k_d Tree

Time Complexity

insert: $O(\log n)$ (average), $O(n)$ (worst)

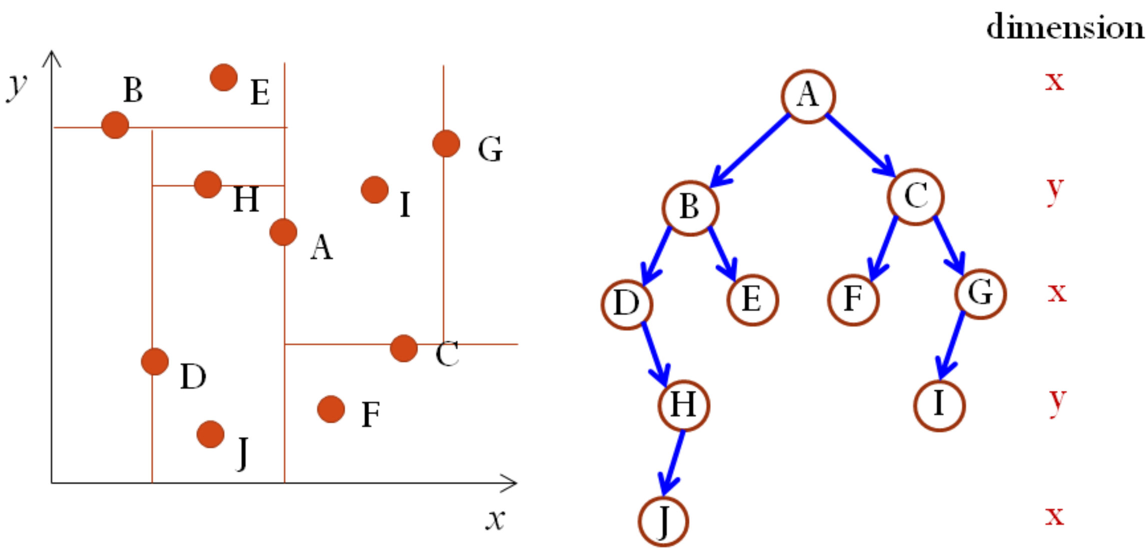
search: $O(\log n)$ (average), $O(n)$ (worst)

remove: $O(\log n)$ (average), $O(n)$ (worst)

Mechanism

- A **binary search tree**
- At each level, keys from a different search dimension is used as the discriminator
  - Nodes on the left subtree (right subtree) of a node have keys with value $<$ ($\geq$) the node's key value **along this dimension
- **Cycle** through the dimensions as we go down the tree

e.g. Insert order: ABCDEFGHIJ



Insertion

```
void insert(node *&root, Item item, int dim) {
    if(root == NULL) {
        root = new node(item);
        return;
    }
    if(item.key == root->item.key)
        // equal in all dimensions
        return;
    if(item.key[dim] < root->item.key[dim])
        // key smaller than that of root's along the dimension of the current
level -> recursion
        // In recursive call, cyclically increment the dimension
        insert(root->left, item, (dim+1)%numDim);
    else
```

```
        insert(root->right, item, (dim+1)%numDim);
}
```

Search

Similar to insertion

```
node *search(node *root, Key k, int dim) {
    if(root == NULL) return NULL;
    if(k == root->item.key)
        return root;
    if(k[dim] < root->item.key[dim])
        return search(root->left, k, (dim+1)%numDim);
    else
        return search(root->right, k, (dim+1)%numDim);
}
```

Removal

Leaf: remove directly

Non-leaf:

- If the node R to be removed has right subtree, find the node M in right subtree with the **minimum** value of the current dimension.
  - Replace the value of R with the value of M
  - Recurse on M until a leaf is reached. Then remove the leaf
- Else, find the node M in left subtree with the **maximum** value of the current dimension. Then replace and recurse.

Find Minimum

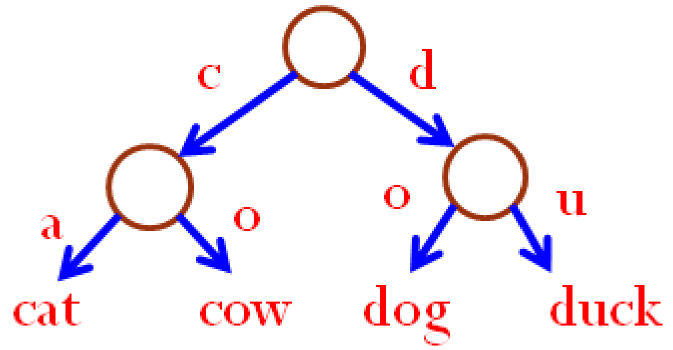
```
node *findMin(node *root, int dimCmp, int dim) {
    // dimCmp: dimension for comparison
    // dim: current dimension
    if(!root) return NULL;
    node *min =
        findMin(root->left, dimCmp, (dim+1)%numDim);
    if(dimCmp != dim) {
        // Then minimum might be in right subtree
        rightMin =
            findMin(root->right, dimCmp, (dim+1)%numDim);
        min = minNode(min, rightMin, dimCmp);
        // minNode returns the smaller node on dimCmp
        // compare leftmin and rightmin
    }
    return minNode(min, root, dimCmp);
    // compare the minimum in subtrees and root, since root might not be in
comparison dimension
}
```

Range Search

赌他不考

Tries

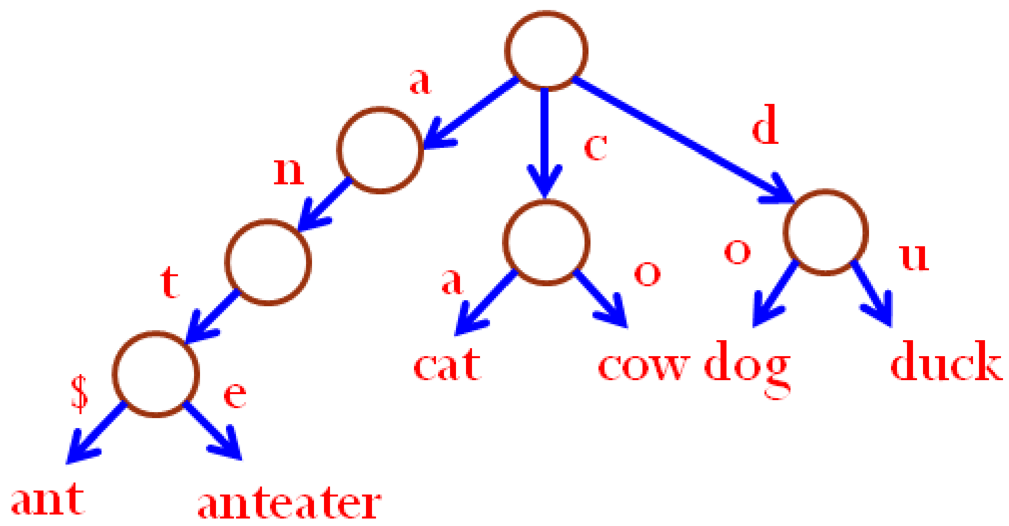
- A trie is a tree that **uses parts of the key**, as opposed to the whole key, to perform search.
- Data records are only stored in **leaf** nodes. Internal nodes do not store records; they are "**branch**" points to direct the search process.



