Chapter XI
Strategies for Static Tables

Dean Kelley
Minnesota State University, Mankato, USA

ABSTRACT

This chapter presents three alternatives for structuring static tables—those tables in which the collection of keys remains unchanged and in which the FIND operation is optimized. Each alternative provides performance guarantees for the FIND operation which can help those who design and/or implement systems achieve performance guarantees of their own. The chapter provides clear and concise algorithms for construction and/or usage and simple guidelines for choosing among the strategies. It is intended that this presentation will help inform system design decisions. It is further intended that this chapter will assist implementation activities for systems which make use of static tables.

INTRODUCTION

System designers and implementers are frequently required to guarantee system performance. Often, performance depends on efficient interaction with tables and dictionaries of data. When table operations require communication via resources whose availability may not be guaranteed (e.g., networked communication with a database), designers/implementers may not be able to make firm guarantees of system or system component performance. Consequently, it may be necessary to incorporate tables into a system rather than rely on external support for them.

Generally, tables fall into two categories depending on the type of operations they are required to support. *Static* tables are ones which, once built, do not change. The set of keys that they contain remains unchanged throughout their lifetime (although the data associated with the keys may change). A *dynamic* table may change its set of keys by means of insertion and deletion during the time that the table is active. Both types of table support searching to determine if a specific key is present as well as to retrieve data associated with a key.

This chapter focuses on techniques for implementing static tables. The first section presents necessary technical background. The second section presents a general technique for situations in which keys can be totally ordered and searching can be accomplished by key comparisons. The third section extends that technique to a situation in which the probabilities for search operations for each key are known. The fourth section presents

a technique in which, at most, a single key comparison is required and that comparison is only for equality, consequently the keys do not need to be totally ordered.

The topics of this chapter provide system designers and implementers with alternatives for static tables which can yield firm guarantees of performance.

BACKGROUND

This chapter is concerned with *static* tables, tables in which data are associated with keys and the set of keys remains unchanged throughout the life of the table. Access to a specific item of data is accomplished by searching for its key by means of a FIND operation.

Table techniques are evaluated by their consumption of time and space. Time is measured by the number of key comparisons which take place during a FIND operation. Space is measured by the size of the structure(s) necessary to contain the keys and additional supporting data necessary to make the technique work. We do not include the "user" data associated with the keys in the space cost.

Key comparisons may be for equality ("k_1 *equals* k_2") or order ("k_1 *is before* k_2"). The search techniques of the second and third sections require comparisons for order. Consequently, the comparison of any two keys in the set of keys must yield information about their relative order. When a relation equivalent to "$\leq$" holds for all elements of a set, that set is said to be *totally ordered*. Thus, the techniques of the second and third sections require that the keys be from a totally ordered set.

The study of efficient techniques for static tables owes much to the paper by Yao (1981). In this paper it was shown that if the number of keys in the table is small relative to the number of possible keys, then the number of comparisons necessary to determine if a particular key is present (and where it is) is $\Omega(\log n)$. That is, it requires at least a constant times $\log n$ comparisons. Consequently, binary search in a sorted array (the topic of the second section) is the best approach under these circumstances.

Alternatives to array-based tables lead to tree-like structures built with links. In the third section, we present a specific type of binary tree structure, the *optimal binary search tree*, which yields a guarantee of the *average* number of key comparisons over all possible searches. Binary search trees were independently discovered by a number of people. Gilbert and Moore (1959) provided the basis for constructing optimal binary search trees when the probabilities of each key being searched for are known.

Yao (1981) also observed that by allowing one additional item to be stored, the $\Omega(\log n)$ bound can be beat and constant-time FIND performance is possible. Typically, the additional item is the name of a hash function. Fredman, Komlòs, and Szemerédi (1984) showed that there is a structuring technique which uses $n+o(n)$ space and attains $O(1)$ time for the FIND operation. The tool of the final section is based on the improved technique of Czech, Havas, and Majewski (1992) and Majewski, Wormald, Havas, and Czech (1996).

SORTED TABLES

A *comparison-based* search technique is one in which information about the location of a key is obtained by comparing it with keys in the table. Comparison-based techniques require that the keys involved be from a totally ordered set so that any key can be compared with any other key. It is known (Yao, 1981) that $\Omega(\log n)$ comparisons are required for comparison-based static tables. Binary search (presented below) attains this cost and is, therefore, an optimal approach.

Binary Search in a Sorted Array

A sorted array $A[\ ]$ can support an efficient static table. Suppose that the key x is being searched for in $A[\ ]$. Because the array is in sorted order, com-

parison of x with the key at $A[k]$ yields 3 possible results:

- x compares equal to the key at $A[k]$, in which case x is found.
- x compares less than the key at $A[k]$. In this case, if x is present in the table, it must be at an index less than k.
- x compares greater than the key at $A[k]$. In this case, if x is present in the table, it must be at an index greater than k.

