7

Searching

In this chapter we consider methods of searching large amounts of data to find one particular piece of information. As we shall see, certain methods of organizing data make the search process more efficient. Since searching is such a common task in computing, a knowledge of these methods goes a long way toward making a good programmer.

7.1 BASIC SEARCH TECHNIQUES

Before we consider specific search techniques, let us define some terms. A ***table*** or a ***file*** is a group of elements, each of which is called a ***record***. Associated with each record is a ***key***, which is used to differentiate among different records. The association between a record and its key may be simple or complex. In the simplest form, the key is contained within the record at a specific offset from the start of the record. Such a key is called an ***internal key*** or an ***embedded key***. In other cases there is a separate table of keys that includes pointers to the records. Such keys are called ***external***.

For every file there is at least one set of keys (possibly more) that is unique (that is, no two records have the same key value). Such a key is called a ***primary key***. For example, if the file is stored as an array, the index within the array of an element is a unique external key for that element. However, since any field of a record can serve as the key in a particular application, keys need not always be unique. For example, in a file of names and addresses, if the state is used as the key for a particular search,

it will probably not be unique, since there may be two records with the same state in the file. Such a key is called a ***secondary key***. Some of the algorithms we present assume unique keys; others allow for duplicate keys. When adopting an algorithm for a particular application the programmer should know whether the keys are unique and make sure that the algorithm selected is appropriate.

A ***search algorithm*** is an algorithm that accepts an argument a and tries to find a record whose key is a. The algorithm may return the entire record or, more commonly, it may return a pointer to that record. It is possible that the search for a particular argument in a table is unsuccessful; that is, there is no record in the table with that argument as its key. In such a case the algorithm may return a special "null record" or a null pointer. Very often, if a search is unsuccessful it may be desirable to add a new record with the argument as its key. An algorithm that does this is called a ***search and insertion*** algorithm. A successful search is often called a ***retrieval***. A table of records in which a key is used for retrieval is often called a ***search table*** or a ***dictionary***.

In some cases it is desirable to insert a record with a primary key *key* into a file without first searching for another record with the same key. Such a situation could arise if it has already been determined that no such record already exists in the file. In subsequent discussions we investigate and comment upon the relative efficiency of various algorithms. In such cases the reader should note whether the comments refer to a search, to an insertion, or to a search and insertion.

Note that we have said nothing about the manner in which the table or file is organized. It may be an array of records, a linked list, a tree, or even a graph. Because different search techniques may be suitable for different table organizations, a table is often designed with a specific search technique in mind. The table may be contained completely in memory, completely in auxiliary storage, or it may be divided between the two. Clearly, different search techniques are necessary under these different assumptions. Searches in which the entire table is constantly in main memory are called ***internal searches***, whereas those in which most of the table is kept in auxiliary storage are called ***external searches***. As with sorting, we concentrate primarily on internal searching; however, we mention some techniques of external searching when they relate closely to the methods we study.

Dictionary as an Abstract Data Type

A search table or a dictionary can be presented as an abstract data type. We first assume two type declarations of the key and record types and a function that extracts the key of a record from the record. We also define a null record to represent a failed search.

```
typedef   KEYTYPE ...      /* a type of key      */
typedef   RECTYPE ...      /* a type of record   */
RECTYPE   nullrec = ...    /* a "null" record    */

KEYTYPE   keyfunct(r)
RECTYPE   r;
{...
};
```

We may then represent the abstract data type *table* as simply a set of records. This is our first example of an ADT defined in terms of a set rather than a sequence. We use the notation [*eltype*] to denote a set of objects of the type *eltype*. The function *inset*(*s*, *elt*) returns *true* if *elt* is in set *s* and *false* otherwise. The set operation $x - y$ denotes the set x with all elements of set y removed.

```
abstract typedef [rectype] TABLE (RECTYPE);

abstract  member(tbl,k)
TABLE(RECTYPE) tbl;
KEYTYPE k;
postcondition if (there exists an r in tbl such that
                                              keyfunct(r) ==  k)
                then member = TRUE
                else member = FALSE

abstract RECTYPE search(tbl,k)
TABLE(RECTYPE) tbl;
KEYTYPE k;
postcondition (not member(tbl, k)) && (search == nullrec)
                || (member(tbl,k) && keyfunct(search) == k);

abstract  insert(tbl,r)
TABLE(RECTYPE) tbl;
RECTYPE r;
precondition member(tbl,keyfunct r') == FALSE
postcondition inset(tbl', r);
              (tbl - [r]) == tbl';

abstract delete(tbl, k)
TABLE(RECTYPE) tbl;
KEYTYPE k;
postcondition tbl == (tbl' - [search(tbl,k)]);
```

Because no relation is presumed to exist among the records or their associated keys, the table that we have specified is called an ***unordered table***. Although such a table allows elements to be retrieved based on their key values, the elements cannot be retrieved in a specific order. There are times when, in addition to the facilities provided by an unordered table, it is also necessary to retrieve elements based on some ordering of their keys. Once an ordering among the records is established, it becomes possible to refer to the first element of a table, the last element of a table, and the successor of a given element. A table that supports these additional facilities is called an ***ordered table***. The ADT for an ordered table must be specified as a sequence to indicate the ordering of the records rather than as a set. We leave the ADT specification as an exercise for the reader.

Algorithmic Notation

Most of the techniques presented in this chapter are presented as algorithms rather than as C programs. The reason for this is that a table may be represented in a wide

variety of ways. For example, a table (keys plus records) organized as an array might be declared by

```
#define TABLESIZE 1000
typedef KEYTYPE ...
typedef RECTYPE ...
struct {
  KEYTYPE k;
  RECTYPE r;
} table[TABLESIZE];
```

or it might be declared as two separate arrays:

```
KEYTYPE k[TABLESIZE];
RECTYPE r[TABLESIZE];
```

In the first case the ith key would be referenced as *table*[i].k; in the second case, as $k[i]$.

Similarly, for a table organized as a list, either the array representation of a list or the dynamic representation of a list could be used. In the former case, the key of the record pointed to by a pointer p would be referenced as *node*[p].k; in the latter case, as $p \rightarrow k$.

However, the techniques for searching these tables are very similar. Thus, in order to free ourselves from the necessity of choosing a specific representation, we adopt the algorithmic convention of referencing the ith key as $k(i)$ and the key of the record pointed to by p as $k(p)$. Similarly, we reference the corresponding record as $r(i)$ or $r(p)$. In this way we can focus on details of technique rather than on details of implementation.

Sequential Searching

The simplest form of a search is the *sequential search*. This search is applicable to a table organized either as an array or as a linked list. Let us assume that k is an array of n keys, $k(0)$ through $k(n-1)$, and r an array of records, $r(0)$ through $r(n-1)$, such that $k(i)$ is the key of $r(i)$. (Note that we are using the algorithmic notation, $k(i)$ and $r(i)$, as described previously.) Let us also assume that *key* is a search argument. We wish to return the smallest integer i such that $k(i)$ equals *key* if such an i exists and -1 otherwise. The algorithm for doing this is as follows:

```
for (i = 0; i < n; i++)
  if (key == k(i))
    return(i);
return (-1);
```

The algorithm examines each key in turn; upon finding one that matches the search argument, its index (which acts as pointer to its record) is returned. If no match is found, -1 is returned.

This algorithm can be modified easily to add a record *rec* with key *key* to the table if *key* is not already there. The last statement is modified to read

```
k(n) = key;      /* insert the new key and  */
r(n) = rec;      /*         record          */
n++;             /* increase the table size */
return(n - 1);
```

Note that if insertions are made using the foregoing revised algorithm only, no two records can have the same key. When this algorithm is implemented in C, we must ensure that incrementing *n* does not make its value go beyond the upper bound of the array. To use a sequential insertion search on an array, sufficient storage must have been previously allocated for the array.

An even more efficient search method involves inserting the argument key at the end of the array before beginning the search, thus guaranteeing that the key will be found.

```
k(n) = key;
for (i = 0; key != k(i); i++)
   ;
if (i < n)
   return(i);
else
   return(-1);
```

For a search and insertion, the entire *if* statement is replaced by

```
if (i == n)
  r(n++) = rec;
return(i);
```

The extra key inserted at the end of the array is called a ***sentinel.***

Storing a table as a linked list has the advantage that the size of the table can be increased dynamically as needed. Let us assume that the table is organized as a linear linked list pointed to by *table* and linked by a pointer field *next*. Then assuming *k*, *r*, *key*, and *rec* as before, the sequential insertion search for a linked list may be written as follows:

```
q = null;
for (p = table; p != null && k(p) != key; p = next(p))
   q = p;
if (p != null)        /* this means that k(p) == key */
  return (p);
/*  insert a new node  */
s = getnode();
k(s) = key;
r(s) = rec;
next(s) = null;
if (q == null)
   table = s;
else
   next(q) = s;
return(s);
```

The efficiency of searching a list can be improved by the same technique jus. suggested for an array. A sentinel node containing the argument key can be added to the end of the list before beginning the search so that the condition in the *for* loop is the simple condition $k(p)$!= *key*. The sentinel method, however, requires maintaining an additional external pointer to the last node in the list. We leave additional details (as, for example, what happens to the newly added node when the key is found within the list) to the reader.

Deleting a record from a table stored as an unordered array is implemented by replacing the record to be deleted with the last record in the array and reducing the table size by 1. If the array is ordered in some way (even if the ordering is not by key), this method cannot be used, and half the elements in the array must be moved on the average. (Why?) If the table is stored as a linked list, it is quite efficient to delete an element regardless of the ordering.

Efficiency of Sequential Searching

How efficient is a sequential search? Let us examine the number of comparisons made by a sequential search in searching for a given key. We assume no insertions or deletions, so that we are searching through a table of constant size n. The number of comparisons depends on where the record with the argument key appears in the table. If the record is the first one in the table, only one comparison is performed; if the record is the last one in the table, n comparisons are necessary. If it is equally likely for the argument to appear at any given table position, a successful search will take (on the average) $(n + 1)/2$ comparisons, and an unsuccessful search will take n comparisons. In any case, the number of comparisons is $O(n)$.

However, it is usually the case that some arguments are presented to the search algorithm more often than others. For example, in the files of a college registrar, the records of a senior who is applying for transcripts for graduate school, or of a freshman whose high school average is being updated, are more likely to be called for than those of the average sophomore and junior. Similarly, the records of scofflaws and tax cheats are more likely to be retrieved from the files of a motor vehicles bureau or the Internal Revenue Service than those of a law-abiding citizen. (As we shall see later in this chapter, these examples are unrealistic because it is unlikely that a sequential search would be used for such large files; but for the moment, let us assume that a sequential search is being used.) Then if frequently accessed records are placed at the beginning of the file, the average number of comparisons is sharply reduced, since the most commonly accessed records take the least amount of time to retrieve.

Let $p(i)$ be the probability that record i is retrieved. ($p(i)$ is a number between 0 and 1 such that if m retrievals are made from the file, $m * p(i)$ of them will be from $r(i)$.) Let us also assume that $p(0) + p(1) + \cdots + p(n - 1) = 1$, so that there is no possibility that an argument key is missing from the table. Then the average number of comparisons in searching for a record is

$$p(0) + 2 * p(1) + 3 * p(2) + \cdots + n * p(n - 1)$$

Clearly, this number is minimized if

$$p(0) >= p(1) >= p(2) >= \cdots >= p(n - 1)$$

(Why?) Thus, given a large stable file, reordering the file in order of decreasing probability of retrieval achieves a greater degree of efficiency each time that the file is searched.

If many insertions and deletions are to be performed on a table, a list structure is preferable to an array. However, even in a list it would be better to maintain the relationship

$$p(0) >= p(1) >= \cdots >= p(n-1)$$

to provide for efficient sequential searching. This can be done most easily if a new item is inserted into the list at its proper place. If *prob* is the probability that a record with a given key is the search argument, that record should be inserted between records $r(i)$ and $r(i+1)$ where i is such that

$$p(i) >= prob >= p(i+1)$$

Of course, this method implies that an extra field p is kept with each record or that p can be computed based on some other information in each record.

Reordering a List for Maximum Search Efficiency

Unfortunately, the probabilities $p(i)$ are rarely known in advance. Although it is usual for certain records to be retrieved more often than others, it is almost impossible to identify those records in advance. Also, the probability that a given record will be retrieved may change over time. To use the example of the college registrar given earlier, a student begins as a freshman (high probability of retrieval) and then becomes a sophomore and a junior (low probability) before becoming a senior (high probability). Thus it would be helpful to have an algorithm that continually reorders the table so that more frequently accessed records drift to the front, while less frequently accessed records drift to the back.

There are two search methods that accomplish this. One of these is known as the ***move-to-front*** method and is efficient only for a table organized as a list. In this method, whenever a search is successful (that is, when the argument is found to match the key of a given record), the retrieved record is removed from its current location in the list and is placed at the head of the list.

The other method is the ***transposition*** method, in which a successfully retrieved record is interchanged with the record that immediately precedes it. We present an algorithm to implement the transposition method on a table stored as a linked list. The algorithm returns a pointer to the retrieved record, or the null pointer if the record is not found. As before, *key* is the search argument, *k* and *r* are the tables of keys and records. *table* is a pointer to the first node of the list.

```
q = s = null;  /* q is one step behind p; */
               /* s is two steps behind p  */
for (p = table; p != null && k(p) != key; p = next(p)) {
   s = q;
   q = p;
} /* end for */
```

```
if (p == null)
   return (p);
/*   We have found the record at position p.    */
/* Transpose the records pointed to by p and q. */
if (q == null)
   /* The key is in the first table position.   */
   /*       No transposition is necessary.      */
   return (p);
/* Transpose node(q) and node(p). */
next(q) = next(p);
next(p) = q;
(s = null)  ?  table = p : next(s) = p;
return (p);
```

Note that the two *if* statements in the foregoing can be combined into the single statement *if* ($p == null \| q == null$) *return* (p); for conciseness. We leave the implementation of the transposition method for an array and the move-to-front method as exercises for the reader.

