

ORDERS OF GROWTH

- $T(n) = \Theta(f(n))$
- $\Leftrightarrow T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$
- $T(n) = O(f(n))$
- if $\exists c, n_0 > 0$ such that for all $n > n_0$, $T(n) \leq cf(n)$
- $T(n) = \Omega(f(n))$
- if $\exists c, n_0 > 0$ such that for all $n > n_0$, $T(n) \geq cf(n)$

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: cost = $\max(c_1, c_2) \leq c_1 + c_2$
 - max: $\max(f(n), g(n)) \leq f(n) + g(n)$
 - $\Theta(f(n))$ time complexity $\Rightarrow O(f(n))$ space complexity
 - space complexity: once we exit the function, release all memory that was used

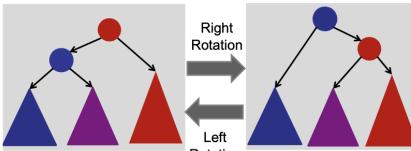
QUICKSORT

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

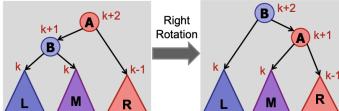


insertion - max 2 rotations; deletion - recurse all the way up;

rebalancing If A is left heavy: opp for right heavy

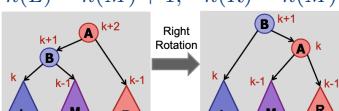
[case 1] B is balanced: right-rotate

$$h(L) = h(M), \quad h(R) = h(M) - 1$$



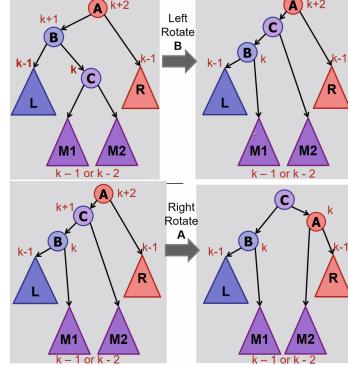
[case 2] B is left-heavy: right-rotate

$$h(L) = h(M) + 1, \quad h(R) = h(M)$$

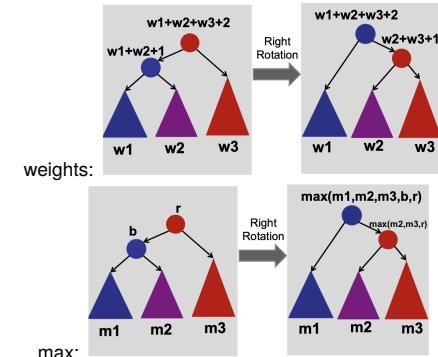


[case 3] B is right-heavy: left-rotate(v.left), right-rotate(v)

$$h(L) = h(M) - 1, \quad h(R) = h(L)$$



updating nodes after 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
- merkle trees
 - binary tree - nodes augmented with a hash of their children
 - same root value = identical tree

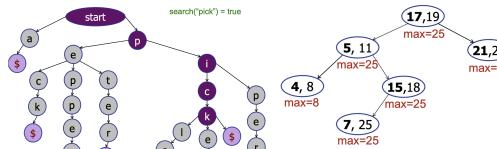
Trie

- 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

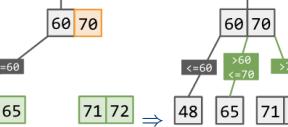
- else recurse left (go left only when can't go right)
- all-overlaps $\Rightarrow O(k \log n)$ for k overlapping intervals



$$T(n) = aT(n-b) + f(n)$$

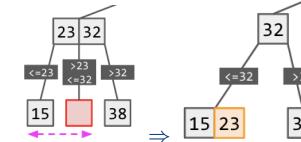
- Case 1: If $a < 1$, then $O(fcn)$
- Case 2: If $a=1$, then $O(n \cdot fcn)$
- Case 3: If $a > 1$, then $O(fcn \cdot a^{n/b})$

- 3. if the parent is now unbalanced, recurse upwards; if the root is reached, median key becomes the new root



$\text{delete}(key) \Rightarrow O(\log n)$

- if the node becomes empty, $\text{merge}(y, z)$ - join it with its left sibling & replace it with their parent



- if the combined nodes exceed max size: $\text{share}(y, z) = \text{merge}(y, z)$ then $\text{split}()$
- B-Tree (aka $(B, 2B)$ -trees)

