D
Data Structures

A computer program is a set of instructions (or statements) that specify operations on data items. Sometimes, data items are independent and each is stored separately in memory. In such a case, each item becomes a *variable* in the program and is given a name. A set of data items may, however, be related, and such items should be stored together in memory as a *data structure*. Examples are:

1. A data base with information about people. Each element of the data base may consist of a name (first, middle, and last), an identifying number, an address, and other data such as age, salary, or title. The name may be stored in an array. The identification number becomes an integer variable, the address is stored in another array, and so on. These arrays and variables are then grouped to become one node in a linked list.

2. A matrix of numbers, such as the quantization matrices used in certain image compression methods. Such numbers can be stored in a two-dimensional array.

Arrays and lists are examples of *data structures*. Two things should be described in a data structure, the way data items are related, and the operations that the program should be able to perform on the structure and on the data items. Examples of operations are inserting/deleting an item, replacing an item, searching for an item, and increasing/decreasing the size of the structure. Once the programmer knows how the data items are related and what operations are needed, the data structure can be designed. Data structures are sometimes simple, such as an array, a stack, or a list, but they can get very complex, since one structure may combine lists, stacks, hash tables, and arrays in a complex way.

The main data structure described in this chapter is the hash table, but we start with a short survey of some basic data structures.

D.1 Arrays

The array is a common, useful, and important data structure. Intuitively, we can think of an array as a set of consecutive memory locations grouped under one name, where each individual location is accessed by its *index*. While this is a practical description of arrays, it is not a good definition, since it defines this data structure by means of its actual representation in memory. A more formal definition is

An array is a set of pairs (index, value).

This defines an array as a mapping from the set of indexes to the set of values.

In principle, the index can be any data type, but in practice it is normally an integer. In the old FORTRAN programming language the first array element has index 1. In the C language the first index is 0, and in Pascal the programmer can specify the first array index. Most programming languages require the size of all arrays to be static, i.e., the size of an array cannot be changed at run time.

A program may use many data items, and storing them in an array is convenient, since the programmer has to memorize just one name, the name of the array, for all these items. However, each item stored in an array has an index, and the programmer either has to memorize the indexes, use an application where it is not necessary to memorize each index, or search the array for any particular item.

An important feature of arrays is that they can have more than one dimension. A two-dimensional array $A[m,n]$ is a matrix. Similarly, a three-dimensional array $A[l,m,n]$ is a three-dimensional matrix. Alternatively, we can think of it as a one-dimensional array of l elements, each a matrix of dimensions $m\times n$.

Even though a two-dimensional array can be considered a matrix, it is physically stored in memory as a set of consecutive locations. The array can be stored in memory row by row or column by column. Figure D.1 shows that in the former case, an array item $A[i,j]$ in an $m\times n$ array is stored in location $(i-1)n+j$ from the start of the array (assuming that row and column indexes start at 1).

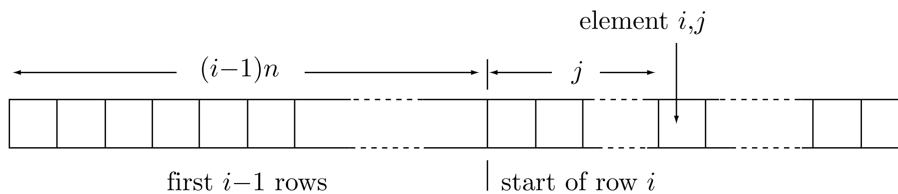

Figure D.1: Position of Array Element $A[i,j]$.

◇ **Exercise D.1:** Assume that an array A of dimensions $m\times n$ is stored in memory column by column and that row and column indexes start at 0. What is the distance, in memory, of array element $A[i,j]$ from the first array element $A[0,0]$?

A simple example of the use of an array is storing the $n+1$ coefficients of a polynomial of degree n in an array of size $n+2$ or longer (see Equation (I.18) for the definition of a polynomial). In this case the programmer implicitly knows what's in each array location.

◇ **Exercise D.2:** Why should the array size be $n + 2$ and not $n + 1$?