If k is chosen as the middle index of the sorted array $A[]$, approximately half of the keys can be eliminated from consideration by the comparison of x and $A[k]$. The process is then repeated on the portion of the array which was not eliminated, again eliminating approximately half of those keys. If x is present, it will eventually be at the middle of some segment of the array that is still under consideration (in the worst case, it would be the middle element of a 1-element sub-array). If x is not present, the segment still under consideration will eventually shrink to a 0-element sub-array.

Because the problem size is approximately halved at each step and only a constant number of comparisons are performed at each step, this *binary search* algorithm requires $O(\log n)$ comparisons to determine if x is present in $A[\ ]$ and, if so, its location.

Algorithm and Implementation

The discussion above leads directly to a simple recursive algorithm. Suppose that the array $A[i...j]$ is to be searched for the key x. The initial values of i and j correspond to the first and last index of the array. Subsequently, i and j are adjusted to reflect the beginning and ending indices of the sub-array under consideration.

To search $A[i...j]$ for x:
  If $j < i$ then
    the key x is not present in the array
  Let m be the middle index of $A[i...j]$
  If x compares equal to the key at $A[m]$ then
    the key x has been found at $A[m]$
  Else, if x compares less than the key at $A[m]$ then recursively search $A[i...(m\text{-}1)]$ for x
  Else,
    recursively search $A[(m+1)...j]$ for x

To implement this recursive algorithm, there are only two significant considerations: how to find the middle index and how to pass the appropriate sub-array to a subsequent recursive call (if one is needed). The middle index is obtained by the calculation $\left\lfloor \frac{i+j}{2} \right\rfloor$. Here, $\lfloor\ \rfloor$ is the floor function which yields the largest integer smaller than or equal to y. This calculation can be realized in many modern programming languages by allowing integer division to truncate or by using a truncation function to eliminate any fractional portion of the result of the division. Care must be taken to correctly deal with potential addition over-flow when i and j are extreme.

Care must also be taken when passing a sub-array to a subsequent recursive call. Programming languages which support call-by-reference can implement the call with the reference to the array and two indices as parameters. In programming languages which do not support call-by-reference, using a global variable for the array and passing two indices may avoid the penalty of an execution-time copy of the array being made at each recursive call.

The comparison of x and the key at $A[m]$ may cost considerably more than the trivial cost of comparing base-types. For example, comparing two character strings involves character-by-character comparison. If a key comparison involves substantial cost, it is advantageous to store the result of the key comparison and then use that result at the two points where the algorithm makes decisions based on that comparison.

Finally, it should be pointed out that the overhead incurred by recursive binary search is not likely to be significant. A binary search of a sorted ar-

ray containing 700 million keys requires roughly 30 recursive calls. Moreover, since the algorithm requires few local variables, activation records associated with each invocation of the function or method are quite small.

Because the recursion in the binary search algorithm is *tail recursion*, this recursive algorithm can easily be converted into an iterative one.

To iteratively binary search $A[i...j]$ for x:
While $i \leq j$
    Let m be the middle index of $A[i...j]$
    If x compares equal to the key at $A[m]$ then
        the key x has been found at $A[m]$
    Else, if x compares less than the key at $A[m]$ then
        $j = m - 1$
    Else,
        $i = m + 1$
x is not in the array

An iterative version of the algorithm may yield slightly faster execution for long searches although the asymptotic behavior of the algorithm is unchanged since $O(\log n)$ comparisons are made in both versions.

Building, Saving, and Recovering the Table

The initial construction of the table consists of loading the keys (and any associated data) into an array. If the keys can be loaded in sorted order, there is no other preparation necessary. If the keys are not available in sorted order, then the array must be sorted into key order before the table can be searched.

Sorting an array of n items can be accomplished with $O(n\log n)$ cost. Most language libraries contain sorting utilities which can efficiently sort an array. Note that sorting the array is purely a preprocessing step. It must be done before the array can be used as a table, but does not need to be done at any other time. In fact, once sorted, the table can be saved in sorted order so that subsequent loading of it requires no sorting step.

In a static table, keys may be associated with additional data which is allowed to change. If the associated data has changed during the lifetime of the table, it may be necessary to save the table at system termination. If the array is saved in order, restoring the table consists of no more than simply re-loading the array.

TABLES OPTIMIZED BY PROBABILITY

Binary search in a sorted table assumes that each key is equally likely to be the objective of a search. Frequently, it is known (or can be determined) that some keys are more likely to be searched for than others. In this case, overall performance can be improved by organizing the table so that frequent searches require fewer comparisons than infrequent ones.

