Algorithmic Analysis

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

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

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

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

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

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 $A[i] \overset{swap}{\leftrightarrow} A[i+1]$	$O(n^2)$	$O(n)$ *pre-sorted*	$O(n^2)$
Insertion *by next unsorted*	$O(n^2)$	$O(n)$ *pre-sorted*	$O(n^2)$
Selection *by smallest*	$O(n^2)$	$O(n)$ *pre-sorted*	$O(n^2)$
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Quick *3-partition*	$O(n^2)$	$O(n)$ *uniform list*	$O(n\log n)$
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 $0 \to h-1$ full, level h left-most

Pre-Order	In-Order	Post-Order
self $\to$ left $\to$ right	left $\to$ self $\to$ right	left $\to$ right $\to$ self

Properties:

- Full level l has 2^l nodes (note $l \geq 0$)
- Max no. of nodes = $2^{l_{\text{MAX}}} - 1$, max internal nodes = $2^{l_{\text{MAX}}-1} - 1$
- 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

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

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

1. Check $k < parent \wedge k \neq root$
2. Swap $k \longleftrightarrow parent_{\text{greater than current}}$
3. Repeat until $k \geq parent \vee k = root \longrightarrow O(\log n)$

RemoveMin: swap root $\longleftrightarrow$ last node then **downHeap** -

1. Check $k > child_{\text{left or right}}$
2. Swap $k \longleftrightarrow child_{\text{less than current}}$
3. Repeat until $k > child_{\text{both}} \vee k = leaf \longrightarrow O(\log n)$

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

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

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

Maps - Hash Tables

General Summary: pre-hash $\to$ compress $\to$ handle collisions $\to$ 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: $p(z) = a_0 z^0 + a_1 z^1 + \ldots \; z \in \mathbb{Z}$
- 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 $i = 0, 1, 2\ldots$
- 1st probe is just $h(k)$ hence $i = 0$ for open addressing, below:

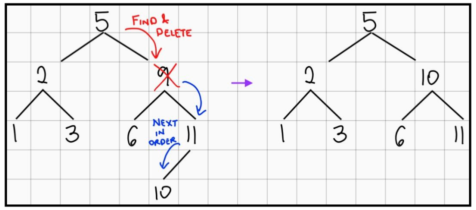
	Linear	Quadratic	Double Hash
$f(i)$	i	i^2	$i \times d(k)$

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

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

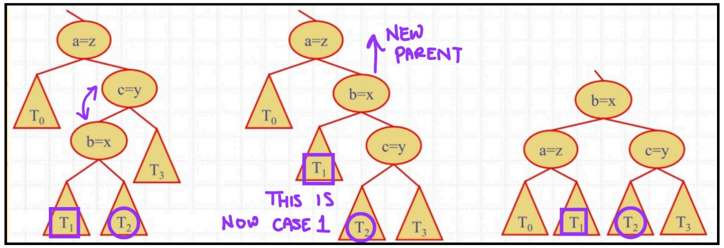
Search Trees

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



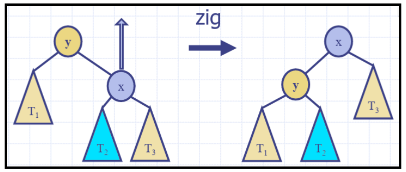
AVL Trees: BST that doesn't degrade to $O(n)$

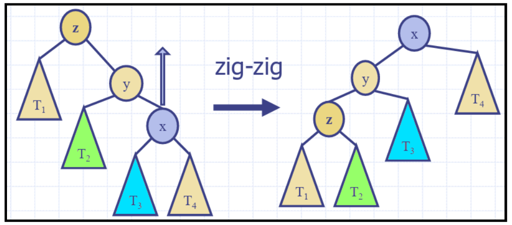
- Insertion: re-balance at **first** unbalanced node from bottom
- Deletion: after BST deletion, re-balance upwards from bottom
- Tri-node inputs: parent, child + grandchild of greater height

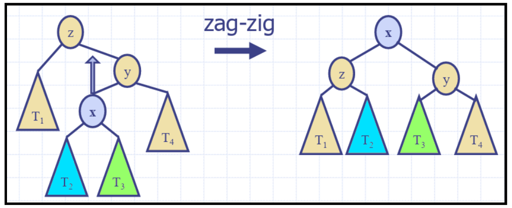


Splay Trees: $O(n)$ worst, $< O(\log n)$ best (popular nodes), $O(\log n)$*

- `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:** $\sum deg(v) = 2 \times |E|$ in undirected graph $\to |E| \leq n(n-1) \div 2$
- **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), ...]* or *[(v1, v2), ...]*
  - Adjacency List (or *map*): for each vertex, store adjacent vertices OR edges - *[(v1: e1, e2), (v2: e1, e3) ...]* or *[(v1: v2, v3), (v2: v1, v4) ...]*
  - 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	deg(v)	n
incomingEdges(v)	m	deg(v)	n
getEdge(v, w)	m	min(deg(v),deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	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
  1. Initialize an empty stack, and push node 0 into it.
  2. Pop a vertex, mark it as visited only if not already visited.
  3. Add the visited vertex to the result collection.
  4. 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

