

# 308-431 Data Structures & Algorithms

Course: 308-431 Data Structures & Algorithms  
Professor: Prof. Newborn  
Email: newborn@cs.mcgill.ca  
Web Site: none  
Text: Data Structures & Algorithm Analysis in C, 2<sup>nd</sup> Edition, by Mark Allen Weiss

Notes By: Laurence Yudkovitch  
Email: [lyudko@ugrad.ece.mcgill.ca](mailto:lyudko@ugrad.ece.mcgill.ca)  
Web Site: <http://www.openface.ca/~yudko>

## Chapter 1

### PROOF BY INDUCTION

1. Start with the base case (usually  $n=0$  or  $n=1$ ), and prove that your theorem is correct for it.
2. Next, state the equation you are trying to prove as the inductive hypothesis.
3. Now show that your theorem is true for  $n + 1$ . Usually, this involves expanding the equation to include the "n" term and the "n + 1" term separately. The n term of course, can be written in terms of the inductive hypothesis. After this is shown, the prove his complete.
4. Example: Fibonacci numbers,  $F_0=1, F_1=1, F_n = F_{n-1} + F_{n-2}$

### PROOF BY CONTRADICTION

1. Start by assuming the theorem is false.
2. Use this assumption to prove that some other fact known to be true is false. Because we know this other fact is true, our assumption which proves it false cannot be true. This completes the proof.
3. Example: Prime Numbers. Fact-there are in infinite number of prime numbers, and every number is either prime for a product of primes.

### RECURSION

1. A recursive function is defined in terms of itself. Factorial  $F(x) = x * F(x-1)$ .
2. Recursive functions must have a base case where the answer is known.  $F(0) = 1$ .
3. Each recursive call must progress towards the base case.

## Chapter 2 - Algorithm Analysis

### ORDERS OF MAGNITUDE

1. Constant,  $\log N$ ,  $\log^2 N$ ,  $N$ ,  $N * \log N$ ,  $N^2$ ,  $N^3$ ,  $2^N$  (Exponential)

2.  $T(N) = O(f)$  if  $T(N) \leq c \cdot f$  when  $N \geq n$ .
3. Keep in mind that the constant can be freely selected, and the relationship need only hold for values of  $N$  larger than a specific value. I.e. place more weight on results shown for larger values of  $N$ .
4. To have logarithmic running time, you need to reduce the size of the problem by a constant value (e.g. half) on each iteration.

### EUCLID'S ALGORITHM FOR GCD

1. GCD is the greatest common divisor.
2. To find  $\text{GCD}(M, N)$ , write:
3.  $M = k \cdot N + r_1$ , where  $k$  is the largest integer multiple of  $N$  such that  $kN < M$ , and the remainder  $r = M - kN$
4.  $N = k \cdot r_1 + r_2$
5.  $r_1 = k \cdot r_2 + r_3$
6. Continue in this fashion until the remainder is zero. The last remainder before this is the greatest common divisor.
7. E.g.:

$$\begin{aligned} \text{GCD}(1738, 3634) &= 158 \\ 1738 &= 0 \cdot 3634 + 1738 \\ 3634 &= 2 \cdot 1738 + 158 \\ 1738 &= 11 \cdot 158 + 0 \end{aligned}$$

### EXPONENTIATION

1. Raising an integer to a large integer power.  $X^p$
2. This is a recursive algorithm, with two base cases.
3. If  $p=0$ , return 1
4. If  $p=1$ , return  $X$
5. If  $p$  is even, then return  $\text{Pow}(X \cdot X, p/2)$
6. If  $p$  is odd, then return  $\text{Pow}(X \cdot X, p/2) \cdot X$
7. Ex:

$$\begin{aligned} X^{51} &= (X^{25})^2 \cdot X \\ &= ((X^{12})^2 \cdot X)^2 \cdot X \\ &= (((X^6)^2)^2 \cdot X)^2 \cdot X \\ &= (((X^3)^2)^2 \cdot X)^2 \cdot X \\ &= (((X \cdot X) \cdot X)^2)^2 \cdot X \end{aligned}$$

Which is very large for  $X=5$

## Chapter 3

### LINKED LISTS

1. Each cell has its own data, and pointer to the next cell in the list.
2. In general, the list can only be read from start to finish. Doubly linked lists can change fact

3. Insertions are generally performed at the front of the list  $O(1)$ .
4. Searches require accessing approximately half the list  $O(N)$ .
5. Deletions require searching the list, and minimal overhead above that.

## STACKS

1. Access to the stack is only permitted at the top.
2. Push is equivalent to insert.
3. Pop deletes the top elements.
4. Top checks to see what the top elements is.
5. Stacks can be useful in balancing symbols.

## QUEUES

1. Queues are like lines; first in, first-out.
2. The Enqueue operation inserts an element at the end of the list.
3. The Dequeue operation returns and deletes the element at the front of the list.
4. A circular queue wraps around.

