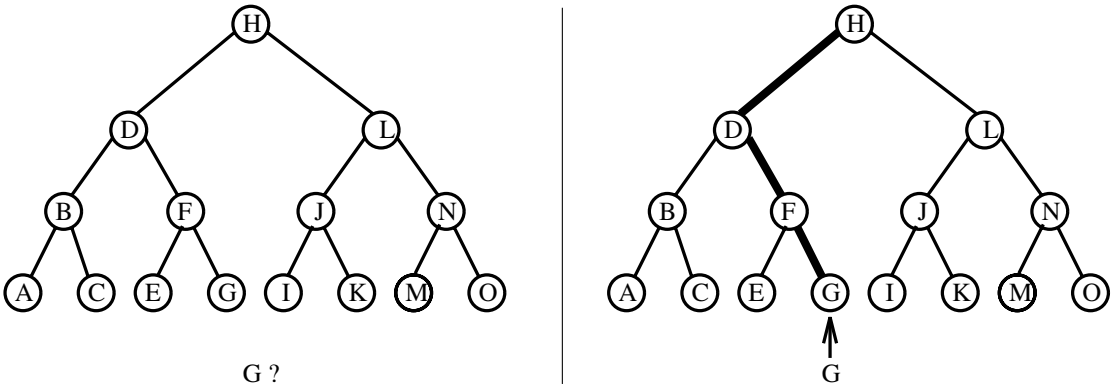

INPUT OUTPUT

14.2 Searching

Input description: A set of n keys S, and a query key q.

Problem description: Where is q in S?

Discussion: "Searching" is a word that means different things to different people. Searching for the global maximum or minimum of a function is the problem of *unconstrained optimization* and is discussed in Section 13.5 (page 407). Chess-playing programs select the best move to make via an exhaustive search of possible moves using a variation of backtracking (see Section 7.1 (page 231)).

Here we consider the task of searching for a key in a list, array, or tree. Dictionary data structures maintain efficient access to sets of keys under insertion and deletion and are discussed in Section 12.1 (page 367). Typical dictionaries include binary trees and hash tables.

We treat searching as a problem distinct from dictionaries because simpler and more efficient solutions emerge when our primary interest is static searching without insertion/deletion. These little data structures can yield large performance improvements when properly employed in an innermost loop. Also, ideas such as binary search and self-organization apply to other problems and well justify our attention.

Our two basic approaches are sequential search and binary search. Both are simple, yet have interesting and subtle variations. In *sequential search*, we start from the front of our list/array of keys and compare each successive item against the key until we find a match or reach the end. In *binary search*, we start with a sorted array of keys. To search for key q, we compare q to the middle key $S_{n/2}$. If q is before $S_{n/2}$, it must reside in the top half of our set; if not, it must reside in the

bottom half of our set. By repeating this process on the correct half, we find the key using $\lceil \lg n \rceil$ comparisons. This is a big win over the $n/2$ comparisons we expect with sequential search. See Section 4.9 (page 132) for more on binary search.

A sequential search is the simplest algorithm, and likely to be fastest on up to about 20 elements. Beyond (say) 100 elements, binary search will clearly be more efficient than sequential search, easily justifying the cost of sorting if there will be multiple queries. Other issues come into play, however, in identifying the proper variant of the algorithm:

- *How much time can you spend programming?* – A binary search is a notoriously tricky algorithm to program correctly. It took seventeen years after its invention until the first *correct* version of a binary search was published! Don't be afraid to start from one of the implementations described below. Test it completely by writing a driver that searches for every key in the set S as well as between the keys.

- *Are certain items accessed more often than other ones?* – Certain English words (such as "the") are much more likely to occur than others (such as "defenestrate"). We can reduce the number of comparisons in a sequential search by putting the most popular words on the top of the list and the least popular ones at the bottom. Nonuniform access is usually the rule, not the exception. Many real-world distributions are governed by *power laws*. A classic example is word use in English, which is fairly accurately modeled by *Zipf's law*. Under *Zipf's law*, the ith most frequently accessed key is selected with probability $(i-1)/i$ times the probability of the $(i-1)$st most popular key, for all $1 \leq i \leq n$.

  Knowledge of access frequencies is easy to exploit with sequential search. But the issue is more complicated with binary trees. We want popular keys close to the root (so we hit them quickly) but not at the expense of losing balance and degenerating into sequential search. The answer is to employ a dynamic programming algorithm to find the *optimal binary search tree*. The key observation is that each possible root node i partitions the space of keys into those to the left of i and those to the right; each of which should be represented by an optimal binary search tree on a smaller subrange of keys. The root of the optimal tree is selected to minimize the expected search costs of the resulting partition.

