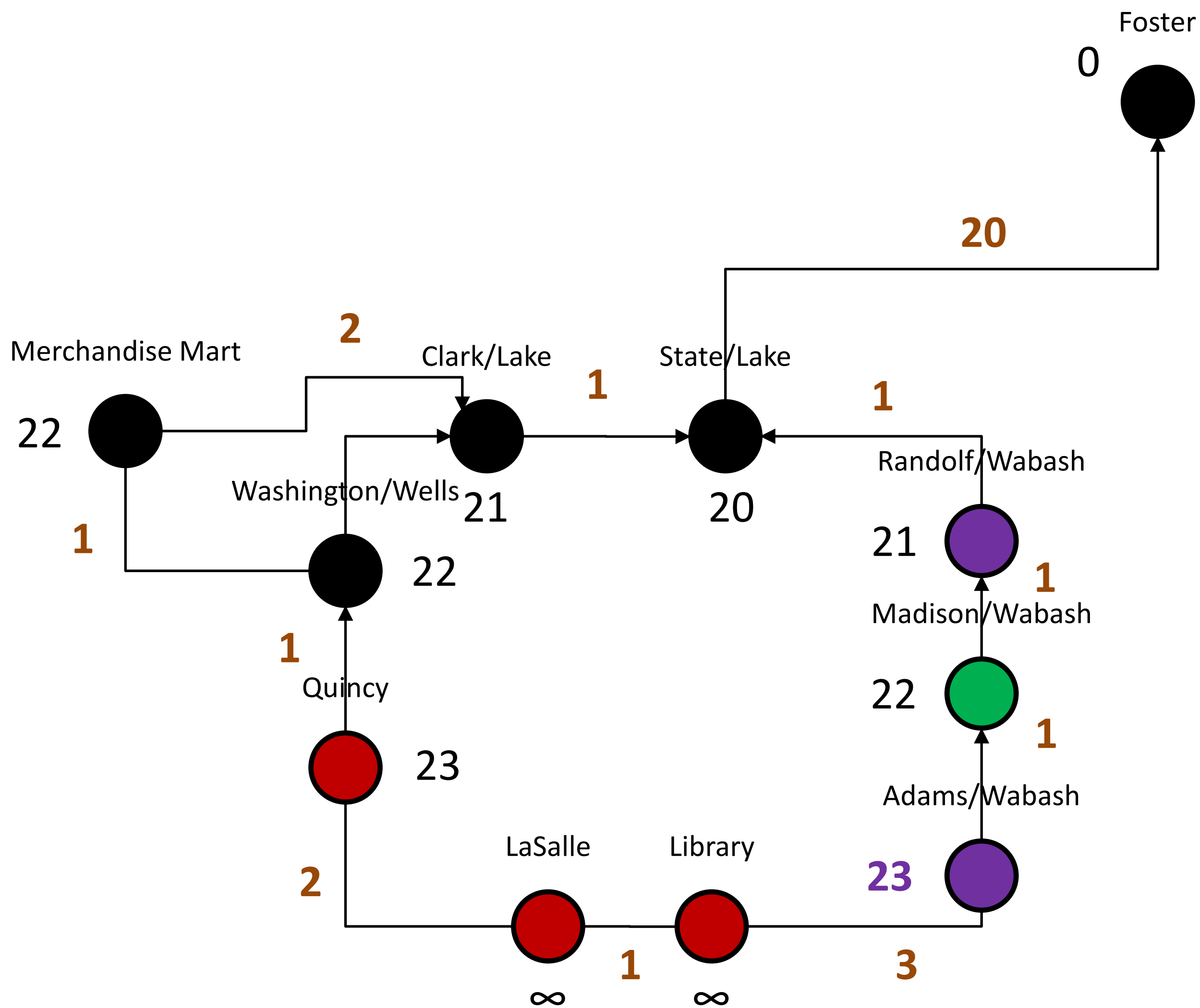
- Quincy (23)
- Adams (∞)
- LaSalle (∞)
- Library (∞)



Update neighbors

Queue:

