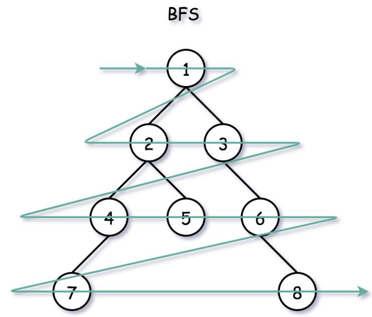
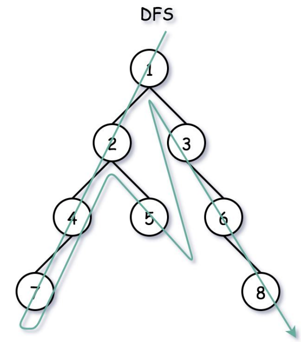
Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	O(V³)
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general



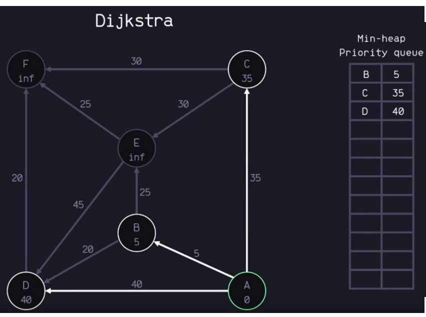
O(V + E)

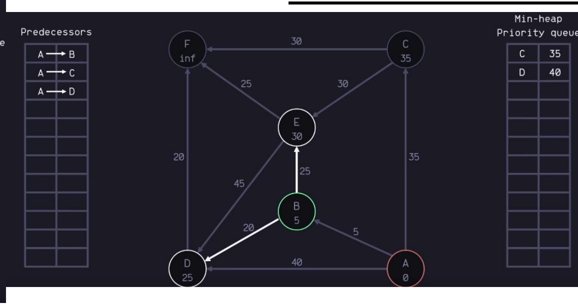


O(V + E)

Dijkstra's Algorithm

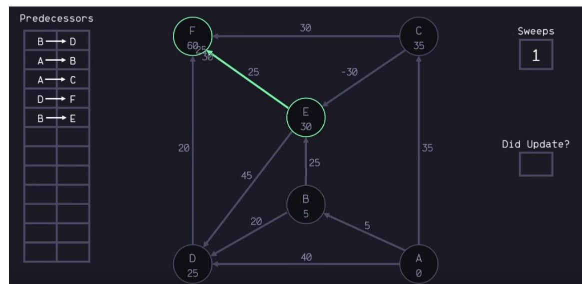
- visit each node only once
- maintain array for prev of relaxed nodes
- follow min heap for order
- relax all neighbours and add to min heap





Bellman-Ford

- sweep graph V-1 times
- relax in any order, follow that order
- boolean for "updated?" every sweep
- if updated is true at Vth sweep, negative weight cycle

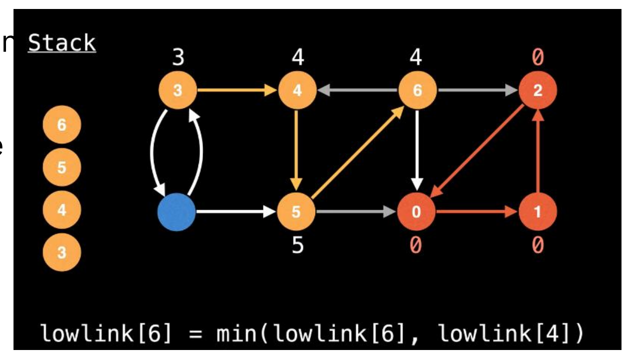


Floyd-Warshall

- considers all possible intermediate paths to reach node B from A
- If adjacency matrix has cell $\neq \infty$, path exists between nodes

Tarjan's SCC

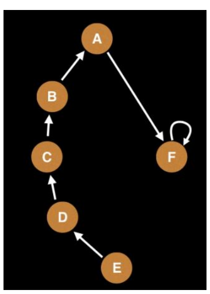
- Strongly Connected Component - every node can be reached from every other node
- Low Link Value of a node - the lowest id node reachable from the node

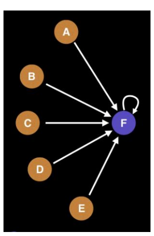


Union Find

- Union - Merge two graphs by making one the subtree of another
- Find - return the "representative" node (root of the tree) of a union

- Path Compression - point all nodes in union to representative





Minimum Spanning Tree

- A subgraph containing the minimum weight connected graph with no cycles

Kruskal's

- Repeatedly find lowest weights in *the entire graph* and add to MST
- Continue until all vertices are added

Prim's

- create visted = []
- pick any node
- pick the lowest weight that connects to an unvisited node and is *reachable from the current visited nodes*

Cycle Property

Let T be a MST of a weighted, undirected graph G. Let e be an edge of G not in T. Let C be the cycle formed by adding e to T. For every edge f ∈ C, weight(f) ≤ weight(e).

