Section 6.3: Decision Trees

March 25, 2025

Abstract

Decision trees are defined, and some examples given (almost every tree will be binary in what follows). Binary search trees store data conveniently for searching later. Some bounds on worst case scenarios for searching and sorting are obtained.

1 Decision Tree Definition

Definition: a **decision tree** is a tree in which

- internal nodes represent actions,
- arcs represent outcomes of an action, and
- leaves represent final outcomes.

2 Examples of decision trees in action

- A decision tree for trees: the Identification of Common Trees of Iowa
- One MAT385 student made a local one: Common Trees of Campbell County

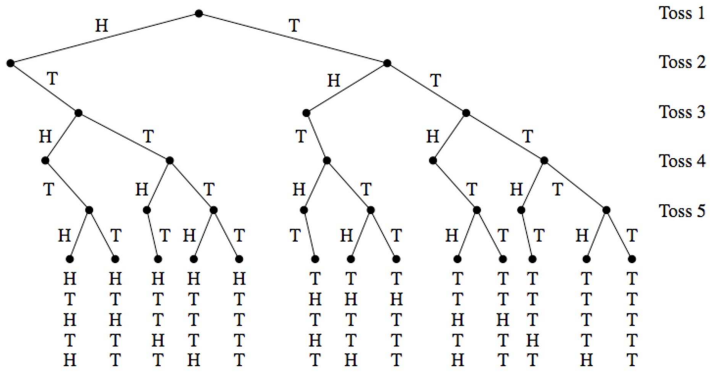


Figure 1: Figure 6.51, p. 529: Results of tossing a coin 5 times, no two heads in a row (binary decision tree). One of my students once made an interesting observation: how many nodes are there at each depth?

This is a model of the Fibonacci rabbit problem, where no two successive heads means that H represents an immature rabbit pair, and it must mature first (turn into a T). A T produces both an H – immature pair – and continues. Internal nodes hence represent different states of pairs in preparation for development, and arcs represent the different outcomes (or processes) – maturation, reproduction, or persistence.

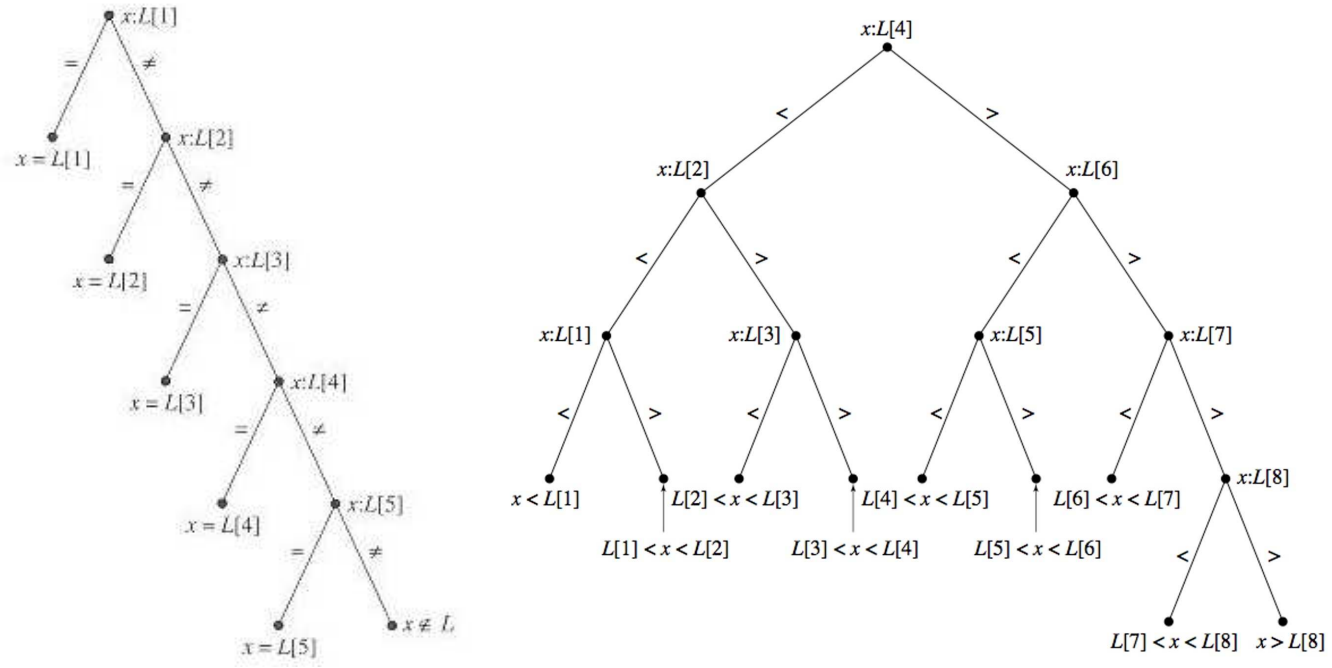


Figure 2: Figure 6.52, p. 530: Sequential Search on 5 elements (binary tree); Figure 6.53, p. 531: Binary Search on a sorted list (ternary tree, although it appears binary since those leaves corresponding to equality have been suppressed). Notice how clumsy this binary tree looks, since a power of two was used (8 elements), rather than one less than a power of two ($7{=}2^3 - 1$), which would have resulted in a **full** binary tree.

