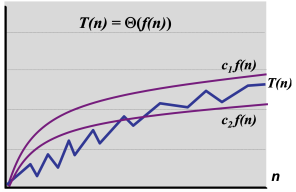
CS2040S
AY23/24 sem 2 midterms

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

$$T(n) = aT(\tfrac{n}{b}) + O(n^d) \quad a \geq 0, b > 1, d \geq 0$$
$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } d < \log_b a \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } d = \log_b a \\ \Theta(n^d) & \text{if } d > \log_b a \text{ polynomially} \end{cases}$$

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

Binary Search

- Invariant: $A[begin] \leq target \leq A[end]$
- Time : $O(\log n)$
- Space : $O(1)$
- When to use:
  - Input is sorted, or can be manipulated to be sorted
  - By checking a value, can eliminate all smaller or bigger
  - Find local minimum or local maximum

SORTING

overview

- **BubbleSort** - compare adjacent items and swap
  - Best Case $O(n)$ - Already Sorted
  - Worst Case $O(n^2)$ - Reverse Sorted / Smallest at back
- **SelectionSort** - takes the smallest element, swaps into place
  - Always finds min(element) $n-1$ times for n elements.
- **InsertionSort** - from left to right: swap element leftwards until it's smaller than the next element. repeat for next element
  - Best Case $O(n)$ - Already Sorted
  - 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
    - 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/largest element
  - Select a pivot and partition the array. The partition will always be in the correct position.
  - If pivot is already the k^{th} smallest element: return it.
  - Else: recurse into the < pivot partition, if pivot position is > k. And vice-versa.
- When to use:
  - Finding median
  - Finding k^{th} smallest elements (use QuickSelect to find k^{th} element, then linear search

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

traversal

pre-order DFS	*in-order DFS*	*post-order DFS*
Root-Left-Right	Left-Root-Right	Left-Right-root

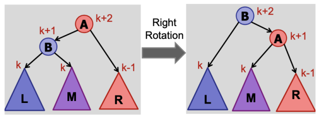
AVL Trees

- **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
- Use AVL when a sorted order is required

rebalancing

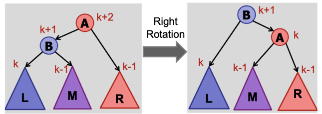
[case 1] B is **balanced: right-rotate**

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



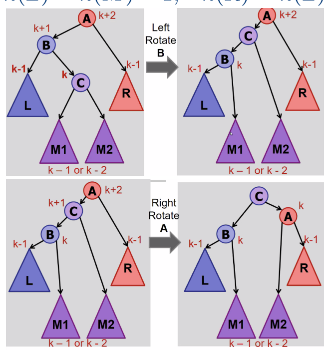
[case 2] B is **left-heavy: right-rotate**

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



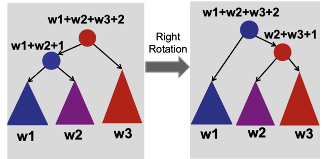
[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

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

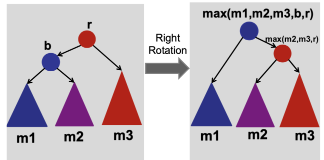


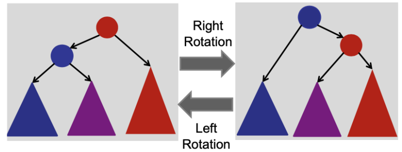
updating nodes after rotation

weights



max





- insertion: max. 2 rotations - $O(1)$
- deletion: recurse all the way up - $O(h)$
- rotations can create every possible tree shape.