In this section, a technique is presented for optimizing tables so that the *average number of comparisons* for all keys is minimized. The underlying structure is a linked structure known as a *binary search tree*. Linked structures have the added benefit that they are less prone to memory allocation and swapping delays than contiguous arrays. On the other hand, linked structures require additional space overhead for their links. Still, the overall space requirement is directly proportional to the table size (i.e., $O(n)$) for a table containing n keys).

Binary Search Trees

A *binary search tree* is a binary tree which has the *search order property*:

For any node X in the tree, the key at X compares greater than any key in its left sub-tree and less than any key in its right sub-tree.

This property facilitates searching in the tree in much the same manner that binary search is facilitated by the order of a sorted array: a comparison of a node's key and the key being searched for will identify which sub-tree, if any, the search should move into next. The search algorithm that results is similar to binary search of a sorted array:

To search for key x in the binary search tree rooted at Y

    If Y is null then
        x is not present in the tree
    Else, if x compares less than the key at Y then
        recursively search for x in the tree rooted at Y's left child
    Else, if x compares greater than the key at Y then
        recursively search for x in the tree rooted at Y's right child
    Else,
        the key x has been found at Y

A binary search tree with n keys can be structured so that its height is $\Theta(\log n)$. Since a search begun at the root of the tree will descend one level at each recursive invocation of the search, the number of comparisons required for the longest search in a tree of height $\Theta(\log n)$ will also be $\Theta(\log n)$. Consequently, a well-structured binary search tree (known as a *balanced binary search tree*) yields the same asymptotic performance as binary search in a sorted array.

For non-static tables, insertion and deletion are considerably faster in a balanced binary search tree than they are in a sorted array. As a result, this kind of tree is very efficient for such tables. In the case of static tables, though, there is little argument for using a binary search tree rather than a sorted array except in cases where some other circumstance comes into play. One such circumstance is the *average cost* of the FIND operation when keys have different probabilities of being searched for.

The average number of comparisons performed by FIND is computed by adding up the number of comparisons necessary to FIND each of the n keys in the binary search tree and then dividing by n. By careful structuring, a binary search tree can be built so that keys with high probability of being searched for contribute smaller comparison counts to the sum than those with low probabilities. As a result, the average number of comparisons can be minimized.

The optimization process which is presented in the next subsection is expensive, requiring $O(n^3)$ time to determine the optimal binary search tree for n keys (notes at the end of the chapter indicate how to improve the time complexity to $O(n^2)$). The algorithm requires $\Theta(n^2)$ space. These costs are preprocessing costs, however. Once the tree structure has been determined (and the tree has been built), it is not necessary to repeat the process unless the keys and/or their probabilities change. This is true even if the data associated with the keys changes and it is necessary to save and reload the tree at system termination and startup.

Optimization Algorithm and the Optimal Tree

The optimization algorithm builds two arrays, C and R. The array C contains information about the smallest average number of comparisons for the optimal tree as well as its sub-trees. As C is being constructed, decisions about the values that are its entries provide information about the keys which root sub-trees in the optimal tree. The array R keeps track of the resulting roots of the sub-trees. In the end, the optimal binary search tree can be constructed by extracting information from R.

Let $k_i \ldots k_n$ be the n keys arranged in sorted order and let p_i be the probability that key k_i will be the objective of a search.

The arrays C and R consist of rows and columns. The entry at $C[i,j]$ will ultimately be the smallest average number of comparisons used by FIND in a binary search tree containing the keys $k_i \ldots k_j$. When

the algorithm is finished, $C[1,n]$ will contain the minimum/optimal average number of comparisons required by FIND for the optimal binary search tree containing all of the keys. To compute the entry at $C[i,j]$, the algorithm searches for the best choice of root among the keys $k_i \ldots k_j$. When it finds the best choice, it sets the appropriate value into $C[i,j]$ and places the index of the best root into $R[i,j]$. That is, $R[i,j]$ contains the index of the best choice for the root of the optimal binary search tree for *keys* $k_i \ldots k_j$. In the end, $R[1,n]$ will contain the index of the root of the optimal binary search tree containing all of the keys.

The process that the algorithm performs begins with pairs of indices i and j that are a distance of 1 apart and then repeatedly increases that distance. For each pair of indices i and j, it determines the smallest average cost of FIND in a tree rooted at each possible key between k_i and k_j (inclusive). The minimum of these costs is then placed in $C[i,j]$ and the index of the corresponding key is placed in $R[i,j]$.

