Algorithm analysis

Big O: Let $f(n)$ and $g(n)$ be functions that map positive integers to real positive numbers.
$f(n)$ is $O\big(g(n)\big)$ if $\boldsymbol{f(n) \le c \cdot g(n), n \ge n_0}$ for some $c \in \mathbb{R}$ and $n_0 \in \mathbb{Z}^+$

Big Omega (Ω): Let $f(n)$ and $g(n)$ be functions that map positive integers to real positive numbers.
$f(n)$ is $\Omega\big(g(n)\big)$ if $\boldsymbol{f(n) \ge c \cdot g(n), n \ge n_0}$ for some $c \in \mathbb{R}$ and $n_0 \in \mathbb{Z}^+$

Big Theta(Θ): Let $f(n)$ and $g(n)$ be functions that map positive integers to real positive numbers.
$f(n)$ is $\Theta\big(g(n)\big)$ if $\boldsymbol{c' \cdot g(n) \le f(n) \le c'' \cdot g(n), n \ge n_0}$ for some $c', c'' \in \mathbb{R}$ and $n_0 \in \mathbb{Z}^+$

Loop invariant: Any predicate/condition that holds true for every iteration of a loop

Recursion

- Understand differences between linear recursion, binary recursion, multiple recursion in terms how many recursive class each function makes

Trees

Tree: If a tree is non-empty, there exists a root node with no parent. Each child of that root is in turn the root of another sub-tree

Descendant: Any node lower by 2 or more levels	**Ancestor:** Any node higher by 2 or more levels
Depth: Number of levels separating node from the root	**Height**: The maximum levels of the tree (excluding root)

Proper binary tree: Each node must have either 0 or 2 children

Complexities: depth→ $O(dp+1)$, height → $O(n)$

Preorder: Visit node before left-right children
Postorder: Visit left-right children before node
Inorder: Visit left child, node, right child

Permutations of n elements in a BST: Given by the n_{th} Catalan number, $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{(n+1)!n!}$

Sorted maps and Balanced Search Trees

Map: Stores key-value pairs. Also known as an associative array

Balanced search tree: A binary tree is balanced if for every internal position p, the heights of p's children differ at most by 1

Deletion policy: If internal node with 2 children, replace with inorder predecessor

AVL tree: Balance after each insertion/deletion with trinode restructuring

2,4 tree: *Size property* – Every internal node has at most 4 children. *Depth property* – all external nodes have the same depth
Insertion: On overflow, promote third child (k3 is promoted from (k1, k2, k3, k4))
Deletion: Remove node, replace and cascade if internal, until we remove a key from an internal node whose children are external nodes. If underflow and sibling is a 3/4-node, transfer. Otherwise if sibling is a 2-node, merge both to form a 3-node

Red-black tree: *Root property* – root is black. *External property* – every external sentinel node is black. *Red property* – the children of a red node are black. *Depth property* – All external nodes have the same black depth
Insertion: x = inserted node, y = parent, z = grandparent. If y is red, double red. If y has black sibling, trinode restructuring on z to form new 4-node. Otherwise, recolor y, y's sibling, and z, propagating if necessary.
Deletion: Delete normally and cascade until reaching a node with external child. If red, no issue. If black with one red child, promote red child and color black. Otherwise, double black. p = promoted child, y = sibling of p, z = common parent of p and y.
1: If y is black with a red child x, trinode restructuring on z (2,4 transfer)
2 :If y is black with black children, recolor y to red, *p* to black and *z* black or double black and propagate (2,4 fusion)
3: If y is red, z must be black. Rotate such that y is the parent of z, recolor y to black, z to red. Go to case 1 or 2

Complexities: search, insertion, removal→ $O(\log(n))$ for AVL (restructuring) and RBT (recoloring)
For red-black tree insertion, ≤ 1 trinode restructure. For red-black tree deletion, ≤ 2 trinode restructures

Hash Tables and Maps

Hashcode implementations: XOR, polynomial function, bitwise cyclic shift

Probing methods: Given a hashcode $h(k)$, we probe $A[\big(h(k)+f(i)\big)\%N]$ for $i = 1,2,\ldots,N$:
Linear probing → $f(i) = i$, Quadratic probing → $f(i) = i^2$, Double hashing → $f(i) = i \cdot h'(k)$

Heaps and Priority Queues

Heap: *Heap-order property* – for every position p,the key is greater than its parent. *Complete binary tree property* – every level of the tree has the maximal number of nodes possible, and the remaining noes reside in the leftmost possible positions.

Complexities: insertion+removal→ $O(n)$ for list implementations, $O(\log(n))$ for heap. heap insertion→ $O(\text{nlog}(n))$, heapify→ $O(n)$

Graphs

Graph: A tuple $G = (V, E)$ where V is the set of vertices and E is the set of edges

Path: A set of alternating vertices and edges from u to v, where each edge is incident on the immediate predecessor and successor vertices. *Simple* if no repeated vertices

Cycle: A path from u to itself, involving at least one other vertex. *Simple* if no repeated vertices

Connectedness: A graph is connected if there is a path between any 2 vertices. *Strongly connected* if for any pair of vertices u and v, u is reachable from v and v is reachable from u

Subgraph: $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \in V$ and $E' \in E$. *Spanning subgraph* if $V' = V$,

Degree: Number of edges incident on a vertex. *In-degree* - incoming edges, *Out-degree* – outgoing edges

Edge list: n vertices and m edges stored in separate unordered lists. Limitations in processing edges for a given vertex

Adjacency list: n vertices stored in an unordered list, where each vertex maintains its own unordered list of all incident edges

Adjacency map: n vertices stored in an unordered list, each vertex maintains a map where key=adjacent vertex and value=edge

Adjacency matrix: 2D array A of $n \times n$, where $A[u][v]$ holds a references to the (u, v) edge if it exists

Topological ordering: Any given graph G has a topological ordering if and only if it is acyclic. Its vertices $V_1, V_2, \dots, V_n$ are ordered such that for every edge (V_i, V_j) of G, we have $i < j$

Minimum spanning tree: Given an undirected, weighted graph G, a minimum spanning tree of G is a tree T containing all the vertices in G, that minimizes the sum of weights, $\sum_{(u,v \in T)} w(u, v)$. If all edges of G has distinct weights, the minimum spanning tree is unique

Method	**Edge List**	**Adj List**	**Adj Map**	**Adj Matrix**
numVertices(), numEdges()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	O(1)	O(1) expected	O(1)
outDegree(v), indegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v), incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertVertex(x), insertEdge(u, v, x), removeEdge(e)	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Prim-Jarnik: Start from any vertex as its own graph, build up by adding lowest cost edges to undiscovered nodes

Kruskal: Each node is its own cluster at the beginning. Iteratively consume lowest cost edges that connect different clusters