# Chapter 4

## TREES

1. A tree is a collection of nodes.
2. A node consists of records (data, or keys) and pointers to sub-nodes.
3. The first (or top) node is called the root.
4. Nodes and sub-nodes have a parent - child relationship (siblings, aunts, grand-parents...).
5. The number of children a node has is the fan-out.
6. A node with no children is called a leaf node.
7. A tree with  $N$  nodes has  $N-1$  edges (because the root has no edge, but every other node does.)
8. The depth of a node is the length of the path (number of nodes between) from the root to the node. (I.e. going down)
  - a) The root is at depth 0.
9. The height of a node is length of the longest path from the node to a leaf.
  - a) All leaves are at height 0.

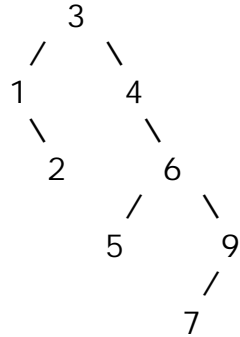
## Tree traversal

1. Pre-Order: root, left, right. (Root pre children)
2. Post order: left, right, root. (Root post children)
3. In order: left, node, right.

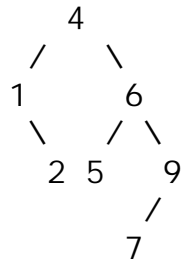
## BINARY TREES

1. Each node has at most two children.
2. The average depth is on the order of  $\sqrt{N}$ .

3. For the binary search tree, the average depth is on the order of  $\log N$ .
4. Fact requisite property for a binary search tree is that for every node in the tree, the values in the left subtree are all less than the value of the node, and the values in the right subtree are all greater than the value at the node.
5. Note that the minimum and maximum values in the tree can be found easily by descending all the way to the bottom left or right of the tree.
6. Result of inserting 3,1,4,6,9,2,5,7



7. When deleting a node with two children, the node to be deleted is replaced by the smallest value in the right subtree. This node is recursively deleted.
8. Deleting the root

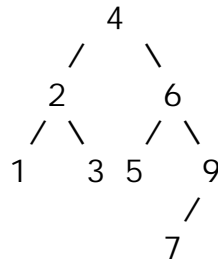


9. Lazy deletion involves leaving the node in and only marking it as deleted.

## AVL TREES

1. Adelson-Velskii and Landis
2. A binary search tree with a balance condition: for every node in the tree, the height of the left and right subtrees can differ by at most one.
3. Note that the height of an empty tree is defined to be -1.
4. To maintain in the tree property, insertions may require rotations. The rotation is performed at the node where the imbalance is located.
5. Single rotation is performed when the insertion is for the left left subtree, or the right right subtree. This involves making the middle node in the new "root" node.
6. Insertion to the left right, or right left subtrees require a double rotation. This involves moving the bottom node up to the root, making the former root the new left child and keeping the middle node as the right child. (Example for a left insertion on the right subtree.)

7. Inserting 2,1,4,5,9,3,6,7



## SPLAY TREES

1. A binary tree with a guarantee that any  $M$  operations take at most  $O(M \log N)$  time.
2. Each time the node is accessed, it is rotated up to the root.
3. A zig-zag operation is similar to a double rotation.
4. A zig-zig operation resembles switching the direction of the tree.
5. Deleting a node involves:
  - a) Access it to bring it to the top of the heap.
  - b) Remove it, resulting in a left and right tree ( $T_L$  and  $T_R$ ).
  - c) Find the largest node in  $T_L$ , and rotate it to the top.
  - d) Being the largest element, it will not have a right child, so  $T_R$  can be assigned as its right subtree.

## Chapter 5

### HASHING

1. Usually the hash table is a fixed array containing keys.
2. The array index starts at 0.
3. The keys are mapped into the hash table using the hashing function. The typical hash function is  $K \bmod TableSize$ .
4. For best results, the table size should be a prime number.

### COLLISION RESOLUTION

1. This determines how to handle different keys hashing to the same location.

#### Separate Chaining

1. The hash table holds pointers to the start of linked lists. Each key belonging to the cell is added onto the end of the chain.
2. Disadvantage of requiring dynamic memory allocation and pointers - time consuming.

#### Open Addressing

1. Involves using multiple hash functions if collisions result.
  - a)  $h_i(X) = (Hash(X) + F(i)) \bmod TableSize$
  - b)  $F(0) = 0$ .

