Announcements

- Exam 1 on Friday
- Please remember to read the instructions at http://cseweb.ucsd.edu/~dakane/CSE101/Exam1Instructions.pdf and complete the exam instructions gradescope assignment.

Last Time

- Shortest Paths
- BFS computes paths with fewest numbers of edges
- Weighted graphs

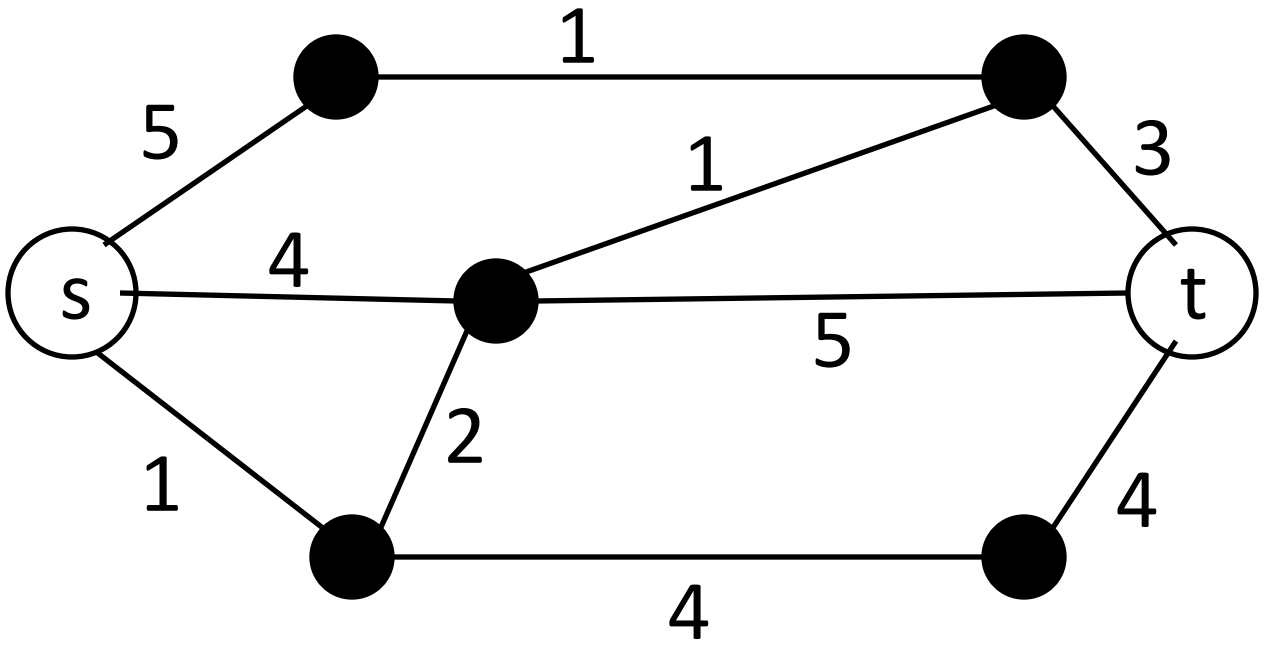
Problem: Shortest Paths

Problem: Given a Graph G with vertices s and t and a length function ℓ, find the shortest path from s to t.

Question: Shortest Path

What is the length of the shortest s-t path below?