- *Might access frequencies change over time?* – Preordering a list or tree to exploit a skewed access pattern requires knowing the access pattern in advance. For many applications, it can be difficult to obtain such information. Better are *self-organizing lists*, where the order of the keys changes in response to the queries. The best self-organizing scheme is move-to-front; that is, we move the most recently searched-for key from its current position to the front of the list. Popular keys keep getting boosted to the front, while unsearched-for

keys drift towards the back of the list. There is no need to keep track of the frequency of access; just move the keys on demand. Self-organizing lists also exploit *locality of reference*, since accesses to a given key are likely to occur in clusters. A hot key will be maintained near the top of the list during a cluster of accesses, even if other keys have proven more popular in the past.

Self-organization can extend the useful size range of sequential search. However, you should switch to binary search beyond 100 elements. But consider using *splay trees*. which are self-organizing binary search trees that rotate each searched-for node to the root. They offer excellent amortized performance guarantees.

- *Is the key close by?* – Suppose we know that the target key is to the right of position p, and we think it is nearby. A sequential search is fast if we are correct, but we will be punished severely when we guess wrong. A better idea is to test repeatedly at larger intervals ($p+1$, $p+2$, $p+4$, $p+8$, $p+16$, ...) to the right until we find a key to the right of our target. Now we have a window containing the target and we can proceed with binary search.

  Such a *one-sided binary search* finds the target at position $p+l$ using at most $2\lceil \lg l \rceil$ comparisons, so it is faster than binary search when $l << n$, yet it can never be much worse. One-sided binary search is particularly useful in unbounded search problems, such as in numerical root finding.

- *Is my data structure sitting on external memory?* – Once the number of keys grows *too* large, a binary search loses its status as the best search technique. A binary search jumps wildly around the set of keys looking for midpoints to compare, and so each comparison requires reading a new page in from external memory. Much better are data structures such as B-trees (see Section 12.1 (page 367)) or Emde Boas trees (see notes below), which cluster the keys into pages to minimize the number of disk accesses per search.

- *Can I guess where the key should be?* – In *interpolation search*, we exploit our understanding of the distribution of keys to guess where to look next. An interpolation search is probably a more accurate description of how we use a telephone book than binary search. Suppose we are searching for *Washington, George* in a sorted telephone book. We would be safe making our first comparison three-fourths of the way down the list, essentially doing two comparisons for the price of one.

  Although an interpolation search is an appealing idea, we caution against it for three reasons: First, you have to work very hard to optimize your search algorithm before you can hope for a speedup over binary search. Second, even if you do beat a binary search, it is unlikely to be by enough to have justified the exercise. Finally, your program will be much less robust and efficient when the distribution changes, such as when your application gets ported to work on French words instead of English.

Implementations: The basic sequential and binary search algorithms are simple enough that you may consider implementing them yourself. That said, the C standard library contains `bsearch`, a generic implementation of (presumably) a binary search. The C++ *Standard Template Library* (STL) provides `find` (sequential search) and `binary_search` iterators. *Java Collections* (JC), provides `binarySearch` in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*).

Many data structure textbooks provide extensive and illustrative implementations. Sedgewick (*http://www.cs.princeton.edu/∼rs/*) [Sed98] and Weiss (*http://www.cs.fiu.edu/∼weiss/*) [Wei06] provide implementation of splay trees and other search structures in both C++ and Java.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides up-to-date surveys on all aspects of dictionary data structures. Other surveys include Mehlhorn and Tsakalidis [MT90b] and Gonnet and Baeza-Yates [GBY91]. Knuth [Knu97a] provides a detailed analysis and exposition on all fundamental search algorithms and dictionary data structures, but omits such modern data structures as red-black and splay trees.

The next position probed in linear interpolation search on an array of sorted numbers is given by

$$next = (low - 1) + \lceil \frac{q - S[low - 1]}{S[high + 1] - S[low - 1]} \times (high - low + 1) \rceil$$

where q is the query numerical key and S the sorted numerical array. If the keys are drawn independently from a uniform distribution, the expected search time is $O(\lg \lg n)$ [DJP04, PIA78].

Nonuniform access patterns can be exploited in binary search trees by structuring them so that popular keys are located near the root, thus minimizing search time. Dynamic programming can be used to construct such optimal search trees in $O(n \lg n)$ time [Knu98]. Stout and Warren [SW86b] provide a slick algorithm to efficiently transform a binary tree to a minimum height (optimally balanced) tree using rotations.

The Van Emde Boas layout of a binary tree (or sorted array) offers better external memory performance than conventional binary search, at a cost of greater implementation complexity. See the survey of Arge, et al. [ABF05] for more on this and other cache-oblivious data structures.

Related Problems: Dictionaries (see page 367), sorting (see page 436).