

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu  
208 Capen Hall

**Lec 24: Shortest Path (revisited)**

# Announcements

- PA2 AutoLab is now up – due Sunday

# Heapify

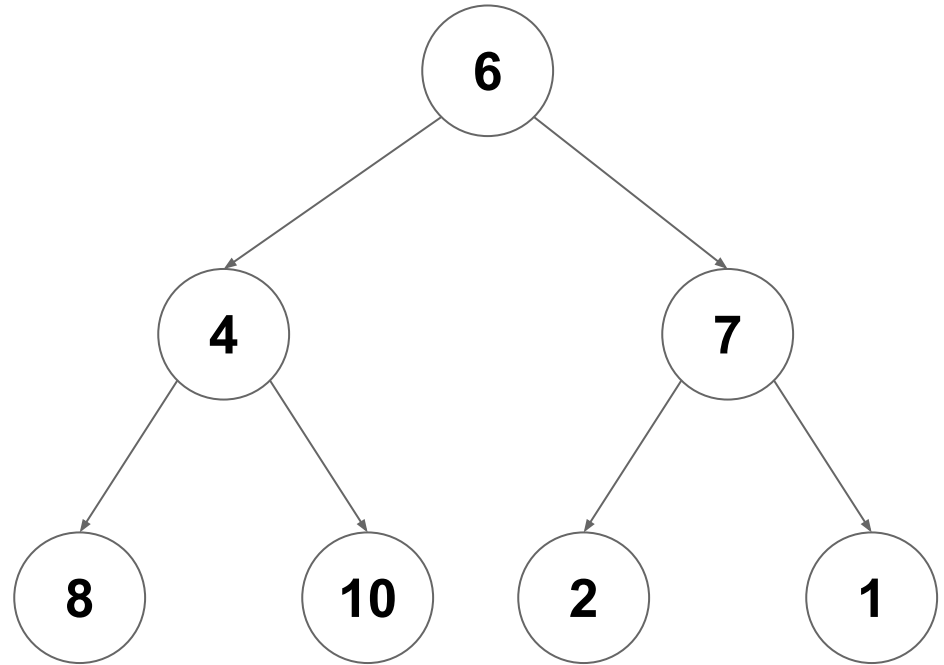
**Input:** Array

**Output:** Array re-ordered to be a heap

**Idea:** `fixUp` or `fixDown` all  $n$  elements in the array

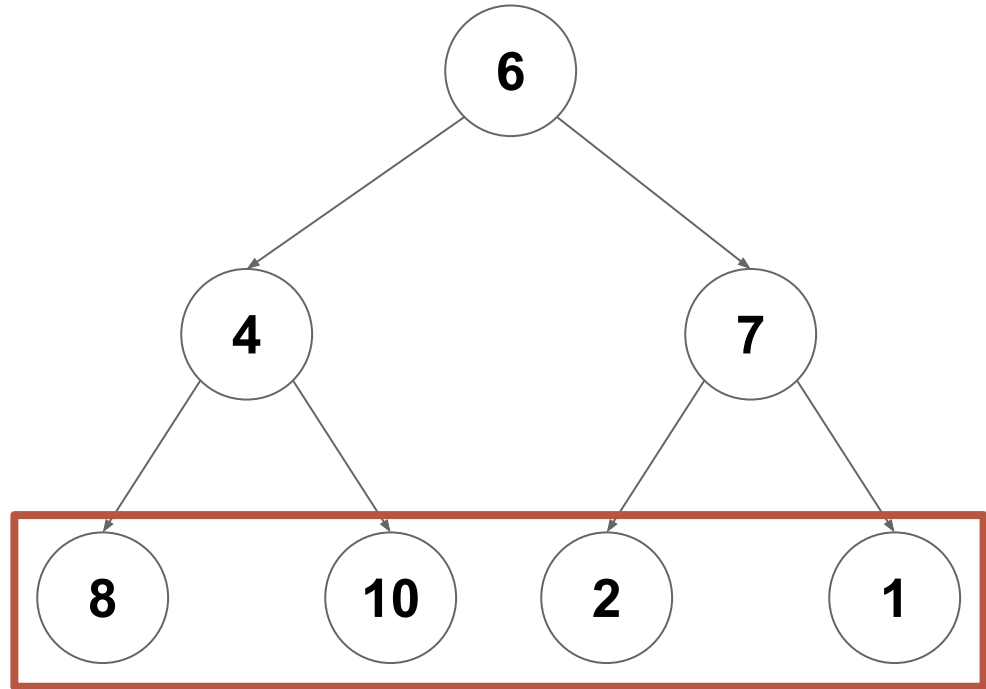
# Heapify

Given an arbitrary array  
(shown as a tree here)  
turn it into a heap



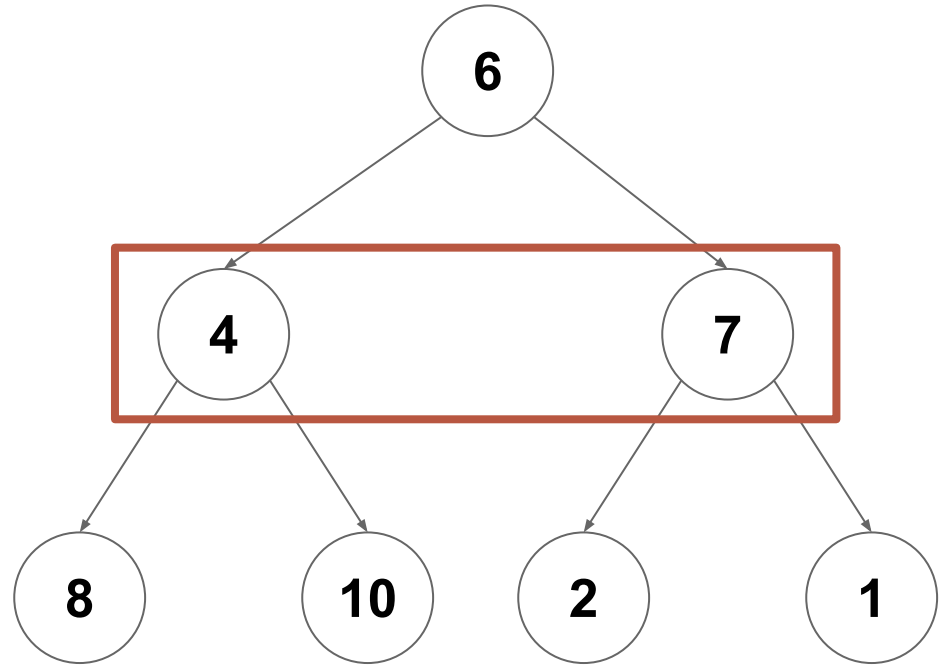
# Heapify

Start at the lowest level,  
and call **fixDown** on each  
node (0 swaps per node)



