

CS 240 Midterm Review (Module 1–7)

Asymptotic Analysis

- Order notations
 - $f(n) \in O(g(n))$ if $\exists c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n) \forall n \geq n_0$
 - f “grows no faster than” g (\leq)
 - $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n) \forall n \geq n_0$
 - f “grows no slower than” g (\geq)
 - $f(n) \in \Theta(g(n))$ if $\exists c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \forall n \geq n_0$
 - f and g grow at the same rate ($=$)
 - $f(n) \in o(g(n))$ if $\forall c > 0, \exists n_0 > 0$ such that $0 \leq f(n) < cg(n) \forall n \geq n_0$
 - f is “*strictly* upper-bounded” by g ($<$)
 - $f(n) \in \omega(g(n))$ if $\forall c > 0, \exists n_0 > 0$ such that $0 \leq cg(n) < f(n) \forall n \geq n_0$
 - f is “*strictly* lower-bounded” by g ($>$)
 - Suppose $L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$
 - If $L = 0$ then $f \in o(g)$
 - If $0 < L < \infty$ then $f \in \Theta(g)$
 - If $L = \infty$ then $f \in \omega(g)$
 - If $f \in O(g)$ and $f \in \Omega(g)$, then $f \in \Theta(g)$
- Loop analysis
 - Begin from the innermost nested loop; use \sum for each outer loop
- Recurrence relations analysis
 - **In general**, $\{T(n) = T(n/2) + c\} \in \Theta(\log n)$

Let $n = 2^k$, i.e. $k = \log n$

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + c \\ &= T(2^{k-2}) + 2c \\ &= \dots \\ &= T(2^0) + kc \\ &= c + (\log n)c \\ &\in \log n \end{aligned}$$

Priority Queues and Heaps

- **Priority queue**: an **abstract data type** containing a collection of items each with a priority

- **Heap:** binary tree with 2 properties
 - Structural property: all levels of filled except the lowest, which is left-justified
 - Ordering property: the parent of any node has greater value than the node itself
- The height of a heap with n nodes is $\Theta(\log n)$
 - Since $2^k \leq n$ (# of nodes on all levels above) and $n \leq 2^{k+1} - 1$ (# of nodes including this level)
- **Bubble-up algorithm:** used for heap insertion
 - If $\text{node.key} > \text{node.parent.key}$ then swap
 - Brings a large value from a leaf node up
- **Bubble-down algorithm:** used for heap deletion
 - If $\text{node.key} < \text{node.largest_child.key}$ then swap
 - Brings a small value from the root node down
- Heapify with bubble-up: insert each item, total runtime = $\Theta(n \log n)$
- Heapify with bubble-down (given an unordered array): since leaf nodes can't bubble-down, start bubbling down from second-last level up ($n/2$ nodes)
 - Total runtime = $\Theta(n)$
- **Heapsort:** heap-insert n times (or just heaptify), then delete-max n times – always $\Theta(n \log n)$
 - Not stable

Sorting, Selection, Randomized Algorithms

- A sorting algorithm is stable if the order of equal (tied) keys are preserved (from the original order in the input)
- **Selection problem:** find the k -th largest element within n elements
- **Quickselect:**
 - Choose pivot = $\Theta(1)$
 - Partition:
 - Go from outermost pair inwards, swap any pairs that are in the wrong order ($\Theta(n)$)
 - Return index of pivot; array is now partitioned by the pivot value
 - Recursively call partition on one of the two partitions, until pivot index = desired index (like binary search)
 - Worst case: every recursive call partitions off 1 element = $\Theta(n^2)$
 - Best case: desired element is returned on first call = $\Theta(n)$
- **Quicksort:**
 - Same as quickselect, except recurse on both partitions instead of just one
 - Worst case = $\Theta(n^2)$

