Binary Search Tree: $key[leftSubtree(x)] \le key(x) \le key[rightSubtree(x)]$

Def. successor(x) = y s.t. key(y) is the smallest key > key(x)

Def. predecessor(x) = y s.t. key(y) is the largest key < key(x)

Tree Insertion:

- Case 1: right(x) is non empty
  - successor(x) = the minimum in right(x)
- Case 2: right is empty
  - go up the tree until the current node is a left child: successor(x) is the parent of the current node
  - If you cannot go further (and you reach the root): x is the largest element

Tree Deletion: Case 1: z has no children: Delete z by making the parent of z point to NIL.
Case 2: z has one child: Delete z by making the parent of z point to z's child, instead of z.
Case 3: z has two children: ① z's successor(y) is the minimum node in z's right subtree; ② y has either no children or one right child (but no left child); ③ Delete y from the tree; ④ Replace z's key and satellite data with y's.

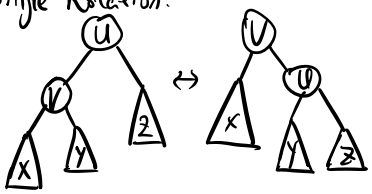
Operations on binary search trees are all O(h): SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, DELETE

Best case: $O(\lg N)$ Worst case: $O(N)$

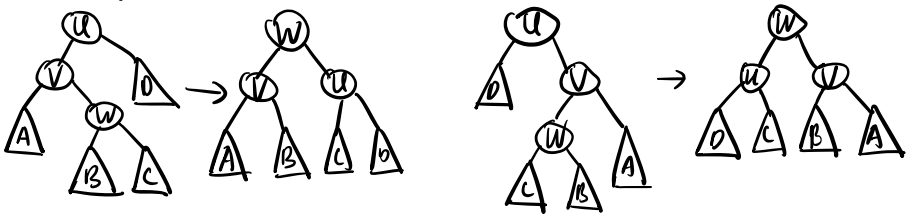
AVL trees are height-balanced binary search trees where the height of two subtrees of a node differs by at most one. **Balance factor** of a node is $|height(leftSubtree) - height(rightSubtree)|$

Insertion: ① Outside Cases: i) Insertion in the left subtree of the left child of U; ii) -- right -- right. (Single)
② Inside Cases: i) Insertion in the right subtree of the left child of U; ii) -- left -- right. (Double)

Single Rotation:



Double Rotation:



Running Time for AVL Trees: A single restructure/rotation is O(1); Find/search is O(log n); Insertion is O(log n); Deletion is O(log n).

Graph: A set of vertices (nodes) with edges (links) between them. $|E| = O(|V|^2)$

Matrix Representation: Memory required: $\Theta(V^2)$, independent on the number of edges in G; Preferred when the graph is dense ($|E| \cong |V|^2$) or we need to quickly determine if there is an edge between two vertices; Time to determine if $(u,v) \in E$: $\Theta(1)$; Disadvantage: No quick way to determine the vertices adjacent to a vertex; Time to list all vertices adjacent to u: $\Theta(V)$

List Representation: Memory required: $\Theta(V+E)$; Preferred when the graph is sparse ($|E| << |V|^2$) or we we need to quickly determine the nodes adjacent to a given node; Disadvantage: No quick way to determine whether there is an edge between u and v; Time to determine $(u,v) \in E$: $\Theta(degree(u))$; Time to list all vertices adjacent to u: $\Theta(degree(u))$

```
BFS(G, s){
  for each u in V{
    color[u] = white;
    d[u] = infinity;
    pred[u] = null;
  }
  color[s] = gray;
  d[s] = 0;
  Q = {s};
  while(Q is nonempty){
    u = Q.Dequeue();
    for each v in Adj[u]{
      if color == white{
        color[v] = gray
        d[v] = d[u] + 1
        pred[v] = u
        Q.Enqueue(v);
      }
    color[u] = black
```

```
DFS(G) {
  for each u in V{
    color(u) = white;
    pred[u] = null;
  }
  time = 0;
  for each u in V{
    if(color(u) == white):
      DFS visit(u);
  }
```

```
DFS visit(u){
  color(u) = gray;
  d[u] = ++time;
  for each v in Adj(u){
    if color[v] == white{
      pred(v) = u;
      DFS visit(v);
    }
  }
  color[u] = black;
  f[u] = ++time;
```

DFS edges: ① Tree edge: encounter new (white) vertex, edge from parent to child in depth first tree. v is white when (u,v) is first explored ② Back edge: from descendant to ancestor in depth first tree. v is gray when (u,v) is first explored ③ Forward edge: from ancestor to descendant. v is black when (u,v) is first explored ④ Cross edge: between two nodes without ancestor-descendant relation in a

depth first tree or in two different depth first tree.

