CSC190 Notes

Aman Bhargava

January-April 2019

Contents

0.1 Introduction and Course Information

- Professor: John Mathai
- Course: Engineering Science
- Term: 2019 Spring

Chapter 1

Efficiency and Complexity

There are two things that effect efficiency of a computer algorithm.

- Space Complexity = cost in terms of memory
- Time Complexity = cost in terms of steps

1.1 Ideal vs. Real Computers

Ideal	Real
Automaton + Workbook	Microprocessor + Memory
Mathematical Algorithms	Programs (Sea of Functions)
Abstract Data Types	Data Structure

`Note: ADTs are Abstract Data Types`

Stack is used by C for temporary variables made by functions. Heap is used by CPU and by malloc. All stuff pushed to the stack by a function is freed when the function exits.

1.2 The RAM Model of Computation

Stands for Random Access Machine Model. Each simple operation takes 1 time step. Each memory access takes 1 time step. It's actually a really good models for how computers perform.

1.2.1 Big Oh Notation

Usually used for worst case scenarios.

- $O(g(n))$ is for upper bound of time complexity.

- $\Omega(g(n))$ is for lower bound of time complexity.
- $\Theta(g(n))$ is for upper AND lower bound (separated by only a constant)

Dominance Classes

One class of algorithms in big Oh notation is said to dominate another if their time complexity grows faster than the other.

1. Linear
2. Logarithmic
3. Linear
4. Super Linear ($nlog(n)$)
5. Quadratic
6. Cubic
7. Exponential
8. Factorial

Chapter 2

List ADT implementation in C

- An array with length L and end index 'end'

2.1 Stacks and Queues

2.1.1 Stacks

- First in, last out
- push and pop

2.1.2 Queues

- First in, first out
- enqueue, dequeue

2.1.3 Stack and Heap in C

Chapter 3

Trees

3.1 Types of General Traversal:

Breadth-First Traversal:

In which you read (1) right to left (2) top to bottom.

Depth-First Traversal:

In which you read (1) top to bottom (2) right to left

3.1.1 Using Stacks and Queues for Traversals

Remember: Stacks → Depth-First. Queues → Breadth-First.

How does it work? On inspection, you would think that recursion is the only way to effectively traverse a tree in a depth/breadth-first manner in an easy way. However, you can employ stacks and queues to do this iteritavely. Here are the steps:

1. Add the root node to the stack, queue per the above statement.
2. Pop/dequeue the dequeueing point and print it. Add its children to the stack, queue
3. Repeat step 2 until finished.

This isn't immediately intuitive, but it holds true. Try this on a tree on paper to convince yourself! It's a really cool property of trees/stacks/queues :)

3.2 Binary Trees:

3.2.1 Mathai's Weird Definition for Binary Trees vs. Normal Trees

The rule: The first child of a node goes as the left child in the binary tree. All of its siblings are added as RIGHT children of that first child. Etc.

3.2.2 Binary Search Trees:

Think about the binary search algorithm. The rule is simple: Start with the first node. If the thing you want to add is greater than that node, you add it to the RIGHT child tree. Otherwise, you add it to the LEFT child tree. The rule is recursive and super elegant.

3.2.3 Advantages/Disadvantages of Binary Search Trees:

1. Advantage: Will (generally) change search from O(n) to O(log(n)) because you only need to visit every LEVEL instead of every NODE (#levels = log(#nodes))
2. Disadvantage: If it's not a balanced tree, you could end up with a BST that has #levels = #nodes

3.2.4 Binary Tree Traversals

1. Pre-Order: NLR
2. Post-Order: LRN
3. In-Order: LNR

Mnemonic: The 'pre'/'post'/'in' refers to the placement of the NODE between/in front of/after the L and R node.

3.3 Self-Balancing BST's (AVL Trees):

The general way we balance binary search trees is with **rotations**

Level of imbalance: calculated as nR (number of levels on the right side of the node) minus nL.

Printing Binary Trees in Order: When you print the left subtree first, then the node, then the right subtree (recursively)

Types of Rotation:

1. Left
2. Right
3. Major Left / Minor Right
4. Major Right / Minor Left

3.3.1 Right Rotation:

The left child becomes the head. The left child's parent (the root) becomes the right child, and the left child of the left child becomes maintains its relationship with the left child of the root. The original left child's RIGHT child becomes the left child of the original root.