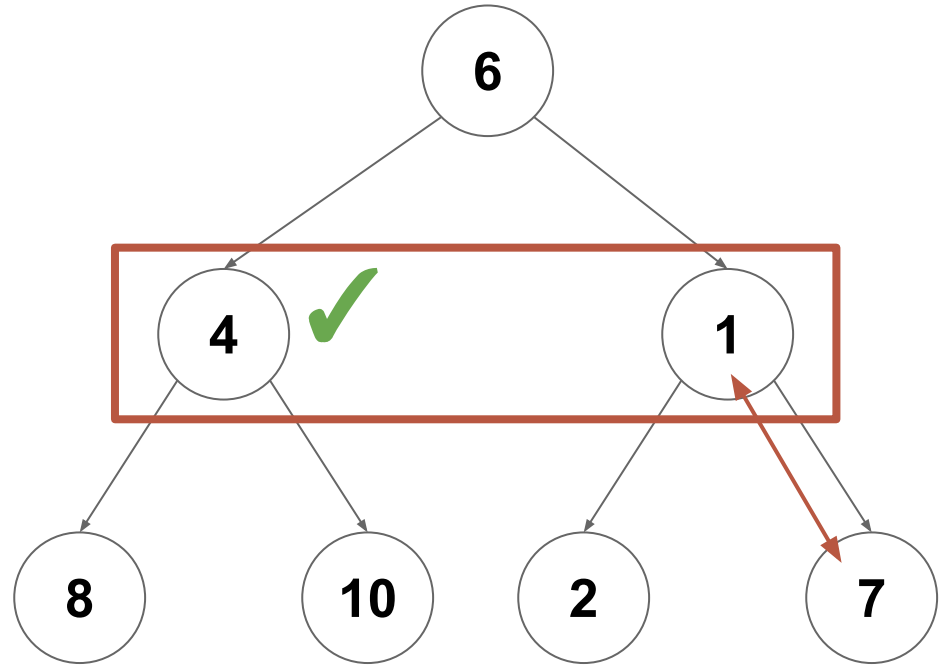
# Heapify

Do the same at the next lowest level (at most one swap per node)



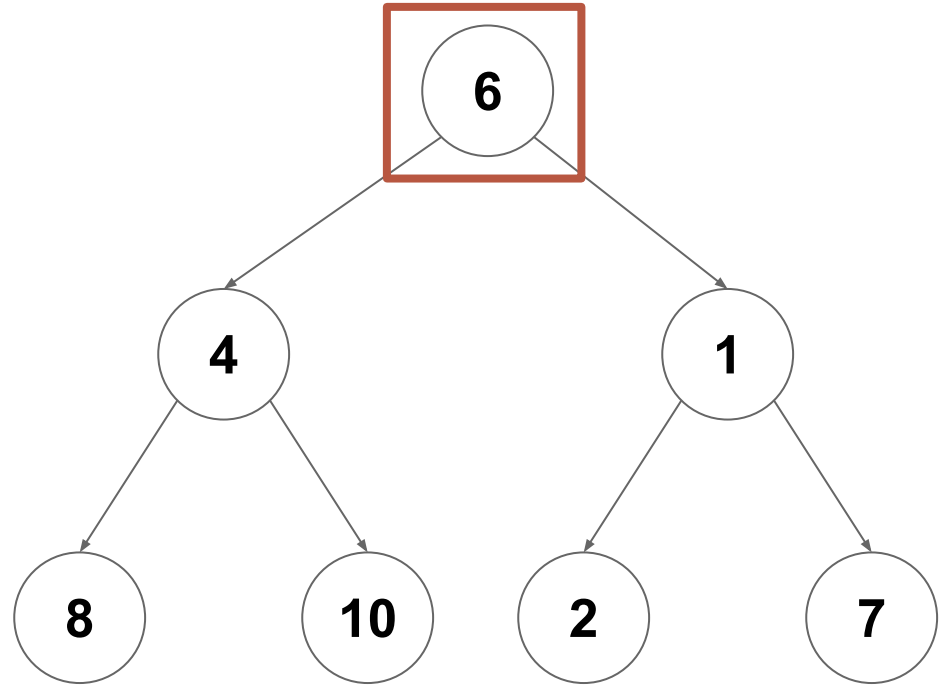
# Heapify

Do the same at the next lowest level (at most one swap per node)



# Heapify

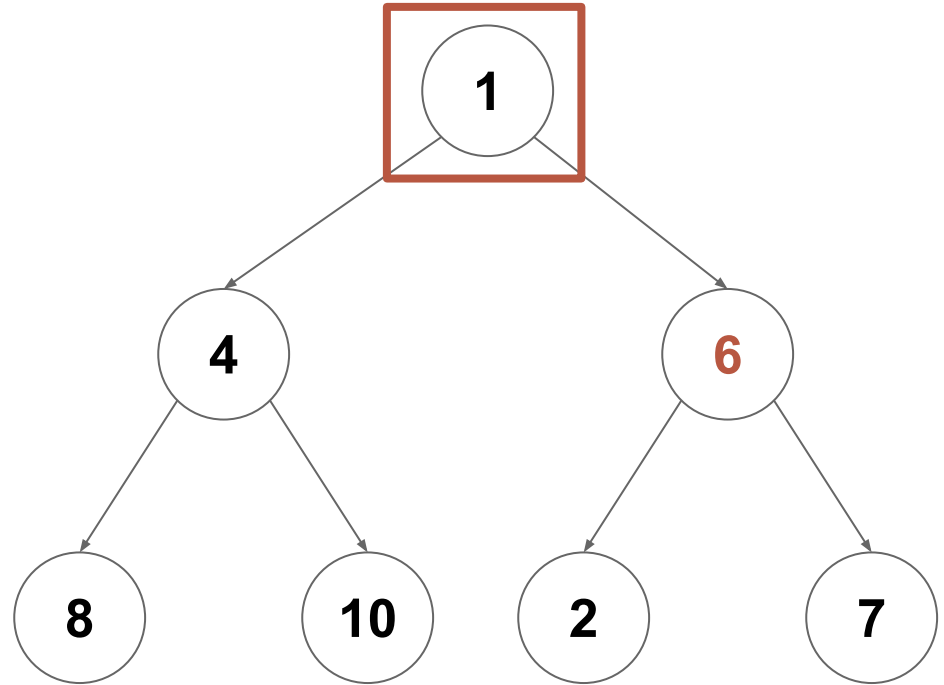
Continue upwards (now at most 2 swaps per node)





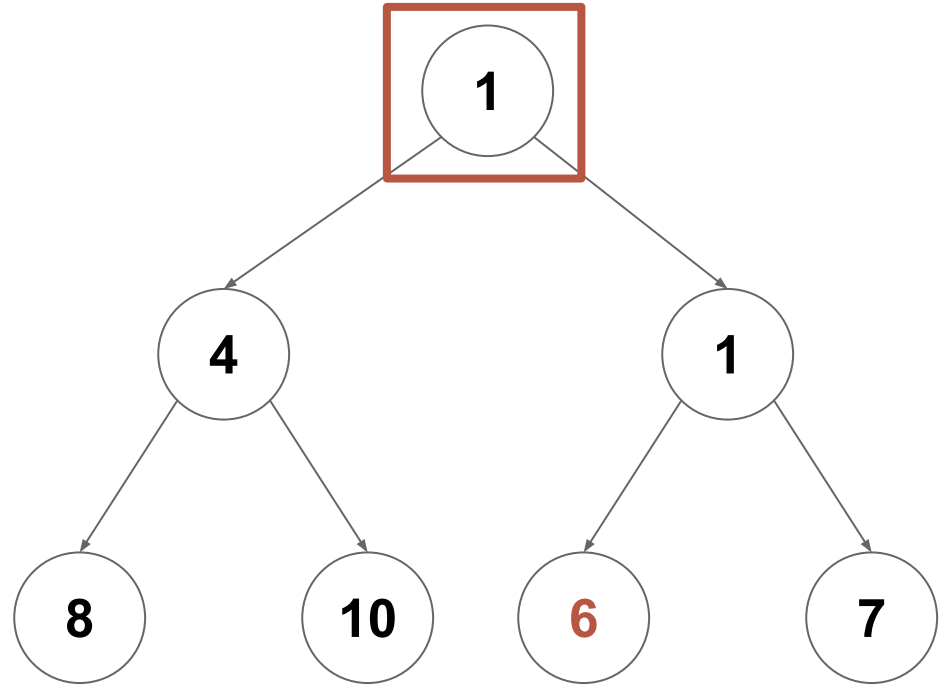
# Heapify

Continue upwards (now at most 2 swaps per node)



