1 Induction

To demonstrate the difference between structural and mathematical induction, consider the following claim which we will prove using both techniques:

Prove that the number of leaves (denoted $L(t)$) in a perfect binary tree t is equal to 2^h, where h is the height of the tree.

Proof by structural induction:

Let $P(t)$: the number of leaves in a perfect binary tree t is 2^h, where h is the height of the tree. ($L(t) = 2^h$)

base case: t a single node. Then t's height is 0. The single node of t is a leaf node. Therefore $L(t) = 1 = 2^0 = 2^h$, so $P(t)$ is true.

induction hypothesis: Suppose $P(t1)$ and $P(t2)$ is true for perfect binary trees $t1$ and $t2$.

inductive step: Suppose we have a tree t consisting of a root node r and two perfect binary subtrees $t1$ and $t2$, both of height $h1$. Then our tree t has height $h = 1 + h1$ by our definition of tree. By our induction hypothesis, we know that the number of leaves in $t1$ is 2^{h1} and the number of leaves in $t2$ is also 2^{h1}. And so the number of leaves in t is just the number of leaves in $t1$ + number of leaves in $t2$:

$$\begin{aligned} L(t) &= L(t1) + L(t2) \\ &= 2^{h1} + 2^{h1} \\ &= 2 \cdot 2^{h1} \\ &= 2^{h1+1} \\ &= 2^h \end{aligned}$$

Proof by mathematical induction:

Let $P(h)$: the number of leaves in a perfect binary tree t of height h is 2^h

base case: tree of height 0. Then there is one leaf node, and so $L(t) = 1 = 2^0 = 2^h$, so $P(0)$ is true.

induction hypothesis: Suppose $P(h1)$ holds for trees of height $h1$.

inductive step: Suppose we have a perfect binary tree of height $h = h1 + 1$. Then this tree has two subtrees (a left and right), call them $t1$ and $t2$. Note that since t is a full binary tree, both $t1$ and $t2$ have the same height, $h1$. Then our tree t has height $h = h1 + 1$. From our induction hypothesis, we know that the number of leaves in $t1$ is 2^{h1} and the number of leaves in $t2$ is also 2^{h1}. And so the number of leaves in t is just the number of leaves in $t1$ + number of leaves in $t2$:

$$\begin{aligned} L(t) &= L(t1) + L(t2) \\ &= 2^{h1} + 2^{h1} \\ &= 2 \cdot 2^{h1} \\ &= 2^{h1+1} \\ &= 2^h \end{aligned}$$

2 Algorithm Analysis

- We are mainly concerned with 4 aspects of an algorithm: correctness, speed/space, ease of programming, and suitability for the application.
- we quantify how good an algorithm is so that we can compare algorithms, and become a millionaire (speaking of which, how's that P=NP problem comin' for ya?)

2.1 Analysis Techniques

1. Big-Oh (Asymptotic Notation):

   - method of measuring running time as a function of program input size n.
   - gives a general estimate for how long an algorithm is going to take.
   - counts the fundamental operations, ignores additive terms and constant factors
   - formally: $g \in O(f)$ if there exists a constant $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

   The following table summarizes the $O(\cdot)$ hierarchy and gives you the names to refer to each 'class' of functions:

complexity	name
$O(1)$	constant complexity
$O(\log n)$	logarithmic complexity
$O(n)$	linear complexity
$O(n \log n)$	$n \log n$ complexity
$O(n^c)$, $c > 1$ const	polynomial complexity
$O(c^n)$, $c > 1$ const	exponential complexity
$O(n!) = O(n^n)$	BAD complexity

2. Best/Worst Case:

   - best case: min of inputs I of size n { time taken by algorithm A on input I}
   - worst case: max of inputs I of size n { time taken by algorithm A on input I}
   - average case: given a probability distribution on inputs,

$$\sum_{\text{inputs I of size n}} Pr[\text{input } I] \times (\text{no. operations used by A on } I)$$

3. Recurrence Relations:

   - used to find the running time of recursive algorithms
   - the first step is determining the recurrence relation, i.e. $C(n) = 1 + C(n/2)$.
   - the next step is solving for $C(n)$ using the iteration method.

3 Program Correctness

- precondition
- postcondition
- loop invariant

4 Abstract Data Types

- An abstract data type is a set of elements and a finite number of well-defined operations (methods) on these elements.
- The specification of an ADT does not have any design or implementation information associated with it.

