

ALGORITHMIC ANALYSIS

Big O: $f(n) \in O(g(n)) \leftrightarrow \exists c \in \mathbb{R}, n_0 \in \mathbb{Z}^{\geq 0}$ s.t. $\forall n \geq n_0, f(n) \leq c \cdot g(n)$

Big Ω : $f(n) \in \Omega(g(n)) \leftrightarrow \exists c \in \mathbb{R}, n_0 \in \mathbb{Z}^{\geq 0}$ s.t. $\forall n \geq n_0, f(n) \geq c \cdot g(n)$

Big Θ : $f(n) \in \Theta(g(n)) \leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$

$\leftrightarrow \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{Z}^{\geq 0}$ s.t. $\forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Growth Rates: $1 \rightarrow \log(n) \rightarrow n \rightarrow n \log(n) \rightarrow n^2 \rightarrow n^3 \rightarrow c^n \rightarrow n!$

RECURSION ANALYSIS

Case 1: work per call follows pattern - $T(n) = T(n-1) + O(1)$

Case 2: work per level is the same - $T(n) = 2T(\frac{n}{2}) + O(n)$

Case 3: work per call is the same - $T(n) = 2T(n-1) + O(1)$

SORTING ALGORITHMS

Sort	Worst	Best	Expected
Bubble $A[i] \xleftrightarrow{\text{swap}} A[i+1]$	$O(n^2)$	$O(n)$ <i>pre-sorted</i>	$O(n^2)$
Insertion <i>by next unsorted</i>	$O(n^2)$	$O(n)$ <i>pre-sorted</i>	$O(n^2)$
Selection <i>by smallest</i>	$O(n^2)$	$O(n)$ <i>pre-sorted</i>	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick <i>3-partition</i>	$O(n^2)$	$O(n)$ <i>uniform list</i>	$O(n \log n)$
Radix	$O(d(n+N))$	$O(d(n+N))$	$O(d(n+N))$

BASIC DATA STRUCTURES

Amortisation: $T(n) \div n$, where n = no. of operations

	SPACE	get	add	remove
Dynamic Array	$O(n)$	$O(1)$	$O(n)^*$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Stack - LIFO	SPACE: $O(n)$	$O(1)^1$	$O(1)$	
Queue - FIFO	SPACE: $O(n)$	$O(1)$	$O(1)$	

¹ amortised if array-based implementation

BINARY TREES

Proper Binary Tree: internal nodes have exactly 2 children

Complete Binary Tree: levels $0 \rightarrow h-1$ full, level h left-most

Pre-Order	In-Order	Post-Order
self \rightarrow left \rightarrow right	left \rightarrow self \rightarrow right	left \rightarrow right \rightarrow self

Properties:

- Full level l has 2^l nodes (note $l \geq 0$)
- Max no. of nodes = $2^{\text{MAX}} - 1$, max internal nodes = $2^{\text{MAX}-1} - 1$
- h = no. of edges from lowest leaf = $\lfloor \log_2 n \rfloor$

⚠ *care with null leaves in implementation vs. conceptual tree*

- l & d use 0-based index.

HEAPS & PRIORITY QUEUES

Binary Heap: complete binary tree with heap property order

$\text{key}(\text{node}) \leq \text{key}(\text{parent}(\text{node}))$ OR $\text{key}(\text{node}) \geq \text{key}(\text{parent}(\text{node}))$

Insertion: insert at last node then **upHeap** -

- Check $k < \text{parent} \wedge k \neq \text{root}$
- Swap $k \leftrightarrow \text{parent}$ greater than current
- Repeat until $k \geq \text{parent} \vee k = \text{root} \rightarrow O(\log n)$

RemoveMin: swap root \leftrightarrow last node then **downHeap** -

- Check $k > \text{child}_{\text{left}}$ or right
- Swap $k \leftrightarrow \text{child}_{\text{less than current}}$
- Repeat until $k > \text{child}_{\text{both}} \vee k = \text{leaf} \rightarrow O(\log n)$