D.2 Stacks and Queues

The stack is also a common data structure. Intuitively we think of a stack as a container open at one end. The only operations on a stack are insertion and deletion (commonly referred to as *push* and *pop*, respectively), and they are done from the open end. Figure D.2a shows how three data items are pushed into an empty stack and how the only item that can be popped out is the last that was pushed in. This is why a stack is sometimes referred to as a LIFO (last-in first-out) structure. The stack is normally implemented as an array, but because of the way it is used, the program needs to keep track of only one item, the top one (i.e., the last one in), at any given time. The program therefore maintains a pointer called *top of stack* that points at that data item.

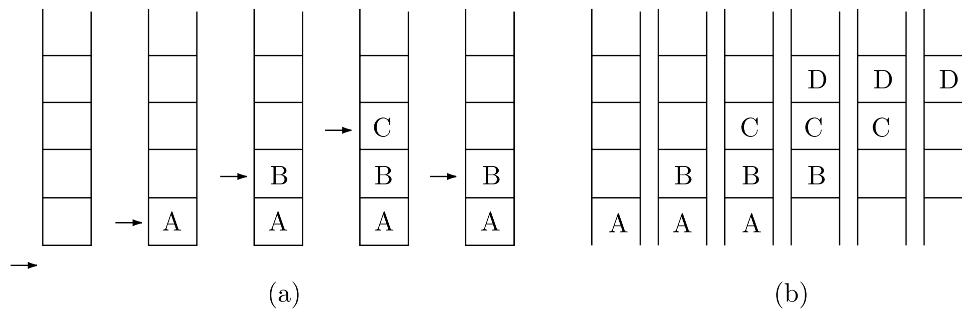

Figure D.2: Stack and Queue.

A *queue* is a data structure where the only item that can be removed is the oldest one. A queue is therefore based on the FIFO (first-in first-out) principle. Figure D.2b shows a queue during four insertions and three deletions. It is obvious that these operations move the data from the start to the end of the queue, which is why the *circular queue* (discussed in Section 3.2.1) is a more useful data structure than the linear queue.

D.3 Lists

A list (or a linked list) is a data structure made of nodes that point to each other. A node may be a single variable, an array, a stack, another list, or any other structure, but the main feature of lists is the use of *pointers*. This makes it easy to control the size of the list dynamically. A programming language may include statements such as `get_node` (to construct a new node from the pool of available storage) and `put_node` (to return the storage used by a node to that pool). Figure D.3 shows several ways to organize lists. A list can be singly-linked or doubly-linked, it can be cyclical, and the list elements may themselves be lists.

A queue may be implemented as a linked list. Inserting a new item is done by creating a new node and adding it to the list. Deleting an item is done by deleting the first (oldest) node.

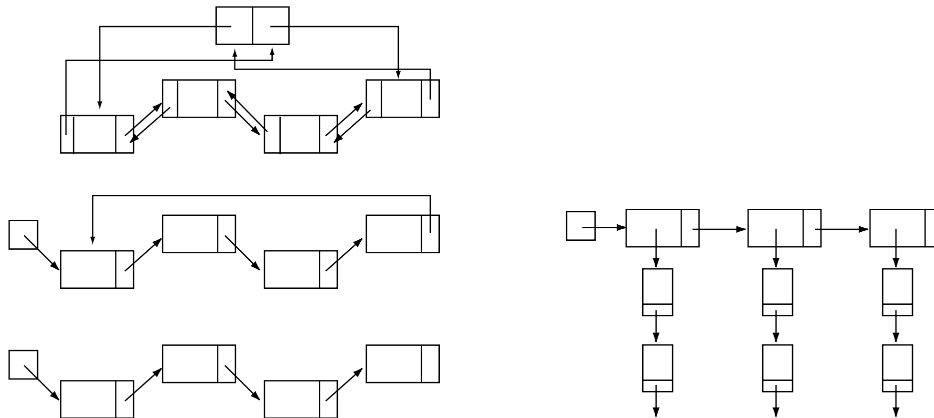

Figure D.3: Linked Lists.

