CS2040S Midterm Summary
AY22/23 Sem 2
github.com/gerteck
Adapted from github.com/jovyntls/cheatsheets/

Data Structures and Algorithms

- Algorithm: finite sequence of steps, unambiguous and precise, to accomplish some task.
- **Desirable Traits** of algorithms: fast, efficient, fair, small memory, parallel, modular, correct, secure. Compare trade-offs
- **Midterms Key Topics**: Asymptotic Notation, Simple Recurrences, Asymptomatic Analysis, Basic Probability.
- **Key Algorithms**: Binary search, Sorting, Balanced Binary Trees, Augmented Trees, Heaps.
- **Key Ideas**: Problem Solving: Identify Invariants (to understand and show how algorithm works), Trade-offs (Choosing), Augmentation of data structures.
- **Strategies:** Try something simple (Naive), Reduce and conquer (binary search), Divide and conquer (Mergesort), Maintaining invariant (AVL trees, keep sorted), Augment existing structure.

ORDERS OF GROWTH

- Notations: Big-O (Upper bounded by), Big-Ω (Lower Bounded by), Big-θ (Tight Bounded by - Grows at same rate)

Complexity Rules

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

SORTING

Consider the **Monotonic** Properties of the problem at hand, and the **Invariants** for each of the sorting algorithms.

- **BubbleSort**
  - 'Bubble' largest element rightmost. Compare adjacent items and swap. `Average: O(n^2)`
  - **Invariant:** largest last k items are sorted
- **SelectionSort**
  - Selects next smallest element, swaps it to the left sorted portion. `Average: O(n^2)`
  - **Invariant:** smallest leftmost k items sorted
- **InsertionSort**
  - Left to Right index, swaps element leftwards till it is smaller than next element. `Average: O(n^2)`
  - **Invariant:** first k items sorted
  - tends to be faster than the other $O(n^2)$ algorithms
- **MergeSort**
  - Split in half, mergeSort 1st half; mergeSort 2nd half; merge. `Deterministic : O(n log n)`
  - **Invariant:** subarray is sorted wishfully
- **HeapSort (Array Implementation)**
  - **(Heapify)** Creating a Heap: Naive insert: `O(n log n)` vs. Recursive divide and conquer. `O(n)`.
  - **Heap as Array:**
    - `left (x)` $= 2x + 1$, `right (x)` $= 2x + 2$
    - `parent (x)` $= \lfloor \frac{x-1}{2} \rfloor$
  - **HeapSort**: Repeated extract Max from heap, placing it at newly available last index. Time Complexity (log n) extract max, for n elements: `Deterministic : O(n log n)`
  - **Invariant:** Last k elements are sorted (Max)
  - Notes: Faster than mergesort, slightly slower than quicksort. In place algorithm. Not stable.
- **QuickSort**
  - partition algorithm: $O(n)$
  - stable quicksort: $O(\log n)$ space (alt implementation)
  - Steps: first element as partition. 2 pointers from left
    - left pointer moves until element ¿ pivot
    - right pointer moves until element ¡ pivot
    - swap elements until left = right.

  then swap partition in, 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 (in an array)
- **Mechanism**: Partition list according to random pivot (e.g. first element). If random pivot is in `index k − 1`, we have found the answer. Other wise, we do quick select on left / right partition if pivot index ¿ k - 1 or ¡ k - 1 respectively. Eventually, if array is size 1, that is answer.
- **Time Complexity**:
  Average: $O(n)$ assuming partition halves on average.
  Worst: $O(n^2)$, (partition n times).
  Average: `T(n) = T(n/2) + O(n) = O(n)`
  Worst: `T(n) = T(n—c) + O(n) = O(n^2)`
- **Invariant / Useful property** 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

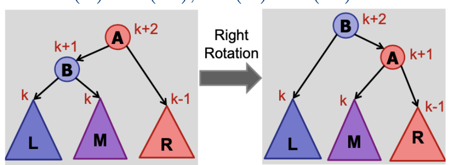
- `height, h(v) = max(h(v. left ), h(v. right ))`
  - leaf nodes: `h(v)` = 0
- modifying operations
  - `search`, `insert` - $O(h)$
  - `delete` - $O(h)$
    - case 1: no children - remove node
    - case 2: 1 child - remove node, connect parent to child
    - case 3: 2 children - delete successor; replace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