Both of these methods are based on the observed phenomenon that a record that has been retrieved is likely to be retrieved again. By advancing such records toward the front of the table, subsequent retrievals are more efficient. The rationale behind the move-to-front method is that, since the record is likely to be retrieved again, it should be placed at the position within the table at which such retrieval is most efficient. However, the counterargument for the transposition method is that a single retrieval does not yet imply that the record will be retrieved frequently; placing it at the front of the table reduces search efficiency for all the other records that formerly preceded it. By advancing a record only one position each time that it is retrieved, we ensure that it advances to the front of the list only if it is retrieved frequently.

It has been shown that, over a large number of search requests with an unchanging probability distribution, the transposition method is more efficient. However, the move-to-front method yields better results for a small to medium number of requests and responds more quickly to a change in probability distribution. It also has better worst-case behavior than does transposition. For this reason, move-to-front is preferred in most practical situations involving sequential search.

If large numbers of searches with an unchanging probability distribution are required, a mixed strategy may be best: use move-to-front for the first s searches to organize rapidly the list in good sequence and then switch to transposition to obtain even better behavior. The exact value of s to optimize overall efficiency depends on the length of the list and the exact access probability distribution.

One advantage of the transposition method over the move-to-front method is that it can be applied efficiently to tables stored in array form as well as to list-structured tables. Transposing two elements in an array is a rather efficient operation, whereas moving an element from the middle of an array to its front involves (on the average) moving half the array. (However, in this case the average number of moves is not so large, since the element to be moved most often comes from the upper portion of the array.)

Searching an Ordered Table

If the table is stored in ascending or descending order of the record keys, there are several techniques that can be used to improve the efficiency of searching. This is especially true if the table is of fixed size. One obvious advantage in searching a sorted file over searching an unsorted file is in the case that the argument key is absent from the file. In the case of an unsorted file, n comparisons are needed to detect this fact. In the case of a sorted file, assuming that the argument keys are uniformly distributed over the range of keys in the file, only $n/2$ comparisons (on the average) are needed. This is because we know that a given key is missing from a file sorted in ascending order of keys as soon as we encounter a key that is greater than the argument.

Suppose that it is possible to collect a large number of retrieval requests before any of them are processed. For example, in many applications a response to a request for information may be deferred to the next day. In such a case, all requests in a specific day may be collected and the actual searching may be done overnight, when no new requests are coming in. If both the table and the list of requests are sorted, the sequential search can proceed through both concurrently. Thus it is not necessary to search through the entire table for each retrieval request. In fact, if there are many such requests uniformly distributed over the entire table, each request will require only a few lookups (if the number of requests is less than the number of table entries) or perhaps only a single comparison (if the number of requests is greater than the number of table entries). In such situations sequential searching is probably the best method to use.

Because of the simplicity and efficiency of sequential processing on sorted files, it may be worthwhile to sort a file before searching for keys in it. This is especially true in the situation described in the preceding paragraph, where we are dealing with a "master" file and a large "transaction" file of requests for searches.

Indexed Sequential Search

There is another technique to improve search efficiency for a sorted file, but it involves an increase in the amount of space required. This method is called the ***indexed sequential*** search method. An auxiliary table, called an ***index,*** is set aside in addition to the sorted file itself. Each element in the index consists of a key *kindex* and a pointer to the record in the file that corresponds to *kindex*. The elements in the index, as well as the elements in the file, must be sorted on the key. If the index is one eighth the size of the file, every eighth record of the file is represented in the index. This is illustrated by Figure 7.1.1.

The algorithm used for searching an indexed sequential file is straightforward. Let r, k, and *key* be defined as before, let *kindex* be an array of the keys in the index, and let *pindex* be the array of pointers within the index to the actual records in the file. We assume that the file is stored as an array, that n is the size of the file, and that *indxsze* is the size of the index.

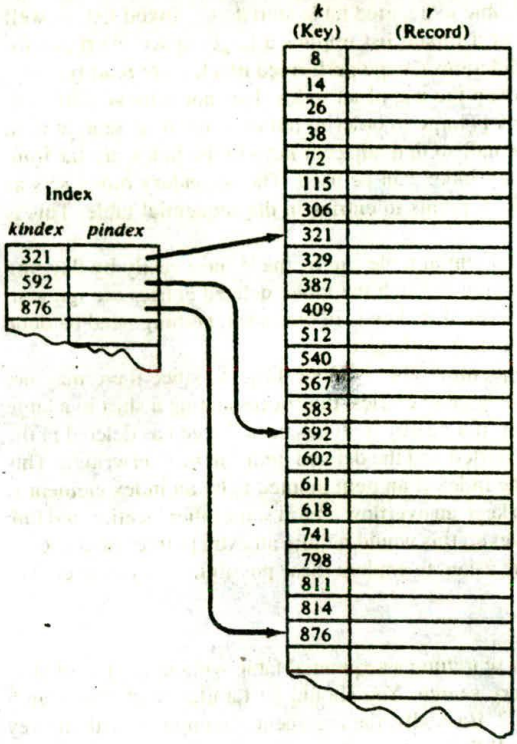


Figure 7.1.1 Indexed sequential file.

```
for (i = 0; i < indxsize && kindex(i) <= key; i++)
   ;
lowlim = (i == 0) ? 0 : pindex(i - 1);
hilim = (i == indxsize) ? n - 1 : pindex(i) - 1;
for (j = lowlim; j <= hilim && k(j) != key; j++)
   ;
return ((j > hilim) ? -1 : j);
```

Note that in the case of multiple records with the same key, the foregoing algorithm does not necessarily return a pointer to the first such record in the table.

The real advantage of the indexed sequential method is that the items in the table can be examined sequentially if all the records in the file must be accessed, yet the search time for a particular item is sharply reduced. A sequential search is performed on the smaller index rather than on the larger table. Once the correct index position has been found a second sequential search is performed on a small portion of the record table itself.

The use of an index is applicable to a sorted table stored as a linked list, as well as to one stored as an array. Use of a linked list implies a larger space overhead for pointers, although insertions and deletions can be performed much more readily.

If the table is so large that even the use of an index does not achieve sufficient efficiency (either because the index is large in order to reduce sequential searching in the table, or because the index is small so that adjacent keys in the index are far from each other in the table), a secondary index can be used. The secondary index acts as an index to the primary index, which points to entries in the sequential table. This is illustrated in Figure 7.1.2.

Deletions from an indexed sequential table can be made most easily by flagging deleted entries. In sequential searching through the table, deleted entries are ignored. Note that if an element is deleted, even if its key is in the index, nothing need be done to the index; only the original table entry is flagged.

Insertion into an indexed sequential table is more difficult, since there may not be room between two already existing table entries, thus necessitating a shift in a large number of table elements. However, if a nearby item has been flagged as deleted in the table, only a few items need to be shifted and the deleted item can be overwritten. This may in turn require alteration of the index if an item pointed to by an index element is shifted. An alternative method is to keep an overflow area at some other location and link together any inserted records. However, this would require an extra pointer field in each record of the original table. You are asked to explore these possibilities as an exercise.

Binary Search

The most efficient method of searching a sequential table without the use of auxiliary indices or tables is the binary search. You should be familiar with this search technique from Sections 3.1 and 3.2. Basically, the argument is compared with the key of the middle element of the table. If they are equal, the search ends successfully; otherwise, either the upper or lower half of the table must be searched in a similar manner.

In Chapter 3 it was noted that the binary search can best be defined recursively. As a result, a recursive definition, a recursive algorithm, and a recursive program were presented for the binary search. However, the overhead associated with recursion may make it inappropriate for use in practical situations in which efficiency is a prime consideration. We therefore present the following nonrecursive version of the binary search algorithm:

```
low = 0;
hi = n - 1;
while (low <= hi) {
   mid = (low + hi)/2;
   if (key == k(mid))
      return(mid);
   if (key < k(mid))
      hi = mid - 1;
   else
      low = mid + 1;
} /* end while */
return(-1);
```

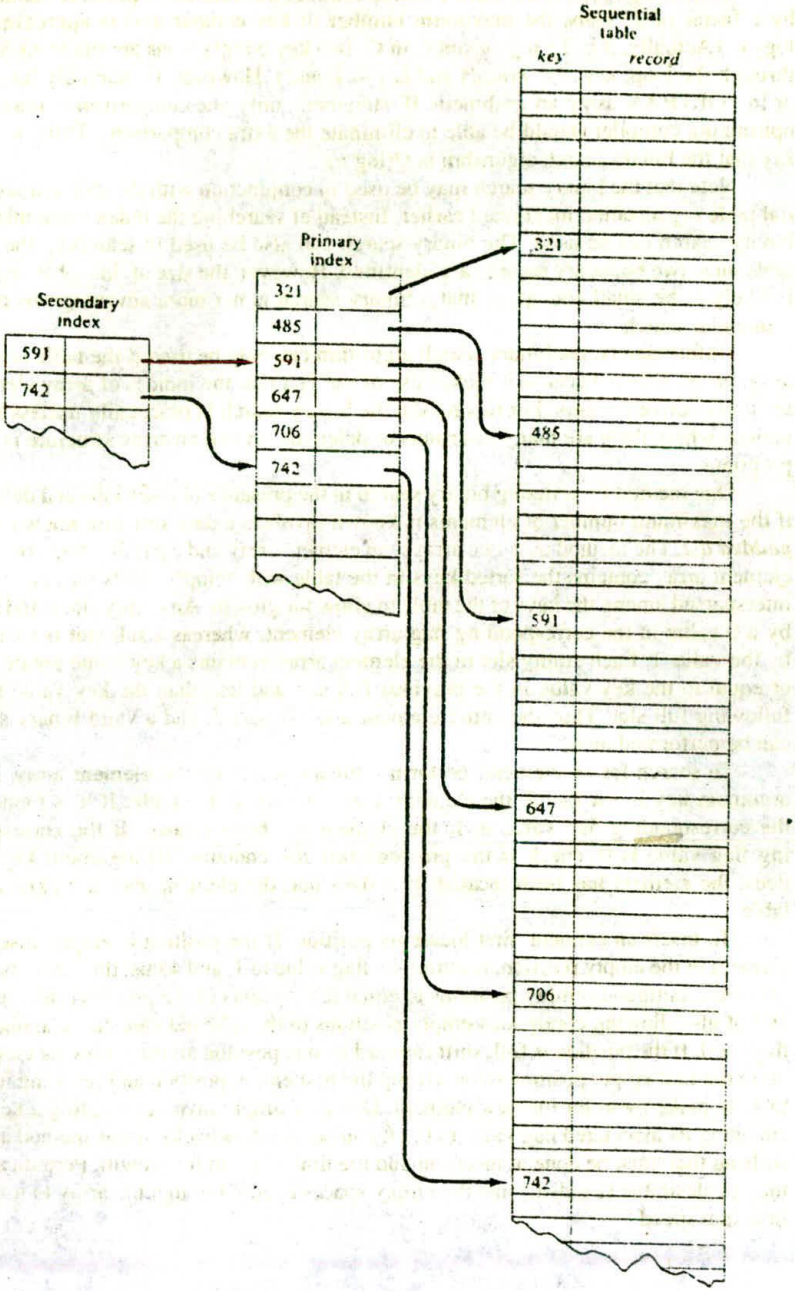


Figure 7.1.2 Use of a secondary index.

Each comparison in the binary search reduces the number of possible candidates by a factor of 2. Thus, the maximum number of key comparisons is approximately $\log_2 n$. (Actually, it is $2 * \log_2 n$ since, in C, two key comparisons are made each time through the loop: $key == k(mid)$ and $key < k(mid)$. However, in assembly language or in FORTRAN using an arithmetic IF statement, only one comparison is made. An optimizing compiler should be able to eliminate the extra comparison.) Thus, we may say that the binary search algorithm is $O(\log n)$.

Note that the binary search may be used in conjunction with the indexed sequential table organization mentioned earlier. Instead of searching the index sequentially, a binary search can be used. The binary search can also be used in searching the main table once two boundary records are identified. However, the size of this table segment is likely to be small enough so that a binary search is not more advantageous than a sequential search.

Unfortunately, the binary search algorithm can only be used if the table is stored as an array. This is because it makes use of the fact that the indices of array elements are consecutive integers. For this reason the binary search is practically useless in situations where there are many insertions or deletions, so that an array structure is inappropriate.

One method for utilizing binary search in the presence of insertions and deletions if the maximum number of elements is known involves a data structure known as the ***padded list***. The method uses two arrays: an element array and a parallel flag array. The element array contains the sorted keys in the table with "empty" slots initially evenly interspersed among the keys of the table to allow for growth. An empty slot is indicated by a 0 value in the corresponding flag array element, whereas a full slot is indicated by the value 1. Each empty slot in the element array contains a key value greater than or equal to the key value in the previous full slot and less than the key value in the following full slot. Thus the entire element array is sorted, and a valid binary search can be performed on it.

To search for an element, perform a binary search on the element array. If the argument key is not found, the element does not exist in the table. If it is found and the corresponding flag value is 1, the element has been located. If the corresponding flag value is 0, check if the previous full slot contains the argument key. If it does, the element has been located; if it does not, the element does not exist in the table.

To insert an element, first locate its position. If the position is empty, insert the element in the empty position, resetting its flag value to 1, and adjust the contents of all previous contiguous empty positions to equal the contents of the previous full element and of all following contiguous empty positions to the inserted element, leaving their flags at 0. If the position is full, shift forward by one position all the following elements up to the first empty position (overwriting the first empty position and resetting its flag to 1) to make room for the new element. Deletion simply involves locating a key and changing its associated flag value to 0. Of course, the drawbacks of this method are the shifting that must be done at insertion and the limited room for growth. Periodically, it may be desirable to redistribute the empty spaces evenly through the array to improve insertion speed.

Interpolation Search

Another technique for searching an ordered array is called ***interpolation search***. If the keys are uniformly distributed between $k(0)$ and $k(n-1)$, the method may be even more efficient than binary search.

Initially, as in binary search, *low* is set to 0 and *high* is set to $n-1$, and throughout the algorithm, the argument key *key* is known to be between $k(low)$ and $k(high)$. On the assumption that the keys are uniformly distributed between these two values, *key* would be expected to be at approximately position

$$mid = low + (high - low) * ((key - k(low))/(k(high) - k(low)))$$

If *key* is lower than $k(mid)$, reset *high* to $mid - 1$; if higher, reset *low* to $mid + 1$. Repeat the process until the key has been found or $low > high$.

