CS2040S Cheatsheet

AY21/22 sem 2

Adapted from github.com/jovyntls

Review

An **invariant** can be of correctness or performance. In the context of binary search, the following invariants are true.

- Correctness: $low \leq val \leq high$
- Performance: $high - low \leq \frac{n}{2^r}$

Important properties

Let $T(n) = O(f(n))$ and $S(n) = O(g(n))$

- multiplication: $T(n) * S(n) = O(f(n) * g(n))$
- composition: $f_1 \circ f_2 = O(g_1 \circ g_2)$ only if both increasing
- if/else statements: $\text{cost} = \max(c1, c2) \leq c1 + c2$
- max: $\max(f(n), g(n)) \leq f(n) + g(n)$
- $\Theta(f(n))$ time complexity $\Rightarrow O(f(n))$ space complexity

Notable examples

- $O(\sqrt{n} \log n) = O(n), \quad O(2^{2n}) \neq O(2^n)$
- $O(\log(n!)) = O(n \log n)$
  - Sterling's Approximation, $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- Insert something

Master theorem

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

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

Quicksort

- stable quicksort: $O(\log n)$ space (due to recursion stack)
- worst case $O(n^2)$: pivot first/last/middle element
- worst case $O(n \log n)$: median/random element/fraction
- choose at random: runtime is a random variable

Trees

AVL Trees

- **height-balanced** (maintained with rotations)
  - $\iff$ `|v.left.height - v.right.height|` ≤ 1
- **insertion** - max 2 rotations; **deletion** - recurse all the way up;

Rebalancing

[case 1] Left heavy $\Rightarrow$ right-rotate

[case 2] Right heavy $\Rightarrow$ left-rotate

[case 3] Left, right-heavy $\Rightarrow$ left-rotate(v.left), right-rotate(v)

[case 4] Right, left-heavy $\Rightarrow$ right-rotate(v.right), left-rotate(v)

Important notes

- $n!$ ways of insertion; $\sim 4^n$ shapes of a tree;
- $h < 2 \log n \implies n > 2^{\frac{h}{2}}$
- min height to trigger > 1 rotations is 3 (zero-indexed)
- rotations reduce height, deletion reduce height. Does not "undo" the change in height.

Binary Search Trees (BST)

- balanced: $O(h) = O(\log n)$ (depends on insertion order)
  - $h > \log n - 1 \implies n < 2^{h+1}$ (Geometric sum)
- for a full-binary tree of size $n, \exists k \in \mathbb{Z}^+$ s.t. $n = 2^k - 1$
- `height, h(v) = max(h(v.left), h(v.right))`
- `delete` - $O(h)$
  - no children - remove the node
  - 1 child - remove the node, connect parent to child
  - 2 children - delete successor; replace node w successor
- `searchMin/Max` - $O(h)$ - recurse into left/right subtree
- `successor` - $O(h)$
  - if node has a right subtree: `searchMin(v.right)`
  - else: traverse upwards and return the first parent that contains the key in its left subtree
- **merkle trees**
  - binary tree - nodes augmented with a hash of their children
  - same root value = identical tree
  - useful for comparing blocks of data

Trie

- `search, insert` - $O(L)$ (for string of length L)
- space: $O(\text{size of text} \cdot \text{overhead})$
- does not depend on the number of strings nor total size of text

Interval Trees

- `search(key)` $\Rightarrow O(\log n)$
  - if value is in root interval, return
  - if value > max(left subtree), recurse right
  - else recurse left (go left only when can't go right)
- all-overlaps $\Rightarrow O(k \log n)$ for k overlapping intervals

Orthogonal Range Searching

- **invariant**: the search interval for a left traversal at node v includes the maximum node sub-rooted at that node.
- binary tree; leaves store points, internal nodes store max value in left subtree
- `buildTree(points[])` $\Rightarrow O(n \log n)$. Similar to iterative merge sort.
- `query(low, hight)` $\Rightarrow O(k + \log n)$ for k points
  - `v=findSplit()` $\Rightarrow O(\log n)$ - find node b/w low & high
  - `leftTraversal(v)` $\Rightarrow O(k)$ - either output all the right subtree and recurse left, or recurse right
  - `rightTraversal(v)` - symmetric
- `insert(key), insert(key)` $\Rightarrow O(\log n)$

kd-Tree

- stores geometric data (points in an (x, y) plane)
- alternates splitting (partitioning) via x and y coordinates
- `construct(points[])` $\Rightarrow O(n \log n)$
- `search(point)` $\Rightarrow O(h)$
- `searchMin()` $\Rightarrow O(\sqrt{n})$ $\Rightarrow T(n) = 2T(\frac{n}{4}) + O(1)$