- Trie usually is used to store a set of **strings from an alphabet**.
  - The alphabet is in the general sense, not necessarily the English alphabet.
  - For example, {0, 1} is an alphabet for binary codes {0010, 0111, 101}. We can store these three codes using a trie.

Mechanism

- Each **edge** of the trie is **labeled with symbols** from the alphabet.
- Labels of edges on the path from the root to any leaf in the trie forms a **prefix** of a string in that leaf.
  - Trie is also called **prefix-tree**.
- The **most significant symbol** in a string determines the **branch direction at the root**.
- As long as there is <u>only one key in a branch</u>, we do not need any further internal node below that branch; we can <u>put the word directly as the leaf of that branch.</u>
- Sometimes, a string in the set is exactly a **prefix** of another string.
  - For example, "ant" is a prefix of "anteater".
- We **add a symbol** to the alphabet to **indicate the end of a string**. For example, use "$" to indicate the end.



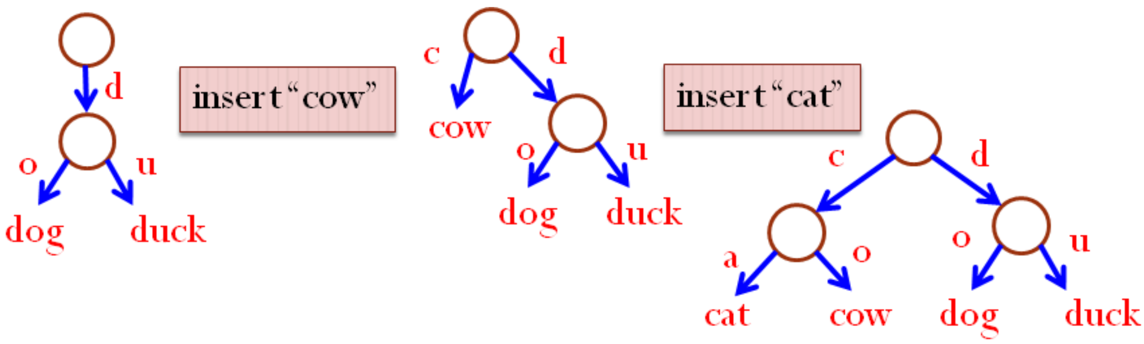
- We can keep an **array of pointers** in a node, which corresponds to **all** possible symbols in the alphabet.
    - or, a **linked list of pointers** to the child nodes, corresponding to a small fraction of the possible symbols in the alphabet.

Search

- Follow the search path, starting from the root.
- When there is **no branch**, return **false**.
- When the search **leads to a leaf**, further **compare with the key** at the leaf.

Insertion

- Follow the search path, starting from the root.
- If a new branch is needed, add it.
- When the search **leads to a leaf**, a conflict occurs. We **need to branch**.
    - **Use the next symbol** in the key
    - The originally-unique word must be moved to lower level



Removal

- The key to be removed is always **at the leaf**.
- After deleting the key, if the parent of that key now has **only one child *C***, remove the parent node and move key *C* one level up.
    - If key *C* is the only child of its new parent, **repeat** the above procedure again.

Time Complexity

Suppose a key has k symbols.

- Insert: $O(k)$ (worst)
- find: $O(k)$ (worst)

This does not depend on the number of keys N.

Sometimes we can access records even **faster**.

- A key is stored at the depth which is enough to distinguish it with others.
- For example, in the previous example, we can find the word "duck" with just "du".

AVL Trees

Balanced Search Trees

- Height of a tree of n nodes = $O(\log n)$
- $O(\log n)$ time to **rebalance**

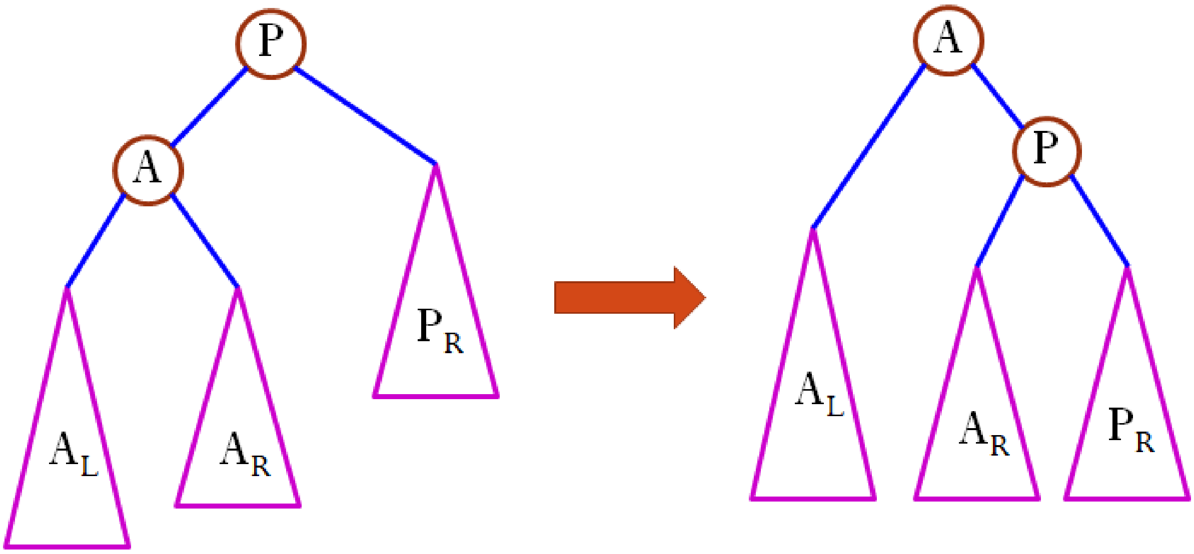
Balance Condition

- An empty tree is **AVL balanced**.
- A non-empty binary tree is **AVL balanced** if
  - Both its left and right subtrees are AVL balanced, and
  - The height of left and right subtrees differ by **at most 1**.