Indeed, if the keys are uniformly distributed through the array, interpolation search requires an average of $\log_2(\log_2 n)$ comparisons and rarely requires much more, compared with binary search's $\log_2 n$ (again, considering the two comparisons for equality and inequality of *key* and $k(mid)$ as one). However, if the keys are not uniformly distributed, interpolation search can have very poor average behavior. In the worst case, the value of *mid* can consistently equal $low + 1$ or $high - 1$, in which case interpolation search degenerates into sequential search. By contrast, binary search's comparisons are never greater than approximately $\log_2 n$. In practical situations, keys often tend to cluster around certain values and are not uniformly distributed. For example, more names begin with "S" than with "Q," and there are likely to be many Smiths and very few Quodnots. In such situations, binary search is far superior to interpolation search.

A variation of interpolation search, called ***robust interpolation search*** (or ***fast search***), attempts to remedy the poor practical behavior of interpolation search while extending its advantage over binary search to nonuniform key distributions. This is done by establishing a value *gap* so that *mid-low* and *high-mid* are always greater than *gap*. Initially, *gap* is set to $sqrt(high - low + 1)$. *probe* is set to $low + (high - low) * ((key - k(low))/(k(high) - k(low)))$, and *mid* is set equal to $min(high - gap, max(probe, low + gap))$ (where *min* and *max* return the minimum and maximum, respectively, of two values). That is, we guarantee that the next position used for comparison (*mid*) is at least *gap* positions from the ends of the interval, where *gap* is at least the square root of the interval. When the argument key is found to be restricted to the smaller of the two intervals, from *low* to *mid* and *mid* to *high*, *gap* is reset to the square root of the new interval size. However, if the argument key is found to lie in the larger of the two intervals, the value of *gap* is doubled, although it is never allowed to be greater than half the interval size. This guarantees escape from a large cluster of similar key values.

The expected number of comparisons for robust interpolation search for a random distribution of keys is $O(\log \log n)$. This is superior to binary search. On a list of approximately 40,000 names, binary search requires an average of approximately 16 key comparisons. Owing to clustering of names in practical situations, interpolation search required 134 average comparisons in an actual experiment, whereas robust

interpolation search required only 12.5. On a uniformly distributed list of approximately 40,000 elements—$\log_2(\log_2 40{,}000)$ is approximately 3.9—robust interpolation search required 6.7 average comparisons. (It should be noted that the extra computation time required for robust interpolation search may be substantial but is ignored in these findings.) The worst case for robust interpolation search is $(O(\log n)^2)$ comparisons, which is higher than that for binary search, but much better than the $O(n)$ of regular interpolation search.

However, on most computers, the computations required by interpolation search are very slow, since they involve arithmetic on keys and complex multiplications and divisions. Binary search requires only arithmetic on integer indexes and division by 2 which can be performed efficiently by shifting one bit to the right. Thus the computational requirements of interpolation search often cause it to perform more slowly than binary search even when it requires fewer comparisons.

EXERCISES

7.1.1. Modify the search and insertion algorithms of this section so that they become update algorithms. If an algorithm finds an i such that *key* equals $k(i)$, change the value of rec_i to *rec*.

7.1.2. Implement the sequential search and the sequential search and insertion algorithms in C for both arrays and linked lists.

7.1.3. Compare the efficiency of searching an ordered sequential table of size n and searching an unordered table of the same size for the key *key*:

(a) If no record with key *key* is present

(b) If one record with key *key* is present and only one is sought

(c) If more than one record with key *key* is present and it is desired to find only the first one

(d) If more than one record with key *key* is present and it is desired to find them all

7.1.4. Assume that an ordered table is stored as a circular list with two external pointers, *table* and *other*. *table* always points to the node containing the record with the smallest key. *other* is initially equal to *table* but is reset each time a search is performed to point to the record that is retrieved. If a search is unsuccessful, *other* is reset to *table*. Write a C routine *search(table, other, key)* that implements this method and that returns a pointer to a retrieved record or a null pointer if the search is unsuccessful. Explain how keeping the pointer *other* can reduce the average number of comparisons in a search.

7.1.5. Consider an ordered table implemented as an array or as a doubly linked list so that the table can be searched sequentially either backward or forward. Assume that a single pointer p points to the last record successfully retrieved. The search always begins at the record pointed to by p but may proceed in either direction. Write a routine *search(table, p, key)* for the case of an array and a doubly linked list to retrieve a record with key *key* and to modify p accordingly. Prove that the numbers of key comparisons in both the successful and unsuccessful cases are the same as in the method of Exercise 7.1.4, in which the table may be scanned in only one direction but the scanning process may start at one of two points.

7.1.6. Consider a programmer who writes the following code:

```
if (c1 )
  if (c2 )
    if (c3 )
      ...
        if (cn )
          { statement }
```

where c_i is a condition that is either true or false. Note that rearranging the conditions in a different order results in an equivalent program, since the {statement} is only executed if all the c_i are true. Assume that *time(i)* is the time needed to evaluate condition c_i and that *prob(i)* is the probability that condition c_i is true. In what order should the conditions be arranged to make the program most efficient?

7.1.7. Modify the indexed sequential search so that in the case of multiple records with the same key it returns the first such record in the table.

7.1.8. Consider the following C implementation of an indexed sequential file:

```
#define INDXSIZE  100;
#define TABLESIZE 1000;
struct indxtype {
   int kindex;
   int pindex;
};
struct tabletype {
   int k;
   int r;
   int flag;
};
struct isfiletype {
   struct indxtype indx[INDXSIZE];
   struct tabletype table[TABLESIZE];
};
struct isfiletype isfile;
```

Write a C routine *create(isfile)* that initializes such a file from input data. Each input line contains a key and a record. The input is sorted in ascending key order. Each index entry corresponds to ten table entries. *flag* is set to *TRUE* in an occupied table entry and to *FALSE* in an unoccupied entry. Two out of every ten table entries are left unoccupied to allow for future growth.

7.1.9. Given an indexed sequential file as in the previous exercise, write a C routine *search(isfile, key)* to print the record in the file with key *key* if it is present and an indication that the record is missing if no record with that key exists. (How can you ensure that an unsuccessful search is as efficient as possible?) Also, write routines *insert(isfile, key, rec)* to insert a record *rec* with key *key* and *delete(isfile, key)* to delete the record with key *key*.

7.1.10. Consider the following version of the binary search, which assumes that the keys are contained in *k*[1] through *k*[*n*] and that the special element *k*[0] is smaller than every possible key:

```
mid = n/2;
len = (n - 1)/2;
while (key != k(mid)) {
   if (key < k(mid))
      mid -= len/2;
   else
      mid += len/2;
   if (len == 0)
      return(-1);
   len /= 2;
} /* end while */
return(mid);
```

Prove that this algorithm is correct. What are the advantages and/or disadvantages of this method over the method presented in the text?

7.1.11. The following search algorithm on a sorted array is known as the ***Fibonaccian search*** because of its use of Fibonacci numbers. (For a definition of Fibonacci numbers and the *fib* function, see Section 3.1.)

```
for (j = 1; fib(j) < n; j++)
   ;
mid = n - fib(j - 2) + 1;
f1 = fib(j - 2);
f2 = fib(j - 3);
while (key != k(mid))
  if (mid < 0    key > k(mid)) {
     if (f1 == 1)
        return(-1);
     mid += f2;
     f1 -= f2;
     f2 -= f1;
  }
  else {
     if (f2 == 0)
        return(-1);
     mid -= f2;
     t = f1 - f2;
     f1 = f2;
     f2 = t;
  } /* end if */
return(mid);
```

Explain how this algorithm works. Compare the number of key comparisons with the number used by the binary search. Modify the initial portion of this algorithm so that it computes the Fibonacci numbers efficiently, rather than looking them up in a table or computing each anew.

7.1.12. Modify the binary search of the text so that in the case of an unsuccessful search it returns the index i such that $k(i) < key < k(i + 1)$. If $key < k(0)$, it returns -1, and if $key > k(n - 1)$, it returns $n - 1$. Do the same for the searches of Exercises 7.1.10 and 7.1.11.

In Section 7.1 we discussed search operations on a file that is organized either as an array or as a list. In this section we consider several ways of organizing files as trees and some associated searching algorithms.

In Sections 5.1 and 6.3 we presented a method of using a binary tree to store a file in order to make sorting the file more efficient. In that method, all the left descendants of a node with key *key* have keys that are less than *key*, and all the right descendants have keys that are greater than or equal to *key*. The inorder traversal of such a binary tree yields the file in ascending key order.

Such a tree may also be used as a binary search tree. Using binary tree notation, the algorithm for searching for the key *key* in such a tree is as follows (we assume that each node contains four fields: *k*, which holds the record's key value, *r*, which holds the record itself, and *left* and *right*, which are pointers to the subtrees):

```
p = tree;
while (p != null && key != k(p))
   p = (key < k(p))  ?  left(p) : right(p);
return(p);
```

The efficiency of the search process can be improved by using a sentinel, as in sequential searching. A sentinel node, with a separate external pointer pointing to it, remains allocated with the tree. All *left* or *right* tree pointers that do not point to another tree node now point to this sentinel node instead of equaling *null*. When a search is performed, the argument key is first inserted into the sentinel node, thus guaranteeing that it will be located in the tree. This enables the header of the search loop to be written *while* ($key != k(p)$) without the risk of an infinite loop. After leaving the loop, if p equals the external sentinel pointer, the search is unsuccessful; otherwise p points to the desired node. We leave the actual algorithm to the reader.

Note that the binary search of Section 7.1 actually uses a sorted array as an implicit binary search tree. The middle element of the array can be thought of as the root of the tree, the lower half of the array (all of whose elements are less than the middle element) can be considered the left subtree, and the upper half (all of whose elements are greater than the middle element) can be considered the right subtree.

A sorted array can be produced from a binary search tree by traversing the tree in inorder and inserting each element sequentially into the array as it is visited. On the other hand, there are many binary search trees that correspond to a given sorted array. Viewing the middle element of the array as the root of a tree and viewing the remaining elements recursively as left and right subtrees produces a relatively balanced binary search tree (see Figure 7.2.1a). Viewing the first element of the array as the root of a tree and each successive element as the right son of its predecessor produces a very unbalanced binary tree (see Figure 7.2.1b).

The advantage of using a binary search tree over an array is that a tree enables search, insertion, and deletion operations to be performed efficiently. If an array is used, an insertion or deletion requires that approximately half of the elements of the array be moved. (Why?) Insertion or deletion in a search tree, on the other hand, requires that only a few pointers be adjusted.

30	a[0]
47	a[1]
86	a[2]
95	a[3]
115	a[4]
130	a[5]
138	a[6]
159	a[7]
166	a[8]
184	a[9]
206	a[10]
212	a[11]
219	a[12]
224	a[13]
237	a[14]
258	a[15]
296	a[16]
307	a[17]
314	a[18]

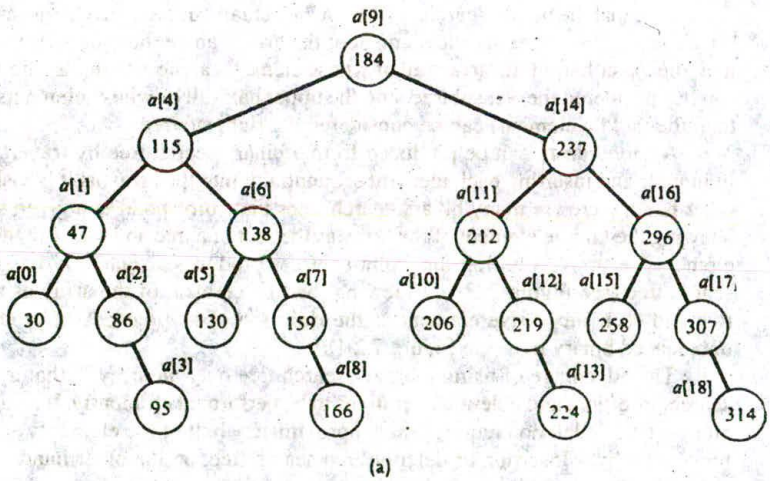
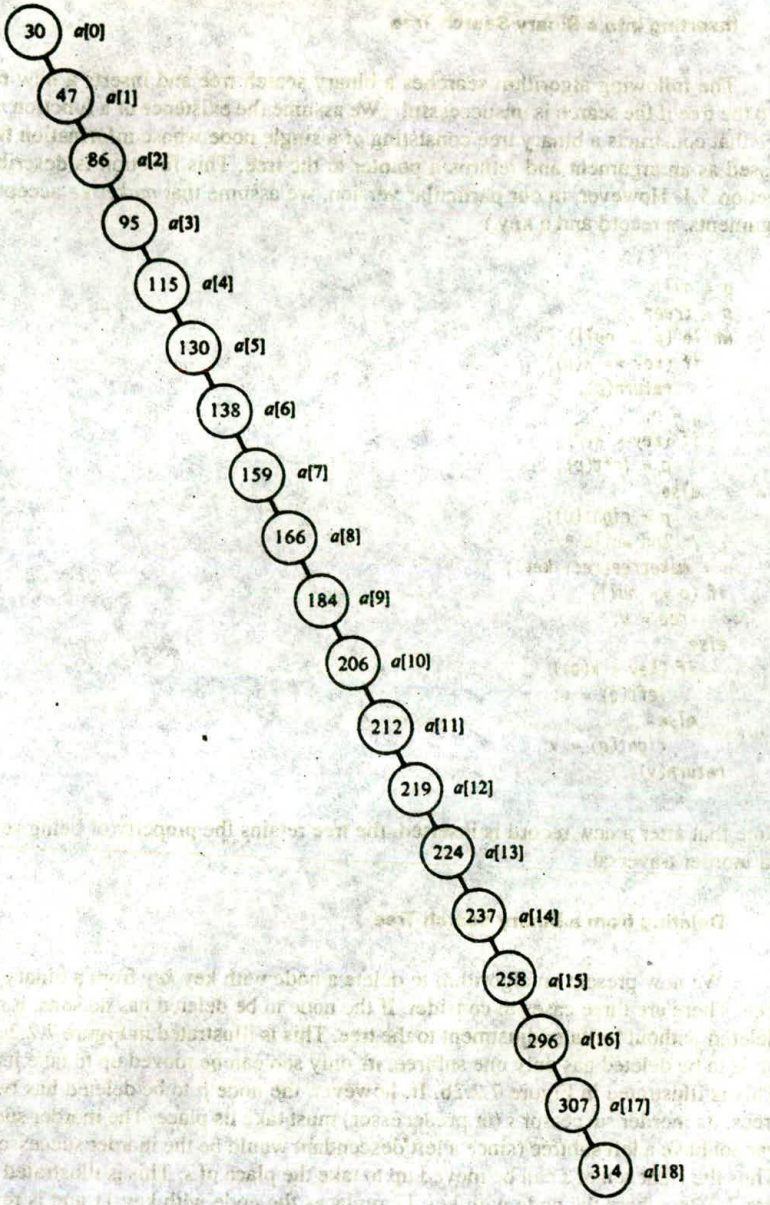


Figure 7.2.1 Sorted array and two of its binary tree representations.



