Section 5.3: Decision Trees

Abstract:

Decision trees are defined, and some examples given. Binary search trees store data conveniently for searching later. Some bounds on worst case scenarios are established.

Decision Tree Definition and Terminology

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

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

Examples of decision trees in action

- A decision tree for trees! For example, the Identification of Common Trees of Iowa.

- Figure 5.51, p. 387: Results of tossing a coin 5 times, no two heads in a row (binary decision tree)
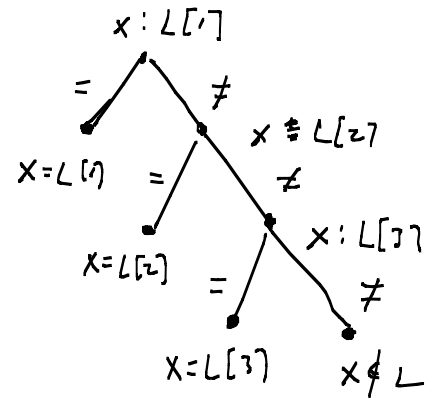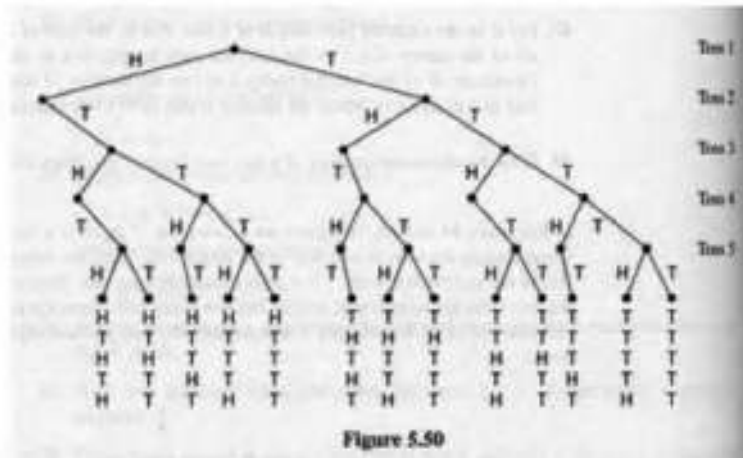
Figure 5.50

- Figure 5.52, p. 388: Sequential Search on 5 elements (binary tree)

  Example: Exercise #1, p. 394.:

- Figure 5.53, p. 389: Binary Search on a sorted list (ternary tree, although it appears binary since those leaves corresponding to equality have been suppressed)
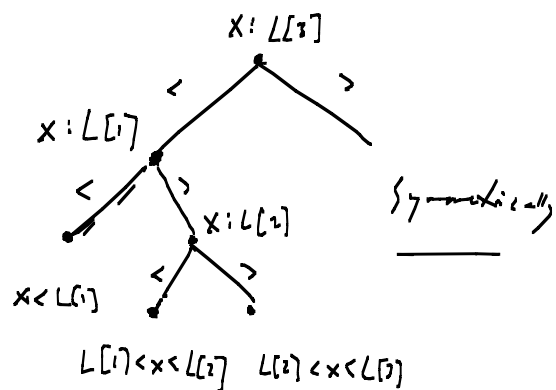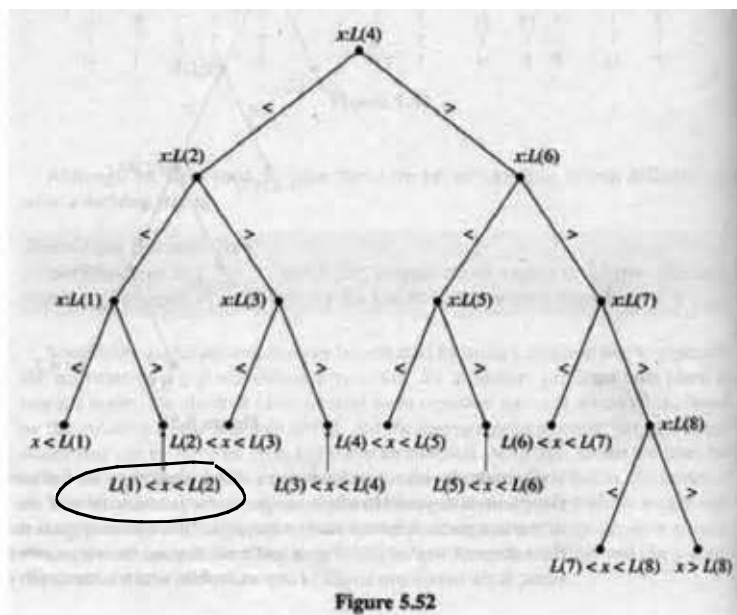
Figure 5.52

Notice how clumsy this tree looks, since a power of two was used (rather than 1 less than a power of two).

