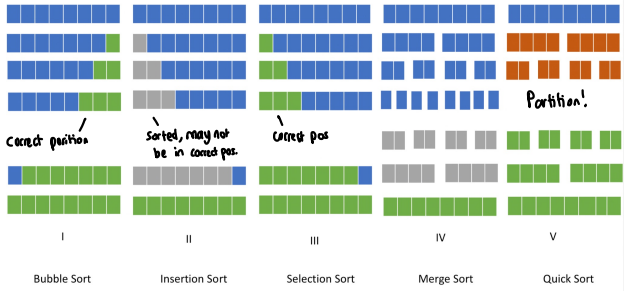
Trees

Binary Search — O(logn)

- Binary Search — (end - begin) ≤ n/2^k in iteration k
- A[begin] ≤ key ≤ A[end]
- Peak Finding — If reverse on right half, peak is in right half
  - Peak is in [begin, end]
  - every peak in [begin, end] is in [0, n-1].

Sorting



Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Quick Sort
O(n) (sorted) O(n²) (average)	O(n) (sorted) O(n²) (average)	O(n²)	O(nlogn)	O(n) (partition alr) O(nlogn) (avg) O(n²) (worst)
Stable!	Stable!	Not Stable!	Stable!	Not Stable!

- Bubble Sort: At the end of iteration j, the biggest j items are correctly sorted in the final j position of the array.
- Insertion Sort: At the end of iteration j: the first j items in the array are in sorted order.
- Selection Sort: At the end of iteration j: the smallest j items are correctly sorted in the first j position of the array.

Quick Sort:
- A[high] > pivot at the end of partition
- At the end of every loop iteration:
  - for all i > high, A[i] > pivot.
  - for all l < j < low, A[j] < pivot.
  - ∴ Array A is partitioned around the pivot.

Quick Select — O(n)

Only do min number of sorting
1. Partition the array.
2. Continue searching in the correct half.

Only recurse once! ⇒ not sorting

Runtime Analysis:

1. T(n) = 2T(n/2) + O(n) log n = O(n log²n).
2. T(n) = T(n/2) + O(1) = O(log n) — binary search.
3. T(n) = T(n/2) + O(n) = O(n) — quickselect
4. T(n) = 2T(n/2) + O(n) = O(n log n) — sorts.
5. T(n) = T(√n) + O(log n) = O(log n)
6. T(n) = T(n-2) + O(n²) = O(n³)

$\log_a b = \frac{\log_c b}{\log_c a}$ — constant.

Note: $O(\sqrt{n}) > O(\log^k n)$ $k \in \mathbb{Z}^+$

Binary Trees

All in left sub-tree < key < all in right sub-tree.

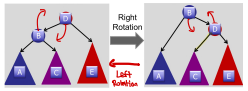
Balanced Trees — AVL Tree

*. A BST is balanced if h = O(log n)

A node is height-balanced if: |v.left.height − v.right.height| ≤ 1

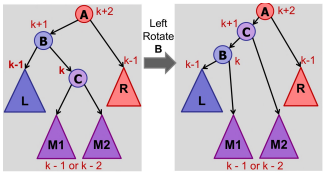
- A height-balanced tree with n nodes has at most height h < 2log(n).
- A height-balanced tree with height h has at least $n \geq 2^{h/2}$ nodes.

Rotation — During insertion (2 at most)
During deletion (O(log n))

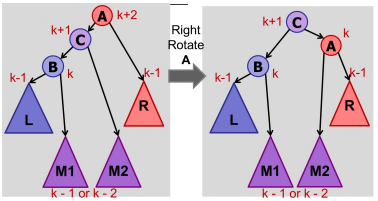


A < B < C < D < E
Rotations maintain ordering of keys.
⇒ Maintains BST property.

If v is unbalanced and left-heavy (rotate right)
If v.left is right-heavy (left rotate v.left then rotate v right)

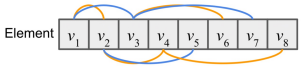


Left-rotate B
After left-rotate B: **A** and **C** still out of balance.