The algorithm is given below. For this presentation, assume that C is an array of $n+2$ rows and $n+1$ columns and that R is an array of $n+1$ rows and columns. In both arrays, the row 0 is unused—it is present so that the discussion can refer to key indices beginning at 1 rather than 0.

```
For each i between 1 and n
    Initialize C[i,i] = p_i, C[i,i-1] = 0 and R[i,i] = i
Initialize C[n+1,n] = 0
For each value of d between 1 and n-1
    For each index i between 1 and n-d
        j = i + d
        min = ∞
        For each m between i and j
            If C[i,m-1] + C[m+1,j] < min then
                min = C[i,m-1] + C[m+1,j]
                rmin = m
        R[i,j] = rmin
        C[i,j] = min + p_i + ··· + p_j
```

When the algorithm is finished, $R[1,n]$ contains the index of the key which is at the root of the optimal binary search tree. Suppose that $R[1,n]$ is the index t. Then, in the optimal tree, the root will have as its left sub-tree the optimal tree containing keys $k_1 \ldots k_{t-1}$. This sub-tree will be rooted by the key with index $R[1,t-1]$. Similarly, the right sub-tree of the root will have key $R[t+1,n]$ and contain keys $k_{t+1} \ldots k_n$. These relationships continue down through the tree's levels. Consequently, the optimal binary search tree may be built by processing the $R[\ ]$ array (described in the subsection Extracting the Optimal Tree).

Unsuccessful Searches

The optimization algorithm presupposes that all searches are successful. That is, it is assumed that every time the FIND operation is performed it locates the key it searches for. If this is not the case and the probabilities of unsuccessful searches are known, the following simple adjustment to the algorithm will build a binary search tree that has the minimal average cost for all searches, successful or not.

Suppose that, in addition to the keys $k_1 \ldots k_n$ and their probabilities $p_1, \ldots, p_n$, the probabilities of searches for keys not in this key set are known. Let q_0 be the probability of a search for a key that compares before k_1, let q_n be the probability of a search for a key that compares after k_n and let q_i be the probability of a search for a key that compares between k_i and k_{i+1}. In this case, the only change necessary in the algorithm is to replace the last line,

$$C[i,j] = min + p_i + \cdots + p_j$$

by this line

$$C[i,j] = min + p_i + \cdots + p_j + q_{i-1} + q_i + \cdots + q_j$$

The three nested loops of the algorithm each contribute at worst $O(n)$ cost to build the C and R arrays. Consequently, the algorithm has a worst-case

cost of $O(n^3)$. Because of the two arrays, the space requirement for the algorithm is $\Theta(n^2)$.

Extracting the Optimal Tree

Once the array $R[\,]$ has been built the optimal binary search tree can be constructed by means of an $O(n)$ algorithm that uses the information in $R[\,]$ to make and link together the nodes of the tree. The arithmetic that computes the indices of $R[\,]$ at each step in an iterative version of the algorithm is cumbersome and tends to induce coding errors. For that reason, the algorithm is presented recursively.

It is assumed that a node structure is available which can contain a key (and perhaps a link for the data associated with that key) as well as links to a left child and a right child. In the general situation, the algorithm locates the key that roots the current sub-tree and makes a node for it. Then the left and right sub-trees are recursively built and linked to the node and the node is returned.

The logic of the algorithm will use $R[\,]$ to build the optimal binary search tree for keys $k_i \ldots k_j$. The initial call to the algorithm asks for the optimal tree for $k_1 \ldots k_n$ (that is, it is initially called with $i = 1$ and $j = n$). Each invocation of the algorithm returns the root of a tree.

Build the optimal tree for keys $k_i \ldots k_j$.
  If $i = j$ then
    return a new node for key k_i (the node has no children)
  Else,
    Make a new node x for the key $k_{R[i,j]}$
    If $i \neq R[i,j]$ then
      Build optimal tree for keys $k_i, \ldots, k_{R[i,j]-1}$ at x's left child
    If $j \neq R[i,j]$ then
      Build optimal tree for keys $k_{R[i,j]+1} \ldots k_j$ at x's right child
  Return x

Since it processes each node of the tree exactly once, this algorithm requires $O(n)$ time for a tree with n nodes. As the tree is being constructed, there are never more recursive calls active than there are nodes in the longest path in the resulting tree. Assuming the $R[\,]$ array is passed via call-by-reference (see the section on Sorted Tables), each recursive call requires only a small amount of space for its activation record. As a result, the space requirement for the algorithm is proportional to the height of the resulting tree, which is at most n.

Saving and Restoring Trees

In this subsection, techniques for saving and restoring binary trees are presented. For even relatively small table sizes, it is considerably faster to restore the saved optimal binary search tree from a file than it is to rebuild it by means of the above algorithms.

