CS2040C Cheat Sheet
Github/ehz0ah
NUS CEG Lee Hao Zhe

Class: Blueprint that defines properties and behaviours and to create objects, while instance is the instantiation of a class.
Public: Attributes/functions can be accessed outside of the class.
Private: No visibility from the outside. Can only be accessed/modified by getters/setters. Other classes can access if they are friend class. By default, members are all private.
Encapsulation: No direct access to data, only through exposed functions, data + functions abstractions. E.g. Classes.
Inheritance: A subclass/derived class inherits attributes/methods from parent class. Subclass override parent class but cannot access the private attributes/methods of its parent class unless its protected.
Polymorphism: Allows objects of different classes to be treated as objects of a common parent class. E.g. Stack pointer (x) that points to a subclass. Without virtual, any methods called by x will call parent class method instead of subclasses.
Constructor is called when an instance of the class is created.
Destructor will be called when an instance of the class is deleted.
Variable-size array (LL): Size changes even during running time, data is not stored in ONE connected trunk of memory.
ADT: Create a class with a set of its own functions using an underlying data structure. Only care the interface but not the implementation.
Kadane: Use 2 variables A (MIN) and B (0). Iterate through the array and update B to the sum of the current element and B. Update A to be the maximum of A and B. A will contain the maximum sum of a contiguous subarray. *O(n).*
Peak-Finding: Recurse to the larger side for 1D array. For 2D arrays, find the middle column and the column maximum for it and it's left & right neighbours. Recurse to the larger side. *O(nlogm).*
Time Complexity:
Worse-case (O): $\exists c, n_0 > 0$: *for all* $n > n_0$, $T(n) \leq c\, f(n)$
Average (Θ): $T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n))$ *and* $T(n) = \Omega(f(n))$
Best-case (Ω): $\exists c, n_0 > 0$: *for all* $n > n_0$, $T(n) \geq c\, f(n)$
Sorting
For small arrays (n < 1000), use Insertion Sort. For large, use Quick sort then insertion sort (Hybrid Quick Sort). This reduces recursion overhead.

Bubble Sort Largest element winds up at the back	Worst n^2, best n, Average n^2. Swaps elements with each other. O(1) memory, stable. Invariant: Largest element is the last element. Number of comparisons = (n – 1) + (n – 2) + … + 1
Selection Sort Selects the smallest element and swap it with first element then increment	Worst/Best/Average n^2. O(1) memory, NOT stable. Invariant: Half the array would be perfectly sorted; other half would be jumbled. Number of comparisons = (n – 1) + (n – 2) + … + 1
Insertion Sort Place the current element at the correct place in each iteration.	Worst n^2, best n, Average n^2. O(1) memory, stable. Invariant: Half of array sorted; the other half totally unchanged.
Merge Sort (D & C) Split array until there are 2 elements, then sort the split elements and merge back in order. Merging takes O(n)	Worst/Best/Average nlog n. O(n) memory, stable. Invariant: Array has parts that are sorted but not merged yet. For example, two halves of array sorted but not merged yet.
Quick Sort ⇒ Probabilistic Uses pivot to partition the array into 2 sub-arrays - Recursively sort arrays such that left ≤ x ≤ right, repeating the partitioning process. Combine the arrays together in sorted fashion. Pivot is good if it divides the array into 2 pieces, each of size at least n / 10. Median/fraction/random.	Worst n^2, best nlog n. Worst when pivot = start/middle/end. Space O(1), not stable by default. If pack duplicates are used, then it can be stable. 2 pivots > 1 pivots in speed. Pivots are randomly picked in O(1) time. Invariant: Array is partitioned, with one element having all larger on right and smaller on left.