(b)

Figure 7.2.1 *(cont.)*

Inserting into a Binary Search Tree

The following algorithm searches a binary search tree and inserts a new record into the tree if the search is unsuccessful. (We assume the existence of a function *maketree* that constructs a binary tree consisting of a single node whose information field is passed as an argument and returns a pointer to the tree. This function is described in Section 5.1. However, in our particular version, we assume that *maketree* accepts two arguments, a record and a key.)

```
q = null;
p = tree;
while (p != null) {
   if (key == k(p))
      return(p);
   q = p;
   if (key < k(p))
      p = left(p);
   else
      p = right(p);
} /* end while */
v = maketree(rec, key);
if (q == null)
   tree = v;
else
   if (key < k(q))
      left(q) = v;
   else
      right(q) = v;
return(v);
```

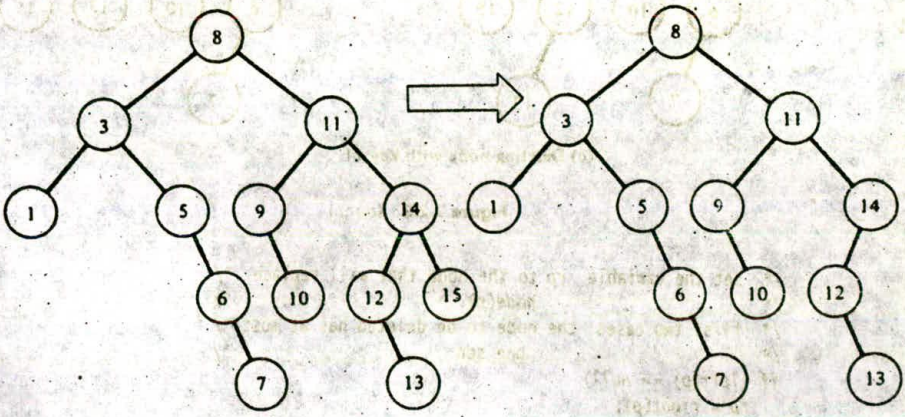
Note that after a new record is inserted, the tree retains the property of being sorted in an inorder traversal.

Deleting from a Binary Search Tree

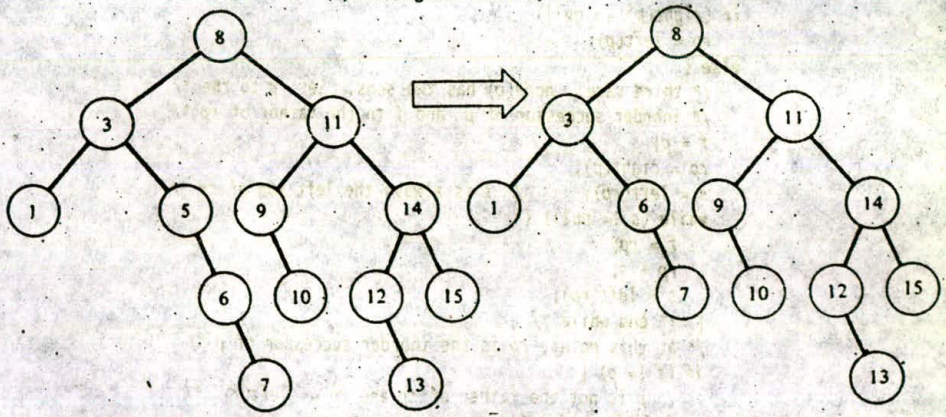
We now present an algorithm to delete a node with key *key* from a binary search tree. There are three cases to consider. If the node to be deleted has no sons, it may be deleted without further adjustment to the tree. This is illustrated in Figure 7.2.2a. If the node to be deleted has only one subtree, its only son can be moved up to take its place. This is illustrated in Figure 7.2.2b. If, however, the node p to be deleted has two subtrees, its inorder successor s (or predecessor) must take its place. The inorder successor cannot have a left subtree (since a left descendant would be the inorder successor of p). Thus the right son of s can be moved up to take the place of s. This is illustrated in Figure 7.2.2c, where the node with key 12 replaces the node with key 11 and is replaced, in turn, by the node with key 13.

In the following algorithm, if no node with key *key* exists in the tree, the tree is left unchanged.

```
p = tree;
q = null;
/*  search for the node with the key key, set p to point */
/*      to the node and q to its father, if any.         */
while (p != null && k(p) != key) {
    q = p;
    p = (key < k(p))  ?  left(p) : right(p);
} /* end while */
if (p == null)
    /*  the key does not exist in the tree  */
    /*      leave the tree unchanged        */
    return;
```

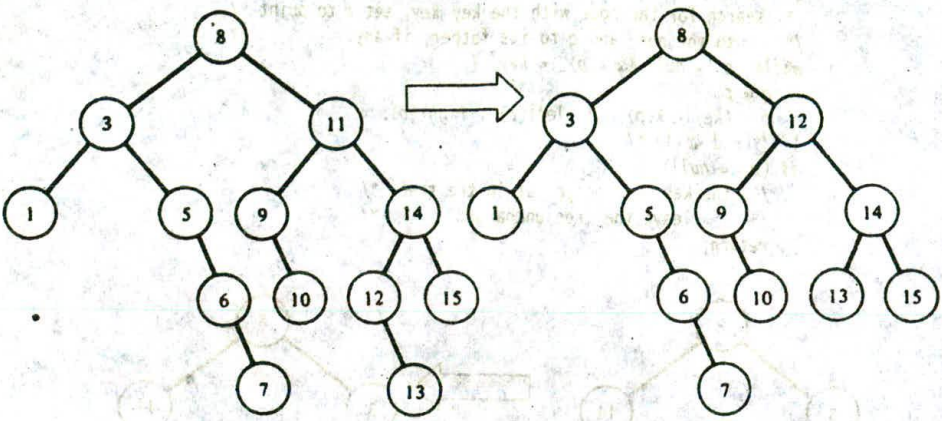


(a) Deleting node with key 15.



(b) Deleting node with key 5.

Figure 7.2.2 Deleting nodes from a binary search tree.



(c) Deleting node with key 11.

Figure 7.2.2 (*cont.*)

```
/*  set the variable  rp to the node that will replace  */
/*                        node(p).                      */
/*  first two cases: the node to be deleted has at most */
/*                        one son                       */
if (left(p) == null)
   rp = right(p);
else
   if (right(p) == null)
      rp = left(p);
   else {
      /* third case:  node(p) has two sons.  Set rp to the */
      /* inorder successor of p  and f to the father of rp */
      f = p;
      rp = right(p);
      s = left(rp);       /* s is always the left son of rp */
      while (s != null) {
         f = rp;
         rp = s;
         s = left(rp);
      } /* end while */
      /* at this point, rp is the inorder successor of p */
      if (f != p) {
         /* p is not the father of rp and rp == left(f)  */
         left(f) = right(rp);
         /* remove node(rp) from its current position and */
         /* replace  it  with  the right  son of node(rp) */
         /*       node(rp) takes the place of node(p)     */
         right(rp) = right(p);
      } /* end if */
```

```
      /*   set the left son of node(rp) so that  */
      /*  node(rp) takes the place of  node(p)   */
      left(rp) = left(p);
   } /* end if */
/* insert node(rp) into the position formerly */
/*            occupied by node(p)             */
if (q == null)
   /* node(p) was the root of the tree */
   tree = rp;
else
   (p == left(q)) ? left(q) = rp : right(q) = rp;
freenode(p);
return;
```

Efficiency of Binary Search Tree Operations

As we have already seen in Section 6.3 (see Figures 6.3.1 and 6.3.2), the time required to search a binary search tree varies between $O(n)$ and $O(\log n)$, depending on the structure of the tree. If elements are inserted into the tree by the foregoing insertion algorithm, the structure of the tree depends on the order in which the records are inserted. If the records are inserted in sorted (or reverse) order, the resulting tree contains all null left (or right) links, so that the tree search reduces to a sequential search. If, however, the records are inserted so that half the records inserted after any given record r with key k have keys smaller than k and half have keys greater than k, a balanced tree is achieved in which approximately $\log n$ key comparisons are sufficient to retrieve an element. (Again, it should be noted that examining a node in our insertion algorithm requires two comparisons: one for equality and the other for less than. However, in machine language and in some compilers, these can be combined into a single comparison.)

If the records are presented in random order (that is, any permutation of the n elements is equally likely), balanced trees result more often than not, so that on the average, search time remains $O(\log n)$. To see this, let us define the ***internal path length***, i, of a binary tree as the sum of the levels of all the nodes in the tree (note that the level of a node equals the length of the path from the root to the node). In the initial tree of Figure 7.2.2, for example, i equals 30 (1 node at level 0, 2 at level 1, 4 at level 2, 4 at level 3, and 2 at level 4: $1*0 + 2*1 + 4*2 + 4*3 + 2*4 = 30$). Since the number of comparisons required to access a node in a binary search tree is one greater than the node's level, the average number of comparisons required for a successful search in a binary search tree with n nodes equals $(i + n)/n$, assuming equal likelihood for accessing every node in the tree. Thus, for the initial tree of Figure 7.2.2, $(30 + 13)/13$, or approximately 3.31, comparisons are required for a successful search. Let s_n equal the average number of comparisons required for a successful search in a random binary search tree of n nodes in which the search argument is equally likely to be any of the n keys, and let i_n be the average internal path length of a random binary search tree of n nodes. Then s_n equals $(i_n + n)/n$.

Let u_n be the average number of comparisons required for an unsuccessful search of a random binary search tree of n nodes. There are $n + 1$ possible ways for an un-

successful search for a key *key* to occur: *key* is less that $k(1)$, *key* is between $k(1)$ and $k(2)$, ... *key* is between $k(n-1)$ and $k(n)$, and *key* is greater than $k(n)$. These correspond to the $n+1$ null subtree pointers in any n-node binary search. (It can be shown that any binary search tree with n nodes has $n+1$ null pointers.)

Consider the ***extension*** of a binary tree formed by replacing each null left or right pointer with a pointer to a separate, new leaf node, called an ***external node***. The extension of a binary tree of n nodes has $n+1$ external nodes, each corresponding to one of the $n+1$ key ranges for an unsuccessful search. For example, the initial tree of Figure 7.2.2 contains 13 nodes. Its extension would add two external nodes as sons of each of the leaves containing 1, 7, 10, 13, and 15 and one external node as an additional son of each of the nodes containing 5, 6, 9, and 12, for a total of 14 external nodes. Define the ***external path length***, e, of a binary tree as the sum of the levels of all the external nodes of its extension. The extension of the initial tree of Figure 7.2.2 has 4 external nodes at level 3, 6 at level 4, and 4 at level 5, for an external path length of 56. Note that the level of an external node equals the number of comparisons in an unsuccessful search for a key in the range represented by that external node. Then if e_n is the average external path length of a random binary search tree of n nodes, $u_n = e_n/(n+1)$. (This assumes that each of the $n+1$ key ranges is equally likely in an unsuccessful search. In Figure 7.2.2 the average number of comparisons for an unsuccessful search is 56/14 or 4.0.) However, it can be shown that $e = i + 2n$ for any binary tree of n nodes (for example, in Figure 7.2.2, $56 = 30 + 2*13$), so that $e_n = i_n + 2n$. Since $s_n = (u_n + n)/n$ and $u_n = e_n/(n+1)$, this means that $s_n = ((n+1)/n)u_n - 1$.

The number of comparisons required to access a key is one more than the number required when the node was inserted. But the number required to insert a key equals the number required in an unsuccessful search for that key before it was inserted. Thus $s_n = 1 + (u_0 + u_1 + \cdots + u_{n-1})/n$. (That is, the average number of comparisons in retrieving an item in an n-node tree equals the average number in accessing each of the first item through the nth, and the number for accessing the ith equals one more than the number for inserting the ith or $1 + u_{i-1}$.) Combining this with the equation $s_n = ((n+1)/n)u_n - 1$ yields

$$(n+1)u_n = 2n + u_0 + u_1 + \cdots + u_{n-1}$$

for any n. Replacing n by $n-1$ yields

$$nu_{n-1} = 2(n-1) + u_0 + u_1 + \cdots + u_{n-2}$$

and subtracting from the previous equation yields

$$(n+1)u_n - nu_{n-1} = 2 + u_{n-1}$$

or

$$u_n = u_{n-1} + 2/(n+1)$$

Since $u_1 = 1$, we have that

$$u_n = 1 + 2/3 + 2/4 + \cdots + 2/(n+1)$$

and, therefore, since $s_n = ((n+1)/n)u_n - 1$, that

$$s_n = 2((n+1)/n)(1 + 1/2 + 1/3 + \cdots + 1/n) - 3$$

As n grows large, $(n + 1)/n$ is approximately 1, and it can be shown that $1 + 1/2 + \cdots + 1/n$ is approximately $\log(n)$, where $\log(n)$ is defined as the natural logarithm of n in the standard C library file *math.h*. Thus s_n may be approximated (for large n) by $2 * \log(n)$, which equals $1.386 * \log_2 n$. This means that average search time in a random binary search tree is $O(\log n)$ and requires approximately only 39 percent more comparisons, on the average, than in a balanced binary tree.