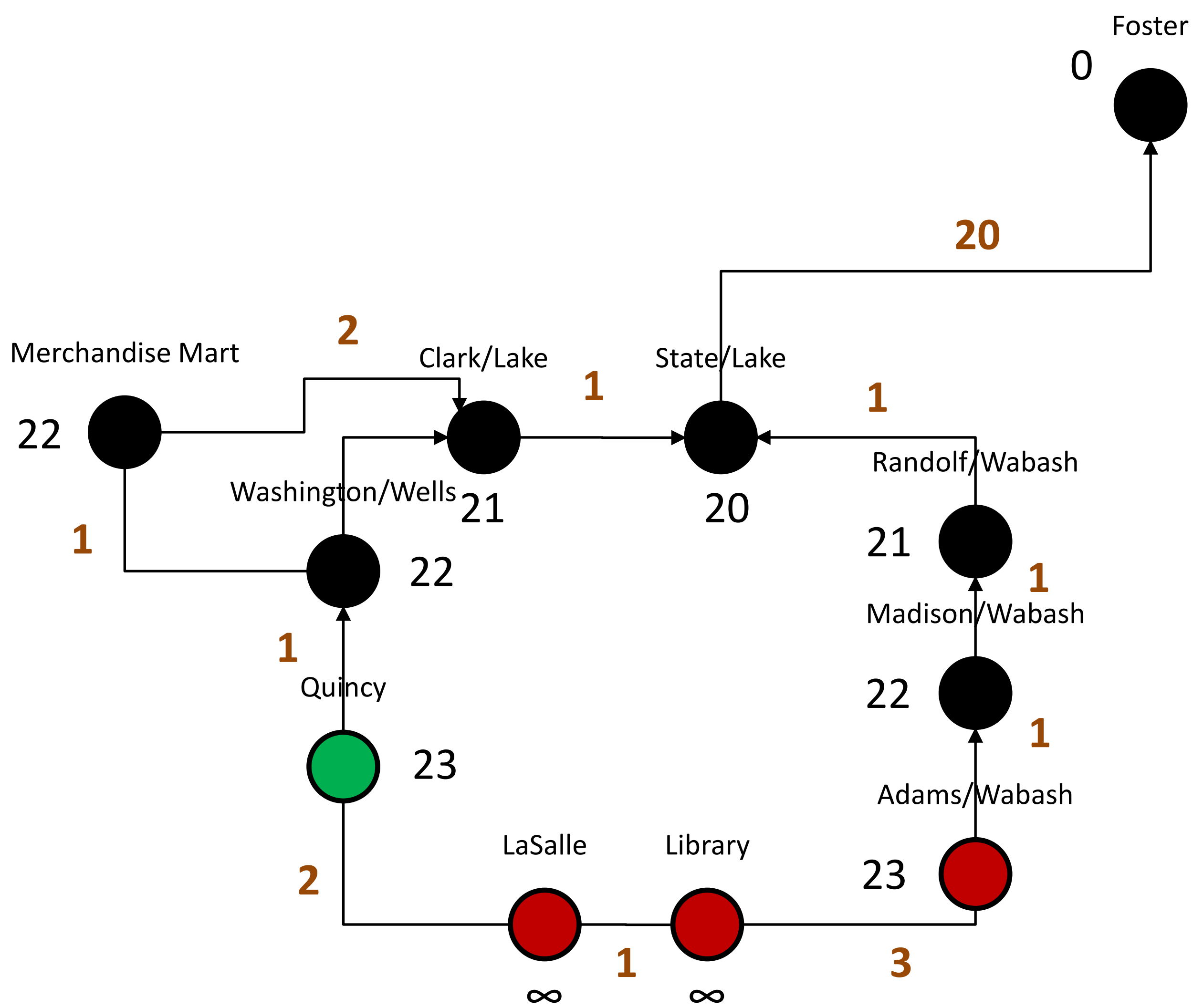
- Quincy (23)
- **Adams (23)**
- LaSalle (∞)
- Library (∞)



Extract min (Quincy)

Queue:

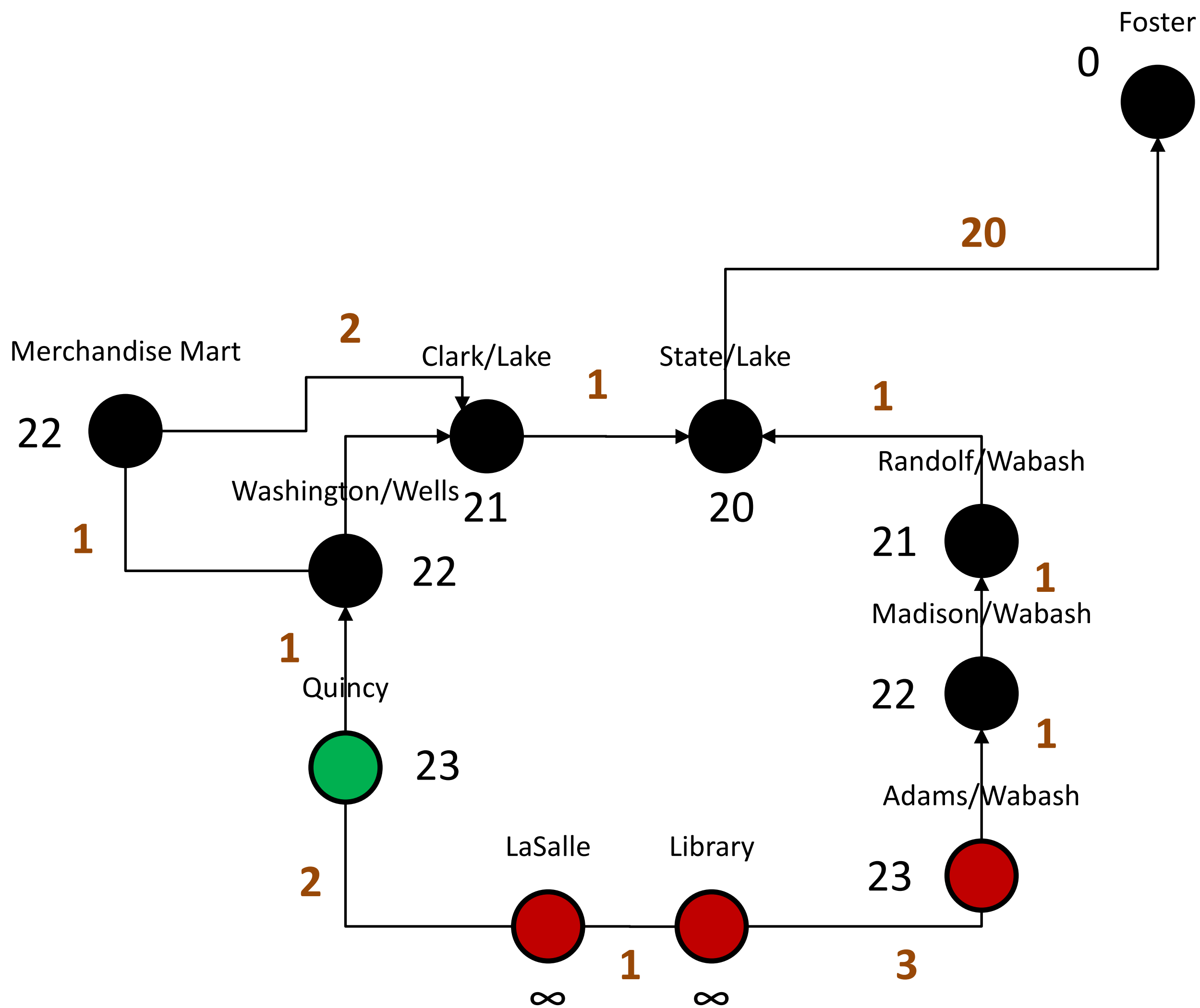
- Adams (23)
- LaSalle (∞)
- Library (∞)



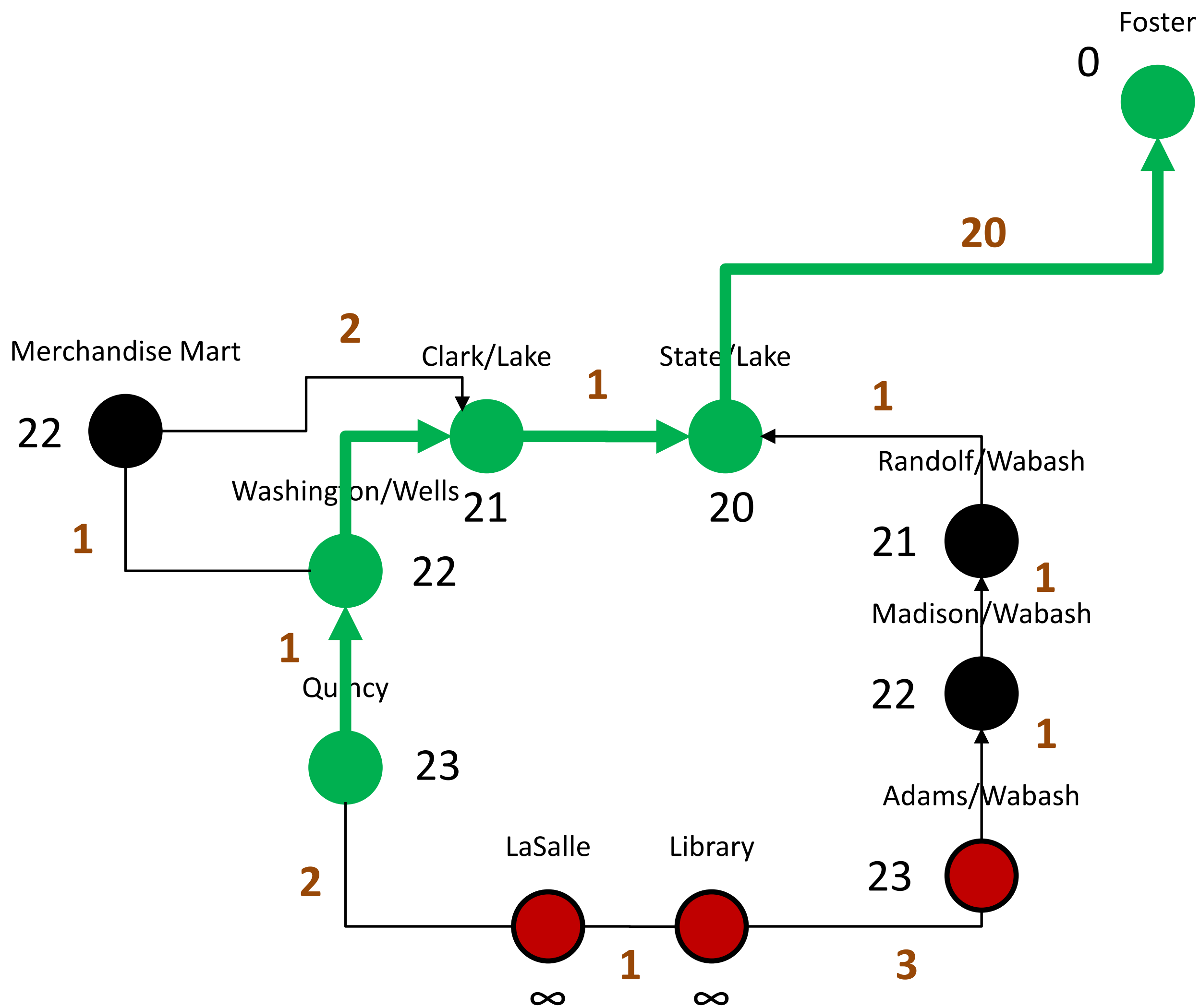
Done!

