CSE 373

NOVEMBER 1ST – EXAM REVIEW

EXAM FRIDAY

- **6-8 questions**
  - Q1 is short answer, but will have many parts. It may be best to save this for last
  - Q2 is algorithm analysis
  - AVL
  - Hash tables
  - Priority Queues/Heaps

EXAM FRIDAY

- **Topics**
  - Definitions
  - Stacks and Queues
  - Runtime Analysis
  - Dictionaries
  - BSTs
  - Traversals
  - AVL Trees
  - Hash Tables
  - Memory Hierarchy
  - B+-trees
  - Priority Queues
  - Heaps

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior

DEFINITIONS

- **Important terms**
  - Abstract Data Type
    - Example: Dictionary
      - Supports functions: insert, find, delete
      - Has expected behavior
  - Data Structure
    - Language independent structure which implements an ADT
      - Example: AVL tree
      - Can be analyzed asymptotically

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific

DEFINITIONS

- **Important terms**
  - Implementation
    - Low-level design decisions
    - Language specific
- **Example**
  - The Queue ADT supports enqueue, dequeue and front.
  - Arrays and Linked Lists are examples of the data structures
  - Implementation: front and back pointers

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order

STACKS AND QUEUES

- **Our first two ADTs**
  - Stack:
    - Supports: push(), pop(), top()
    - LIFO order
  - Queue:
    - Supports: enqueue(), dequeue(), front()
    - FIFO order

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation
    - Resizing time

STACKS AND QUEUES

- **Data structure choices**
  - Arrays and Linked Lists
  - Considerations
    - Memory usage
    - Ease of implementation
    - Resizing time
  - Runtimes:
    - O(1) for all functions

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed

RUNTIME ANALYSIS

- **Counting the number of operations**
  - Comparisons, mathematical operations, assignments
- **For loops and while statements**
  - Count the number of times relevant code is executed
- **Important summations**
  - Sum of all numbers from 1 to n
  - Sum of the powers of two

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change

RUNTIME ANALYSIS

- **Asymptotic Analysis**
  - Best-case, worst-case, average-case
  - Usually we discuss worst-case complexity
  - If we increase the input size, how does the computation time change
- **BigO notation**
  - Upper bound for a given function
  - f(n) = O(g(n) if there exists a c and n_0 for which f(n) $\leq$ c*g(n) for all n $\geq n_0$

RUNTIME ANALYSIS

- **Recurrences**
  - Analysis of recursive functions
  - Break the function into recursive and non-recursive
  - Produce the recurrence relation
  - Roll out the recurrence or produce the recurrence tree
  - Find the closed form of the recurrence
  - Upper bound this recurrence with a bigO bound.

RUNTIME ANALYSIS

- **Amortized analysis**
  - Used when an expensive operation occurs with predictable frequency (e.g. resizing an array)
  - Describe the state of the data structure
  - Indicate the number of operations
  - Determine how many are the costly operation and how many are the cheap operations
  - # of costly * costly runtime + # cheap * cheap runtime
  - Divide by the number of operations

RUNTIME ANALYSIS

- **Memory analysis**
  - Calculating how much memory an algorithm needs
  - This is in addition to the data itself
  - Think about any secondary data structures you might use
  - Also, remember that recursive functions consume memory on the call stack

RUNTIME ANALYSIS

- **Basic ideas**
  - If we increase the size of the input by one, how does our total computation change?

RUNTIME ANALYSIS

- **Basic ideas**
  - O(1): Input size has no effect on runtime
  - O(log n): doubling the input increases the runtime by some constant amount
  - O(n): linear time, each additional input increases execution time by a constant amount
  - $O(n^2)$: doubling the input increases the runtime by a factor of 4.
  - $O(2^n)$: exponential, increasing the input by one doublies the runtime

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)

DICTIONARIES

- **ADT**
  - Supports the following functions
    - Insert(key k, value v)
    - find(key k)
    - delete(key k)
  - Data is stored in key, value pairs
  - In this course, duplicate keys are not allowed
  - Most data structures can implement a dictionary

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent

BINARY SEARCH TREES

- **Binary trees**
- **Nodes with two children**
- **Maintains search property**
  - All values in the left subtree must be less than the parent
  - All values in the right subtree must be greater than the parent
- **With each increase in height, the number of nodes in a tree roughly doubles**
- **A perfect tree has 2^{h+1}-1 nodes**
- **Roughly half of a binary search tree are leaves**

TRAVERSALS

- **Two main traversal families**
  - Depth First Search
  - Breadth First Search

TRAVERSALS

- **Two main traversal families**
  - Depth First Search
  - Breadth First Search
- **DFS**
  - Usually implemented recursively
  - Whether the parent is processed before, after or in the middle of its children determines if the traversal is pre-order, post-order or in-order respectively

TRAVERSALS

- **Two main traversal families**
  - Depth First Search
  - Breadth First Search
- **DFS**
  - Usually implemented recursively
  - Whether the parent is processed before, after or in the middle of its children determines if the traversal is pre-order, post-order or in-order respectively
- **BFS**
  - Put the root into a queue
  - Dequeue a node, process it and enqueue its children
  - Top to bottom left to right traversal
  - Queue is largest at the widest part of the tree

