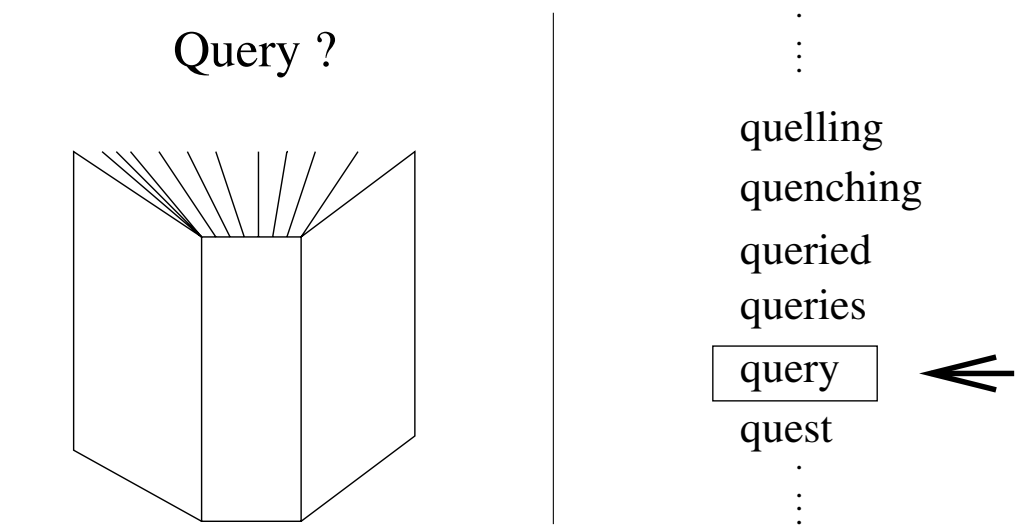

12.1 Dictionaries

Input description: A set of n records, each identified by one or more key fields.

Problem description: Build and maintain a data structure to efficiently locate, insert, and delete the record associated with any query key q.

Discussion: The abstract data type "dictionary" is one of the most important structures in computer science. Dozens of data structures have been proposed for implementing dictionaries, including hash tables, skip lists, and balanced/unbalanced binary search trees. This means that choosing the best one can be tricky. It can significantly impact performance. *However, in practice, it is more important to avoid using a bad data structure than to identify the single best option available.*

An essential piece of advice is to carefully isolate the implementation of the dictionary data structure from its interface. Use explicit calls to methods or subroutines that initialize, search, and modify the data structure, rather than embedding them within the code. This leads to a much cleaner program, but it also makes it easy to experiment with different implementations to see how they perform. Do not obsess about the procedure call overhead inherent in such an abstraction. If your application is so time-critical that such overhead can impact performance, then it is even more essential that you be able to identify the right dictionary implementation.

In choosing the right data structure for your dictionary, ask yourself the following questions:

- *How many items will you have in your data structure?* – Will you know this number in advance? Are you looking at a problem small enough that a simple data structure will suffice, or one so large that we must worry about running out of memory or virtual memory performance?

- *Do you know the relative frequencies of insert, delete, and search operations?* – Static data structures (like sorted arrays) suffice in applications when there are no modifications to the data structure after it is first constructed. *Semi-dynamic* data structures, which support insertion but not deletion, can have significantly simpler implementations than fully dynamic ones.

- *Can we assume that the access pattern for keys will be uniform and random?* – Search queries exhibit a skewed access distribution in many applications, meaning certain elements are much more popular than others. Further, queries often have a sense of temporal locality, meaning elements are likely to be repeatedly accessed in clusters instead of at fairly regular intervals. Certain data structures (such as splay trees) can take advantage of a skewed and clustered universe.

- *Is it critical that individual operations be fast, or only that the total amount of work done over the entire program be minimized?* – When response time is critical, such as in a program controlling a heart-lung machine, you can't wait too long between steps. When you have a program that is doing a lot of queries over the database, such as identifying all criminals who happen to be politicians, it is not as critical that you pick out any particular congressman quickly as it is that you get them all with the minimum total effort.