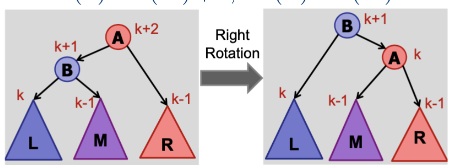
- Adelson-Velskii and Landis Tree $\to$ self-balancing tree.
- **height-balanced** (maintained with rotations)
  - $\iff$ |v.left.height - v.right.height| ≤ 1
- bBST where each node is augmented with its height - `v. height = h(v)`
- **Invariant**: BST is height balanced if every node is height balanced. Has at most height `height h < 2 log (n)`.
- space complexity: $O(LN)$ for N strings of length L

rebalancing

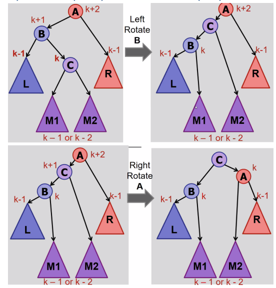
[case 1] B is **balanced: right-rotate**
$h(L) = h(M), \quad h(R) = h(M) - 1$



[case 2] B is **left-heavy: right-rotate**
$h(L) = h(M) + 1, \quad h(R) = h(M)$

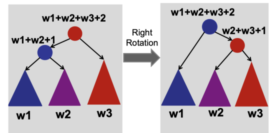


[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**
$h(L) = h(M) - 1, \quad h(R) = h(L)$

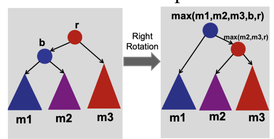


updating nodes after rotation

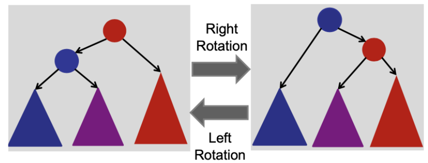
Order Statistics: weights



Interval Trees: max endpoint in subtree



AVL Tree Rotations:



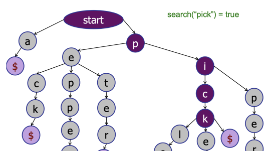
- insertion: max. 2 rotations
- deletion: recurse all the way up
- rotations can create every possible tree shape.

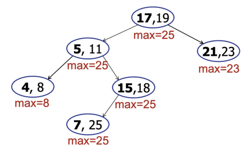
Trie

- `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

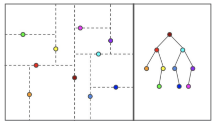




orthogonal range searching (bBST)

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

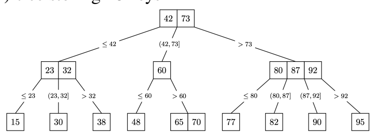
kd-Tree



- 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

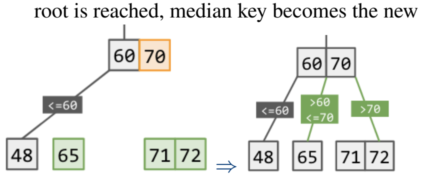
e.g. a (2, 4)-tree storing 18 keys



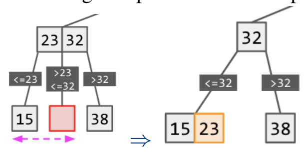
- **Rules:**
  1. (a, b)-child policy where $2 \leq a \leq (b+1)/2$

node type	# keys min	# keys max	# children min	# children 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

HEAPS

Heaps vs. AVL Trees as Priority Queues: Same asymptotic cost for operations. Heaps faster real cost, simpler (no rotations), slightly better concurrency.

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

- **Supported 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 larger key
  - `delete`: swap w bottomrightmost in subtree; bubble down
  - `extractMax`: Extract root then `delete(root)`, bubble down swapped last node towards larger key
- 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 arr to heap: $O(n)$ (bubble down, low to high)
  - heap to sorted arr: $O(n \log n)$ (extractMax, swap to back)

UNION-FIND

- Disjoint-Set Data Structure
- **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

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.

---

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	$O(L)$	$O(L)$
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

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