D.4 Trees

A tree can be loosely defined as a data structure consisting of nodes connected with directed edges, where all the nodes are connected, there are no cycles, and one node is considered special. This node is called the *root* of the tree. Figure D.4a shows such a structure where any of the nodes can be considered the root. Notice the dashed edge. Including this edge in the structure would introduce a cycle, and thus change it from a tree to a general graph. Figure D.4b shows the same structure with node a chosen as the root and the remaining nodes rearranged to form the familiar shape of a tree. If there are edges leading directly from a node a to nodes b, c, and d, then the three nodes are called the *children* of a and a is their *parent.* If a node does not have any children, it is a *leaf.* A node that is neither a leaf nor the root is an *interior* node. The *depth* (or level) of a node is the length of the path from the root to the node. The root itself has depth zero. The height of a tree is the largest depth (or, alternatively, one less than the number of levels of the tree). Figure D.4b shows that each node in a tree is the root of a subtree (which may be empty).

In a typical practical implementation, a node a is a short array containing a data item and pointers. In most cases the array contains 1, 2, or 3 pointers, each pointing to the start of a linked list. The first pointer may point to a list containing nodes that are siblings of a (notice that the root does not have any siblings). The second pointer may point to a list containing nodes that are children of a (if a is a leaf, this pointer is null), and the third pointer may point to the parent of a (a null pointer, if a is the root). This way it is possible to travel in the tree to the right (from a to its next sibling on the right), down (from a to its first child), and up (from a to its parent). Such a tree may also change dynamically, with nodes being added, deleted, and modified.

Sometimes, it is useful to add another component (or field) to the array, with a code marking the node as either existing or deleted. This way a node can be

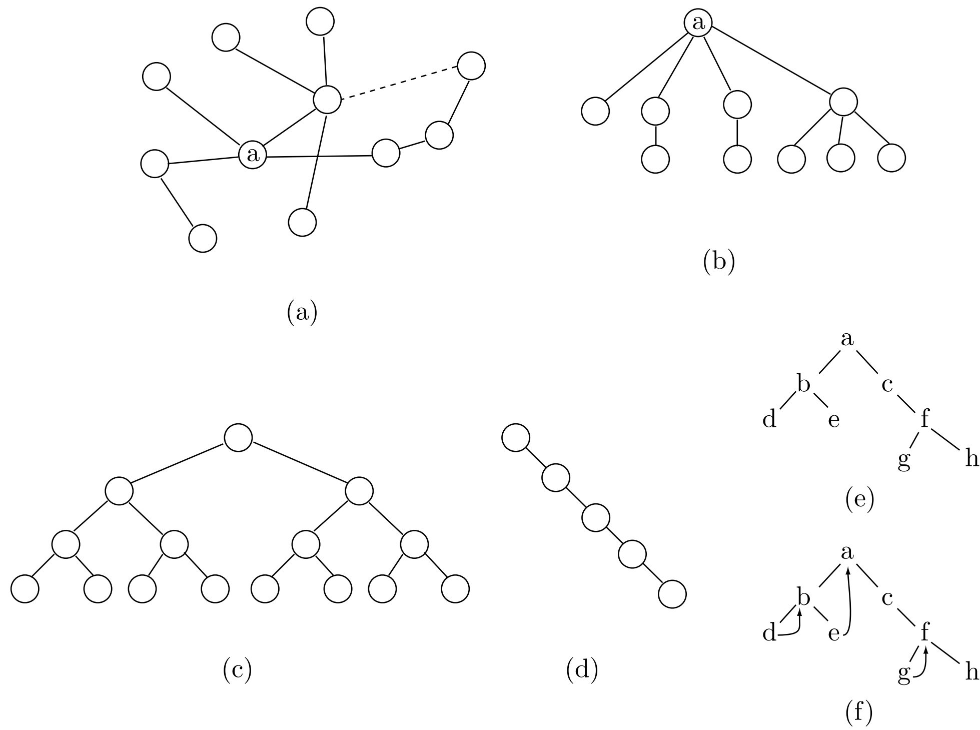

Figure D.4: Various Trees.

effectively deleted from the tree by setting this field to "deleted," without having to actually delete it and change pointers. This technique is called "lazy deletion" and is useful in applications where there is no need to return deleted nodes to the pool of available storage.