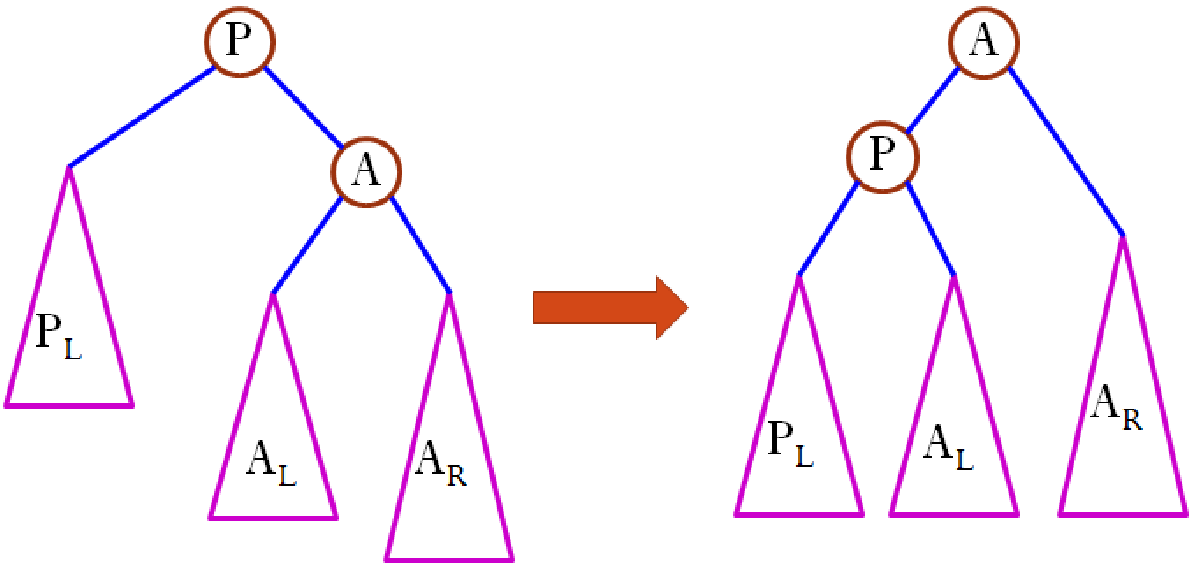
Re-Balance

via rotation

Right Rotation



Left Rotation



Balance Factor

Let h_l, h_r be the height of left and right subtrees of T respectively.

The balance factor B_T of T is: $B_T = h_l - h_r$

After Insertion

Inserting an item in a tree affects potentially the heights of all of the nodes along the **access path**, i.e., the path from the root to that leaf.

-> Recompute height along the access path and check AVL balance condition.

-> Fix **the first unbalanced node** in the access path **from the leaf**.

	characteristic	solution
LL Insertion	Node P becomes unbalanced with a **positive** balance factor and the **left** subtree of the node also has a **positive** balance factor.	LL Rotation (Right rotation at P)
RR Insertion	Node P becomes unbalanced with a **negative** balance factor and the **right** subtree of the node also has a **negative** balance factor.	RR Rotation (Left rotation at P)
LR Insertion	Node P becomes unbalanced with a **positive** balance factor but the **left** subtree A of the node has a **negative** balance factor.	LR Rotation (Left rotation at A, then right rotation at P)
RL Insertion	Node P becomes unbalanced with a **negative** balance factor but the **right** subtree A of the node has a **positive** balance factor.	RL Rotation (Right rotation at A, then left rotation at P)

The height of the tree **after the rotation** is the same as the height of the tree before insertion.

Removal

- First remove node as with BST
- Then **update** the balance factors of those ancestors in the access path and **rebalance** as needed.
- Time Complexity: O(log n)
  - Only rebalance along the ancestor path

Supporting Data Members and Functions

```
struct node {
    Item item;
    int height;
    node *left;
    node *right;
};
int Height(node *n) {
  if(!n) return -1;
  return n->height;
}
```

```
void AdjustHeight(node *n) {
    if(!n) return;
    n->height = max( Height(n->left),Height(n->right) ) + 1;
}
int BalFactor(node *n) {
  if(!n) return 0;
  return (Height(n->left) - Height(n->right));
}
void Balance(node *&n) {
  if(BalFactor(n) > 1) {
    if(BalFactor(n->left) > 0) LLRotation(n);
    else LRRotation(n);
  }
  else if(BalFactor(n) < -1) {
    if(BalFactor(n->right) < 0) RRRotation(n);
    else RLRotation(n);
  }
}
void insert(node *&root, Item item){
    if(root == NULL) {
        root = new node(item);
        return;
    }
    if(item.key < root->item.key)
        insert(root->left, item);
    else if(item.key > root->item.key)
        insert(root->right, item);

  Balance(root);
  AdjustHeight(root);
}
```

Time Complexity

- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$

Red-Black Tree

- A binary search tree.
- The data structure requires an extra one-bit color field in each node.
    - Every node is **either red or black**.

Rules

- **Root rule**: The root is black.
- **Red rule**: Red node can **only have** black children.
    - No consecutive red nodes on a path
- **Path rule**: **Every** path from a node x to NULL must have the **same number** of black nodes (including x itself).
    - Same **black height**

Implication of the Rules

If a **red** node has **at least one** child, it **must have two children** and they must be **black**.

- A red node's child can only be black.
- If has only one black child, then violate the **path rule**.

If a black node has **only one** child, that child **must be** a **red leaf**.

- Can't be black. (or violate path rule)
- Must be a leaf. (or the red child will have black children, then violate path rule)

Height Guarantee

Claim: every red-black tree with n nodes has height $\leq 2\log_2(n+1)$.

Proof:

In a binary tree with n nodes, there is a root-NULL path with **at most** $\log_2(n+1)$ nodes.

Thus: # black nodes on that path$\leq \log_2(n+1)$.

By path rule: every root-NULL path has $\leq \log_2(n+1)$ **black nodes**.

By red rule: every root-NULL path has $\leq 2\log_2(n+1)$ **total nodes**.

Insertion

New node is always added as a **red leaf**.

