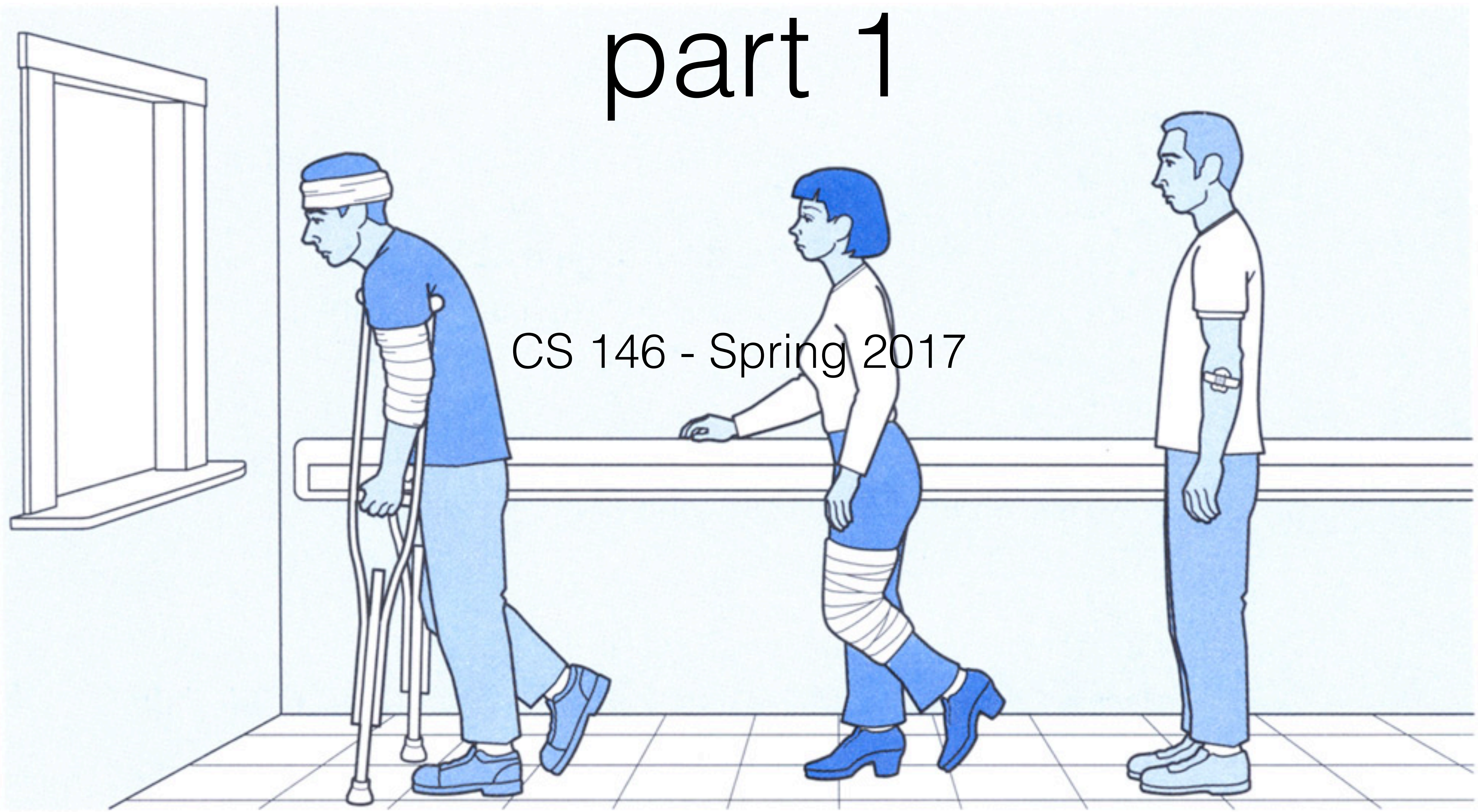
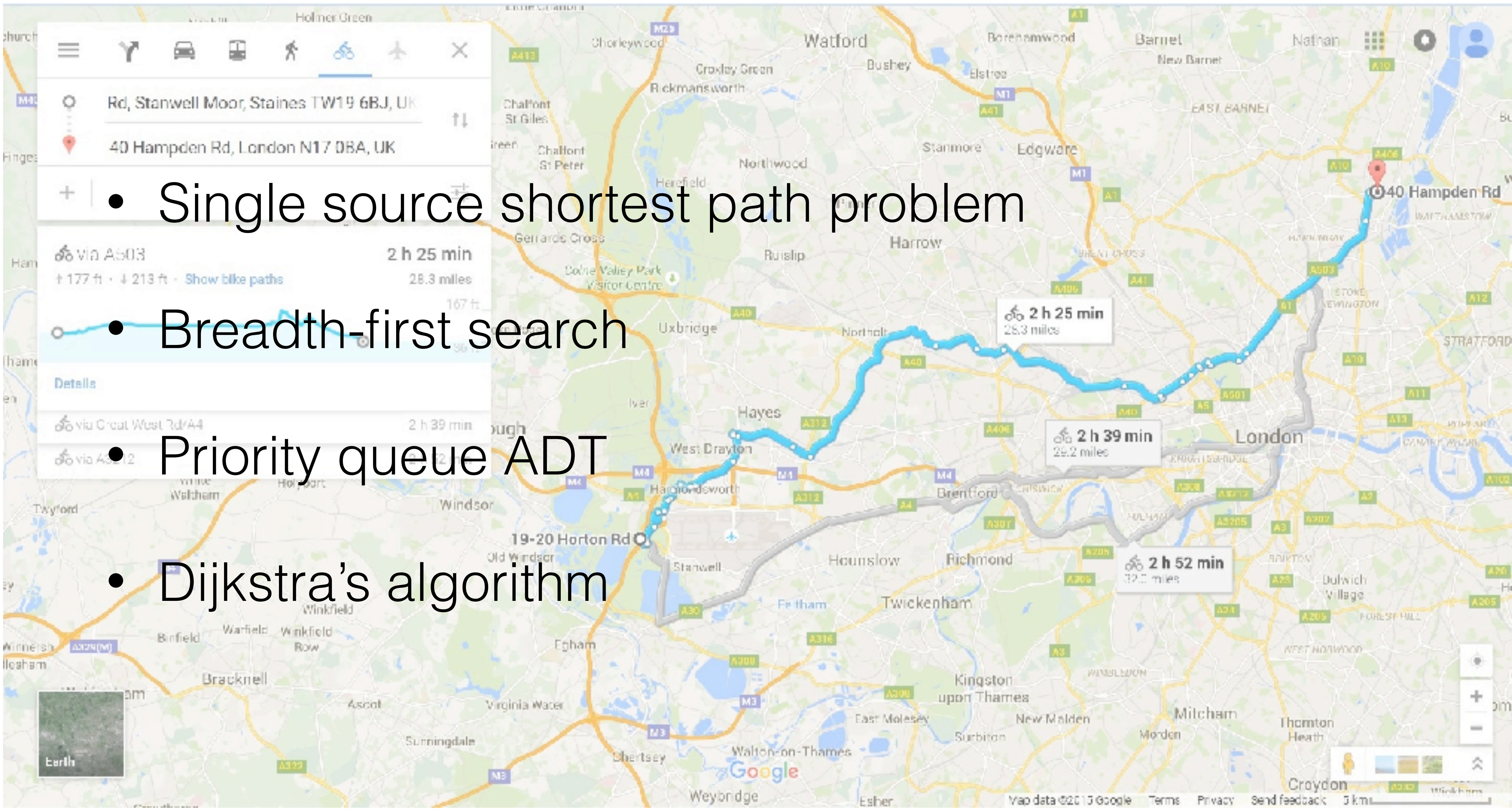
Priority queue ADT
part 1