If a node can have just a few children, it may be possible to implement the entire tree in an array, without any pointers. The simplest example is a complete binary tree, where the two children of node a are stored in locations $2a$ and $2a+1$ and the parent of a can be found at array location $\lfloor a/2 \rfloor$ (this is discussed in Section 2.15).

◇ **Exercise D.3:** Can a ternary tree be implemented in this way?

A binary tree can be complete, skewed, or anything in between. This is illustrated by Figure D.4c,d,e.

An important operation on a tree is a *traversal*. A traversal follows pointers in such a way that each node of the tree is visited once. There are four types of tree traversals:

Post-order. All the children of a node are visited, then the node itself is visited. This is done recursively by the two recursive calls `postorder(L)`; `postorder(R)`; (where L and R are the two children of the root), followed by `visit(root)`. The post-order traversal of the tree of Figure D.4e is $(((D,E),B),(((G,H),F),C)),A$.

Pre-order: Visit a node, then all its children.

In-order: This is for binary tree traversal. Visit the left subtree of a node, then the node, then its right subtree.

Level-order: Visit all the nodes of level L, then proceed to level $L+1$.

⋄ **Exercise D.4:** Show the pre-order, in-order, and level-order traversals of the tree of Figure D.4e.

A tree traversal is normally done recursively, except that level-order traversal uses a queue instead of a stack.

In a simple implementation, each node of a binary tree has two pointers, for its left and right children. It is obvious that not all the pointers are used. For example, in the binary tree of Figure D.4e there are 9 unused pointers (8 in the leaves and one in node C). Such unused pointer fields can be used for extra pointers called *threads*. One way to define threads is to find a node A with an unused field of a right child, and to store in this field a pointer to the successor of A in inorder traversal. The resulting three threads of the tree of Figure D.4e are shown in Figure D.4f. Obviously, threads can be defined differently and can be very useful in special applications, where the tree has to be traversed or searched in nonstandard ways. The price of adding the threads is an extra bit (a flag) in each node A, indicating whether an ordinary pointer to the right child, or a thread is stored in A.

Imagine an interior node a in a binary tree. It has two children, l and r, that are the roots of the left and right subtrees of a, respectively. If l has the largest data value in its subtree, and r satisfies the same thing for its subtree, then the binary tree is called a *heap*.

A binary search tree is an important type of tree. In such a tree all nodes in the left subtree of node a have data values that are smaller than the data value of a. Similarly, all the nodes in the right subtree of a have data values larger than a. If the data are not numbers, the relations "less than" and "greater than" have to defined. Binary search trees are described and used in Section 3.3. The main use of such a tree is quick search. Searching for a node in a binary search tree takes at most H steps, where H is the height of the tree.

Imagine a binary search tree that starts empty. When nodes are inserted into the tree, it grows. The order in which the nodes are inserted determines the shape of the tree. If the nodes being inserted have random data values, the resulting tree will be balanced, i.e., very similar to a complete tree. Its height will be approximately $\log_2 n$ where n is the number of nodes. Searching such a binary search tree with a million nodes takes at most 20 steps. If, on the other hand, the new nodes have monotonous data values (ascending or descending), the tree will end up being skewed. Its height (and thus the maximum search time) will be n.

Binary search trees are commonly used in applications that require many searches, so it is desirable to find a way to keep such a tree as close to balanced as possible, regardless of the order of node insertions. The AVL tree (named for its two developers, Adelson-Velskii and Landis) is such a data structure. Each node in an AVL tree has a *balance factor*, defined as the height of its left subtree minus the height of its right subtree. This balance factor is allowed to take only the three values 0, 1, and -1, i.e., the heights of the two subtrees of any node may differ by at most 1.

When the balance factor of a node becomes greater than 1, we say that the node is "out of balance on the left;" when it becomes less than −1, we say it is "out of balance on the right." These situations are immediately corrected, and the node brought back into balance. An AVL tree is a special case of a *height-balanced binary tree*, a structure defined by:

1. An empty binary tree is height balanced.
2. A nonempty binary tree is height balanced if its left and right subtrees are height balanced with balance factors of 0, 1, or −1.