Stable sort is one where the relative order of equal elements remains unchanged after sorting.
Selection Sort becomes stable with *O(n)* extra space.
Trees Tree Operations: log(n) –1 ≤ n
Stirling's Approx: log (n!) > n log (n/e) = Ω (n log n) (Height of tree with n! leaves, also the no. of steps to any leaf)
In-order: Left, print, right. (Touch twice) → O(n)
Pre-order: Print, left, right. (Touch once) → O(n)
Post-Order: Left, Right, Print (Touch thrice / Leaving) → O(n)
Successors never have 2 children because successors are minimum of the right subtree.
Delete → Remove node from tree.
- No children → Remove node
- 1 child → Remove and connect child to parent.
- 2 child → Replace node with successor, then remove old node.
AVL Tree
Invariant: For any node v, abs(v.left.height– v.right.height) ≤ 1
Height-balanced tree contains at least $n > 2^{h/2}$ node → O (log n).
Insertion → Needs 0 to 2 rotations to balance.
Deletion → Needs up to O (log n) rotations to balance.
Just mark the deleted note as "deleted". Performance degrades over time, Clean up later (Amortized time)
Rebalancing (Only rotations and deletion reduce height)
Left heavy ⇒ right-rotate(v)
Right heavy ⇒ left-rotate(v)
Left, right-heavy ⇒ left-rotate(v.left), right-rotate(v)
Right, left-heavy ⇒ right-rotate(v.right), left-rotate(v)
Red-Black Tree
More loosely balanced • O(1) rotations for all operations. • Java TreeSet implementation • Faster (than AVL) for insert/delete • Slower (than AVL) for search.
Order Statistics
Weight(v) = w(v.left) + w(v.right) + 1, store it into each node.
Select(k): set rank = root.left.weight + 1. If k == rank then return that node, else if k < rank: return root.left.select(k), else if k > rank: return root.right.select(k–rank).
Rank(v): First find the node v and initialise its rank to v.left.weight + 1. Recurse upwards, if the current node is the right child of its parent, add parent.left.weight + 1 to rank(v).
Update weight and rank of nodes during operations.
Orthogonal Range Searching
Given two numbers a < b:
Range List Query: List out all the elements x such that a ≤ x ≤ b
Range Count: Count all the elements x such that a ≤ x ≤ b. rank (b) – rank(a) + 1, if a or b are not in BST, use their successor or predecessor respectively.
Query cost: $O(\log^d n + k)$ k is the number of output elements
buildTree cost: $O(n \log^{d-1} n)$
Space: $O(n \log^{d-1} n)$
Insertion, deletion, rotation maybe O(n).
Store d–1 dimensional range-tree in each node of a 1D range-tree. Construct the d–1-dimensional range-tree recursively.
Hashing
Simple Uniform Hashing Assumption: Every key equally likely to map to every bucket, and keys are mapped independently.
Uniform hashing assumption: every key is equally likely to be mapped to every permutation, independent of every other key. NOT fulfilled by linear probing.
Chaining: Each bucket contains a linked list of items. Space = O (m + n), m = table size, n = list size, worst case is O(n) if all items in same bucket.
Expected search time = 1 + n / m. If m > n, O (1). Else O(n).
When m == n, we can still add new items to the hash table and still search efficiently.
Division: h(k) = k mod m; Division is slow. m is prime
If k and m are divisible by common divisor d, then only 1/d space used.
Multiplication: h(k) = (Ak) mod 2^w >> (w – r) (Takes r bits of the front/left second half)
Table size m = 2r, word size of a key in bits = w, A = odd constant.
Faster than Division and works decently when A is odd and more than 2^{w-1}.
Open Addressing (Probing)
If there's collision, find new bucket by going traversing down the table via (h(F) + f(i)) mod m
Linear: f(i) = i, Quadratic: f(i) = i^2, Double Hash: f(i) = i x g(key)
Double hashing > Quadratic > Linear because of minimum collisions!
Performance of open addressing, expected cost of an op = 1 / (1 – a) where a (load) = n / m
Advantages: Saves space (less empty); Rarely allocate memory; Better cache performance as table is all in one place in memory compared to linked list.
Disadvantages: Open addressing fails when load > 80% and is more sensitive to choose of hash functions. Sensitive to clustering as well. When the table is full, we cannot insert any more items and cannot search efficiently.
Hash table size: Assuming chaining and simple uniform hashing,
Increment by 1: O(n) resize and $O(n^2)$ insert
Square: $O(n^2)$ resize and O(n) insert
Double: n == 0.8 * m1 then m2 = 2m1; n < m1 / 4, then m2 = m1 / 2. *O(n).* Average insert of an item is O(1).
Delete item by setting it to DELETED, stop searching only when hit null or the key.
Good Hashing Functions: (1) h(key, i) must be able to reach all slots • True for linear probing. (2) Simple Uniform Hashing Assumption.
Heaps (Not a BST)
Heap ordering: priority[parent] ≥ priority[child]
Complete binary tree: Every level (except last level) is full; all nodes as far left as possible.
Operations: all O(max height) = O(log n)
Heap as an array: left(x) = 2x+1, right(x)= 2x+2, parent(x) = $\lfloor \frac{x-1}{2} \rfloor$
HeapSort: ⇒ O(nlogn) max/min: bubble down larger/smaller child
In-place, Faster than Merge slower than Quick, Deterministic, Unstable, Ternary (n-way) HeapSort is faster, allow duplicates.
Unsorted list to heap (Heapify): O(n) (bubble down, n-1 to 0)
Heap to sorted list: O(nlogn) (extractMax, swap to back)
Heap with n nodes has at least $\frac{n}{2}$ nodes that are leaves.
Each level z starts from index $2^z - 1$.
Union Find Connectivity is transitive.
Quick-find - int[] componentId – flat trees
- O(1) find: Check if 2 objects have the same componentId
- O(n) union (p,q): Loop through array and update componentId of every node in q to the componentId of p.
Quick-Union - int[] parent – deeper/taller trees (unbalanced)
- O(n) find: Check if they have the same root (Need recurse to find)
- O(n) union: ^ recurse until you find both parents and set one to be the parent of the other.
Weighted-Union - int[] parent & - int[] size - Balanced
- O(log n) find: Check if they have the same root/parent
- O(log n) union: Same as Quick-Union but now assign the larger size tree as the parent and update size.
Path-Compression: First find both roots. As you traverse up the tree via recursively calling parent, update the parent of EVERY traversed node to be the root for their own disjoint sets then union the roots. Both Find and Union are O(log n).
Weighted-Union + Path-Compression: Any sequence of m union/find operations on n objects takes O(n + mα(m, n)). Inverse Ackermann is ≤ 5 in this universe. (Very flat trees, average linear time)
Graph
Degree of a node = Number of adjacent edges of a node.
Degree of a graph = MAXIMUM number of adjacent edges of a node.
Diameter = Max distance between 2 nodes, following the shortest path
Clique = Complete graph, all pairs connected by edges. Diameter = 1, Degree = n-1. Diameter of a cycle is $\frac{n}{2}$ or $\frac{n}{2} - 1$, Degree = 2.
Prove: Lower bound = Random generation, Upper = All combinations
• graph is **dense** if $|E| = \theta(V^2)$

