CS151 Intro to Data Structures

AVL
Splay Trees
Graphs

Announcements

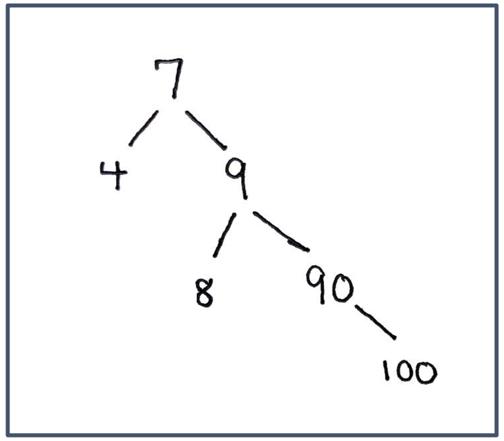
HW7 Due last night

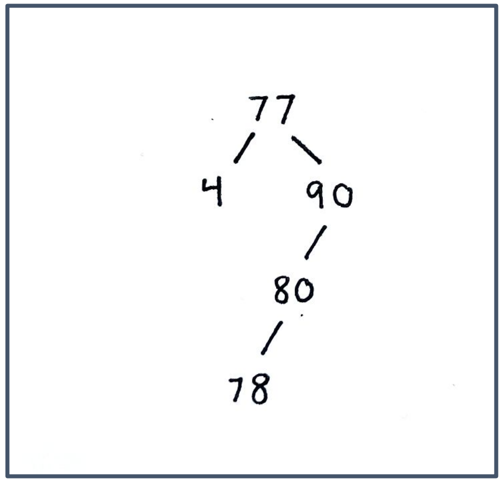
Two late days remaining

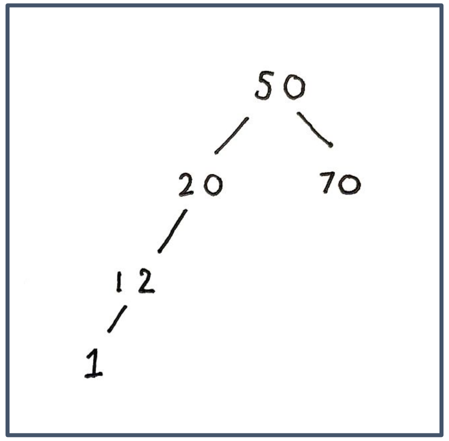
HW8 Due May 9th

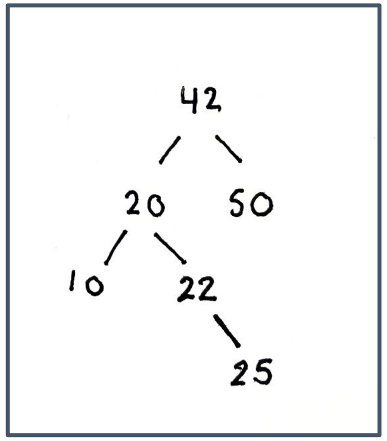
No office hours this Friday

Examples - which way should I rotate?