An AVL tree is a height-balanced binary *search* tree, a special case of a height-balanced binary tree. Figure D.5a shows an example of an AVL tree that illustrates one important attribute of these trees, namely they don't have to be complete. The leaves of an AVL tree don't have to be on the same level or even on adjacent levels. However, it can be shown that an AVL tree is not very different from a complete binary tree because its height can be at most $\sqrt{2}\log_2 n$. An AVL tree is created either empty of with 1 node, so initially it is height-balanced. It is kept height-balanced after each insertion and deletion by special adjustments (called *rotations*) that restore the balance. The details can be found in texts on data structures.

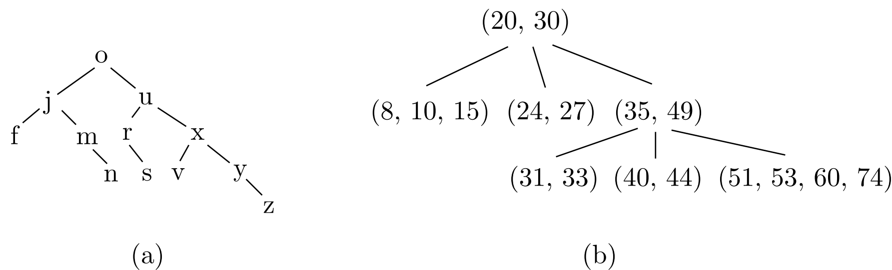

Figure D.5: (a) An AVL Tree. (b) A B-Tree.

The last type of tree to be mentioned here is the B-tree. This is an important type of tree because it is used by many operating systems to maintain the file directory of a disk. The growth of a B-tree is especially controlled to keep it well-balanced, leading to fast searches, but a B-tree, in spite of its name, is not a binary tree.

A node in a B-tree can have several children, and it is this property, together with the tree being well-balanced, that makes it a natural candidate for a disk directory. The following extreme example shows why. Imagine a tree where each node can have 100 children. If the tree has a height of 4, it can contain up to

$$1 + 100 + 100^2 + 100^3 = 1,010,101$$

nodes. One node among more than a million can be found in at most four steps! The price for this is, of course, a complex node structure, allowing for up to 100 children. In a disk directory each node is kept on the disk as a block. The size of a disk block varies, but is typically a few hundred bytes. Therefore, a block has

room for much information in the form of keys, pointers, and flags. At the same time, a disk access is much slower than memory access, so finding an item in the directory should involve as few disk accesses as possible. Once the disk is accessed, an entire block is read into memory. In memory, the block can be searched and its data processed quickly.

A node in a B-tree contains a count field, a pointer to a list of entries (records to be searched, where each record has a key and possibly some other data), and a pointer to a list of branches. The lists of entries and branches must be ordered, but they can be any ordered structures, such as linked lists, arrays, or binary search trees. The count field m contains the number of entries (m) and the number of branches ($m+1$).

Figure D.5b shows a simple B-tree. Only the list of entries is shown, and for each entry only the key is shown. This is enough to understand how the tree is searched. The root contains keys 20 and 30. This means that branch 0 from the root leads to an entry list for all entries with keys that are less than 20. Branch 1 from the root leads to an entry list for all entries with keys in the range $[20, 30)$ and branch 2 leads to the list for entries with keys ≥ 30. To search for 34, for example, we start at the root, take branch 2 (since $33 \geq 30$), then branch 0 (since $33 < 35$). We arrive at the node with entry list $(31, 33)$ and take branch 2 (since $33 \geq 35$). This brings us to a null node, which is how we discover that key 34 is not in the tree.

