# CS2040S

AY23/24S2 Finals

Original by: github.com/jovyntls

Modified by: github.com/zeepheru

## ORDERS OF GROWTH

$$T(n) = \Theta(f(n))$$
$$\iff T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$
$$T(n) = O(f(n))$$
if $\exists c, n_0 > 0$ such that for all $n > n_0, T(n) \le cf(n)$
$$T(n) = \Omega(f(n))$$
if $\exists c, n_0 > 0$ such that for all $n > n_0, T(n) \ge cf(n)$

### properties

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

- addition: $T(n) + S(n) = O(f(n) + g(n))$
- 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: $\text{cost} = \max(c1, c2) \le c1 + c2$
- max: $\max(f(n), g(n)) \le f(n) + g(n)$
- $\Theta(f(n))$ time complexity $\Rightarrow O(f(n))$ space complexity
- space complexity: once we exit the function, release all memory that was used

### Peak-Finding

- 1D-array: $O(\log n)$
- 2D-array: $O(n \log m)$ or $O(n + m)$
  - $T(n, m) = T(n/2, m/2) + O(n + m)$

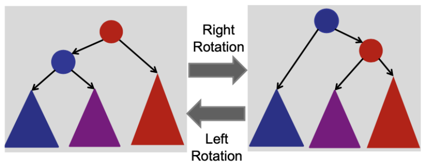
## QUICKSORT

- 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|` $\le 1$
- each node is augmented with its height - `v.height = h(v)`
- space complexity: $O(LN)$ for $N$ strings of length $L$




**insertion** - max 2 rotations; **deletion** - recurse up to root;

### rebalancing

[case 1] B is **balanced: right-rotate**
$$h(L) = h(M), \quad h(R) = h(M) - 1$$
[case 2] B is **left-heavy: right-rotate**
$$h(L) = h(M) + 1, \quad h(R) = h(M)$$
[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**
$$h(L) = h(M) - 1, \quad h(R) = h(L)$$
Note: need to update nodes (weight/max) aft. rotation

### binary search trees (BST)

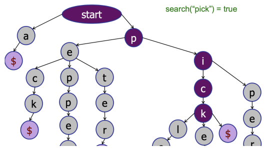
- balanced: $O(h) = O(\log n)$ (depends on insertion order)
- for a full-binary tree of size $n, \exists k \in \mathbb{Z}^+$ s.t. $n = 2^k - 1$
- `height, h(v) = max(h(v.left), h(v.right))`
  - leaf nodes: `h(v)` = 0
- `search`, `insert` - $O(h)$
- `delete` - $O(h)$
  - no children - remove the node
  - 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

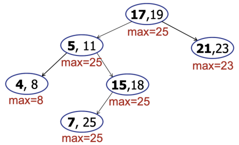
### Trie

- `search, insert` - $O(L)$ (for string of length $L$)
- space: $O(\text{size of text} \cdot \text{overhead})$

### 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



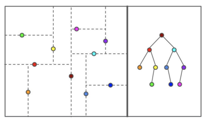




### orthogonal range searching

- binary tree; leaves store points, internal nodes store max value in left subtree
- `buildTree(points[])` $\Rightarrow O(n \log n)$ (space is $O(n)$)
- `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)` $\Rightarrow O(\log n)$
- `2D_query()` $\Rightarrow O(\log^2 n + k)$ (space is $O(n \log n)$)
  - build x-tree from x-coordinates; for each node, build a y-tree from y-coordinates of subtree
- `2D_buildTree(points[])` $\Rightarrow O(n \log n)$

### kd-Tree

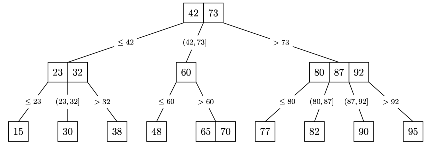
- stores geometric data (points in an $(x, y)$ plane)
- alternates splitting (partitioning) via $x$ and $y$ coordinates



- `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

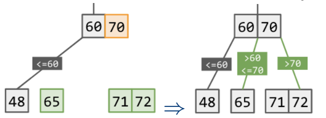
e.g. a (2, 4)-tree storing 18 keys



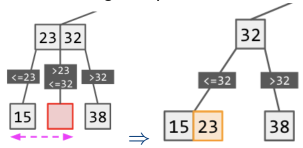

- rules
  1. $(a, b)$-child policy where $2 \le a \le (b + 1)/2$

| | # 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
- 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
  1. use median to split the keylist into 2 halves
  2. move median key to parent; re-connect remaining nodes
  3. (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()`