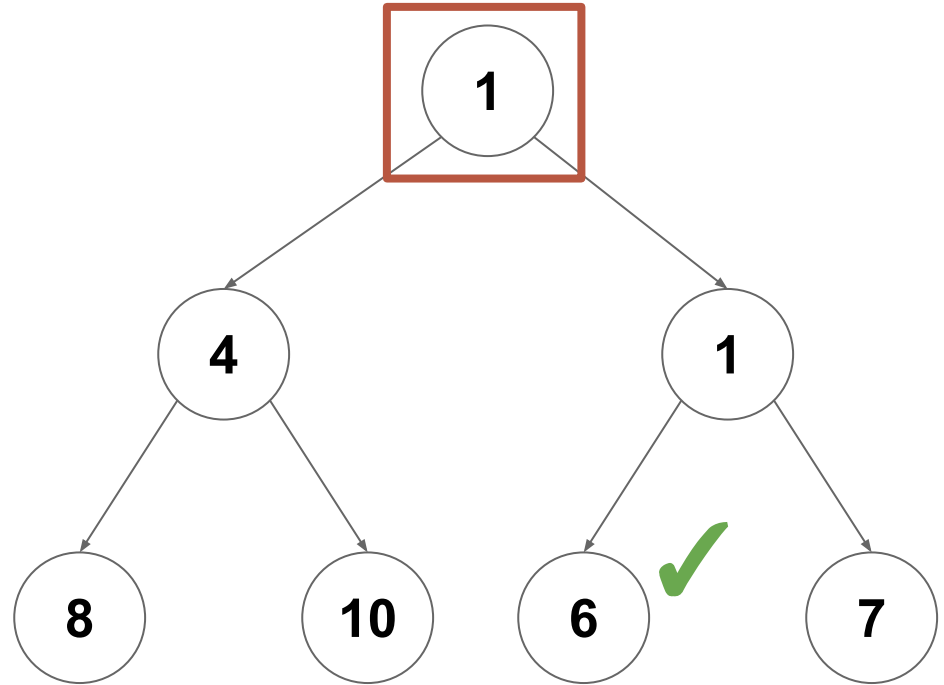
# Heapify

Continue upwards (now at most 2 swaps per node)



# Heapify

Continue upwards (now at most 2 swaps per node)



# Heapify

# Heapify

**At level  $\log(n)$ :** Call **fixDown** on all  $n/2$  nodes at this level (max 0 swaps each)

# Heapify

**At level  $\log(n)$ :** Call **fixDown** on all  $n/2$  nodes at this level (max 0 swaps each)

**At level  $\log(n)-1$ :** Call **fixDown** on all  $n/4$  nodes at this level (max 1 swaps each)

# Heapify

**At level  $\log(n)$ :** Call **fixDown** on all  $n/2$  nodes at this level (max 0 swaps each)

**At level  $\log(n)-1$ :** Call **fixDown** on all  $n/4$  nodes at this level (max 1 swaps each)

**At level  $\log(n)-2$ :** Call **fixDown** on all  $n/8$  nodes at this level (max 2 swaps each)

# Heapify

**At level  $\log(n)$ :** Call **fixDown** on all  $n/2$  nodes at this level (max 0 swaps each)

**At level  $\log(n)-1$ :** Call **fixDown** on all  $n/4$  nodes at this level (max 1 swaps each)

**At level  $\log(n)-2$ :** Call **fixDown** on all  $n/8$  nodes at this level (max 2 swaps each)

...

**At level 1:** Call **fixDown** on all 1 nodes at this level (max  $\log(n)$  swaps each)



# Heapify

Sum the number of swaps  
required by each level

$$O \left( \sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i + 1) \right)$$

# Heapify

Pull out the  $n$  as a constant and distribute multiplication

$$O \left( \sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i + 1) \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i} \right)$$

# Heapify

Focus on the dominant term only

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

# Heapify

Change  $\log(n)$  to infinity  
(can only increase  
complexity class if  
anything)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i}\right)$$

# Heapify

We can now treat the sum as a constant

$$O \left( \sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i + 1) \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\infty} \frac{i}{2^i} \right)$$

This is known to converge to a constant

# Heapify

Therefore we can heapify  
an array of size  $n$  in  $O(n)$

$$O \left( \sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i + 1) \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\infty} \frac{i}{2^i} \right) = O(n)$$

# Heapify

Therefore we can heapify  
an array of size  $n$  in  $O(n)$

(but heap sort still  
requires  $n \log(n)$  due to  
dequeue costs)

$$O \left( \sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i + 1) \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\log(n)} \frac{i}{2^i} \right)$$

$$O \left( n \sum_{i=1}^{\infty} \frac{i}{2^i} \right) = O(n)$$

# Heapify

Consider the time required to add  $n$  items to a heap:

- Each add takes  $O(\log(n))$
- In total,  $n$  adds will take  $O(n \log(n))$



# Heapify

**Consider the time required to add  $n$  items to a heap:**

- Each add takes  $O(\log(n))$
- In total,  $n$  adds will take  $O(n \log(n))$

**Now, consider the time required to turn  $n$  items into a heap with heapify:**

- The total cost of heapify is  $O(n)$

# Heapify

**Consider the time required to add  $n$  items to a heap:**

- Each add takes  $O(\log(n))$
- In total,  $n$  adds will take  $O(n \log(n))$

**Now, consider the time required to turn  $n$  items into a heap with heapify:**

- The total cost of heapify is  $O(n)$

**Often we can save time by performing a task in one big batch on all of the data, rather than handling each element one at a time**

# Another Example

What is the cost to add  $n$  items to a sorted list, one at a time?

# Another Example

**What is the cost to add  $n$  items to a sorted list, one at a time?**

- Each item requires  $O(n)$  to add
- Adding all  $n$  items requires  $O(n^2)$  total

# Another Example

**What is the cost to add  $n$  items to a sorted list, one at a time?**

- Each item requires  $O(n)$  to add
- Adding all  $n$  items requires  $O(n^2)$  total

**What is the cost to sort  $n$  items in a list?**

# Another Example

**What is the cost to add  $n$  items to a sorted list, one at a time?**

- Each item requires  $O(n)$  to add
- Adding all  $n$  items requires  $O(n^2)$  total

**What is the cost to sort  $n$  items in a list?**

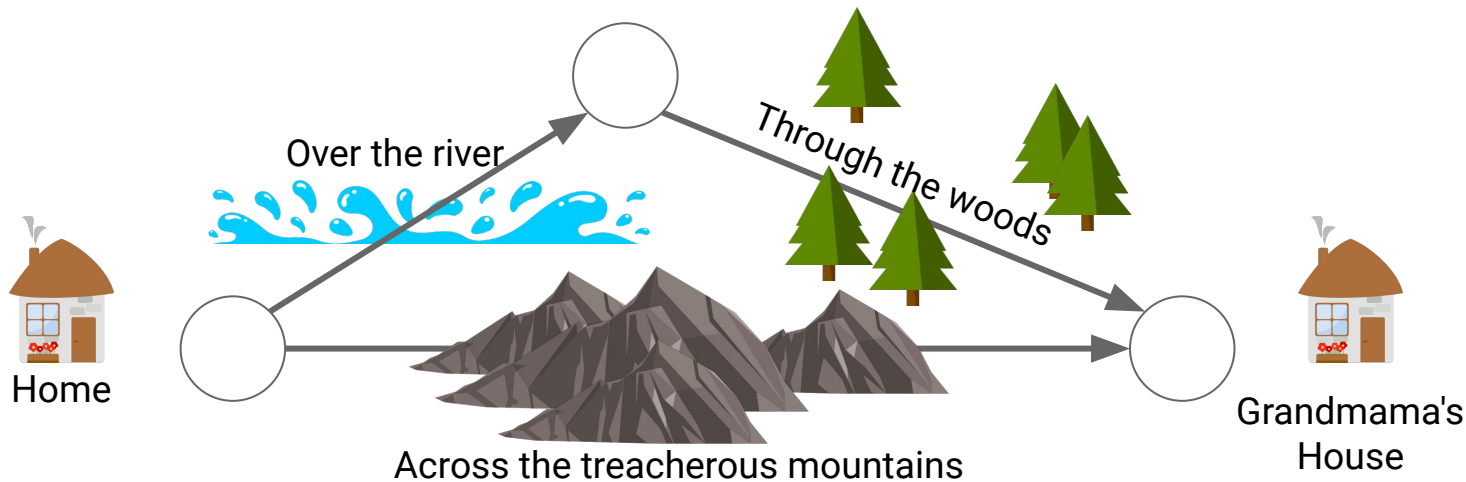
- Using Merge Sort or Heap Sort, it would take  $O(n \log(n))$