D.5 Graphs

A graph is a general data structure consisting of nodes (or vertices) and edges (or arcs) connecting them. The graph is a general structure because not all nodes have to be connected and no node has to be special (such as the head or root of the graph). Figure D.6 shows examples of graphs. A graph may even consist of several, disconnected units. Edges may be directed or undirected, and may have labels (indicating weights or costs) associated with them. The main operations on graphs are the following:

1. Construct an empty graph g.
2. Insert a node a into an existing graph g.
3. Construct an edge e between nodes a and b of graph g.
4. Delete a node a from g. All the edges adjacent to the node should also be deleted.
5. Delete edge e between nodes a and b of graph g.
6. Determine whether graph g is empty.
7. Construct a list of all edges adjacent to node a of g.
8. Traverse graph g (i.e., visit each node once).
9. Find the minimum-cost path that leads from node a to node b in g.

There may be other, specialized operations for specific applications.

In an undirected graph, an edge between a and b is adjacent to both nodes. In a directed graph, an edge from a to b is said to be "adjacent from a" and "adjacent to b."

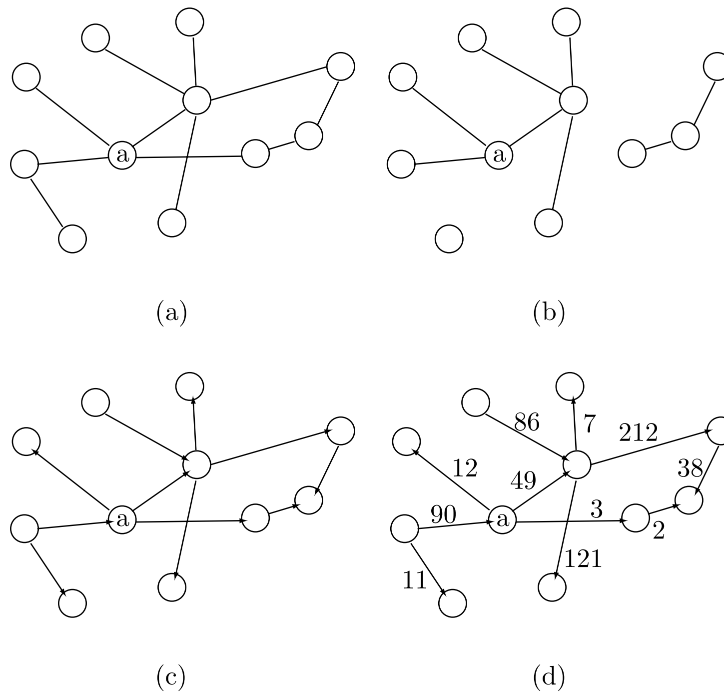

Figure D.6: Various Graphs.

D.6 Hashing

A hash table is a data structure allowing for fast insertions, searches, and deletions of data items. The table itself is just an array H, and the principle of hashing is to define a function h such that $h(k)$ produces an index to array H, where k is the key of a data item. The following examples illustrate the meaning of the terms "data item" and "key."

1. The LZRW1 method (Section 3.8), uses hashing to store pointers. The method uses the first three characters in the look-ahead buffer as a key which is hashed into a 12-bit number I used to index the hash table, an array of $2^{12} = 4,096$ pointers. The actual data stored in each location of the LZRW1 hash table is a pointer.

2. Virtually all computer languages use variables. A variable provides a name for a value that will be stored in memory, in a certain address A, when the program is eventually executed. When the program is compiled, each variable has two attributes, its name N (a string of characters assigned by the programmer) and its memory address A, assigned by the compiler. The compiler uses a hash table to store all the information about variables. The data item in this example is the address A of a variable; the key is the variable's name N. The compiler reads the name from the program source file, hashes it, finds it in the hash table, and retrieves the address in order to compile the current instruction. If the variable is not found

in the hash table, it is assigned an address, and both the name and address are stored in the table (in principle, only the address need be stored, but the name is also stored because of collisions; see below).

The hash function h takes as argument a key, which may be a number or a string. It scrambles or hashes the bits of the key to produce an index to array H. In practice the array size is normally 2^n, so the result produced by h should be an n-bit number. Hashing is a good data structure, since any operation on the hash table, adding, searching, or deleting, can be done in one step, regardless of the table size. The only problem is *collisions*. In most applications it is possible for two distinct keys k_1 and k_2 to get hashed to the same index, i.e., $h(k_1) = h(k_2)$ for $k_1 \neq k_2$. Example 2 above makes it easy to understand the reason for this. Assuming variable names of five letters, there may be $26^5 = 11,881,376$ variable names. Any program would use just a small percentage of this number, perhaps a few hundred or a few thousand names. The size of the hash table thus doesn't have to exceed a few thousand entries, and hashing 11.8 million names into a few thousand index values must involve many collisions.