adj	space	(cycle)	(clique)	use for
list	$O(V+E)$	$O(V)$	$O(V^2)$	sparse
matrix	$O(V^2)$	$O(V^2)$	$O(V^2)$	dense

Adjacency Matrix: Fast query = are v and w neighbours? Slow query = find me any neighbour of v. Slow query = List all neighbours. Adjacency List is exactly opposite for all 3 queries.
Breadth-first Search ⇒ O(V+E) - queue
O(V): every vertex is added exactly once to a frontier/current level.
O(E): every neighbour of the vertex pop from queue is visited once.
Parent edges (For that vertex, store who its parents is) form a tree & shortest path from S. Fails to visit every node when graph is disconnected.
Depth-first Search ⇒ O(V+E) - stack
O(V): DFSvisit is called exactly once per node
O(E): DFSvisit visits each neighbour
Adjacency matrix: O(V) per node total $O(V^2)$. Keep track of visited nodes for both searches to prevent double visiting.
Directed Graphs
- In-degree = Number of incoming edges
- Out-degree = Number of outgoing edges
- For BFS, follow outgoing edges, ignore incoming edge for directed graph. For DFS, recurse via outgoing edges and backtrack using incoming edges.
Single Source Shortest Paths
- Can use BFS if edges are all same weight. (MIN hops not distance)
Bellman Ford ⇒ O(VE)
- |V| iterations of relaxing every edge – terminate when an entire sequence of |E| operations have no effect. If the sum of weighted edges

is negative for the cycle, the overall paths will just keep on reducing. (Can use to detect negative weight cycles in graph)

Invariant: after 1 iteration, 1 hop away weight from source is correct, hence n relaxation.