When to do this? whenever nL - nR ≥ 2

3.3.2 Left Rotation:

The right child becomes the head. The right child's parent (the root) becomes the left child, and the right vhild of the right child maintains its relationship with the right child of the original root. The original right child's LEFT child becomes the right child of the original root.

3.3.3 Major Right / Minor Left:

When to do this: Whenever the balance on the root and the balance on a subtree is opposite, and the absolute balance of the root is greater than or equal to 2. Tl;dr: You want the subtrees to have the same SIGN of balance as the root node before you do big switches. Condition for doing compelex switches is: balance (root) ¿=2, sign balance (sub-node) doesn't equal the sign balance (root node)

3.3.4 Summary of Tree Rotations:

1. For easy, child-free lines to the right and left (root balance = +-2, child banace = +-1 with same sign), use the corresponding rotation to make them triangle trees.

2. For bent ones with root balance = +-2 and child balance +-1 of opposite sign), use minor rotation to get back to case 1 and do a major rotation on the root.

Figure 3.1: Summary of All Rotation Types

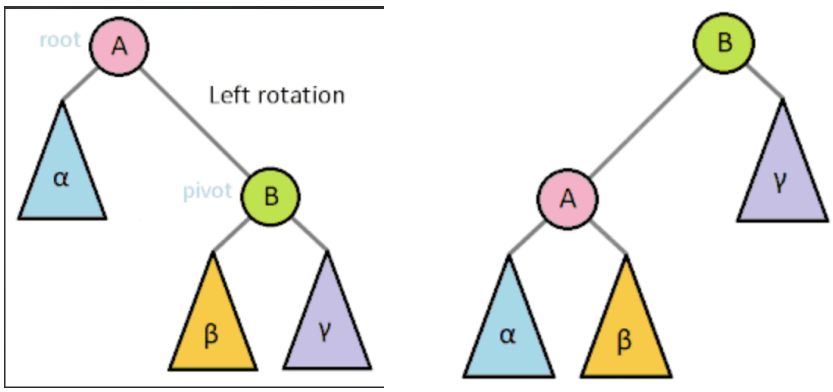
3.3.5 Inserting to a Binary Tree

1. If we are passed a pointer to a NULL pointer, then add this info as the NULL pointer.

2. If we are passed a node:

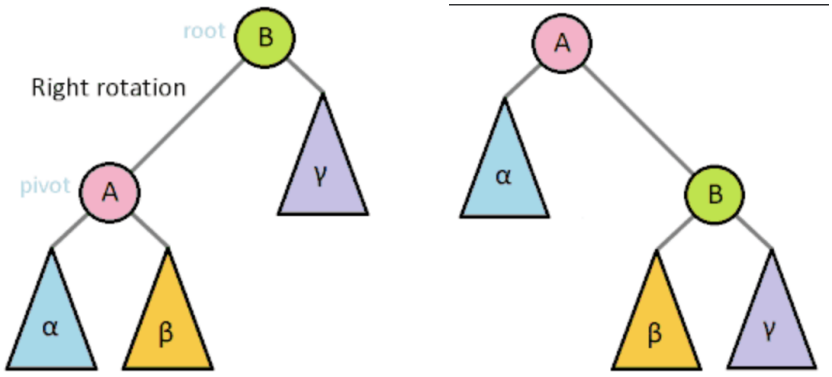
3. Add to the left subtree if we are smaller than that particular node.

4. Add to the right subtree otherwise.



(a) label 1 (b) label 2

Figure 3.2: Visual Representation of a Left Rotation



(a) label 1 (b) label 2

Figure 3.3: Visual Representation of a Right Rotation

3.3.6 Deleting from a Binary Tree

Swap the node you need to delete with the left child's furthest right child. Then delete the initial node. Also works with the right child's firthest left descendant. Think about how repeated elements might change this up.

3.4 AVL Tree

AVL trees are self-balancing BST's.

3.4.1 When to use which rotation/HOW TO BALANCE:

Lines of two and zigzags of two are the worst case scenario and they're all you'll ever really have to deal with. Always do rotations on things that have +/- 2.

3.4.2 Adding to AVL Trees:

Do binary insert. Then balance it.

3.4.3 Deleting from AVL Trees:

Do binary deletion. Then balance it.

3.5 Priority Queues

Operations:

1. Insert element
2. Find minimum
3. Find maximum

The queue can be an ordered list, unordered list, binary search tree, and more.