4.1 Advantages of Abstract Data Types

1. creates natural modular decompositions (chunks)
2. separates the design of programs from the implementation of data structures
3. allows the programmer to delay final implementation decisions

elements	operations	complexity	comments
Binary Tree	leftChild(N) sibling(N) depth(N) isLeaf(N) height(N)	O(1) $O(1)$ $O(\log n)$ $O(1)$ $O(\log n)$	
Binary Search Tree	insert(B, v) remove(B, v) find(B, v) findMin(B) findMax(B)	$O(\log n)$ $O(\log n)$ $O(\log n)$ $O(\log n)$ $O(\log n)$	
Stack (LIFO)	push(I) pop() find()	$O(1)$ $O(1)$ $O(n)$	all access restricted to most-recently-inserted elements fast to get top item independent of total num of items arbitrary access (such as find) is bad
Queue (FIFO)	queue(N) dequeue() find()	$O(1)$ $O(1)$ $O(n)$	all access restricted to least-recently-inserted elements fast to get last item, independent of total num of items arbitrary access is bad
Linked List			stores 'nodes' not contiguously iteration is OK finding item bad
Hashtable	insert(I) remove(I) find(I)	$O(1)$ $O(1)$ $O(1)$	good for dynamic search based only on name fast operations (access time not dependent on num items) need a hash function
Heap			
Priority Queue	findMin() deleteMin() insert()	$O(1)$ $O()$ $O()$	want to access & remove smallest item in collection

Terminology

Tree A tree is either empty, or a root node r with zero or more nonempty subtrees, each of whose roots are connected by an edge from the root r.

Parent & Child Every node c except the root is connected by an edge from exactly one other node p. p is c's parent, and c is p's child.

Leaf Tree nodes that have no children are called leaves.

Internal Nodes An internal node is any tree node that is not a leaf.

Depth The depth of a node in a tree is the number of edges you must traverse to from the root to that node.

Height The height of a tree is the maximum depth of all nodes.

Length The length of a path between n and m is the number of edges in the path.

Tree facts

- in a tree, there is a unique path between any two nodes
- a tree is connected
- every tree with n nodes has $n-1$ edges

5 Questions

1. Given this predecessor vector for a graph on 7 vertices (labelled 0 through 6), reconstruct the tree it represents:

   ```
   null | 0 | 1 | 4 | 1 | 4 | 4 |
   ```

2. Do a preorder, postorder, and inorder traversal of the tree you constructed in the previous question.

3. Prove that the unweighted shortest path algorithm works, ie that it finds the shortest path from s to all other nodes.

4. Draw the following graph:

   $$\begin{aligned} V(G) &= \{v_1, v_2, v_3, v_4, v_5\} \\ E(G) &= \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\} \end{aligned}$$

   (a) What vertices are adjacent to v_4?

   (b) What edges are incident to v_3?

   (c) Name 3 cycles in the graph.

   (d) What is the adjacency matrix for the above graph?

   (e) What is the adjacency list for each vertex in the above graph?

   (f) What is the degree of each vertex?

5. What's the difference between a graph and a digraph?

6. What's the difference between a digraph and a network?

7. Prove that $\log(\log n) \in O(\log n)$.

8. Prove that $\log(n^{\log n}) \in O(n^2)$.

9. Prove that $a^n \in O(n!)$ for $a > 1$.

10. Put the following in order using $\subset, =$

    $$(\log n)^2 \qquad \log n \qquad \log(n^{\log n}) \qquad n^n \qquad n^3 \qquad \log n^3 \qquad \log n^n$$

6 Suggestions, If You Want 'Em

- Studying this package is not enough! You have to look over the notes, the examples, and some code, too!
- Try answering some questions *on paper*. Your test will be on paper, so you should practice being able to write stuff down without looking at your notes. For example, try to write down some definitions from memory.
- Try writing out sample Java code... you know there will be some on there. Practice writing a class. Practice putting in fields, methods, and constructors. Practice using `extends` and the modifiers `public`, `protected`, and `private` (yuck!). Practice writing recursive methods for anything you can think of. Practice implementing an interface. Write it out!
- Read over your notes often. Don't just read the facts, but read the examples and explanations, too. They may help you remember the facts later!
- Look over your assignments and try to understand what you did, and why you got it right or wrong. Just because you aced the assignment doesn't necessarily mean you still remember what you were doing when you wrote the assignment, unfortunately.
- Relax, don't stress, and remember that you're a smart person! Remember that you can reason your way through questions. And if you don't understand what a question is asking then put up your hand!