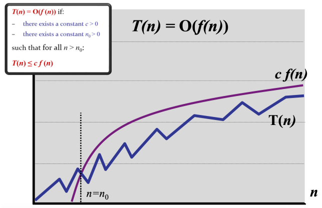
CS2040S
AY20/21 sem 2
by jovyntls

ORDERS OF GROWTH

definitions

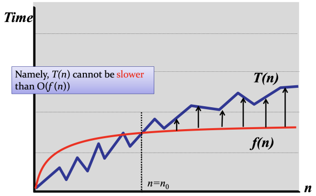
$$T(n) = O(f(n))$$

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



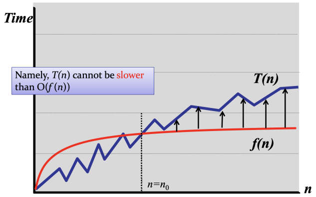
$$T(n) = \Omega(f(n))$$

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

- $O(n)$ - 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: $O(h) = O(\log n)$

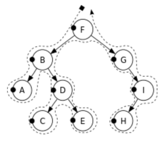
BST operations

- `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 >

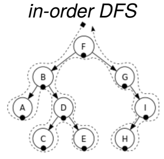
traversal

pre-order DFS



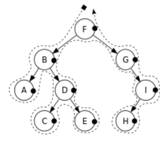
F-B-A-D-C-E-G-I-H

in-order DFS



A-B-C-D-E-F-G-H-I

post-order DFS



A-C-E-D-B-H-I-G-F

AVL Trees

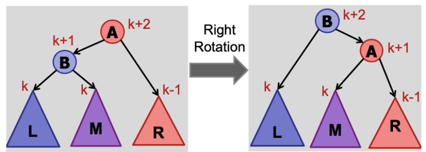
- **height-balanced**
  - $\iff$ `|v.left.height - v.right.height|` ≤ 1
- each node is augmented with its height - `v.height = h(v)`

rebalancing

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

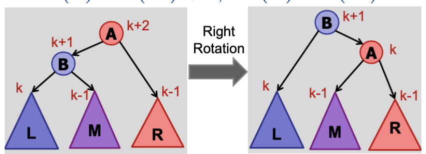
[case 1] B is **balanced: right-rotate**

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



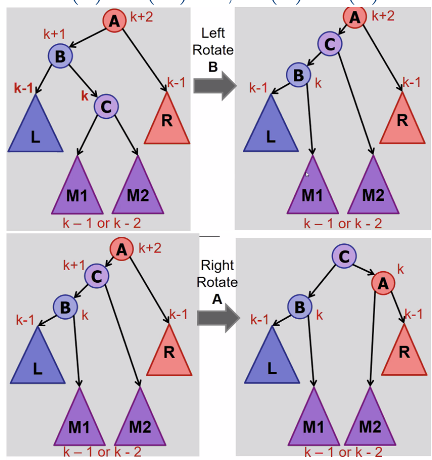
[case 2] B is **left-heavy: right-rotate**

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



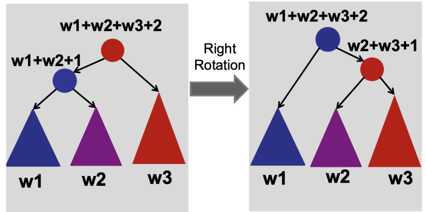
[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

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

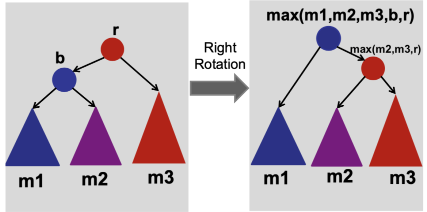


updating nodes after rotation

weights

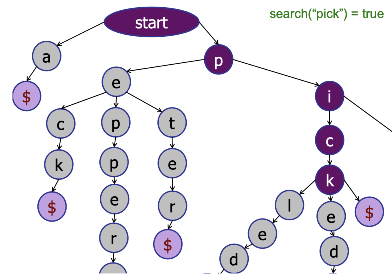


max



Trie

- `search, insert` - $O(L)$ (for string of length L)
- space: $O($size of text · overhead$)$



HASH TABLES

- very fast - faster than BST
- disadvantage: no successor/predecessor operation

hashing

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

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 the 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

- time complexity
  - `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, then there will be clusters of size $\Theta(\log n)$
  - expected cost of an operation, $E[\#probes] \leq \frac{1}{1-\alpha}$ (assume $\alpha < 1$ and uniform hashing)
- **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	$O(n^2)$	$O(n)$

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

---

sorting

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	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	?

sorting invariants

sort	**invariant** (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted
insertion	first k elements are in order
merge	–
quick	partition is in the right position

searching

search	average
linear	$O(n)$
binary	$O(\log n)$
quickSelect	$O(n)$

data structures (search/insert)

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