rotateLeft

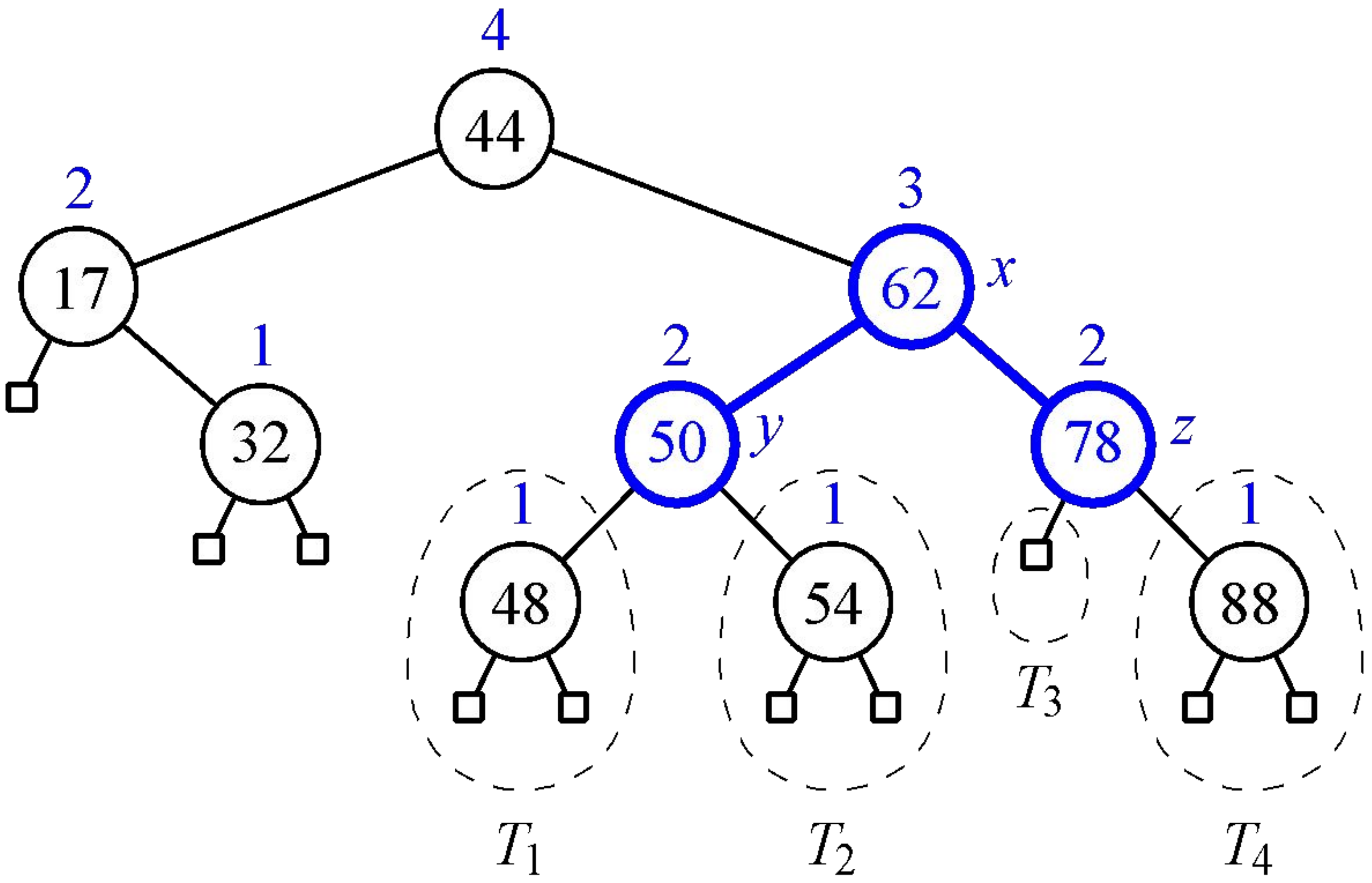
rotateRightLeft

rotateRight

rotateLeftRight

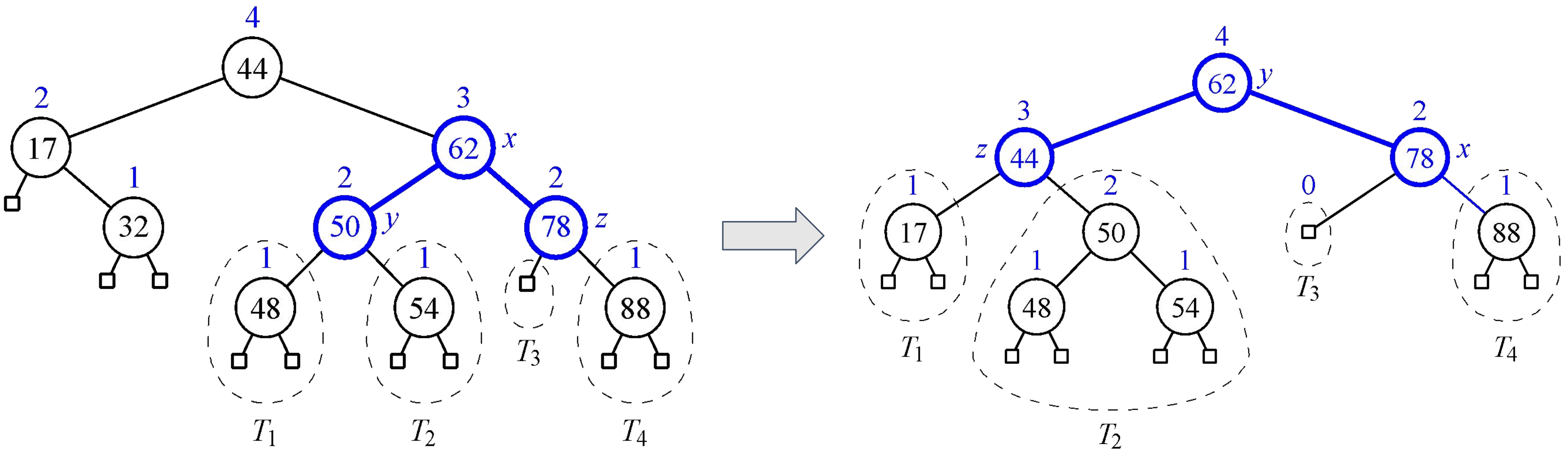
Deletion

Delete Example 1: 32

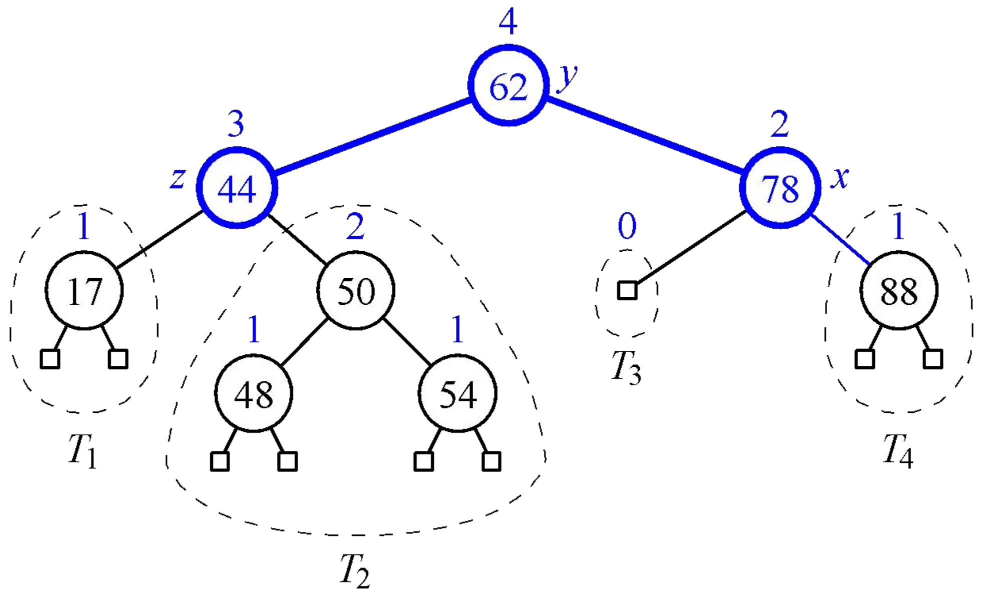


Delete Example 1: 32

rotateLeft

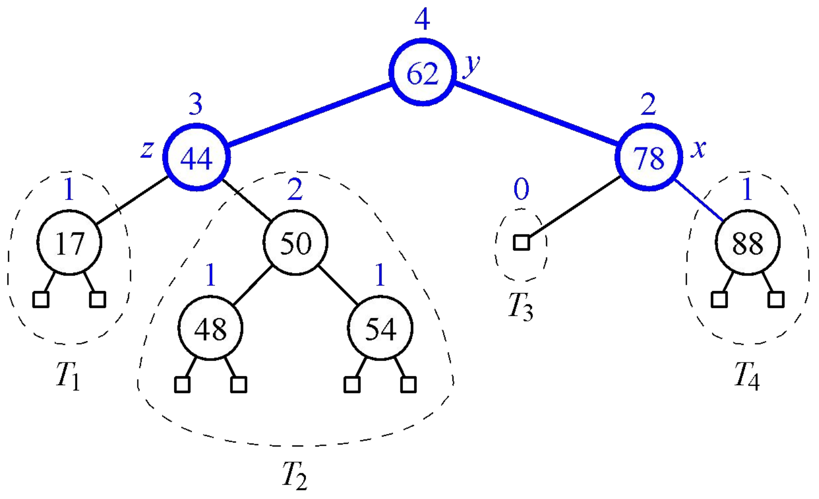


Delete Example 2: 88

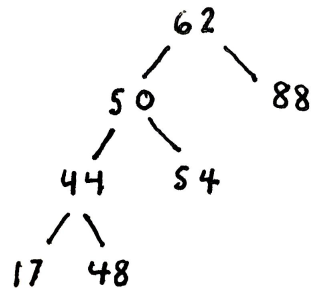


Delete Example 2: 88

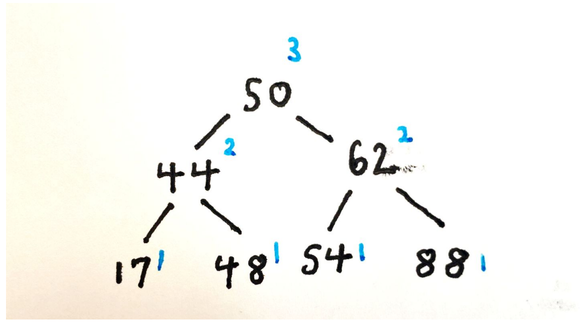
rotateLeftRight



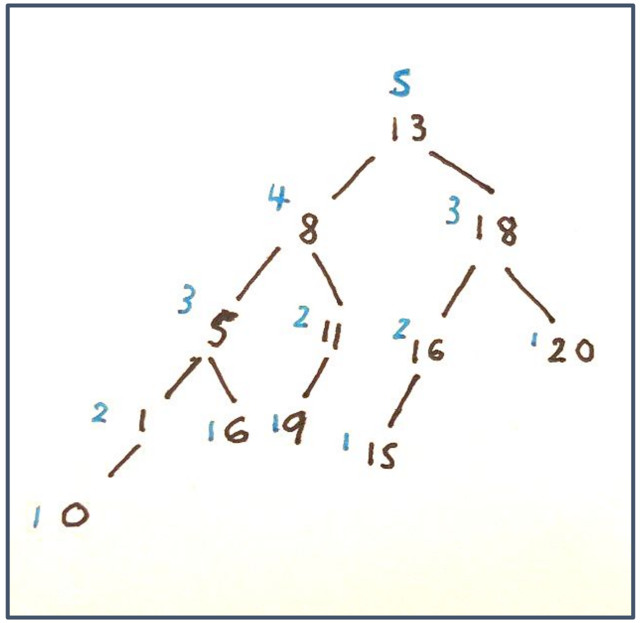
rotateLeft



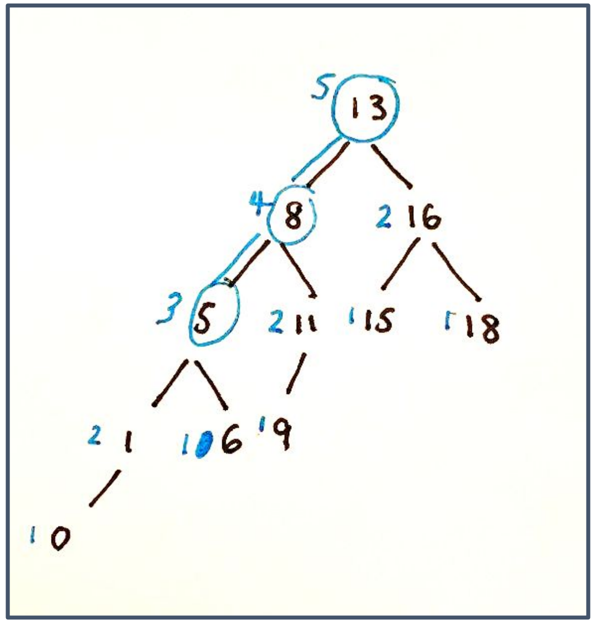
rotateRight



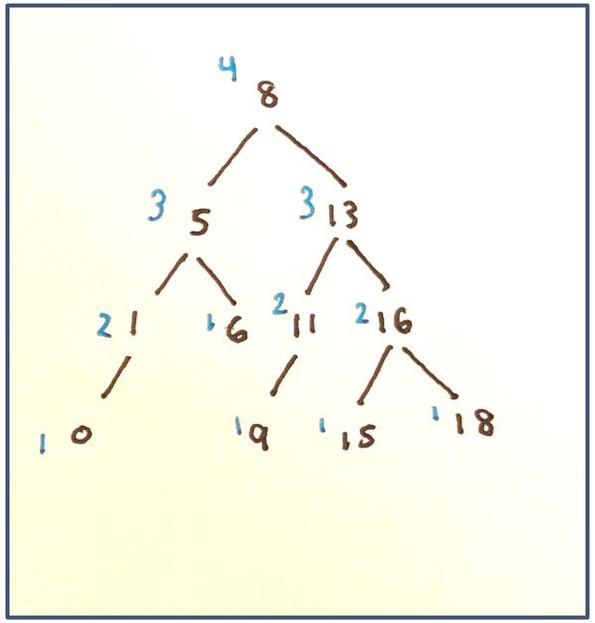
Delete Example 3: 20



rotateRight



rotateRight



Delete Example 3: 20

- Deletion can cause more than one rotation

- Worst case requires O(logn) rotations
  - deleting from a deepest leaf node and rotating each subtree up to the root

Removal

Runtime Complexity?

a. search + find node to rebalance + rotate
b. O(logn) + O(logn) + O(1) = **O(logn)**

Still O(logn) even though we may need multiple rotations?

Why?

-> Even though we may need to find multiple nodes to rebalance we only traverse the height of the tree once

Performance of BSTs

Runtime complexity:

search?
- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

insert?

- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

remove?

- BST:
  - O(n)
- AVL:
  - O(logn)

Splay Trees

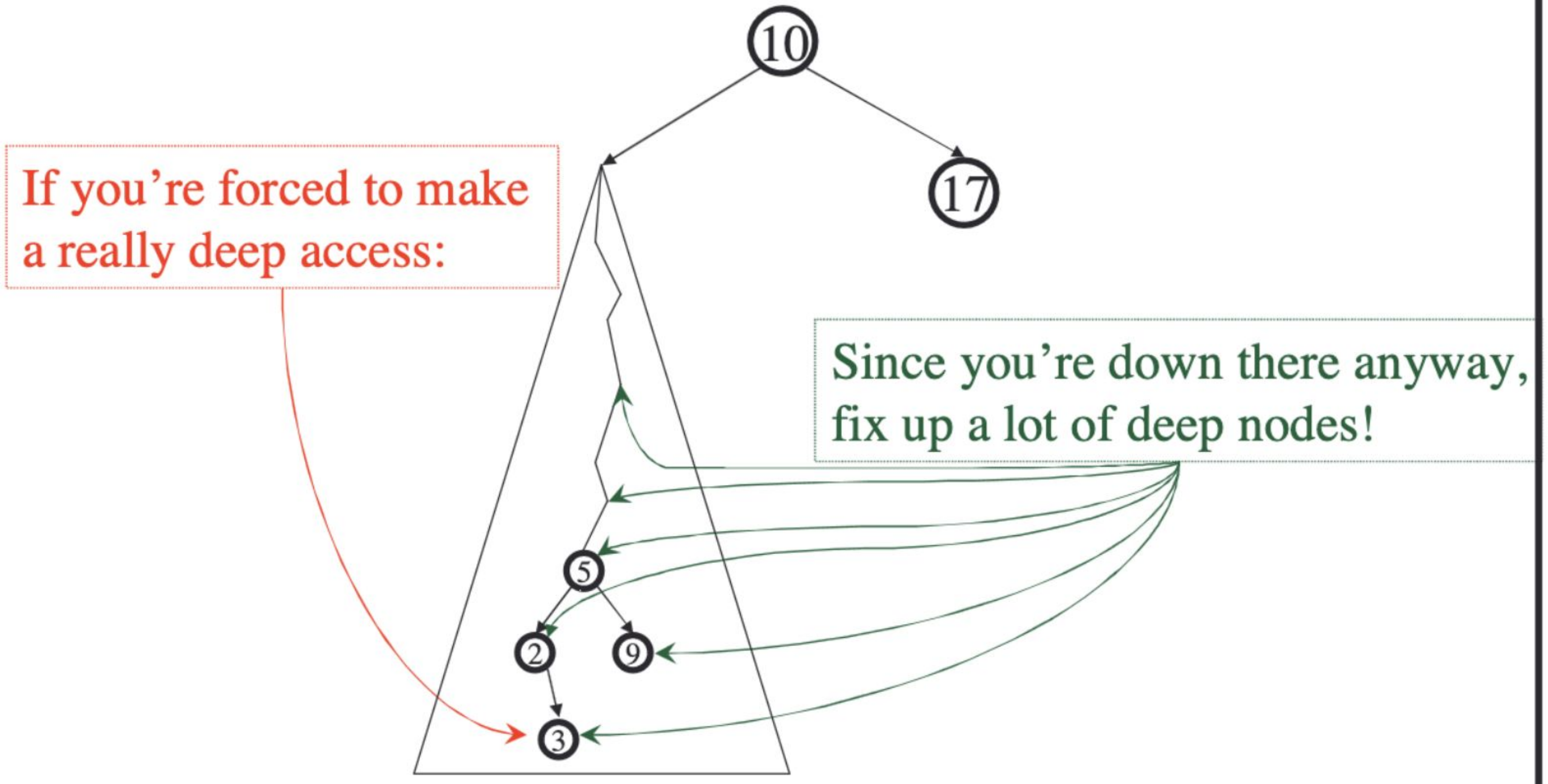
Splay Trees

- No enforcement on height

- Instead, exploits *principle of locality*
  - items that have been recently accessed are more likely to be accessed again in the near future

- “Move to root” operation
  - When a node is accessed (searched, inserted, or deleted), it becomes the root of the tree by performing a series of rotations called "*splays*"

The Splay Tree Idea



If you're forced to make a really deep access:

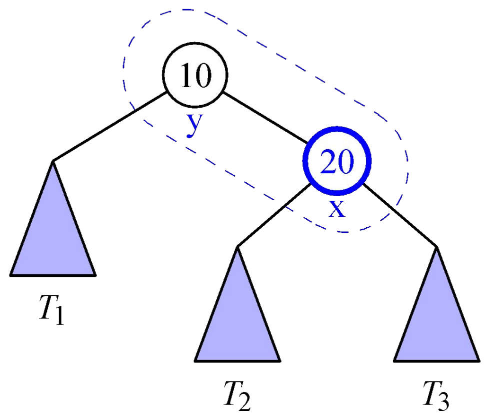
Since you're down there anyway, fix up a lot of deep nodes!

