CS2040S

Java

- Use `Object.equals(Object o)` to compare objects
- String concatenation takes $O(\text{length})$

Big-O notation

Upper bound $T(n) = O(f(n))$ if for some $c > 0$ and $n_0 > 0$,
$$T(n) \leq cf(n)$$
for all $n > n_0$.

Lower bound $T(n) = \Omega(f(n))$ if for some $c > 0$ and $n_0 > 0$,
$$T(n) \geq cf(n)$$
for all $n > n_0$.

Tight bound $T(n) = \Theta(f(n))$ if
$$T(n) = O(f(n)) \quad \text{and} \quad T(n) = \Omega(f(n))$$

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

1. If $T(n)$ is a polynomial of degree k then
$$T(n) = O(n^k)$$
2. Addition: $T(n) + S(n) = O(f(n) + g(n))$
3. Multiplication: $T(n) \times S(n) = O(f(n) \times g(n))$
4. Max: $\max(T(n), S(n)) = O(f(n) + g(n))$
5. Composition: $T(S(n)) = O(f \circ g(n))$ only if both functions are increasing

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

Stirling's approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
Used to show that $\log(n!) = O(n \log n)$

Geometric series
$$S_n = a + ar + ar^2 + \cdot + ar^{n-1} = \frac{a(1-r^n)}{1-r}$$

Harmonic series $\sum_{i=1}^{\infty} \frac{1}{i} = O(\log n)$

Logarithm change of base $\log_b a = \dfrac{\log_x b}{\log_x a}$

Master theorem Comparing $f(n)$ with $n^{\log_b(a)}$ as polynomials, for $a \geq 1$ and $b > 1$,
$$\begin{aligned} &T(n) \\ &= aT\left(\frac{n}{b}\right) + f(n) \\ &= \begin{cases} \Theta(n^{\log_b(a)}) & \text{if } f(n) < n^{\log_b(a)} \\ \Theta(n^{\log_b(a)} \log n) & \text{if } f(n) = n^{\log_b(a)} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b(a)} \end{cases} \end{aligned}$$

Algorithms

(with binary search as example algo)

Precondition Fact that is true before algorithm runs. e.g. Array is sorted

Postcondition Fact that is true when algorithm ends. e.g. If element in array, then `A[begin] = key`

Invariant Relationship between variables that is always true

Loop invariant Invariant at beginning/end of loop e.g. `A[begin] <= key <= A[end]`, i.e. key is in range

Peak finding

Assume `A[-1]` and `A[n]` are `-INT_MAX`.

```
FindPeak(A, n)
  mid = n/2
  if A[mid+1] > A[mid] then recurse on
      right half
  elif A[mid-1] > A[mid] then recurse
      on left half
  else A[mid] is a peak
```

If we recurse into right half, then:

- Given that `A[mid] < A[mid+1]` (condition for recursing into right half)
- Assuming no peak, then `A[mid] < A[mid+1] < A[mid+2] < ... < A[n-1] > A[n] = -INT_MAX`
- Hence peak must exist

Sorting

Bubble compare adjacent and swap to make right > left

Selection find smallest element and swap it to end of sorted region

Insertion swap each new element until it is correctly placed within sorted region

Small arrays Insertion sort is stable, works fast for nearly sorted arrays, has little overhead.

Merge (recursive) sort each half and merge

Quick (recursive) partition about chosen pivot

In-place partitioning $O(n)$

1. Choose a random pivot (for good quicksort performance), and swap it to the start
2. Increase `lptr` while element at left < pivot
3. Decrease `rptr` while element at right > pivot
4. If not yet at centre, swap pointers and repeat

Quicksort variations

- Is stable only if the partitioning is stable. Requires $O(n)$ extra space to store initial indices
- If pivot splits by a fraction, good enough
- 3-way partitioning: pack duplicates in middle to eliminate duplicate elements worst case
- Paranoid: force a good pivot
- k-pivots: $O(k \log k)$ to sort pivots + $O(n \log k)$ to binary search the pivots to choose correct bucket to place the new item. $T(n) = O(n \log_k n \log k) = O(n \log n)$
- 2-pivot quicksort is in practice better, because of cache performance

Probability

Linearity of expectation For random variables X and Y:
$$E(aX + bY) = aE(X) + bE(Y)$$

Expected trials If an event X has probability of success p, then an expected $\frac{1}{p}$ trials is required for 1 success.