3 Lower Bounds on Searching

Table 1: Adding one more node to a full binary tree (the most efficient way of storing those nodes) bumps the depth up 1, so that if there are 2^d nodes, the depth is (at least) d. Hence, in the case of powers of 2, $d = \log m$.

Depth d	Nodes m	Nodes by depth
0	$1 = 2^1 - 1$	1
1	$3 = 2^2 - 1$	1+2
2	$7 = 2^3 - 1$	1+2+4
3	$15 = 2^4 - 1$	1+2+4+8
$\vdots$	$\vdots$	$\vdots$
d	$2^{d+1} - 1$	$1 + 2 + \ldots + 2^d$

In particular, here are some properties about binary trees:

a. Any binary tree of depth d has at most $m \leq 2^{d+1} - 1$ nodes. (Proof: look at the full binary tree, Table (1), as it has the most nodes per depth.)

b. Any binary tree with m nodes has depth $d \geq \lfloor \log_2 m \rfloor$, where $\lfloor x \rfloor$ is the **floor** function (meaning the greatest integer less than or equal to x). Again, the proof can be motivated simply by studying the full binary tree situation. However, a more formal proof is by contradiction and interesting (p. 532):

Proof: (Any binary tree with m nodes has depth $d \geq \lfloor \log_2(m) \rfloor$.)

- Assume $d < \lfloor \log_2(m) \rfloor$: then $d \leq \lfloor \log_2(m) \rfloor - 1$.
- From property a above,

$$m \leq 2^{d+1} - 1 \leq 2^{\lfloor \log_2(m) \rfloor - 1 + 1} - 1 \leq 2^{\log_2(m)} - 1 = m - 1$$

Therefore, since $m \leq m - 1$ is a contradiction, $d \geq \lfloor \log_2(m) \rfloor$.

These facts lead to the following

Theorem (on the lower bound for searching): Any algorithm that solves the search problem for an m-element list by comparing the target element x to the list items must do at least $\lfloor \log_2(m) \rfloor + 1$ comparisons in the worst case (the depth of the tree).

Since a general searching algorithm must offer the possibility of comparing x to each element of the list, one must have at least m comparisons (hence m internal nodes in the search tree).

The "+1" comes about because a decision tree representing the search problem has leaves which report the outcome of the search: hence its depth – which actually reports the number of comparisons in the worst case – is 1 more than the depth of a tree containing only the internal nodes (representing the comparisons themselves). So the result we've used ($d \geq \lfloor \log_2(m) \rfloor$) refers to the comparison tree, and we tack on 1 to give the actual decision tree.

For example, at depth 0 we make the first comparison: the depth of the last internal node is actually 1 less than the number of comparisons we make, which is given by the depth of the tree (including its leaves).

If, in its worst case, an algorithm does at most this lower bound on worst case behavior is an **optimal algorithm** in its worst-case behavior. Binary search is optimal (as seen, for example, in Practice 24).

Example: Practice #25, p. 532:

PRACTICE 25

a. Draw the decision tree for the binary search algorithm on a sorted list of five elements.
b. Find the depth of the tree and compare to $1 + \lfloor \log 5 \rfloor$. ■

4 Binary Search Tree

The Binary search algorithm required a sorted list; if your data is unsorted (it may be changing dynamically in time, if you're updating a database of customers, for example), you can populate a tree which approximates a sorted list, and then use a modified search algorithm (**binary tree search**) to search the list. A **binary search tree** is constructed as follows:

- The first item in the list is the root;
- Successive items are inserted by comparing them to existing nodes, from the root node: if less than a node, descend to the left child and iterate; if greater than, descend to the right child.
- If, in descending, there is no child, you create a new node.

For example,

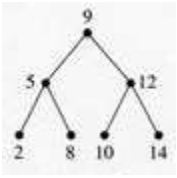


Figure 3: Figure 6.55: tree obtained from entering 9, 12, 10, 5, 8, 2, 14, in that order.

Example: Practice #26, p. 535. Construct the tree obtained by entering the same data but in the order 12, 9, 14, 5, 10, 8, 2. What is the depth of the tree? (Here is the construction process if the elements are entered in the order 5, 8, 2, 12, 10, 14, and 9.)