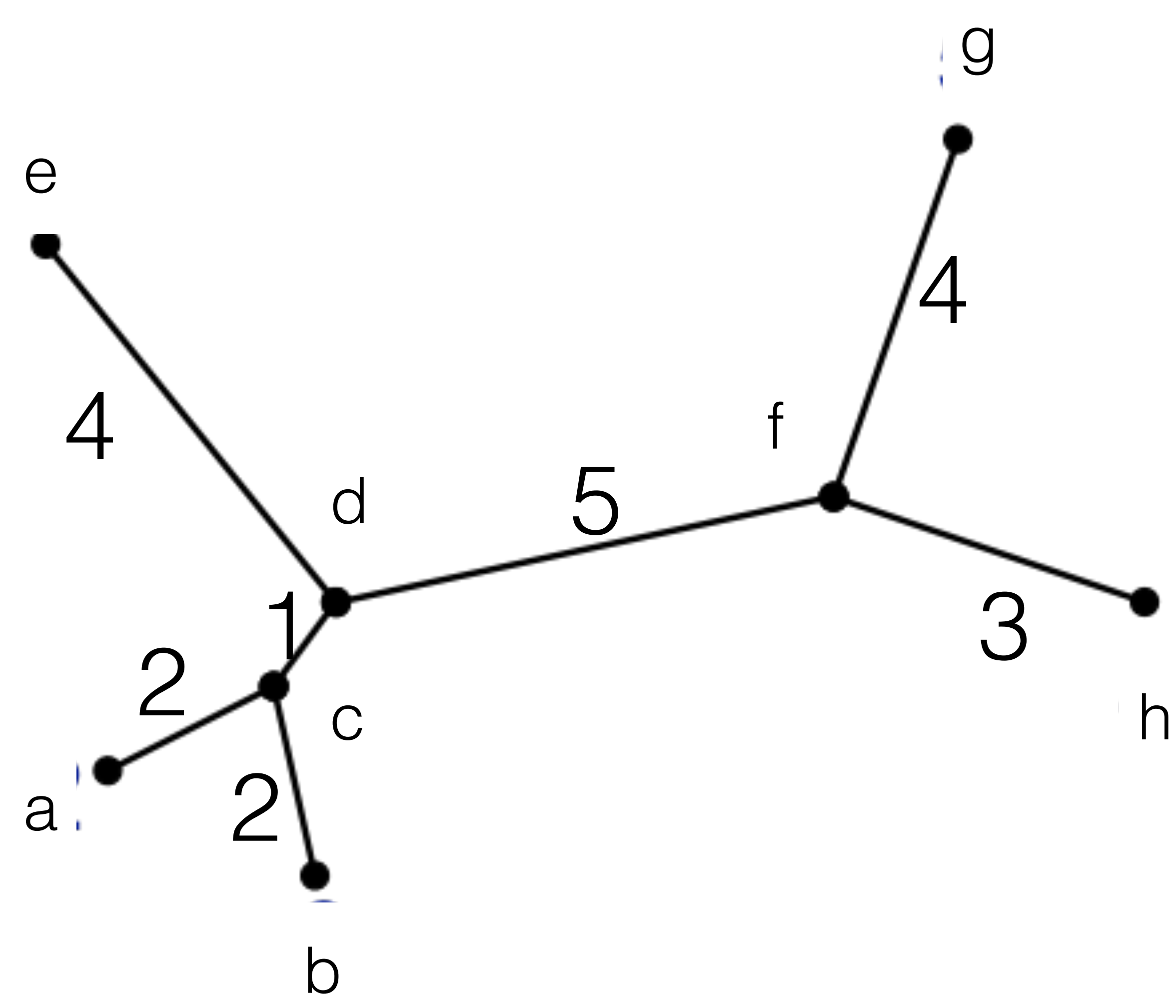
CS 146 - Spring 2017

Today



- Single source shortest path problem
- Breadth-first search
- Priority queue ADT
- Dijkstra's algorithm