- If parent is black, done (trivial case).
- If parent is red, violate the **red rule**! -> fix by moving the violation **up the tree** by rotation/recoloring

Violation at Leaf

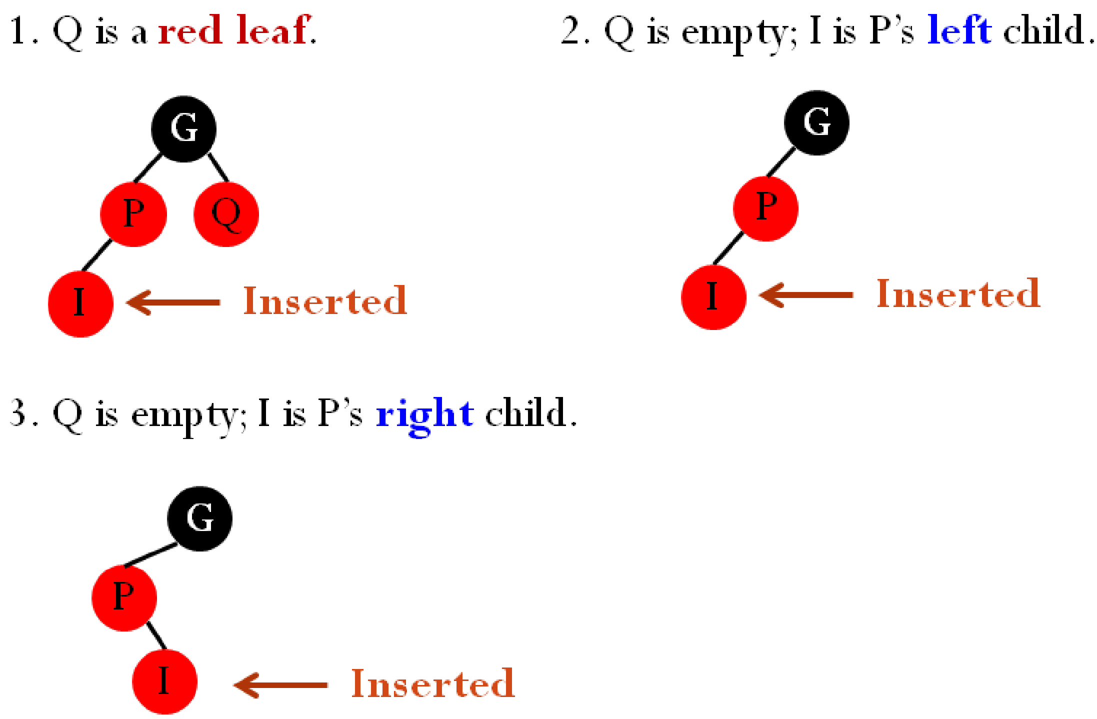
Note: only **red rule** may be violated by inserting a (red) node as a leaf.

When violating, its **parent** is **red** and its **grandparent** is **black** (since the tree was valid before insertion).

Denote: the inserted node as "I", its parent as "P", its grandparent as "G", the other child of G as "Q".

- P was a leaf, Q was either a red leaf or a NULL before insertion.

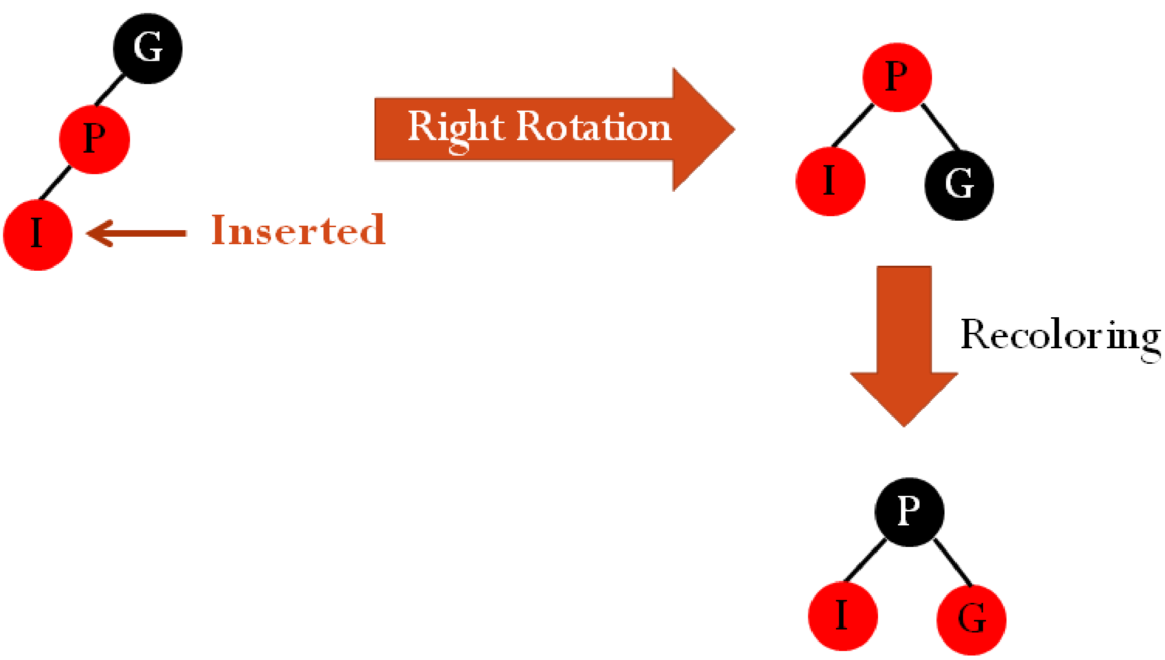
Then there are three cases:



Case 1

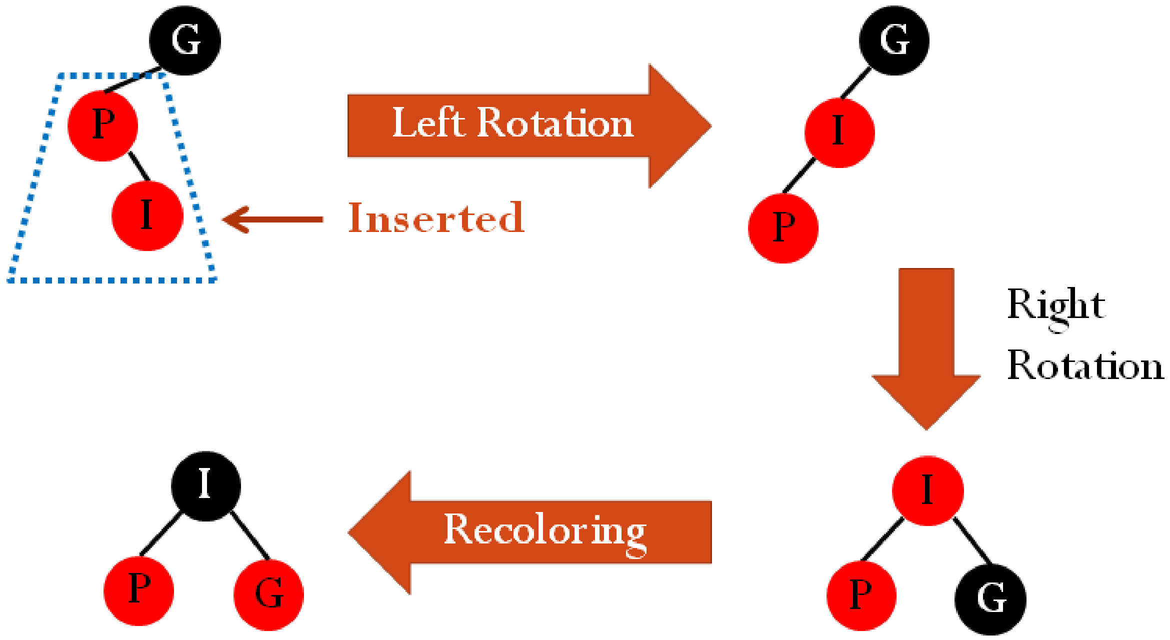
May **recurse**, since G's parent may be red.

Case 2



No recursion

Case 3



No recursion

Violation at Internal Nodes

Caused by **moving the violation up** the tree.

When violating, its **parent** is **red** and its **grandparent** is **black**.

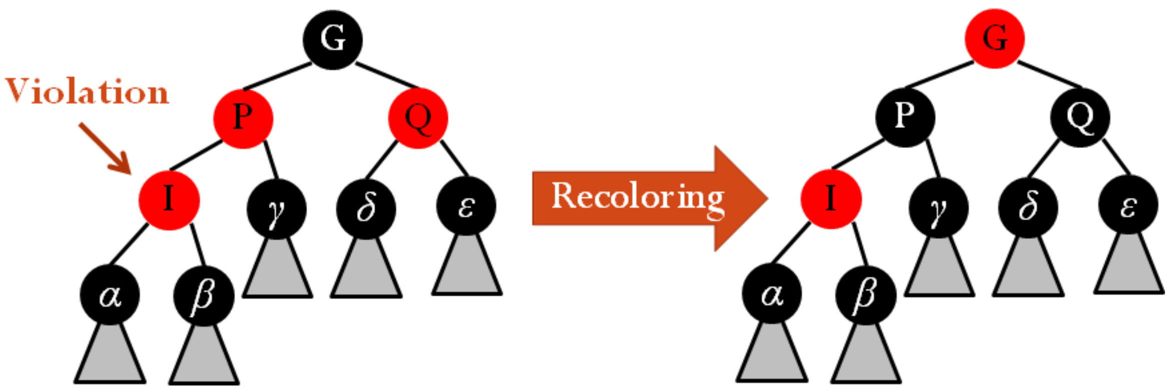
Assume: the parent “P” is the **left child** of the grandparent “G”. (The “right child” case is **symmetric**.)

Denote: the right child of the grandparent to be Q.

There are three cases.

Case 1

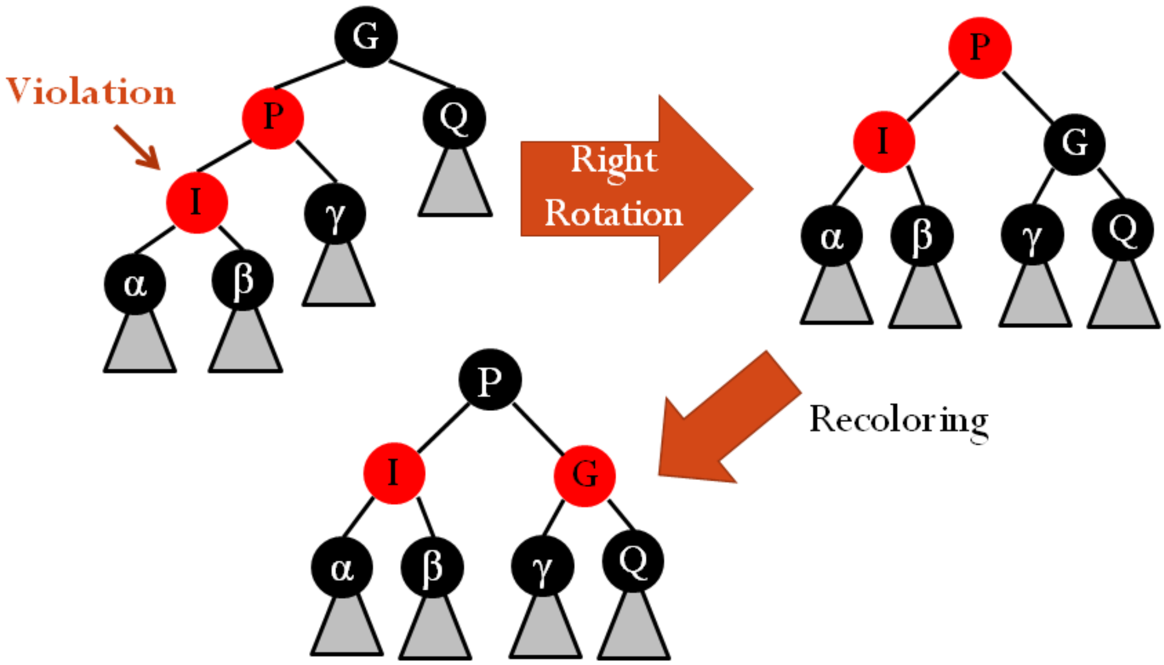
Q is a **red node**.



May **recurse**, since G’s parent may be red.