A) 5
B) 6
C) 7
D) 8
E) 9



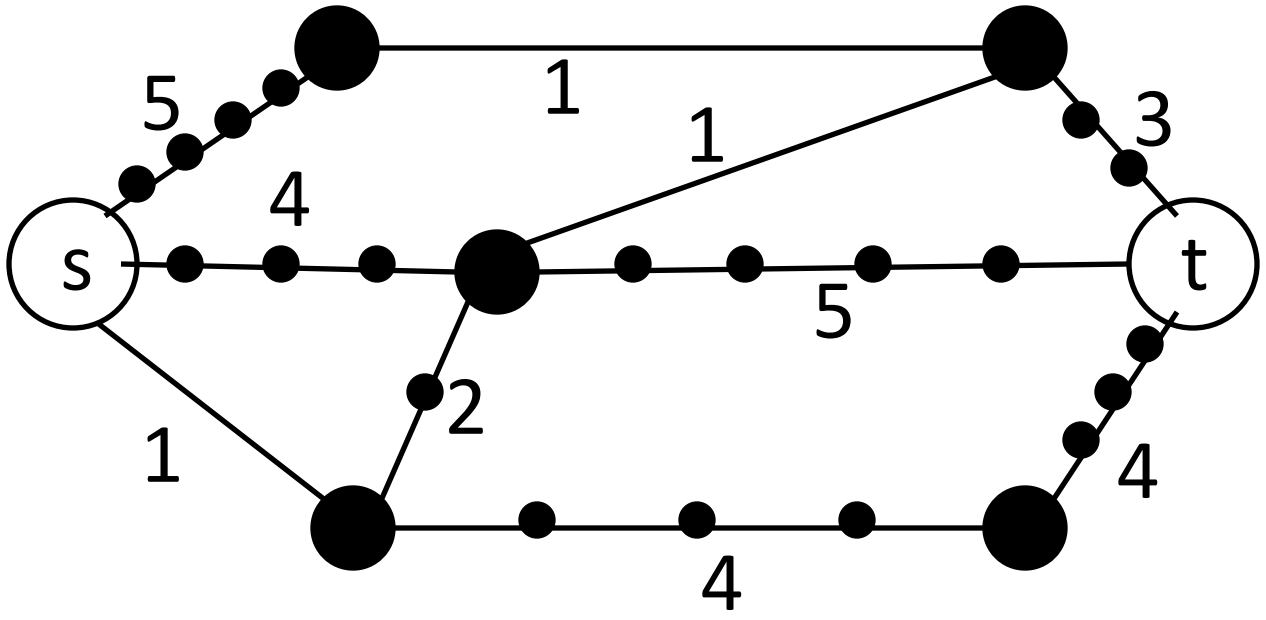
1+2+1+3 = 7

Today

- Dijkstra’s Algorithm

How to Compute

Idea: Break edges into unit length edges.



Runtime: O(sum of edge lengths)

This is a problem if some edge lengths are large.