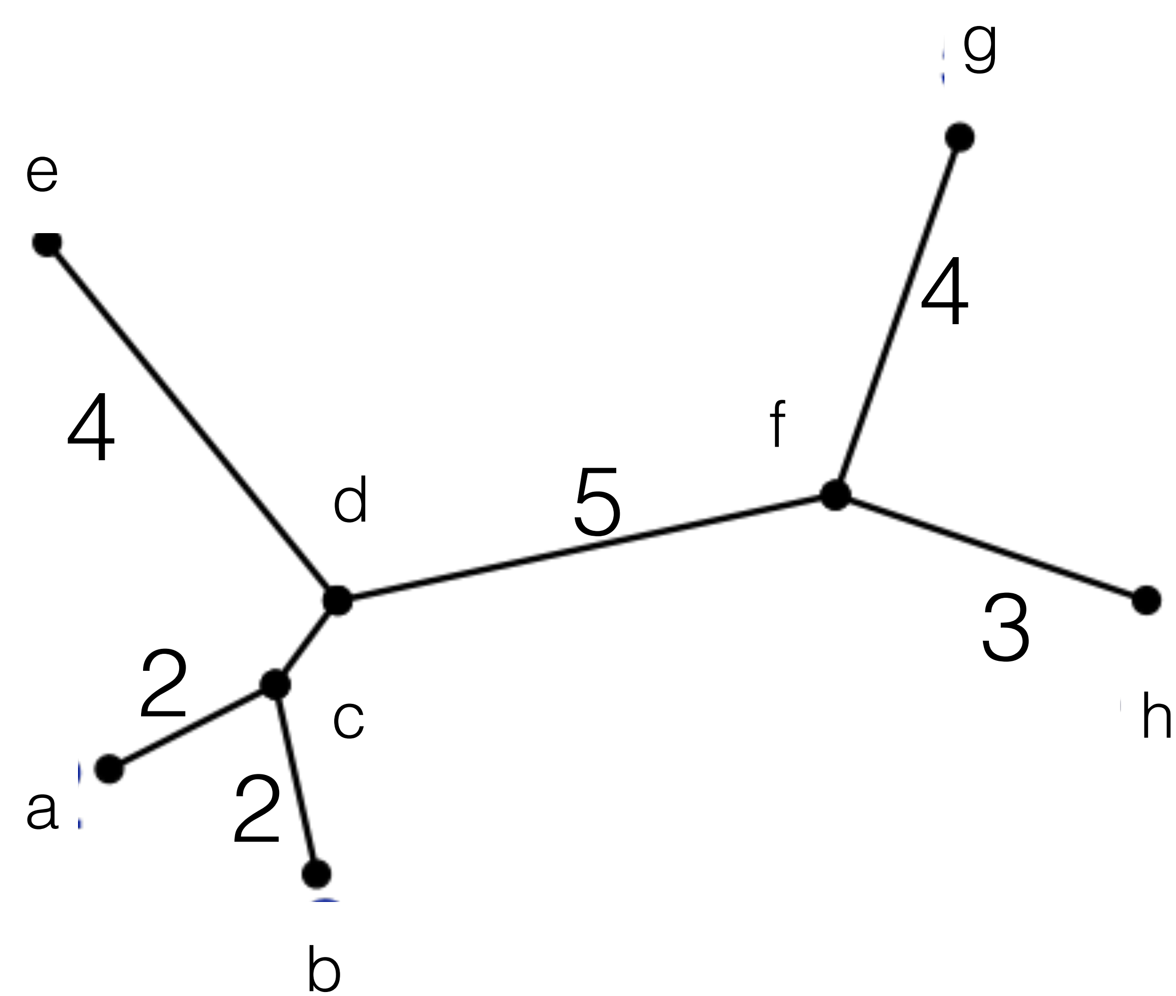
Question



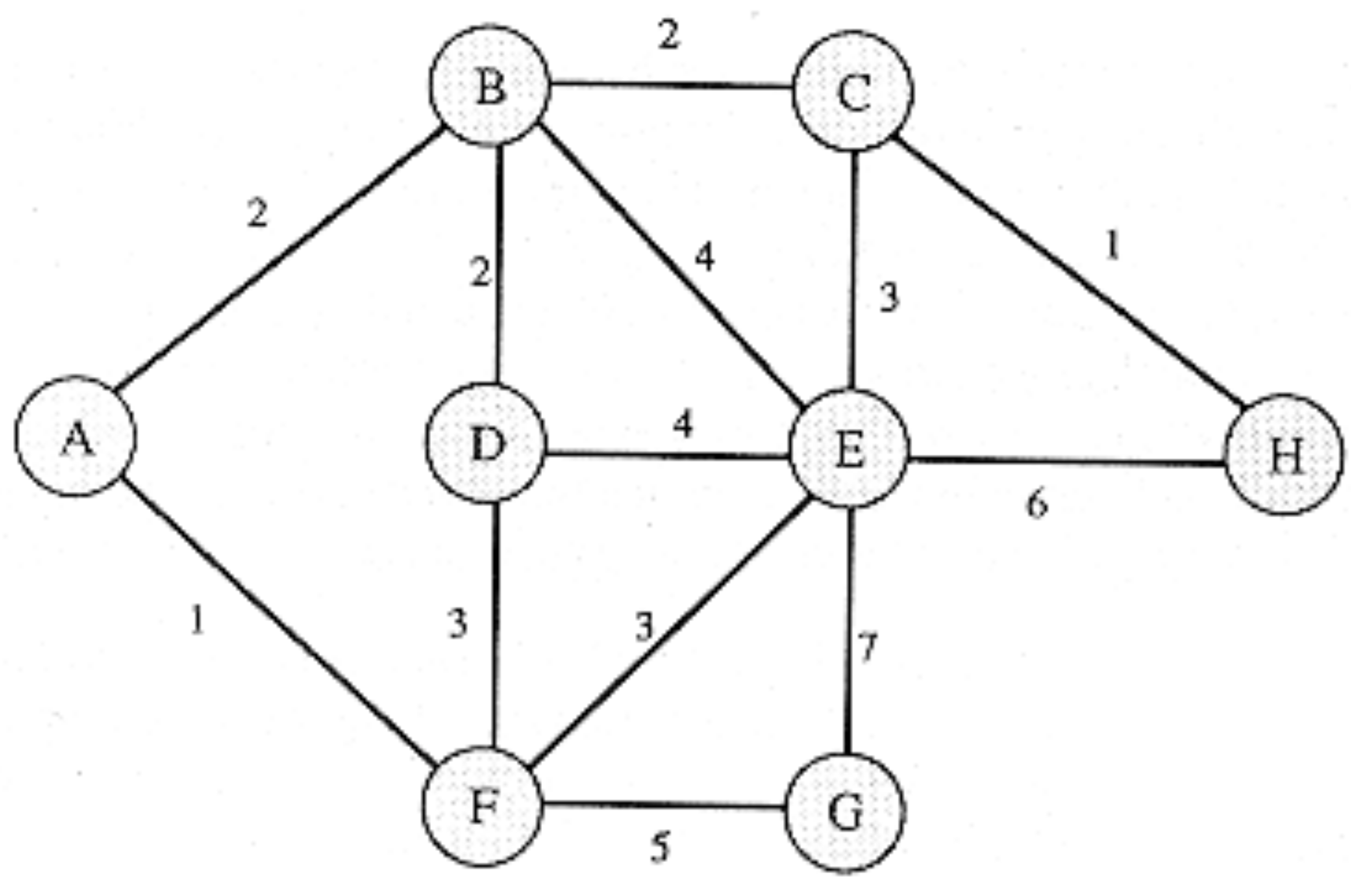
Find an algorithm for finding the length of the shortest path between vertex b and g

Observation 1

- to solve this problem, we end up solving a more general problem:
- find the distance from b to all other nodes
- we call this problem the **single source shortest path problem**



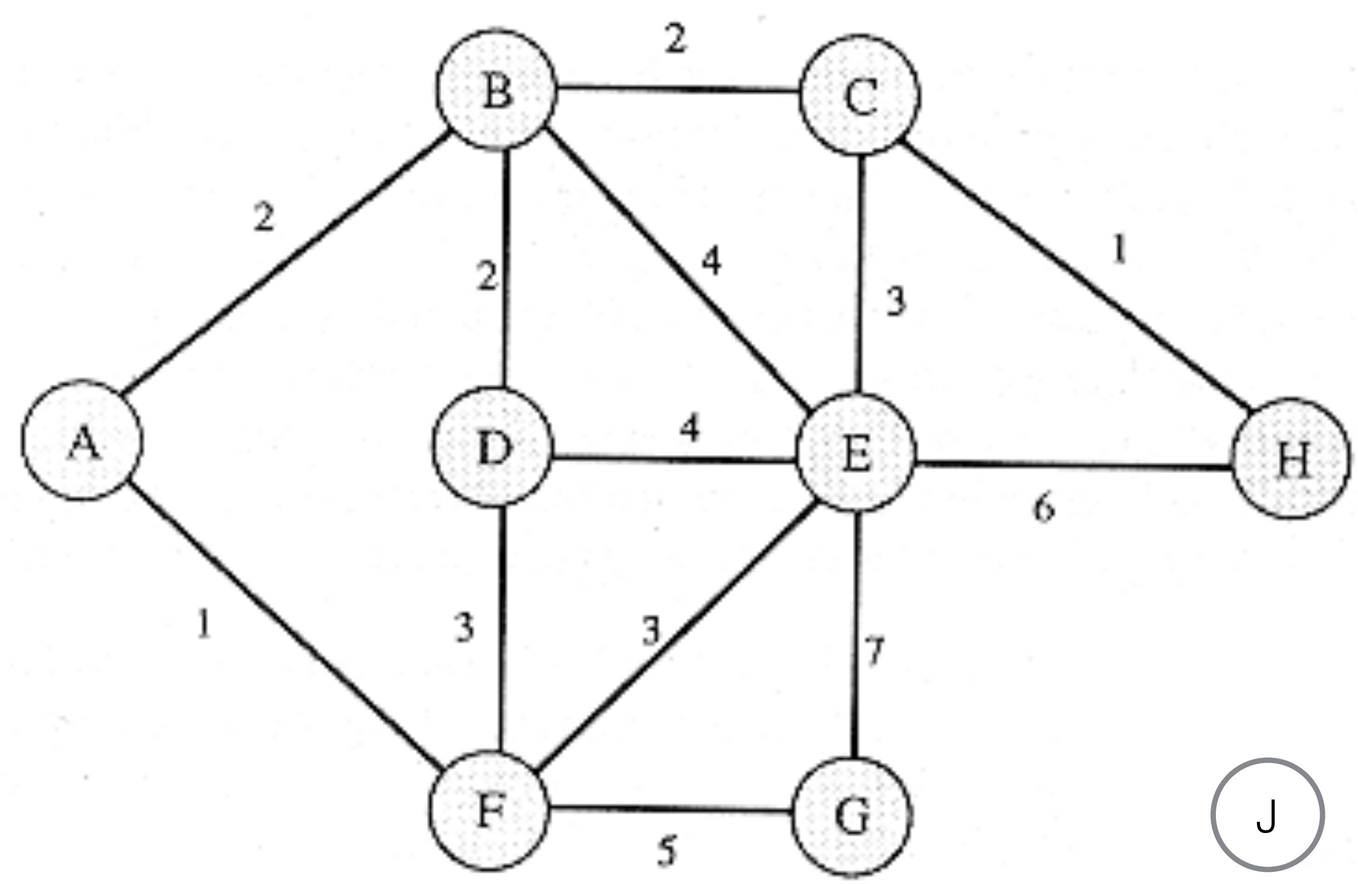
The single source shortest path problem formal definition



Input: a **weighted graph** G and a vertex s in G called **source**

Output: **for each vertex** v in G, the **distance** from s to v

What is distance anyways?