Order Statistics

Find kth smallest element Quicksort but only recurse on relevant side - $O(n)$

Interval Tree

Each node stores an interval, keyed on left endpoint, augmented with max right endpoint in subtree

Search $O(\log n)$

1. If value in node interval, return node
2. If `value > node.left.max`, recurse right
3. Else recurse left

All-overlaps If we want to find all k intervals that contain value - $O(k \log n)$

1. Search for interval
2. Add to list and delete interval

Orthogonal Range Searching

- Leaf nodes contain points
- Internal nodes contain max in left subtree
- Build tree: $O(n \log n)$
- Space: $O(n)$

Search Find number of points in range $[a, b]$ - $O(k + \log n)$, where k is number of points found

1. Find split node, highest node with key in range $[a, b]$
2. Do left child traversal
   - If node in query range, then add entire right subtree to list, and recurse left
   - Else recurse right
3. Do right child traversal (similar)

n dimensional search

- Recursively store $d-1$ dim range tree in each node of a 1D range tree
- Build tree: $O(n \log^{d-1} n)$
- Query: $O(\log^d n + k)$
- Space: $O(n \log^{d-1} n)$
- 2D tree Rotate: $O(n)$

Trees

Binary trees

- A binary tree is either empty, or a node pointing to two binary trees.
- In a balanced tree, $h = O(\log n)$
- Height is no. of edges on longest path to leaf

The following operations are all $O(\log n)$:

searchMin keep going left

searchMax keep going right

search, insert go left/right depending on comparison of new key with cur key

successor

- If node has right child: `right.searchMin()`
- Else traverse upwards until ancestor contains node in left subtree, then return ancestor

predecessor

- If node has left child: `left.searchMax()`
- Else traverse upwards until ancestor contains node in right subtree, then return ancestor

delete

- 0 children: remove node
- 1 child: remove node, connect parent to child
- 2 children: delete successor (at most 1 child), replace node value with successor value

The following operations are all $O(n)$:

in-order left, self, right

pre-order self, left, right

post-order left, right, self

level-order decreasing height (BFS)

Ancestor Node x is an ancestor of node $y \Leftrightarrow x$ comes before y in pre-order, AND y comes before x in post-order.

$O(1)$ work

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

$T(n) = kT(\frac{n}{k}) + O(1) = O(n)$

$T(n) = kT(\frac{n}{2k}) + O(1) = O(\sqrt{n})$

Not $O(1)$ work

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

$T(n) = kT(\frac{n}{k}) + O(n) = O(n \log n)$

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

Subtract in recurrence

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

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

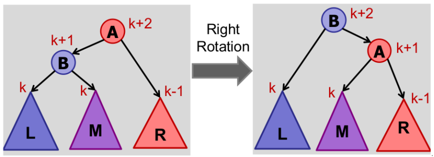
$T(n) = T(n-1) + T(n-2) + O(1) = O(\phi^n)$

AVL trees

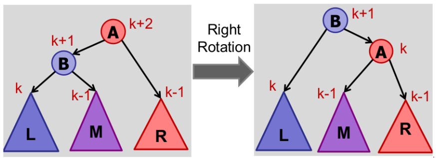
- A node v is height balanced if `|v.L.height - v.R.height| <= 1`
- A height balanced tree has at $h < 2\log n$ (actually, approximately $\frac{1}{\log \phi} \log n$)

Balancing Assume A is left-heavy. Otherwise, if A is right-heavy, substitute ALL left, right with right, left

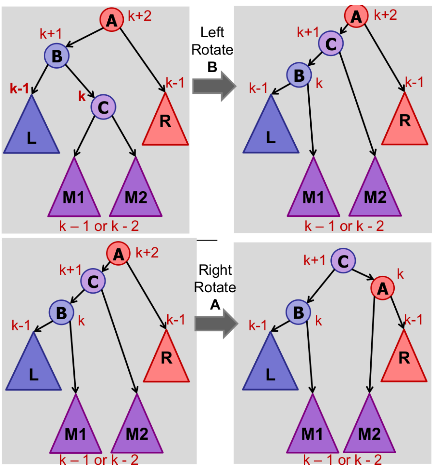
Case 1: B is **balanced:** `rightRotate(A)`



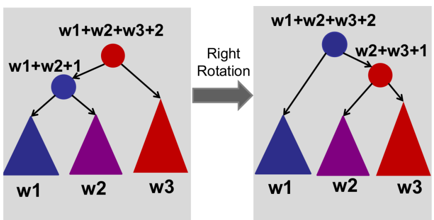
Case 2: B is **left-heavy:** `rightRotate(A)`