(a, b)-trees

- rules

1. (a, b)-child policy where $2 \leq a \leq (b+1)/2$

	# keys		# children	
node type	min	max	min	max
root	1	$b-1$	2	b
internal	$a-1$	$b-1$	a	b
leaf	$a-1$	$b-1$	0	0

2. an internal node has 1 more child than its number of keys
3. all leaf nodes must be at the **same depth** from the root

- terminology (for a node z)
  - key range - range of keys covered in subtree rooted at z
  - keylist - list of keys within z; treelist - list of z's children
- max height $= O(\log_a n) + 1$; min height $= O(\log_b n)$
- `search(key)` $\Rightarrow O(\log n)$
  - $= O(\log_2 b \cdot \log_a n)$ for binary search at each node
- `insert(key)` $\Rightarrow O(\log n)$
- `split()` a node with too many children.
  - O(B)/O(1) amortized. use median to split the keylist into 2 halves
  - move median key to parent; re-connect remaining nodes
  - if the parent is now unbalanced, recurse upwards; if the root is reached, median key becomes the new root
- `delete(key)` $\Rightarrow O(\log n)$
  - if the node becomes empty, `merge(y, z)` - join it with its left sibling & replace it with their parent
  - if the combined nodes exceed max size: `share(y, z)` = `merge(y, z)` then `split()`
  - **always** look for the offending node and separating key

B-Tree (aka $(B, 2B)$-trees)

- possible augmentation: use a linkedList to connect between each level

Hash Tables

- disadvantage: no successor/predecessor operation

Let the m be the table size; let n be the number of items; let $cost(h)$ be the cost of the hash function

- $load$(hash table), $\alpha = \frac{n}{m}$
  - = average & expected number of items per bucket
- designing hashing techniques
  - **division method**: $h(k) = k \bmod m$ (m is prime)
    - don't choose $m = 2^x$
    - if k and m have common divisor d, only $\frac{1}{d}$ of the table will be used
  - **multiplication method** - $h(k) = (Ak) \bmod 2^w \gg (w - r)$ for odd constant A and $m = 2^r$ and w = size of a key in bits
- **simple uniform hashing assumption**
  - **(1)** every key has an equal probability of being mapped to every bucket; **(2)** 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
- **properties of a good hash function**

1. able to enumerate all possible buckets - $h : U \rightarrow \{1..m\}$
   - for every bucket j, $\exists i$ such that $h(key, i) = j$
2. simple uniform hashing assumption

hashCode

rules for the `hashCode()` method

1. always returns the same value, if object hasn't changed
2. if two objects are equal, they return the same hashCode

rules for the `equals` method

- reflexive, symmetric, transitive for $xRy \iff x.equals(y)$
- consistent - always returns the same answer
- null is null - `x.equals(null) => false`

Chaining

- `insert(key, value)` - $O(1 + cost(h)) \Rightarrow O(1)$
  - for n items: expected maximum cost $= O(\log n)$
    - $= \Theta(\frac{\log n}{\log(\log(n))})$
- `search(key)`
  - worst case: $O(n + cost(h)) \Rightarrow O(n)$
  - expected case: $O(\frac{n}{m} + cost(h)) \Rightarrow O(1)$
- total space: $O(m + n)$

Open Addressing - linear probing

- redefined hash function: $h(k, i) = h(k, 1) + i \bmod m$
- `delete(key)`: use a *tombstone value* - DON'T set to `null`
- **performance** (assume $\alpha < 1$ and uniform hashing)
  - if the table is $\frac{1}{4}$ full, there will be clusters of size $\Theta(\log n)$
  - expected cost of an operation, $E[\#probes] \leq \frac{1}{1-\alpha}$

Double Hashing

for 2 functions f, g, define

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

- if $g(k)$ is relatively prime to m, then $h(k, i)$ hits all buckets
  - e.g. for $g(k) = n^k$, n and m should be coprime.
  - to terminate, $\alpha < 0.5$ and m must be prime

table size

assume chaining & simple uniform hashing

growing the table: $O(m_1 + m_2 + n)$

table growth	resize	insert n items
increment by 1	$O(n)$	$O(n^2)$
double	$O(n)$	$O(n)$, average $O(1)$
square	$O(n^2)$	$O(n)$

Probability Theory

- if an event occurs with probability p, the expected number of iterations needed for this event to occur is $\frac{1}{p}$.
- for **random variables**: expectation is always = probability
- **linearity of expectation**: $E[A + B] = E[A] + E[B]$

Uniformly Random Permutation

- for an array of n items, every of the $n!$ possible permutations are producible with probability of exactly $\frac{1}{n!}$
  - the number of outcomes should distribute over each permutation uniformly. (i.e. $\frac{\text{\# of outcomes}}{\text{\# of permutations}} \in \mathbb{N}$)
  - the number of unique choices
- probability of an item remaining in its initial position $= \frac{1}{n}$
  - $Pr(\text{picking item i and replaced}) = (\frac{k}{i} \times \frac{1}{k} = \frac{1}{i}) \times (1 - \frac{1}{i+1}) \times \cdots \times \frac{n-1}{n} = \frac{1}{n}$
- **KnuthShuffle** $\Rightarrow O(n)$ - for (i = 1...n-1) { swap(i, rand(0, i)) }
  - On the ith iteration, we have a uniformly random permutation of length i

Amortized Analysis

an operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$.