- c) You start with your base hash function, and then add the result of the conflict resolution function for each successive iteration (until it fits) to the base hash function.
2. Linear Probing
  - a)  $F(i) = i$ , so you just keep going higher...
  - b) Results in primary clustering
3. Quadratic Probing
  - a)  $F(i) = i^2$
  - b) No primary clustering, but secondary clustering may result.
  - c) Guaranteed to find a free location if the table is at least half empty.
4. Double Hashing
  - a)  $F(i) = i * hash_2(x)$
  - b) Takes a long time to compute second hash function, and if  $hash_2$  is not selected well, you can end up with no free slots.
  - c) If implemented well, can be as good as a random distribution.
5. Question 5.5 in the text:  $F(i) = r_i$  where  $r_i$  are random integers between 1 and  $N$ , and each number appears only once.
  - a) Prove that under this strategy, if the table is not full, then the collision can always be resolved. *This resolution strategy is the same as linear probing, only the numbers are in random order. Since the resolution function is added to the base function, you will effectively step through each cell in the table (although not in order).*
  - b) Would this strategy be expected to eliminate clustering? *It should have a similar effect on clustering as quadratic probing does - eliminate primary clustering, but make secondary clustering possible.*

## Chapter 6

### PRIORITY QUEUES (HEAPS)

1. The Insert operation allows adding items to the queue/heap.
2. The DeleteMin operation finds and removes the minimum element in the queue/heap.

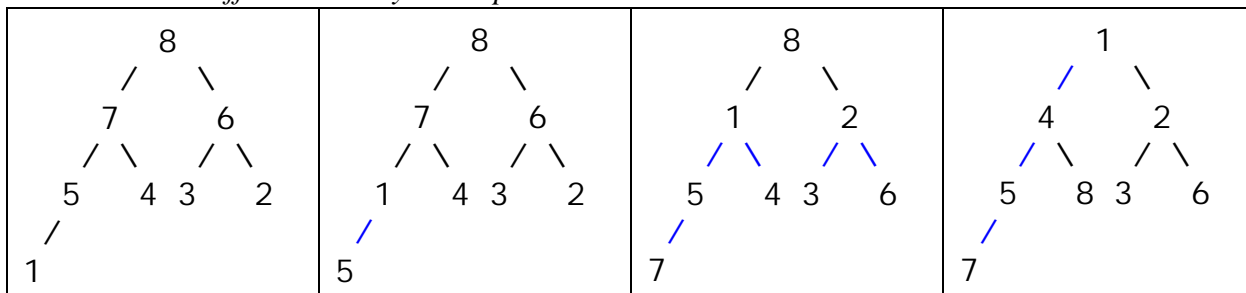
### BINARY HEAP

1. A heap is a binary tree, with two properties.
2. Structure property: the binary tree is completely filled except for possibly the bottom level which is filled from left to right.
  - a) Therefore the height is  $O(\log N)$
  - b) It can be stored in an array, without pointers. The left child of node  $i$  is in location  $2i$ , and the right child in  $2i + 1$ . Parent is in  $\lfloor i/2 \rfloor$ .
3. Order property: The smallest element must be at the root. Each subtree is considered a heap as well, so all descendants of a node must be of lesser value than the node.
4. Inserting X:

- a) Create a “hole” in the next available position in the tree, and place X there.
  - b) While X is less than its parent, swap them. This is called *percolating up*.
5. DeleteMin X:
- a) Replace the root node with the last element in the tree (i.e. the rightmost element at the last level). This has effectively deleted the root, and decreased the size of the tree by one.
  - b) Now make sure that the order property is maintained. While the node is larger than any of its children, swap it with its smallest child.
    - ▶ The first node is the root.
    - ▶ This is called *percolating down*.

### Build Heap

1. Two methods to build a heap, assuming the N inputs are being passed all together.
2. Method 1: N successive calls to the insert function.
  - a) Average running time is  $O(N)$ , but worst case is  $O(N \log N)$ .
3. Method 2: Place the N items into the tree in any order, and for  $(i = N/2; i > 0; -i)$  PercolateDown(i)
  - a) This has linear average running time:  $O(N)$ .
  - b) The percolate down calls essentially go through each non-leaf node and ensure that the order property is maintained for each subtree, starting at the subtrees and working up to the big tree.
4. Question 6.7 in the text:
  - a) Prove that for binary heaps, Build Heap does at most  $2N-2$  comparisons between elements. *Theorem 6.1: For the perfect binary tree of height h containing  $2^{h+1}-1$  nodes, the sum of the heights of the nodes is  $2^{h+1}-1-(h+1)$ . For a tree with  $N=2^h$  nodes, we have  $2N-1-h-1 < 2n-2$ . For a tree with  $N=2^{h+1}$  (because complete trees will fall between the two ranges), we have  $N-1-h-1 < 2n-2$ .*
  - b) Show that a heap of eight elements can be constructed in eight comparisons between heap elements. *Interesting, I seem to be getting otherwise. Create a heap with the elements 8, 7, 6, 5, 4, 3, 2, 1 and check. As shown below, this requires 9 comparisons in the worst case. If we build a heap of only 7 elements, then we can suffice with only 8 comparisons.*



## Chapter 7