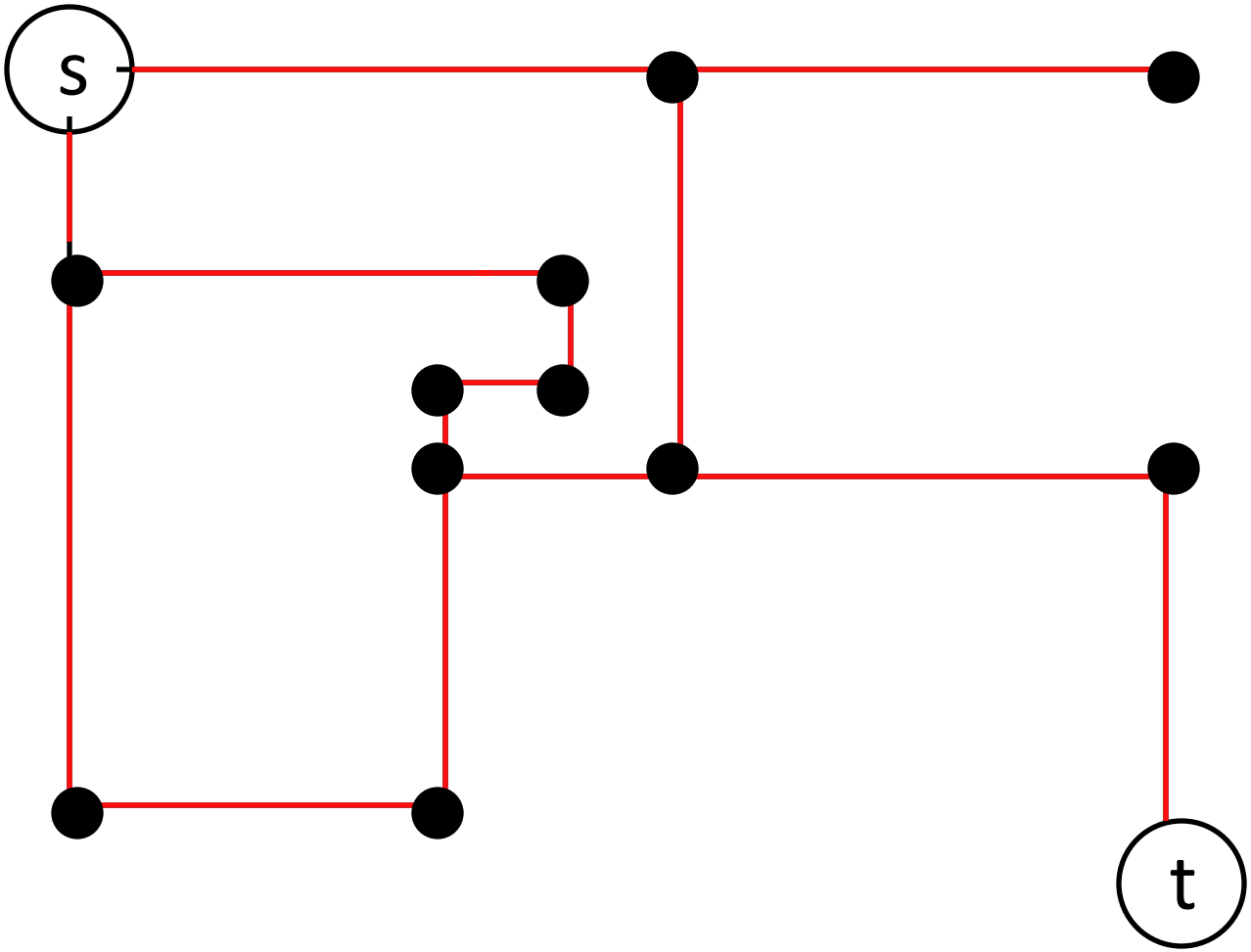
Another Way

If you have very long edge lengths, most steps will just consist of advancing slightly along a bunch of edges.

Try to find a way to fast forward through these boring steps.

Occasionally have interesting steps where the wavefront hits a new vertex.

