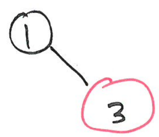
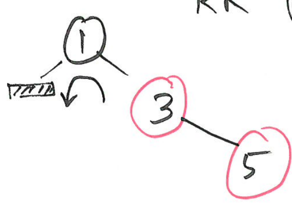
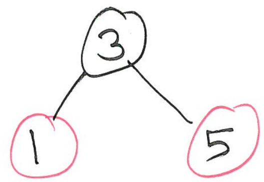
Add: 1 3 5 7 9 6 4 2

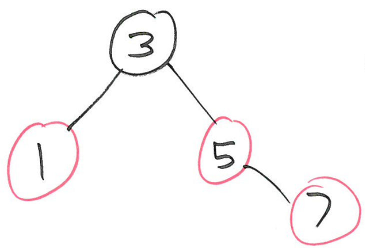


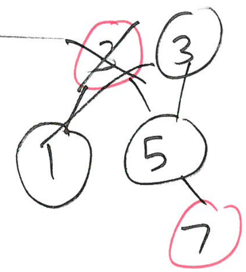
RR (Case 2b)



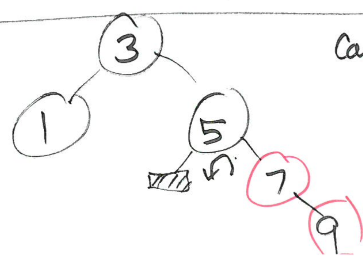


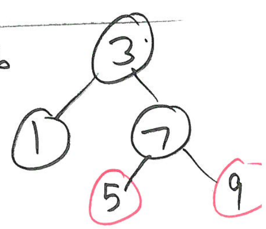
Case 1

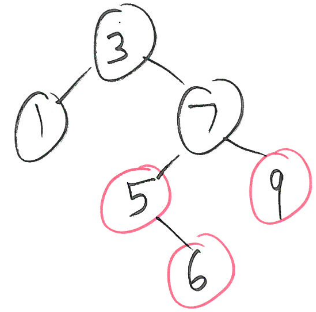




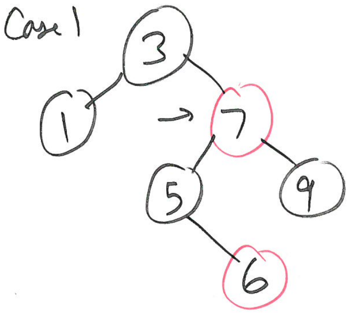
Case 2b
RR

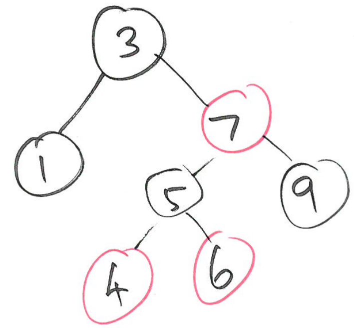


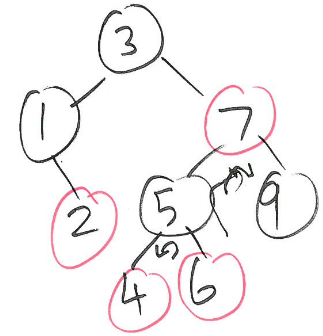




Case 1

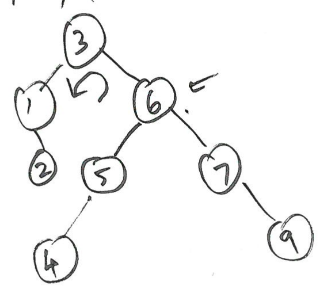




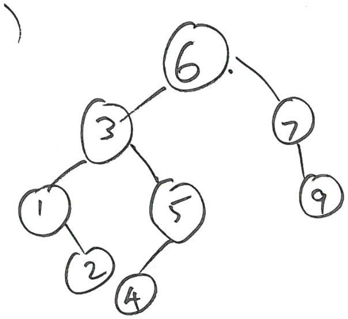


Splay (6):

Zig-Zag (LR)



Zig (Left)



LP3

Q: Is it necessary to check if root can reach all vertices of G at the start of the algorithm?

A: Yes. If r cannot reach all vertices of G, there is no spanning tree rooted at r. Raise exception or return null after printing error message.

Q: How can it be verified if the output of our algorithm is correct?

A: At the end, we have a spanning tree rooted at r, and Δ_u for all nodes of G and the new vertices V' added to V, representing shronked scc.

steps:

1. check if t is a spanning tree rooted at r
2. $\sum_{e \in t} w(e) = \sum_{u \in V \cup V'} \Delta_u$
3. For all edges $e \in G$, $e = (u,v)$: e.weight $\geq \sum_{v \in X} \Delta_x$ ← Delta of those super nodes that contain v and Δv.
4. For $e \in t$: $e.\text{weight} = \sum_{v \in X} \Delta x$

Finding odd-length cycles using BFS

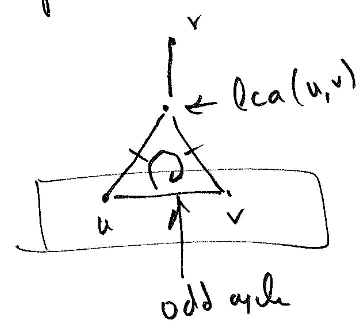
Fact: $(u,v) \in G \Rightarrow \begin{cases} u.d = v.d + 1 \\ u.d = v.d \\ u.d = v.d - 1 \end{cases}$

Undirected graph   BFS.