- Best case = average case = $\Theta(n \log n)$
- Not stable
- **Randomized algorithm:** algorithm whose output depends on the input as well as some random numbers
 - $T(I, R)$ = runtime given input I and set of random numbers R
 - **Expected runtime** = $T^{exp}(I) = \sum_R T(I, R) \times P(R)$
 - For uniform distribution, $P(R) = 1/\#$ possibilities generated by R
 - For quickselect and quicksort, randomizing the pivot makes the expected time = average time
 - **Monte Carlo algorithm:** always fast, not always correct
 - **Las Vegas algorithm:** always correct, not always fast
- **Comparison model:**
 - Data can only be accessed by:
 - Comparing two elements
 - Moving elements around
 - **Theorem:** any correct comparison-based sorting algorithm is $\Omega(n \log n)$
- **Countsort:** input of size n only contains numbers in a consecutive range of size k
 - Count the # of occurrences of each element (i.e. a histogram)
 - Place elements back into array in-order, based on their # of occurrences
 - $\in \Theta(n + k) \in \Theta(n)$ if $k \in O(n)$
 - Is stable
- **Radix sort:** represent all elements in base r (radix)
 - Sort by each digit (count sort), starting from most/least significant digit (m digits)
 - $\in \Theta(m(n + r)) \in \Theta(n)$ if $m, r \in O(n)$
 - MSD is not stable, LSD is stable

Balanced Search Trees

- **Binary search tree**
 - Left[right] subtree nodes $<[>]$ root node
 - Search, insert = $\Theta(\log n)$
 - Delete = $\Theta(\log n)$
 - If is leaf, just delete
 - If has one child, replace with child
 - If has two children, swap with *predecessor* or *successor* then delete

- Worse-case height = $\Theta(n)$
- **AVL tree**
 - **Balance** of each node = height(right subtree) – height(left subtree)
 - Height of empty subtree = -1
 - If $|\text{balance}| > 1$, tree is out of balance
 - **Right/left rotation:** when node balance = ± 2 and a child has balance = ± 1 of the same sign
 - **Double right/left rotation:** when node balance = ± 2 and a child has balance = ± 1 of the opposite sign
 - Rotations are $\Theta(1)$
 - Insert & delete normally as in BSTs, update balances from bottom up and rotate if any subtree is out of balance ($\Theta(\log n)$)
 - # of nodes in a subtree of a given height is at least = $N(h) = 1 + N(h-1) + N(h-2) \geq 2N(h-2)$
 - From this recurrence we get $N(h) \geq 2^{\lfloor h/2 \rfloor}$ or $h \in O(\log n)$
 - # of nodes is also at most $N(h) = 2^{h+1} - 1$, or $h \in \Omega(\log n)$
 - Therefore the height of an AVL tree is $\Theta(\log n)$

Dictionaries

- **Dictionary:** a collection of **key-value pairs** (KVP)
- **Optimal static ordering:** elements are stored in decreasing order by probability of access
 - Given L of n elements, expected access cost is $E(L) = \sum_{i=1}^n P(x_i)T(x_i)$
 - If P is uniform then $E(L) = \sum_{i=1}^n \frac{i}{n} \in \Theta(n)$
- **Dynamic ordering:**
 - **Move-to-front (MTF):** move searched item to the front of list
 - Transpose: swap searched item with item preceding it
- **Skip list:**
 - A series of lists $S_0 \dots S_h$ containing keys in increasing order
 - Each starts and ends special keys $-\infty$ and $+\infty$
 - Each level contains a subset of the level below; i.e. $S_h \subseteq S_{h-1} \subseteq \dots \subseteq S_0$
 - S_h only contains $-\infty$ and $+\infty$
 - Skip search: navigate down and to the right from the top-left
 - Peek ahead on current level; if went over target, descend one level
 - Height of an inserted element is randomly computed
 - e.g. height = k = # of heads flipped before a tail

- $P(\text{height} = k) = (1/2)^k$
- Expected # of nodes on level k is $\frac{n}{2^k}$
- Expected total # of nodes for C levels is $\sum_{k=0}^C \frac{n}{2^k} = 2n$ as $C \rightarrow \infty$
- Expected space = $O(n)$
- Expected height = $O(\log n)$
- Expected search, insert, delete = $O(\log n)$