Heap

- Usually implemented using array. Exploit array's index to find left child/right child/parent of a node.



index
children: (2i+1) & (2i+2)
parent: ⌊(i-1)/2⌋

insert	Insert to the end of the array and bubble up	O(logn)
extractMax	Get the max value, swap the first element to the last element of the array and bubble down	O(logn)
updateKey	Change a key's priority and bubble up or bubble down accordingly	O(logn)
deleteKey	Change a key's priority to infinity and extractMax	O(logn)

(a, b) trees

- $2 \leq a \leq \frac{b+1}{2}$
  - a: min no. of children in an internal node of a tree.
  - b: max no. of children in an internal node of a tree.

Binary trees	(a, b)-trees
Each node *has at most* 2 children	Each node *can have more than* 2 children
Each node *stores exactly one* key	Each node *can store multiple* keys

root: 1 ≤ keys ≤ b-1
2 ≤ children ≤ b
— no. of children always one more than keys.

internal: a-1 ≤ keys ≤ b-1
a ≤ children ≤ b

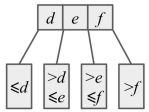
leaf: a-1 ≤ keys ≤ b-1
no children!

- A non-leaf node must have one more child than its number of keys.
- All leaf nodes must all be at the same depth (from root).

B-Trees / (B, 2B) Trees

Node:	Root	Internal	Leaf
#Keys	[1, a−1]	[a−1, 2a−1]	[a−1, 2a−1]
#Children	[2, a]	[a, 2a]	N.A

Number of keys in a parent node will determine number of children that node has.



search (key)	At each node, search its keylist to determine which child node to go next. Cost for search a key list is O(logb) and we need to search to at max O(logn) times => total cost is O(logn*logb) = O(logn)	O(logn)
insert (key)	Search for a leaf to insert a key into : O (logn), insert into the node, then resolve any violations occur with split (node) operation upward.	O(log n)
split (node)	Find a median key and split the node into 2	O(1)
delete (key)	Find a key's successor or predecessor in the leaf nodes, swap the two and delete the key in the node. Resolve any violation occur with split and merge operation upward	O(log n)
merge (node1, node2)	Combine 2 nodes into 1 node, re-connect edges	O(1)

Augmented Trees

- mostly from AVL Trees
- eg. · Use rank (index of the element in the sorted array)
  - ↳ do not store rank in every node ⇒ need update all nodes per insert.
  - ↳ Store size (weight) of sub-tree in every node
    - w(leaf) = 1
    - w(v) = w(v.left) + w(v.right) + 1.
    - "rank in sub-tree" = left.weight + 1.

nodes as: · intervals, sort by lower bound.
· guides, store max of left
- leaves as obtainable values.

Hashing — Symbol Table / Direct Access Table — No comparisons

- Hash function — O(1) time to compute + time to access bucket ≈ 1 (unless Strings)
  - SUHA: Each key is equally likely to map to each bucket $p = \frac{1}{m}$
  - UHA: Given a random key k, the *probe sequence* $h(k, 1)$, $h(k, 2)$, …, $h(k, M)$ has equal probability (i.e. $1/M!$) of being any of the $M!$ bucket permutations. (only applicable to double hashing.)
- Collision Resolution — Chaining — insert O(cost(h)+1) — n elements takes O(n log n)
  - Search O(cost(h)+ $\frac{n}{m}$) — expected no. of items per bucket
  - Space: O(m+n) — no. of buckets / size of list. (OK, load)
- Open Addressing (more sensitive to load)
  - eg. Linear Probing — Clusters of O(log n) if $n > \frac{1}{4}m$
  - For n items, in a table of size m, assuming uniform hashing, the expected cost of an operation is $\leq \frac{1}{1-\alpha}$ (during collision) — $\alpha \approx 0.9 \Rightarrow$ no longer O(1)
  - Double Hashing

$$h(k, i) = f(k) + i\, g(k, i) \bmod M$$

Starting bucket / Step number / Step size / Table wraparound

if $g(k)$ is relatively prime to m, then $h(k,i)$ hits all buckets.
(2 no. have no common factor except 1.)

Table Resizing

Grow and Shrink table as necessary
- Choose new size m.
- Choose new hash function (depends on table size)
- For every item in old table, compute & copy to new table.