Let G be connected.

Run BFS from some node as source.

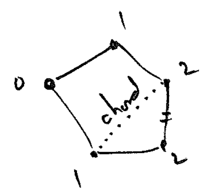
G is not bipartite $\Leftrightarrow \exists\, u, v$: $u.d = v.d$, $(u,v) \in E$



$u \ldots lca(u,v) \ldots v.u$

Finding shortest odd-length cycle:

Fact: A shortest odd-length cycle does not have a chord.



Run BFS from a node on the shortest odd-cycle to find shortest odd cycle.

Shortest paths:
Input: Graph G = (V, E) (usually, directed), source vertex s ∈ V, edge weights $w : E \to \mathbb{Z}$ (more generally, $\mathbb{R}$).

Weight (or length) of a path P, $w(P) = \sum_{e \in P} w(e)$. A cycle C is called a negative cycle if w(C) < 0.

Output: For each u ∈ V, find a simple path from s to u, of minimum weight.

Overview of shortest path algorithms

Algorithm	Condition	Class of graph	Running time
Breadth-First Search (BFS)	No weights on edges	Directed or undirected	$E + V$
DAG-shortest-path	No cycles	DAG	$E + V$
Dijkstra's algorithm	No negative edges	Directed or undirected	$E \log V$
Bellman-Ford algorithm	No cycles of negative length	Directed	$E\ V$

Breadth-First Search (BFS): Find shortest number of hops from a source s to all nodes of G.

```
bfs(g, s):
    for u ∈ V do { u.d ← ∞;   u.π ← null;   u.seen ← false }
    Create a queue q of vertices
    s.d ← 0;   s.seen ← true;   q.add( s )
    while q is not empty do
        u ← q.remove( )
        for edge (u, v) ∈ E do
            if ! v.seen then
                v.d ← u.d+1;   v.π ← u;   v.seen ← true;   q.add( v )
```

Applications of BFS: (1) Broadcast trees, (2) Test if an undirected graph is bipartite, (3) Find diameter of an unrooted tree, (4) Find shortest paths in graphs whose edges have small integer weights, (5) Find an odd-length cycle in a non-bipartite undirected graph, (6) Find a shortest odd-length cycle of an undirected graph, (7) Used as a subroutine in maximum flow algorithms of Edmonds and Karp, and, Dinitz.

Basis of all shortest path algorithms: subpath of a shortest path is a shortest path. In other words, if a shortest path from s to v is composed of a path from s to u and the edge (u,v), then $\delta(s,v) = \delta(s,u) + w(u,v)$.

Therefore, shortest paths can be encoded as an up-tree, where each node stores its predecessor in a shortest path from s to that node. In the example above, we can set $v.\pi = u$. The following utility functions are used by all shortest path algorithms that use edge weights:

```
initialize( s ):
    for u ∈ V do
        u.d ← ∞
        u.π ← null
        u.seen ← false
    s.d ← 0
```

```
boolean relax( e ):
    u ← e.from;   v ← e.to
    if v.d > u.d + e.weight then
        v.d ← u.d + e.weight;   v.π ← u
        return true
    return false
```

DAG-shortest-paths algorithm:In a DAG, the nodes in any path are in strictly increasing order of their topological numbers, in any topological ordering of V. This can be exploited to design the following efficient algorithm for shortest paths in DAGs:

// Pull algorithm: difficult to code without revAdj:
dagSP(g, s):
    Find a topological ordering of g
    initialize(s)
    for u ∈ V in topological order do
        // LI: All predecessors of u are done
        for edge e = (p, u) into u do
            relax(e)
	// Push algorithm: much nicer code
	dagSP(g, s):
	    Find a topological ordering of g
	    initialize(s)
	    for u ∈ V in topological order do
	        // LI: u.d = δ(s, u)
	        for edge e = (u, v) out of u do
	            relax(e)

Dijkstra's algorithm: applicable in graphs without any edges of negative weight. Idea:

*Maintain a set of nodes S for which shortest paths are known.

*For v ∈ V − S, store in v.d, the length of a shortest path from s to v that goes through only nodes of S.

*In each iteration, select a node u in V − S with minimum u.d, and add it to S.

*Relax edges out of u to update distance estimates of other nodes in V − S.

Code resembles Prim's algorithm that uses indexed priority queues:

```
dijkstraSP( G, s ):  // Implementation #2 using indexed priority queue of vertices
    // v ∈ V − S stores in v.d, the weight of a shortest path from s to v that goes through only nodes of S
    //
   initialize( s )    // for u ∈ V do { u.seen ← false;   u.π ← null;   u.d ← ∞ }
   s.d ← 0
   q ← new indexed priority queue of vertices with u.d as priority of u
   while q is not empty do
        u ← q.remove( )
        u.seen ← true
        for Edge e = ( u, v ) ∈ E  do
            changed ← relax( e )
            if changed then
                q.decreaseKey( v.getIndex( ) )
```