Parenthesis Theorem: In any DFS of directed or undirected graph, for any two vertices u and v, one of the following three conditions holds: 1. interval [d(u), f(u)] and [d(v), f(v)] are disjoint; 2. [d(u), f(u)] entirely inside [d(v), f(v)]; 3. [d(v), f(v)] entirely inside [d(u), f(u)].

Corollary: v is a proper descendant of u in DFS forest iff d(u) < d(v) < f(v) < f(u)

White-path Theorem: Vertex v is a descendant of u in a DFS tree iff at time d(u) that u was discovered, vertex v can be reached from u along a path consisting entirely of white vertices.

Theorem: In a depth first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Theorem: An undirected graph is acyclic iff a DFS yields no back edge.

Theorem: A directed graph is acyclic iff a DFS yields no back edge.

Directed Acyclic Graph (DAG) is a directed graph that contains no directed cycles.

A **topological order / topological sort** of a DAG $G=(V,E)$ is an ordering of its nodes as $v_1, v_2, \cdots, v_n$ so that for every edge (v_i, v_j) we have $i<j$.

TOPOLOGICAL-SORT(G): ① call DFS(G) to compute finishing time f(v) for each node v; ② as each vertex is finished, insert it into the front of a linked list; ③ return the linked list of vertices.

Running time for topological sort is $\Theta(V+E)$: DFS takes $\Theta(V+E)$, insertion takes $O(1)$

A digraph is **strongly connected** if for every pair of vertices $u, v \in V$, u can reach v and vice versa.

Kosaraju's Algorithm: ① call DFS(G) to compute finishing time f(u) for each vertices u; ② compute G^T; ③ call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f(u); ④ output the vertices of each tree in the depth-first forest formed in ③ as separate strongly connected component.

Running Time: $\Theta(V+E)$

BOTTOM-UP-CUT-ROD(p, n):

```
let r[0,...,n] be a new array;
r[0] = 0;
for j = 1 to n:
    q = -∞;
    for i = 1 to j:
        q = max(q, p[i] + r[j-i]);
    r[j] = q;
return r[n];
```

EXTENDED-BOTTOM-UP-CUT-ROD(p,n):

```
let r[0,...,n] and s[0,...,n] be new arrays;
r[0] = 0;
for j = 1 to n:
    q = -∞;
    for i = 1 to j:
        if q < p[i] + r[j-i]:
            q = p[i] + r[j-i];
            s[j] = i;
    r[j] = q;
return r and s;
```

MEMOIZED-CUT-ROD(p,n):

```
let r[0,...,n] be a new array;
for i = 0 to ∞:
    r[i] = -∞;
return MEMOIZED-CUT-ROD-AUX(p,n,r);
```

MEMOIZED-CUT-ROD-AUX(p, n, r):

```
if r[n] ≥ 0: return r[n];
if n == 0: q = 0;
else: q = -∞;
    for i = 1 to n:
        q = max(q, p[i] + MCRA(p, n-i, r));
r[n] = q;
return q;
```

PRINT-CUT-ROD-SOLUTION(p,n):

```
(r, s) = EBUCR(p, n)
while n > 0:
    print s[n];
    n = n - s[n];
```

Longest Common Subsequence.

Define c[i][j] as the LCS of sequence X[1,...,i] and Y[1,...,j]

Thus: c[i][0] = 0 for all i; c[0][j] = 0 for all j.

