Direct Addressing - key is index into array => O(1) lookup
Hash table:
-hash function maps key to index in table
-if |universe of keys| > # table entries then hash functions collision are guaranteed => need

Collision resolution - how to handle collisions
Changing - table entries are linked lists, colliding elements are elements of the same list
    load factor: alpha = N/M = average number of elements per bucket
    For perfect hash function (ie: Pr[h(i) = h(j)] = 1/M)
    -search takes O(1 + alpha)
    -insert takes O(1) [just have to append to linked list]
    -searches gradually take longer as load factor increases
Open addressing - if collision occurs keep searching for empty space
    Linear probing: ith probe at [h(k) + i] % M
        pros: if empty space exists, guaranteed to find it
        cons: clusters for in table => decreased performance
            cluster - group of adjacent occupied cells
            if first half full then insert is O(n)
    Quadratic probing: ith probe at [h(k) + i^2] % M
        -mitigates clustering problem (still can have 2nd order clusters)
    Double hashing:
        -use two hash functions
        -ith probe at [h_1(k) + h_2(k)*i] % M
    Cons:
        -to disambiguate between empty slot and one that used to be occupied need ghost
        -must add ghost elements when an element is deleted
Dynamic hashing - increase table size and rehash when load factor get too high

Hashing
Pros: O(1) insert, search, remove (if done right)
Cons:
    -table does not maintain element order ie: nth element is O(n)
    -requires more memory than trees (in order for load factor to be small)
Hash Functions:
Hash code: maps key to integer
Compression function: maps integer to index in table (use modulus)
-should be deterministic and fast
-want to minimize collisions
ex:
Hash(i) -> i % M [M = table size)
Hash(i) -> floor(i * alpha) % M
Hash(c_0 || c_1 || … c_{l-1}):

return $c\_0*a^{l-1} + \ldots + c_{l-1}a^0$

Amortized Analysis:
-consider average cost over a sequence of operations
-occasionally pay high cost (ex: rehashing), but over sequence of operations, average still ok

Trees
-direct connected acyclic graph
-each node has unique parent (except root which has none)
Node: (for binary tree)
value
left child
right child
Internal node - node with children
Leaf node - node with no children
Binary tree - each none has at most 2 children
proper binary tree - all internal nodes have two children
Complete binary tree - all levels have max # nodes possible except for lowest which is filled left to right
max height = max depth

Implementation
-array based (think binary heaps)
-not space efficient for sparse trees [$O(2^n)$ for "linked list" tree]
-pointer based

General Tree - can have arbitrarily many children
Converting general tree to binary tree (think pairing heaps)
-left pointer points to first child, right pointer points to next sibling

Tree Traversal
Preorder: Node, Left, Right
Inorder: Left, Node, Right
Post order: Left, Right, Node
-forms of depth first search (the only difference between these modes is when a value is handled)
-pre order, post order and level order generalize to general trees
Level order: each level traverse left to right and then top down
-think breadth first search, use queue

Binary Search Trees
-invariant left child's key <= node's key <= right child's key [if left and right children exist]
-for complete binary search tree searches take O(log(n)) time, O(n) for "linked list" tree O(n) for unordered trees

-order => finding nth largest possible (in order traversal that stops at nth element) Q: how to do this in < O(n)
Given set of keys, if you always insert the largest or smallest left => tree becomes zig-zagged listed list => search, insert, O(n)
Complexity depends on height => want balanced tree with low height
Deletion:

- if node has <= 1 child then deletion easy
- if node has 2 children, swap with either its inorder successor, or its inorder predecessor
  then remove (inorder predecessor guaranteed not to have right child)

Insert, delete, search O(n) in worst case :(
=> need tree that maintains balance of tree
tree vs hash table: search tree maintains elt order

AVL Trees
balance(node) = height(left child) - height(right child)
invariant: for each node, -1 <= balance(node)) <= 1

Claim: if invariant holds, then height of tree is O(log N)
Let n(h) = min number of nodes for tree of height h
n(0) = 0, n(1) = 1
For h > 1 minimal tree formed by taking minimal trees whose heights differ by 1
n(h) = n(h - 1) + n(h - 2)
=> Fibonacci numbers are recurrences closed form solution
=> $n(h) \approx \frac{\Phi^n}{\sqrt{5}}$