Trie

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

Augmented Data Structures

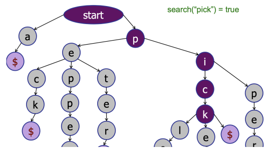
- When num of inserts > num of selects : Unsorted Array
- When num of selects > num of inserts : Sorted Array
- Best of both worlds : Binary Search Tree

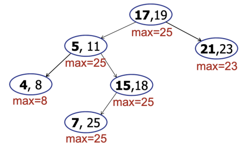
Order Statistics Tree

- AVL tree where each node store count of nodes in subTree
- Select(rank) using weight in $O(\log n)$
  - weight(v) = weight(v.left) + weight(v.right) + 1
  - Rank in subTree = left.weight + 1
  - If go right subTree, remember to minus left weight
- Use when need a count of elements smaller/larger than some value, for cheap counting

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

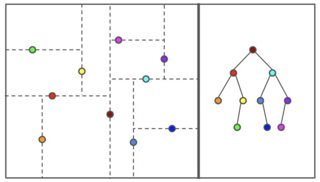




orthogonal range searching

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

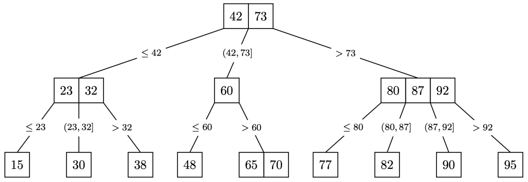
kd-Tree



- 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

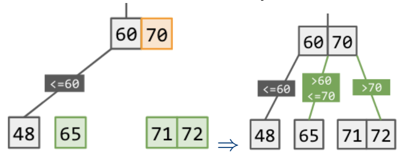
e.g. a (2, 4)-tree storing 18 keys



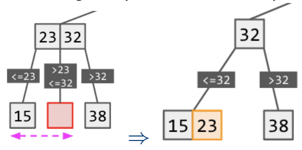
- rules
  1. (a, b)-child policy where $2 \le a \le (b+1)/2$

	# 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)` $\Rightarrow O(\log n)$
  - $O(\log_2 b \cdot \log_a n)$ - binary search for key at node $\cdot$ height
- `insert(key)` $\Rightarrow O(\log n)$ - insert only at leaves
  - Navigate to (Search for) suitable leaf and add to key list
  - If leaf node becomes too large, redistribute using `split()`
- `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()`
  - if root node is too large, promote a new root
- Proactive vs Passive insertion approaches:
  - Proactive: Preemptively spilt node at full capacity during search phase
  - Passive: Insert first, then check parent for violation (may have to split all the way to the top)
- Insertion will never violate Rule 3 as the tree grows root upwards. Leaves are always on the same level

B-Tree

- $(B, 2B)$-trees $\Rightarrow (a, b)$-tree where $a = B, b = 2B$
- Augmentation: A linkedList connects each level

Stack and Queue

- Stack (LIFO / FILO)
  - Push: $O(1)$ - Push item to top of the stack
  - Pop: $O(1)$ - Remove and return item at the top of stack
  - Peek: $O(1)$ - Return item at top only
- Queue (FIFO)
  - Enqueue: $O(1)$ - Push item to back of the queue
  - Dequeue: $O(1)$ - Remove and return item at front of queue
  - Peek: $O(1)$ - Return item at front of queue only

Implementing Stacks and Queues

Array

- Stack:
  - Push: Append to end of array - O(1)
  - Pop: Remove from end of array - O(1)
- Queue:
  - Enqueue: Append to end of array - O(1)
  - Dequeue: Remove from front of array - O(n) !!

Linked List

- Stack:
  - Push: Append to end of linked list - O(1) or O(n)? (O(1) if has tail ptr)
  - Pop: Remove from end of linked list - O(1) or O(n)? (O(1) if has tail ptr)
- Queue:
  - Enqueue: Append to end of linked list - O(1) or O(n)? (O(1) if has tail ptr)
  - Dequeue: Remove from front of linked list: O(1)

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**: $E[A + B] = E[A] + E[B]$

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

sorting invariants

sort	invariant (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted into final positions
insertion	first k slots are sorted (not final positions)
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	$O(k + \log^2 n)$

Logarithm Properties

$$\log_a c \Rightarrow \frac{log_b c}{log_b a}$$

$$\log n^c \Rightarrow c \log n$$

$$\log ab \Rightarrow loga + logb$$

$$\log \frac{a}{b} \Rightarrow \log a - \log b$$

$$a^{\log b} \Rightarrow b^{\log a}$$

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	$O(L)$	$O(L)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$

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

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

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

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

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

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

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

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

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

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