Case 2

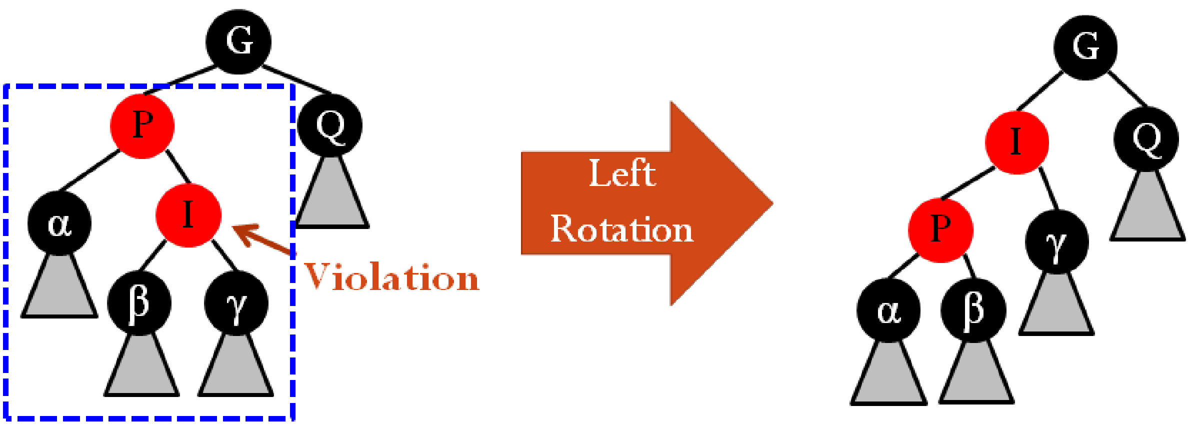
Q is a **black node**; I is P's **left** child.



No recursion

Case 3

Q is a **black node**; I is P's **right** child.



<u>Then treat as case 2.</u>

Violation Fix at the Root

- By **moving the violation up** the tree the root may become **red**.
- Final step: set root to be **black**.
  - All red-black tree properties are now **restored**.

Removal

Three basic cases:

Red leaf

Remove directly

Black Node with Single Red Child

- Delete black node and link the red child to black node's parent
- Recolor the red node to black

Black leaf

哎 回头再说 摸了

Time Complexity

worst case: $O(\log n)$

Compared Against AVL Tree

Tree is **less balanced**

- Bad for search
- Good for insertion/deletion

Graph

A **graph** is a set of **nodes** $V = \{v_1, v_2, \ldots, v_n\}$ and **edges** $E = \{e_1, e_2, \ldots, e_m\}$that connects pairs of nodes.

- Nodes also known as **vertices**.
- Edges also known as **arcs**.

Two nodes are **directly connected** if there is an edge connecting them.

- Directly connected nodes are **adjacent** to each other, and one is the **neighbor** of the other.
- The edge directly connecting two nodes are **incident** to the nodes, and the nodes **incident** to the edge.

Simple Graphs

- Two nodes may be directly connected by more than one **parallel edges**.
- An edge connecting a node to itself is called a **self-loop**.

A **simple graph** is a graph <u>without parallel edges and self-loops</u>.

Complete Graphs

A **complete graph** is a graph where every pair of nodes is directly connected.

- $N(N-1)/2$ edges for N nodes

Directed Graphs

Directed graph (digraph): edges are directional.

Nodes incident to an edge form an **ordered** pair.

- $e = (v_1, v_2)$ means there is an edge **from** v_1 **to** v_2. However, there is no edge **from** v_2 **to** v_1.

Undirected Graphs

Undirected graph: all edges have no orientation.

There is no ordering of nodes on edges.

- $e = (v_1, v_2)$ means there is an edge **between** v_1 **and** v_2.

Paths

A **path** is a series of nodes $v_1, \ldots, v_n$ that are connected by edges.

- For a directed graph, if $v_1, \ldots, v_n$ is a path, then there is an edge **from** v_i **to** v_{i+1} for each i.
- For an undirected graph, if $v_1, \ldots, v_n$ is a path, then there is an edge **between** v_i **and** v_{i+1} for each i.

Simple Paths

A **simple path** is a path with no node appearing twice.

Connected Graphs

A **connected graph** is a graph where a simple path exists between all pairs of nodes.

A directed graph is **strongly connected** if there is a **simple directed path** between any pair of nodes.

A directed graph is **weakly connected** if there is a **simple path** between any pair of nodes in the underlying undirected graph.

Node Degree

Undirected graph

The **degree** of a node is the number of edges incident to the node.

- $Sum(degrees) = 2 * Number(edges)$

Directed graph

For directed graphs:

- **in-degree**: number of **incoming edges** of a node
- **out-degree**: number of **outgoing edges** of a node

Nodes with zero in-degree are **source** nodes.

Nodes with zero out-degree are **sink** nodes.

Cycles and Directed Acyclic Graphs

A **cycle** is a path starting and finishing at the same node.

- A self-loop is a cycle of length 1.
- A **simple cycle** has no repeated nodes, except the first and the last node.

A graph with no cycle is called an **acyclic graph**.

- A directed graph with no cycles is called a **directed acyclic graph**, or **DAG** for short.

Weighted Graphs

Edges of a graph may have different costs or weights.

Graph Size and Complexity

The size of a graph and the complexity of a graph algorithms are usually defined in terms of

- number of edges |E|
- number of vertices |V|
- or both

Sparse graph: $|E| << |V|^2$ or $|E| \approx \Theta(|V|)$

- Example: tree

Dense graph: $|E| \approx \Theta(|V|^2)$

- Example: complete graph

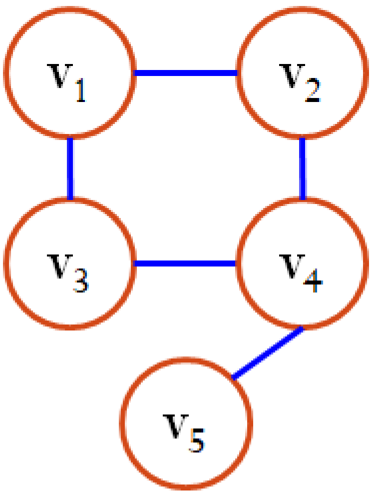
Graph Representation

Adjacency Matrix

a $|V| \times |V|$ matrix representation of a graph.

Unweighted

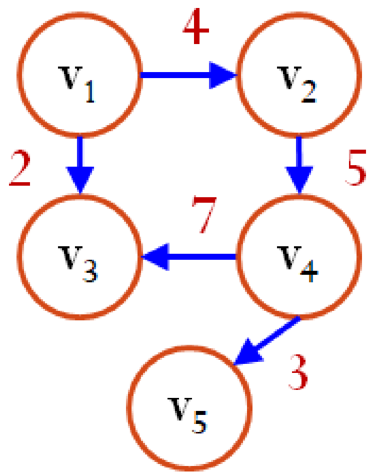
$A(i, j) = 1$, if (v_i, v_j) is an edge; otherwise $A(i, j) = 0$.