Amortized Analysis

#Insertion	Budget	Cost	ΔSavings	Total Savings
1	$3	$1	$2	$2
2	$3	$1	$2	$4
3	$3	$1	$2	$6
…	…	…	…	…
$M-1$	$3	$1	$2	$2(M-1)$
M	$3	$1+M$	$3-(1+M)$	$3M$
$M+1$	$3	$1	$2	$3M+2$
…	…	…	…	…
$2M-1$	$3	$1	$2	$3M-2$
$2M$	$3	$1+2M$	$3-(1+2M)$	$3M$

cost of ins. / cost of copying.

∴ O(1) amortized cost

- $1 to cover the cost of inserting current item
- $1 to cover its cost for the next transfer
- $1 to "pay it forward" for an older item for the next transfer

Union Find

- Path Compression
  - After finding the root, set the parent of each traversed node to the root.
  - Can do path compression during find/union/whenever walking up the tree.
- Weighted Union
  - put shorter tree under bigger tree. (height of tree smaller, more balanced)
  - height increase when total size doubles
    - A tree of height k has at least 2^k items
    - height of tree size n is at most $\log n$.
    - all operations are O(log n)

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\log^*(m,n))$ time. (Connected for m operations)
- $\log^* n \leq 5$ in this universe
- not possible for linear time.

Weighted-union is faster:
- Trees are flat: O(log n)
- Union *and* find are O(log n)

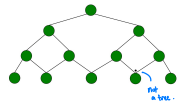
Weighted Union + Path Compression is very fast:
- Trees very flat.
- On average, almost linear performance per operation.

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	α(m, n)	α(m, n)

Dynamic Programming — Optimal Substructure

- Optimal solution can be constructed from optimal solutions to a smaller sub problem.
- eg. Sorting, reversing a string, merging 2 arrays, shortest path, Minimum Spanning Tree. (Mergesort / Combine reversals / A→B→C, A→C / 2 smaller MST)

Optimal substructure (overlapping sub-problems):
The same smaller problem is used to solve multiple different bigger problems.



Strategies

Bottom up
1. Solve smallest problem (leaves)
2. Combine smaller problems
3. Solve root problem.

DAG + Toposort
1. Topologically sort DAG.
2. Solve problems in reversed order

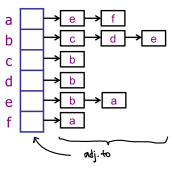
Top-down
1. Start at root and recurse
2. Reuse
3. Solve and memoize, only compute each solution once

Graphs — Representation — Adj. List

Adj Matrix

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

property: A^2 = length of 2 hop path.



Comparison:

Adj. List (O(E) ≈ V)
- Memory usage:
  - Array of size |V|
  - Linked list of size |E|
  - Total: O(V+E)
- For a cycle: O(V)
- For a clique: O(V²)
- If don't list edges explicitly:
  - Fast for:
    - Find any neighbours
    - Enumerate all neighbours
  - Slow for:
    - Checking if 2 nodes are neighbours.

Adj. Matrix (For dense graphs O(E) ≈ V²)
- Memory usage:
  - Array of size |V|*|V|
  - Total: O(V²)
- For a cycle: O(V²)
- For a clique: O(V²)
- Fast for:
  - Checking if 2 nodes are neighbours.
- Slow for:
  - Find any neighbours of v (enumerate given row in the list.)
  - Enumerate all neighbours

Graph Modelling

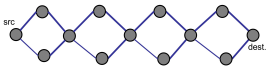
- Super Node w/ O edges
- Duplicates graph
  - Usually with hops limited
  - Or change of directions
  - also
- State Capture
  - nodes and edges must not have additional info
  - lack info about neighbours
  - edges are determined.
  - Cannot depend on prev node

Searching — undirected, unweighted graph

DFS
- Follow a path; if stuck, backtrack
- O(V+E)
- Uses a Stack

BFS
- Explore Frontier by Frontier
- Finds Shortest Path
- Skips already Visited nodes
- O(V+E)
- Uses a Queue

They will visit every node and every edge in the graph
But not every path in the graph *
- once a node is visited, it is never explored again.
- some graph have an exponential number of paths. ⇒ Too expensive!
  ⇒ It takes exponential time to explore all paths.



Example: $2^k > 2^{nk}$ different s->d paths.

** If you modify BFS/DFS to backtrack / re-visit existing nodes, you *probably* have an exponential time algorithm (or an algorithm that never terminates because of loops).

DAG

TopoSort
- Sort it in such a way that respects all the dependencies.
- All the edges have to point forward in the list.