Array-Based: left child @ $2i+1$ & right child @ $2i+2$

Heap Sort: for a in $A \rightarrow$ add to heap \rightarrow removeMin back to A

Bottom Up Heap Construction: get $l_{\text{MAX}} = \lfloor \log_2 n \rfloor \rightarrow$ get no. of leaves on bottom level (fill remaining with nulls) \rightarrow add nodes to merge heaps

MAPS - HASH TABLES

General Summary: pre-hash \rightarrow compress \rightarrow handle collisions \rightarrow rehash

Complexities: $O(1)$ expectation, $O(n)$ worst-case

Pre-hash to Hash Code:

- Component sum: e.g. sum of all char in string (collision risk)
- Polynomial accumulation: $p(z) = a_0 z^0 + a_1 z^1 + \dots z \in \mathbb{Z}$
- Cyclic shift: replace z with bit-shifted version (e.g. $z \ll 5$)

Compress to Hash Value: fow below, N = table size $\wedge N \in \mathbb{Z}^{\text{prime}}$

- Division: $h(k) = k \bmod N$
- MAD: $h(k) = ((ak + b) \bmod p) \bmod N$
 $p > N \wedge p \in \mathbb{Z}^{\text{prime}}, a \in [1, p-1], b \in [0, p-1]$

Collision Handling: separate chaining vs. open addressing

- Probe idx computed as $(h(k) + f(i)) \bmod N$ for $i = 0, 1, 2, \dots$
- 1st probe is just $h(k)$ hence $i = 0$ for open addressing, below:

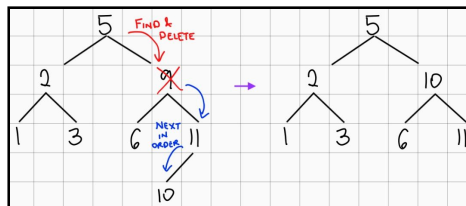
	Linear	Quadratic	Double Hash
$f(i)$	i	i^2	$i \times d(k)$

Load Factor $\alpha = \frac{n}{N} \rightarrow 0.8 \leq \alpha \leq 1$ chaining & $\alpha < \frac{2}{3}$ open addressing

- Expected no. of probes = $\frac{1}{1-\alpha}$

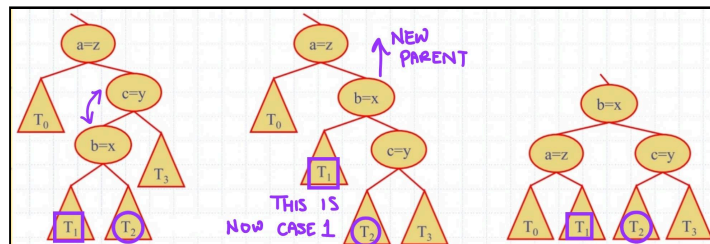
SEARCH TREES

BST: $O(\log n)$ expected get, add, remove (in-order traversal), $O(n)$ worst



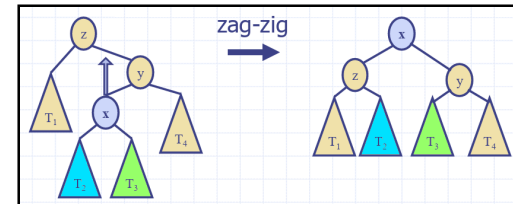
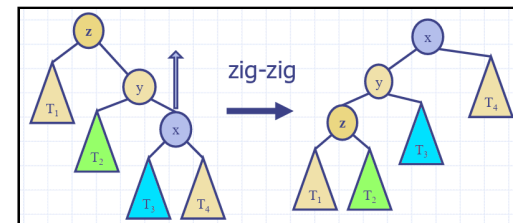
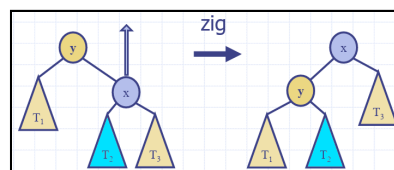
AVL Trees: BST that doesn't degrade to $O(n)$