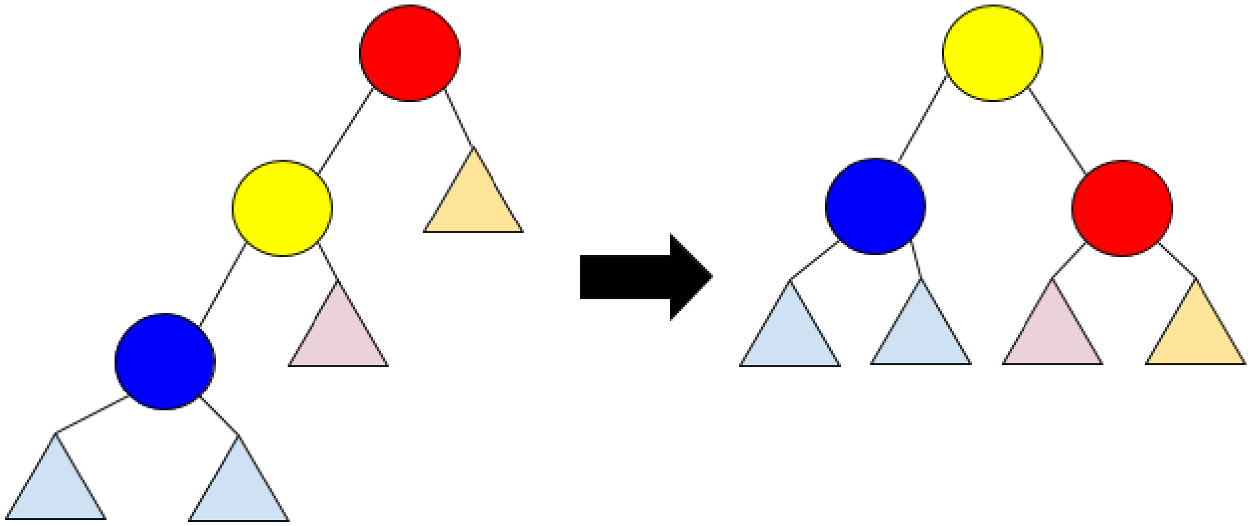
So n(h) = Omega(2^h) => h = O(log(n)) TODO: check this

Corollary: search is O(log N)

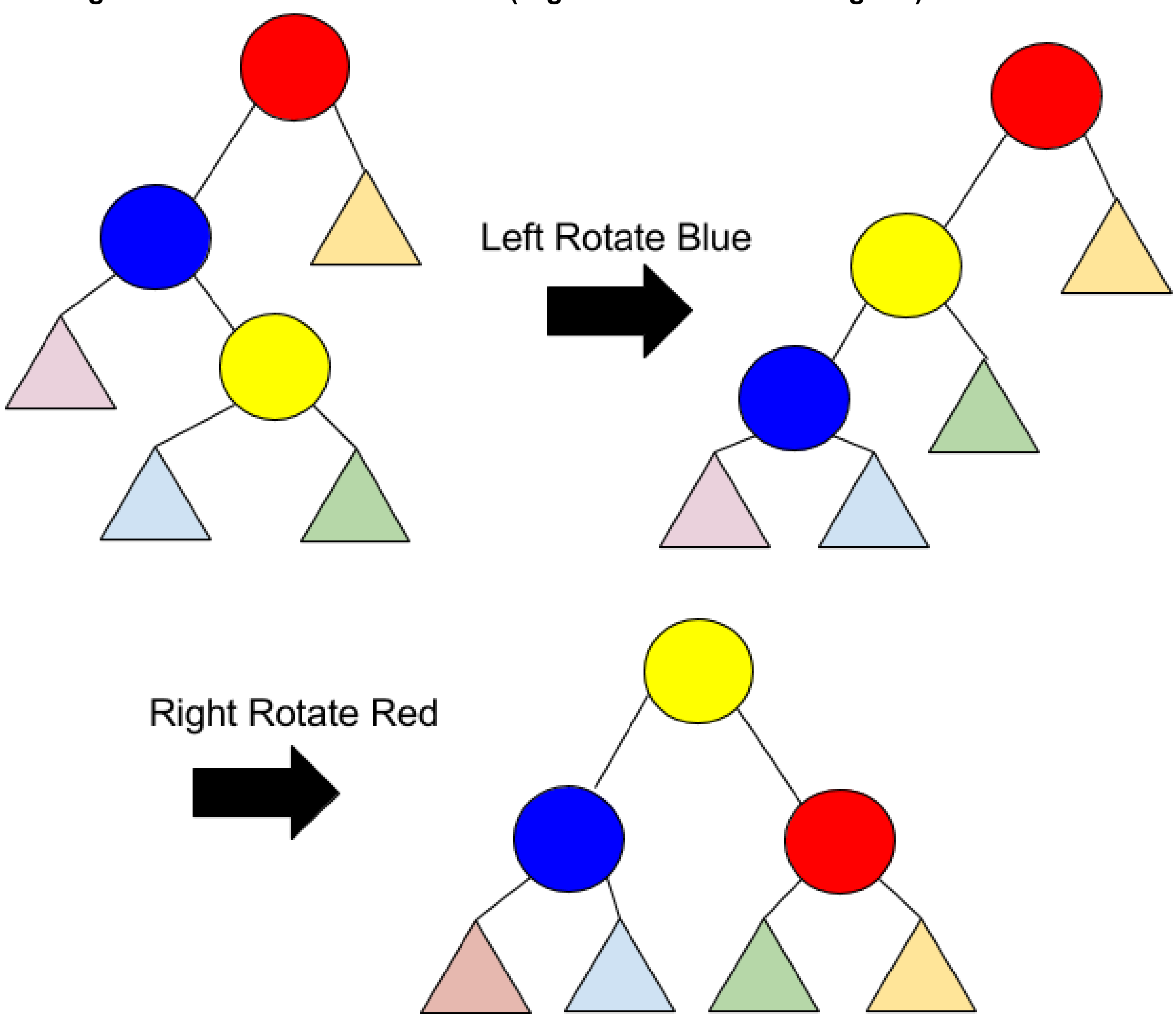
Problem: Normal insert or delete could make tree unbalance.
Solution: starting with newly inserted or deleted node, more up tree and rebalance using rotations