1. POST-order DFS
   - process each node when it is last visited.
   - aka put it at the last place first (can be reversed)
   - unconnected parts needs restart DFS
   - Time complexity: O(V+E)
   - Only process a node after all incoming edges have been relaxed.
   - Fix 1-D shortest path, path is relaxed in order, dist is correct after relaxation.
2. Kahn's algorithm
   - does not depend on DFS. (Similar to BFS)
   - build things forward instead of backwards.

Input: directed graph G (only outgoing edge)
Repeat:
- S ← all nodes in G that have no incoming edges.
- Add nodes in S to the topo-order
- Remove all edges adjacent to nodes in S
- Remove nodes in S from the graph

Time:
- O(V + E) time complexity

Longest Path: negate weights
- Does not work if graph is cyclic
  ⇒ positive weight cycle is the original graph
  + negative weight cycle in new graph.

Paths — SSSP — Triangle Inequality

$\delta(S,C) \leq \delta(S,A) + \delta(A,C)$

might have shorter path than $\delta(S,A)+\delta(A,C)$ but not longer path

Shortest Path to every node from a specific node.
Shortest Path Tree

Dijkstra's
- only relax each edge once
- All edge weights ≥ 0. — If neg. ⇒ needs to recalculate same node / reweighing the node changes the shortest path.
- Priority Queue — decreaseKey() to update est.
- O(V+E)(log V) & O(E log V)
  - each node added to PQ once
  - each edge is relaxed once.
  - PQ operations.

Bellman-Ford (BF) — O(VE)

Key property:
If P is the shortest path from S to D, and if P goes through X, then P is also the shortest path from S to X (and from X to D). ⇒ must also be the shortest path to every intermediate node.

```
n = V.length;
for (i=0; i<n-1; i++)
  for (Edge e : graph)
    relax(e)
```

∴ After k iterations, k hop estimate on the shortest path is correct.
⇒ max no. of iterations = V-1.
If all we know is whenever the shortest path is, we have correctly updated the estimate in it.
∵ Every node in T has come shortest path after one iteration ⇒ no!

Detect negative weight cycles: run |V| iter see if no est. changes.

Invariant: Let T be a shortest paths tree of graph G rooted at source s.
After iteration i, if node u is i hops from s on tree T, then $est[u]$ = distance(s,u)

Single-Source Shortest Paths
- BFS: Unweighted graphs
- Bellman-Ford: simple, general
- Dijkstra: faster, only non-negative weights
- DAGs: DAG_SSSP = topological sort + relax
- Longest Path on DAG = shortest path on DAG.

MST

Weighted, undirected.
* Cannot find shortest path

Properties:
1. No cycles
2. If you cut an MST, the 2 pieces are both MSTs. (Converse does not hold!)
3. Cycle property: For every cycle, the minimum weight edge is not in the MST
   False cycle property: For every cycle, the minimum weight edge may or may not be in the MST.
4. Cut Property: For every partition of the nodes, the minimum weight edge across the cut is in the MST.

Algorithms:

Red rule:
If C is a cycle with no red arcs, then color the max-weight edge in C red.

Blue rule:
If D is a cut with no blue arcs, then color the min-weight edge in D blue. (part of MST)

Prim's
Goal: identify cuts
- S: set of nodes connected by blue edges
- Initially: S = {A}.
- Repeat!
  - Identify cut: {S, V-S} (Separate from special cut and others)
  - Find minimum weight edge on cut.
  - Add new node to S.
- Priority Queue. O(E log V)

Kruskal's
Basic idea:
- Sort edges by weight from smallest to biggest.
- Consider edges in ascending order:
  - one edge:
    - If both endpoints are in the same blue tree, color the edge red. (most be the heaviest edge on the cycle.)
    - Otherwise, color the edge blue.
- UnionFind: Connect 2 nodes if they are in the same Blue tree
- O(E log V) due to sorting edges ≈ O(E log V) since $E \approx V^2$

MST variants:
- unweighted: BFS/DFS
- directed w one root: add min weight incoming edge O(E)
- Maximum Spanning Tree:
  - Add constant k to weight of every edge (relative weight edge is the same)
  - negative edges have no impact.
    ∴ multiply by −1.
  - or run Kruskal's in reverse (try heaviest edge prioritized)