,

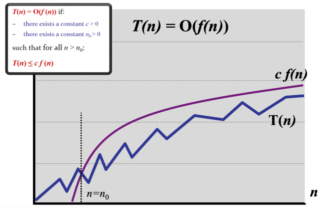
CS2040S
AY20/21 sem 2
by jovyntls

ORDERS OF GROWTH

definitions

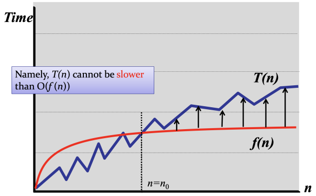
$$T(n) = O(f(n))$$

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



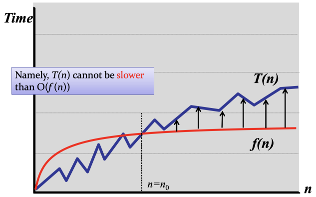
$$T(n) = \Omega(f(n))$$

if $\exists c, n_0 > 0$ such that for all $n > n_0, T(n) \geq cf(n)$



$$T(n) = \Theta(f(n))$$

$\iff T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$



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: $f_1 \circ f_2 = O(g_1 \circ g_2)$
- if/else statements: $\text{cost} = \max(c1, c2) \leq c1 + c2$

notable

- $\sqrt{n} \log n$ is $O(n)$
- $O(2^{2n}) \neq O(2^n)$
- $O(\log(n!)) = O(n \log n)$

SORTING

overview

Bubble Sort
- compare adjacent items and swap

Selection Sort
- takes the smallest element and swaps into place
- after k iterations: the first k elements are sorted

Insertion Sort
- 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

Merge Sort
- divide and conquer algorithm
- mergeSort first half; mergeSort second half; merge

Quick Sort
- partition algorithm: $O(n)$
  - take 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 time of $O(n^2)$
  - first/last/middle element
- worst case (expected) time of $O(n \log n)$:
  - median/random element
  - split by fractions: O(nlogn)
- choose at random: runtime is a random variable

quickSelect

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

TREES

binary search trees (BST)

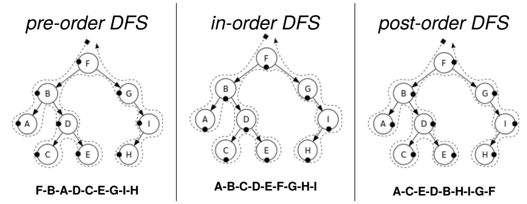
- a BST is either empty, or a node pointing to 2 BSTs.
- tree balance depends on order of insertion
- balanced tree: $O(h) = O(\log n)$

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

< successor code >

traversal



AVL Trees

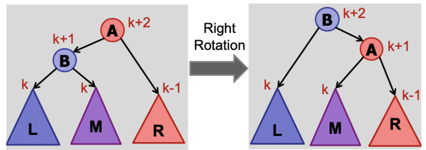
- **height-balanced**
  - $\iff$ `|v.left.height - v.right.height|` ≤ 1
- each node is augmented with its height - `v.height = h(v)`

rebalancing

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

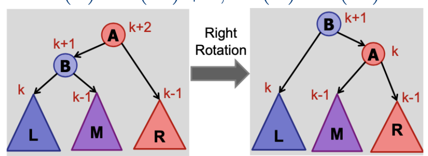
[case 1] B is **balanced: right-rotate**

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



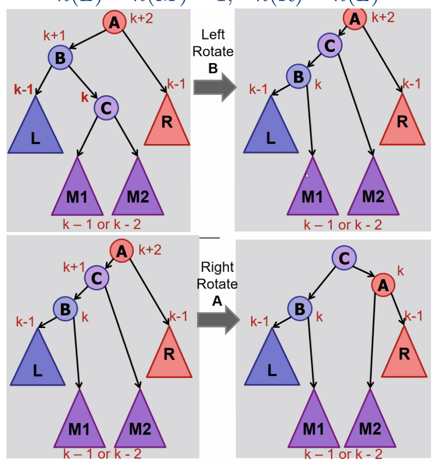
[case 2] B is **left-heavy: right-rotate**

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

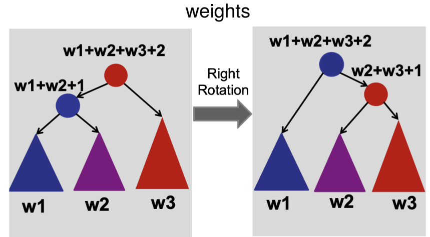


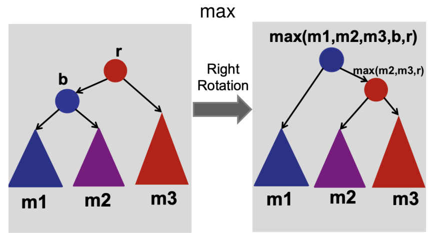
[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

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



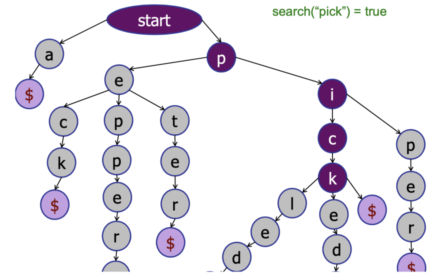
updating nodes after rotation





Trie

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



PROBABILITY THEORY

if an event occurs with probability p, the expected number of iterations needed for this event to occur is $\frac{1}{p}$.

random variables: expectation is always equal to the probability

data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree	$O(\log n)$	$O(\log n)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n + cost(h))$	$O(1 + cost(h))$
open addressing	$O(1)$	$O(1)$

algo	**best**	**average**	**worst**	**stable?**	**memory**
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\checkmark$	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	$\times$	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\checkmark$	$O(1)$
merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	$\checkmark$	$O(n)$
quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	$\times$	$?$