As already noted, insertion in a binary search tree requires the same number of comparisons as does an unsuccessful search for the key. Deletion requires the same number of comparisons as does a search for the key to be deleted, although it does involve additional work in finding the inorder successor or predecessor. It can be shown that the deletion algorithm that we have presented actually improves the subsequent average search cost of the tree. That is, a random n-key tree created by inserting $n + 1$ keys and then deleting a random key has lower internal path length (and therefore lower average search cost) than does a random n-key tree created by inserting n keys. The process of deleting a one-son node by replacing it with its son, regardless of whether that son is a right or a left son, yields a better-than-average tree; a similar deletion algorithm which only replaces a one-son node by its son if it is a left son (that is, if its successor is not contained in its subtree) and otherwise replaces the node with its inorder successor does produce a random tree, assuming no additional insertions. The latter algorithm is called the ***asymmetric deletion algorithm***.

Strangely enough, however, as additional insertions and deletions are made using the asymmetric deletion algorithm, the internal path length and search time initially decrease but then begin to rise rapidly again. For trees containing more than 128 keys, the internal path length eventually becomes worse than that for a random tree, and for trees with more than 2048 keys, the internal path length eventually becomes more than 50 percent worse than that of a random tree.

An alternative ***symmetric deletion algorithm,*** which alternates between deleting the inorder predecessor and successor on alternate deletions (but still replaces a one-son node with its son only when the predecessor or successor, respectively, is not contained in its subtree), does eventually produce better than random trees after additional mixed insertions and deletions. Empirical data indicate that path length is reduced after many alternating insertions and symmetric deletions to approximately 88 percent of its corresponding random value.

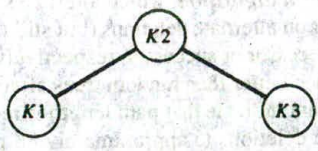
Efficiency of Nonuniform Binary Search Trees

All of the preceding assumes that it is equally likely for the search argument to equal any key in the table. However, in actual practice it is usually the case that some records are retrieved very often, some moderately often, and some are almost never retrieved. Suppose that records are inserted into the tree so that a more commonly accessed record precedes one that is not so frequently accessed. Then the most frequently retrieved records will be nearer the root of the tree; so that the average successful search time is reduced. (Of course, this assumes that reordering the keys in order of reduced frequency of access does not seriously unbalance the binary tree, since if it did the reduced number of comparisons for the most frequently accessed records might be offset by the increased number of comparisons for the vast majority of records.)

If the elements to be retrieved form a constant set, with no insertions or deletions, it may pay to set up a binary search tree that makes subsequent searches more efficient. For example, consider the binary search trees of Figure 7.2.3. Both the trees of Figure 7.2.3a and b contain three elements $k1$, $k2$, and $k3$, where $k1 < k2 < k3$, and are valid binary search trees for that set. However, a retrieval of $k3$ requires two comparisons in Figure 7.2.3a but requires only one comparison in Figure 7.2.3b. Of course, there are still other valid binary search trees for this set of keys.

The number of key comparisons necessary to retrieve a record equals the level of that record in the binary search tree plus 1. Thus a retrieval of $k2$ requires one comparison in the tree of Figure 7.2.3a but requires three comparisons in the tree of Figure 7.2.3b. An unsuccessful search for an argument lying immediately between two keys a and b requires as many key comparisons as the maximum number of comparisons required by successful searches for either a or b. (Why?) This is equal to 1 plus the maximum of the levels of a or b. For example, a search for a key lying between $k2$ and $k3$ requires two key comparisons in Figure 7.2.3a and three comparisons in Figure 7.2.3b, whereas a search for a key greater than $k3$ requires two comparisons in Figure 7.2.3a, but only one comparison in Figure 7.2.3b.

Suppose that $p1$, $p2$, and $p3$ are the probabilities that the search argument equals $k1$, $k2$, and $k3$, respectively. Suppose also that $q0$ is the probability that the search argument is less than $k1$, $q1$ is the probability that it is between $k1$ and $k2$, $q2$ is the probability that it is between $k2$ and $k3$, and $q3$ is the probability that it is greater than $k3$. Then $p1 + p2 + p3 + q0 + q1 + q2 + q3 = 1$. The ***expected number*** of compar-



(a) Expected number of comparisons:
$2p1 + p2 + 2p3 + 2q0 + 2q1 + 2q2 + 2q3$

Figure 7.2.3 Two binary search trees.

isons in a search is the sum of the probabilities that the argument has a given value times the number of comparisons required to retrieve that value, where the sum is taken over all possible search argument values. For example, the expected number of comparisons in searching the tree of Figure 7.2.3a is

$$2p1 + p2 + 2p3 + 2q0 + 2q1 + 2q2 + 2q3$$

and the expected number of comparisons in searching the tree of Figure 7.2.3b is

$$2p1 + 3p2 + p3 + 2q0 + 3q1 + 3q2 + q3$$

This expected number of comparisons can be used as a measure of how "good" a particular binary search tree is for a given set of keys and a given set of probabilities. Thus, for the following probabilities on the left, the tree of Figure 7.2.3a is more efficient; for the probabilities listed on the right, the tree of Figure 7.2.3b is more efficient.

$p1 = .1$	$p1 = .1$
$p2 = .3$	$p2 = .1$
$p3 = .1$	$p3 = .3$
$q0 = .1$	$q0 = .1$
$q1 = .2$	$q1 = .1$
$q2 = .1$	$q2 = .1$
$q3 = .1$	$q3 = .2$
Expected number for 7.2.3a = 1.7	Expected number for 7.2.3a = 1.9
Expected number for 7.2.3b = 2.4	Expected number for 7.2.3b = 1.8

Optimum Search Trees

A binary search tree that minimizes the expected number of comparisons for a given set of keys and probabilities is called ***optimum***. The fastest-known algorithm to produce an optimum binary search tree is $O(n^2)$ in the general case. This is too expensive unless the tree is maintained unchanged over a very large number of searches. In cases in which all the $p(i)$ equal 0 (that is, the keys act only to define range values with which data are associated, so that all searches are "unsuccessful"), an $O(n)$ algorithm to create an optimum binary search tree does exist.

However, although an efficient algorithm to construct an optimum tree in the general case does not exist, there are several methods for constructing near-optimum trees in $O(n)$ time. Assume n keys, $k(1)$ through $k(n)$. Let $p(i)$ be the probability of searching for a key $k(i)$, and $q(i)$ the probability of an unsuccessful search between $k(i-1)$ and $k(i)$ (with $q(0)$ the probability of an unsuccessful search for a key below $k(1)$, and $q(n)$ the probability of an unsuccessful search for a key above $k(n)$). Define $s(i,j)$ as $q(i) + p(i+1) + \cdots + q(j)$.

One method, called the ***balancing method***, attempts to find a value i that minimizes the absolute value of $s(0,i-1) - s(i,n)$ and establishes $k(i)$ as the root of the binary search tree, with $k(1)$ through $k(i-1)$ in its left subtree and $k(i+1)$ through $k(n)$ in its right subtree. The process is then applied recursively to build the left and right subtrees.

Locating the value i at which $abs(s(0,i-1) - s(i,n))$ is minimized can be done efficiently as follows. Initially, set up an array $s0[n+1]$ such that $s0[i]$ equals $s(0,i)$. This can be done by initializing $s0[0]$ to $q(0)$ and $s0[j]$ to $s0[j-1] + p(j) + q(j)$ for j from 1 to n in turn. Once $s0$ has been initialized, $s(i,j)$ can be computed for any i and j as $s0[j] - s0[i-1] - p(i)$ whenever necessary. We define $si(j)$ as $s(0,j-1) - s(j,n)$. We wish to minimize $abs(si(i))$.

After $s0$ has been initialized, we begin the process of finding an i to minimize $abs(s(0,i-1) - s(i,n))$, or $si(i)$. Note that si is a monotonically increasing function. Note also that $si(0) = q(0) - 1$, which is negative, and $si(n) = 1 - q(n+1)$, which is positive. Check the values of $si(1)$, $si(n)$, $si(2)$, $si(n-1)$, $si(4)$, $si(n-3)$, ..., $si(2^j)$, $si(n+1-2^j)$ in turn until discovering the first positive $si(2^j)$ or the first negative $si(n+1-2^j)$. If a positive $si(2^j)$ is found first, the desired i that minimizes $abs(si(i))$ lies within the interval $[2^{j-1}, 2^j]$; if a negative $si(n+1-2^j)$ is found first, the desired i lies within the interval $[n+1-2^j, n+1-2^{j-1}]$. In either case, i has been narrowed down to an interval of size 2^{j-1}. Within the interval, use a binary search to narrow down on i. The doubling effect in the interval size guarantees that the entire recursive process is $O(n)$, whereas if a binary search were used on the entire interval $[0, n]$ to start, the process would be $O(n \log n)$.

A second method used to construct near-optimum binary search trees is called the ***greedy method***. Instead of building the tree from the top down, as in the balancing method, the greedy method builds the tree from the bottom up. The method uses a doubly linked linear list in which each list element contains four pointers, one key value, and three probability values. The four pointers are left and right list pointers used to organize the doubly linked list and left and right subtree pointers used to keep track of binary search subtrees containing keys less than and greater than the key value in the node. The three probability values are the sum of the probabilities in the left subtree, called the ***left probability***, the probability $p(i)$ of the node's key value $k(i)$, called the ***key probability***, and the sum of the probabilities in the right subtree, called the ***right probability***. The ***total probability*** of a node is defined as the sum of its left, key, and right probabilities. Initially, there are n nodes in the list. The key value in the ith node is $k(i)$, its left probability is $q(i-1)$, its right probability is $q(i)$, its key probability is $p(i)$, and its left and right subtree pointers are *null*.

Each iteration of the algorithm finds the first node *nd* on the list whose total probability is less than or equal to its successor's (if no node qualifies, *nd* is set to the last node in the list). The key in *nd* becomes the root of a binary search subtree whose left and right subtrees are the left and right subtrees of *nd*. *nd* is then removed from the list. The left subtree pointer of its successor (if any) and the right subtree pointer of its predecessor (if any) are reset to point to the new subtree, and the left probability of its successor and the right probability of its predecessor are reset to the total probability of *nd*. This process is repeated until only one node remains on the list. (Note that it is not necessary to begin a full list traversal from the list beginning on each iteration; it is only necessary to begin from the second predecessor of the node removed on the previous iteration.) When only one node remains on the list, its key is placed in the root of the final binary search tree, with the left and right subtree pointers of the node as the left and right subtree pointers of the root.

Another technique of reducing average search time when search probabilities are known is a ***split tree***. Such a tree contains two keys rather than one in each node. The first, called the ***node key***, is tested for equality with the argument key. If they are equal, the search ends successfully; if not, the argument key is compared with the second key in the node, called the ***split key***, to determine if the search should continue in the left or right subtree. A particular type of split tree, called a ***median split tree***, sets the node key in each node to the most frequent among the keys in the subtree rooted at that node and sets the split key to the median of the keys in that subtree (that is, the key k such that an equal number of keys in the subtree are less than and greater than k). This has the twin advantages of guaranteeing a balanced tree and assuring that frequent keys are found near the root. Although median split trees require an extra key in each node, they can be constructed as almost complete binary trees implemented in an array, saving space for tree pointers. A median split tree from a given set of keys and frequencies can be built in time $O(n \log n)$, and a search in such a tree always requires fewer than $\log_2 n$ node visits, although each visit does require two separate comparisons.

Balanced Trees

As noted in the foregoing, if the probability of searching for a key in a table is the same for all keys, a balanced binary tree yields the most efficient search. Unfortunately, the search and insertion algorithm presented in the foregoing does not ensure that the tree remains balanced; the degree of balance is dependent on the order in which keys are inserted into the tree. We would like to have an efficient search and insertion algorithm that maintains the search tree as a balanced binary tree.

Let us first define more precisely the notion of a "balanced" tree. The ***height*** of a binary tree is the maximum level of its leaves (this is also sometimes known as the ***depth*** of the tree). For convenience, the height of a null tree is defined as -1. A ***balanced binary tree*** (sometimes called an ***AVL tree***) is a binary tree in which the heights of the two subtrees of every node never differ by more than 1. The ***balance*** of a node in a binary tree is defined as the height of its left subtree minus the height of its right subtree. Figure 7.2.4a illustrates a balanced binary tree. Each node in a balanced binary tree has a balance of 1, -1, or 0, depending on whether the height of its left subtree is greater than, less than, or equal to the height of its right subtree. The balance of each node is indicated in Figure 7.2.4a.

Suppose that we are given a balanced binary tree and use the preceding search and insertion algorithm to insert a new node p into the tree. Then the resulting tree may or may not remain balanced. Figure 7.2.4b illustrates all possible insertions that may be made to the tree of Figure 7.2.4a. Each insertion that yields a balanced tree is indicated by a B. The unbalanced insertions are indicated by a U and are numbered from 1 to 12. It is easy to see that the tree becomes unbalanced only if the newly inserted node is a left descendant of a node that previously had a balance of 1 (this occurs in cases $U1$ through $U8$ in Figure 7.2.4b) or if it is a right descendant of a node that previously had a balance of -1 (cases $U9$ through $U12$). In Figure 7.2.4b, the youngest ancestor that becomes unbalanced in each insertion is indicated by the numbers contained in three of the nodes.