# Heaps as Priority Queues

We now have an efficient implementation of the **PriorityQueue** ADT

- Java's **PriorityQueue** implementation uses a heap as well
- By default, it's a min heap, but can use a custom comparator as well
- Now we have what we need to revisit the shortest path problem

# Shortest Paths



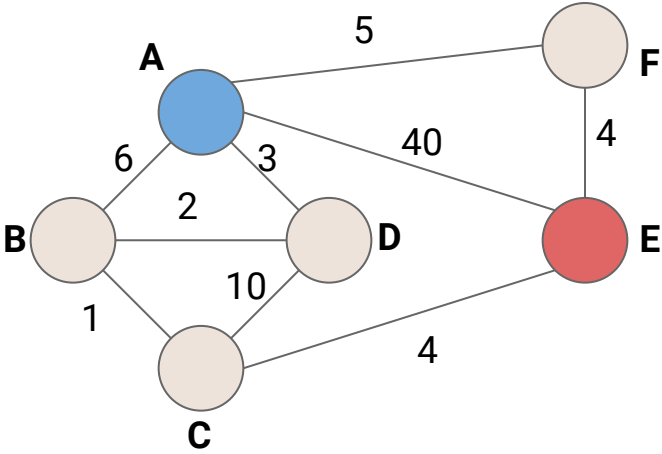
BFS will always find the path with the **fewest edges**...

Not all edges in a real world graph are necessarily created equal!

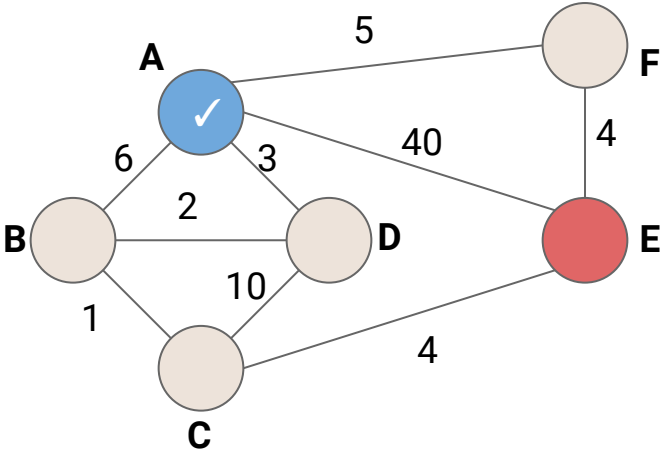
*Which path is actually the best/shortest?*



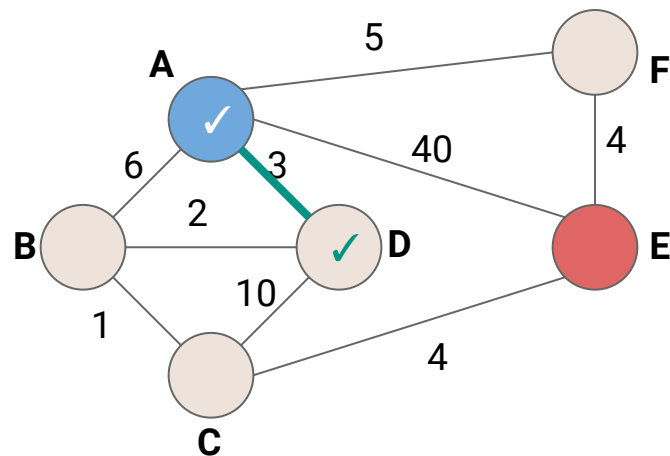
# Desired Exploration Order - Closest Vertex










# Desired Exploration Order - Closest Vertex

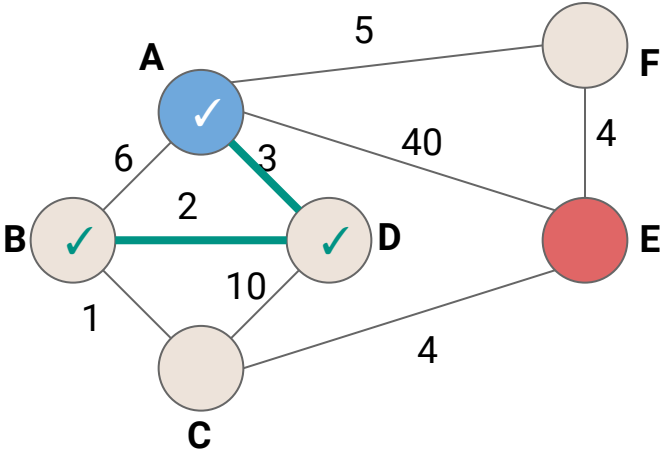


# Desired Exploration Order - Closest Vertex

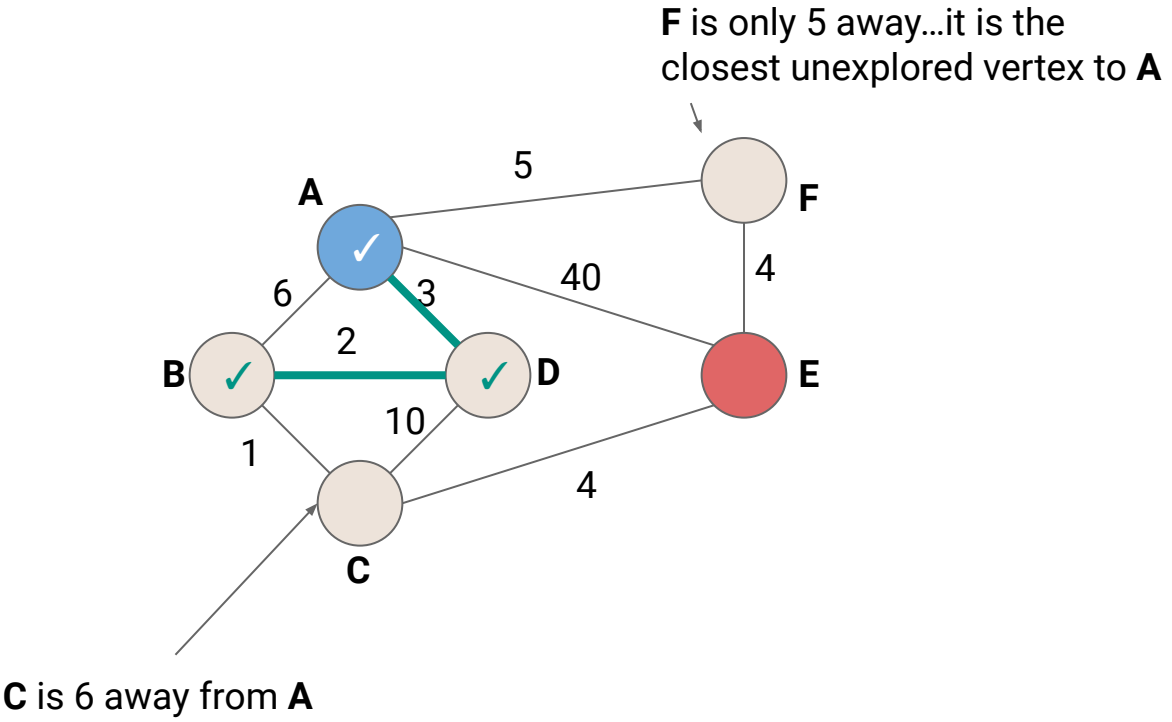
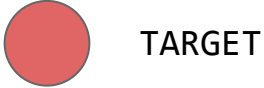