1

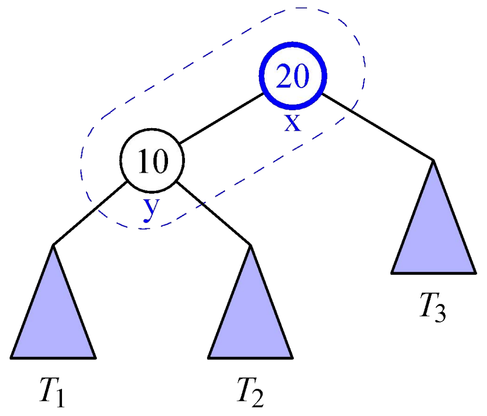
Splaying

- Move to root operation requires a **zig / zag** restructuring

- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



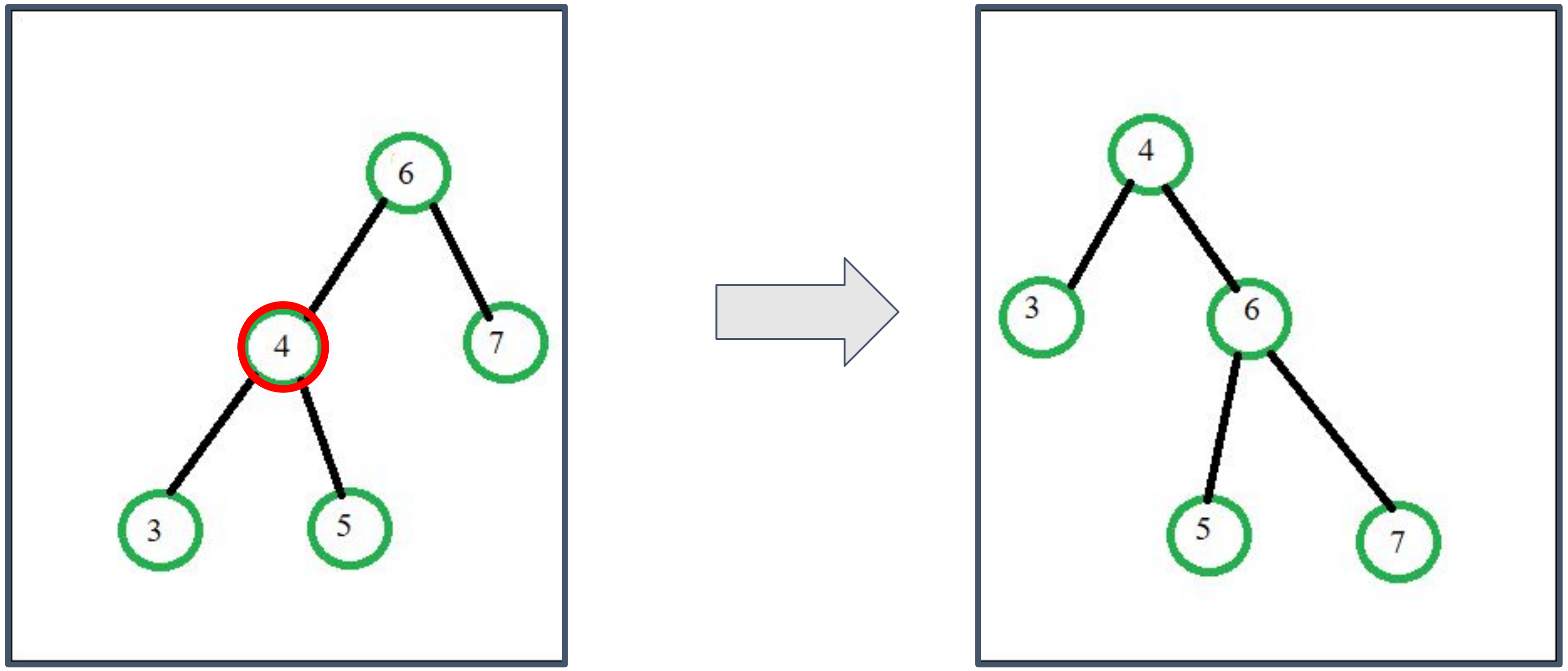
before



after

Splaying - Zig

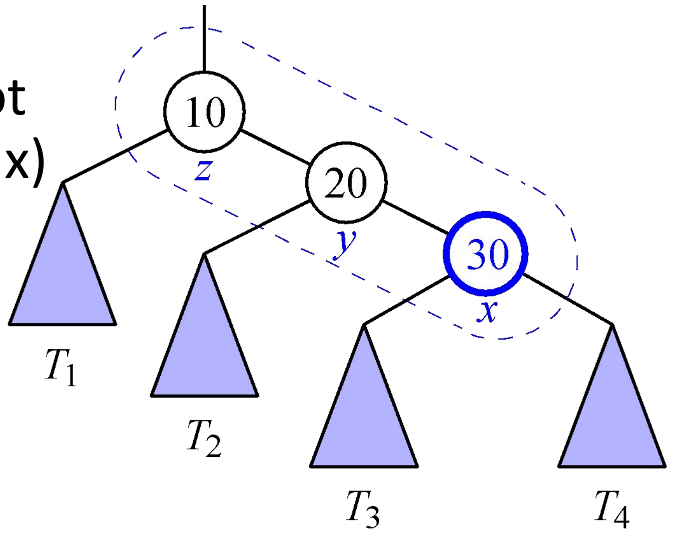
- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



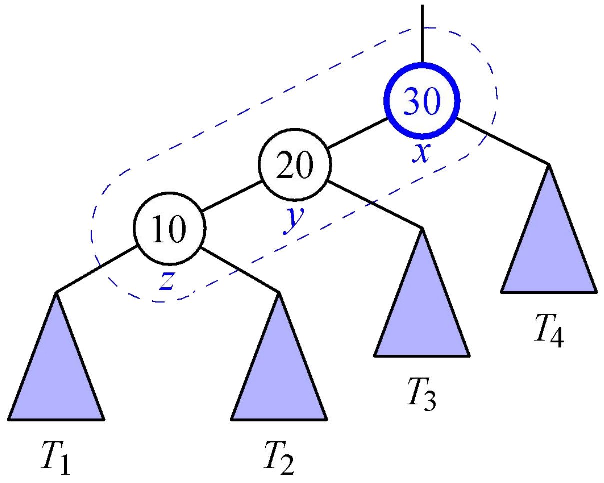
Splaying: Zig-Zig

zig-zig:

- step 1: **zig**
  a. accessed node's parent (y) becomes root
  b. parent (of y) becomes child (of y)
- step 2: **zig**
  a. accessed node (x) becomes root
  b. parent (of x) becomes child (of x)



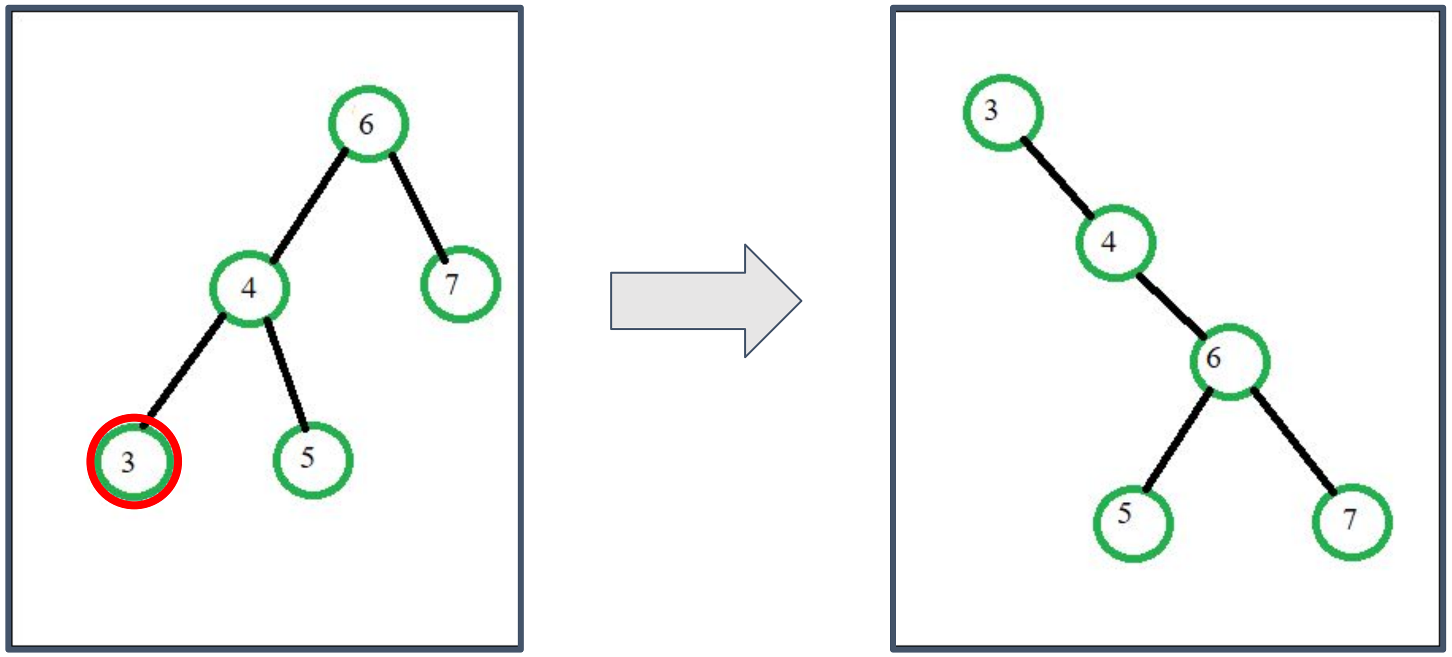
before



after

zig-zig:

- step 1: **zig**
  a. accessed node’s parent (4) becomes root
  b. parent becomes child
- step 2: **zig**
  a. accessed node (3) becomes root
  b. parent becomes child

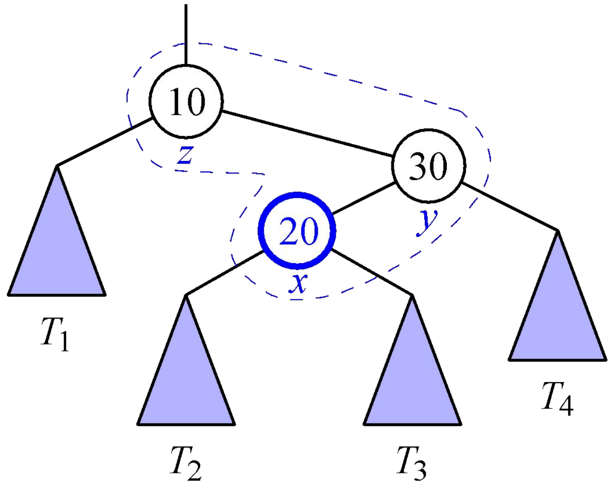


Splaying: Zig-Zag

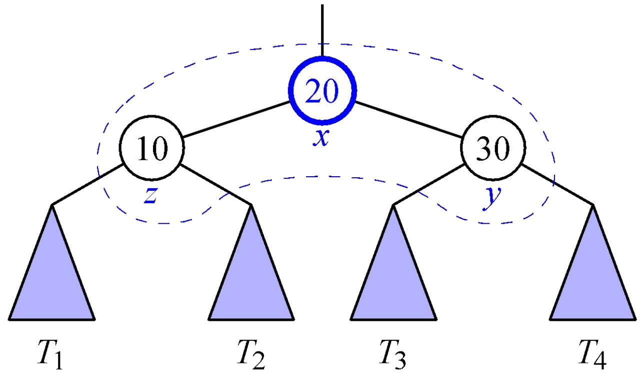
zig-zag:

- step 1: **zig**
  a. accessed node (x) becomes root of subtree
  b. parent (of x) becomes child (of x)
- step 2: **zag**
  a. accessed node (x) becomes root of tree
  b. parent (of x) becomes child (of x)

Called zig-zag because the second step is a rotation in the **opposite direction**

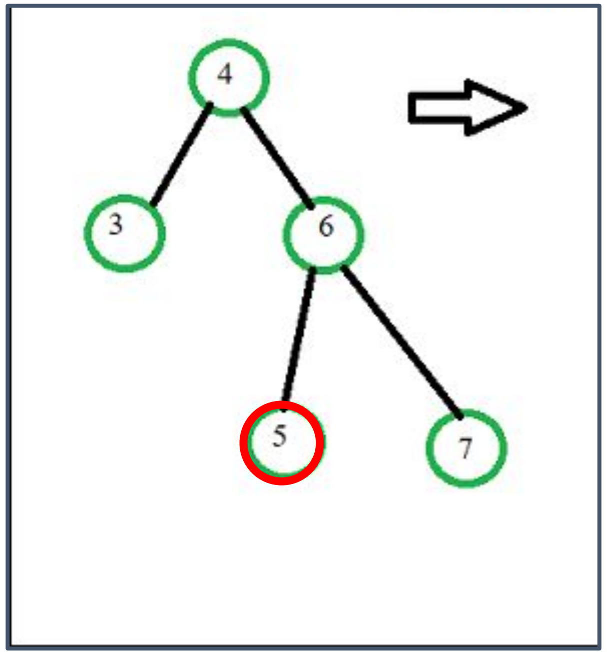


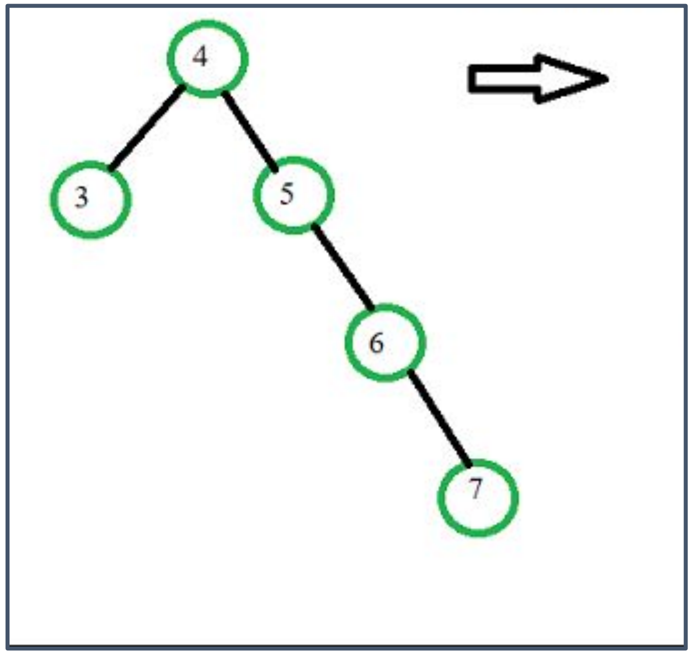
before

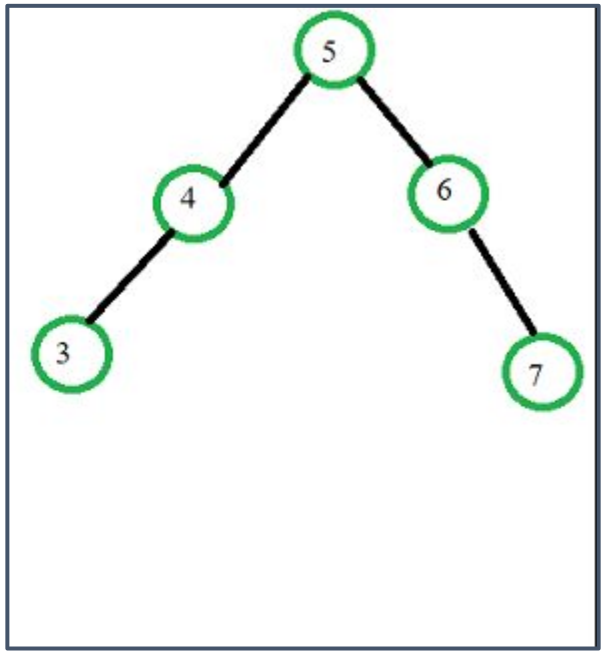


after

Splaying: Zig-Zag







Splaying: Zig-Zag and Zig-Zig

- Analogous to a double rotation in AVLs

- Zig-Zag
  - Two rotations in opposite directions

- Zig-Zig
  - Two rotations in the same direction

Which Transformation to Perform

1. **Zig**: accessed node does not have a grandparent. Only one rotation required

2. **Zig-Zig**: accessed node and its parent are both children on the same side
   a. x is the left child of y and y is the left child of z OR
   b. x is the right child of y and y is the right child of z

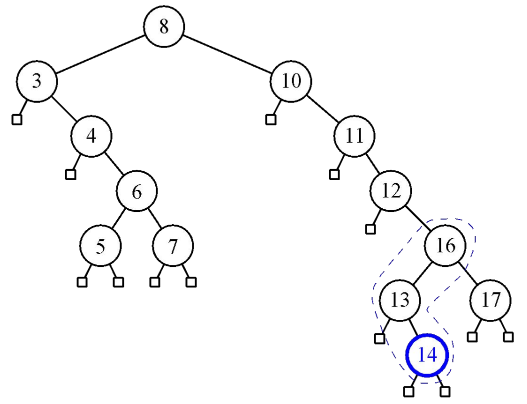
3. **Zig-Zag:** one of x and y is a right child and the other is a left child
   a. Analogous to double rotations in AVLs

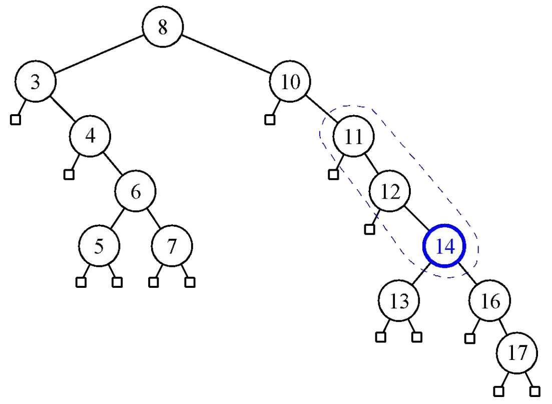
Splaying

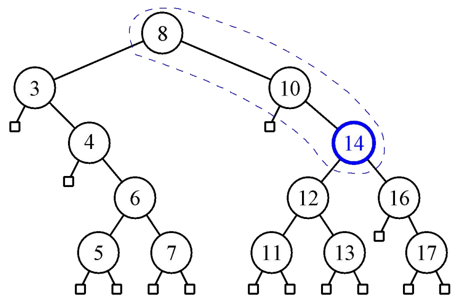
Repeating restructurings until the accessed node x is at the root of the tree.

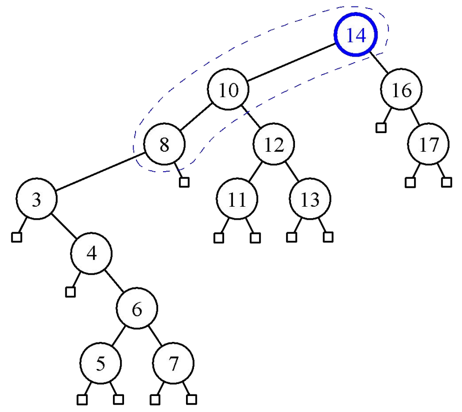
Series of zig, zig-zig, and zig-zag rotations

Example - insert(14)