3.6 Hashes and Strings

Hashes = way to maintain a dictionary. Takes advantage of $O(1)$ complexity of finding something once you know its key (location) in a list/array.

Example: indexing strings. Hash function uses each letter as a number in a base-alphabet number system. Then you map any string to unimaginably large numbers. To find where they should go in an array, take the output % the size of the array.

Problem: you're likely to have some collisions in terms of where you assigned a string. Let m be the length of the array.

Solution 1: Chaining where you have linked lists for each hash table entry that has multiple items hashing to it. But this uses a lot of memory for pointers.

Solution 2: Open Addressiong where you just chuck it in the next sequentially available slot. It's not great though because repeated deletion and addition may result in an entirely random hash table.

3.7 Heap

Max heap = top heavy. Min heap = bottom heavy.

3.7.1 Implementation as List (0-indexed)

1. Child 1 Index = Parent Index*2 + 1
2. Parent Index = (Child Index-1) // 2

3.7.2 Adding to a Heap

1. Add to the last element in the heap/list
2. If parent and child relationship is OK, then stop.
3. Otherwise, swap and go back to step 2.

3.7.3 Deleting from Heap

1. Replace deleted item with the item furthest down and to the right in the heap.
2. Heapify by swapping with parent if necessary or with child if necessary.

3.7.4 Searching a Heap

You have to search everything. But you can stop if you reach something larger (in the case of a min heap) or smaller (in the case of a max heap).

Chapter 4

Graphs

What's a graph? Bunch of nodes with connections (verticies and edge). Each connection/edge can have a weight. Connections/edges can either be directional or non-directional. The path length between two is the sum of the weights of the edges that you traverse to get there.

$$G = (V, E)$$

Graph is made of a set of verticies (V) connected by set of edges (E).

What about notation? $E_{a \to b}$ connects a to b. If it's non-directional, then $E_{b \to a}$ is therefore valid and existent. However, if it is a directional edge, then it does not necessary exist.

4.1 Types of Graphs

1. Weighted and unweighted graphs (verticies can have weight)
2. Directed vs. undirected graphs

4.1.1 Paths

A set of connected edges and nodes that leads from one node to another.

4.1.2 Cycles

A path that leads back to where it started without using any edges twice. Must use an edge.

4.1.3 Strength of Connections

If $A \rightarrow B$ in a non-directional way, it is a strong connection. However, if $A \rightarrow B$ is directional, it is considerred weak. Also, if it is indirectly connected, it doesn't affect the strength of the connection.

4.1.4 Articulation Vertex

A vertex that, if removed, will change the connectivity of other verticies.

4.2 Traversal of Graph: Prim's Algorithm

What it does: finds a minimum spanning tree. It's a greedy algorithm (i.e. it takes the locally optimal path at each point). It always converges to an optimal solution, but that's rare for greedy algorithms.

Steps in Prim's Algorithm:

1. Start with a vertex chosen at random.
2. Choose the minimally-weighted edge that's connected to that (set of) vertex.
3. Add the vertex connected by that minimally-weighted edge to the minimum spanning tree.
4. Now repeat steps 2-3 with the new set of vertices until you get the minimum spanning tree.

Topological Ordering in a Graph All about finding an "order" to a graph (e.g. pre-requisite courses represented with directed edges).

You can only find topological ordering for some graphs. They must be **directed** and **acyclic** (i.e. **DAG** graphs).

For the courses example:

- If you don't need any pre-requisite (i.e. no traversal from start is needed), it has *in-degree* of 0.

4.3 Representations of Graphs: Adjacency Matrices

- $n \times n$ matrix
- If $A + ij$ is non-zero value a then vertex i is connected to j with weight a

4.4 Dijkstra's Algorithm: Pathfinding

Here are the steps to run this algorithm by hand:

1. Start with a set of visited nodes, unvisited nodes, and a set of tentative weights all set to infinity (except for the starting node)
2. At each step, see what the minimum distance from each visited node to its neighbour is. If the total distance from the starting node is less than the tentative weight currently says, update the weight.
3. Continue updating the tentative weights and the set of visited/unvisited nodes, along with the paths needed to get to each node.

Complexity of the Algorithm Time complexity: $O(n^2)$ (worst case, discussed in textbook)

Chapter 5

Bit Manipulation

Have you ever wanted to be a cool computer person who does things with ones and zero's instead of actual letters and numbers like a normal person? If so, this is the right chapter for you!