# Desired Exploration Order - Closest Vertex








-  UNEXPLORED
-  START
-  TARGET
-  VISITED
-  UNEXPLORED
-  SPANNING
-  CROSS

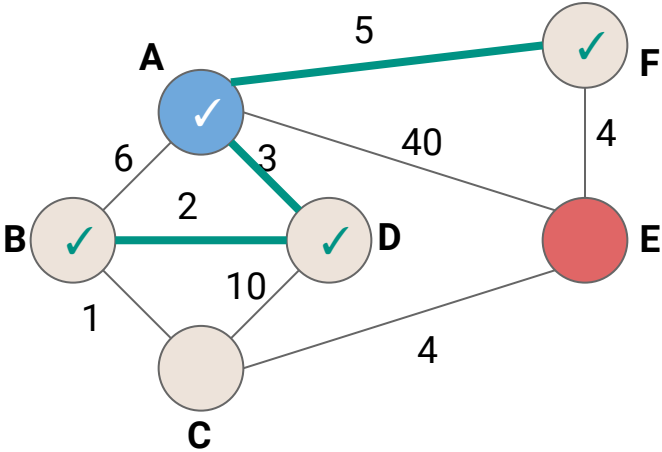


# Desired Exploration Order - Closest Vertex

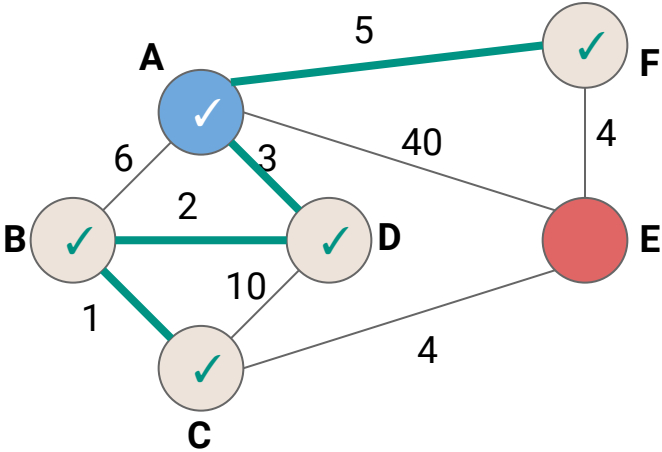
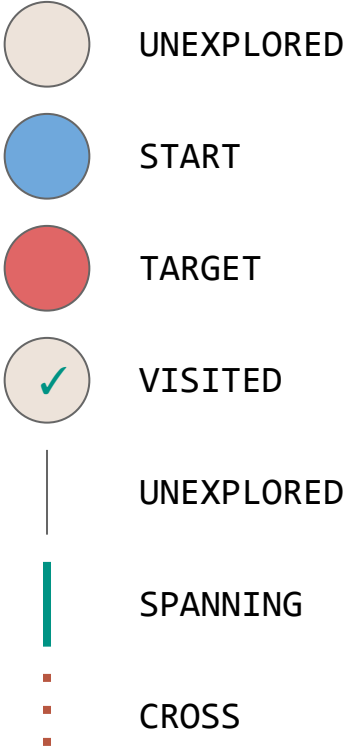


# Desired Exploration Order - Closest Vertex

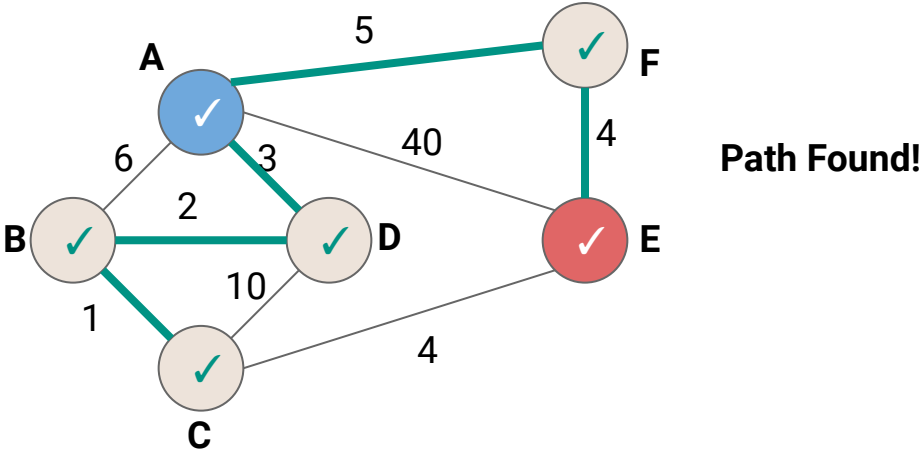
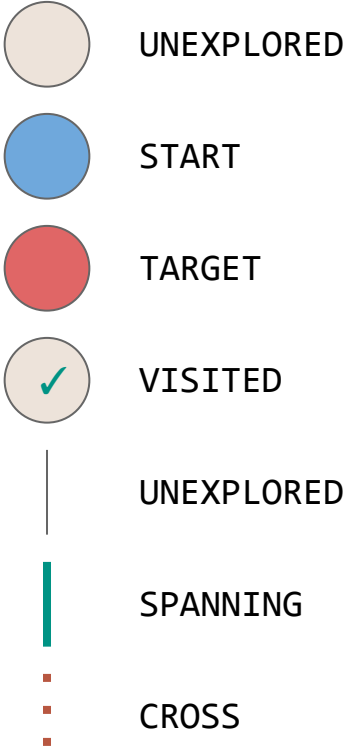
-  UNEXPLORED
-  START
-  TARGET
-  VISITED
-  UNEXPLORED
-  SPANNING
-  CROSS



# Desired Exploration Order - Closest Vertex



# Desired Exploration Order - Closest Vertex





```
1 public void BFS(Graph graph, Vertex v) {
2     Queue<TodoEntry> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(new TodoEntry(v,0));
5     while (!todo.isEmpty()) {
6         TodoEntry curr = todo.dequeue();
7         for (Edge e : curr.vertex.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
13                }
14            }
15        }
16    }
17 }
```

We want to be able to dequeue  
in order of weight...but how?

```
1 public void ShortestPath(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     v.setLabel(VISITED);
4     todo.add(new TodoEntry(v, 0));
5     while (!todo.isEmpty()) {
6         TodoEntry curr = todo.poll();
7         for (Edge e : curr.vertex.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    todo.add(new TodoEntry(w, curr.weight + e.weight));
13                }
14            }
15        }
16    }
17 }
```

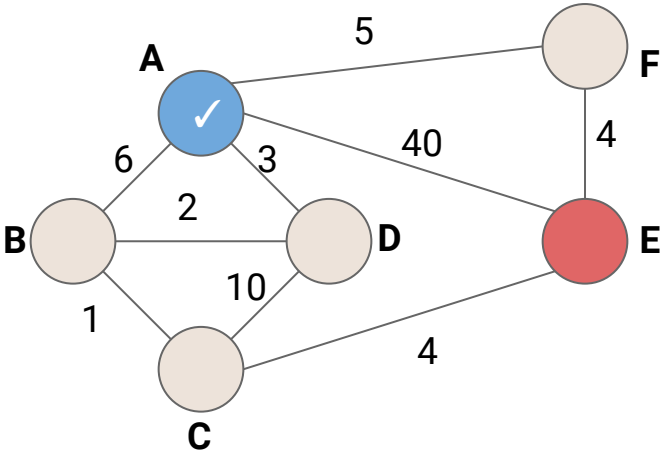
Use a PriorityQueue (with lower weights having high priority)

Is this enough?