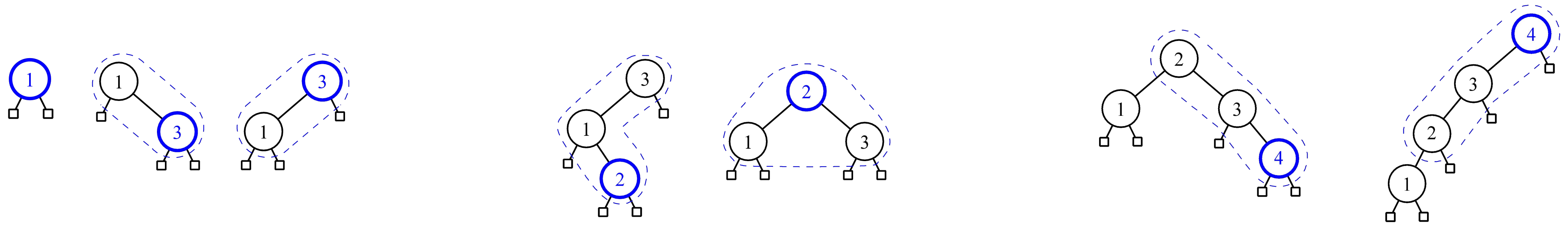




When/what to Splay

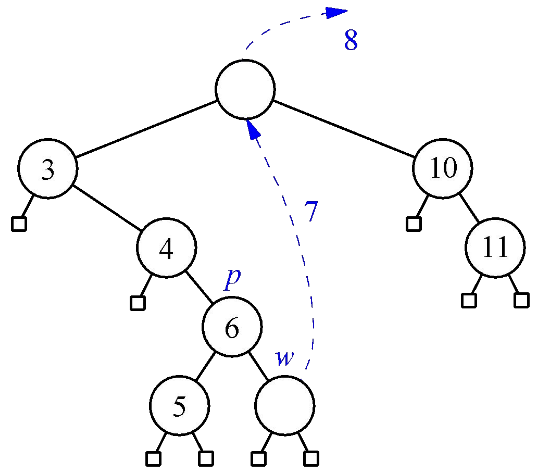
on search for x: if x is found, splay x. else splay x’s parent

on insert x: splay x after insertion



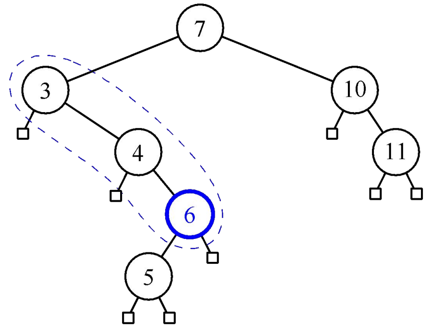
on remove x: splay parent of removed leaf node

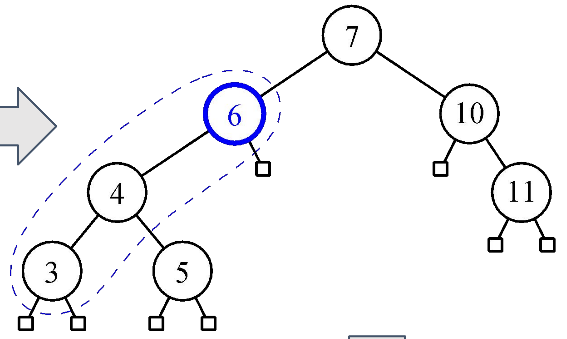
Deletion: remove(8)

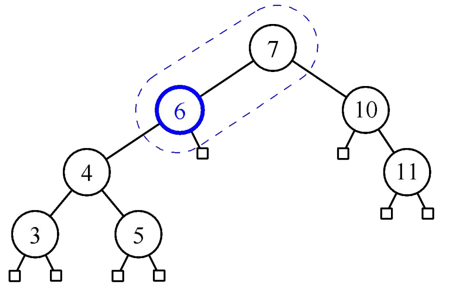


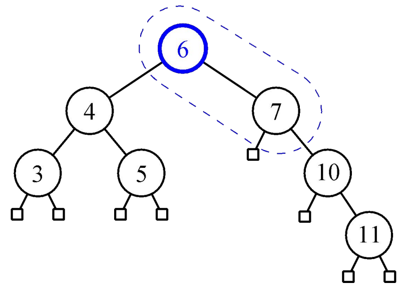
remove 8 and replace it with 7 (largest node on left)

splay 6 (parent of removed node)









Analysis of Splaying

Runtime of restructuring operations:

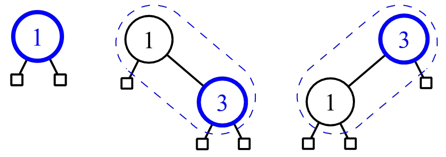
1. zig
   a. O(1)
2. zig-zig
   a. O(1)
3. zig-zag
   a. O(1)

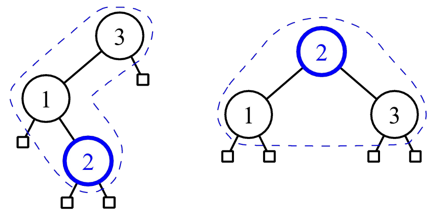
Analysis of Splaying

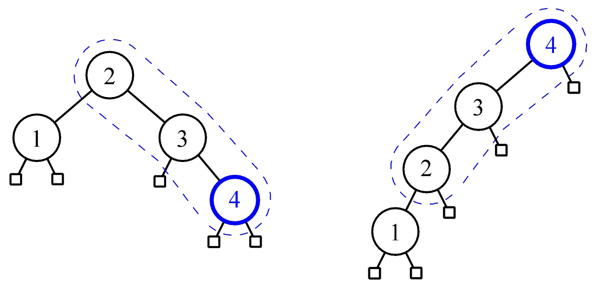
Splay trees do rotations after every operation (including search)

Each rotation is constant time..

What is the max number of rotations we may need to perform?







insert(0)

Analysis of Splaying

Each rotation is constant time..

What is the max number of rotations we may need to perform?

O(n)

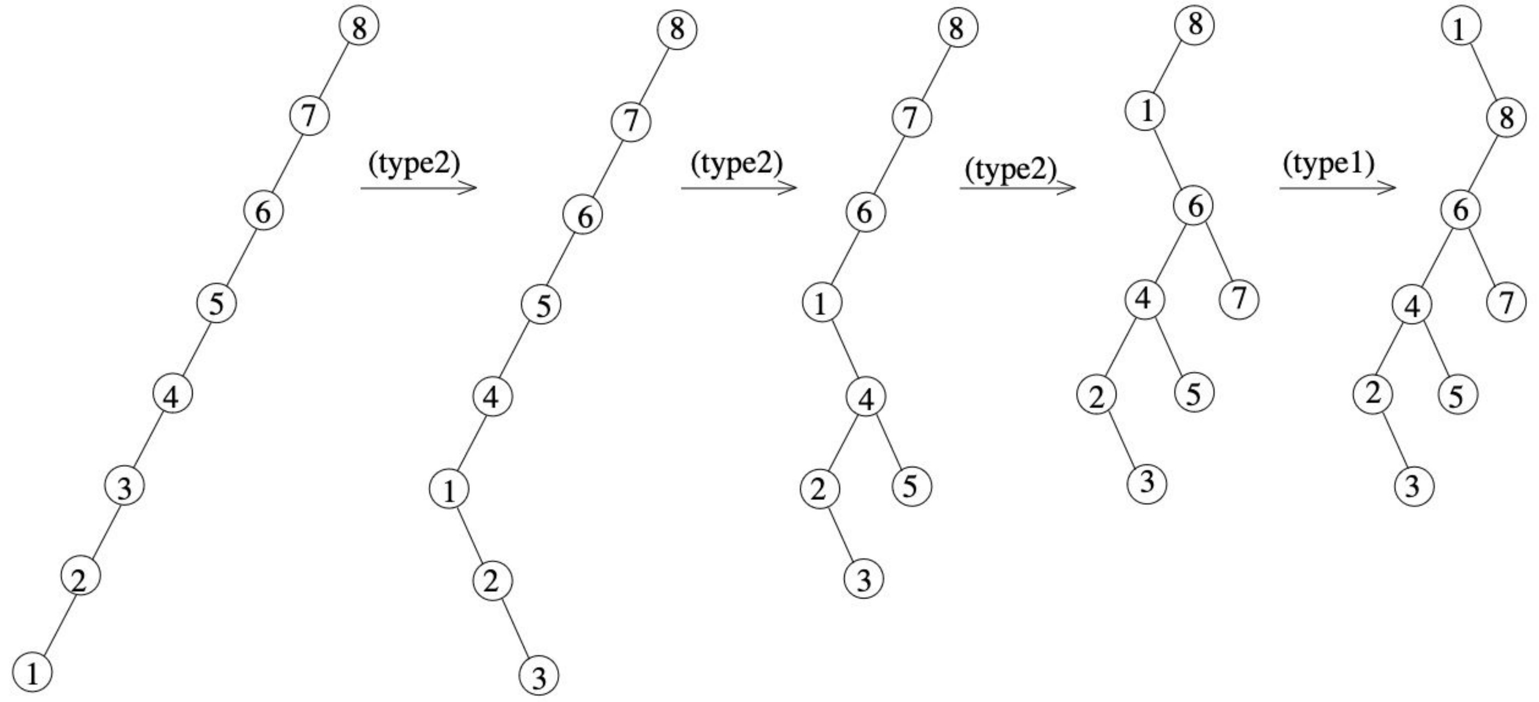
Analysis of Splaying

Worst case:

- Search:
  - O(n)
- Remove:
  - O(n)
- Insert:
  - O(n)

Analysis of Splaying

High cost operations often balance the tree
Amortized: O(logn)

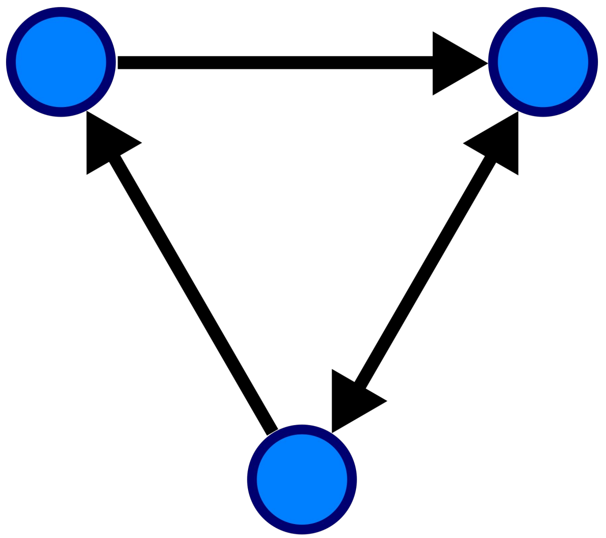


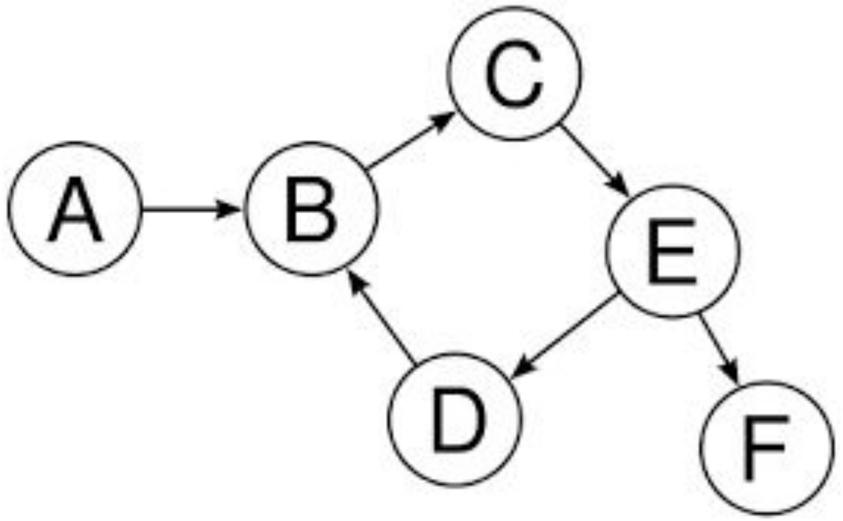
Graphs

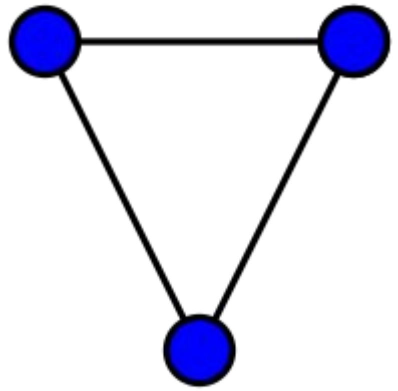
- **Terminology**
- Data Structures for Graphs
  - Adjacency Lists
  - Adjacency Matrix
- Traversals
- Shortest Paths
  - Djikstra’s Algorithm