- **binary counter ADT**: increment $\Rightarrow O(1)$
- **hash table resizing**: $O(k)$ for k insertions $\Rightarrow O(1)$
  - search operation: *expected* $O(1)$ (not amortized)

Graphs

- **degree (node)**: number of adjacent edges
- **degree (graph)**: max. degree of a node
- **in-/out-degree**: number of incoming/outgoing edges
- **diameter**: max. shortest path
- even cycles are bipartite!
- 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

Searching

Modifying BFS and DFS is usually a bad choice!

- **breadth-first search** $\Rightarrow O(V+E)$ - queue
  - $O(V)$ - every vertex is added exactly once to a frontier
  - $O(E)$ - every neighbourList is enumerated once
  - parent edges form a tree & shortest path from S
- **depth-first search** $\Rightarrow O(V+E)$ - stack
  - $O(V)$ - `DFSvisit` is called exactly once per node
  - $O(E)$ - `DFSvisit` enumerates each neighbour
    - with adjacency matrix: $O(V)$ per node $\Rightarrow$ total $O(V^2)$

Shortest paths

When modifying the relax condition, check that dijkstra properties still holds.

- **Bellman-Ford** $\Rightarrow O(VE)$
  - **invariant**: On the ith iteration, nodes that are ith hops away on the shortest paths have their estimates determined.
  - $|V|$ iterations of relaxing every edge - terminate when an entire sequence of $|E|$ operations have no effect
- **Dijkstra** $\Rightarrow O((V+E)\log V) = O(E \log V)$
  - no negative weight edges!
  - extending the path **does not** make it shorter
  - using a PQ to track the min-estimate node, relax its outgoing edges and add incoming nodes to the PQ
  - $|V|$ times of `insert/deleteMin` ($\log V$ each)
  - $|E|$ times of `relax/decreaseKey` ($\log V$ each)
  - with fibonacci heap $\Rightarrow O(E + V\log V)$
- **for DAG** $\Rightarrow O(E)$ (topo-sort and relax in this order)
  - longest path: negate the edges/modify relax function
- **for Trees** $\Rightarrow O(V)$ (relax each edge in BFS/DFS order)

Topological ordering

A topological sort order is unique if all pairs of consecutive vertices are connected by an edge.

- **post-order DFS** $\Rightarrow O(V+E)$
  - prepend each node from the post-order traversal
- **Kahn's algorithm (lecture vers.)** $\Rightarrow O(E \log V)$
  - add nodes without incoming edges to the topological order
  - remove min-degree node from PQ $\Rightarrow O(V \log V)$
  - decreaseKey (in-degree) of its children $\Rightarrow O(E \log V)$
- **Kahn's algorithm (tutorial vers.)** $\Rightarrow O(E+V)$
  - add nodes with in-degree=0 to a queue; decrement the in-degree of its adjacent nodes. dequeue & repeat

Spanning trees

Every edge is either blue or red, but never both!

- any 2 subtrees of the MSTs are also MSTs
- for every cycle, the maximum weight edge is NOT in the MST
- for every partition 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.
- **Steiner Tree**: (NP-hard) MST containing a given set of nodes
  1. calculate the shortest path between any 2 vertices
  2. construct new graph on required nodes
  3. MST the new graph and map edges back to original

MST algorithms

- **Prim's** - $O(E \log V)$
  - add the minimum edge across the cut to MST
  - PQ to store nodes (priority: lowest incoming edge weight)
  - each vertex: one insert/extractMin $\Rightarrow O(V \log V)$
  - each edge: one decreaseKey $\Rightarrow O(E \log V)$
- **Kruskal's** - $O(E \log V)$
  - sort edges by weight, add edges if unconnected
  - sorting $\Rightarrow O(E \log E) = O(E \log V)$
  - each edge: find/union $\Rightarrow O(\log V)$ using union-find DS
- **Boruvka's** - $O(E \log V)$
  - each node: store a componentId $\Rightarrow O(V)$
  - one Boruvka step: for each cc, add minimum weight outgoing edge to merge cc's $\Rightarrow O(V+E)$ dfs/bfs
    - at most $O(\log V)$ Boruvka steps
  - update componentIds $\Rightarrow O(V)$
- **directed MST with one root** $\Rightarrow O(E)$
  - for every node, add minimum weight **incoming** edge