Let us examine further the subtree rooted at the youngest ancestor to become unbalanced as a result of an insertion. We illustrate the case where the balance of this

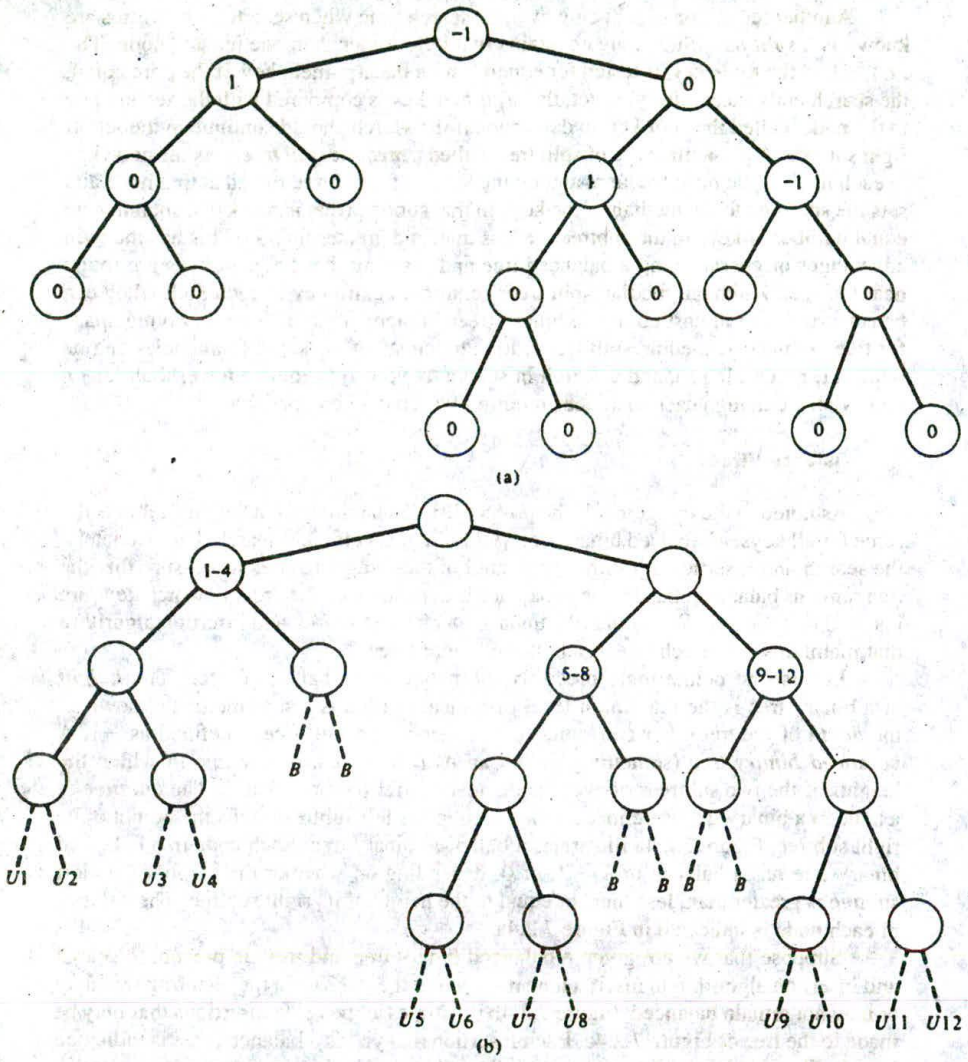
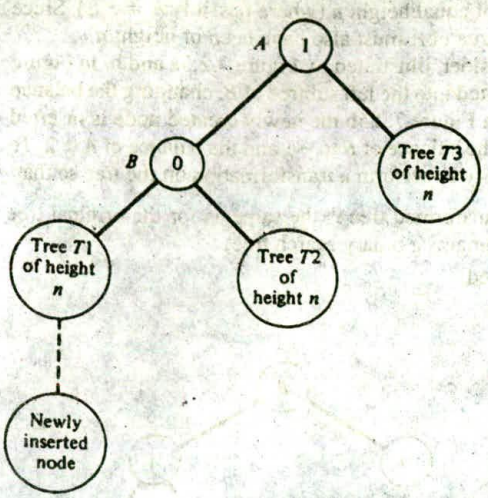
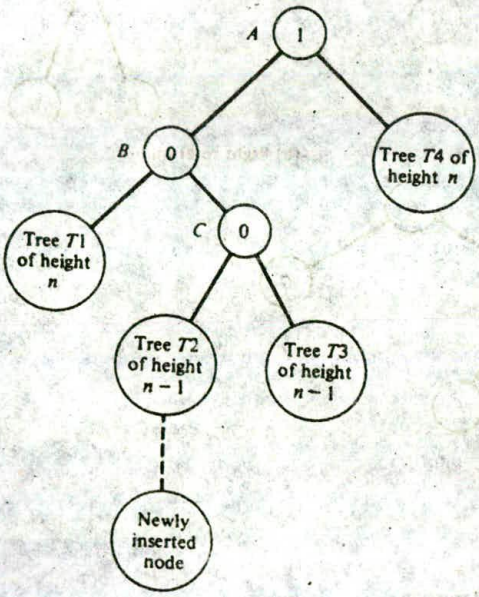


Figure 7.2.4 Balanced binary tree and possible additions.

subtree was previously 1, leaving the other case to the reader. Figure 7.2.5 illustrates this case. Let us call the unbalanced node A. Since A had a balance of 1, its left subtree was nonnull; we may therefore designate its left son as B. Since A is the youngest ancestor of the new node to become unbalanced, node B must have had a balance of 0. (You are asked to prove this fact as an exercise.) Thus, node B must have had (before



(a)



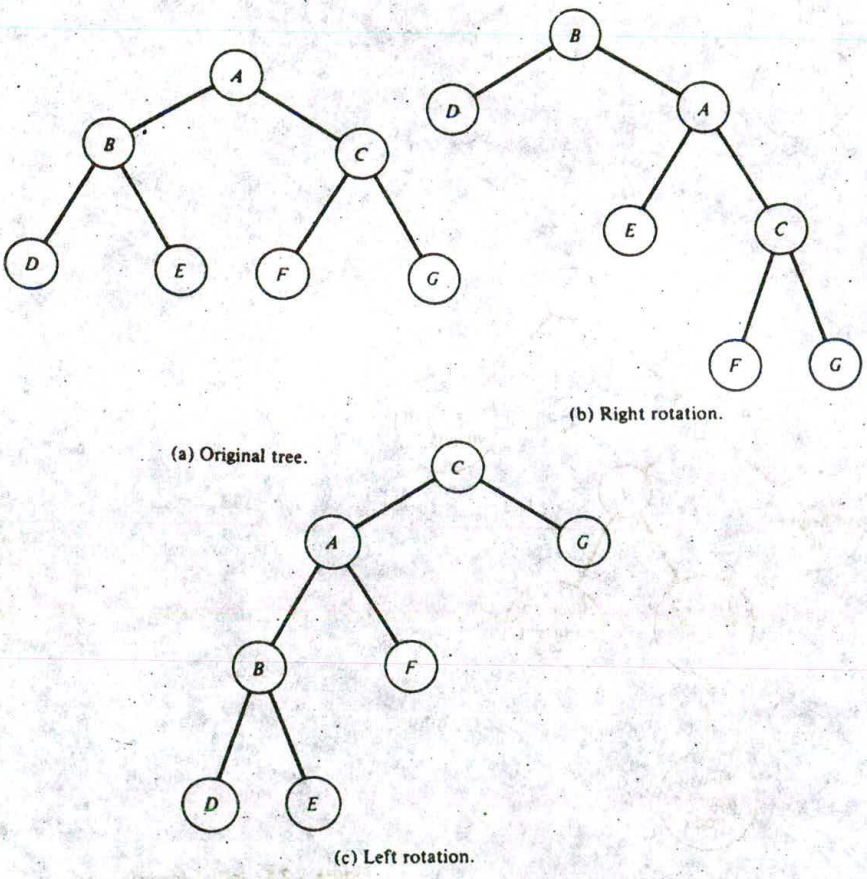
(b)

Figure 7.2.5 Initial insertion; all balances are prior to insertion.

the insertion) left and right subtrees of equal height n (where possibly $n = -1$). Since the balance of A was 1, the right subtree of A must also have been of height n.

There are now two cases to consider, illustrated by Figure 7.2.5a and b. In Figure 7.2.5a the newly created node is inserted into the left subtree of B, changing the balance of B to 1 and the balance of A to 2. In Figure 7.2.5b the newly created node is inserted into the right subtree of B, changing the balance of B to -1 and the balance of A to 2. To maintain a balanced tree it is necessary to perform a transformation on the tree so that

1. The inorder traversal of the transformed tree is the same as for the original tree (that is, the transformed tree remains a binary search tree).
2. The transformed tree is balanced.



(a) Original tree.

(b) Right rotation.

(c) Left rotation.

Figure 7.2.6 Simple rotation on a tree.

Consider the trees of Figures 7.2.6a and b. The tree of Figure 7.2.6b is said to be a ***right rotation*** of the tree rooted at A of Figure 7.2.6a. Similarly, the tree of Figure 7.2.6c is said to be a ***left rotation*** of the tree rooted at A of Figure 7.2.6a.

An algorithm to implement a left rotation of a subtree rooted at p is as follows:

```
q = right(p);
hold = left(q);
left(q) = p;
right(p) = hold;
```

Let us call this operation *leftrotation(p)*. *rightrotation(p)* may be defined similarly. Of course, in any rotation the value of the pointer to the root of the subtree being rotated must be changed to point to the new root. (In the case of the foregoing left rotation, this new root is q.) Note that the order of the nodes in an inorder traversal is preserved under both right and left rotations. It therefore follows that any number of rotations (left or right) can be performed on an unbalanced tree to obtain a balanced tree, without disturbing the order of the nodes in an inorder traversal.

Let us now return to the trees of Figure 7.2.5. Suppose that a right rotation is performed on the subtree rooted at a in Figure 7.2.5a. The resulting tree is shown in Figure 7.2.7a. Note that the tree of Figure 7.2.7a yields the same inorder traversal as that of Figure 7.2.5a and is also balanced. Also, since the height of the subtree of Figure 7.2.5a was $n + 2$ before the insertion and the height of the subtree of Figure 7.2.7a is $n + 2$ with the inserted node, the balance of each ancestor of node A remains undisturbed. Thus replacing the subtree of Figure 7.2.5a with its right rotation of Figure 7.2.7a guarantees that a balanced binary search tree is maintained.

Let us now turn to the tree of Figure 7.2.5b, where the newly created node is inserted into the right subtree of B. Let C be the right son of B. (There are three cases: C may be the newly inserted node, in which case $n = -1$, or the newly inserted node may be in the left or right subtree of C. Figure 7.2.5b illustrates the case where it is in the left subtree; the analysis of the other cases is analogous.) Suppose that a left rotation on the subtree rooted at B is followed by a right rotation on the subtree rooted at A. Figure 7.2.7b illustrates the resulting tree. Verify that the inorder traversals of the two trees are the same and that the tree of Figure 7.2.7b is balanced. The height of the tree in Figure 7.2.7b is $n + 2$, which is the same as the height of the tree in Figure 7.2.5b before the insertion, so that the balance in all ancestors of A is unchanged. Therefore by replacing the tree of Figure 7.2.5b with that of Figure 7.2.7b wherever it occurs after insertion, a balanced search tree is maintained.

Let us now present an algorithm to search and insert into a nonempty balanced binary tree. Each node of the tree contains five fields: k and r, which hold the key and record respectively, *left* and *right*, which are pointers to the left and right subtrees respectively, and *bal*, whose value is 1, -1, or 0, depending on the node's balance. In the first part of the algorithm, if the desired key is not found in the tree already, a new node is inserted into the binary search tree without regard to balance. This first phase also keeps track of the youngest ancestor, *ya*, that may become unbalanced upon insertion. The algorithm makes use of the function *maketree* described previously and routines *rightrotation* and *leftrotation*, which accept a pointer to the root of a subtree and perform the desired rotation.

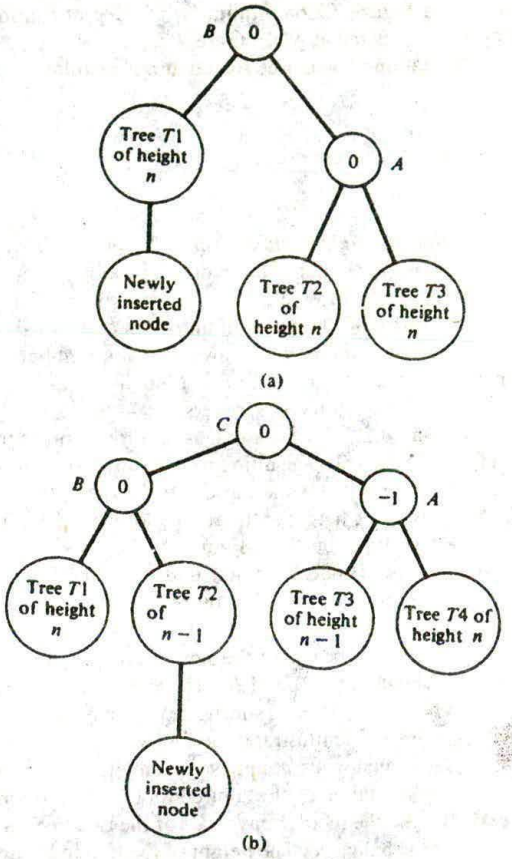


Figure 7.2.7 After rebalancing, all balances are after insertion.

```
/* PART I: search and insert into the binary tree      */
fp = null;
p = tree;
fya = null;
ya = p;
/* ya points to the youngest ancestor which may become  */
/* unbalanced. fya points to the father of ya, and fp   */
/*             points to the father of p.               */
while (p != null) {
   if (key == k(p))
      return(p);
   q = (key < k(p))  ?  left(p) : right(p);
```

```
      if (q != null)
         if (bal(q) != 0) {
            fya = p;
            ya = q;
         } /* end if */
      fp = p;
      p = q;
   } /* end while */
   /*                  insert new record                    */
   q = maketree(rec, key);
   bal(q) = 0;
   (key < k(fp)) ? left(fp) = q : right(fp) = q;
   /* the balance on all nodes between node(ya) and node(q) */
   /*                  must be changed from 0               */
   p = (key < k(ya)) ? left(ya) : right(ya);
   s = p;
   while (p != q) {
      if (key < k(p)) {
         bal(p) = 1;
         p = left(p);
      }
      else {
         bal(p) = -1;
         p = right(p);
      } /* end if */
   } /* end while */
   /*    PART II: ascertain whether or not the tree is     */
   /*  unbalanced. If it is, q is the newly inserted node, */
   /*   ya is its youngest unbalanced ancestor, fya is the */
   /*  father of ya and s is the son of ya in the direction */
   /*                    of the imbalance                   */
   imbal = (key < k(ya)) ? 1 : -1;
   if (bal(ya) == 0) {
      /*  another level has been added to the tree  */
      /*        the tree remains balanced           */
      bal(ya) = imbal;
      return(q);
   } /* end if */
   if (bal(ya) != imbal) {
      /*  the added node has been placed in the  */
      /*  opposite direction of the imbalance    */
      /*        the tree remains balanced        */
      bal(ya) = 0;
      return(q);
   } /* end if */
   /*  PART III: the additional node has unbalanced the tree */
   /*  rebalance it by performing the required rotations and */
   /*     then adjust the balances of the nodes involved     */
   if (bal(s) == imbal) {
      /* ya and s have been unbalanced in the same direction; */
```

```
    /*      see Figure 7.2.5a where ya = a and s = b        */
    p = s;
    if (imbal == 1)
       rightrotation(ya);
    else
       leftrotation(ya);
    bal(ya) = 0;
    bal(s) = 0;
 }
 else {
    /*  ya and s are unbalanced in opposite directions; */
    /*              see Figure 7.2.5b                   */
    if (imbal == 1) {
       p = right(s);
       leftrotation(s);
       left(ya) = p;
       rightrotation(ya);
    }
    else {
       p = left(s);
       right(ya) = p;
       rightrotation(s);
       leftrotation(ya);
    } /* end if */
    /*   adjust bal field for involved nodes  */
    if (bal(p) == 0) {
       /*  p was inserted node  */
       bal(ya) = 0;
       bal(s) = 0;
    }
    else
       if (bal(p) == imbal) {
          /*    see Figures 7.2.5b and 7.2.7b    */
          bal(ya) = -imbal;
          bal(s) = 0;
       }
       else {
          /*    see Figures 7.2.5b and 7.2.7b    */
          /*  but the new node was inserted into t3  */
          bal(ya) = 0;
          bal(s) = imbal;
       } /* end if */
    bal(p) = 0;
 } /* end if */
 /*   adjust the pointer to the rotated subtree   */
 if (fya == null)
    tree = p;
 else
    (ya == right(fya))  ?  right(fya) = p : left(fya) = p;
 return(q);
```