Ooze



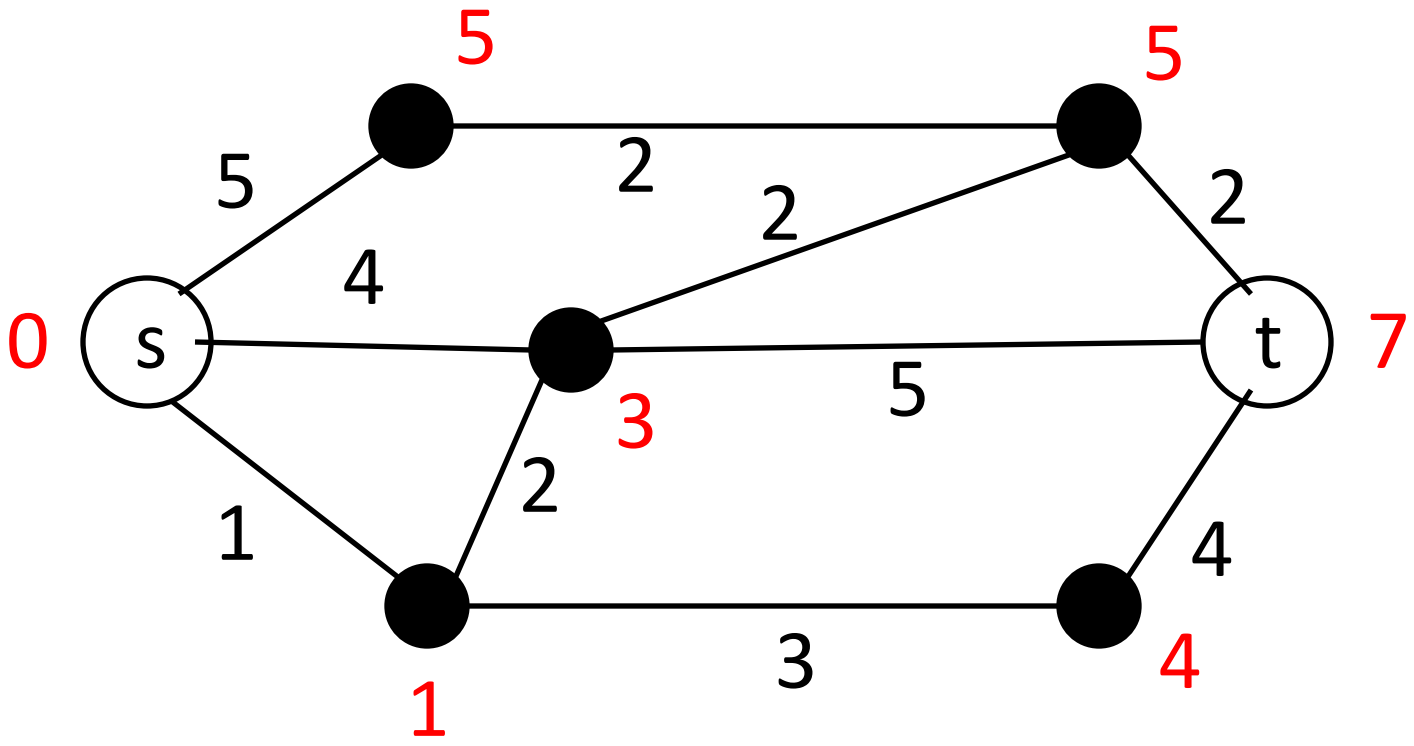
Algorithm

- How do we make this an algorithm?
- Simulate the above process keep track of when new vertices are discovered.
- If v is discovered at time t(v), the ooze from v will reach neighbor w at time $t(v)+\ell(v,w)$.
- Next vertex to be discovered is one with minimal $t(v)+ \ell(v,w)$.

Algorithm

```
Distances(G,s,ℓ)
  dist(s) ← 0
  While(not all distances found)
    Find minimum over (v,w) ∈ E
      with v discovered w not
      of dist(v)+ℓ(v,w)
    dist(w) ← dist(v)+ℓ(v,w)
    prev(w) ← v
```

Example



0+3=3

0+5=5

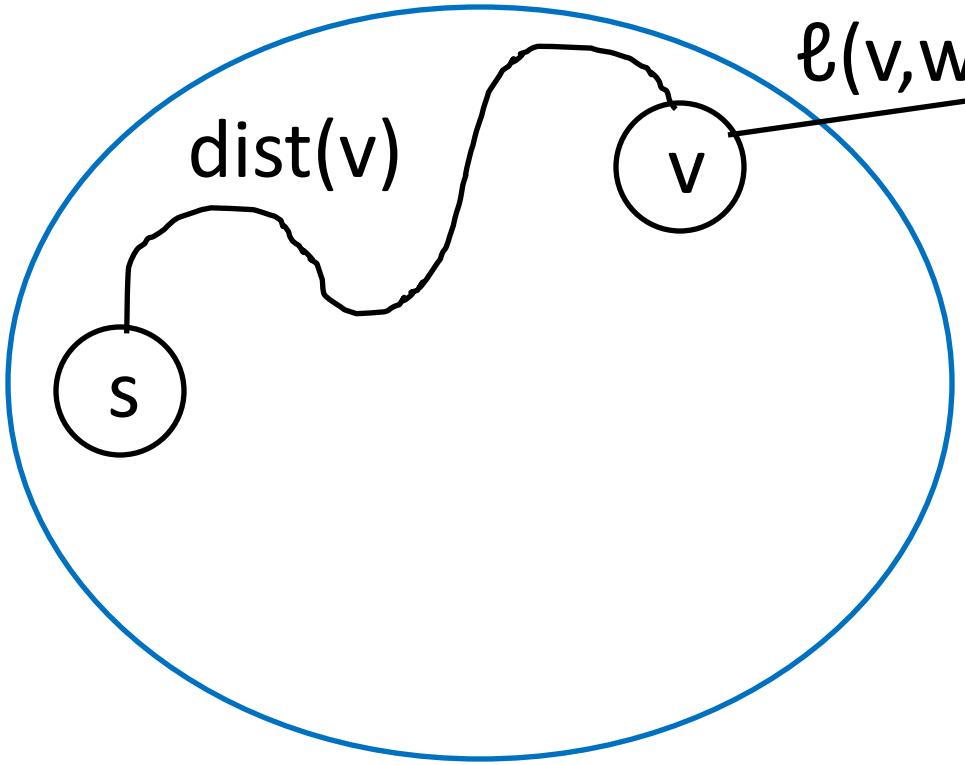
Why does this work?

Claim: Whenever the algorithm assigns a distance to a vertex v that is the length of the shortest path from s to v.

Proof by Induction:

- dist(s) = 0 [the empty path has length 0]
- When assigning distance to w, assume that all previously assigned distances are correct.

Inductive Step



This is the shortest path from s to *any* vertex outside the bubble.

