CS2040S

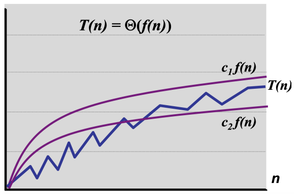
AY20/21 sem 2

github.com/jovyntls

ORDERS OF GROWTH

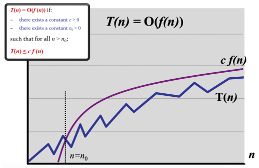
definitions

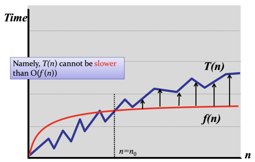
$$T(n) = \Theta(f(n))$$
$$\iff T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$



$$T(n) = O(f(n))$$
if $\exists c, n_0 > 0$ such that for all $n > n_0, T(n) \leq cf(n)$

$$T(n) = \Omega(f(n))$$
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 functions are increasing
- if/else statements: $\text{cost} = \max(c1, c2) \leq c1 + c2$
- max: $\max(f(n), g(n)) \leq f(n) + g(n)$

notable

- $\sqrt{n} \log n$ is $O(n)$
- $O(2^{2n}) \neq O(2^n)$
- $O(\log(n!)) = O(n \log n) \Rightarrow$ sterling's approximation
- $T(n-1) + T(n-2) + \cdots + T(1) = 2T(n-1)$

master theorem

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

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

space complexity

- $\Theta(f(n))$ time complexity $\Rightarrow O(f(n))$ space complexity
- the maximum space incurred **at any time at any point**
- NOT the maximum space incurred altogether!
- assumption: once we exit the function, we release all memory that was used

SORTING

overview

- **BubbleSort** - compare adjacent items and swap
- **SelectionSort** - takes the smallest element, swaps into place
- **InsertionSort** - from left to right: swap element leftwards until it's smaller than the next element. repeat for next element
  - tends to be faster than the other $O(n^2)$ algorithms
- **MergeSort** - mergeSort 1st half; mergeSort 2nd half; merge
- **QuickSort**
  - partition algorithm: $O(n)$
  - stable quicksort: $O(\log n)$ space (due to recursion stack)
    - first element as partition. 2 pointers from left to right
      - left pointer moves until element > pivot
      - right pointer moves until element < pivot
      - swap elements until left = right.
    - then swap partition and left=right index.

optimisations of QuickSort

- array of duplicates: $O(n^2)$ without 3-way partitioning
- stable if the partitioning algo is stable.
- extra memory allows quickSort to be stable.

choice of pivot

- worst case $O(n^2)$: first/last/middle element
- worst case $O(n \log n)$: median/random element
  - split by fractions: $O(n \log n)$
- choose at random: runtime is a random variable

quickSelect

- $O(n)$ - to find the k^{th} smallest element
- after partitioning, the partition is always in the correct position

TREES

binary search trees (BST)

- a BST is either empty, or a node pointing to 2 BSTs.
- tree balance depends on order of insertion
- balanced tree: $O(h) = O(\log n)$
- for a full-binary tree of size $n, \exists k \in \mathbb{Z}^+$ s.t. $n = 2^k - 1$

BST operations

- `height, h(v) = max(h(v.left), h(v.right))`
  - leaf nodes: `h(v)` = 0
- modifying operations
  - `search`, `insert` - $O(h)$
  - `delete` - $O(h)$
    - case 1: no children - remove the node
    - case 2: 1 child - remove the node, connect parent to child
    - case 3: 2 children - delete the successor; replace node with successor
- query operations
  - `searchMin` - $O(h)$ - recurse into left subtree
  - `searchMax` - $O(h)$ - recurse into 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

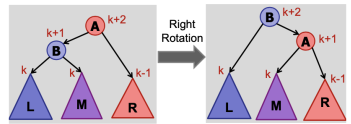
AVL Trees

- **height-balanced** (maintained with rotations)
  - $\iff$ `|v.left.height - v.right.height|` ≤ 1
- each node is augmented with its height - `v.height = h(v)`
- space complexity: $O(LN)$ for N strings of length L

rebalancing

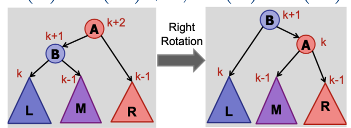
[case 1] B is **balanced: right-rotate**