5.1 Converting to and from Different Bases

Base 10, 2, and 16 are most commonly used. Base 16 is just a way to read base 2 in a more efficient manner. In order to work with bits it's pretty important to know how to convert back and forth because the test is all on paper.

5.1.1 Converting from base 10 $\rightarrow$ base 2

You keep dividing by two, keeping track of the remainder. Eventually the number you will be trying to divide by two will be 1. You keep going until it's zero + remainder(1). Then you read the reaminders upward from that final 1.

5.1.2 Converting from base 2 $\rightarrow$ base 16

Any hex number can be expressed as 4 binary digits. Make a correspondence table between quadruplets of binary numbers and hex (1-f, inclusive). To convert to base 16 subdivide from right to left in groups of four binary digits. Pad the leftmost part with leading zeros and convert using the table.

5.1.3 Converting from base 10 → base 16 (and vice versa)

Just go through base 2 fam.

5.2 Bitwise Operators

5.2.1 How to print bits of a variable

- printf("%x\n", var1);
- You usually use unsigned ints and unsigned chars.
- Unsigned chars are 8 bits (two hex characters, 1 byte).
- Unsigned ints are 32 bits (8 hex characters, 4 bytes).
- Each hex character represents half a byte.
- When you print out a low-valued int, it won't necessarily show all 8 hex characters (it eliminates leading zeros).

5.2.2 Altering Bits

1. & - And operator; Only outputs 1 when both inputs at that bit are 1.
2. | - Or operator; Outputs 1 if at least one of the inputs at that bit are 1.
3. ˆ- Exclusive or operator (xor); Outputs one if only one fo the inputs is 1.
4. ˜- Not operator; flips the bits (e.g. a = opposite of a)
5. << - shift left operator; shifts all bits left, padding the right in with zeros (e.g. $c = a << 2$;)
6. >> - shift right operator; shifts all bits right, padding left with zeros (e.g. $c = a >> 2$;)

Chapter 6

Sorting and Searching Algorithms

Sorting is pretty darn important. You can probably reason out why that is. Here are some of the major sorting algorithms, and below is a table summarizing the time complexity of these algorithms:

Algorithm	**Time Complexity**		
	Best	*Average*	*Worst*
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(nlog(n))$
Merge Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(nlog(n))$
Quick Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

6.1 Selection Sort

1. Define start of array as $s = 0$ and $e = len(array)$
2. Find minimum element in array $s \rightarrow e$ and swap it with the element at s
3. Add 1 to s and repeat step 2 until $s = e$.

6.2 Insertion Sort

1. The "sorted" partition of the array starts with one element.

2. You then take the next element after the "sorted" partition and insert it into the sorted partition in order.

3. Repeat until the sorted partition is the size of the array.

6.3 Bubble Sort

1. Go through element 0 to element len(array)-1. If the element is not in order with the element in front of it, switch them.

2. Keep doing this until swapping is no longer necessary.

6.4 Heap Sort

1. Make a heap out of the array

2. Keep popping the max/min element from the heap and re-heapifying until the heap is empty.

3. You now have a sorted list.

6.5 Merge Sort

1. Have a function called merge that 'zips' two sorted lists together into one big sorted list (pretty simple if you think about it)

2. Unless the given partition is of length 2 or 1, keep calling the mergesort function on each half of the given partition.

3. If the partition is of length 2, swap them if they're out of order. Otherwise, return the partition.

4. If the partition is of length 1, return the partition.

5. Zip up the two return values from the recursive call of the mergesort algorithm.

6.6 Quick Sort

6.6.1 Partition

This takes a list and 2 inputs including the begin and end index of the section.

1. The end element is the 'pivot'.
2. You set your 'first high' to the beginning element
3. You go through with a counter i
4. If the thing at 'i' is smaller than the pivot, you swap it with whatever is at 'first high' and add one to 'first high'.
5. Repeat this, then swap the element at 'pivot' with the element at 'first high'
6. Return the index of 'first high'. You just partitioned the list into 2 parts around the pivot.

6.6.2 Quick Sort

1. Just keep running the partition algorithm recursively, fam.

6.7 Radix Sort

1. Make k bins where k is equal to the number of potential place valuese (i.e. 10 in base 10)
2. Start by binning based on the lowest place value.
3. Repeat the process with the second lowest place value and so on.
4. Continue until everything is sorted.
5. Runs in $O(nk)$ time where k is the maximum number of place values.