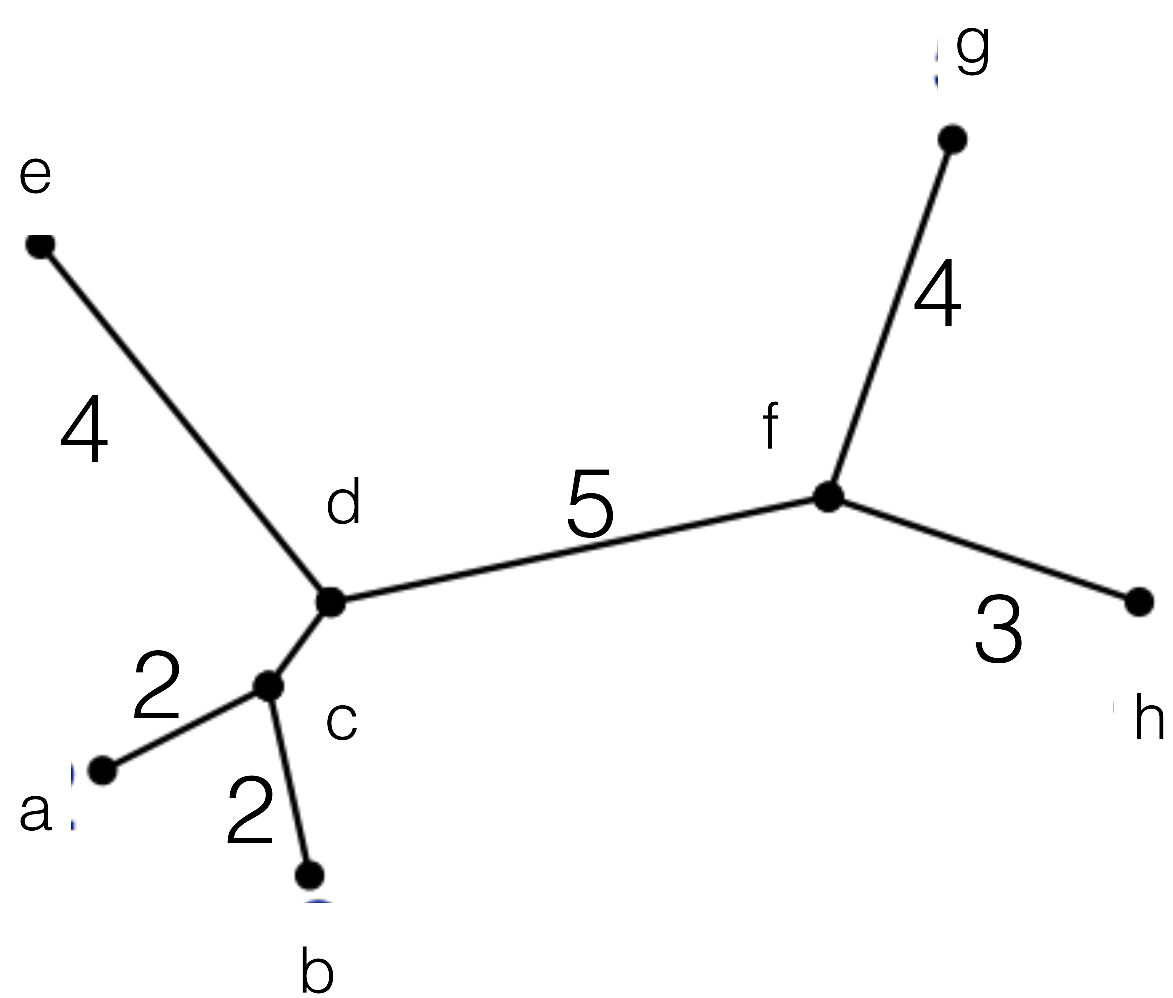


What is the distance between A and E?

What is the distance between A and J?

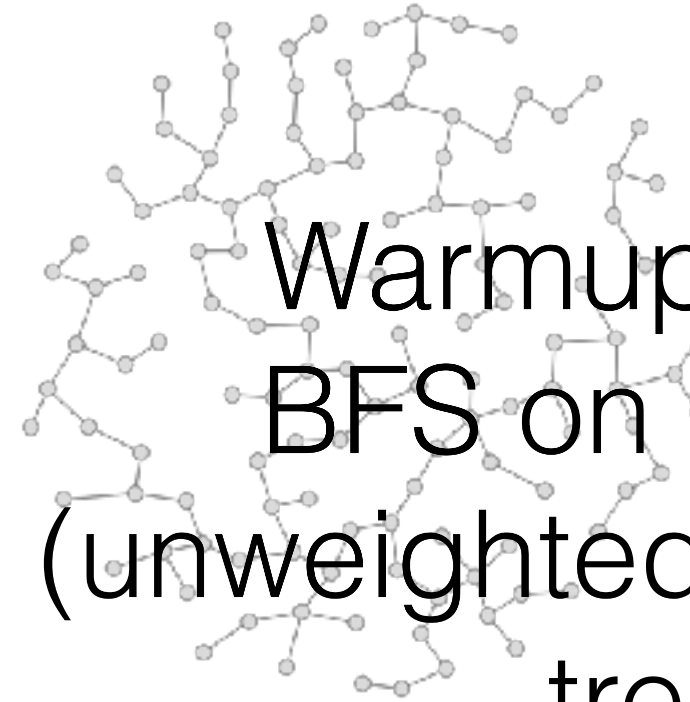
The distance between two vertices is
the length of a **shortest** path between them.

Observation 2



- the idea of the algorithm is to….
- start at the start vertex b
- compute the distances to b's neighbors a & d
- compute the distances to a & d's neighbors, etc

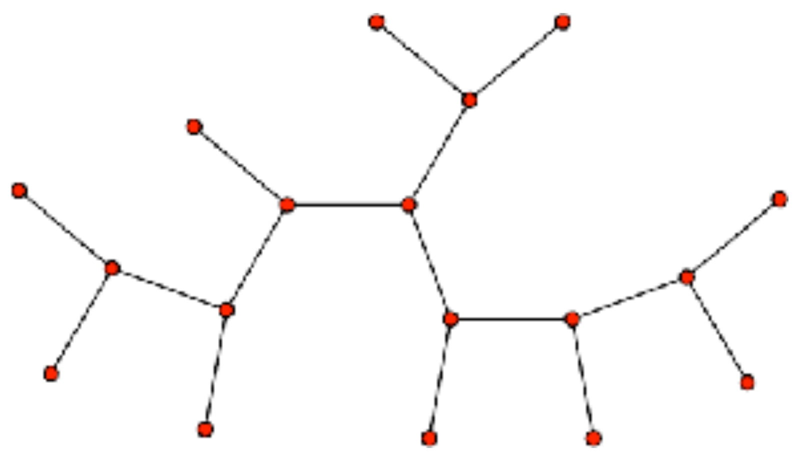
how do we make sure we don't go back?