The techniques for saving and restoring, presented here, will work for *any* binary tree regardless of whether or not it is a binary search tree. The saving technique will save a binary tree so that it can be restored to exactly the same shape. This is significant for optimal binary search trees because it allows the tree to be saved and restored to its optimal shape (which optimizes the average cost of searches) without having to be rebuilt.

Saving the Tree

In order to restore a tree's node to its correct location, it is necessary to know the node's data, how many children that node had, and where they were (i.e., left child, right child). Additionally, during restoration, it is necessary to encounter a node's information at the time when it can have it's children reattached and be attached to its parent. Thus, the saving algorithm is closely tied to the restoration algorithm.

The technique for saving a binary tree so that it can be restored will save each node's data (which will usually consist of a key and its associated data) together with a code indicating the number and placement of its children (left or right). The data/code will be saved in pre-order by means of

an adapted pre-order traversal beginning at the tree's root.

The following algorithm uses the codes '0,' 'L,' 'R,' and '2' to indicate 0 children, a single left child, a single right child, and two children, respectively. Its input is a node *x* for which it saves the tree rooted at *x*. Initially, it is called with the root of the entire tree. It then is recursively called on the left and right children of *x*.

```
Save tree rooted at x
    c = appropriate code for x (i.e., 0, L, R, 2)
    Store (write out) x's data and c
    If x has a left child then
      recursively save tree rooted at x's left child
    If x has a right child then
      recursively save tree rooted at x's right child
```

Because the algorithm is basically a simple pre-order traversal which performs a constant number of operations when it visits a node, it requires $\Theta(n)$ time on a binary tree of n nodes. It's space requirement is the space needed for the recursion which is proportional to the height of the tree (i.e., it requires O(height) space) which is at worst $O(n)$ for a tree with n nodes.

Restoring the Tree

During restoration, the data and codes saved from the nodes will be encountered in pre-order. Consequently, a node can be constructed for the data and then a decision may be made about how many children to attach to the node and where to attach them.

The tree-building algorithm described below also uses the codes '0,' 'L,' 'R,' and '2.' Each time it is invoked, it will return the node which roots the tree it was able to build during that invocation. It first loads the current node data (key and associated data) into a node then it makes decisions about children based on the code associated with the node's data. As it does so, it attaches sub-trees returned by recursive invocations according to the code.

```
BuildTree()
    Get the next node data and code in D and C
    Create a new node x with data D
    If C is 'L' or '2' then
      recursively call BuildTree() and attach the
      tree it builds as x's left child
    If C is 'R' or '2' then
      recursively call BuildTree() and attach the
      tree it builds as x's right child
    Return x
```

This algorithm encounters each node once and therefore requires $O(n)$ time to restore a tree of n nodes. The recursion requires Θ(height) space for the recursion (referring to the height of the resulting tree). The height of a tree is at most $O(n)$ for an n-node tree.

An optimal binary search tree gives minimal *average* performance for FIND. The cost to build the tree ($O(n^3)$ time and $\Theta(n^2)$ space), save the tree ($O(n)$ time and Θ(height) space for the recursion), and restore a saved tree (also $O(n)$ time and Θ(height) space for the recursion) are pre-processing or post-processing costs which are not incurred during the use of the tree.

CONSTANT-TIME TABLES

This section presents an approach for tables in which searching is composed of some computation together with zero or more tests. Because the testing (if done) is for equality, the keys are not required to be from a totally ordered set.

Hashing and Perfect Hash Tables

Hash tables are based on a simple concept. Suppose that the keys are integers in the range 0,1,...n-1. Let $A[0..n\text{-}1]$ be an array and store the data for key k at $A[k]$. (An appropriate flag can be stored in $A[j]$ in the case that there is no data in the table with key j.) The FIND operation is then trivial. To look up the key k, simply go to $A[k]$ and determine if $A[k]$

contains the key or the flag. That is, FIND is a constant-time ($O(1)$) operation in such a table.

Generalizing the above to a useful table structure encounters a number of problems. While a detailed presentation of these problems and their solutions is beyond the scope of this chapter, a general understanding of the problems is important for understanding the powerful and efficient technique of this section.

Hash Functions and Hash Tables

If the keys for the table entries are not integers or not integers in a convenient range, some process must be employed to adapt them for use as indices in an array. This is part of the role of a *hash function*.

A hash function takes a key as an argument and produces an index into the array as its output. Since the output is required to be a viable index for the array, any hash function is closely tied to the size of the array used to store the key's data. For a given key k, the value (index) that the hash function $h()$ produces for k is called the *hash value* of k, denoted $h(k)$. The data associated with k is then found at $A[h(k)]$ in the array (or $A[h(k)]$ will contain a flag indicating that no data associated with key k exists in the table).