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



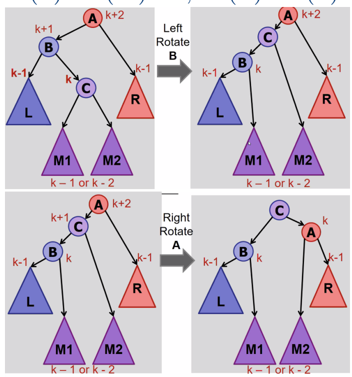
[case 2] B is **left-heavy: right-rotate**

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



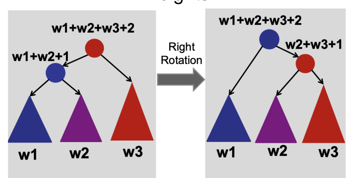
[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

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

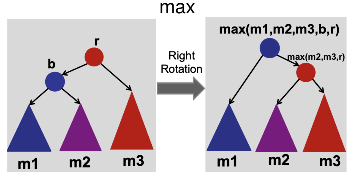


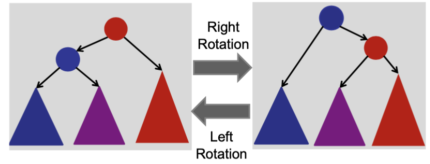
updating nodes after rotation

weights



max





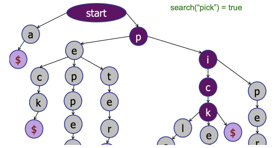
- insertion: max. 2 rotations
- deletion: recurse all the way up
- rotations can create every possible tree shape.

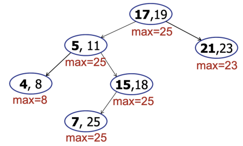
Trie

- `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

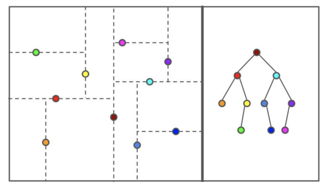




orthogonal range searching

- 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[])` $\Rightarrow O(n \log n)$

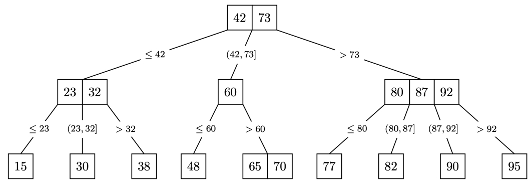
kd-Tree



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

(a, b)-trees

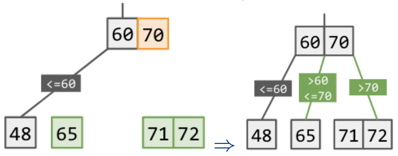
e.g. a (2, 4)-tree storing 18 keys



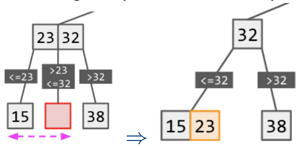
- rules
  1. (a, b)-child policy where $2 \le a \le (b+1)/2$

	# 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
- 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
  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

- $(B, 2B)$-trees $\Rightarrow (a, b)$-tree where $a = B, b = 2B$
- possible augmentation: use a linkedList to connect between each level

Merkle Trees

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

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 number of items per bucket
  - = 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

hashing assumptions

- **simple uniform hashing assumption**
  - every key has an equal probability of being mapped to every bucket
  - 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

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 - `x.equals(x) => true`
- symmetric - `x.equals(y)` $\Rightarrow$ `y.equals(x)`
- transitive - `x.equals(y), y.equals(z)` $\Rightarrow$ `x.equals(z)`
- 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**
  - if the table is $\frac{1}{4}$ full, there will be clusters of size $\Theta(\log n)$
  - expected cost of an operation, $E[\#probes] \le \frac{1}{1-\alpha}$ (assume $\alpha < 1$ and uniform hashing)
  - degrades badly as $\alpha \to 1$
- **advantages**
  - saves space (use empty slots vs linked list)
  - better cache performance (table is one place in memory)
  - rarely allocate memory (no new list-node allocation)
- **disadvantages**
  - more sensitive to choice of hash function (clustering)
  - more sensitive to load (as $\alpha \to 1$, performance degrades)