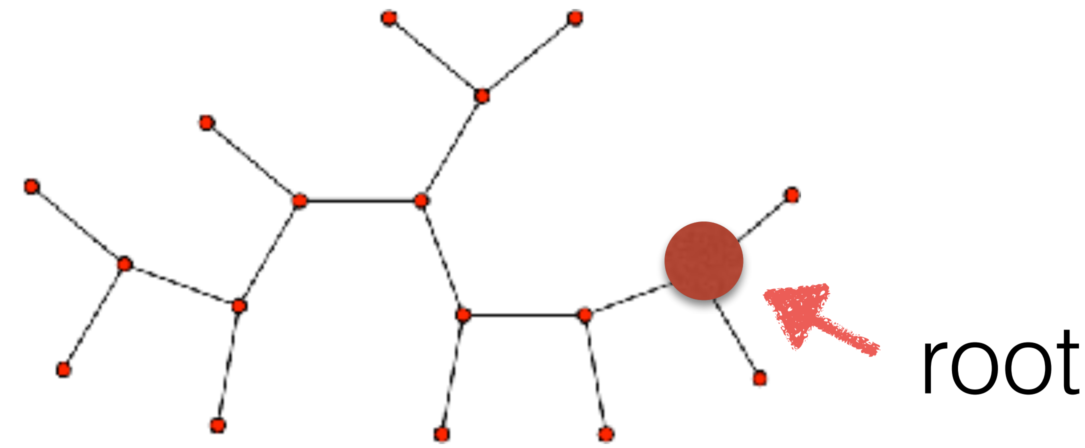
Warmup: BFS on a (unweighted) tree

```
map bfs(tree T, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in T
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
       for each neighbor w of v {
          if dist.get(w) == +inf {
              dist.put(w, dist.get(v) + 1)
              queue.enqueue(w)
          }
       }
    }
    return dist
}
```

Just so we're clear

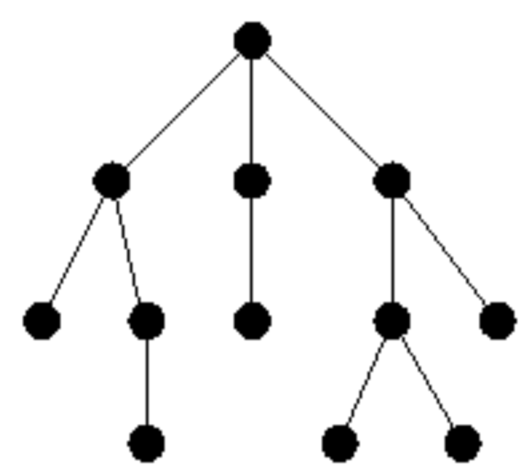


this is a tree

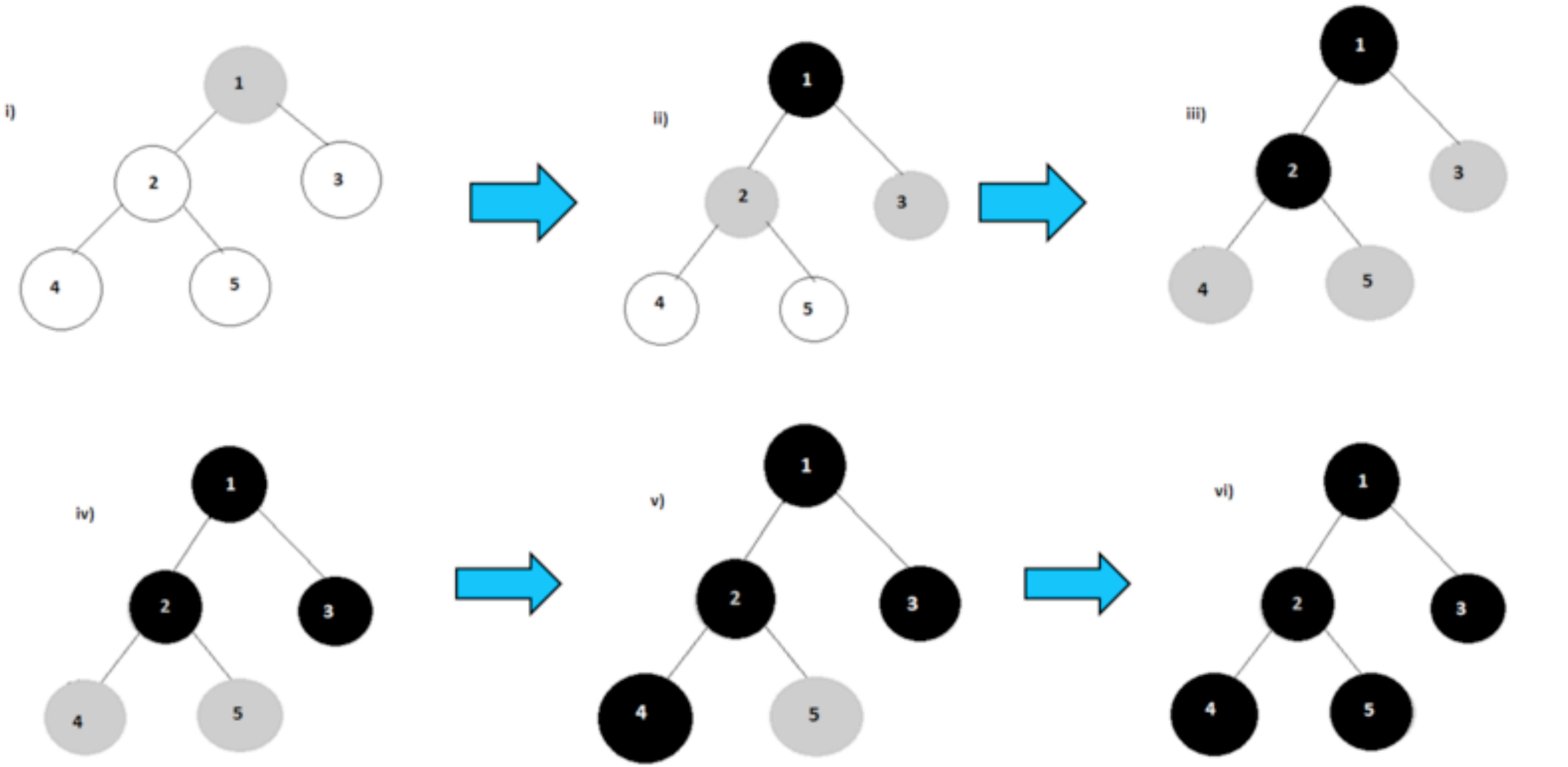


this is a rooted tree:
tree + special node marked as root

this is a rooted tree
it's implied the root
is the top node



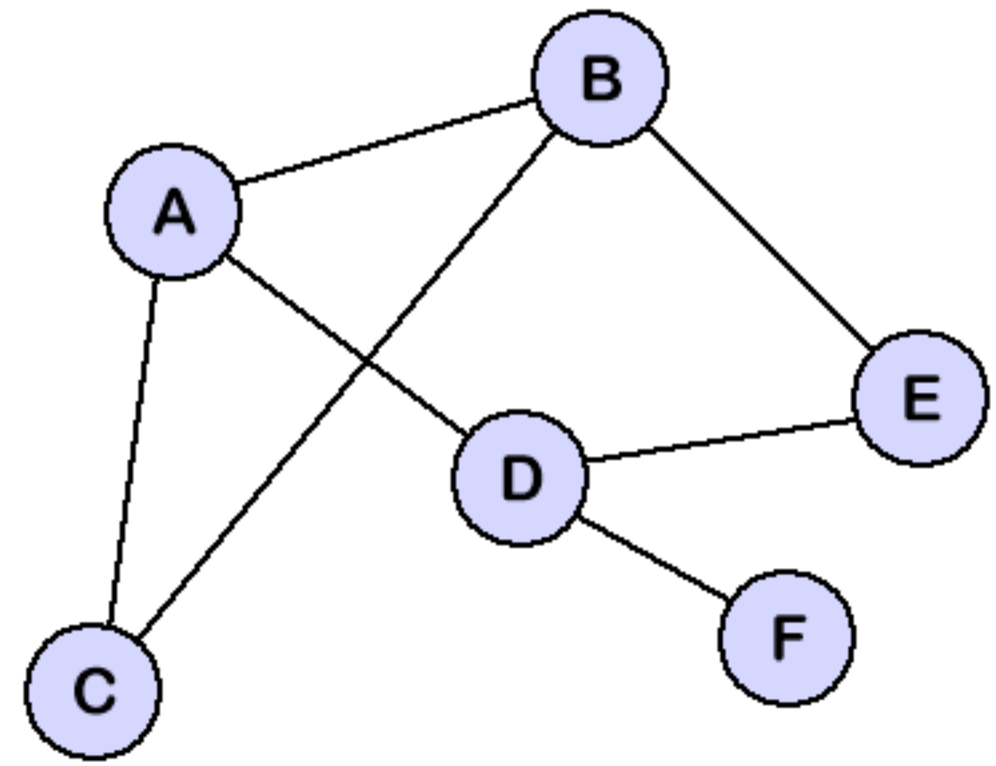
Example



white=not in queue grey=in queue black=out of queue

Warmup:
BFS on an unweighted graph

```
map bfs(graph G, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
       for each neighbor w of v {
          if dist.get(w) == +inf {
             dist.put(w, dist.get(v) + 1)
             queue.enqueue(w)
          }
       }
    }
    return dist
}
```



Breadth-first search why does it still work on a graph?

