CSC190 Notes

Aman Bhargva

January-April 2019

Contents

0.1 Introduction and Course Information

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`

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

Chapter 3

Trees

3.1 Types of Traversal:

Level Order Traversal:

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

Breadth Order Traversal:

In which you put stuff on the same level of the tree at the same indentation level, and the children of nodes are indented underneath it.

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

3.3 Self-Balancing BST's:

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 $\geq$ 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.

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.

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.

Figure 3.1: Summary of All Rotation Types

3.4 AVL Tree

AVL trees are self-balancing BST's.

3.4.1 When to use which rotation/HOW TO BALANCE:

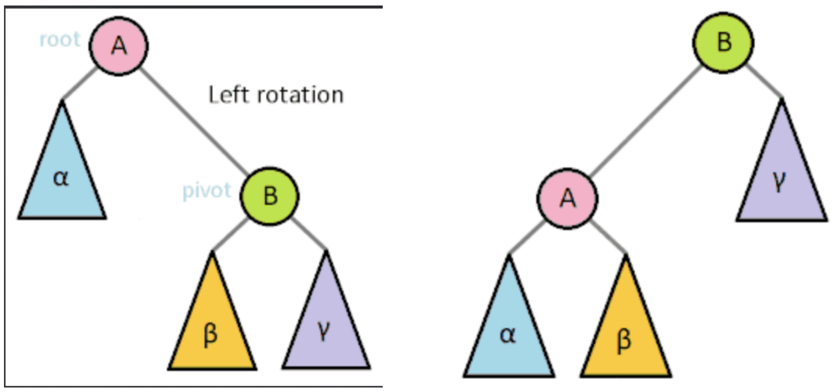
Lines of two and zigzags of two are the worst case scenario and they're all you'll ever really have to deal with.

3.4.2 Adding to AVL Trees:

Do binary insert. Then balance it.

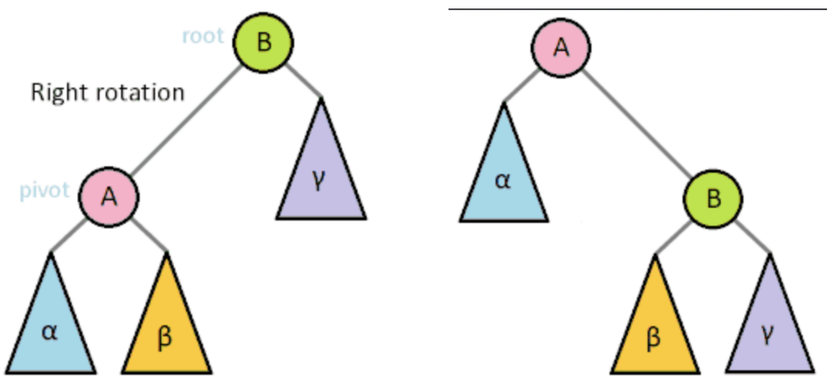
3.4.3 Deleting from AVL Trees:

Do binary deletion. Then balance it.



(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

3.5 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 $O(n)$ 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

1. Child Index = Parent Index...
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).