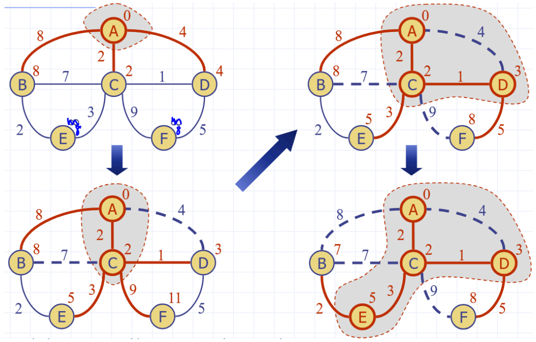
- If G is simple digraph $\rightarrow m \leq n(n-1)$
- Directed DFS algorithm tracks discovery, back, forward, cross edges
- **DAG** = Directed Acyclic Graph
- **Topological Sort** = linear ordering of vertices such that for every directed edge (u,v), $u<v$

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 Dikjstra'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 $d(u)$

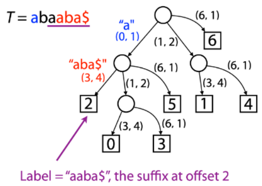
Kruskal's Algorithm: Start with single-vertex clusters. Store PQ of edge weights. Extract edges in increasing weight order, "accept" edge only if it connects distinct clusters.

Strings, Patterns, Tries

Tries: Edge = char, each node has a map/sorted list of edges for traversal. To check if string present, traverse until leaf hit (leaf not required for substring/prefix). $O(n)$ worst-case.

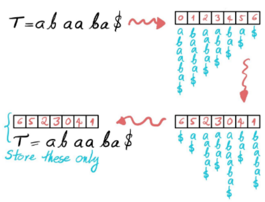
Suffix Tries: Append special char $ to string end. Insert all suffixes of string into trie. $O(m)$ time to search for pattern of length m. $O(n^2)$ time to construct suffix trie for string of length n. Note: a substring is a prefix of a suffix. $O(n^2)$ worst space (no prefix sharing, all distinct chars), $O(n)$ best. Common uses:

- Check for substring P - traverse - true if no fall-off - $O(|P|)$
- Count substring occurrences - 0 if fall-off, else it is no. of leaf descendants of last node - $O(|P|)$
- Longest repeated substring - find **deepest** internal node with >1 children



Suffix Trees:
Compress paths
Replace edge labels by (offset, length) pairs
Keep T stored separately for ref
Store "final" offsets at leaves
$O(m^2)$ time to construct
$O(m)$ final space, $O(m^2)$ to build

- Check P substring of T - $O(|P|)$ worst
- Count no. of P in T - $O(|P|+k)$ worst - traverse until fall off or found - count = k leaf descendants of last node
- **TIP:** go thru T, skip to building compressed paths, go thru suffixes of each distinct char in T, start from shortest suffix

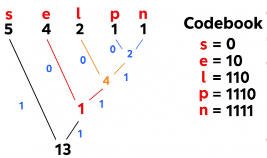


Suffix Arrays:

- Write all suffixes of T w/ offsets
- Sort suffixes lexicographically ($ is 1st)
- Store offsets in array

- Construction is $O(|T|^2 \cdot \log(|T|))$ as we sort $|T|$ suffixes, each up to $|T|$ length
- Check substring P of T - $O|P| \cdot \log(|T|)$ - binary search and compare suffixes at each offset vs. P
- Count no. of P in T - binary search for lower bound (entry **must** start w/ P) and upper bound - count = upperIndex - lowerIndex + 1 (**not** offsets)

Huffman Encoding

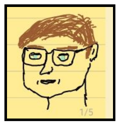


Codebook
s = 0
e = 10
l = 110
p = 1110
n = 1111

Given a set n of positive weights (e.g. char counts in T)
Compute set of n codeword lengths s.t. sum is minimal
i.e. use to get smallest number of bits to represent T