The maximum height of a balanced binary search tree is $1.44 \log_2 n$, so that a search in such a tree never requires more than 44 percent more comparisons than that for a completely balanced tree. In actual practice, balanced binary search trees behave even better, yielding search times of $\log_2 n + 0.25$ for large n. On the average, a rotation is required in 46.5 percent of the insertions.

The algorithm to delete a node from a balanced binary search tree while maintaining its balance is even more complex. Whereas insertion requires at most a double rotation, deletion may require one (single or double) rotation at each level of the tree, or $O(\log n)$ rotations. However, in practice, an average of only 0.214 (single or double) rotations has been found to be required per deletion.

The balanced binary search trees that we have looked at are called ***height-balanced trees*** because their height is used as the criterion for balancing. There are a number of other ways of defining balanced trees. In one method, the ***weight*** of a tree is defined as the number of external nodes in the tree (this equals the number of null pointers in the tree). If the ratio of the weight of the left subtree of every node to the weight of the subtree rooted at the node is between some fraction a and $1 - a$, the tree is a ***weight-balanced tree of ratio a*** or is said to be in the class $wb[a]$. When an ordinary insertion or deletion on a tree in class $wb[a]$ would remove the tree from the class, rotations are used to restore the weight-balanced property.

Another type of tree, called a ***balanced binary tree*** by Tarjan, requires that for every node nd, the length of the longest path from nd to an external node is at most twice the length of the shortest path from nd to an external node. (Recall that external nodes are nodes added to the tree at every null pointer.) Again, rotations are used to maintain balance after insertion or deletion. Tarjan's balanced trees have the property that at most one double and one single rotation restore balance after either insertion or deletion, as opposed to a possible $O(\log n)$ rotations for deletion in a height-balance tree.

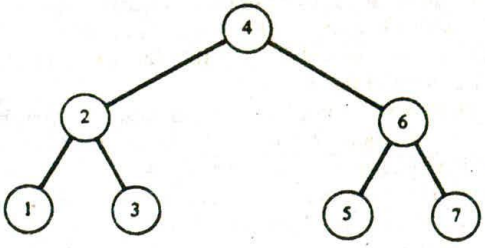
Balanced trees may also be used for efficient implementation of priority queue (see Sections 4.1 and 6.3). Inserting a new element requires at most $O(\log n)$ steps to find its position and $O(1)$ steps to access the element (by following left pointers to the leftmost leaf) and $O(\log n)$ or $O(1)$ steps to delete that leaf. Thus, like a priority queue implemented using a heap (Section 6.3), a priority queue implemented using a balanced tree can perform any sequence of n insertions and minimum deletions in $O(n \log n)$ steps.

EXERCISES

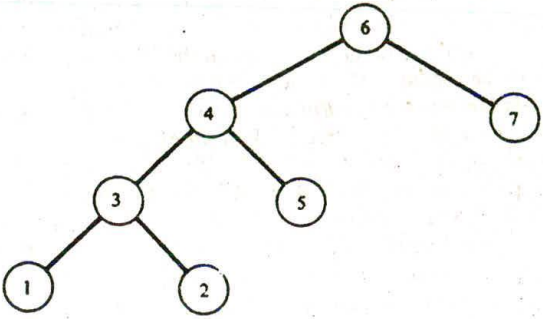
7.2.1. Write an efficient insertion algorithm for a binary search tree to insert a new record whose key is known not to exist in the tree.

7.2.2. Show that it is possible to obtain a binary search tree in which only a single leaf exists even if the elements of the tree are not inserted in strictly ascending or descending order.

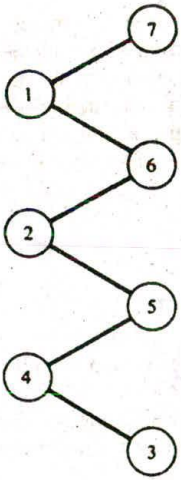
7.2.3. Verify by simulation that if records are presented to the binary tree search and insertion algorithm in random order, the number of key comparisons is $O(\log n)$.



(a)



(b)



(c)

Figure 7.2.8

7.2.4. Prove that every n-node binary search tree is not equally likely (assuming items are inserted in random order), and that balanced trees are more probable than straight-line trees.

7.2.5. Write an algorithm to delete a node from a binary tree that replaces the node with its inorder predecessor, rather than its inorder successor.

7.2.6. Suppose that the node type of a binary search tree is defined as follows:

```
struct nodetype {
   int k;
   int r;
   struct nodetype *left;
   struct nodetype *right;
}
```

The k and r fields contain the key and record of the node; *left* and *right* are pointers to the node's sons. Write a C routine *sinsert*(*tree*, *key*, *rec*) to search and insert a record *rec* with key *key* into a binary search tree pointed to by *tree*.

7.2.7. Write a C routine *sdelete*(*tree*, *key*) to search and delete a record *record* with key *key* from a binary search tree implemented as in the previous exercise. If such a record is found, the function returns the value of its r field; if it is not found, the function returns 0.

7.2.8. Write a C routine *delete*(*tree*, *key*1, *key*2) to delete all records with keys between *key*1 and *key*2 (inclusive) from a binary search tree whose nodes are declared as in the previous exercises.

7.2.9. Consider the search trees of Figure 7.2.8.

(a) How many permutations of the integers 1 through 7 would produce the binary search trees of Figure 7.2.8a, b, and c, respectively?

(b) How many permutations of the integers 1 through 7 produce binary search trees similar to the trees of Figure 7.2.8a, b, and c, respectively? (See Exercise 5.1.9.)

(c) How many permutations of the integers 1 through 7 produce binary search trees with the same number of nodes at each level as the trees of Figure 7.2.8a, b, and c, respectively?

(d) Find an assignment of probabilities to the first seven positive integers as search arguments that makes each of the trees of Figure 7.2.8a, b, and c optimum.

7.2.10. Show that the Fibonacci tree of order $h + 1$ (see Exercise 5.3.5) is a height-balanced tree of height h and has fewer nodes than does any other height balanced tree of height h.

7.3 GENERAL SEARCH TREES

General nonbinary trees are also used as search tables, particularly in external storage. There are two broad categories of such trees: multiway search trees and digital search trees. We examine each in turn.

Multiway Search Trees

In a binary search tree, each node *nd* contains a single key and points to two subtrees. One of these subtrees contains all the keys in the tree rooted at *nd* that are less

than the key in *nd*, and the other subtree contains all the keys in the tree rooted at *nd* that are greater than (or equal to) the key in *nd*.

We may extend this concept to a general search tree in which each node contains one or more keys. A ***multiway search tree of order n*** is a general tree in which each node has n or fewer subtrees and contains one fewer key than it has subtrees. That is, if a node has four subtrees, it contains three keys. In addition, if $s_0, s_1, \ldots, s_{m-1}$ are the m subtrees of a node containing keys $k_0, k_1, \ldots, k_{m-2}$ in ascending order, all keys in subtree s_0 are less than or equal to k_0, all keys in the subtree s_j (where j is between 1 and $m - 2$) are greater than k_{j-1} and less than or equal to k_j, and all keys in the subtree s_{m-1} are greater than k_{m-2}. The subtree s_j is called the ***left subtree*** of key k_j and its root is called the ***left son*** of key k_j. Similarly, s_j is called the ***right subtree***, and its root the ***right son***, of key k_{j-1}. One or more of the subtrees of a node may be empty. (Sometimes, the term "multiway search tree" is used to refer to any nonbinary tree used for searching, including the digital trees that we introduce at the end of this section. However, we use the term strictly for trees that can contain complete keys in each node.)

Figure 7.3.1 illustrates a number of multiway search trees. Figure 7.3.1a is a multiway search tree of order 4. The eight nodes of that tree have been labeled *A* through *H*. Nodes *A*, *D*, *E*, and *G* contain the maximum number of subtrees, 4, and the maximum number of keys, 3. Such nodes are called ***full nodes***. However, some of the subtrees of nodes *D* and *E* and all of the subtrees of node *G* are empty, as indicated by arrows emanating from the appropriate positions in the nodes. Nodes *B*, *C*, *F*, and *H* are not full and also contain some empty subtrees. The first subtree of *A* contains the keys 6 and 10, both smaller than 12, which is the first key of *A*. The second subtree of *A* contains 25 and 37, both greater than 12 (the first key of *A*) and less than 50 (the second key). The third subtree contains 60, 70, 80, 62, 65, and 69, all of which are between 50 (the second key of *A*) and 85 (the third key). Finally, the last subtree of *A* contains 100, 120, 150, and 110, all greater than 85, the last key in node *A*. Similarly, each subtree of any other node contains only keys between the appropriate two keys of that node and its ancestors.

Figure 7.3.1b illustrates a ***top-down multiway search tree*** of order 3. Such a tree is characterized by the condition that any nonfull node is a leaf. Note that the tree of Figure 7.3.1a is not top-down, since node *C* is nonfull, yet contains a nonempty subtree. Define a ***semileaf*** as a node with at least one empty subtree. In Figure 7.3.1a, nodes *B* through *H* are all semileaves. In Figure 7.3.1b, nodes *B* through *G* and *I* through *R* are semileaves. In a top-down multiway tree, a semileaf must be either full or a leaf.

Figure 7.3.1c is yet another multiway search tree of order 3. It is not top-down, since there are four nodes with only one key and nonempty subtrees. However, it does have another special property in that it is ***balanced***. That is, all its semileaves are at the same level (3). This implies that all semileaves are leaves. Neither the tree of Figure 7.3.1a (which has leaves at levels 1 and 2) nor that of Figure 7.3.1b (leaves at levels 2, 3, and 4) are balanced multiway search trees. (Note that although a binary search tree is a multiway search tree of order 2, a balanced binary search tree as defined at the end of Section 7.2 is not necessarily balanced as a multiway search tree, since it can have leaves at different levels.)

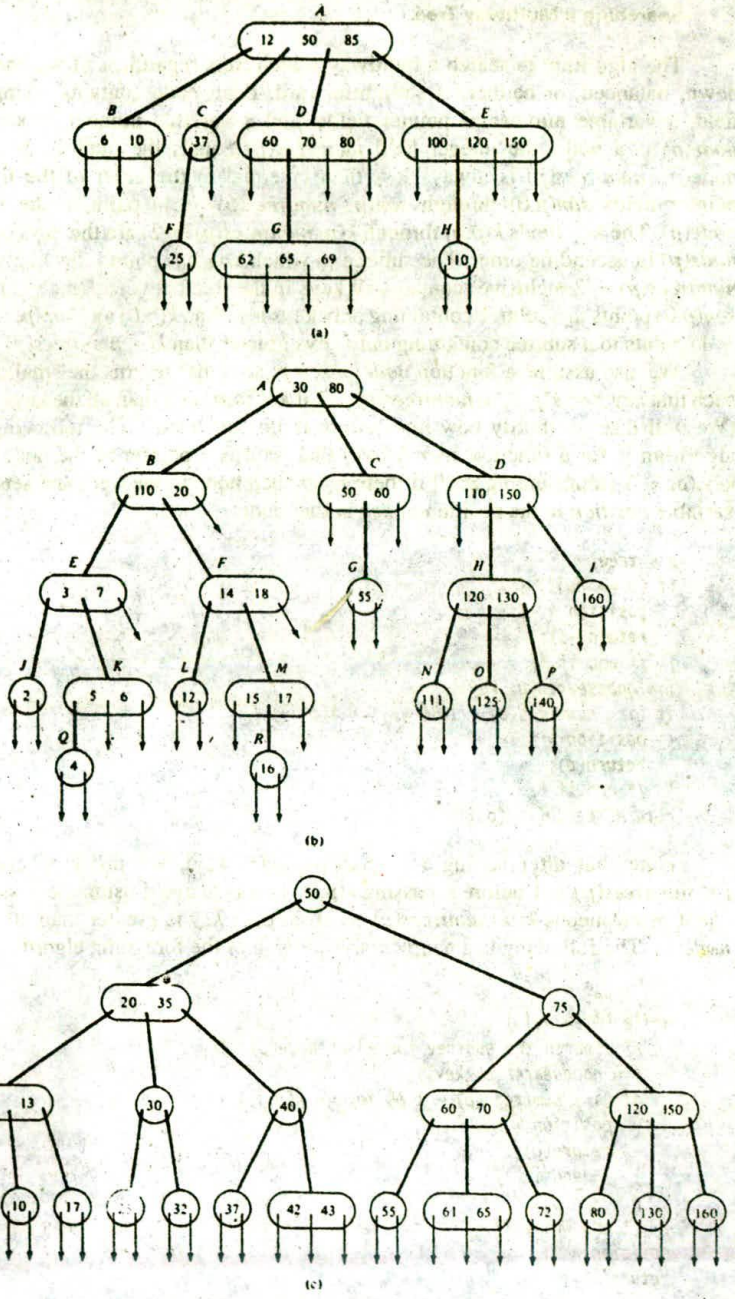


Figure 7.3.1 Multiway search trees.