Tries

- **(Binary) Trie/radix tree:** a bitwise binary tree
 - Left child = 0, right child = 1
 - A node is flagged if the binary string generated by the path from root to it is in the dictionary
 - Insert: search for node
 - If it exists, flag it
 - If not, extend from last matching node by creating new nodes
 - Delete: search for node
 - If it's not a leaf, unflag it
 - If it's a leaf, delete it and all ancestors until a flagged node or node with 2 children is reached
 - Search, insert, delete $\in \Theta(h) = \Theta(|x|)$ where $|x| = \#$ of bits
- **Compressed trie/Patricia trie**
 - Reduce each path through unflagged nodes with one child to a single edge
 - Each node stores the next index/digit to be tested
 - Time complexity same as uncompressed; space complexity is improved

Hashing

- **Theorem:** any comparison-based search on a size- n dictionary is $\Omega(\log n)$
- **Direct addressing:**
 - Each key k is integer $0 \leq k < M$ for some M
 - Each value v corresponding to k is stored at $A[k]$
 - Search, insert, delete $\in \Theta(1)$
 - Total storage $\in \Theta(n)$
- **Hash function** = $h : U \rightarrow \{0, 1, \dots, M-1\}$
 - Any key $k \in U$ is mapped to some index in an array of size M (a hash table)

- i.e. value v for key k is stored at $A[h(k)]$
- **Load factor** $= \alpha = \frac{n}{M}$
 - If load factor is too high/too low, increase/decrease M and rehash, i.e. recreate hash table ($\Theta(M + n)$)
- **Closed addressing**: each entry in the hash table can hold more than one KVP (bucket)
 - Can use unordered linked list (chaining)
 - Average bucket size/chain length $= \alpha$
 - Thus search has average-case $\Theta(1 + \alpha)$, worst-case $\Theta(n)$
- **Opening addressing**: if collision occurs, search linearly until a slot is available
 - $h(k, i) = h(k) + i \bmod M$
 - Must distinguish between empty and deleted slots
- **Double hashing**:
 - $h(k, i) = h_1(k) + h_2(k)i \bmod M$
- **Cuckoo hashing**: always insert new element at $h_1(k)$
 - If collision, kick out old element and re-insert at $h_2(k)$
 - Repeat for at most n times; rehash if exceeds n times
 - Any particular element is guaranteed to be at either $h_1(k)$ or $h_2(k)$

Multi-Dimensional Data

- **Range search**: return all elements within a certain range of values
 - Nodes are either *boundary*, *inside*, or *outside* based on the paths from root to the left & right boundary values
 - One-dimensional range search $= O(\log n + k)$ for k reported elements
- **Quad trees**: divide elements into 4 quadrants until each quadrant contains only 1 element
 - Each node has 4 children representing 4 quadrants
 - **Spread factor** $= \beta = \frac{d_{max}}{d_{min}}$, where $d_{max/min} = \text{max/min distance between 2 points}$
 - Height $= h \in \Theta(\log \beta)$
 - Range search $\in \Theta(nh)$
- **kd-trees**: split elements into 2 equal regions until each region contains only 1 element
 - Sort by x-coordinate; divide elements by median
 - Median is stored at root; elements with $x \leq [>]$ median go in left[right] subtree
 - Sort each subset by y-coordinate; repeat; alternate sorting between x- and y-coord
 - Height $= h \in \Theta(\log n)$
 - Range search $\in O(k + \sqrt{n})$

- **Range trees:**
 - Construct T w.r.t. x-coords
 - Every node in T has an associated tree T_{assoc} of its subtree in T w.r.t. to y-coords
 - Range search:
 - Perform 1D range search on T w.r.t. x-coords
 - For outside nodes, do nothing
 - For boundary nodes, check individually
 - For inside nodes that are immediate children of boundary nodes, perform 1D range search on their T_{assoc} w.r.t. y-coords
 - Time complexity $\in O(k + \log^2 n)$
 - Space complexity $\in O(n \log n)$

String Matching

-