An object-oriented generation has emerged as no more likely to write a container class than fix the engine in their car. This is good; default containers should do just fine for most applications. Still, it is good sometimes to know what you have under the hood:

- *Unsorted linked lists or arrays* – For small data sets, an unsorted array is probably the easiest data structure to maintain. Linked structures can have terrible cache performance compared with sleek, compact arrays. However, once your dictionary becomes larger than (say) 50 to 100 items, the linear search time will kill you for either lists or arrays. Details of elementary dictionary implementations appear in Section 3.3 (page 72).

  A particularly interesting and useful variant is the *self-organizing list*. Whenever a key is accessed or inserted, we always move it to head of the list. Thus, if the key is accessed again sometime in the near future, it will be near the front and so require only a short search to find it. Most applications exhibit

both uneven access frequencies and locality of reference, so the average time for a successful search in a self-organizing list is typically much better than in a sorted or unsorted list. Of course, self-organizing data structures can be built from arrays as well as linked lists and trees.

- *Sorted linked lists or arrays* – Maintaining a sorted linked list is usually not worth the effort unless you are trying to eliminate duplicates, since we cannot perform binary searches in such a data structure. A sorted array will be appropriate if and only if there are not many insertions or deletions.

- *Hash tables* – For applications involving a moderate-to-large number of keys (say between 100 and 10,000,000), a hash table is probably the right way to go. We use a function that maps keys (be they strings, numbers, or whatever) to integers between 0 and $m-1$. We maintain an array of m buckets, each typically implemented using an unsorted linked list. The hash function immediately identifies which bucket contains a given key. If we use a hash function that spreads the keys out nicely, and a sufficiently large hash table, each bucket should contain very few items, thus making linear searches acceptable. Insertion and deletion from a hash table reduce to insertion and deletion from the bucket/list. Section 3.7 (page 89) provides a more detailed discussion of hashing and its applications.

  A well-tuned hash table will outperform a sorted array in most applications. However, several design decisions go into creating a well-tuned hash table:

  - *How do I deal with collisions?:* Open addressing can lead to more concise tables with better cache performance than bucketing, but performance will be more brittle as the load factor (ratio of occupancy to capacity) of the hash table starts to get high.
  - *How big should the table be?:* With bucketing, m should about the same as the maximum number of items you expect to put in the table. With open addressing, make it (say) 30% larger or more. Selecting m to be a prime number minimizes the dangers of a bad hash function.
  - *What hash function should I use?:* For strings, something like

    $$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j}) \bmod m$$

    should work, where α is the size of the alphabet and *char(x)* is the function that maps each character x to its ASCII character code. Use Horner's rule (or precompute values of α^x) to implement this hash function computation efficiently, as discussed in Section 13.9 (page 423). This hash function has the nifty property that

    $$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

so hash codes of successive m-character windows of a string can be computed in constant time instead of $O(m)$.

Regardless of which hash function you decide to use, print statistics on the distribution of keys per bucket to see how uniform it *really* is. Odds are the first hash function you try will not prove to be the best. Botching up the hash function is an excellent way to slow down any application.

- *Binary search trees* – Binary search trees are elegant data structures that support fast insertions, deletions, and queries. They are reviewed in Section 3.4 (page 77). The big distinction between different types of trees is whether they are explicitly rebalanced after insertion or deletion, and how this rebalancing is done. In *random search trees*, we simply insert a node at the leaf position where we can find it and no rebalancing takes place. Although such trees perform well under random insertions, most applications are not really random. Indeed, unbalanced search trees constructed by inserting keys in sorted order are a disaster, performing like a linked list.

