- Passing by value / reference/pointer   C++

(different return type cannot)

- Function overloading (Must have different #parameters/type of parameter)

- Delete Orphans or mem leak

- Class = New data type (blueprint) - Instance = Object of class
- Constructors = Called to create object automatically
- Friend = Allow other class to access private
- Destructors ~ (can be called @ end brackets)
- Big O can calculate total — if/else choose most expensive.
- Inheritance = child = subclass / Parent = Superclass
  All the functionality of parent ↳ Override parent ← [child ≠ private / child & parent = protected]
- Polymorphism = add virtual so the child functn can be called
- Template = class < Type T> [template <class Type T> (generic)
- Overloading operators (from class then compare the indiv variables)

- Searching (Linear = O(n)) Searching
  ↳ (Binary Search = O(log n))
  left small | k | right big
- Global Max = O(n) unsorted
  O(1) sorted
- Local Max = O(log n)
  ↳ just go to larger half
- 2D peak = O(n log n)
- Θ = Worst = Best
- O = Worst/Upper
- Ω = Best/Lower

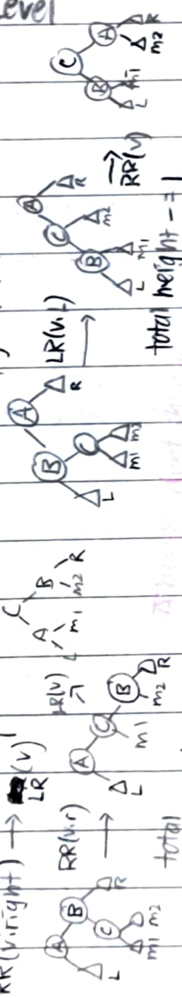
Heap: Complete binary tree ↳ push to left.
bubbleUp: child check parent.
bubbleDown: check child and swap w larger child.
delete: swap last node w deleting, then last()'s deleted or bubble ↑ or ↓
extract max: delete root
printing array = level-order traversal.
Each level starts at $2^i - 1$
left child = 2x+1
right child = 2x+2
parent = floor((x-1)/2)
Heapsort = extractMax, then now have empty space just leave it there. (inplace sorting) O(n log n)
Heapify = unsorted list → Heap (recursn to solve smaller heaps) O(n)

Heap vs AVL tree
- same operatn cost
- simpler
- better concurrency
- #heap instructn

FILO: First In Last Out — push/pop — Stack (ADT)
FIFO: First In First Out — Q (ADT)
enqueue - add tail, dequeue - rm head; array? append/prepend; List
Dictionary → key, value ↳ succ/prev
Prio Q → scheduling ↳ extraction

Traversal: Pre / In / Post / Level
(n) (n) (n) (log n)

Tree: Operations same only depend on height
Best case: balanced O(log n)
Worst: Linked list O(n)
Successor: 3 cases 1) Key in tree 2) result > key 3) result < key — return parent which went left.
Delete: 3 cases 1) Root 2) 2 child 3) 1 child — link parent w child ↳ find successor
Left rotate: Og root goes left
Right heavy = LR(v); r.r heavy = LR(v); R heavy RR (v.r left heavy) → LR(v)
Right rotate: Og root goes right
- Left heavy = RR(v); Left heavy RR (v.left right heavy) → LR(v.left) → RR(v)



total height = -1
- Insertn = max 2 rotate to balance.
- Deletn: O(log n) all
AVL: O(log n) rotate
RB: O(1) rotate, fast insert/del, slow search
Amortised Performance (mark deleted & clean up after)

Augment: Order statistic = [illegible]
- Dynamic: Keep weight of each node [illegible]
if l.w > ith, find in L child [illegible]
Augment: Range search — Range List Query [illegible]
Range count = rank(b) - rank(a) + 1
2d: Preprocess main tree by x-coordinate
then using x-coor, created mini tree w root as y-coor
Query: $O(\log^2 n + k)$ Space = n log n
Insert/del/rot = O(n)

Sorting

	Bubble	Select	Insert	Merge	Quick
Best	n	n²	n	n log n	n log n
Avg	n²	n²	n²	n log n	n log n
Worst	n²	n²	n²	n log n	n²
Mem	1	1	1	N	1
Stable	Y	—	Y	Y	N

Selection = Swap with [illegible]
Insertion = Insert next smallest to correct [illegible]
Merge = O(n) to merge but O(log n) level
Quick = Instead of splitting ½ split by partition ↳ fast for small case
↳ randomized analysis n log n
↳ paranoid Q-sort solve
1/10 find partition
Expected outcome / depth

Union-Find = Dynamic Connectivity — Utilise Transitivity (p-q, q-r ∴ p-r) — Connected components (max set of connected)

— Array data structure (same parent then connected)

Union(m,n) = Combine m,n Find(m,n) = return if same parent

Quick Find: "Path compression" so all will have same parent during union()

~~Quick Union: Bad.~~ weighted union: Union will connect via weight
↳ smaller join bigger so $h = \log n$.