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0

Weighted

If (v_i, v_j) is an edge and its weight is w_ij, then $A(i, j) = w_ij$; otherwise $A(i, j) = \infty$.



	1	2	3	4	5
1	∞	4	2	∞	∞
2	∞	∞	∞	5	∞
3	∞	∞	∞	∞	∞
4	∞	∞	7	∞	3
5	∞	∞	∞	∞	∞

Properties

Space complexity: $|V|^2$ units

For an undirected graph, may store only the lower or upper triangle. Thus, $(|V|-1)|V|/2$ units.

Time complexity for finding a specific neighbour: $O(1)$

Time complexity for finding **all** nodes adjacent to a given node v_i: $O(|V|)$

Adjacency List

An array of $|V|$ **linked lists**.

- Each array element represents a node and its linked list represents the node's neighbors.
- Each edge in an undirected graph is represented twice.
    - Each edge is treated as **bidirectional**.
- Each edge in a directed graph is represented once.
- Weighted graph stores **edge weight** in linked-list node.

Properties

Space complexity: $O(|E|+|V|)$

The **worst case** time complexity for checking if node v_i is adjacent to node v_j: $O(|V|)$

The **worst case** time complexity for finding all nodes adjacent to a given node v_i: $O(|V|)$

Comparison of Graph Representation

Adjacency **list** often requires **less space** than adjacency matrix.

Dense graphs are more efficiently represented as adjacency **matrices** and sparse graphs as adjacency lists.

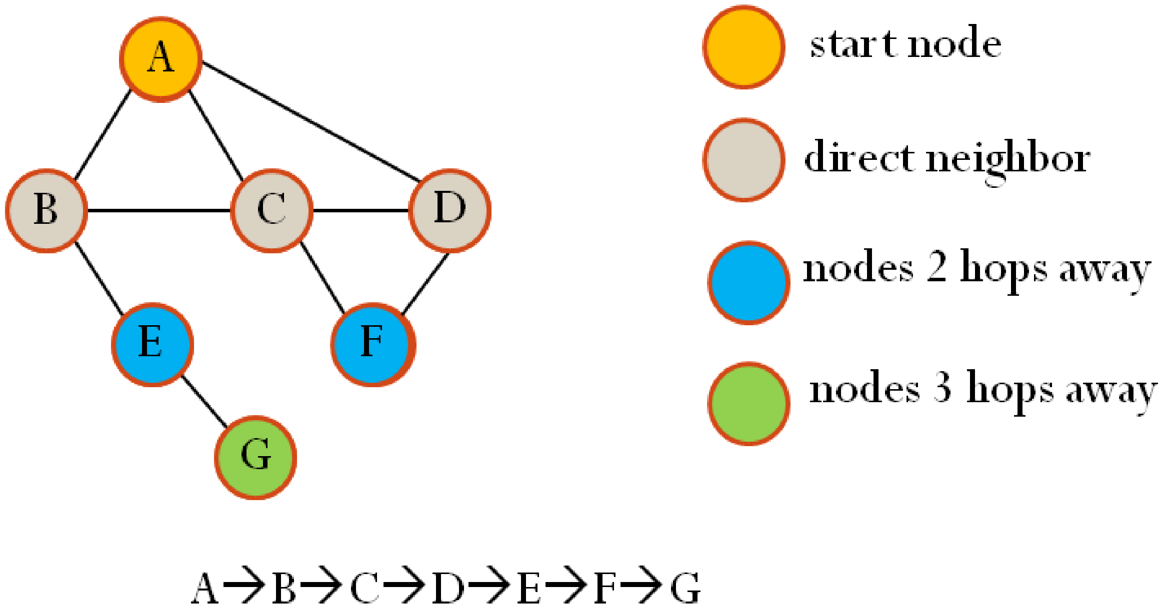
Graph Search

A node u is **reachable** from a node v if and only if there is **a path from** v **to** u.

A graph search method starts at a given node v and visits **every** node that is **reachable** from v **exactly once**.

Breadth-First Search (BFS)

Given a start node, visit all directly connected neighbors first, then nodes 2 hops away, 3 hops away, and so on.



Implemented by a **queue**

```
BFS(s) {
    queue q; // An empty queue
    visit s and mark s as visited;
    q.enqueue(s);
    while(!q.isEmpty()) {
        v = q.dequeue();
        for(each node u adjacent to v) {
            if(u is not visited) {
                visit u and mark u as visited;
                q.enqueue(u);
            }
        }
    }
}
```

Depth-First Search (DFS)

Recursion

```
DFS(v) {
    visit v;
    mark v as visited;
    for(each node u adjacent to v)
        if(u is not visited) DFS(u);
}
```

Time Complexity

adjacency matrix:

Visit each node exactly once: $O(V)$.

The row of each node in the adjacency matrix is **scanned** once: $O(|V|)$ for each node.

Total running time: $O(|V|^2)$.

adjacency list:

Visit each node exactly once: $O(|V|)$.

Adjacency list of each node is **scanned** once. Size of entire adjacency list is $2|E|$ for undirected graph and $|E|$ for directed graph.

Total running time: $O(|V| + |E|)$.

Topological Sorting

An ordering on nodes of a **directed graph** so that **for each** edge **from** v_i **to** v_j in the graph, v_i is before v_j in the ordering.

DAG is guaranteed to have a topological ordering.

Topological sorting is not necessarily **unique**.

Implementation

Based on a **queue**.

Algorithm:

1. Compute the in-degrees of all nodes.
2. **Enqueue** all in-degree 0 nodes into a queue.
3. While queue is not empty
    1. **Dequeue** a node v from the queue and visit it.
    2. **Decrement** in-degrees of node v's neighbors.
    3. If any neighbor's in-degree becomes 0, **enqueue** it into the queue.

Time Complexity

Assume: adjacency list representation

Total running time is $O(|V| + |E|)$.

Minimum Spanning Tree (MST)

A **tree** is an **acyclic, connected undirected** graph.

For a tree, $|E| = |V| - 1$.

Claim: Any **connected** graph with N nodes and $N - 1$ edges is a tree.

$G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$.

A **spanning tree** of a **connected undirected** graph G is a subgraph of G that

1. contains all the nodes of G;
2. is a tree.

Given a weighted, connected, undirected graph $G = (V, E)$, a **minimum spanning tree** T of G is a spanning tree of G whose **sum** of all edge weights is the **minimal**.

Prim's Algorithm

Separate V into two sets:

T: the set of nodes that have been added to the MST.