◇ **Exercise D.5:** How many names are possible if a name consists of exactly eight letters and digits?

Terminology: Two different keys that hash to the same index are called *synonyms*. If a hash table contains m keys out of a set of M possible ones, then m/M is the *density* of the table, and $\alpha = m/2^n$ is its *loading factor*.

D.7 Hash Functions

A hash function should be easy to compute and should minimize collisions. The function should make use of all the bits of the key, such that changing even one bit would normally (although not always) produce a different index. An ideal hash function should also produce indexes that are uniformly distributed (calling the function many times with random keys should produce each index the same number of times). A function that produces, e.g., index 118 most of the time is obviously biased and leads to collisions. The function should also assume that many keys may be similar. In the case of variable names, e.g., programmers sometimes assign names such as "`A1`", "`A2`", "`A3`" to variables. A hash function that uses just the leftmost bits of a key would produce the same index for such names, leading to many collisions. Following are some examples of hash functions used in practice.

Mid-Square: The key k is considered an integer, it is squared and the middle n bits of k^2 extracted to become the index. Squaring k has the advantage that the middle bits of k^2 depend on *all* the bits of k. Thus two keys differing in one bit would tend to produce different indexes. A variation, suitable for large keys, is to divide the bits of the original key into several groups, add all the groups, square the result, and extract its middle n bits.

The keys "`A1`", "`A2`", and "`A3`", e.g., become the 16-bit numbers

$$01000001|00110001, \quad 01000001|00110010, \quad \text{and } 01000001|00110011.$$

After squaring and extracting the middle 8 bits, the resulting indexes are 158, 166, and 175, respectively.

Modulo: $h(k) = k \bmod m$. The result is the remainder of the integer division k/m, a number in the range $[0, m-1]$. In order for the result to be a valid index, the hash table size should be m. The value of m is critical and should be selected carefully. If m is a power of 2, say, 2^i, then the remainder of k/m is simply the i rightmost bits of k. This would be a very biased hash function. If m is even, then the remainder of k/m has the same parity as k (it is odd when k is odd and even when k is even). This again is a bad choice for m, since it produces a biased hash function that maps odd keys to odd location of H and even keys to even locations. If p is a prime number evenly dividing m then keys that are permutations of each other (e.g., "`ABC`", "`ACB`", and "`CBA`") may often be mapped to indexes that differ by p or by a multiple of p, again causing non-uniform distribution of the keys.

It can be shown that the modulo hash function achieves best results when m is a prime number that does not evenly divide $8^a \pm b$ where a and b are small numbers. In practice, good choices for m are prime numbers whose prime divisors are > 20.

Folding: This function is suitable for large keys. The bits constituting the key are divided into several groups, which are then added. The middle n bits of the sum are extracted to become the index. A variation is *reverse folding* where every other group of bits is reversed before being added.

> No, no. Look, here's the hash on the side because I didn't know how much you took.
>
> —Amy Wright as Shelley in *Stardust Memories* (1980).

D.8 Collision Handling

When an index i is produced by the hash function $h(k)$, the software using hashing should first check $H[i]$ for a collision. There must, therefore, be a way for the software to tell whether entry $H[i]$ is empty or occupied. Initializing all entries of H to zero is normally not enough, since zero may be a valid data item. A simple approach is to have an additional array F, of size $2^n/8$ bytes, where each bit is associated with an entry of H. Each bit of F acts as a flag indicating whether the corresponding entry of H is empty or not. The entire array F is initially set to zeros, indicating that all entries of H are empty. When the software decides to insert a data item in $H[i]$ it has to find the bit in F that corresponds to entry i and check it. The software should therefore calculate $j = \lfloor i/8 \rfloor$, $k = i - 8j$, and check bit k of byte F[j]. If the bit is zero, entry H[i] is empty and can be used for a new data item. The bit then has to be set, which is done by using k to select one of the eight masks

00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

and logically OR it with $F[j]$. If the bit is 1, entry $H[i]$ is already occupied, and this is a collision. The software should be able to check and tell whether entry $H[i]$ contains the data item d that corresponds to key k. This is why the keys have to be saved, together with the data items, in the hash table.