- $O(n + d \log d)$ where $n = |X|$, d = distinct char #
- Produce codebook from tree by traversing left = 0, right = 1
- As each codeword can be variable length, no codeword is a prefix of another, e.g. 1101 may be 1-101 or 11-01

1. Get weights of each distinct char in T
2. Start with each tree = single node. Put into PQ -> (weight, tree)
3. RemoveMin twice -> merge (w1, T1), (w2, T2)
4. Put back into PQ -> (w1 + w2, new tree with T1 left, T2 right)
5. Repeat until only 1 tree remains - return this tree



ADT Methods

Map	`get(K)` `entrySet()`	`put(K, V)` `keySet()`	`remove(K)` `values()`
Graph	`numVertices()` `numEdges()` `outDegree(V)` `outgoingEdges(V)` `insertVertex(x)` `removeVertex(V)` `getEdge(V1, V2)` `opposite(V, E)`	`vertices()` `edges()` `inDegree(V)` `incomingEdges(V)` `insertEdge(V1,V2,x)` `removeEdge(E)` `endVertices(E)`	

Pseudocode & Scuffed Helpers

Algorithm Huffman (X):
Input: String X of length n
Output: Optimal encoding tree **for** X
$P \leftarrow$ new empty Priority Queue
for each character c **in** alphabet of X **do**
  $T \leftarrow$ single node binary tree storing c
  $P.\text{insert}(f(c), T)$
  while $P.\text{size}() > 1$ **do**
    $(f_1, T_1) = P.\text{removeMin}()$
    $(f_2, T_2) = P.\text{removeMin}()$
    $T \leftarrow$ new binary tree T with left subtree T_1 **and** right subtree T_2
    $P.\text{insert}(f_1 + f_2, T)$
    $(f, T) = P.\text{removeMin}()$
  return T

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 $e \in G.\text{outgoingEdges}(v)$ **do**
  if e is **not** explored **then**
    $w \leftarrow G.\text{opposite}(v, e)$
    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

DFS Iterative (SCUFFED)

```
1. Push START vertex to empty stack; init visited tracker + RESULT list.
2. While stack not empty: pop A.
3. If A unvisited, mark visited + add to RESULT.
4. For each outgoing edge from A:
5. If edge unvisited & adjacent unvisited, mark edge visited + push adjacent (reverse order).
6. If edge unvisited & adjacent visited, it's a back edge (cycle possible).
```

BFS Iterative (SCUFFED)

```
1. Queue START vertex to empty queue; init visited tracker + RESULT list.
2. While queue not empty: dequeue A.
3. If A unvisited, mark visited + add to RESULT.
4. For each outgoing edge from A:
5. If edge unvisited & adjacent unvisited, mark edge visited + queue adjacent (reverse order).
6. If edge unvisited & adjacent visited, it's a cross edge (cycle possible).
```

Algorithm BFS-Recursive (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
$Q.\text{enqueue}(u)$
Mark vertex u as visited
while $Q.\text{isEmpty}()$ **do**
  $v \leftarrow Q.\text{dequeue}()$
  for all e **in** $G.\text{incidentEdges}(v)$ **do**
    if e is **not** explored **then**
      $w \leftarrow G.\text{opposite}(v, e)$
      if w has **not** been visited **then**
        Record edge e as discovery edge **for** vertex w
        $Q.\text{enqueue}(w)$
        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** $G.\text{outgoingEdges}(v)$ **do**
  $w \leftarrow G.\text{opposite}(v, e)$
  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** $G.\text{vertices}()$ **do**
  if $v = s$ **then**
    $\text{setDistance}(v, 0)$
  else
    $\text{setDistance}(v, \infty)$
    $P.\text{insert}(\text{getDistance}(v), v)$
    while $P.\text{isEmpty}()$ **do**
      $u \leftarrow P.\text{removeMin}()$
      for all e **in** $G.\text{incidentEdges}(u)$ **do**
        $z \leftarrow G.\text{opposite}(u, e)$
        $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$
        if $r < \text{getDistance}(z)$ **then**
          $\text{setDistance}(z, r)$
          $P.\text{replaceKey}(\text{getLocator}(z), r)$

Dijkstras (SCUFFED)

```
1. Init min-heap PQ + enqueue START vertex (dist=0).
2. Init locator map (vertex → PQ position).
3. Init visited tracker for processed nodes.
4. Init RESULT map for shortest path distances.
5. While PQ not empty: dequeue A.
6. If A unvisited mark visited + record in RESULT.
7. For each edge (A→B,w): if B unvisited & dist[A]+w<dist[B], relax edge (update dist[B], update/enqueue B via locator); if B visited → ignore.
8. When PQ empty RESULT holds shortest path dists from START.
```