Correctly Assigned Distances

Question: Runtime

What is the runtime of this algorithm?

A) $O(|V|+|E|)$
B) $O(|V|\log|V|+|E|)$
C) $O(|V||E|)$
D) $O(|E|^2)$
E) $O(2^{|V|})$

```
Distances(G,s,ℓ)
  dist(s) ← 0
  While(not all distances found)
    Find minimum over (v,w) ∈ E
      with v discovered w not
      of dist(v)+ℓ(v,w)
    dist(w) ← dist(v)+ℓ(v,w)
    prev(w) ← v
```

Runtime

```
Distances(G,s,ℓ)
   dist(s) ← 0
   While(not all distances found)
      Find minimum over (v,w) ∈ E
         with v discovered w not
         of dist(v)+ℓ(v,w)
      dist(w) ← dist(v)+ℓ(v,w)
      prev(w) ← v
```

O(|V|) iterations

O(|E|) edges

Runtime O(|V||E|)

Runtime

- This is too slow.
- **Problem:** Every iteration we have to check every edge.
- **Idea:** Most of the comparison doesn't change much iteration to iteration. Use to save time.
- **Specifically:** Record for each w best value of dist(v,w)+ ℓ(v,w).

Attempt II

Runtime: $O(|V|^2+|E|)$

```
Distances(G,s,ℓ)
  For v ∈ V
    dist(v) ← ∞, done(v) ← false
  dist(s) ← 0, done(s) ← true
  While(not all vertices done)
    Find v not done with minimum dist(v)
    done(v) ← true
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
```

O(|V|)

O(|V|) iterations

O(|V|)

O(|E|) total

See if better path to w & update dist(w)

Still too Slow

- Repeatedly ask for smallest vertex
  - Even though not much is changing from round to round, the algorithm is computing the minimum from scratch every time
- Use a data structure!
  - Data structures help answer a bunch of similar questions faster than answering each question individually
- For this kind of question, want a priority queue.

Priority Queue

A Priority Queue is a datastructure that stores elements sorted by a key value.

Operations:

- Insert – adds a new element to the PQ.
- DecreaseKey – Changes the key of an element of the PQ to a specified *smaller* value.
- DeleteMin – Finds the element with the smallest key and removes it from the PQ.

Attempt III

```
Distances(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, done(v) ← false
    Q.Insert(v) //using dist(v) as key
  dist(s) ← 0, done(s) ← true
  While(not all vertices done)
    Find v not done with minimum dist(v)
    done(v) ← true
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
        Q.DecreaseKey(w)
```

Distances(G,s,ℓ)
Initialize Priority Queue Q
For v ∈ V
dist(v) ← ∞, done(v) ← false
Q.Insert(v) //using dist(v) as key
dist(s) ← 0, done(s) ← true
While(Q not empty)
v ← Q.DeleteMin()

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

O(|V|) times

O(|V|) times

O(|E|) times

Runtime:
O(|V|) Inserts +
O(|V|) DeleteMins +
O(|E|) DecreaseKeys

What is the Runtime?

So what is the actual runtime then?

This depends on which implementation you use for your priority queue.

We will discuss a few.