# PriorityQueue Attempt #1



PriorityQueue  
(A, 0)



# PriorityQueue Attempt #1



UNEXPLORED



START



TARGET



VISITED



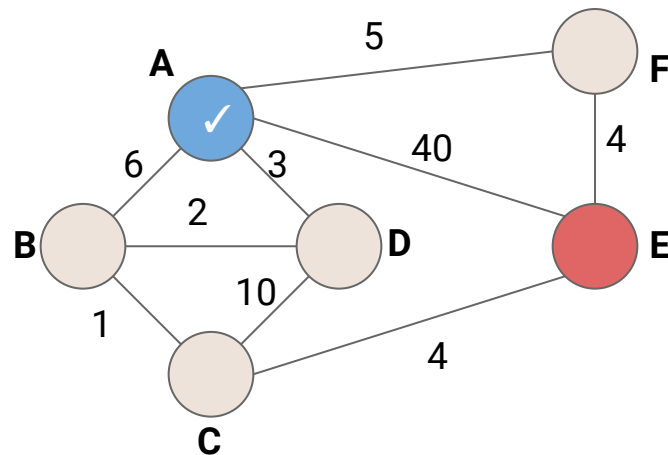
UNEXPLORED



SPANNING



CROSS



PriorityQueue  
(A, 0)

Remove (A, 0) from the PriorityQueue...

# PriorityQueue Attempt #1



UNEXPLORED



START



TARGET



VISITED



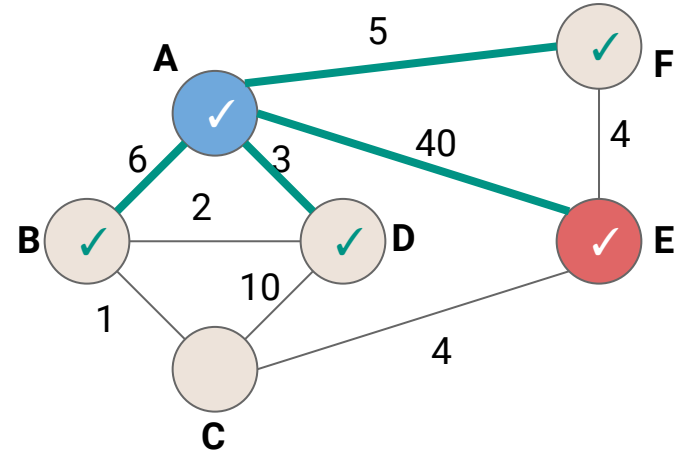
UNEXPLORED



SPANNING



CROSS



## PriorityQueue

(A,0)

(B,6)

(D,3)

(F,5)

(E,40)

Remove (A, 0) from the PriorityQueue...  
...and add it's neighbors

# PriorityQueue Attempt #1



UNEXPLORED



START



TARGET



VISITED



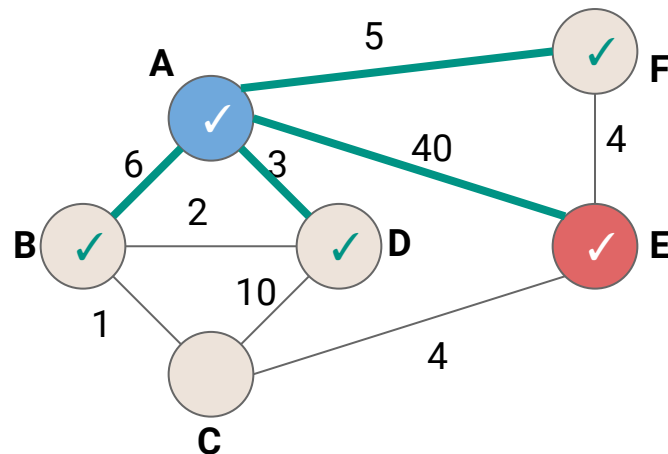
UNEXPLORED



SPANNING



CROSS



## PriorityQueue

(A,0)

(B,6)

(D,3)

(F,5)

(E,40)

We've just marked E as visited! Have we found the shortest path to E?

# PriorityQueue Attempt #1



UNEXPLORED



START



TARGET



VISITED



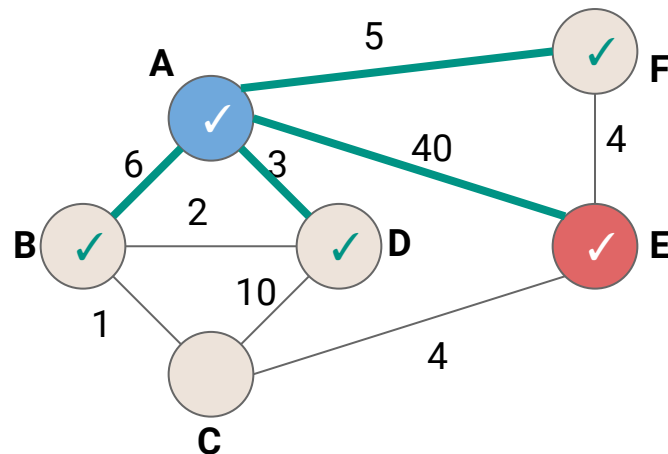
UNEXPLORED



SPANNING



CROSS



## PriorityQueue

(A,0)

(B,6)

(D,3)

(F,5)


(E,40)

We've just marked E as visited! Have we found the shortest path to E?

When should we consider something VISITED?

```
1 public void ShortestPath(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     v.setLabel(VISITED);
4     todo.add(new TodoEntry(v,0));
5     while (!todo.isEmpty()) {
6         TodoEntry curr = todo.poll();
7         for (Edge e : curr.vertex.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    todo.add(new TodoEntry(w, curr.weight + e.weight));
13                }
14            }
15        }
16    }
17 }
```

When we add **w** to the **PriorityQueue**, there still may be other shorter paths, so we can't consider **w** VISITED yet





```
1 public void ShortestPath(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     v.setLabel(VISITED);
4     todo.add(new TodoEntry(v,0));
5     while (!todo.isEmpty()) {
6         TodoEntry curr = todo.poll();
7         for (Edge e : curr.vertex.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    todo.add(new TodoEntry(w, curr.weight + e.weight));
13                }
14            }
15        }
16    }
17 }
```

When we add **w** to the **PriorityQueue**, there still may be other shorter paths, so we can't consider **w** **VISITED** yet

**When can we consider a vertex visited?**

**In other words, when do we know we've found the shortest path to a vertex?**

```
1 public void ShortestPath(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     v.setLabel(VISITED);
4     todo.add(new TodoEntry(v, 0));
5     while (!todo.isEmpty()) {
6         TodoEntry curr = todo.poll();
7         curr.vertex.setLabel(VISITED);
8         for (Edge e : curr.vertex.outEdges) {
9             if (e.label == UNEXPLORED) {
10                Vertex w = e.to;
11                if (w.label == UNEXPLORED) {
12                    todo.add(new TodoEntry(w, curr.weight + e.weight));
13                }
14            }
15        }
16    }
17 }
```

How about when we dequeue?