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

let $m_1 =$ size of the old hash table; $m_2 =$ size of the new hash table; $n =$ number of elements in the hash table

- growing the table: $O(m_1 + m_2 + n)$
- rate of growth

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 ordering
- ✓ no false negatives × may have false positives
- **hash table**: more space, but resolves collisions

fingerprint hash table

- only stores m bits - does not store the key in a table
- P(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(no false positives) $< p$, need $\frac{n}{m} \le \log(\frac{1}{1-p})$

bloom filter

- 2 hash functions - requires 2 collisions for a false positive
- for k hash functions (assume independent slots):
  - P(a given bit is **0**) $= (1 - \frac{1}{m})^{kn} \approx (\frac{1}{e})^{kn/m}$
  - P(false positive) $= (1 - (\frac{1}{e})^{kn/m})^k$
  - P(no false positives) $< p$, need $\frac{n}{m} \le \frac{1}{k} \log(\frac{1}{1-p^{1/k}})$
- optimal $k = \frac{m}{n} \ln 2$ $\Rightarrow$ 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

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 equal to the 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. $\frac{\text{\# of outcomes}}{\text{\# of permutations}} \in \mathbb{N}$)
- probability of an item remaining in its initial position $= \frac{1}{n}$
- **KnuthShuffle** $\Rightarrow O(n)$ - for every element in array A, swap it with a random index in array A.

AMORTIZED ANALYSIS

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

- **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

- **breadth-first search** $\Rightarrow O(V + E)$
  - $O(V)$ - 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
  - implement with queue
- **depth-first search** $\Rightarrow O(V + E)$
  - $O(V)$ - `DFSvisit` is called exactly once per node
  - $O(E)$ - `DFSvisit` enumerates each neighbour
    - with adjacency matrix: $O(V)$ per node $\Rightarrow$ total $O(V^2)$
  - implement with stack

shortest paths

- **Bellman-Ford** $\Rightarrow O(VE)$
  - $|V|$ iterations of relaxing every edge - terminate when an entire sequence of $|E|$ operations have no effect
- **Dijkstra** $\Rightarrow O((V + E) \log V) = O(E \log V)$
  - using a PQ to track the min-estimate node, relax its outgoing edges and add incoming nodes to the PQ
  - no negative weight edges!
  - $|V|$ 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
- 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

- 2 properties:
  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 leftwards
  - `delete`: swap w bottomrightmost in subtree; bubble down
    - `extractMax`: `delete(root)`
- 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 array to heap: $O(n)$ (bubble down, low to high)
  - heap to sorted array: $O(n \log n)$ (extractMax, swap to back)

UNION-FIND

- **union** - connect 2 objects
- **find** - check if objects are connected
- **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

DYNAMIC PROGRAMMING

1. **optimal sub-structure** - optimal solution can be constructed from optimal solutions to smaller sub-problems
   - greedy algorithms / divide-and-conquer algorithms
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 $O(V^2)$
- all pairs shortest path: dijkstra all $\Rightarrow O(VE \log V)$
- diameter of a graph: SSSP all $\Rightarrow O(V^2 \log V)$
- floyd warshall $\Rightarrow O(V^3)$
  - $S[v, w, P_k]$ = shortest path from v to w only using nodes from set P
  - $S[v, w, P_8] = \min(S[v, w, P_7], S[v, 8, P_7] + S[8, w, P_7])$

---

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

searching

search	average
linear	$O(n)$
binary	$O(\log n)$
quickSelect	$O(n)$
interval	$O(\log n)$
all-overlaps	$O(k \log n)$
1D range	$O(k + \log n)$
2D range	$O(k + \log^2 n)$

data structures assuming $O(1)$ comparison cost

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)/binary)	$O(\log n)$ or $O(h)$	$O(\log n)$ or $O(h)$
trie	$O(L)$	$O(L)$
heap	$O(\log n)$ or $O(h)$	$O(\log n)$ or $O(h)$
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	$O(\log n)$	$O(\log n)$

orders of growth

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$$
$$\log_a n < n^a < a^n < n! < n^n$$

orders of growth

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