The combination of a hash function and an array for which it generates indices is referred to as a *hash table*.

Collisions, Perfect Hashing, and Minimal Perfect Hashing

Translating non-numeric keys into integers is relatively simple (see, for example, Chapter 11 of Cormen, Leiserson, Rivest, & Stein, 2001). In the following, we will assume that the keys are integer values and focus on the problem of translating those values into usable array indices.

A problem arises when the range of *possible* hash values is large relative to the number of *actual* hash values. For example, suppose that keys are 9-digit integers but that there are only a few thousand keys which actually correspond to items in the table. The mechanism which makes array indexing work efficiently is based on the array being allocated as a contiguous segment of memory. Creating an array of a size sufficient to accommodate all 9-digit integers as indices may cause serious runtime delays as the operating system swaps blocks of memory. Such delays are undesirable, particularly when only a relatively small number of the array slots will actually be used.

Consequently, it would be preferred that the range of hash values to be associated with keys be restricted to approximately n (the actual number of keys). It is an unfortunate fact that for any hash function, if the number of possible keys is at least the product of the number of slots in the table and n, then *some* set of n keys will all yield the same hash value—the same array index!

When two or more keys have the same hash value, a *collision* occurs and some strategy must be employed to resolve it. Resolution strategies necessarily involve some form of searching to locate the desired key and, as a result, can yield $O(n^2)$ performance for individual FIND operations.

Things are not as dismal as the previous paragraph seems to suggest. In the case of *static* tables, where all the keys are known in advance, it is possible to construct a *perfect hash function* which will map keys to indices with no collisions. Perfect hash functions which yield indices in the range 0,1,... n-1 (where n is the number of keys) are known as *minimal perfect hash functions*. Hash tables which use minimal perfect hash functions attain theoretically optimal time and space efficiency.

The construction of a minimal perfect hash function is a search process which searches through the space of all potential minimal perfect hash functions on a given set of keys. Because the search space is large and checking an individual function to see if it is perfect may be costly, the process can be very time consuming. However, once the function is constructed, the process never need be repeated unless the set of keys changes.

All currently known techniques for generating minimal perfect hash functions make use of algo-

rithms and representations considerably beyond the scope of this chapter. However, there are several software tools available which implement these techniques and generate minimal perfect hash functions. One of these tools is described in the next subsection and references to other tools are given in the notes at the end of this chapter.

Minimal Perfect Hash Table Tools

This section describes the MinimalPerfectHash class, a freely available software tool which generates minimal perfect hash functions. It can be used to create efficient tables for programs written in Java. At the time of this writing, it is stable and is distributed under the LGPL (Lesser Gnu Public License).

The MinimalPerfectHash Class

The MinimalPerfectHash class is a component of the MG4J Project (Managing Gigabytes for Java). This project is "aimed at providing a free Java implementation of inverted-index compression techniques" (MG4J, 2006). As a by-product, the project offers several general-purpose optimized classes including a minimal perfect hash class. The MG4J project encompasses far more than minimal perfect hash tables, but we focus only on that aspect of it here.

The MinimalPerfectHash class provides an order-preserving minimal perfect hash table constructed for a set of keys. *Order-preserving* means that the order of the keys in the table is the same as the order in which they were provided to the table's constructor. Having the same order simplifies creating any necessary additional structure to contain data associated with the keys.

Constructing and Using the Hash Table

The MinimalPerfectHash class constructors convert a sequence of unique character-based keys into a minimal perfect hash table in which the order of the keys is preserved. A number of constructors are provided. We focus on only one here.

The constructor:

```
public MinimalPerfectHash(String keyFile)
```

creates a new order-preserving minimal perfect hash table for the given file of keys. The parameter keyFile is the name of a file in the platform-default encoding containing one key on each line. It is assumed that the file does not contain duplicates. Behavior of the constructor when duplicate keys are present is undefined.

The minimal perfect hash table constructed by the constructor provides methods to look up individual keys. These methods return the *ordinal number* of a key—its position in the original order of the key sequence. We focus on only one here. The method:

```
public int getNumber(CharSequence key)
```

returns the ordinal number of the key in the original order that the constructor encountered the keys, beginning at 0. (CharSequence is a Java interface for a readable sequence of char values). This method effectively computes the hash value of the key using the minimal perfect hash function associated with the minimal perfect hash table that was built by the constructor.