Rotations
4 cases

Right Rotation (Left Ration analogous)



Left Right Rotation ie: double rotation (Right Left Rotation analogous)



insertion: insert as normal then rebalance
deletion: delete as normal then rebalance
search, min, max, successor, predecessor - same as BST
Time to rebalance after insert or delete is O(log(n)) => insert, delete O(log(n))

Graphs
Def: Let V be a set of vertices and $E \subseteq V \times V$ be a set of edges connecting these vertices. Then G = (V, E) is called a graph

Directed graph - vertices that comprise edges are ordered ie: edges have directions
Undirected graphs - edges do not have directions
Weighted graph - edges have weights

Simple graph - no parallel edges of self loops
Multi graph - allows parallel edges

Representations:
adjacency list: array with entry for each vertex, array entries are lists of elements adjacent to vertex
-requires O(|V| + |E|) space [technically |V| + 2|E| for undirected graphs]
-check for existence of edge takes O(|V|) worst case
adjacency matrix: m_{ij} = 1 iff edge from i to j
-matrix symmetric for undirected graphs
-$O(V^2)$ space
-O(1) time to check for edge
-store edge weights for weighted graphs or infinity if edge does not exist

Sparse graph: $|E| << |V^2|$ or $|E| \sim |V|$
    -use adjacency list
Dense graph: $|E| \sim |V^2|$
    -use adjacency matrix

Path - sequences of vertices where each is connected to the previous
Simple path - path the does not contain the same vertex twice
Cycle - path from a vertex to itself (removing starting/ending vertex should yield simple path)
Connected - paths exist between all pairs of vertices

Depth First Search
put starting node on stack
while stack not empty
    visit top node (and pop it from stack)
    add unvisited neighbors of node to stack

Visits each vertex once, follows each edge once => O(V + E) for adjacency list, O(V^2) for adjacency matrix
Always finds path between nodes of one exists

Breadth First Search
put starting node at front of queue
while queue not empty
    visit front element (and pop from queue)
    add unvisited neighbors to queue

Finds shortest path to node if all edges have same weight
Use BFS to print tree in level order
Sample complexity analysis as DFS

Minimum Spanning Tree
Problem: Given G = (V, E), find subset E' of E such that G' = (V, E') is a tree with minimal edge weight [assuming G is connected, if G not connect, then find minimum spanning forest]

-For unweighted graphs all spanning trees are minimum spanning trees
-All MSTs have V - 1 edges (the minimum needed to connect all vertices)

Making Change
Using coins with values: 1, 7, 15 make 21 cents in change
15,1,1,1,1,1,1 <= greedy
7,7,7 <= optimal

Knapsack
Integer weights and capacity knapsack

```
knapsack(capacity, items)
    max_val = [0] * (capacity + 1)
    for i in range(1, capacity + 1)
        //try adding each item
        for w, v in items
            if w <= i and max_val[i - w] + v > max_val[i]
                max_val[i] = max_val[i - w] + v
    return max_val[capacity]
```