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

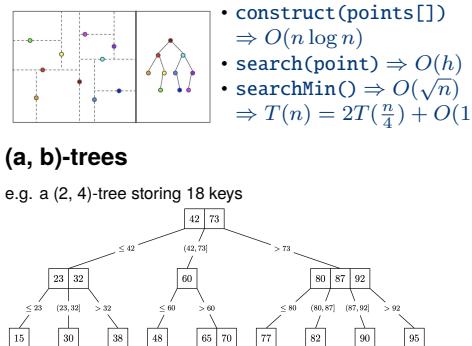
HASH TABLES

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

- $\text{load(hash table)}, \alpha = \frac{n}{m}$
 - = 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 $w = \text{size of a key in bits}$

(a, b)-trees

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



rules

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

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

- an internal node has 1 more child than its number of keys
- 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) $\Rightarrow O(\log n)$
 - = $O(\log_b b \cdot \log_a n)$ for binary search at each node
- insert(key) $\Rightarrow O(\log n)$
- split() a node with too many children
 - use median to split the keylist into 2 halves
 - move median key to parent; re-connect remaining nodes

hashCode() method

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

equals() method

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

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: $O(m + n)$
- everyting mapped to same bucket

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.

table size

assume chaining & simple uniform hashing
growing the table: $O(m_1 + m_2 + \dots + m_n)$

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

SSSP

Weighted

- ① Cyclic: ② Dijkstra ⑥ Bellman-Ford
- ② Acyclic: DAG-SSSP (for directed only)

Unweighted

① Cyclic/Acyclic : BFS

Trees (Undirected Acyclic)

① Weighted/Unweighted : DFS

Cycles

① Undirected : ② BFS ⑥ DFS

② Directed : ② DFS

Inorder \rightarrow Left \rightarrow Root \rightarrow Right

Preorder \rightarrow Root \rightarrow Left \rightarrow Right

Postorder \rightarrow Left \rightarrow Right \rightarrow Root

topological ordering

• post-order DFS $\Rightarrow O(V + E)$

• prepend each node from the post-order traversal

• Kahn's algorithm (lecture vers.) $\Rightarrow O(E \log V)$

$O(V + E)$

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	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✗	$O(n)$

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

- tut07 Qn6 - Queue w enqueue, dequeue and sum methods

- Adjacency Matrix (Used when graph dense)
- ① Dijkstra: $\Theta(n^3)$
 - ② Bellman-Ford: $\Theta(n^3)$
 - ③ BFS/DFS: $\Theta(n^2)$
 - ④ Prim: $\Theta(n^3 + m\log n) \approx \Theta(mn)$

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

- 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 After K iter, distance of every vertex whose shortest weighted path from S was at most K edges is settled

• Bellman-Ford $\Rightarrow O(VE)$

• $|V|$ 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)

• for Graph w no weight \Rightarrow BFS

topological ordering

• post-order DFS $\Rightarrow O(V + E)$

• prepend each node from the post-order traversal

• Kahn's algorithm (lecture vers.) $\Rightarrow O(E \log V)$

$O(V + E)$

- 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

longest Path Problem

• DAG \rightarrow Toposort + relax edges (using max)

• For graphs w no tue cycles \rightarrow Bellman-Ford w relax(cusing max)

• To detect tue cycles, run BF n times and check if dist.est got "better". If yes, we cycle.

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
- For decreasekey we need,
① Hashable (ID \rightarrow index) ② Array (index \rightarrow ID)

- 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 $O(\max \text{height}) = O(\lfloor \log n \rfloor)$

- **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)

- Fibonacci Heap does - insert, decrease in O(1).

DYNAMIC PROGRAMMING

extrusion: $O(\log n)$

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 $\Rightarrow O(V^3)$ **dense graphs better**

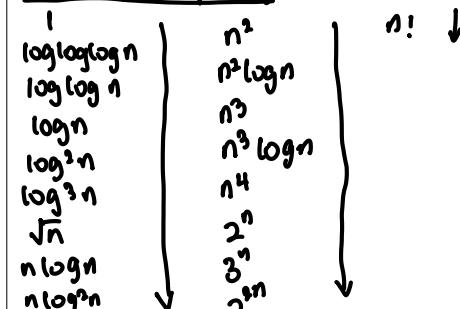
• $S[v, w, P_k]$ = shortest path from v to w only using nodes from set P

• $S[v, w, P_8] = \min(S[v, w, P_7], S[v, 8, P_7] + S[8, w, P_7])$

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` **Worst Case: $O(\log n)$**
 - $O(\log n)$ 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 **Worst case: $O(n)$**
 - 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

Order of complexity



$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$

$T(n) = T(n/2) + O(n) \Rightarrow O(n)$

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

$T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$

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

$T(n) = 2T(n/2) + O(n \log n) \Rightarrow O(n(\log n)^2)$

$T(n) = 2T(n/4) + O(1) \Rightarrow O(sqrt(n))$

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

master theorem

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

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

orders of growth

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < 2^n < 2^{2n}$

$\log_a n < n^a < a^n < n! < n^n$