Cut Property

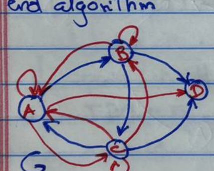
Consider a partition of a graph G into subsets U and V. Let e be an edge of minimum weight across U and V- There is an MST of G that includes e.

```
Dijksta SSSP (G(V,E), s∈V)
    let dist: V → ℤ
    let prev: V → V
    let Q be a min priority queue

    dist[s] ← 0
    for each v ∈ V do
        if v ≠ s then
            dist[v] ← ∞
        end if
        Prev[v] ← -1
        Q.add(dist[v], v)
    end for

    While Q is not empty do
        u ← Q.getmin()
        for each w ∈ V adjacent to u and still in Q do
            d ← dist[u] + weight(u, w)
            if d < dist[w] then
                dist[w] ← d
                Prev[w] ← u
                Q.set(d, w)
            end if
        end for
    end while
    return dist, prev
end algorithm
```

```
Bellman Ford SSSP (G(V,E), s∈V)
    let dist: V → ℤ
    let prev: V → V
    for each v ∈ V do
        dist[v] ← ∞
        Prev[v] ← -1
    end for
    dist[s] ← 0

    for i from 1 to |V|-1 do
        for each e = (u,v) ∈ E do
            d ← dist[v] + weight(u, v)
            if d < dist[v] then
                dist[v] ← d
                prev[v] ← u
            end if
        end for
    end for
    for each e = (u,v) ∈ E do
        if dist[u] + weigh(u,v) < dist[v] then
            throw an exception: "Negative Weight Cycle"
        end if
    end for

    return dist, prev
end algorithm
```

```
function strongconnect(v):
    // Set the depth index for v to the smallest unused index
    v.index   := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
        if w.index is undefined then
            // Successor w has not yet been visited; recurse on
            strongconnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        else if w.onStack then
            // Successor w is in stack S and hence in the curre
            // If w is not on stack, then (v, w) is an edge poi
            // Note: The next line may look odd - but is correc
            // It says w.index not w.lowlink; that is deliberat
            v.lowlink := min(v.lowlink, w.index)
        end if
    end for

    // If v is a root node, pop the stack and generate an SCC
    if v.lowlink = v.index then
        start a new strongly connected component
        Repeat
            w := S.pop()
            w.onStack := false
            add w to current strongly connected component
        while w ≠ v
        output the current strongly connected component
    end if
end function
```

```
Floyd Warshall (M: Adj. Matrix representing G(V, E))
    R^(-1) ← M                          G*
    n ← |V|
    for k from 0 to n-1 do              "bridge"     O(|V|^3)
        for i from 0 to n-1 do
            for j from 0 to n-1 do
                R^(k)[i][j] ← R^(k-1)[i][j] or (R^(k-1)[i][k] and R^(k-1)[k][j])
            end for
        end for
    end for
    return R^(n-1)
end algorithm
```