Unsorted List

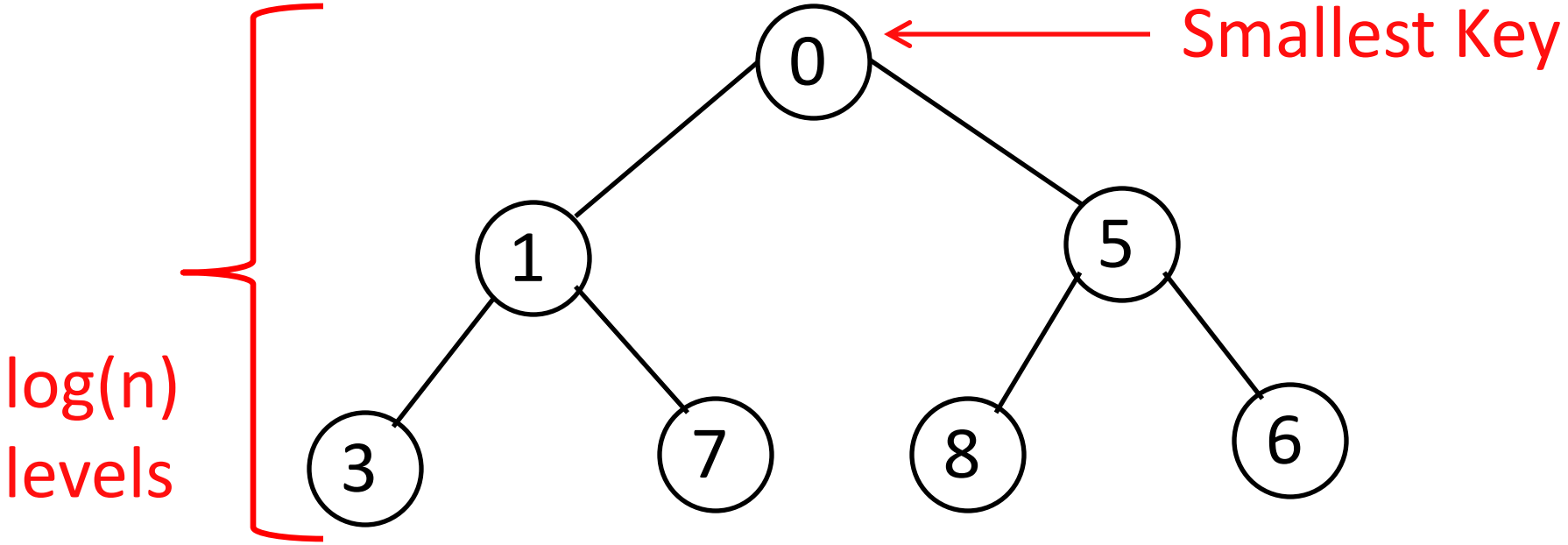
Store n elements in an unsorted list.

Operations:

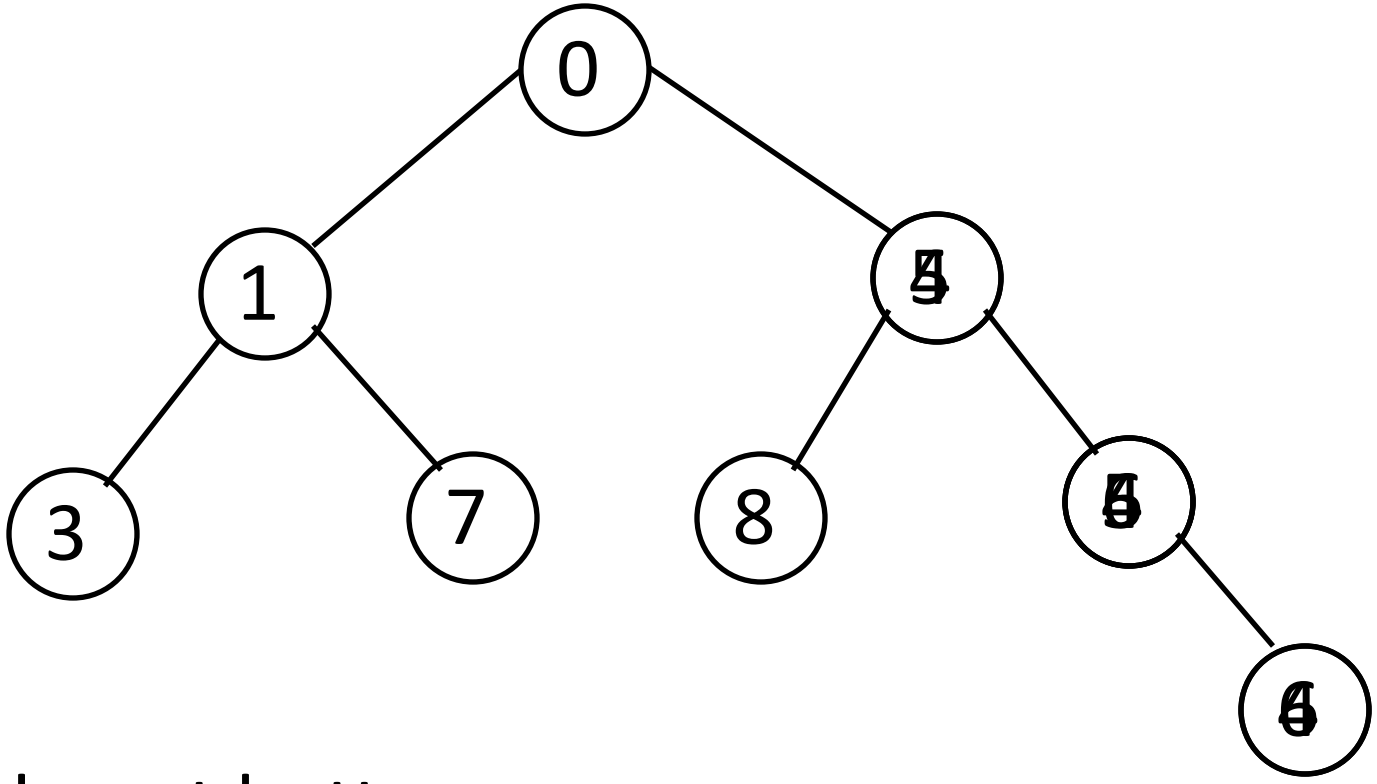
- Insert - O(1)
- DecreaseKey – O(1)
- DeleteMin – O(n)
- Dijkstra – $O(|V|^2+|E|)$