Graphs

- A way of representing relationships between pairs of objects
- Consist of ***Vertices (V)*** with pairwise connections between them ***Edges (E)***
- A ***Graph G*** is a set of vertices and edges **(V, E)**

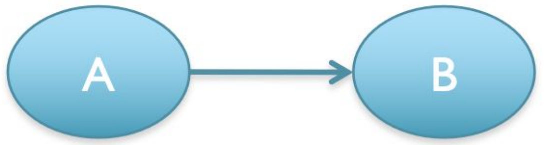




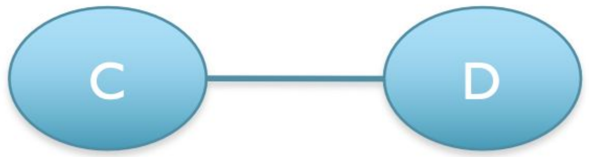


Edges

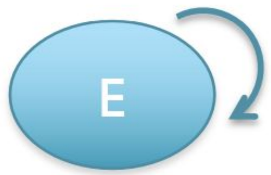
- An edge (u, v) connects vertices u and v
- Edges can be ***directed*** or ***undirected***
- An edge is said to be ***incident*** to a vertex if the vertex is one of the endpoints



Directed Edge

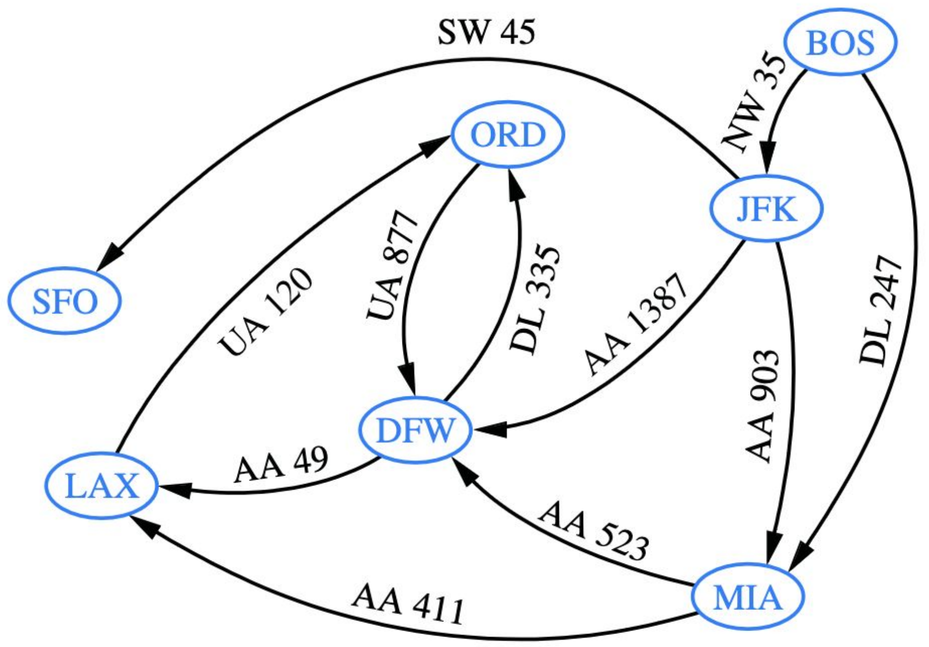


Undirected Edge



Self Edge
(Unusual but usually allowed)

Directed vs Undirected Graphs



Example of a directed graph representing a flight network.

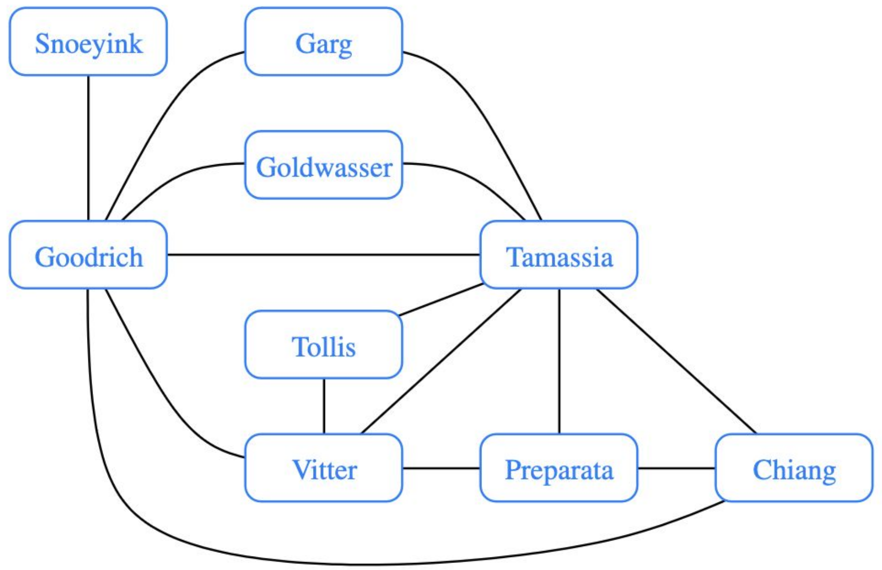


Figure 14.1: Graph of coauthorship among some authors.

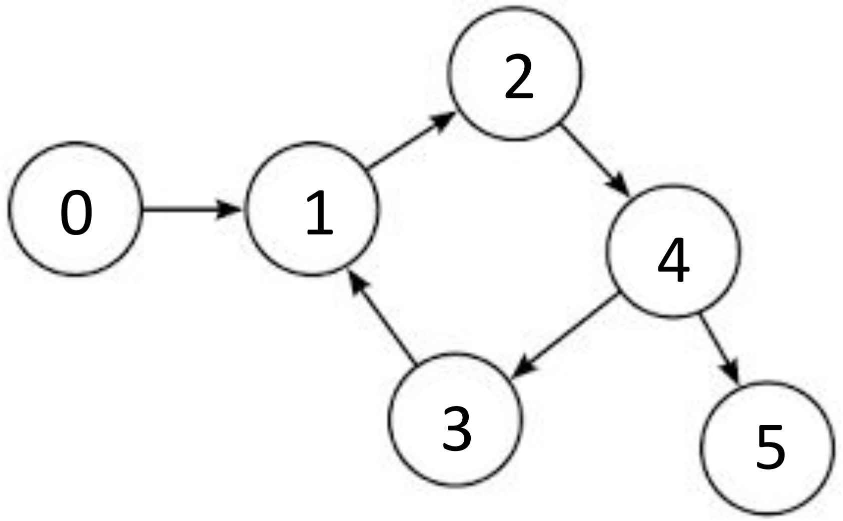
Graphs

- Terminology
- **Data Structures for Graphs**
  - Adjacency Lists
  - Adjacency Matrix
- Traversals
- Shortest Paths
  - Djikstra’s Algorithm

Representing a graph

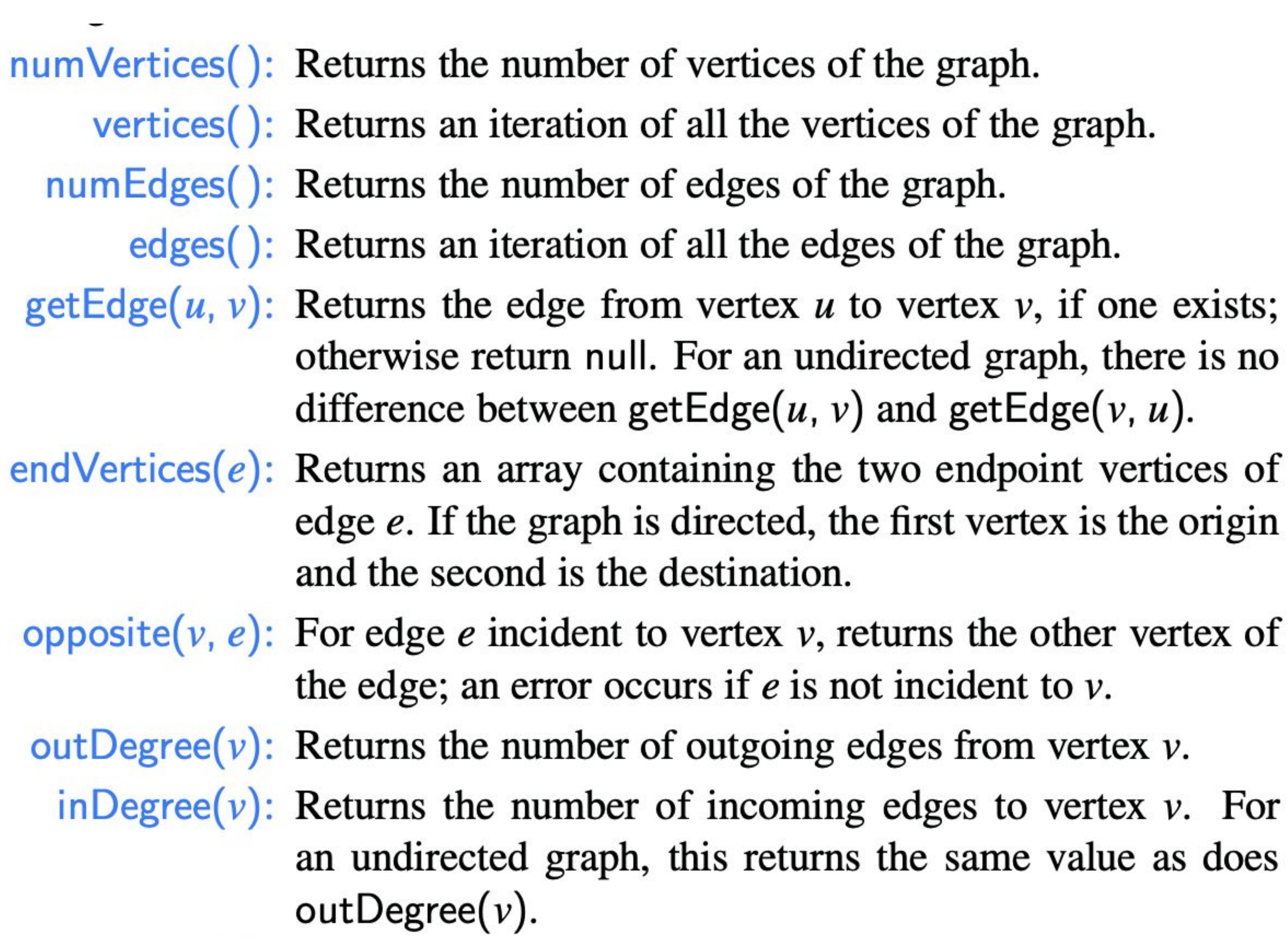
Adjacency List -

For each vertex v, we maintain a separate list containing the edges that are outgoing from v