Dijkstra ⇒ O((V + E) log V) = O(E log V)

- Edges cannot be negative (may work sometimes). Cannot reweight the edges. Choose to relax the edge connected to the node with the shortest estimate distance.
- Use a PQ to track the min-estimate node, relax all its neighbours that are in the PQ and update their distance in the PQ. Keep track of nodes that are dequeued already to avoid adding them back in PQ.

Invariant: estimate of a node distance ≥ actual shortest distance

- |V| deleteMin as each node is added to PQ once. |E| relaxation as each edge is relaxed once. Fibo Heap O(E + Vlog V).

d-way Heap O(E $\log_{E/V}$ V). AVL O(E logV). Array O(V^2).

For DAG: O(E) (Topological sort and relax in this order).

For trees: O(V) (relax each edge in BFS/DFS order).

Longest Paths

Negate the weights and the shortest path in negated = longest path in regular. However, make sure that the negated graph has no cycle.

If directed acyclic (no cycle) graph, can solve efficiently using topological sort.

Topological Order O(V + E) MUST be DAG

Properties: Sequential total ordering of all nodes. Edges in the order only point forward.

Pre-Order DFS: Process each node when it is first visited.

Post-Order DFS: Process each node when it is last visited. Or, when all the neighbours are visited, Or, when it is finished.

- For post DFS, if you reach back your original node and not every node is visited, pick a unvisited node and do the same.
- Topological orderings are not unique, but some are unique.

Connected Components

There must be a path between the 2 nodes.

Strongly connected component: 2-way path between 2 nodes.

Graph of strongly connected components is acyclic.

Planar Graphs: If there exists an embedding for a graph, it's planar. If a graph is planar, $V - E + F = 1 + C$. Has O(n) edges. Degree < 6.

Minimum Spanning Trees

- V – 1 number of edges in a MST.
- Not same as shortest paths, can DFS/BFS if same weight edge.
- **Acyclic** subset of the edges that connects all nodes with min weight
- Any 2 subtrees of the MSTs are also MSTs
- For every cycle, the max weight edge is NOT in the MST, but the min weight edge may or may not be inside the MST.
- For every partition/cut of the nodes, the minimum weight edge across the cut is in the MST.
- For every vertex, the minimum outgoing edge is in the MST, but max weight edge may or may not be inside. Inside for star graph, or 1 outgoing edge.
- The shortest edge in a graph is always in the minimal spanning tree of that graph if no two edges have the same weight.

Greedy Method ALL O(Elog V)

Prim: Add all nodes to PQ with ∞ distance except source with 0. ExtractMin will choose the node with the min incoming edge weight (blue) then update its neighbours that are still in the PQ if new distance is lesser. Repeat extractMin until PQ is empty. No need to visit nodes that are visited already as those are finalised.

- Each vertex: one insert/extractMin ⇒ O(VlogV) for |V| nodes
- Each edge: one decreaseKey ⇒ O(ElogV) for |E| edges
- O(E) if all edges have known weight. E.g. 1-10 then use PQ array of size 10 with each bucket containing a linked list.

Kruskal

Sort edges by weight, loop through the edges and add edges (blue) if unconnected (else red). Sorting ⇒ O(ElogE) = O(ElogV)

Each edge: find & union ⇒ O(logV) or O(α) using union-find DS

O(α(V)E) if all edges have known weight. E.g. 1-10, array of 10 linked list, still using UFDS.

Boruvka O(E $\log^2$ V / P), where P is # of processors

Parallelizable (Each CPU core can handle different components until they link up), faster in "good" graphs (e.g., planar graphs), flexible.

At the start, for every node: add minimum adjacent edge.

Repeat: for every connected component, add minimum outgoing edge.

To find the min edge, use DFS/BFS on each node in the component.

In each step, assuming there are k components, at least k/2 edges are added, k/2 components merged and at most k/2 components remaining.