The binary tree search algorithm works in the same way as you'd introduce a new node, only the algorithm terminates if

- the element is equal to a node, or
- the element is unequal to a leaf of the binary search tree.

In this case the binary search tree serves as the trunk of the decision tree for the binary tree search algorithm (minus the leaves).

Example: Exercise #9, p. 537.

9. a. For a set of six data items, what is the minimum worst-case number of comparisons a search algorithm must perform?
   b. Given the set of data items $\{a, d, g, i, k, s\}$, find an order in which to enter the data so that the corresponding binary search tree has the minimum depth.

What's the worst way to enter the data into a binary search tree, if one is seeking to create a balanced tree?

5 Sorting

Examine Figure 6.56, p. 535:

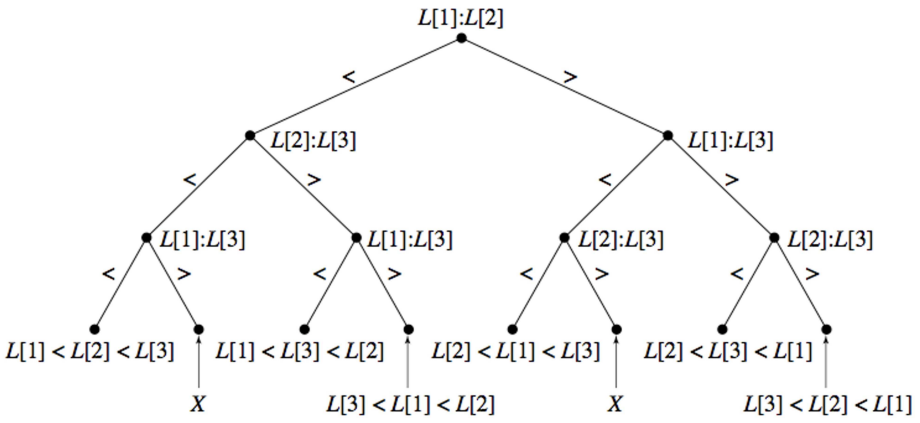


Figure 4: Figure 6.56, p. 535: Sorting a list (binary tree, provided distinct list elements)

In this case, we're sorting a three-element list using a decision tree. The author says this algorithm is "not particularly astute": why?

Assuming no equal elements in the list, then this is indeed a binary (rather than ternary tree, with = included). In this case, we can also get a lower bound on sorting a list with n elements:

- There are $n!$ possible sorted lists, and there must be at least that many leaves p ($p \geq n!$). (In Figure 6.56, there are eight leaves, but only 6=3! different sorted lists).
- A worst-case final outcome in the decision tree is given by the depth d of the tree (with the number of comparisons being equal to the depth).
- Since the tree is binary, $p \leq 2^d$ (the maximum number of leaves possible at depth d).
- Taking logs, we get $\log_2(p) \leq d$, or $d \geq \lceil \log_2(p) \rceil$, where $\lceil x \rceil$ is the **ceiling** function, which yields the smallest integer greater than or equal to x.
- Hence, $d \geq \lceil \log_2(n!) \rceil$.

This is the Theorem on the Lower Bound for Sorting: that you have to go to at least a depth of $\lceil \log_2(n!) \rceil$ in the worst case. Exercise #23, p. 539, shows that this lower bound ($\lceil \log_2(n!) \rceil$) is on the order of $n \log_2(n)$ (as we discovered for mergesort).

It's easy to show that $\log_2(n!) \leq n \log_2(n)$:

$$\log_2(n!) = \sum_{i=1}^{n} \log_2(i) \leq \sum_{i=1}^{n} \log_2(n) = n \log_2(n)$$

It's not quite so easy to show that $\exists c > 0$ such that $cn \log_2(n) \leq \log_2(n!)$....

$$\begin{aligned}
\log(n!) &= \log[n \cdot (n-1) \cdots 2 \cdot 1] \\
&= \log(n) + \log(n-1) + \ldots + \log(2) + \log(1) \\
&\geq \log(n) + \log(n-1) + \ldots + \log(\lceil \tfrac{n}{2} \rceil) \\
&\geq \log(\lceil \tfrac{n}{2} \rceil) + \log(\lceil \tfrac{n}{2} \rceil) + \ldots + \log(\lceil \tfrac{n}{2} \rceil) \\
&\geq (\lceil \tfrac{n}{2} \rceil) \log(\lceil \tfrac{n}{2} \rceil) \\
&\geq \tfrac{n}{2} \log(\tfrac{n}{2}) = \tfrac{n}{2}(\log(n) - \log(2)) \\
&\geq \tfrac{n}{2}(\log(n) - 1) = \tfrac{n}{4} \log(n) + \tfrac{n}{4} \log(n) - \tfrac{n}{2} = \tfrac{n}{4} \log(n) + \tfrac{n}{4}(\log(n) - 2) \\
&\geq \tfrac{n}{4} \log(n) \text{ (when } n \geq 4)
\end{aligned}$$