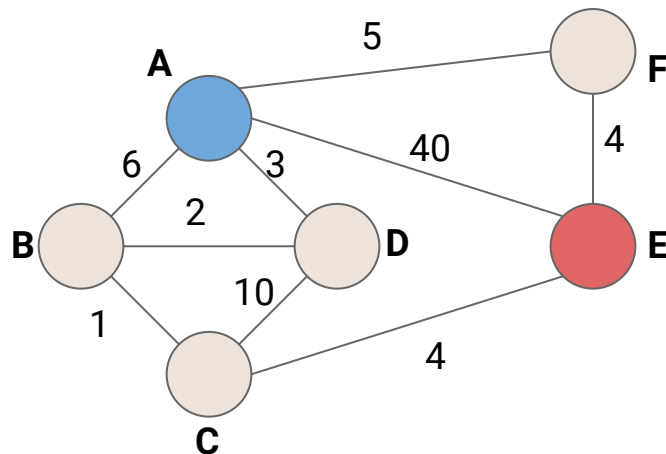
**When can we consider a vertex visited?**

**In other words, when do we know we've found the shortest path to a vertex?**

# PriorityQueue Attempt #2



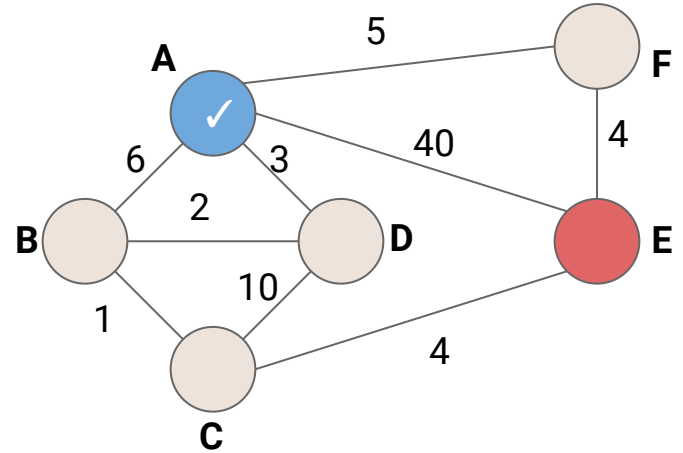
PriorityQueue  
(A, 0)



# PriorityQueue Attempt #2



PriorityQueue  
(A, 0)



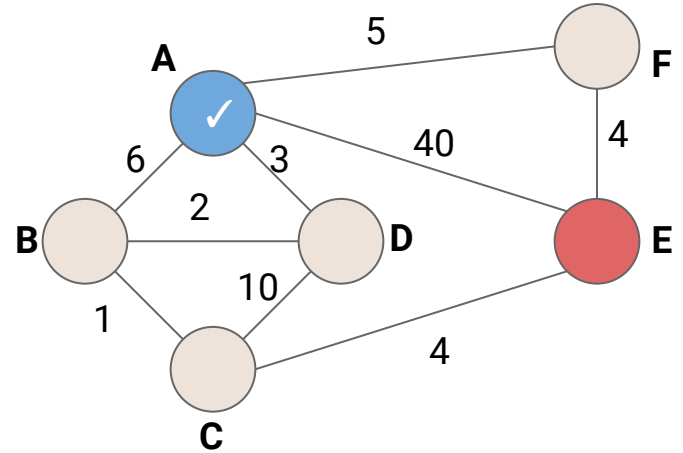
Mark A as VISITED because we just dequeued it. We now know there's no better path to A

# PriorityQueue Attempt #2



## PriorityQueue

~~(A, 0)~~  
(B, 6)  
(D, 3)  
(F, 5)  
(E, 40)



Mark A as VISITED because we just dequeued it. We now know there's no better path to A

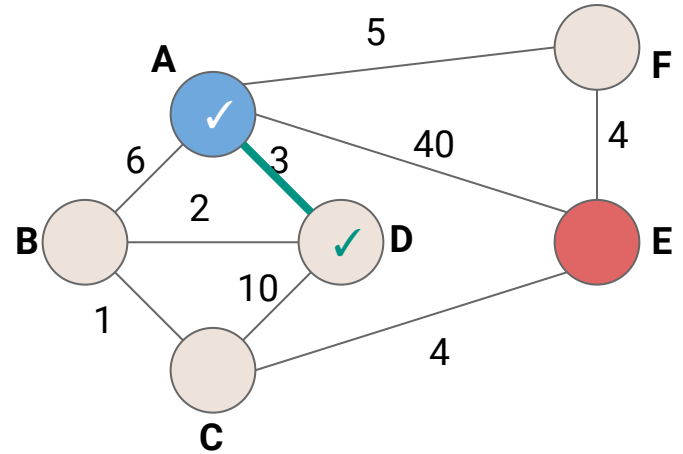
Add the neighbors of A to the Priority Queue, but don't mark as VISITED yet

# PriorityQueue Attempt #2



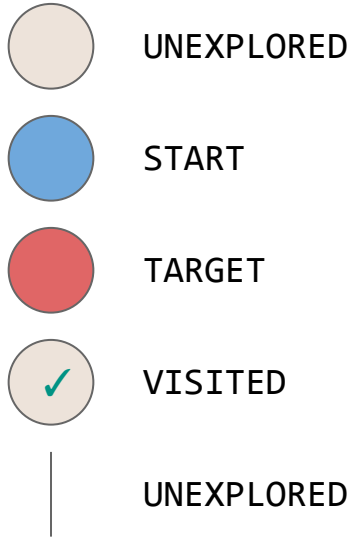
## PriorityQueue

(A, 0)  
(B, 6)  
~~(D, 3)~~  
(F, 5)  
(E, 40)



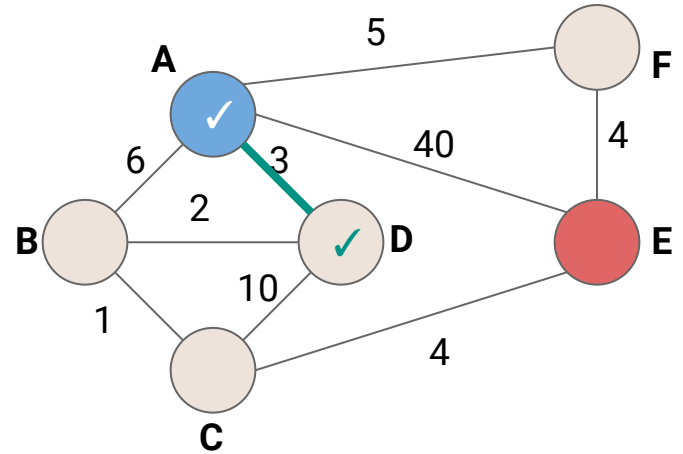
Mark D as visited because we've found the shortest path to D. What should we add to the PQ now?

# PriorityQueue Attempt #2



## PriorityQueue

(A, 0)  
(B, 6)  
~~(D, 3)~~  
(F, 5)  
(E, 40)  
(B, 5)  
(C, 13)



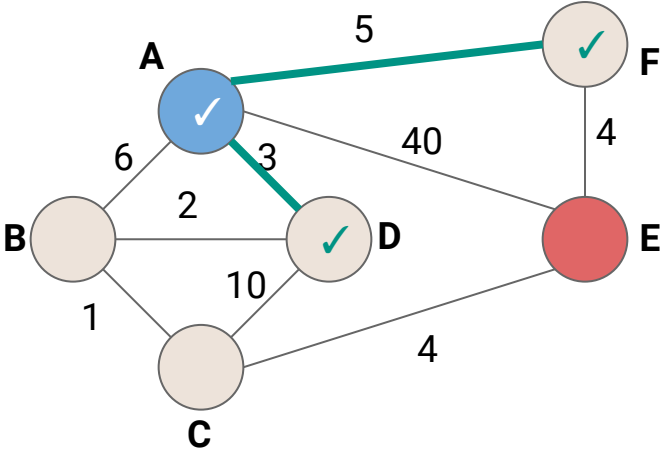
We know now we could get to B in at least 5, and C in at most 13.