Euclidean MST Naïve solution: Compute a complete graph of P with each edge equal to the Euclidean distance ⇒ O(n^2). Then run MST ⇒ O(n^2log n). Better solution: O(n log n) by Delaunay Triangulation.

- Cut/cycle property, generic MST algorithm do not work for directed MST.

Directed MST with one root ⇒ O(E). For every node, add minimum weight incoming edge.

Maximum spanning tree: Multiply the weight with -1 or add a large positive number and solve normally. Does not affect MST.

Binary Space Partitioning Tree

Subdivide the entire space by a binary tree. Each internal node is a division of a partition/space. Each leaf is a part of the space with only 1 polygon. Just choose a random polygon as a root to start partitioning. But without reference to a viewpoint, the left and right child of a BSP tree do not matter. With a viewpoint, polygons are rendered from back to front. Thus, BSP is constructed with the furthest polygon as the leftmost child and the nearest as the rightmost child. Rendering is done via In-order traversal of the BSP tree.

Advantage: Once the tree is computed, the tree can handle all viewpoints without reconstructing the tree (efficient). It can also handle transparency.

Disadvantages: Cannot handle moving/changing environments. Preprocessing time for tree construction is long.

Computational Geometry

Solving computational problems by geometric methods. E.g. Simplex algorithm in linear programming

Mathematical Thinking ⇒ Does a solution exist? Is there more than 1? How many? Can be solved without finding an actual solution.

Computational Thinking ⇒ Is it computable? How fast? Given a set of disks, find the area of union. Use Inclusion-Exclusion formula.

Convex Hull O(nlog n)

A convex hull is the set of all convex combinations. A point in a set P is convex if ∀x,y ∈ P, the edge xy ⊆ P.

Jarvis' March (Gift Wrapping)

Take the leftmost vertex and draw a downward arrow (0°). Repeat: Search for the next vertex on the convex hull by choosing the one with the minimal turning angle with reference to the arrow (Arrow only increment in degrees w.r.t original 0°). Complexity: O(hn), where h is the number of faces of the convex hull, O(n) for looking through all the points to find the min angle point.

- Can go clockwise or anti-clockwise.
- Best case: points are sorted already O(n).

Graham Scan

Take the left most vertex. Sort O(nlog n) the rest according to their angles. Connect ALL points in that order and form a polygon. Starting from the left most vertex, go around the polygon. If it is a concave vertex, "make it convex" by "filling" it through connecting its two neighbours. At most one vertex is "buried" thus at most 1 step back.

Divide and Conquer (D & C)

Sort O(nlog n) vertices in a direction, e.g. y direction. Repeat: Merge every neighbouring convex hull, similar to Merge Sort. For any 2 convex hulls to merge, take the highest point of the lower hull and the lowest point of the higher hull and form a line. Walk until the angle between the line and the boundaries of both hulls ≥ 180° (convex). Concavity and convexity consider internal angle of the polygon.

Incremental Method

Adding a point according to a sorted direction incrementally to a convex hull.

Similar to D & C but different. Consider the point and the existing hull as 2 hulls and merge them.

3D Convex Hull

D & C: Expected O(nlog n) with a complicated data structure.

Quickhull: First find the maximum and minimum in x and y directions and draw a polygon. Discard any points inside and for each side of the polygon, find the furthermost point and include it in the convex hull. Eliminate any point within. O(n^2) in 2D and 3D.

In general: O($n^{\text{ceil}(d/2)-1}$ + nlog n) due to the complexity of the number of "faces" on a convex hull. Every two dimension increases complexity by one.

Degeneracy

- More than two points that are collinear: Especially on the boundary. can be corrected by changing the "less than" to "less than or equal to" when walking.
- Same coordinates in sorting: "Slightly" rotate all the points by an infinitely small angle. If they have the same x values, compare their y values. Perturbation to improve algorithm's stability and robustness.

Combining Data Structures

Queue + Linked Lists: Enqueue ⇒ O(n), Dequeue ⇒ O(1) printInOrder ⇒ O(n), Peek ⇒ O(1).