Example: Exercise #15, p. 538 (using a ternary tree)

15. One of five coins is counterfeit and is lighter than the other four. The problem is to identify the counterfeit coin.
    a. What is the number of final outcomes (the number of leaves in the decision tree)?
    b. Find a lower bound on the number of comparisons required to solve this problem in the worst case.
    c. Devise an algorithm that meets this lower bound (draw its decision tree).

The solution boils down to this: how many comparisons can one cram into a **ternary** tree of a given depth? (What if there were nine coins?)

6 Catalan Numbers

Problem: Given $n \in \mathbb{N}$, find C_n – the number of structurally unique binary search trees that store values 1 through n. (For convenience I defined $C_0 = 1$.) Figure 5 shows that $C_3 = 5$:

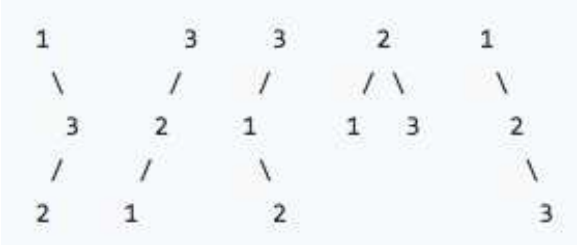
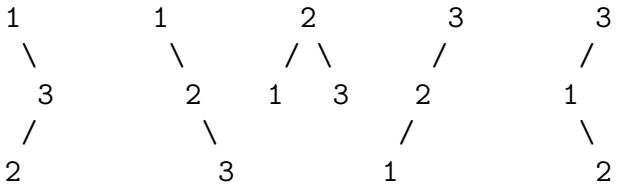


Figure 5: I don't know if the person who created this figure was being disingenuous, or just careless – but it might have helped to emphasize the symmetry this problem possesses. From this website.

Using the same quaint graphing technique, but emphasizing the symmetry, we see that there are five possible BSTs for $n = 3$:



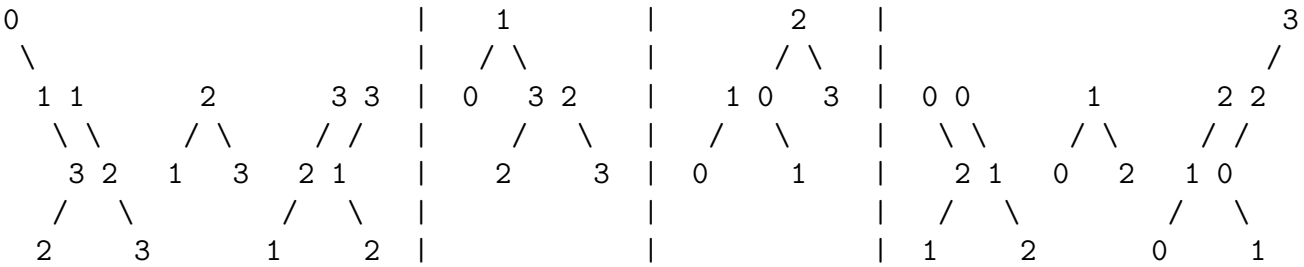
Or maybe this representation is more suggestive (with the $n = 2$ and $n = 1$ cases thrown in for good measure):

Maybe then it is becomes clearer that

$$C_3 = C_0 * C_2 + C_1 * C_1 + C_2 * C_0$$

which I think of as (nothin' to the left) * (two to the right) + (one to the left)*(one to the right) + (two to the left)*(nothin' to the right): so we multiply the number of ways of arranging things to the left times the number of ways of arranging things to the right, and so it goes....

For C_n we step from starting elements 1 to n, with successively more numbers of elements to the left, and successively fewer elements to the right; the fractal nature of the Catalan numbers appearing in successive figures (it's perhaps easier to see the pattern if we add a "0" element for this next case):



$$C_4 = C_0 * C_3 + C_1 * C_2 + C_2 * C_1 + C_3 * C_0$$

After some considerations like these, of the recursive nature of the problem, I wrote down the recurrence relation. Only later did I discover that Catalan numbers are defined recursively in this same way:

$$\begin{cases} C_0 = 1 \\ C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \end{cases}$$

of which the first few values are

$$\{1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, \ldots\}$$

This is a non-linear recurrence relation, and the best method for solving for C_n is generating functions – which are beyond the scope of this class, alas!