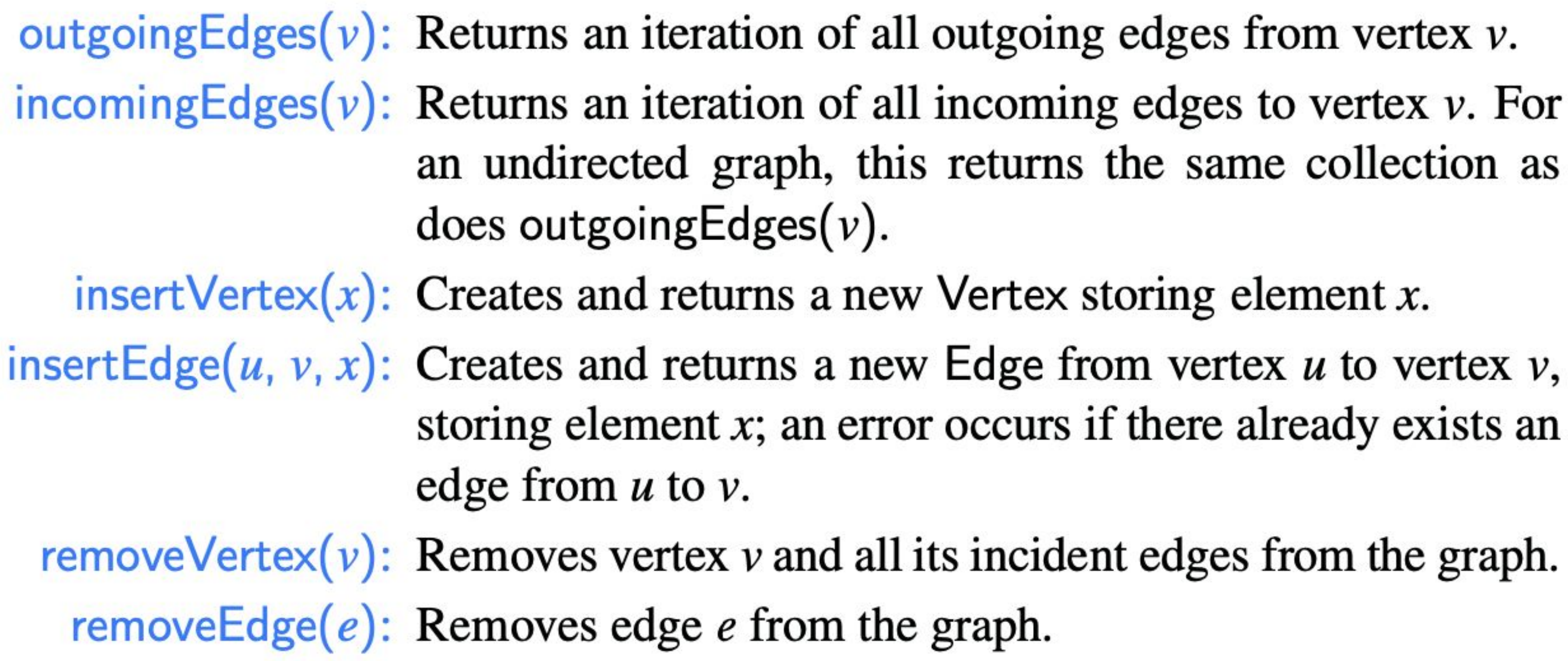
- Each index in the array represents a vertex

Graph ADT



numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

Graph ADT



outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Representing a graph

Let’s implement a graph as an Adjacency List

Representing a graph - Adjacency List

Runtime Complexity: (In terms of V and E rather than n)

- addVertex:
  - O(V)
- addEdge:
  - O(E)
- removeVertex:
  - O(V*E)
- removeEdge:
  - O(E)

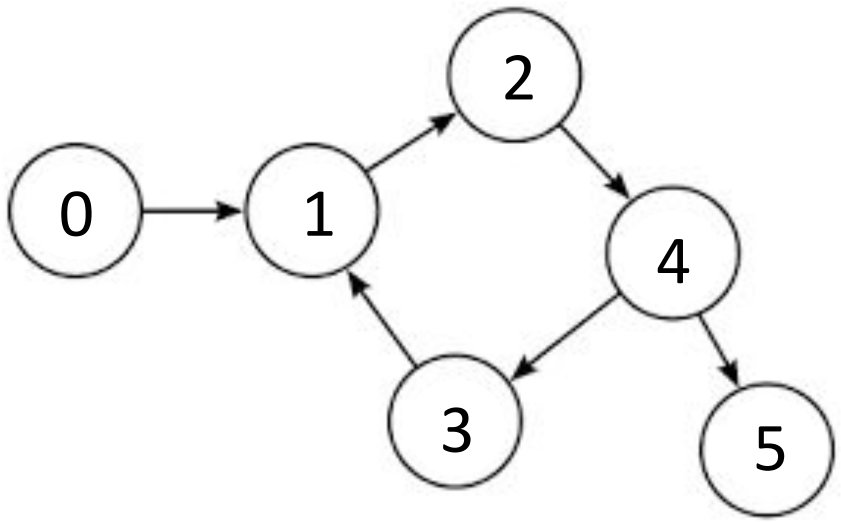
Representing a graph

Adjacency Matrix -

each index in the array is another array

Maintains an VxV matrix

where each slot (i,j) represents an outgoing edge from i to j



	1				
		1			
				1	
	1				
			1		1

Representing a graph

Let’s implement a graph as an Adjacency Matrix

Representing a graph - Adjacency Matrix

Runtime Complexity: (In terms of V and E rather than n)

- addVertex:
  - O(V)
- addEdge:
  - O(1)
- removeVertex:
  - O(V)
- removeEdge:
  - O(1)

Graphs

- Terminology
- Data Structures for Graphs
  - Adjacency Lists
  - Adjacency Matrix
- **Traversals**
- Shortest Paths
  - Djikstra’s Algorithm

Reachability

Reachability is determining if there exists a path between two vertices in a graph

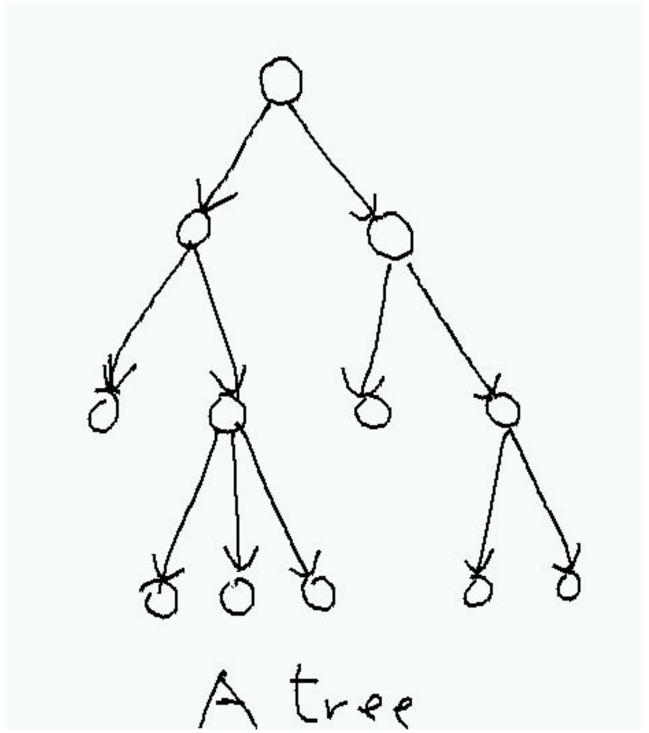
Common questions about graphs involve **Reachability**

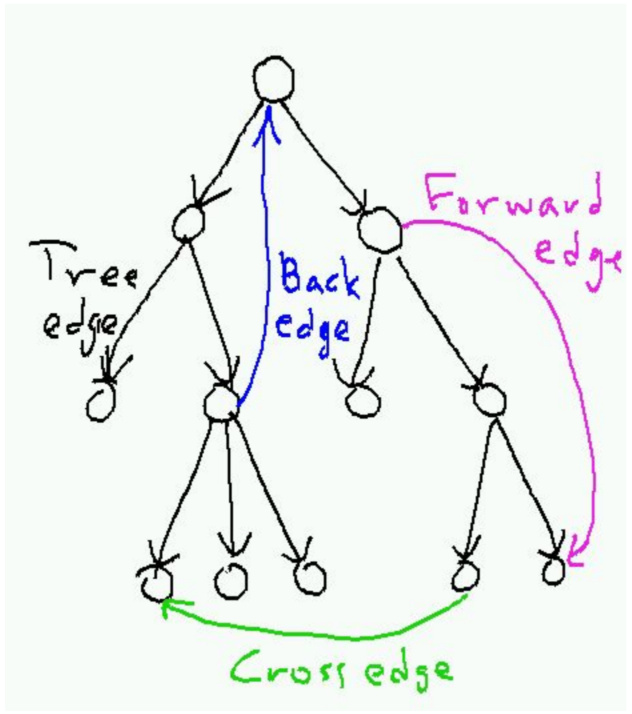
- Does a path exist from vertex *u* to vertex *v*?
- Find all vertices that are reachable from *v*

Depth First Traversal

```
void DFS(root) {
    for each child of root:
        DFS(child)
}
```

Does this work for graphs?

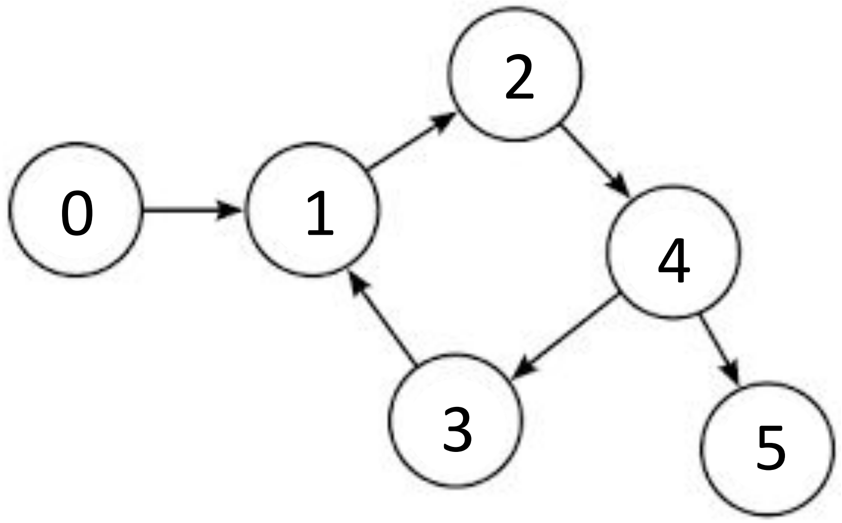




Depth First Traversal

How can we modify the code to deal with cycles?

```
void DFS(root) {
    for each child of root:
        DFS(child)
}
```



Keep track of what we've already visited!