AVL TREES

- Specific type of binary search tree
- Still must implement binary search
- Nodes in AVL trees have two extra fields, height and balance
- Balance = | height(left) – height(right) |
- Balance for each node must be less than or equal to 1
- Trees with this condition still have O(log n) height
- No covering delete in this course
- Find: O(log n): Insert O(log n)

AVL ROTATIONS

- **AVL Rotations occur when an insertion makes a node out of balance**
  - Relative to the node that is unbalanced, there are four rotations depending on which grandchild received the new node.
  - Left-left and right right rotations involve the child of the affected node being rotated up into position
  - Left-right and right-left rotations involve the grandchild being rotated up into position. The grandparent and parent become the two children
  - It is important that these rotations preserve BST property

HASH TABLES

- **A large data set M with a smaller set that should be saved, D**
- **A hash function maps M onto D**
  - It should run in O(1) time
  - It should distribute into all of the available spots evenly
- **Hashtables provide O(1) runtime IF**
  - Collisions are not a problem
  - Decrease the chance of collisions by increasing the amount of memory (**load factor**)
    - Resizing is costly

COLLISIONS

- **Probing**
  - Linear probing
    - Try the appropriate hash table row first
    - Increase the index by one until a spot is found
    - Guaranteed to find a spot if it is available
    - If the array is too full, its operations reach O(n) time. **Primary clustering**

COLLISIONS

- **Probing**
  - Quadratic Probing
    - Rather than increasing by one each time, we increase by the squares
    - k+1, k+4, k+9, k+16, k+25
    - Certain tables can cause **secondary clustering**
    - Can fail to insert if the table is over half full

COLLISIONS

- **Probing**
  - Secondary Hashing
    - If two keys collide in the hash table, then a secondary hash indicates the probing size
    - Need to be careful, possible for infinite loops with a very empty array
    - If the secondary hash value and the table size are coprime (they share no factors), then secondary hashing will succeed if there is an open space
    - If table size is prime, only need to check if hash is a multiple

COLLISIONS

- **Chaining**
  - Rather than probing for an open position, we could just save multiple objects in the same position
  - Some data structure is necessary here
  - Commonly: a linked list, AVL tree or secondary hash table.
  - Resizing isn't **necessary**, but if you don't, you will get O(n) runtime.

MEMORY HIERARCHY

- **Memory is not uniformly accessible**
  - OS manages access to computer resources
  - Some memory is on disk and some is in cache
  - Dictated by two types of behavior
    - Spatial locality – Items near each other are moved together (memory pages)
    - Temporal locality – memory used recently will be used again

B-TREES

- **To reduce disk accesses we introduce the B-tree**
  - Two types of nodes
    - Signposts: Have M pointers and M-1 keys
    - Leaves: Have L <K,V> Pairs and a pointer to the next leaf
  - Signposts must have at least M/2 pointers and leaves must have at least L/2 data points, unless it is the root
  - Keys in signposts are the smallest item in the next pointer

B-TREES

- **Insertion**
  - Find the leaf that should hold the inserted element
  - Insert the new k,v pair in sorted order in the leaf node
  - If it overflows (i.e. the leaf is full when inserted)
    - Split the leaf into two nodes and add the new leaf to the parent
  - If the signpost overflows, split the signpost into two signposts and try to add the new signpost to the parent
  - Split back up to the root and create a new root if necessary

HEAPS

- **Priority Queue ADT**
  - Supports: insert(), findMin(), deleteMin(), changePriority()
  - Data is stored in priority, value pairs
  - In this class, we use the min-heap, where a lower value means it should dequeue first

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically
  - Runtimes
    - Insert: O(log n), findMin: O(1), deleteMin O(log n)
    - ChangePriority: O(log n)

HEAPS

- **Data Structure**
  - Heap
    - Complete binary tree
    - Heap property
  - Implementation
    - Array
    - Find parents/children arithmetically
  - Runtimes
    - Insert: O(log n), findMin: O(1), deleteMin O(log n)
    - ChangePriority: O(log n)
    - buildHeap, O(n)

HEAPS

- **Percolate up**
  - After you've inserted an element in the next location in order to preserve completeness
  - Compare the current element against its parent
  - Swap if the child is less than the parent
  - Repeat until the child is greater than the parent or the new element is swapped up to the root

HEAPS

- **Percolate down**
  - After deleting an element, move the last element (from completeness) up to the root
  - Compare the current node against both of its children
  - Swap the node with the smaller child provided the child is still smaller than the parent
  - Continue until the node is smaller than both children, or it is a leaf.

HEAPS

- **Floyd's method**
  - For each element in the array from size/2 to the first element
  - Percolate that element down as much as necessary
  - Because most elements are near the bottom, they do not need to percolate down very far, this results in O(n) overall runtime

GOOD LUCK!

- Practice Exam solution tomorrow
- Review in section tomorrow
- Email/Piazza any questions
- No office hours Friday or next Monday
- Grades back in class on Monday