- nodes that are closer to the source are processed before the nodes that are further from the source
- in the case of an unweighted graph, the order of closeness to source is the same as the order in which a node is added to queue

Running time of BFS

```
map bfs(graph G, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
      for each neighbor w of v {
          if dist.get(w) == +inf {
              dist.put(w, dist.get(v) + 1)
              queue.enqueue(w)
          }
       }
    }
    return dist
}
```

once per vertex

once per edge **over the entire algorithm** (not just inside loop)

NOT a nested loop analysis!

Running time of BFS

```
map bfs(graph G, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
      for each neighbor w of v {
            if dist.get(w) == +inf {
                dist.put(w, dist.get(v) + 1)
                queue.enqueue(w)
            }
       }
    }
    return dist
}
```

true once per vertex

NOT a nested loop analysis!

Running time of BFS

V x time(queue.dequeue)

V x time(dist.put)

V x time(queue.enqueue)

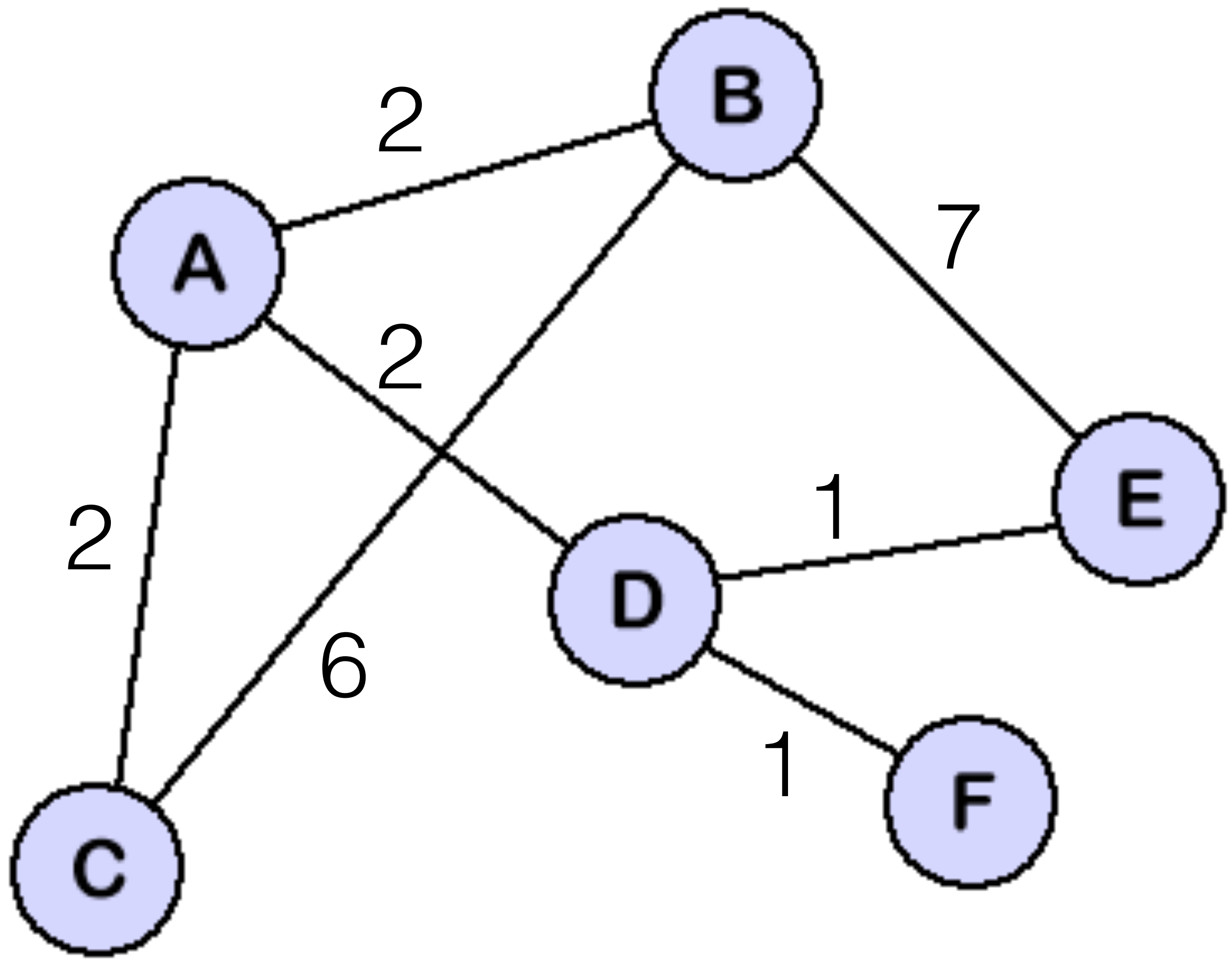
<= 2E x time(dist.get)

if dist is a hash table, each dist operation is constant time

queue is a FIFO queue, each queue operation is const time

BFS time is O(V + E)

Question



Will BFS work if we replace the weight of 1 with the weight?

Problem: nodes are no longer processed in order of distance

- processed in order of (unweighted) distance to s
- want them to be processed in order of **weighted** distance to s
- solution: replace FIFO queue with **priority queue**

Priority queue ADT

key means priority
in this context

- generalization of a FIFO queue
- **insert(key, value)**
- **extractMin()** <- removes node with min priority
- **decreaseKey(entry, newKey)**
- **makeQueue(dict of key-value pairs)**
  or new PriorityQueue(dict of key-value pairs)
- no way to delete an arbitrary node, extract-min is the equivalent of dequeue in a FIFO queue

Priority queue implementations

- linked list: simplest, but not very efficient
- (binary) **heap**: most popular, will study later
- d-ary heap: a generalization of binary heap
- fibonacci heap: esoteric, theoretical best
- binomial heaps…
- binary heap has nothing to do with Java heap memory

```
map bfs(graph G, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
      for each neighbor w of v {
          if dist.get(w) == +inf {
              dist.put(w,
                  dist.get(v) + 1)
              queue.enqueue(w)

          }
       }
    }
    return dist
}
```