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

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

## HASH TABLES

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

### hashCode

**(Java) 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

**(Java) 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, $E[\#probes] \le \frac{1}{1-\alpha}$

### 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.

### 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 | $O(n^2)$ | $O(n)$ |

## SET ADT

- ✓ speed ✓ space ✓ no false negatives × no ordering

## fingerprint hash table

- only stores $m$ bits - does not store the key in a table
- $P(\text{no false positives})$ with SUHA $= (1-\frac{1}{m})^n \approx (\frac{1}{e}^{n/m})$
  - i.e. probability of nothing else in the given (same) bucket
  - for $P(\text{no false positives}) < p$, need $\frac{n}{m} \leq \log(\frac{1}{1-p})$

## bloom filter

- 2 hash functions - requires 2 collisions for a false positive
- for $k$ hash functions (assume independent slots):
  - $P(\text{a given bit is } \mathbf{0}) = (1-\frac{1}{m})^{kn} \approx (\frac{1}{e})^{kn/m}$
  - $P(\text{false positive}) = (1-(\frac{1}{e})^{kn/m})^k$
  - $P(\text{no false positives}) < p$, need $\frac{n}{m} \leq \frac{1}{k}\log(\frac{1}{1-p^{1/k}})$
- optimal $k = \frac{m}{n}\ln 2 \quad \rightarrow$ error probability $= 2^{-k}$
- `delete` operation: store **counter** instead of 1 bit
- `insert, delete, query` $\rightarrow O(k)$
- `intersection` (bitwise AND), `union` (OR) $\rightarrow O(m)$
  - gives the same false positives as both

**KnuthShuffle**: $O(n)$ - for (i = n-1..0) { swap(i, rand(0, i)) }

# AMORTIZED ANALYSIS

an operation has **amortized cost** $T(n)$ if for every integer $k$, the cost of ANY $k$ 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

- even cycles are bipartite!
- **Strongly connected**: every $v$ is reachable from $u$.
- 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

- **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

- **Bellman-Ford** $\rightarrow O(VE)$
  - Works on all weighted graphs.
  - Except negative weight cycles.
  - $|V|$ 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!**
  - 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

- **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

- any 2 subtrees of the MSTs are also MSTs (cutting an MST)
- for every **cycle**, the **max 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
- **directed MST with one root** $\rightarrow O(E)$
  - for every node, add minimum weight **incoming** edge
- **Boruvka**
  - Start by adding all minimum adjacent edges of all nodes.
  - Then add minimum outgoing edges of components.

# 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 items have the same componentId
  - $O(n)$ union - enum 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

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 | $O(\log n)$ | $O(\log n)$ |

$$T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow O(n\log n)$$

$$T(n) = T(\frac{n}{2}) + O(n) \Rightarrow O(n)$$

$$T(n) = 2T(\frac{n}{2}) + O(1) \Rightarrow O(n)$$

$$T(n) = T(\frac{n}{2}) + O(1) \Rightarrow O(\log n)$$

$$T(n) = 2T(n-1) + O(1) \Rightarrow O(2^n)$$

$$T(n) = 2T(\frac{n}{2}) + O(n\log n) \Rightarrow O(n(\log n)^2)$$

$$T(n) = 2T(\frac{n}{4}) + O(1) \Rightarrow O(\sqrt{n})$$

$$T(n) = T(n-c) + O(n) \Rightarrow O(n^2)$$

$3^n \neq O(2^n)$ and $2^{log(n)} = O(n)$

$$n^{1/p}\log^k n < O(n); \ \log^p n < O(n)$$

**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? | space | invariant |
|---|---|---|---|---|---|---|
| bubble | $\Omega(n)$ | $n^2$ | $n^2$ | ✓ | $O(1)$ | largest $k$ elem sorted |
| selection | $\Omega(n^2)$ | $n^2$ | $n^2$ | × | $O(1)$ | smallest $k$ elem sorted |
| insertion | $\Omega(n)$ | $n^2$ | $n^2$ | ✓ | $O(1)$ | first $k$ slots sorted |
| merge | $\Omega(n\log n)$ | $n\log n$ | $n\log n$ | ✓ | $O(n)$ | given subarray sorted |
| quick | $\Omega(n\log n)$ | $n\log n$ | $n^2$ | × | $O(1)$ | partition in right position |
| heap | $\Omega(n\log n)$ | $n\log n$ | $n\log n$ | × | $O(n)$ | |

# 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 $O(V^2E)$
- APSP: floyd warshall $\rightarrow O(V^3)$
  - $S[v,w,P_k]$ = shortest path from $v$ to $w$ only using nodes from set $P$
  - $S[v,w,P_8]{=}\min(S[v,w,P_7],S[v,8,P_7]{+}S[8,w,P_7])$