	Find()	Union()
Q-F	O(1)	O(n)
Q-U	O(n)	O(n)
W-U	O(logn)	O(logn)
W-U Wpc	α(m,n)	α(m,n)

↳ $O(n + m\alpha(m,n))$ inverse ackermann

Path compressⁿ, after find, just connect
↳ findroot() set parent of traversed node to the root.

Star graph: Degree = n-1 Diameter = 2
Clique: Complete graph Degree = n-1 Diameter = 1
Line = Degree = 2 Diameter = n-1
Cycle = Degree = 2 Diameter = $\frac{n}{2}$ or $\frac{n}{2} - 1$

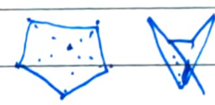
Proving lower bound → show an example that at least n.
Provg upper → Show all combinatⁿ

Hash Table
- No successor/predec
- Hash map = <key, value>
- Direct access Table = array index
  ↳ uses up too much space
- Hash functⁿ h()
  ↳ store key in bucket h(k)
  ↳ Time to compute h + access bucket
  ↳ Collisⁿ $h(k_1) = h(k_2)$
  ↳ chaining (LL connected to bucket) [illegible]

Open Addressing = Probe until empty/null/deleted
- Linear = h(k) + i mod m
- Quad: h(k) + i² mod m
- Double hash = h(k,i) = h(k) + i·g(k)
- Search = Until null, worst case = O(n)

Performance = 1/(1-load)
✓ Clustering is problem since many collisⁿ (sensitive to load)
✓ Save space ✓ performance
Resize: (n = 2m) → n = 2m O(n)
↓ n < m/4 → m = m/2
- Track deleted items.
- amortised cost T(n)

Planar: embed graph on plane.
max degree = v-1 (star) avg deg = <6



project other triangle out to find

—Drawg = Push A, add artificial if neighbour to A not neighbour to each other, repeat until left 3, draw triangle, then draw back.

ConvexHull = all combi of convex
any 2 points in P will have a line connecting in P.
- Jarvis March O(hn) foreach vertex find angle
- Graham Scan O(nlogn) connect all dots, then if concave, remove vertex & connect neighbour & fall back again to check.
- QuickHull, construct quad w/ 4 points furthest away. For each face, find furthest point & add in.

Graph
- Connected components = vertex connected by edges (degree of a node = # adjacent edges)
- Diameter = max distance using shortest path
- Adjacency list = Node(array) Edges (LL per node) — O(V+E) / cycle E = O(V) — slow query if v,w neighbour
- Adjacency matrix — O(V²) fast query if neighbour

BFS — Find shortest path: explore level by level (skip already visited nodes) [using Q]
[Cannot visit if component are separated] O(V+E)

DFS — Follow path until stuck = backtrack to find new edge O(V+E) [using stack]

Bellman Ford — for all v { for all e { relax e }} It is the slow way O(VE)
- Terminate early when all relax has no effect
- Repeat V times, relax every edge
- Detect negative weight cycle by |V|+1 iterate
- BFS if all weight same — can detect/use w̄ -ve weights

Dijkstra — Each node added to PQ once — each E relaxed once — PQ operates log(V)
↳ O(E log V) — If dequeued early, then can stop since deque means curr min
Cannot have mix of -ve weights & positive weights

Topological Sort — Post-Order DFS O(V+E) process node when last visited
— Kahn's (BFS) O(V+E) — After relax edge, add in-deg = 0 to Q

PQ	ins	del	deckey	Total
array	1	V	1	O(V²)
AVL	logV	logV	logV	O(E logV)
d-way heap	d logV?	d logV	logV	$O(E \log_{E/V} V)$
Fibo Heap	1	logV	1	O(E + VlogV)

BFS = Take edge discovered least recently
DFS = " " " most "
Dij = " " closest to source.

SSSP = Topo Sort + relax in the order O(E)
LSP: Negate edge → Topo sort + relax
SSSP in tree: E = V-1 use DFS/BFS.

MST
MST Property 1: No cycle 2: Cut MST → 2 MSTs
3: for every cycle, max weight is not in MST
4: Cut property — for every partition of vertex, min weight edge across cut is in MST
4b: for every v, min outgoing edge is part of MST

Prim: find next min weight of tree by relaxing O(ElogV)

Kruskal: keep adding min overall unless form cycle
sorting = O(ElogV) Union find = α() or O(logV)

Boruvka: Add min edge btwn components
- Start off all as individual vertex
Each step: O(E+V) Total: O(E log V) since each step half total

Directed MST: add all min incoming edge to all nodes (if only 1 root)