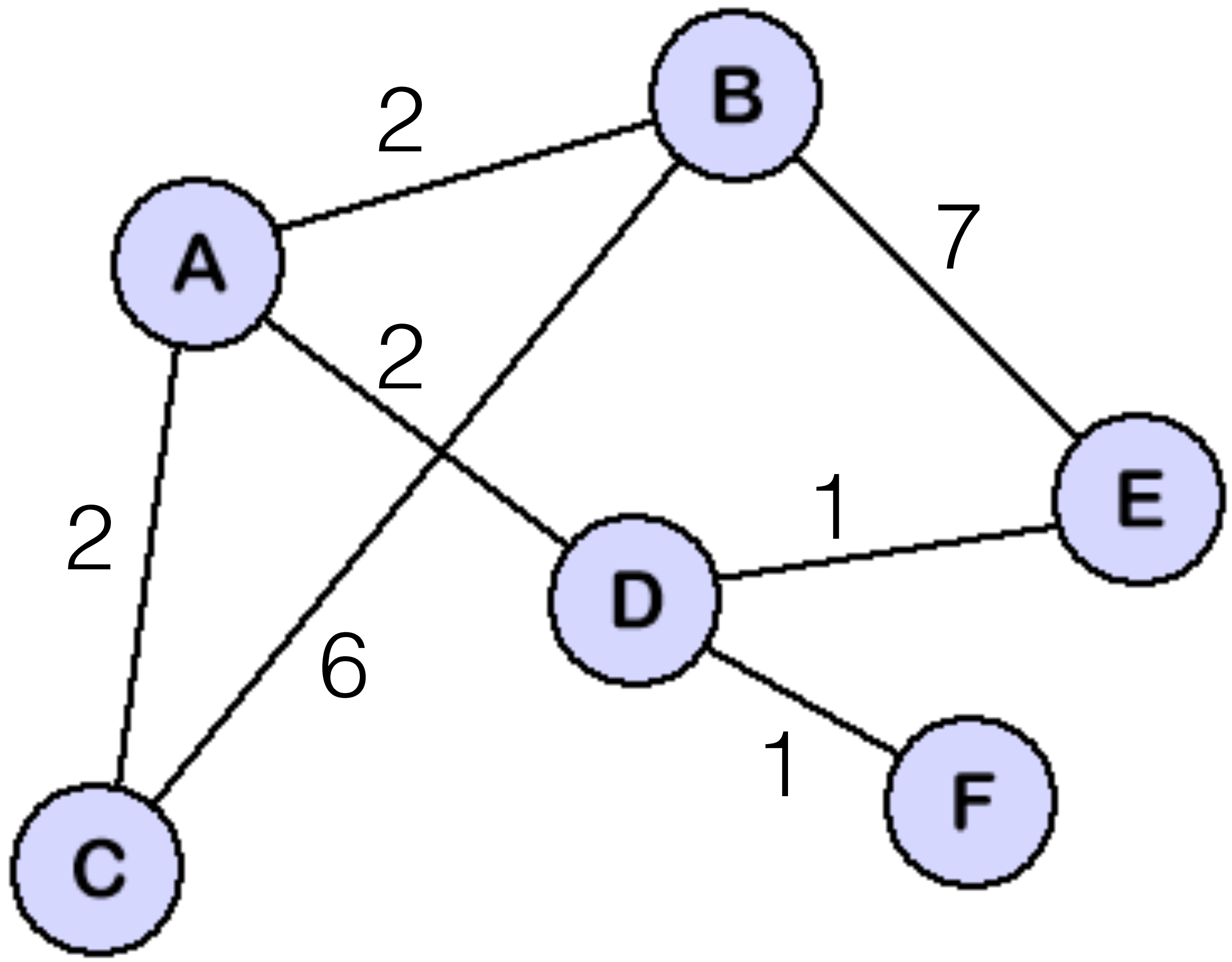
```
map dijkstra(weighted-graph G, vertex s)
    dist = new map()
    queue = new priorityQueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
          if w should be updated {
              dist.put(w,
                dist.get(v)+ weight(v,w)
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

Dijkstra's algorithm: first steps (incomplete)

```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

Dijkstra’s algorithm

Example: Dijkstra's algorithm



```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

relax(v->w)

updates dist to w
via path through edge v->w

```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

once per vertex

once per edge **over the entire algorithm** (not just inside loop)

Running time of Dijkstra's algorithm

$\leq$ V+4E x time(dictionary op)

1 x time(queue.makeQueue)

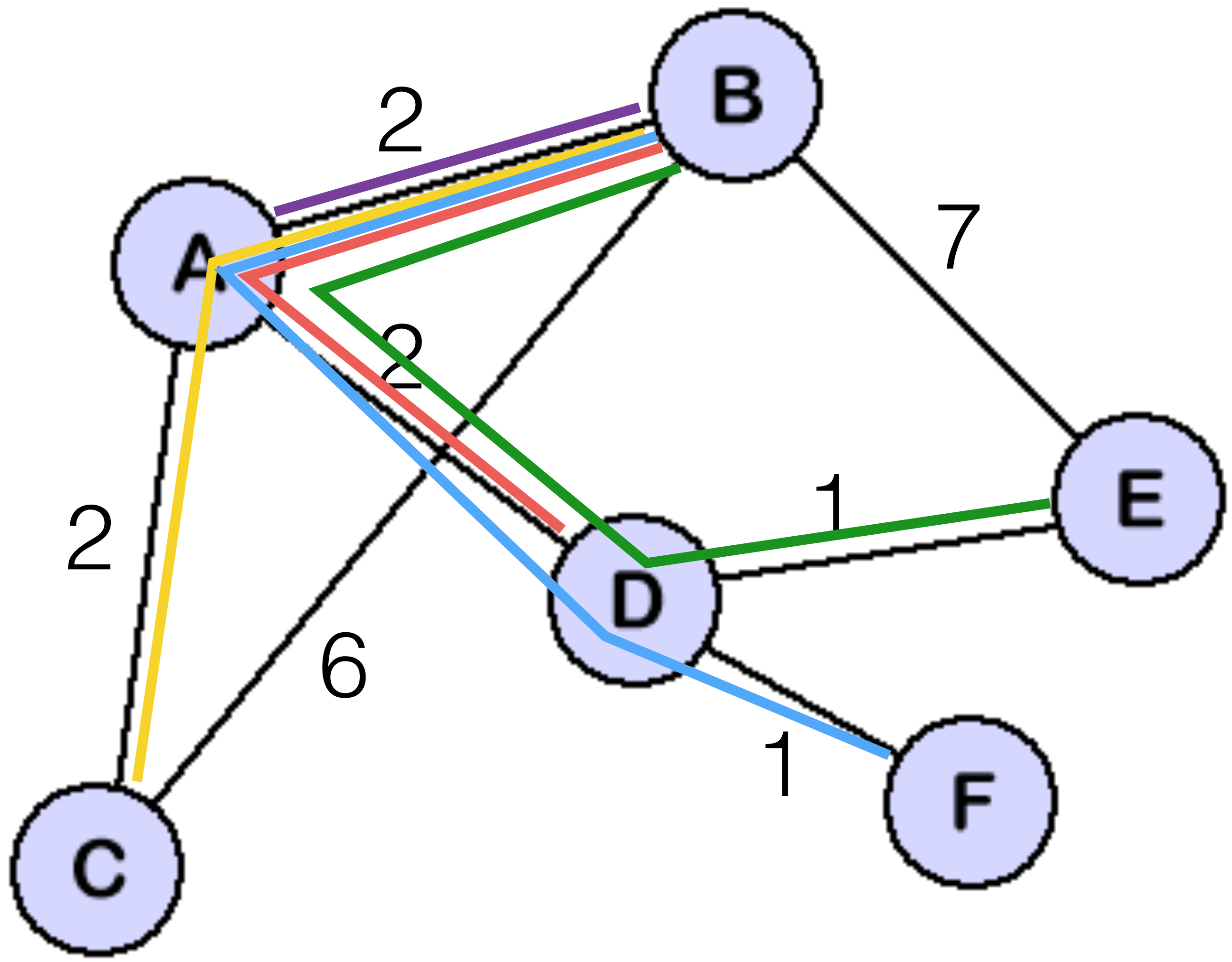
V x time(queue.extractMin)

$\leq$ **E** x time(queue.decreaseKey)

up to a constant factor, #queue ops = #dict ops,
but dictionary ops are constant time with hash table
running time is dominated by queue operations

$O(T(\text{makeQueue}(V)) + V \times T(\text{extractMin}(V)) + E \times T(\text{decKey}(V)))$