Binary Heap

Store elements in a balanced binary tree with each element having smaller key than its children.



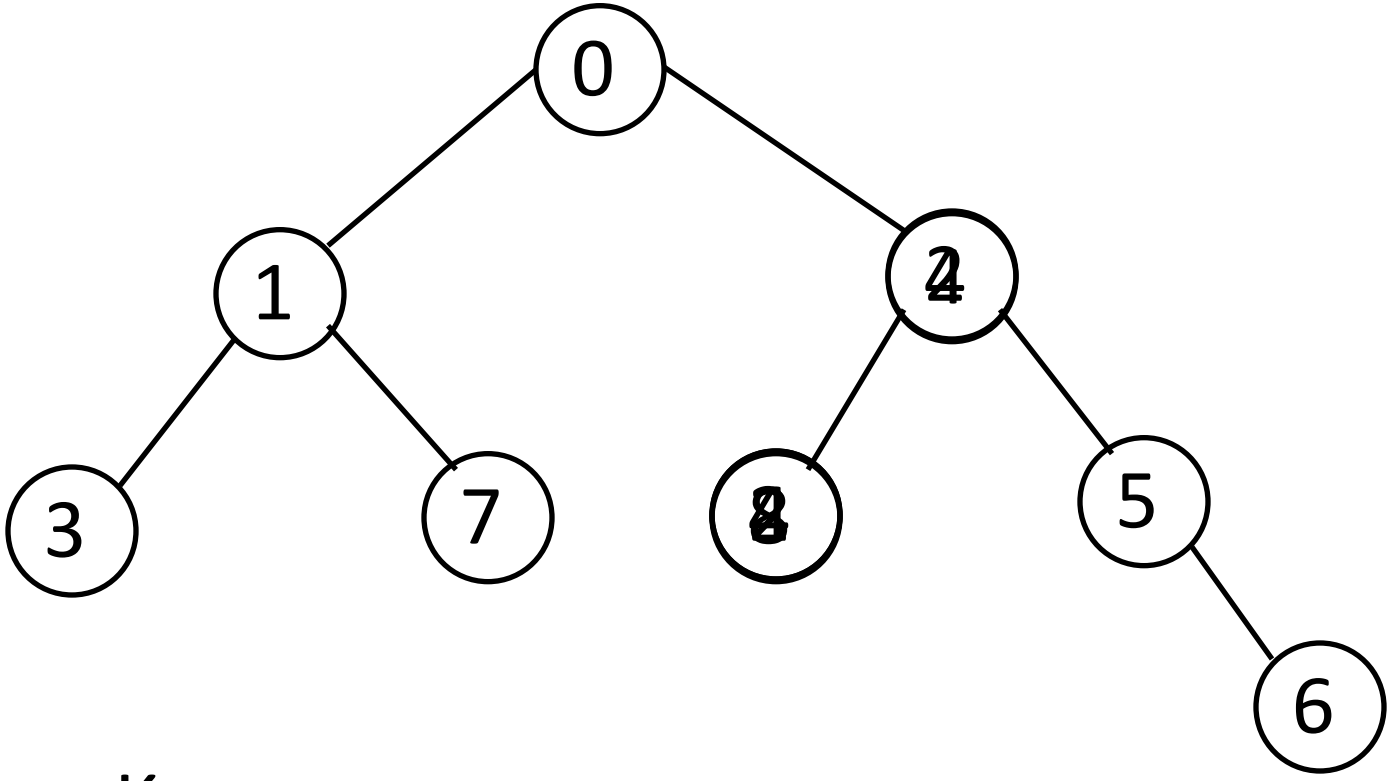
Insert



- Add key at bottom
- Bubble up

O(log(n)) time

Decrease Key



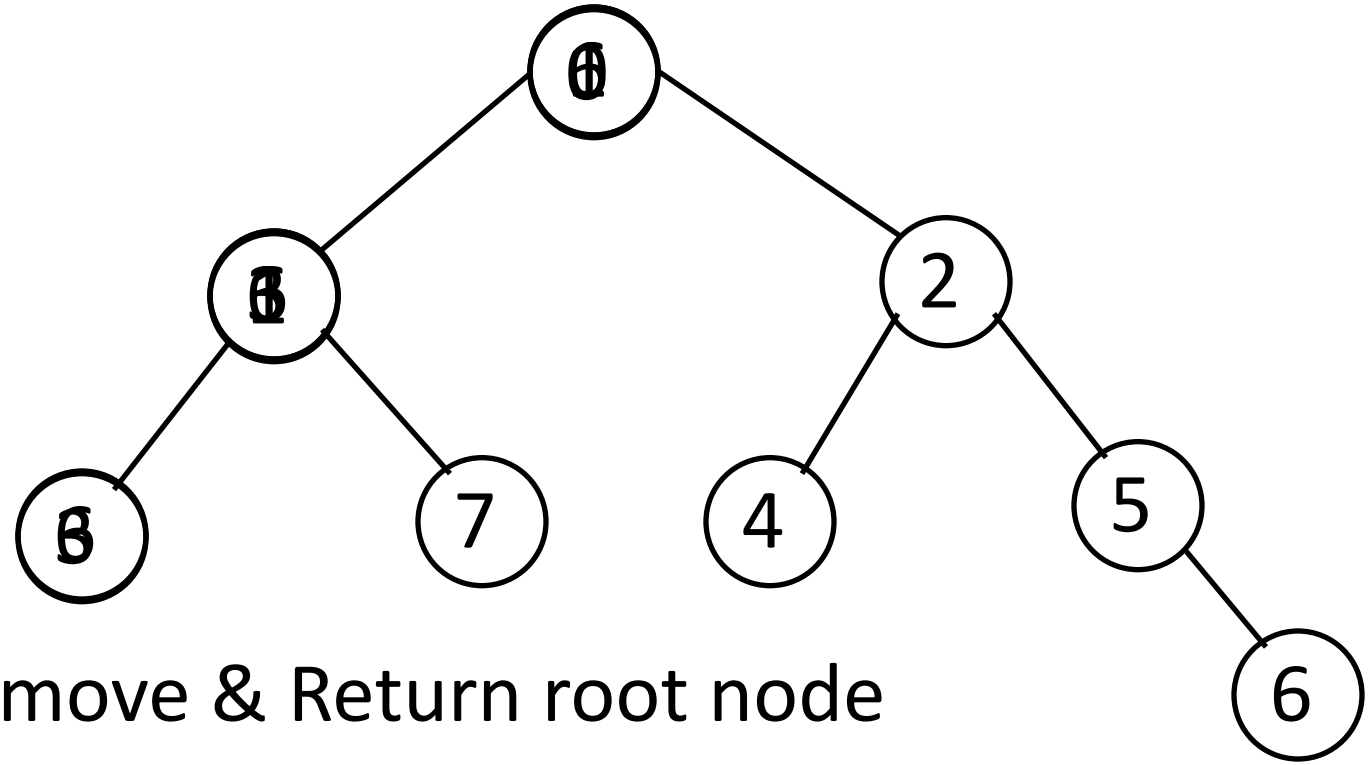
- Change Key
- Bubble up

O(log(n)) time

DeleteMin

Output:

0



- Remove & Return root node
- Move bottom to top
- Bubble down

Runtime O(log(n))

Summary

Runtime:

- Insert – O(log(n))
- DecreaseKey – O(log(n))
- DeleteMin – O(log(n))
- Dijkstra – O(log|V|(|V|+|E|))

Almost linear!