- Insertion: re-balance at **first** unbalanced node from bottom
- Deletion: after BST deletion, re-balance upwards from bottom
- Tri-node inputs: parent, child + grandchild of greater height



Splay Trees: $O(n)$ worst, $< O(\log n)$ best (popular nodes), $O(\log n)^*$

- get(K) - splay found node, or last accessed node before null leaf
- insert(K, V) - splay inserted/updated node
- remove(K) - splay original parent of removal node



GRAPHS

- Incident edge** = connected to vertex, **adjacent vertices** = connected by edge
- Vertex Degree** = no. of incident edges (in vs out degree for directed)
- Simple Graph** = no loops or multi-edges
- Simple Path** = distinct edges AND vertices
- Simple Path** = distinct edges AND vertices
- Simple Cycle** = simple path with same start/end points
- Subgraph** = subset of vertices/edges
- Spanning Subgraph** of G = subgraph that contains *all* vertices of G
- Connected Graph** = \exists simple path between any 2 vertices
- Unrooted Tree** = connected graph with no simple cycles (can be spanning)
- Forest** = unconnected graph with no simple cycles (can be spanning)
- Fully Dense Graph:** $\sum \text{deg}(v) = 2 \times |E|$ in undirected graph $\rightarrow |E| \leq n(n-1) \div 2$
- Graph Density:**
 - $\frac{2 \times |E|}{n(n-1)}$ undirected or $\frac{|E|}{n(n-1)}$ directed
 - $|E| \sim O(n)$ sparse or $|E| \sim O(n^2)$ dense
- Graph Representations:**
 - Edge list: simple list of edge pointers only $[(e1), (e2), \dots]$ or $[(v1, v2), \dots]$
 - Adjacency List (or *map*): for each vertex, store adjacent vertices OR edges - $[(v1: e1, e2), (v2: e1, e3) \dots]$ or $[(v1: v2, v3), (v2: v1, v4) \dots]$
 - Adjacency Matrix: $V \times V$ grid where $(u, v) = 1 \leftrightarrow$ edge exists

Feature	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
outgoingEdges(v)	m	$\text{deg}(v)$	n
incomingEdges(v)	m	$\text{deg}(v)$	n
getEdge(v, w)	m	$\min(\text{deg}(v), \text{deg}(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\text{deg}(v)$	n^2
removeEdge(v)	1	1	1
Good for	small graphs	sparse graphs	dense graphs

- DFS:** track unexplored + discovery + back edge, unexplored + explored vertex
 - Initialize an empty stack, and push node 0 into it.
 - Pop a vertex, mark it as visited only if not already visited.
 - Add the visited vertex to the result collection.
 - Push all unvisited adjacent vertices to the stack in reverse order.

Properties of DFS/BFS traversal:

- Back edge (DFS): edge to an ancestor (visited) vertex
- Cross edge (BFS): edge to a visited vertex, either same or earlier level
- Traversal visits all edges & vertices of connected component only
- Discovery edges from DFS/BFS form spanning tree of connected component
- DFS and BFS $O(n + m)$

Digraphs

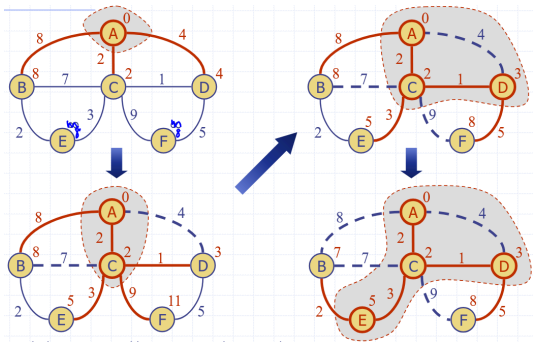
- If G is simple digraph $\rightarrow m \leq n(n - 1)$
- Directed DFS algorithm tracks discovery, back, forward, cross edges
- DAG = Directed Acyclic Graph
- **Topological Sort** = linear ordering of vertices such that for every directed edge (u, v) , $u < v$