  Balanced search trees use local *rotation* operations to restructure search trees, moving more distant nodes closer to the root while maintaining the in-order search structure of the tree. Among balanced search trees, AVL and 2/3 trees are now passé, and *red-black trees* seem to be more popular. A particularly interesting self-organizing data structure is the *splay tree*, which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree, allowing faster searches.

  Bottom line: Which tree is best for your application? Probably the one of which you have the best implementation. The flavor of balanced tree is probably not as important as the skill of the programmer who coded it.

- *B-trees* – For data sets so large that they will not fit in main memory (say more than 1,000,000 items) your best bet will be some flavor of a B-tree. Once a data structure has to be stored outside of main memory, the search time grows by several orders of magnitude. With modern cache architectures, similar effects can also happen on a smaller scale, since cache is much faster than RAM.

  The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed. With B-tree we can access enormous numbers of keys using only a few disk accesses. To get the full benefit from using a B-tree, it is important to understand how the secondary storage device and virtual memory interact, through constants such as page size and virtual/real address space. *Cache-oblivious algorithms* (described below) can mitigate such concerns.

Even for modest-sized data sets, unexpectedly poor performance of a data structure may result from excessive swapping, so listen to your disk to help decide whether you should be using a B-tree.

- *Skip lists* – These are somewhat of a cult data structure. A hierarchy of sorted linked lists is maintained, where a coin is flipped for each element to decide whether it gets copied into the next highest list. This implies roughly $\lg n$ lists, each roughly half as large as the one above it. Search starts in the smallest list. The search key lies in an interval between two elements, which is then explored in the next larger list. Each searched interval contains an expected constant number of elements per list, for a total expected $O(\lg n)$ query time. The primary benefits of skip lists are ease of analysis and implementation relative to balanced trees.

Implementations: Modern programming languages provide libraries offering complete and efficient container implementations. The C++ *Standard Template Library* (STL) is now provided with most compliers, and available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01], and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

LEDA (see Section 19.1.1 (page 658)) provides an extremely complete collection of dictionary data structures in C++, including hashing, perfect hashing, B-trees, red-black trees, random search trees, and skip lists. Experiments reported in [MN99] identified hashing as the best dictionary choice, with skip lists and 2-4 trees (a special case of B-trees) as the most efficient tree-like structures.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) is more comprehensive, and available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Notes: Knuth [Knu97a] provides the most detailed analysis and exposition on fundamental dictionary data structures, but misses certain modern data structures as red-black and splay trees. Spending some time with his books is a worthwhile rite of passage for all computer science students.

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]. Good textbook expositions on dictionary data structures include Sedgewick [Sed98], Weiss [Wei06], and Goodrich/Tamassia [GT05]. We defer to all these sources to avoid giving original references for each of the data structures described above.

The 1996 DIMACS implementation challenge focused on elementary data structures, including dictionaries [GJM02]. Data sets, and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

The cost of transferring data back and forth between levels of the memory hierarchy (RAM-to-cache or disk-to-RAM) dominates the cost of actual computation for many

problems. Each data transfer moves one block of size b, so efficient algorithms seek to minimize the number of block transfers. The complexity of fundamental algorithm and data structure problems on such an external memory model has been extensively studied [Vit01]. *Cache-oblivious* data structures offer performance guarantees under such a model without explicit knowledge of the block-size parameter b. Hence, good performance can be obtained on any machine without architecture-specific tuning. See [ABF05] for an excellent survey on cache-oblivious data structures.

Several modern data structures, such as splay trees, have been studied using *amortized analysis*, where we bound the total amount of time used by any sequence of operations. In an amortized analysis, a single operation can be very expensive, but only because we have already benefited from enough cheap operations to pay off the higher cost. A data structure realizing an amortized complexity of $O(f(n))$ is less desirable than one whose worst-case complexity is $O(f(n))$ (since a very bad operation might still occur) but better than one with an average-case complexity $O(f(n))$, since the amortized bound will achieve this average on any input.

Related Problems: Sorting (see page 436), searching (see page 441).