For example, suppose that a collection of n records, $R_0 ... R_{n-1}$, each of which is identified by a unique key, is to be used as a static table. First, the n keys, $k_0 ... k_{n-1}$, are extracted from the records into a file (in the indicated order, i.e., k_i is the key of R_i). This file is then used as the parameter for MinimalPerfectHash's constructor, which builds a MinimalPerfectHash table for them. The records are then loaded into an array of size n putting R_i into the array at index i. To look up the data associated with key k, getNumber() is first called with parameter k to find the index of the data which is associated with key k. That index is then used to access the data.

Table construction is purely a preprocessing step. It consists of building the minimal perfect hash table for the keys and loading the data associated with the keys into an array. Once this has been done, use of the table costs $O(1)$-time for each FIND operation.

The MinimalPerfectHash class implements the Serializable interface, so the hash table may be saved and reused later. Consequently, once the hash table has been built it is not necessary to rebuild it unless the key set changes. The data associated with the keys may change during the table's use, so it may be necessary to save that data at termination, but this can be accomplished simply by looping through the array and saving the data in that order.

Other Considerations

It should be noted that in the above, the FIND operation consists of looking a given key up in the MinimalPerfectHash table via getNumber() and then using that result to index into an array. This process assumes that FIND will always be successful—that every key for which the FIND operation is performed is a key which is in the table. As a result, no actual key comparisons are required: given a key, getNumber() computes an index into the data array and the data is then retrieved from the array slot at that index.

If it is necessary to establish whether a given key is or is not in the table, a slightly different approach must be taken because the getNumber() method of MinimalPerfectHash returns a random position if it is called with a key that was not in the original collection.

The SignedMinimalPerfectHash class, which is also a component of MG4J, will always return a usable result, even for keys that are not in the table (here, "signed" refers to "signature" rather than mathematical sign). In a signed minimal perfect hash table, every key has a signature which is used to detect false positives. When getNumber() is called in SignedMinimalPerfectHash, the hash value of the given key is computed as before. Then a comparison is made to determine if the given key has the same signature as the key in the original collection which has the same hash value. If the signatures match, the position of the key is returned as before. If the signatures do not match, getNumber() returns -1.

SignedMinimalPerfectHash provides (via subclasses) considerable flexibility in how signatures can be created and/or evaluated. The additional comparison of signatures may require significant cost at execution time depending on how the comparison takes place. Nevertheless, it is possible to perform the comparison in time proportional to the size of the largest key.

CONCLUSION

To determine which of the techniques to use for constructing a static table, one has to take into consideration the properties of the key set. In particular, if the keys are *not* from a totally ordered set, neither the technique of binary search in a sorted array nor the optimal binary search tree technique can be used because both rely on comparisons yielding order information about the keys. On the other hand, minimal perfect hashing yields an efficient table for key sets regardless of whether they are from totally ordered sets or not.

Binary search in a sorted array guarantees $O(\log n)$ performance in the worst case and is quite easy to set up. The optimal binary search tree technique guarantees minimal average performance and is only slightly harder to set up but requires at least information about the probability of each key being sought. If that information is unavailable, then the technique cannot be used. Both of these techniques are easily programmable in modern programming languages.

The minimal perfect hash table tools described in this chapter (and those referred to in the notes below) are language-specific. As a result, if development is not taking place in Java, C++, or C, extra steps will be necessary to convert the hash table into the development language after it has been built.

Notes on Static Tables

The binary search technique of the first section is nearly folklore in computer science. Knuth (1988) reports that the first published description appeared in 1946 but that the first *correct* description did not appear until 1962!

The optimal binary search tree algorithm given is essentially from Knuth (1971), although the presentation here is adapted from Levitin (2003), with the extension to failed searches adapted from Cormen, Leiserson, Rivest, and Stein (2001). Gilbert and Moore (1959) gave a similar algorithm and discussed it's relationship to other problems. Knuth (1971), in fact, showed how to speed up the algorithm by a factor of n (yielding an $O(n^2)$-time algorithm. The speedup is accomplished by altering control of the innermost loop to:

For each m between $R[i,j\text{-}1]$ and $R[i\text{+}1,j]$.

The justification for this is beyond the scope of this chapter. Interested readers should consult Knuth (1971), Knuth (1988), or Bein, Golin, Larmore, and Zhang (2006) for more information. Subsequently, Vaishnavi, Kriegel, and Wood (1980) and Gotlieb and Wood (1981) investigated adapting the algorithm to multi-way search trees (in which the nodes of the tree may have more than 2 children). In doing so, they showed that the "monotonicity principle," which is responsible for the speed up of Knuth, does not extend to optimal multiway search trees in general. Spuler and Gupta (1992) investigated alternatives to optimal binary search trees in which the average cost of the FIND operation is nearly optimized.