# PriorityQueue Attempt #2



### PriorityQueue

- ~~(A, 0)~~
- (B, 6)
- ~~(D, 3)~~
- ~~(F, 5)~~
- (E, 40)
- (B, 5)
- (C, 13)
- (E, 9)



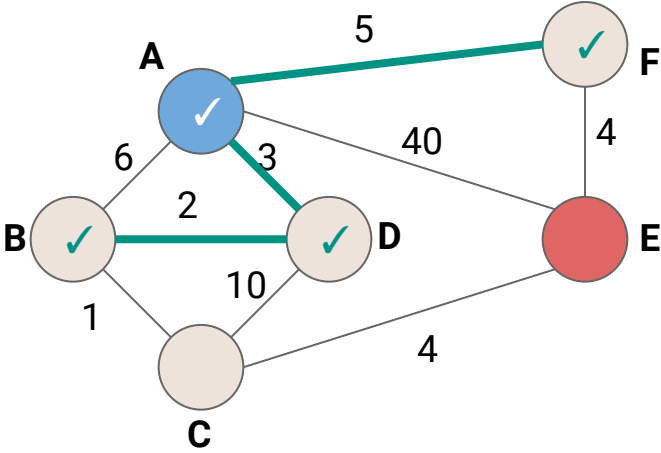


# PriorityQueue Attempt #2



### PriorityQueue

- ~~(A, 0)~~
- (B, 6)
- ~~(D, 3)~~
- ~~(F, 5)~~
- (E, 40)
- ~~(B, 5)~~
- (C, 13)
- (E, 9)
- (C, 6)

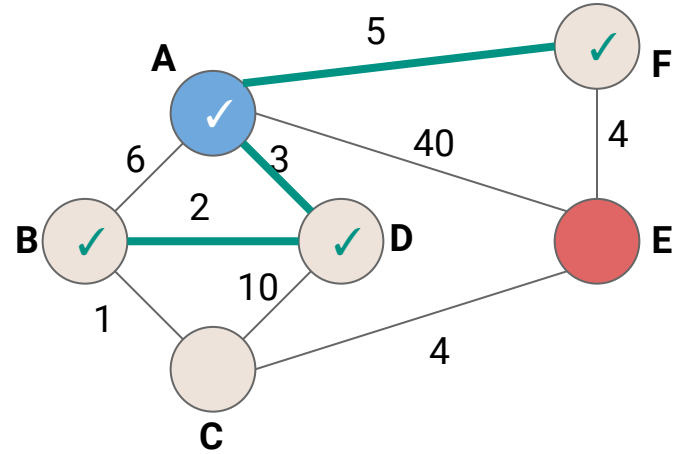


# PriorityQueue Attempt #2



## PriorityQueue

~~(A, 0)~~  
~~(B, 6)~~  
~~(D, 3)~~  
~~(F, 5)~~  
(E, 40)  
~~(B, 5)~~  
(C, 13)  
(E, 9)  
(C, 6)



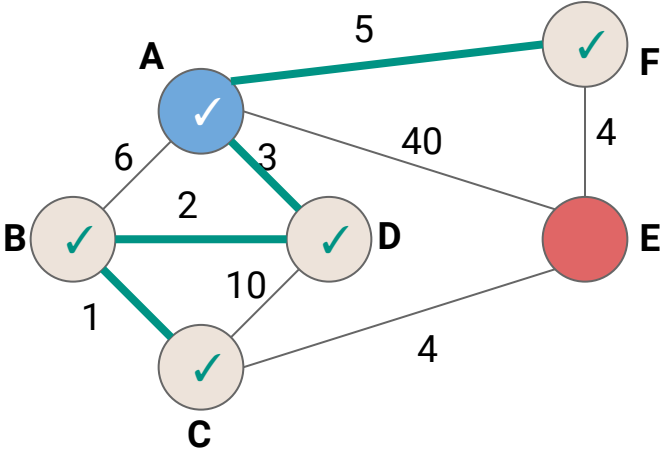
We've already visited B so we can ignore this

# PriorityQueue Attempt #2



### PriorityQueue

- ~~(A, 0)~~
- ~~(B, 6)~~
- ~~(D, 3)~~
- ~~(F, 5)~~
- (E, 40)
- ~~(B, 5)~~
- (C, 13)
- (E, 9)
- ~~(C, 6)~~
- (E, 10)

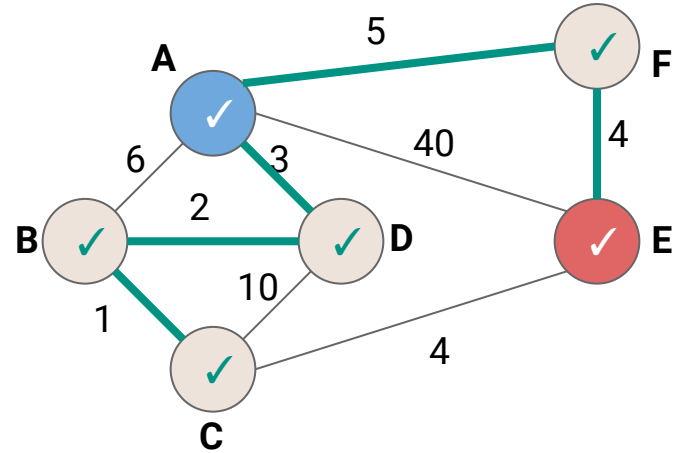


# PriorityQueue Attempt #2



## PriorityQueue

~~(A, 0)~~  
~~(B, 6)~~  
~~(D, 3)~~  
~~(F, 5)~~  
(E, 40)  
~~(B, 5)~~  
(C, 13)  
~~(E, 9)~~  
~~(C, 6)~~  
(E, 10)



We've dequeued E, so we've found the shortest possible path to get there! (Anything else still left in the PriorityQueue is a longer path)

```
1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v,0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }
```

```
1 public void Dijkstra(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v, 0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }
```

Create a new PriorityQueue and insert the starting point with a distance of 0

```
1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v,0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }
```

When we pull something out of the PriorityQueue, if it is still UNEXPLORED then we just found the shortest path to that vertex, and we can mark it as VISITED

```
1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v,0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }
```

Add each unexplored neighbor to the PriorityQueue.  
Set its distance equal to our current distance plus the weight of the edge to get to the neighbor.



```
1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v,0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }
```

What is the complexity?

```

1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v,0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }

```

We know removal from a  
PriorityQueue is  
 $O(\log(\text{todo.size()}))$

How big can **todo** get?

What is the complexity?

```

1 public void Djikstras(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v, 0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }

```

We know removal from a  
PriorityQueue is  
 $O(\log(\text{todo.size()}))$

How big can **todo** get?  $|E|$

Each vertex may be added once per incoming edge. So  
the size of the PriorityQueue can get as large as  $|E|$

```

1 public void Dijkstra(Graph graph, Vertex v) {
2     PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
3     todo.add(new TodoEntry(v, 0));
4     while (!todo.isEmpty()) {
5         TodoEntry curr = todo.poll();
6         if (curr.vertex.label == UNEXPLORED) {
7             curr.vertex.setLabel(VISITED);
8             for (Edge e : curr.vertex.outEdges) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    todo.add(new TodoEntry(w, curr.weight + e.weight));
12                }
13            }
14        }
15    }
16 }

```

We know removal from a  
PriorityQueue is  
 $O(\log(\text{todo.size()}))$

How big can **todo** get?  $|E|$

Label the  $|V|$  vertices     $|E|$  adds/removes to the PriorityQueue

What is the complexity?  $O(|V| + |E| \log(|E|))$

# Dijkstra's Algorithm

- Many tweaks can be made
  - What if instead of enqueueing a vertex we've already seen we just update the existing value in our heap?
  - How can we track the actual path?
    - Build a map of reverse lookups (just like for BFS/DFS)