What should the software do in case of a collision? The simplest thing is to check entries $H[i+1]$, $H[i+2]$,..., $H[2^n-1]$, $H[0]$, $H[1]$... until an empty entry is found or until the search reaches entry $H[i-1]$. In the latter case the software knows

that the data item is not in the table (if this was a search for an item) or that the table is full (if this was an attempt to insert a new item in the table). This process is called *linear search*. Searching for a data item, which in principle should take one step, can now, because of collisions, take up to 2^n steps. Experience also shows that a linear search causes occupied entries in the table to cluster, which is intuitively easy to understand. If the hash function is not ideal and hashes many keys to, say, index 54, then table entries 54, 55,... will quickly fill up, creating a cluster. Clusters also tend to grow and merge, creating even larger clusters and thereby increasing the search time. A theoretical analysis shows that the expected number of steps needed to locate an item when linear search is used is $(2-\alpha)/(2-2\alpha)$, where α is the loading factor (percent full of the table). For $\alpha = 0.5$ we can expect 1.5 steps on the average, but for $\alpha = 0.75$ the expected number of steps rises to 2.5, and for $\alpha = 0.9$ it become 5.5. It is clear that when linear search is used, the loading factor should be kept low (perhaps below 0.6–0.7). If more items need to be added to the table, a good solution is to declare a new table, twice as large as the original one, transfer all items from the old table to the new one (using a new hash function), and delete the old table.

> Dinner was at one o'clock; and on Monday, Tuesday, and Wednesday it consisted of beef, roast, hashed, and minced, and on Thursday, Friday, and Saturday of mutton. On Sunday they ate one of their own chickens.
>
> —W. Somerset Maugham, *Of Human Bondage*

A more sophisticated method of handling collisions is *quadratic search*. Assume that H is an array of size N. When entry $H[i]$ is found to be occupied, the software checks entries $H[(i \pm j^2) \bmod N]$ where $0 \le j \le (N-1)/2$. It can be shown that if N is a prime number of the form $4j+3$ (where j is an integer) quadratic search will end up examining every entry of H.

A third way to treat collisions is to rehash. The software should have a choice of several hashing functions $h_1, h_2, \ldots$. If $i = h_1(k)$ and $H[i]$ is occupied, the software should calculate $i = h_2(k)$ then try the new $H[i]$. Still another way is to generate an array R of N pseudo-random numbers in the range $[0, N-1]$ where each number appears once. If entry $H[i]$ is occupied, the software should set $i = (i + R[i]) \bmod N$ and try the new $H[i]$.

It is possible to design a *perfect hash function* that, for a given set of data items, will not have any collisions. This makes sense for sets of data that never change. Examples are the Bible, the works of Shakespeare, or any data written on a CD-ROM. The size N of the hash table should, in such a case, be normally larger than the number of data items. It is also possible to design a *minimal perfect hash function* where the hash table size equals the size of the data (i.e., no entries remain empty after all data items have been inserted). See [Czech 92], [Fox 91], and [Havas 93] for details on these special hash functions.

Bibliography

Czech, Z. J., et al. (1992) "An Optimal Algorithm for Generating Minimal Perfect Hash Functions," *Information Processing Letters* **43**:257–264.

Fox, E. A. et al. (1991) "Order Preserving Minimal Perfect Hash Functions and Information Retrieval," *ACM Transactions on Information Systems* **9**(2):281–308.

Havas, G. et al. (1993) *Graphs, Hypergraphs and Hashing* in Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93), Berlin, Springer-Verlag.

"Why," said he, "a magician could call up a lot of genies, and they would hash you up like nothing before you could say Jack Robinson. They are as tall as a tree and as big around as a church."

Mark Twain, *The Adventures Of Huckleberry Finn*