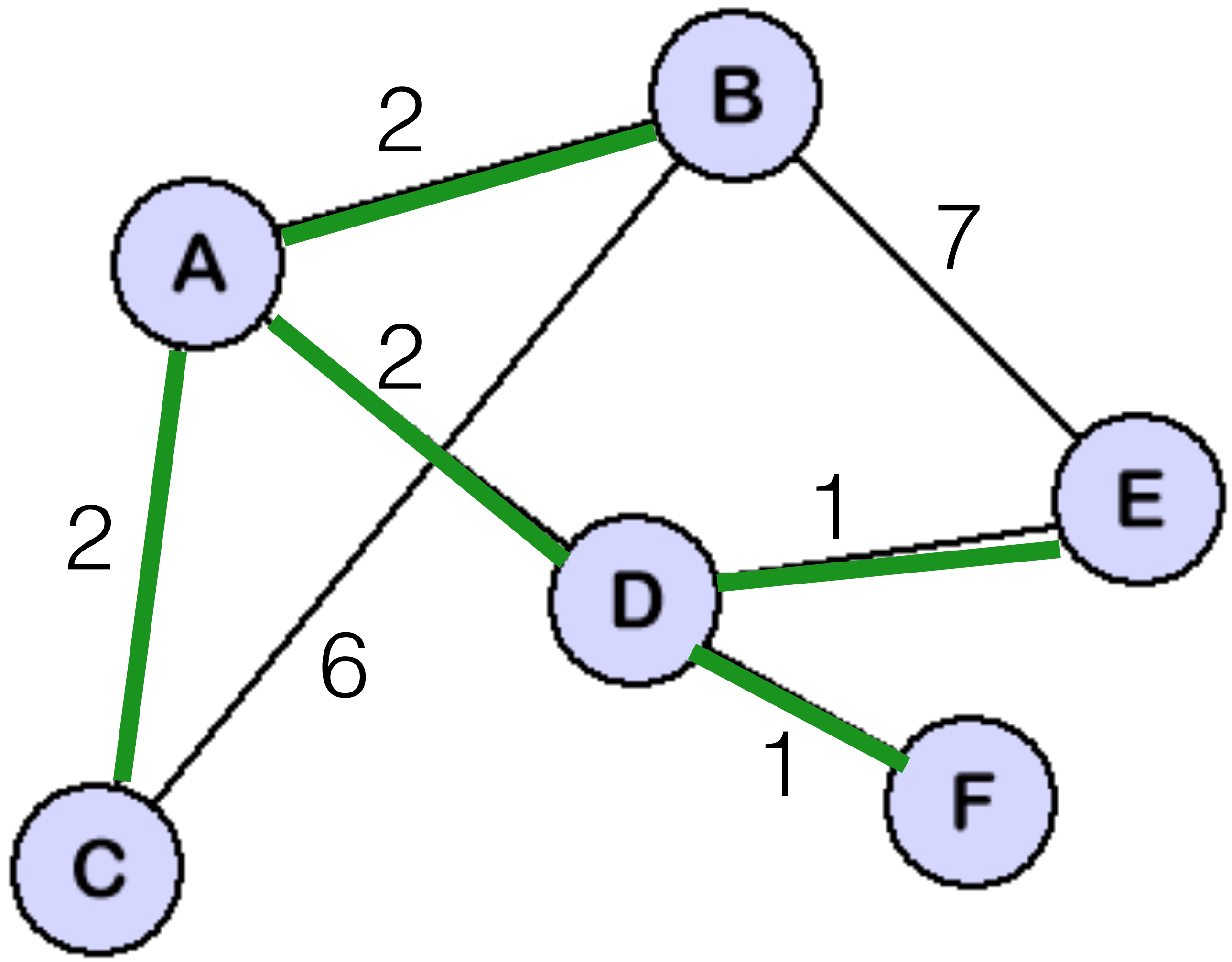
Shortest paths from B



The shortest paths from a vertex to all other nodes **form a tree**.

Why?

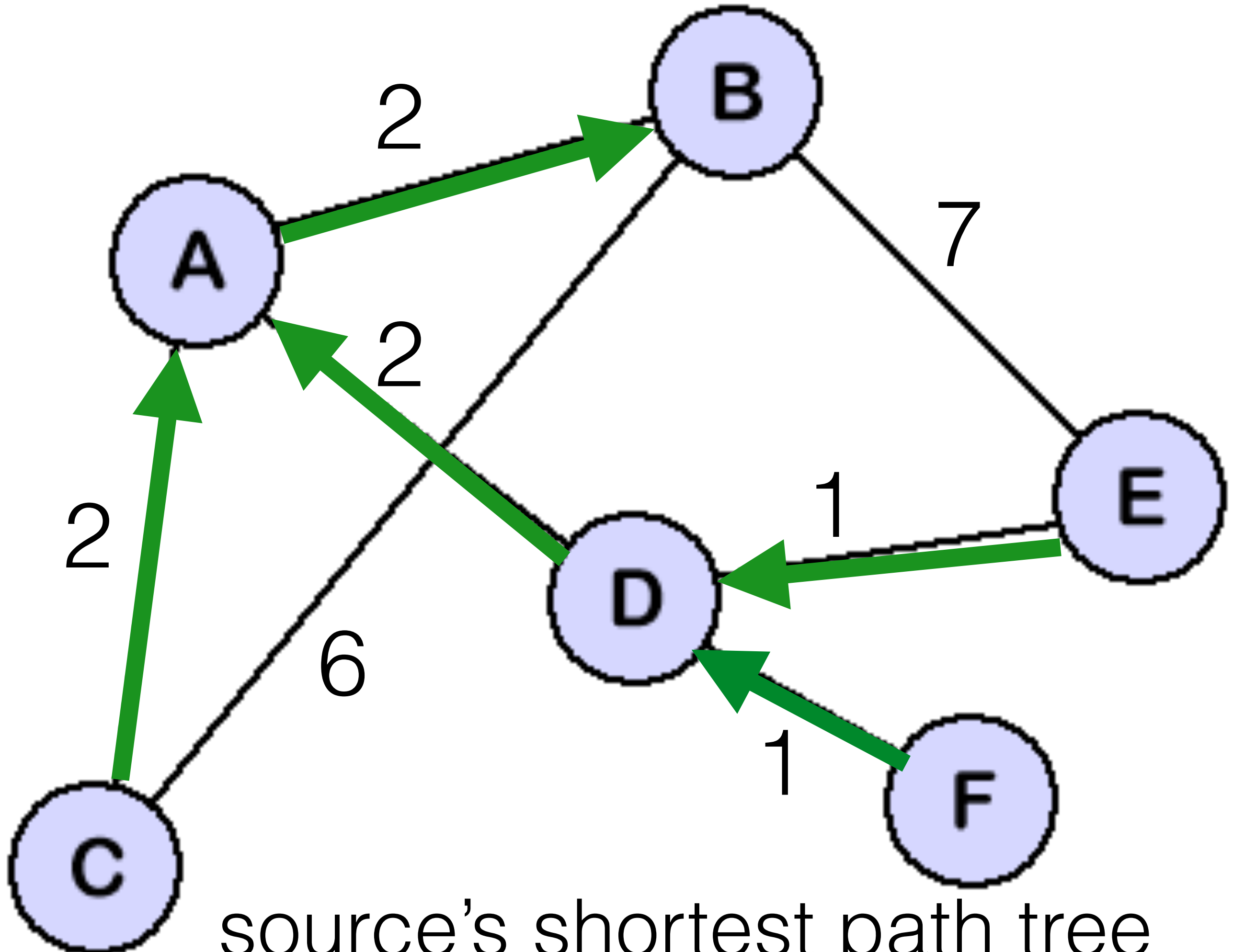
B’s shortest path tree



```
map augmented-dijkstra(weighted-graph G, vertex s) {
    dist = new map()
    prev = new map()   <- maps nodes to previous node on
                              source's shortest path tree
    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
              prev.put(w, v)
          }
       }
    }
    return prev
}
```

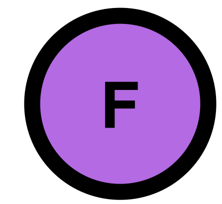
Retrieving the actual path

step 1: record shortest path tree



source's shortest path tree
encoded in "prev" map

+



target node

Retrieving the actual path

step 2: reconstruct path to some target node