Heaps

1. **heap ordering** - priority[parent] $\geq$ priority[child]
2. **complete binary tree** - every level (except last level) is full; all nodes as far left as possible

- operations: all $O(\text{max height}) = O(\lfloor \log n \rfloor)$
  - `insert`: insert as leaf, bubble up to fix ordering
  - `increase/decreaseKey`: bubble up/down larger key
  - `delete`: swap w bottomrightmost in subtree; bubble down
    - `extractMax`: `delete(root)`, bubble down larger key
- heap **as an array**:
  - `left(x)` $= 2x+1$, `right(x)` $= 2x+2$
  - `parent(x)` $= \lfloor \frac{x-1}{2} \rfloor$
- **HeapSort**: $\Rightarrow O(n \log n)$ always
  - unsorted arr to heap: $O(n)$ (bubble down, low to high)
  - heap to sorted arr: $O(n \log n)$ (extractMax, swap to back)

Union Find

- **quick-find** - `int[] componentId`, flat trees
  - $O(1)$ find - check if objects have the same componentId
  - $O(n)$ union - enumerate all items in array to update id
- **quick-union** - `int[] parent`, deeper trees
  - $O(n)$ find - check for same root (common parent)
  - $O(n)$ union - add as a subtree of the root
- **weighted union** - `int[] parent`, `int[] size`
  - $O(\log n)$ find - check for same root (common parent)
  - $O(\log n)$ union - add as a smaller tree as subtree of root
- **path compression** - set parent of each traversed node to the root - $O(\log n)$ find, $O(\log n)$ union
  - a binomial tree remains a binomial tree
- **weighted union + path compression** - for m union/find operations on n objects: $O(n + m\alpha(m,n))$
  - $O(\alpha(m,n))$ find, $O(\alpha(m,n))$ union

Graph tips and tricks

- Add a super node if you need to find min/max across all paths
- You can find longest paths and MST in a DAG in O(V+E) time
- Each layer in the graph should represent one of the graph states. Each edge should represent a state transition.
- If there is a montonically increasing/decreasing property, try using binary search.
- Reversing the edges is a trick you can use to find min paths across different starting points, but same end point.
- Graph duplication is a great way to force the edges to follow a certain path.

data structures assuming $O(1)$ comparison

data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/bst)	$O(\log n), O(h)$	$O(\log n), O(h)$
trie	$O(L)$	$O(L)$
heap	$O(n)$	$O(\log n), O(h)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$
priority queue	(contains) $O(1)$	$O(\log n)$
skip list	$O(\log n)$	$O(\log n)$

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$$
$$T(n) = T(n/2) + O(n) \Rightarrow O(n)$$
$$T(n) = 2T(n/2) + O(1) \Rightarrow O(n)$$
$$T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$$
$$T(n) = 2T(n-1) + O(1) \Rightarrow O(2^n)$$
$$T(n) = 2T(n/2) + O(n \log n) \Rightarrow O(n(\log n)^2)$$
$$T(n) = 2T(n/4) + O(1) \Rightarrow O(\sqrt{n})$$
$$T(n) = T(n-c) + O(n) \Rightarrow O(n^2)$$

orders of growth

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < 2^n < 2^{2n}$$
$$\log_a n < n^a < a^n < n! < n^n$$

sort	best	average	worst	stable?	memory
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$
quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$
heap	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	×	$O(n)$

sorting invariants

sort	invariant (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted
insertion	first k slots are sorted
merge	given subarray is sorted
quick	partition is in the right position

Dynamic Programming

1. **optimal sub-structure** - optimal solution can be constructed from optimal solutions to smaller sub-problems
2. **overlapping sub-problems** - can memoize

- optimal substructure but no overlapping subproblems = divide-and-conquer
- prize collecting: $\Rightarrow O(kE)$ or $O(kV^2)$ for k steps
- vertex cover (set of nodes where every edge is adjacent to at least one node) of a tree: $\Rightarrow O(V)$ or $O(V^2)$
- diameter of a graph: SSSP all $\Rightarrow O(V^2 \log V)$
- APSP: dijkstra all $\Rightarrow O(VE \log V)$ or $O(V^2 E)$
- APSP: floyd warshall $\Rightarrow O(V^3)$
  - $S[v, w, P_k]$ = shortest path from v to w only using nodes from set P
  - $S[v,w,P_8] = \min(S[v,w,P_7], S[v,8,P_7] + S[8,w,P_7])$