Example: Practice 24, p. 390.:

* Figure 5.56, p. 393: Sorting a list (binary tree, provided distinct list elements)

Lower Bounds on Searching

In particular about binary trees:

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

2. Any binary tree with m nodes has depth $d \geq \lfloor \log m \rfloor$, 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 looking at the full binary tree situation:

Depth d	Nodes m
0	$1 = 2^1 - 1$
1	$3 = 2^2 - 1$
2	$7 = 2^3 - 1$
3	$15 = 2^4 - 1$
⋮	⋮

Table: Adding one more node 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, $d = \log m$.

A more formal proof is by contradiction and interesting (p. 390):

By contradiction

- ○ Assume $d < \lfloor \log m \rfloor$: then $d \leq \lfloor \log m \rfloor - 1$.
- ○ From fact 1 (above the table),
$$m \leq 2^{d+1} - 1 \leq 2^{\lfloor \log m \rfloor - 1 + 1} - 1 \leq 2^{\log m} - 1 = m - 1.$$

By contradiction, $d \geq \lfloor \log m \rfloor$.

These facts lead to the following

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

The ``+1'' comes about because a decision tree representing the search problem has leaves which report the outcome of the search: hence its depth is 1 more than the depth of a tree containing only the internal nodes (representing the comparisons themselves). It's actually the arcs we should be counting....

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).

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 5.55:
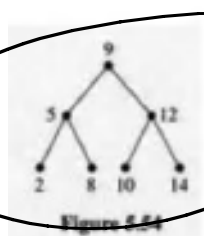
9, 12, 10, 5, 8, 2, 14

Example: Practice #25, p. 392.:   12, 9, 14, 5, 10, 8, 2

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 decision tree for the binary tree search algorithm.
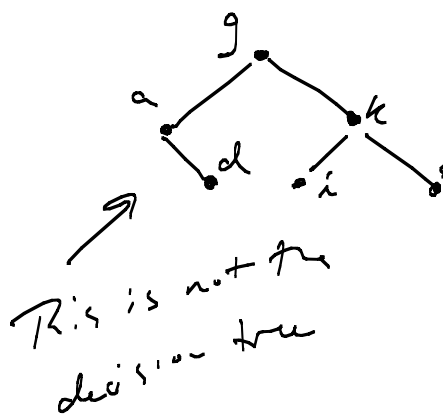
Example: Exercise #9, p. 395.:

$n = 6$ $1 + \lfloor \log_2 6 \rfloor = 3$

$\{a, d, g, i, k, s\}$: g, a, k, s, i, d
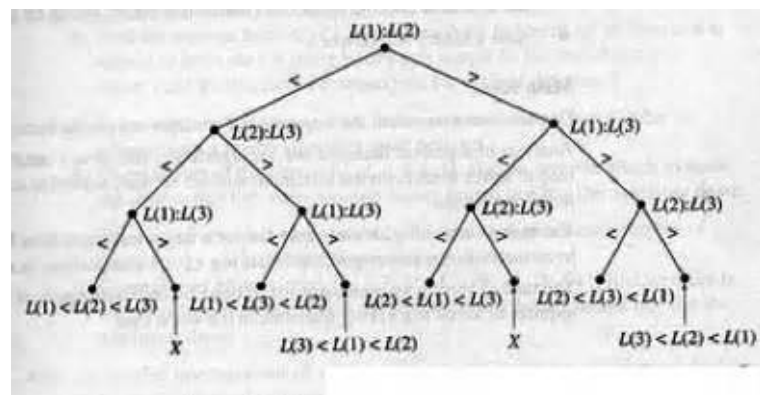
What's the worst way to enter the data into a binary search tree, if one is seeking to create a balanced tree?

Sorting

Sorted.

This is not the decision tree

Examine Figure 5.56, p. 393:

In this case, we're sorting a three-element list using a decision tree. The author calls this a stupid algorithm (actually, ``not particularly astute''): why?

Example: (Practice #26, p. 393: . How would we modify Figure 5.56?) 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 are at least that many leaves p ($p \geq n!$). (In Figure 5.56, 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.
- Since the tree is binary, $p \leq 2^d$ (the maximum number of leaves possible at depth d).
- Taking logs (base 2), we get $\log p \leq d$, or $d \geq \lceil \log p \rceil$, where $\lceil x \rceil$ is the **ceiling** function, which yields the smallest integer greater than or equal to *x*.
- Hence, $d \geq \lceil \log n! \rceil$.

This is the Theorem on the lower bound for sorting: that you have to go to at least a depth of $\lceil \log n! \rceil$ in the worst case.

Example: Exercise #23, p. 397: shows that this lower bound ($\lceil \log n! \rceil$) is on the order of $n \log n$ (as we discovered for mergesort).

Example: Exercise #15, p. 395: .

Wed Mar 15 17:40:18 EST 2006