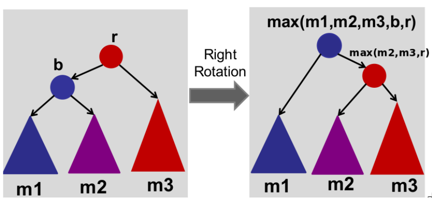
Case 3: B is **right-heavy:** `leftRotate(A.left); rightRotate(A)`



Update weights after `rightRotate(A)`



Update max after `rightRotate(A)`



- Insertion: max 2 rotations
- Deletion: max $O(\log n)$ rotations

Rank of node Position in in-order

```
rank(node)
rank = node.left.weight + 1
while node != null
  if node is right child
    rank += node.parent.left.weight + 1
  node = node.parent
return rank
```

Trie

- search, insert: $O(L)$
- space: $O\left(\sum L + \text{overhead}\right)$

k-d tree

- Stores coordinates in $x - y$ plane
- Levels alternate between splitting plane by x or y

Search node $O(\log n)$

1. If horizontal split, compare x-coordinate
2. If vertical split, compare y-coordinate
3. $O(h)$ time

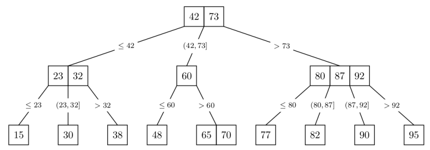
Search min $O(\sqrt{n})$ (e.g. min x)

1. If horizontal split, recurse left child
2. If vertical split, recurse on both children
3. $T(n) = 2T(\frac{n}{4}) + O(1)$

Build $O(n \log n)$

1. Choose either x or y.
2. Quickselect median of x or y
3. Split array into two halves using median
4. Paritioning: $O(n)$

(a, b)-tree



Sample $(2, 4)$-tree

Rules

1. (a, b) child policy

	# Keys		# Children	
Node	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. A non-leaf node must have one more child than its number of keys
3. All leaf nodes must all be at the same depth

Definitions

- Key range: Range of keys allowed in a subtree (wrt parent)
- Key list: List of keys in node (assume sorted)
- Tree list: List of children

B-tree simply $(B, 2B)$ trees

Search $O(\log n)$:

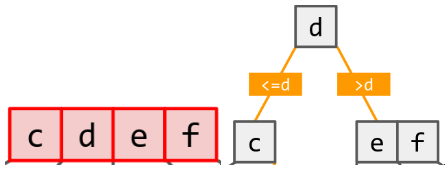
- $O(\log b)$ binary search keylist for subtree containing the key to search
- Repeat along height of $O(\log_a n)$

Insert $O(\log n)$: Like search, then perform split/merge as necessary

- Proactive: preemptively split nodes at full capacity (only applies if $b \geq 2a$)
- Passive: insert then check (potentially splitting all the way to root)

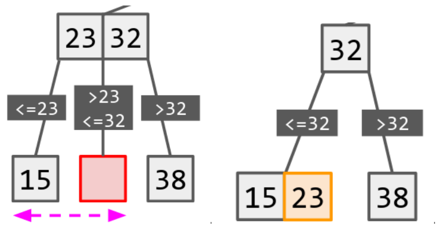
Delete $O(\log n)$: Like search, then perform split/merge as necessary

Split



1. Pick median key of overfull range z as new split key k
2. Put k into parent
3. Split z into LHS and RHS of k
4. If parent is overfull, `split(parent)`

Merge



Let d be deleted node, and l be left sibling of d. Assume keylist of l and d have $< b-1$ keys in total. Otherwise use share below

1. Pick key k from parent, on left of d
2. Move k to keylist of l
3. Merge d keylist, treelist into l
4. Delete d

Share Merge; split newly combined node

Hashing

Hash functions

- Maps universe to keys in $[1, m]$
- Store item with key k in bucket $hash(k)$
- Since universe size larger, collisions inevitable by pigeonhole principle

Simple uniform hash

- Each key has equal probability of being mapped to each bucket
- Keys are mapped independently

Chaining Assume n keys have been inserted into hash table of size m.

- Each bucket c stores linked list of items with $hash(k) = c$
- Insert: $O(1 + hash) = O(1)$
- Total space: $O(n + m)$

Search

- Worst case: $O(n + hash) = O(n)$
- Expected case: $O(1 + \frac{n}{m})$, good m: O(1)

Expected max cost of n inserts

- $O(\log n) = \Theta(\frac{\log n}{\log \log n})$

Sort	Best	Average	Worst	Stable	Memory	Invariant (after k iterations)
Bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$	last k elements in correct position
Selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$	first k elements in correct position
Insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$	(original) first k elements in relative sorted order
Merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$	subarray is sorted
Quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$	pivot is in correct position