In addition to the MG4J tools described in this chapter, several others exist to generate minimal (and near-minimal) perfect hash tables. GPERF described in Schmidt (1990) and distributed under the Gnu GPL, produces a minimal perfect hash table in the form of C or C++ code. It is distributed via the Free Software Foundation. GGPERF, by Kong (1997) is similar to GPERF but written in Java (GPERF is written in C++ and is also available in C), can produce output in C/C++ and Java, and is faster than GPERF because it is based on the improved algorithm of Czech, Havas, and Majowski (1992). Botelho, Kohayakawa, and Ziviani describe the BMZ algorithm which has appeared in the development of another minimal perfect hash table generator.

REFERENCES

Bein, W., Golin, M., Larmore, L., & Zhang, Y. (2006). The Knuth-Yao quadrangle-inequality speedup is a consequence of total-monotonicity. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 31-40).

Botelho, F., Kohayakawa, Y., & Ziviani, N. (2005). *A practical minimal perfect hashing method*. Paper presented at the 4th International Workshop on Efficient and Experimental Algorithms (WEA05) (vol. 3505, pp. 488-500), Springer-Verlag Lecture Notes in Computer Science.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

Czech, Z., Havas, G., & Majewski, B. (1992). An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters, 43*, 257-264.

Fredman, M. L., Komlòs, J., & Szemerédi, E. (1984). Storing a sparse table with $O(1)$ worst case access time. *Journal of the Association for Computing Machinery, 31*, 538-544.

Gilbert, E., & Moore, E. (1959). Variable-length binary encodings. *Bell System Technical Journal, 38*, 933-967.

Gotlieb, L., & Wood, D. (1981). The construction of optimal multiway search trees and the monotonicity principle. *International Journal of Computer Mathematics, 9*, 17-24.

Knuth, D. E. (1971). Optimal binary search trees. *Acta Informatica, 1*, 14-25.

Knuth, D. (1998). *The art of computer programming: Sorting and searching* (vol. 3) (3rd ed.). Reading, MA: Addison-Wesley.

Kong, J. (1997). *GGPERF: A perfect hash function generator.* Retrieved June 22, 2006, from http://www.cs.ucla.edu/ jkong/public/soft/GGPerf

Levitin, A. (2003). *Introduction to the design and analysis of algorithms*. Boston: Addison-Wesley.

Majewski, B., Wormald, N., Havas, G., & Czech, Z. (1996). A family of perfect hashing methods. *The Computer Journal, 39*, 547-554.

MG4J. (2006). *MinimalPerfectHash documentation page*. Retrieved June 22, 2006, from http://mg4j.dsi.unimi.it/docs/it/unimi/dsi/mg4j/MinimalPerfectHash.html

Schmidt, D. C. (1990, April 9-11). GPERF: A perfect hash function generator. In *Proceedings of the 2nd C++ Conference, USENIX*, San Francisco, (pp. 87-102).

Spuler, D. A., & Gupta, G. K. (1992). *An empirical study of nearly optimal binary search trees and split trees* (Tech. Rep. 92-2). Department of Computer Science, James Cook University.

Vaishnavi, V. K., Kriegel, H. P., & Wood, D. (1980). Optimum multiway search trees. *Acta Informatica, 14*, 119-133.

Yao, A. C. (1981). Should tables be sorted? *Journal of the Association for Computing Machinery, 29*, 615-628.

KEY TERMS

Binary Search: An efficient ($O(\log n)$) search technique for sorted arrays.

Binary Search Tree: A binary tree in which the search order property holds. Binary search trees may be searched by means of a technique similar to binary search in a sorted array.

Hash Table: An indexed data structure in which the index of a key is computed by means of a function called a *hash function*.

Minimal Perfect Hash Table: A hash table in which every key hashes to a unique index in the range $0,...,(n-1)$ for n keys. Minimal perfect hash tables attain theoretically optimal space and time performance.

Optimal Binary Search Tree: A binary search tree in which the average number of comparisons required to search for all keys is minimized.

Pre-Order, Pre-Order Traversal: For any tree, pre-order visits, lists, or processes a node before it visits, lists, or processes the node's children.

Search Order Property: A property of binary trees which facilitates searching. The key at any node compares greater than any key in the node's left sub-tree and less than any key in the node's right sub-tree.

Static Table: A table consisting of keys and associated data for which the set of keys is static (does not change). The data associated with the keys may change, however.

Tail Recursion: A form of recursion in which the last action of a recursive function is a recursive call. Tail recursion can easily be converted to iteration.

Totally-Ordered Set: A set upon which a reflexive, transitive, antisymmetric, relation which also satisfies the trichotomy law is defined.