Queue + BST: Enqueue ⇒ O(log n), Dequeue ⇒ O(1), printInOrder ⇒ O(n), Peek ⇒ O(1)

Skip List: Multiple linked lists with same list with 'skips' between nodes. Skips of length m reduces search operation to at most $\frac{n}{m}$+ m.

Initialise k = 0, while (!done) ⇒ Insert element into level k list. Flip a fair coin: 50% done = true, 50% k += 1. Each search/insert takes O(log n) on average. Probabilistic searching. (Expected time)

Additional

T(n) = 2 T(n/2) + O(n) ⇒ O(nlog n)
T(n) = T(n/2) + O(n) ⇒ O(n)
T(n) = 2 T(n/2) + O(1) ⇒ O(n)
T(n) = T(n/2) + O(1) ⇒ O(log n)
T(n) = 2 T(n−1) + O(1) ⇒ O(2^n)
T(n) = T(n−1) + O(1) ⇒ O(n)
T(n) = 2 T(n/2) + O(nlog n) ⇒ O(n(log n)^2)
T(n) = 2 T(n/4) + O(1) ⇒ O($\sqrt{n}$)
T(n) = T(n−c) + O(n) ⇒ O(n^2)
O($\sqrt{n}$log n) = O(n)
O(log(n^2)) = O(2log n)

Concatenation of strings itself is O(n).

Harmonic series: j=i; j < n; j+=i ⇒ O(log n) but outer loop i is O(n).

Orders of growth: 1 < log n < $\sqrt{n}$ < n < nlog n < n^2 < 2^n < 2^{2n} < $\log_a n$ < n^a < a^n < n! < n^n

Diameter of a graph: SSSP all ⇒ O(V^2 logV)

APSP: Dijkstra all ⇒ O(V E logV) or O(V^2 E)

APSP: Floyd Warshall ⇒ O(V^3)

Update/relax weight: If current node dist + weight of edge < dist in PQ

Tips and Tricks

Sorting: Only merge and heap use O(n) space, rest O(1).

Graph: Consider adding a super node that connects 2 graphs when computation cheapest paths or shortest path is required. The weight of the edge can be 0 if the points in the 2 graphs are the same.

Hashing: Consider collisions (Not uniform distribution) and usage of the whole table when considering if a hashing method is good. Hash keys are immutable and cannot duplicate but values can be mutated. When given a hash function, see if its multiplication or division. Consider if it meets the requirements for it. NEVER square!

Sorting: If an array of size > c has only distinct c duplicate items, you can sort it using counting sort in O(n) time. To sort m sorted arrays that adds up to n elements into 1 sorted array takes O(nlog m).

Traversal: In-order is always sorted order for BST not Binary Tree. Pre-order is not the reverse of Post-order. Given a connected graph with a source and a destination, BFS may traverse fewer vertices from the source to the destination than DFS. DFS from the root is pre-order DFS.

Master theorem

$$T(n) = aT(\tfrac{n}{b}) + f(n) \quad a \geq 0, b > 1$$

$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

Recurrence

For any recurrence of the form, T(n) ≤ T(an) + T(bn) + cn, if a + b < 1, the recurrence will solve to O(n), and if a + b > 1, the recurrence is usually equal to Ω(nlog n).

Linear Sort

Counting Sort: O(n + k)

Radix Sort: O($\frac{b}{r}(n + 2^r)$)

Amortized Analysis

Given a sequence of c operations, total amortized cost is given by f(n) ∗ c ≥ c ∗ actual cost but this does not necessary mean that the amortized cost, f(n) ≥C_a.

Aggregate Analysis

Total time / total # of operations. Lacks precision and may not work for some cases. Useful for single operation.

Accounting Method

Impose extra charges on **inexpensive** operations and use it to pay for expensive operations. Any amount that is not used is stored in the bank for use by subsequent operations. The bank balance must not go negative.

Potential Method

Denote ϕ(i) to be the potential at the end of i^{th} operation. ϕ(0) = 0 ϕ(i) ≥ 0 for all i.

Amortized analysis guarantees the average performance of each operation in the worst case.

If we want to show that the actual cost of n operations is O(g(n)), it suffices to show that the amortized cost is O(g(n)).