T': those nodes that have not been added to the MST, i.e., $T' = V - T$.

We keep $P(v)$ for each node v: $(P(v), v)$ is the edge chosen in the MST.

1. **Arbitrarily pick** one node s. Set $D(s) = 0$. For any other node v, set $D(v)$ as **infinite** and $P(v)$ as unknown.
2. Set $T' = V$.
3. While $T' \neq \emptyset$
   1. Choose node v in T' such that $D(v)$ is the **smallest**. Remove v from the set T'.
   2. For each of v's **neighbors** u that is **still** in T', if $D(u) > w(v, u)$, then **update** $D(u)$ as $w(v, u)$ and $P(u)$ as v.

Time Complexity

Method 1: **linear scan** the set T' to find the smallest $D(v)$

- Each time to find minimum: $O(|V|)$
- Maximal number of times to update the neighbors: $|E|$.
  - Each Update cost: $O(1)$

Total runtime: $O(|E| + |V|^2) = O(|V|^2)$.

Method 2: use a **binary heap** to store D(v)'s

- Each time to find minimum: $O(\log|V|)$ (dequeue needs to percolate down)
- Maximal number of times to update the neighbors: $|E|$.
  - Each Update cost: $O(\log|V|)$ (percolate up)

Total runtime: $O(|V| \log|V| + |E| \log|V|) = O((|V| + |E|) \log|V|)$.

Method 3: use a **Fibonacci heap** to store $D(v)$

- Each time to find minimum: $O(\log|V|)$
- Maximal number of times to update the neighbors: $|E|$.
  - Each Update cost: $O(1)$

Total runtime: $O(|V| \log|V| + |E|)$.

- For sparse graphs, i.e., $|E| \approx \Theta(|V|)$, using binary heap has same runtime complexity as Fibonacci heap. The runtime complexity is $O(|V| \log|V|)$

Shortest Path

Shortest path problem: given a weighted graph $G = (V, E)$ and two nodes $s, d \in V$, find the shortest path from s to d.

Unweighted Graphs

For an unweighted graph, **path length** is defined as **the number of edges on the path**.

Use **BFS**.

Weighted Graphs -- Dijkstra's Algorithm

- Works only when all weights are **non-negative**
- A **greedy algorithm** for solving single source all destinations shortest path problem

Keep **distance estimate** $D(v)$ and **predecessor** (the previous node on the shortest path) $P(v)$ for each node v.

1. Initially, $D(s) = 0$; $D(v)$ for other nodes is $+\infty$; $P(v)$ is unknown.
2. Store all the nodes in a set R.
3. While R is not empty
    1. Choose node v in R such that $D(v)$ is the **smallest**. Remove v from the set R.
    2. Declare that v's shortest distance is $D(v)$.
    3. For each of v's **neighbors** u that is **still in** R, **update** distance estimate $D(u)$ and predecessor
        - If $D(v) + w(v, u) < D(u)$, then update $D(u) = D(v) + w(v, u)$ and the predecessor $P(u) = v$.

Time Complexity

Same as Prim's Algorithm

Dynamic Programming

Limitation of Divide and Conquer:

Recursively solving subproblems can result in the **same computations** being repeated when the subproblems **overlap.**

-> Dynamic Programming

- Used when a problem can be divided into **subproblems** that **overlap**.
    - Solve each subproblem **once** and **store** the solution in a table.
    - If a subproblem is encountered **again**, simply look up its solution in the table.
    - **Reconstruct** the solution to the original problem from the solutions to the subproblems.
- The more overlap the better, as this reduces the number of subproblems.

Dynamic programming can be applied to solve **optimization problem**.

- A problem in which some function (called the **objective function**) is to be optimized (usually minimized or maximized) subject to some **constraints**.
- The solutions that satisfy the constraints are called **feasible solutions**.
- The number of feasible solutions is typically very large.
- We obtain the optimal solution by **searching** the feasible solution space.

Takeaway:

- Dynamic Programming is often linked with **Induction**!
- Book-keep partial results to avoid redundant computation!

Example: Matrix-Chain Multiplication

Cost of multiplying two matrices A and B (use the number of scalar multiplications as the complexity measure):

- Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix.
- $C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}$. -> We need q scalar multiplications to calculate C_{ij}.
- C is of size $p \times r$.
- The number of scalar multiplications is pqr.

Cost of multiplication of three matrices $A \times B \times C$:

Suppose A is of size 100×1, B is of size 1×100, and C is of size 100×1.

- If we multiply as $(A \times B) \times C$, the number of scalar multiplications is 20000.
- If we multiply as $A \times (B \times C)$, the number of scalar multiplications is 200.

What is the **best order** of multiplication to **minimize the cost** of multiplying a chain of matrices $A_1 \times A_2 \times \cdots \times A_n$, where A_i is of size $p_{i-1} \times p_i$?

- an optimization problem

Define the **problem** of finding the optimal order to multiply $A_i \times A_{i+1} \times \cdots \times A_j$ as Q_{ij}. The **minimal number** of scalar multiplications is m_{ij}.

If we know the optimized partition position k that in the optimal order for $A_i \times \cdots \times A_j$ the **last** multiplication is $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$, then:

- Q_{ij} can be divided into two subproblems Q_{ik} and $Q_{(k+1)j}$.
- $m_{ij} = m_{ik} + m_{(k+1)j} + p_{i-1} p_k p_j$

To know k, we need to consider all possible divisions $i \leq k \leq j-1$

- Thus, in order to solve Q_{ij}, we need to consider all subproblems Q_{ik} and $Q_{(k+1)j}$, for all $i \leq k \leq j-1$.
- $m_{ij} = \min_{i \leq k \leq j-1}(m_{ik} + m_{(k+1)j} + p_{i-1} p_k p_j)$

Use a **tabular**, **bottom-up** approach:

- Initial situation $m_{11} = m_{22} = \cdots = m_{nn} = 0$.
- In **the l-th round**, we compute $m_{1(l+1)}, m_{2(l+2)}, \ldots, m_{(n-l)n}$.
- Finally, we compute m_{1n}.
- To obtain the multiplication order, we also **record the partition** k which gives the minimal m_{ij} as s_{ij}.

Time Complexity

Get the **minimum number** of scalar multiplications:

- We need to obtain all m_{ij} and s_{ij}, for $1 \leq i \leq j \leq n$.
  - $O(n^2)$ records
  - Each m_{ij} is the minimum of $O(n)$ terms.
- Total time complexity is $O(n^3)$.

Obtain (cout) the optimal order: $O(n)$