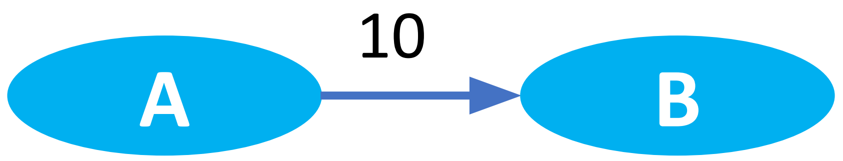
Let's code this for a Matrix Graph

Graphs

- Terminology
- Data Structures for Graphs
    - Adjacency Lists
    - Adjacency Matrix
- Traversals
- **Shortest Paths**
    - Djikstra’s Algorithm

Weighted Graphs

Edges have weights/costs



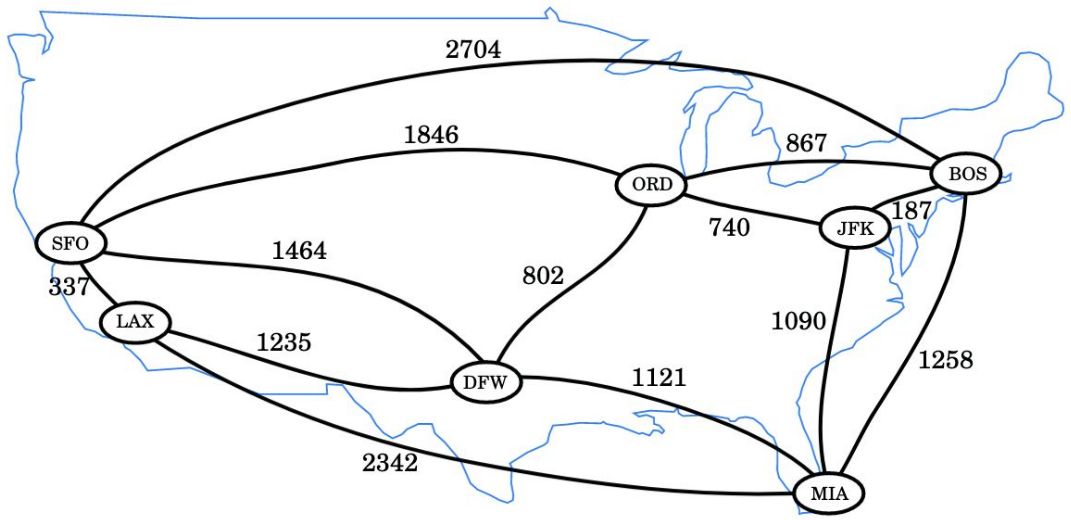


Figure 14.14: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum-weight path in the graph from JFK to LAX.

Shortest Paths

A **path** is defined as a set of edges

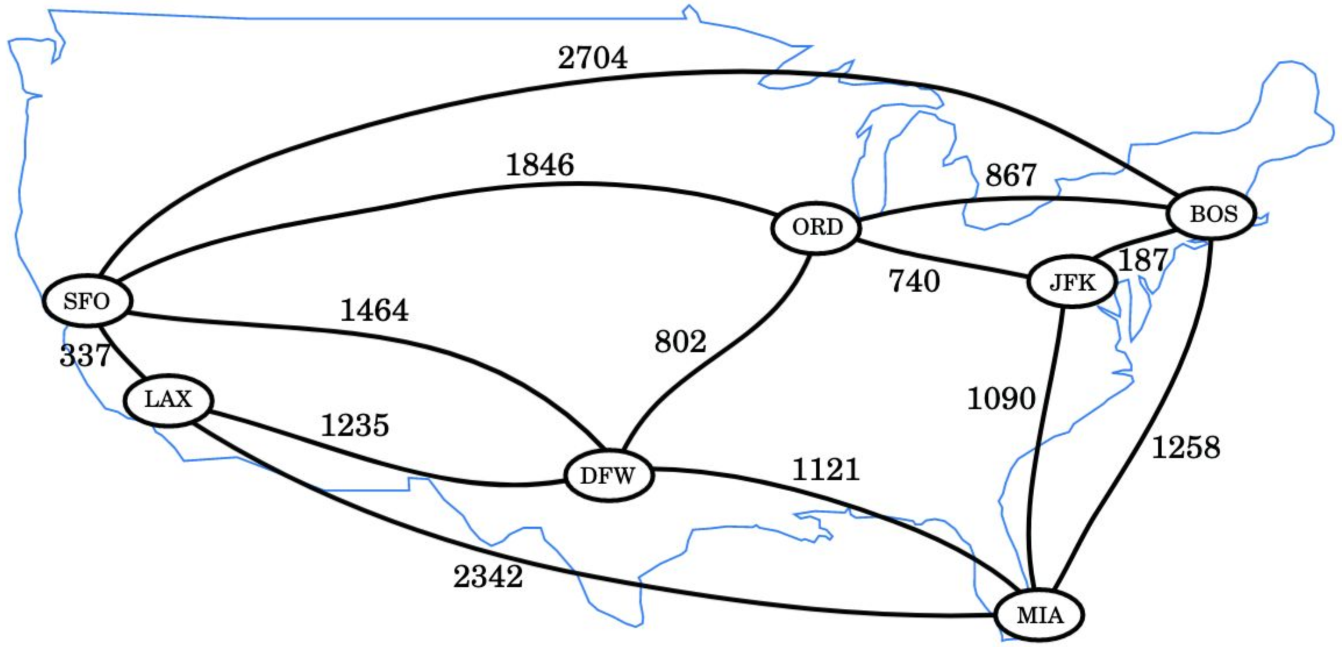
$$P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$$

The *length* of a path is the sum of the weights of the edges

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

Shortest Paths

What is the length of the path P = ((SFO, DFW), (DFW, MIA), (MIA, JFK))



Shortest Paths

What is the shortest path from SFO to JFK?

There are many possible paths...

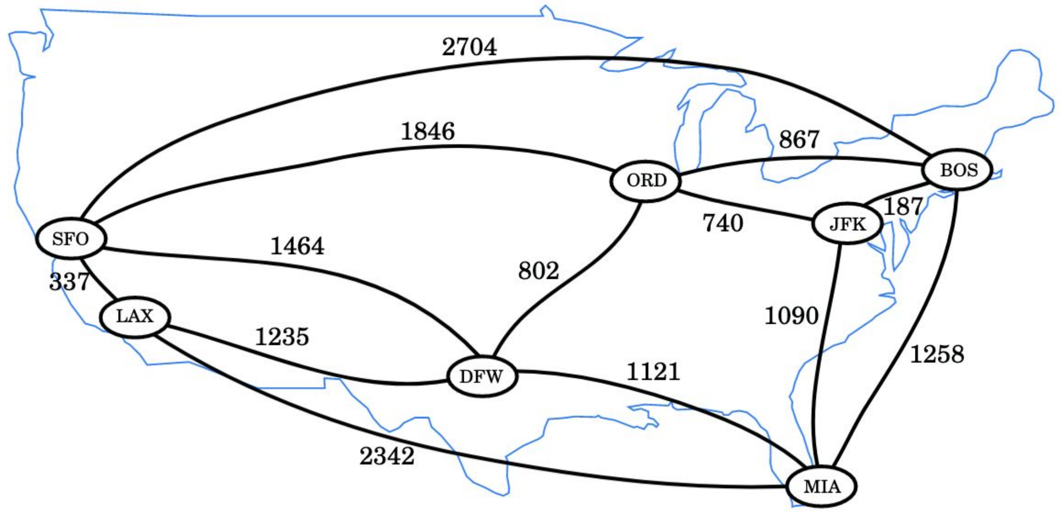
((SFO, ORD), (ORD, JFK))

((SFO, LAX), (LAX, MIA), (MIA, JFK))

((SFO, BOS), (BOS, JFK))

....

((SFO, DFW), (DFW, ORD), (ORD, JFK))

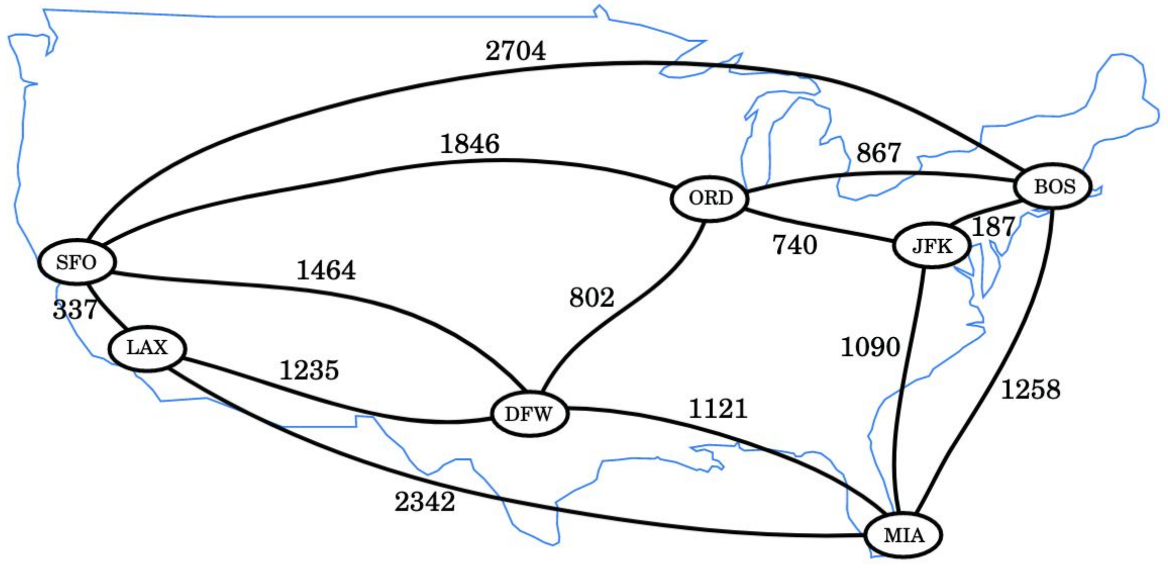


Dijkstra's algorithm

- graph search algorithm that finds the shortest path between nodes in a weighted graph

- maintains a set of vertices whose shortest distance from the *source* has already been determined
    - uses a *min heap* to select the vertex with the smallest distance

Dijkstra's algorithm



1. init:
    a. assign a init distance for each node
    b. create a min-heap with source
2. while heap is non-empty:
    a. poll node p
    b. For each neighboring node not yet visited:
        i. distance of neighbor = dist(p) + weight of edge (p, neighbor)
        ii. if neighbor == dst: return dist
        iii. If this distance is less than the current dist, update it.
    c. update the heap if distances changed