$R^{(-1)}$	A	B	C	D
A	-	1	-	-
B	-	-	1	1
C	1	-	-	1
D	-	-	-	-

k=A

$R^{(0)}$	A	B	C	D
A	-	1	-	-
B	-	-	1	1
C	1	1	-	1
D	-	-	-	-

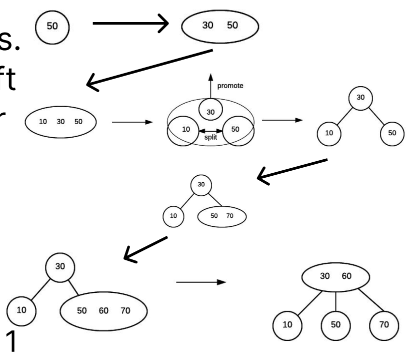
```
Kruskal MST (G(V,E))          undirected
    let Q be an empty min-heap
    let UF be a Union-Find with |V| components
    for each edge e ∈ E do
        Q.insert(weight(e), e)
    end for
    T ← {}
    while |T| < |V|-1 do
        (u,v) ← Q.getmin()
        if u and v are not conneceted in UF Then
            T.insert((u,v))
            UF.union(u, v)
        end if
    end while
    return T
end algorithm
```

```
TopologicalSort (G(V,E), s∈V)
    let H be a copy of G
    n ← 0
    let T: v∈V → ℤ≥0
    while H is not empty do
        pick v ∈ H s.t. indeg(v) = 0
        T[v] ← n
        n ← n+1
        remove v and its incident edges from H
    end while
    return T
end algorithm
```

B-Trees

Special, balanced trees. Sorted, searchable (left < right). m children per node

insertion



2-3 Trees

A specific form of B-tree

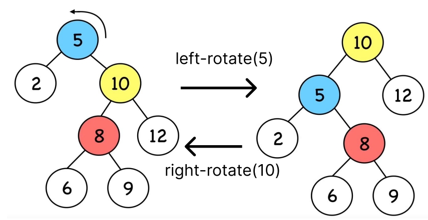
1. each node has either 1 value or 2 values
2. 1 value → 2 nodes
3. 2 values → 3 nodes
4. all leaf nodes are at the same level of the tree

for deletes, do the insertion in reverse: imagine the node to delete was just inserted.

Red-black Tree

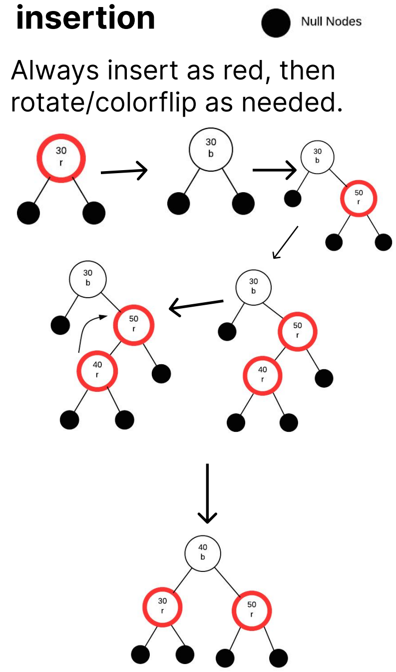
Special type of BST.

1. every node be either red or black.
2. Root must be black
3. Red node → black children
4. Null nodes are black.
5. Every path from root to null must have exactly the same number of black nodes.



insertion

Always insert as red, then rotate/colorflip as needed.

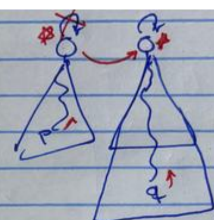



```
Prim MST (G(V,E), s∈V)      → undirected
  let dist: V → Z
  let edge: V → E
  let visited: V → {T, F}
  let Q be a min-heap (empty)
  for each v ∈ V do
    dist[v] ← ∞
    edge[v] ← -1
    visited[v] ← F
  end for
  dist[s] ← 0
  Q.insert(0, s)

  While Q is not empty do
    u ← Q.getmin()
    visited[u] ← T
    for each edge e=(u,v) ∈ E incident to u do
      if visited[v] then
        continue
      end if
      if weight(e) < dist[v] then
        edge[v] ← e
        dist[v] ← weight(e)
        if v ∈ Q then
          Q.set(dist[v], v)
        else
          Q.insert(dist[v], v)
        end if
      end if
    end for
  end while
  return edge, dist
end algorithm
```

Weighted Q-U:

id

size




```
Union (P, q)
idP ← find(P)
idQ ← find(q)
exit if idP = idQ
if size[idP] < size[idQ] then
  id[idP] ← idQ
  size[idQ] ← size[idQ] + size[idP]
else
  id[idQ] ← idP
  size[idP] ← size[idP] + size[idQ]
end if
  n ← n-1
end algorithm
```

1 0 1 2

```
Quick-Find:
Union (P, q)
  exit if id[P] = id[q]
  idP ← id[P]
  idQ ← id[q]                # ind. components
  for i from 0 to m-1 do
    if id[i] = idP then
      id[i] ← idQ
    end if
  end for
  n ← n-1
end algorithm
```