2024-04-24 / Reading Comprehension / English for Software Development

Chapter 5. The Search for the Perfect Data Structure

[Introduction; second to fourth paragraphs]

We are searching for things all the time. Often this happens unconsciously, but sometimes we are made painfully aware of it, such as when a mundane activity like getting your car keys can turn into an agonizing search. The more places we have to search and the more items we have go through, the more ( ) is to find what we're looking for. And we ( ) accumulate lots of artifacts over the years—after all, we are descendants of hunters and gatherers. In addition to collector's item such as stamps, coins, or sports cards, we also hoard lots of books, pictures, or clothes over time. Sometimes this happens as a side effect of some hobby or passion—I know several home improvement aficionados who have amassed an impressive collection of tools.

If your bookshelf is ( ) alphabetically or by topic, or if your picture collection is electronic and has location and time tags, finding a particular book or picture might be easy, but if the number of items is large and lacks any form of organization, such a search can become arduous.

The situation gets much worse when it comes to electronically stored data. Since we have access to ( ) unlimited storage space, the size of stored data grows rapidly. For example, ( ) YouTube, 300 hours of video are uploaded to its site every minute.

Searching is a ( ) problem in real life, and it is also an important topic in computer science. Algorithms and data structures can speed up the search process ( ). And what works with data can sometimes help with storing and ( ) physical artifacts. There is indeed a very simple method that relieves you from ever having to search for your car keys again—if you follow the method meticulously, that is.

[Survival by Boogle; second to fifth paragraphs]

Finding those titles that are safe to step on is not an easy task, since without any further restrictions the number of possibilities is ( ): more than one thousand trillion—1 followed by 15 zeros. The clue to finding a viable sequence of tiles that leads safely to the other side of the floor is that the titles should spell the name *Iehova*. Although this information basically solves the puzzle, the identification of the correct sequence of tiles still requires some work. Surprisingly, it involves a search that systematically ( ) the space of possibilities.

This task is similar to playing Boogle, where the goal is to find strings of connected letters on a grid that form words. Indiana Jone's task seems to be much easier because he knows the word already. However, in Boogle, ( ) characters must be on ( ) tiles, which is not a constraint for the tile floor challenge and thus leaves more possibilities for Indiana Jones to consider and makes it harder again.

To illustrate the search problem to be solved, let us assume for simplicity that the tile floor ( ) six rows, each containing eight different letters, which ( ) a grid of 48 tiles. If the correct path consists of one tile in each row, there are $8 \times 8 \times 8 \times 8 \times 8 \times 8 = 262{,}144$ paths through all possible combinations of tiles in the six rows. Of these only one is viable.

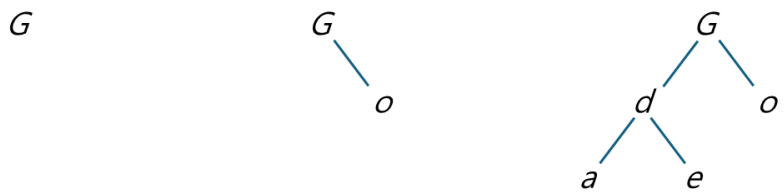

Figure 1. Binary Tree Examples

How does Indiana Jones now find the path? Guided by the clue word, he finds the tile that contains the letter *I* in the first row and steps on it. This identification is itself a search process that ( ) multiple steps. If the letter on the tiles do not ( ) in alphabetical order, Indiana Jones has to look at them one by one until he finds the tile with the letter *I*. He then steps on it, and ( ) searching for the tile with the letter *e* in the second row of tiles and so forth.

[Lean Is Not Always Better; first three paragraphs]

The main ( ) of the list data structure implementation of a dictionary is the high cost for repeatedly accessing items that are located toward the end of the list.

The binary search tree data structure tries to avoid this problem by partitioning the search space more evenly to support faster search. A binary tree is a tree ( ) each node has at most two children. As mentioned, nodes without any children are called *leaves*. The nodes with children are called *internal nodes*. Also, the node that has no parent is called the *root* of the tree.

Let's look at some binary tree examples shown in Figure 5.1. The left one is a tree of a single node—the simplest tree one can imagine. The root, which is also a leaf, consists of the letter G. This tree doesn't really look much like a tree, similar to a list of one element that doesn't look like a list. The middle tree has one more node, a child o, added to the root. In the right tree a child *d* itself has children a and e. This example illustrates that if we cut the link to its parent, a node becomes the root of a separate tree, which is called a subtree of its parent. In this case the tree with root *d* and two children a and e is a subtree of the node G, as is the single-node tree with root o. This suggests that a tree is ( ) a recursive data structure and can be defined in the following way. A tree is either a single node, or it is a node that has one or two subtrees (whose roots are the children of the node). This recursive structure of trees was indicated Chapter 4, which presented a recursive algorithm to traverse a family tree.

Key sentences

- The more places we have to search, the more difficult it is to find what we are looking for.
- The binary search tree data structure tries to avoid this problem by partitioning the search space more evenly.
- A tree is either a single node, or it is a node that has one or two subtrees whose roots are the children of the node.