Queue:

- Adams (23)
- LaSalle (∞)
- Library (∞)



And we have our minimum cost path



How long does it take?

	Runs
Dijkstra(G, s, e)	
PQ = new priority queue	Once
Set all node costs to infinity	Once
s.cost = 0	Once
for each node n in G	
PQ.Insert(n, n.cost)	$O(V)$ times
while PQ not empty	
u = PQ.ExtractMin()	$O(V)$ times
if u == end then done!	
for each neighbor v of u	
w = weight of edge from u to v	$O(E)$ times
newCost = u.cost+v	
if newCost<v.cost	
PQ.DecreaseKey(v, newCost)	
v.cost = newCost	
v.predecessor = u	

How long does it take?

	Time per execution
Dijkstra(G, s, e)	
PQ = new priority queue	$O(1)$
Set all node costs to infinity	$O(V)$
s.cost = 0	$O(1)$
for each node n in G	
PQ.Insert(n, n.cost)	$O(\log V)$
while PQ not empty	
u = PQ.ExtractMin()	$O(\log V)$
if u == end then done!	$O(1)$
for each neighbor v of u	
w = weight of edge from u to v	$O(1)$
newCost = u.cost+v	
if newCost<v.cost	
PQ.DecreaseKey(v, newCost)	$O(\log V)$
v.cost = newCost	$O(1)$
v.predecessor = u	

How long does it take?

Dijkstra(G, s, e)	Runs	Time	Total
PQ = new priority queue	Once	$O(1)$	$O(1)$
Set all node costs to infinity	Once	$O(V)$	$O(V)$
s.cost = 0	Once	$O(1)$	$O(1)$
for each node n in G			
PQ.Insert(n, n.cost)	$O(V)$ times	$O(\log V)$	$O(\mathrm{V}\log V)$
while PQ not empty			
u = PQ.ExtractMin()	$O(V)$ times	$O(\log V)$	$O(V \log V)$
if u == end then done!		$O(1)$	$O(V)$
for each neighbor v of u			
w = weight of edge from u to v	$O(E)$ times	$O(1)$	$O(E)$
newCost = u.cost+v			
if newCost<v.cost			
PQ.DecreaseKey(v, newCost)		$O(\log V)$	$O(\mathrm{E}\log V)$
v.cost = newCost		$O(1)$	$O(E)$
v.predecessor = u			

How long does it take?

$$
\begin{aligned}
&O(1) + O(V) + O(E) \\
&\qquad + O(V \log V) + O(E \log V) \\
&= O(1) + O(V + E) + O((V + E) \log V) \\
&= O(V + E) + O((V + E) \log V) \\
&= O\big((V + E)(1 + \log V)\big) \\
&= O((V + E) \log V)
\end{aligned}
$$