Single Source Shortest Path (SSSP) for Weighted Graphs

- Subpath of a shortest path is itself a shortest path
- \exists tree of shortest paths from a vertex to all others

Dijkstras Algorithm: assumes connected, undirected and non-negative edge weights

- Tracks distance from source $d(v)$ for each vertex
- Relaxation of edge e updates distance:
 $d(v) \leftarrow \min(d(v), d(u) + \text{weight}(e))$



Shortest Paths on DAGs

- Negative weights OK (no cycles to loop infinitely)
- Visit topological order, relax all outgoing edges for each vertex
- Does not require additional data structures

Minimum Spanning Trees: spanning tree w/ min edge weight sum

Prim-Jarnik's Algorithm: similar to Dijkstra's algorithm, except update step doesn't consider dist. from source - just edge weights. $d(v)$ = smallest edge weight connected v to the current cloud Add vertex u to the cloud which has smallest $d(u)$

Kruskal's Algorithm: Start with single-vertex clusters. Store PQ of edge weights. Extract edges in increasing weight order, "accept" edge only if it connects distinct clusters.

PSEUDOCODE

ALGORITHM DFS-RECURSIVE (G, v):

Input: Graph G and a vertex v of G
Output: Collection of vertices reachable from v + their discovery & back edges
Mark vertex v as visited
for all $e \in G.outgoingEdges(v)$ **do**
 if e is not explored **then**
 $w \leftarrow G.opposite(v, e)$
 if w has not been visited **then**
 Record edge e as discovery edge **for** vertex w
 DFS(G, w)
 else
 Mark e as a back edge **for** vertex w

ALGORITHM BFS (G, u):

Input: Graph G and a vertex u of G
Output: Collection of vertices reachable from u + their discovery & cross edges
 $Q \leftarrow$ new empty queue
 $Q.enqueue(u)$
Mark vertex u as visited
while $Q.isEmpty()$ **do**
 $v \leftarrow Q.dequeue()$
 for all e in $G.incidentEdges(v)$ **do**
 if e is not explored **then**
 $w \leftarrow G.opposite(v, e)$
 if w has not been visited **then**
 Record edge e as discovery edge **for** vertex w
 $Q.enqueue(w)$
 Mark vertex w as visited
 else
 Mark e as a cross edge

ALGORITHM TOPOLOGICALDFS (G, v):

Mark vertex v as visited
for all e in $G.outgoingEdges(v)$ **do**
 $w \leftarrow G.opposite(v, e)$
 if w has not been visited **then**
 { e is a discovery edge }
 topologicalDFS(G, w)
 else
 { e is a forward or cross edge }
 Label v with topological number n
 $n \leftarrow n - 1$

ALGORITHM DIJKSTRADISTANCES (G, s):

$P \leftarrow$ new heap-based priority queue
for all v in $G.vertices()$ **do**
 if $v = s$ **then**
 setDistance($v, 0$)
 else
 setDistance(v, ∞)
 $P.insert(\text{getDistance}(v), v)$
 while $P.isEmpty()$ **do**
 $u \leftarrow P.removeMin()$
 for all e in $G.incidentEdges(u)$ **do**
 { relax edge e }
 $z \leftarrow G.opposite(u, e)$
 $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$
 if $r < \text{getDistance}(z)$ **then**
 setDistance(z, r)
 $P.replaceKey(\text{getLocator}(z), r)$

ADT METHODS

Stack	push(V)	pop()	top() or peek()
	enqueue(V)	dequeue()	front() or peek()
Queue			
Priority	insert(K, V)	removeMin()	min()
Q	getKey()	getValue()	compareTo(Entry)
Entry	get(K)	put(K, V)	remove(K)
Map	entrySet()	keySet()	values()
	numVertices()		vertices()
	numEdges()		edges()
	outDegree(V)		inDegree(V)
Graph	outgoingEdges(V)		incomingEdges(V)
	insertVertex(x)	insertEdge(V1, V2, x)	
	removeVertex(V)	removeEdge(E)	
	getEdge(V1, V2)	endVertices(E)	
	opposite(V, E)		