Goal: Find c[M][N]

$$c[i,j] = \begin{cases} 0, & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1, & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]), & \text{if } i,j>0 \text{ } x_i \neq y_j \end{cases}$$

A matrix b[i,j], for a subproblem, tell us what choice was made to obtain the optimal value.

if $x_i = y_j$: b[i][j] = ↖;
else if c[i-1,j] > c[i,j-1]: b[i][j] = ↑;
else: b[i][j] = ←;

Greedy choice property: A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. **Optimal substructure property**: A problem exhibits optimal substructure if an optimal solution to the problem contains the optimal solutions to subproblems

Activity-Selection Problem

Greed-Choice Property:

Suppose $A \subseteq S$ is an optimal solution. Order the activities in A by finish time. The first activity in A is k. If k=1, the schedule A begins with a greedy choice. If $k \neq 1$, then let $B = A - \{k\} + \{1\}$, B has the same number of activities as A. Thus B is optimal

Optimal Substructure Property:

If A is optimal to S, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S : s_i \geq f_1\}$. Why? If we could find a solution B' to S' with more activities than A', adding {1} to B' would yield a solution B to S with more activities than A → contradicting the optimality of A.

Knapsack Problem 0-1: Each item is either taken or not taken. Fractional: Allow to take fraction of items.

Both 0-1 and fractional knapsack have optimal substructure

0-1: If item j is removed from an optimal packing, the remaining is an optimal packing with weight $W-w_j$ that can be taken from the $n-1$ items other than j.

fractional: If w pounds of items j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W-w$ that can be taken from other $n-1$ items plus w_j-w of item j.

Only fractional knapsack has the greedy choice property

Dynamic Programming for 0-1 knapsack: $B[k,w] = \begin{cases} B[k-1,w], & \text{if } w_k > w \\ \max(B[k-1,w],\ B[k-1,w-w_k]+v_k), & \text{if } w_k \le w \end{cases}$, where $B[k,w]$ is the solution over items from 1 to k under the weight budget of w.

Huffman Coding: ① Place the elements into a min heap; ② Remove the first two elements from the heap; ③ Combine these two elements into one; ④ Insert the new element back into the heap.

Huffman(C) — Time Complexity: $O(n \log n)$

```
n = |C|; Q = C
for i = 1 to n-1:
    Allocate a new node z
    z.left = x = EXTRACT-MIN(Q);
    z.right = y = EXTRACT-MIN(Q);
    z.freq = x.freq + y.freq;
    Insert(Q, z);
return EXTRACT-MIN(Q);
```

EXTRACT-MIN(Q) needs $O(\log n)$ by a heap operation. Requires initially an time to build a binary heap

Prim's Algorithm:

```
Q ← V;                                   } Θ(V)
key[v] ← ∞ for all v ∈ V;
key[s] ← 0 for some arbitrary s ∈ V;
while Q ≠ ∅:                             } |V| times
    do u ← EXTRACT-MIN(Q)
    for each v ∈ Adj[u]                  } degree(u) time
        do if v ∈ Q and w(u,v) < key(v)
            then key[v] ← w(u,v);  π(v) ← u;
```

DECREASE-KEY

Minimum Spanning Tree (MST): The subset of edges that connected all vertices in the graph, and has minimum total weight.

MST Problem: Input: A connected, undirected graph $G=(V,E)$, with weight function $w: E \to R$

Output: A spanning tree — a tree that conneces all vertices — of minimum weight.

Prim's Algorithm

Optimal Substructure: Remove any edge $(u,v) \in T$. then T is partitioned into T_1 and T_2.

Theorem: The subtree T_1 is an MST of $G_1=(V_1,E_1)$, similarly for T_2.

Greedy Choice: Theorem: Let T be the MST of $G=(V,E)$, and Let $A \subseteq V$. Suppose that $(u,v) \in E$ is the least-weight edge connecting A to $V-A$. Then $(u,v) \in T$.

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$	$O(E + V \lg V)$

Kruskal(V,E) $[O(E \lg V)]$

```
A = ∅;
for each v ∈ V:
    Make-disjoint-set(v);
Sort E by weight;
for each (v1, v2) ∈ E:
    if Find(v1) ≠ Find(v2):
        A = A ∪ {(v1, v2)};
        Union(v1, v2);
return A;
```

Path in graph: Consider a digraph $G=(V,E)$ with edge-weight function $w: E \to R$. The weight of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$.

The shortest path weight from u to v is defined as $\delta(u,v) = \min\{w(p): p \text{ is a path from } u \text{ to } v\}$

Theorem: A subpath of a shortest path is a shortest path (Optimal Substructure)

Theorem: For all $u,v,x \in V$, we have $\delta(u,v) \le \delta(u,x) + \delta(x,v)$ (Triangle Inequality)

Single-source shortest path problem: from a given source vertex $s \in V$, find all shortest-path weight $\delta(s,v)$ for all $v \in V$.

Dijkstra's Algorithm:

```
d[s] ← 0; for each v ∈ V-{s}: d[v] ← ∞;
S ← ∅; Q ← V;
while Q ≠ ∅:
    u ← EXTRACT-MIN(Q);
    S ← S ∪ {u};
    for each v ∈ Adj[u]:
        if d[v] > d[u] + w(u,v):        ← Relaxation Step
            d[v] = d[u] + w(u,v);
```

Time $= \Theta(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}})$

Bellman-Ford Algorithm:

```
d[s] ← 0; for each v ∈ V-{s}: d[v] ← ∞;
for i ← 1 to |V|-1:
    for each edge (u,v) ∈ E:
        if d[v] > d[u] + w(u,v):
            d[v] ← d[u] + w(u,v);
for each edge (u,v) ∈ E:
    if d[v] > d[u] + w(u,v):
        report that a negative-weight cycle exists;
```

Time: $\Theta(VE)$

Corollary: If a value $d[v]$ fails to converge after $|V|-1$ passes, there exists a negative-weight cycle in G reachable from s.

Bellman-Ford Algorithm: Finds all shortest-path length from a source $s \in V$ to all $v \in V$ or determine that a negative-weight cycle exists.

Recall: If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Shortest Paths in Directed Acyclic Graphs: Topological sort + one pass Bellman-Ford

```
DAG-SHORTEST-PATHS(G, w, s):
    topologically sort the vertices of G;
    INITIALIZE-SINGLE-SOURCE(G, s);
    for each vertex u, taken in topologically sorted order:
        for each vertex v ∈ G.Adj[u]:
            RELAX(u, v, w);
```

```
FLOYD-WARSHALL(W):
    n = W.rows; D^(0) = W;
    for k = 1 to n:
        let D^(k) = (d_ij^(k)) be a new n×n matrix;
        for i = 1 to n:
            for j = 1 to n:
                d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1));
    return D^(n);
```

All-pairs shortest path: Given a weighted digraph $G = (V, E)$ with weight function $w: E \to R$, determine the length of the shortest path between all pairs of vertices in G.

Nonnegative edge weights: $|V|$ times of Dijkstra's algorithm: $O(VE + V^2 \lg V)$

Unweighted graph: $|V|$ times of BFS: $O(VE + V^2)$

General case: Bellman-Ford algorithm: $O(V^2 E)$

The Floyd-Warshall Algorithm: let $d_{ij}^{(k)}$ be the length of shortest path from i to j s.t. all intermediate vertices on the path (if any) are in the set $\{1, 2, \cdots, k\}$. Our aim is to compute $D^{(n)}$ ($d_{ij}^{(n)}$). Subproblem: compute $D^{(k)}$ for $k = 0, 1, 2, \cdots, n-1$

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, & \text{if } k \geq 1 \end{cases}$$

Let $\pi_{ij}^{(k)}$ be the predecessor of vertex j on the shortest path from vertex i with all intermediate vertices in the set $\{1, \cdots, k\}$.

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & \text{if } i = j \text{ or } w_{ij} = \infty \\ i, & \text{if } i \neq j \text{ and } w_{ij} \neq \infty \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)}, & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Transitive Closure of G is the graph $G^* = (V, E^*)$, where $E^* = \{(i, j)$: there is a path from vertex i to vertex j in $G\}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$

$$t_{ij}^0 = \begin{cases} 0, & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1, & \text{if } i = j \text{ or } (i, j) \in E \end{cases}, \quad \text{for } k \geq 1: \; t_{ij}^k = t_{ij}^{k-1} \vee (t_{ik}^{k-1} \wedge t_{kj}^{k-1})$$