Searching a Multiway Tree

The algorithm to search a multiway search tree, regardless of whether it is top-down, balanced, or neither, is straightforward. Each node contains a single integer field, a variable number of pointer fields, and a variable number of key fields. If *node(p)* is a node, the integer field *numtrees(p)* equals the number of subtrees of *node(p)*. *numtrees(p)* is always less than or equal to the order of the tree, *n*. The pointer fields *son(p,0)* through *son(p, numtrees(p) − 1)* point to the subtrees of *node(p)*. The key fields *k(p,0)* through *k(p, numtrees(p) − 2)* are the keys contained in *node(p)* in ascending order. The subtree to which *son(p,i)* points (for *i* between 1 and *numtrees(p) − 2* inclusive) contains all keys in the tree between *k(p,i − 1)* and *k(p,i)*. *son(p,0)* points to a subtree containing only keys less than *k(p,0)* and *son(p, numtrees(p) − 1)* points to a subtree containing only keys greater than *k(p, numtrees(p) − 2)*.

We also assume a function *nodesearch(p, key)* that returns the smallest integer *j* such that *key* <= *k(p,j)*, or *numtrees(p) − 1* if *key* is greater than all the keys in *node(p)*. (We will discuss shortly how *nodesearch* is implemented.) The following recursive algorithm is for a function *search(tree)* that returns a pointer to the node containing *key*.(or − 1 [representing *null*] if there is no such node in the tree) and sets the global variable *position* to the position of *key* in that node:

```
p = tree;
if (p == null) {
   position = -1;
   return(-1);
} /* end if */
i = nodesearch(p, key);
if (i < numtrees(p) - 1 && key == k(p,i)) {
   position = i;
   return(p);
} /* end if */
return(search(son(p,i)));
```

Note that after setting *i* to *nodesearch(p, key)*, we insist on checking that *i < numtrees(p) − 1* before accessing *k(p,i)*. This is to avoid using the possibly nonexistent or erroneous *k(p, numtrees(p) − 1)*, in case *key* is greater than all the keys in *node(p)*. The following is a nonrecursive version of the foregoing algorithm:

```
p = tree;
while (p != null) {
   /* search the subtree rooted at node(p) */
   i = nodesearch(p, key);
   if (i < numtrees(p) - 1 && key == k(p,i)) {
      position = i;
      return(p);
   } /* end if */
   p = son(p,i);
} /* end while */
position = -1;
return(-1);
```

The function *nodesearch* is responsible for locating the smallest key in a node greater than or equal to the search argument. The simplest technique for doing this is a sequential search through the ordered set of keys in the node. If all keys are of fixed equal length, a binary search can also be used to locate the appropriate key. The decision whether to use a sequential or binary search depends on the order of the tree, which determines how many keys must be searched. Another possibility is to organize the keys within the node as a binary search tree.

Implementing a Multiway Tree

Note that we have implemented a multiway search tree of order n using nodes with up to n sons rather than as a binary tree with son and brother pointers, as outlined in Section 5.5 for general trees. The reason for this is that in multiway trees, unlike in general trees, there is a limit to the number of sons of a node, and we can expect most nodes to be as full as possible. In a general tree there was no such limit, and many nodes might contain only one or two items. Therefore the flexibility of allowing as many or as few items in a node as necessary and the space saving when a node was nearly empty was worth the overhead of extra brother pointers.

Nevertheless, when nodes are not full, multiway search trees as implemented here do waste considerable storage. Despite this possible waste of storage, multiway trees are frequently used, especially to store data on an external direct-access device such as a disk. The reason for this is that accessing each new node during a search requires reading a block of storage from the external device. This read operation is relatively expensive in terms of time because of the mechanical work involved in positioning the device properly. However, once the device is positioned, the task of actually reading a large amount of sequential data is relatively fast. This means that the total time for reading a storage block (that is, a "node") is only minimally affected by its size. Once a node has been read and is contained in internal computer memory, the cost of searching it at electronic internal speeds is minuscule compared with the cost of initially reading it into memory. In addition, external storage is fairly inexpensive so that a technique that improves time-efficiency at the expense of external storage space utilization is real-cost (that is, dollars) effective. For this reason, external storage systems based on multiway search trees try to maximize the size of each node, and trees of order 200 or more are not uncommon.

The second factor to consider in implementing multiway search trees is storage of the data records themselves. As in any storage system, the records may be stored with the keys or remotely from the keys. The first technique requires keeping entire records within the tree nodes, whereas the second requires keeping a pointer to the associated record with each key in a node. (Still another technique involves duplicating keys and keeping records only at the leaves. This mechanism is discussed later in more detail when we discuss B^+-trees.)

In general we would like to keep as many keys as possible in each node. To see why this is so, consider two top-down trees with 4000 keys and minimum depth, one of order 5 and the other of order 11. The order-5 tree requires 1000 nodes (of 4 keys each) to hold the 4000 keys, whereas the order-11 tree requires only 400 nodes (of 10 keys each). The depth of the order-5 tree is at least 5 (level 0 contains 1 node, level 1 contains 5, level 2 contains 25, level 3 contains 125, level 4 contains 625, and level 5

contains the remaining 219), whereas the depth of the order-11 tree can be as low as 3 (level 0 contains 1 node, level 1 contains 11, level 2 contains 121, and level 3 contains the remaining 267). Thus 5 or 6 nodes must be accessed in searching the order-5 tree for most of the keys, but only 3 or 4 nodes must be accessed for the order-11 tree. But as we noted above, accessing a node is the most expensive operation in searching external storage, where multiway trees are most used. Thus a tree with a higher order leads to a more efficient search process. The actual storage required by both situations is approximately the same, since, although fewer large nodes are required to hold a file of a given size when the order is high, each node is larger.

Since the size of a node is usually fixed by other external factors (for example, the amount of storage physically read from disk in one operation), a higher order tree is obtained by keeping the records outside the tree nodes. Even if this causes an extra external read to obtain a record after its key has been located, keeping records within a node typically reduces the order by a factor of between 5 and 40 (which is the typical range of the ratio of record size to key size), so the trade-off is not worthwhile.

If a multiway search tree is maintained in external storage, a pointer to a node is an external storage address that specifies the starting point of a storage block. The block of storage that makes up a node must be read into internal storage before any of the fields *numtrees*, *k*, or *son* can be accessed. Assume that the routine *directread(p, block)* reads a node at external storage address *p* into an internal storage buffer *block*. Assume also that the *numtrees*, *k*, and *son* fields in the buffer are accessed by the C-like notations *block.numtrees*, *block.k*, and *block.son*. Assume also that the function *nodesearch* is modified to accept an (internal) storage block rather than a pointer [that is, it is invoked by *nodesearch(block, key)* rather than by *nodesearch(p, key)*]. Then the following is a nonrecursive algorithm for searching an externally stored multiway search tree:

```
p = tree;
while (p != null) {
   directread(p, block);
   i = nodesearch(block, key);
   if (i < block.numtrees - 1 && key == block.son(i)) {
      position = i;
      return(p);
   } /* end if */
   p = block.son(i);
} /* end while */
position = -1;
return(-1);
```

The algorithm also sets *block* to the node at external address *p*. The record associated with *key* or a pointer to it may be found in *block*. Note that *null* as used in this algorithm references a null external storage address rather than the C pointer *null*.

Traversing a Multiway Tree

A common operation on a data structure is ***traversal***: accessing all the elements of the structure in a fixed sequence. The following is a recursive algorithm *traverse(tree)* to traverse a multiway tree and print its keys in ascending order

```
if (tree != null) {
   nt = numtrees(tree);
   for (i = 0; i < nt - 1; i++) {
      traverse(son(tree, i));
      printf("%d", k(tree, i));
   } /* end for */
   traverse(son(tree, nt));
} /* end if */
```

In implementing the recursion, we must keep a stack of pointers to all the nodes in a path beginning with the root of the tree down to the node currently being visited.

If each node is a block of external storage and *tree* is the root node's external storage address, a node must be read into internal memory before its *son* or *k* fields can be accessed. Thus the algorithm becomes

```
if (tree != null) {
   directread(tree, block);
   nt = block.numtrees;
   for (i = 0; i < nt - 1; i++) {
      traverse(block.son(i));
      printf("%d", block.k(i));
   } /* end for */
   traverse(block.son(nt));
} /* end if */
```

where *directread* is a system routine that reads a block of storage at a particular external address (*tree*) into an internal memory buffer (*block*). This requires keeping a stack of buffers as well. If d is the depth of the tree, $d + 1$ buffers must be kept in memory.

Alternatively, all but one of the buffers can be eliminated if each node contains two additional fields: a *father* field pointing to its father, and an *index* field indicating which son of its father the node is. Then when the last subtree of a node has been traversed, the algorithm uses the *father* field of the node to access its father and its *index* field to determine which key in the father node to output and which subtree of the father to traverse next. However, this would require numerous reads for each node and is probably not worth the savings in buffer space, especially since a high-order tree with a very large number of keys requires very low depth. (As previously illustrated, a tree of order 11 with 4000 keys can be accommodated comfortably with depth 3. A tree of order 101 can accommodate over a million keys with a depth of only 2.)

Another common operation, closely related to traversal, is ***direct sequential access***. This refers to accessing the next key following a key whose location in the tree is known. Let us assume that we have located a key $k1$ by searching the tree and that it is located at position $k(n1, i1)$. Ordinarily, the successor of $k1$ can be found by executing the following routine *next*$(n1, i1)$. (*nullkey* is a special value indicating that a proper key cannot be found.)

```
p = son(n1, i1 + 1);
q = null;  /*  q is one node behind p  */
while (p != null) {
   q = p;
   p = son(p, 0);
} /* end while */
if (q != null)
   return(k(q, 0));
if (i1 < numtrees(n1) - 2)
   return(k(n1, i1 + 1));
return(nullkey);
```

This algorithm relies on the fact that the successor of $k1$ is the first key in the subtree that follows $k1$ in *node*($n1$), or if that subtree is empty [*son*($n1$,$i1$ + 1) equals *null*] and if $k1$ is not the last key in its node ($i1 < numtrees(n1) - 2$), the successor is the next key in *node*($n1$).

However, if $k1$ is the last key in its node and if the subtree following it is empty, its successor can only be found by backing up the tree. Assuming *father* and *index* fields in each node as outlined previously, a complete algorithm *successor*($n1$,$i1$) to find the successor of the key in position $i1$ of the node pointed to by $n1$ may be written as follows:

```
p = son(n1, i1 + 1);
if (p != null    i1 < numtrees(n1) - 2)
   /*  use the previous algorithm  */
   return(next(n1, i1));
f = father(n1);
i = index(n1);
while (f != null && i == numtrees(f) - 1) {
   i = index(f);
   f = father(f);
} /* end while */
if (f == null)
   return(NULLKEY);
return(k(f, i));
```

Of course, we would like to avoid backing up the tree whenever possible. Since a traversal beginning at a specific key is quite common, the initial search process is often modified to retain, in internal memory, all nodes in the path from the tree root to the located key. Then, if the tree must be backed up, the path to the root is readily available. As noted previously, if this is done, the fields *father* and *index* are not needed.

Later in this section, we examine a specialized adaptation of a multiway search tree, called a B$^+$-tree, that does not require a stack for efficient sequential traversal.

Insertion in a Multiway Search Tree

Now that we have examined how to search and traverse multiway search trees, let us examine insertion techniques for these structures. We examine two insertion tech-

niques for multiway search trees. The first is analogous to binary search tree insertion and results in a top-down multiway search tree. The second is a new insertion technique and produces a special kind of balanced multiway search tree. It is this second technique, or a slight variation thereof, that is most commonly used in direct-access external file storage systems.

For convenience we assume that duplicate keys are not permitted in the tree, so that if the argument *key* is found in the tree no insertion takes place. We also assume that the tree is nonempty. The first step in both insertion procedures is to search for the argument *key*. If the argument *key* is found in the tree, we return a pointer to the node containing the key and set the variable *position* to its position within the node, just as in the search procedure presented previously. However, if the argument *key* is not found, we return a pointer to the semileaf *node*(s) that would contain the key if it were present, and *position* is set to the index of the smallest key in *node*(s) that is greater than the argument *key* (that is, the position of the argument *key* if it were in the tree). If all the keys in *node*(s) are less than the argument *key*, *position* is set to *numtrees*(s) − 1. A variable, *found*, is set to *true* or *false* depending on whether the argument key is or is not found in the tree.

Figure 7.3.2 illustrates the result of this procedure for an order-4 top-down balanced multiway search tree and several search arguments. The algorithm for *find* is straightforward:

```
q = null;
p = tree;
while (p != null) {
   i = nodesearch(p, key);
   q = p;
   if (i < numtrees(p) - 1 && key == k(p, i)) {
      found = TRUE;
      position = i;
      return(p);  /*  the key is found in node(p)   */
   } /* end if */
   p = son(p, i);
} /* end while */
found = FALSE;
position = i;
return(q);  /*  p is null. q points to a semileaf   */
```

To implement this algorithm in C, we would write a function *find* with the following header:

```
NODEPTR find(NODEPTR tree, KEYTYPE key, int *pposition, int *pfound)
```

References to *position* and *found* in the algorithm are replaced by references to **pposition* and **pfound*, respectively, in the C function.

Let us assume that s is the node pointer returned by *find*. The second step of the insertion procedure applies only if the key is not found (remember, no duplicate keys are permitted) and if *node*(s) is not full (that is, if *numtrees*(s) < n, where n is the order

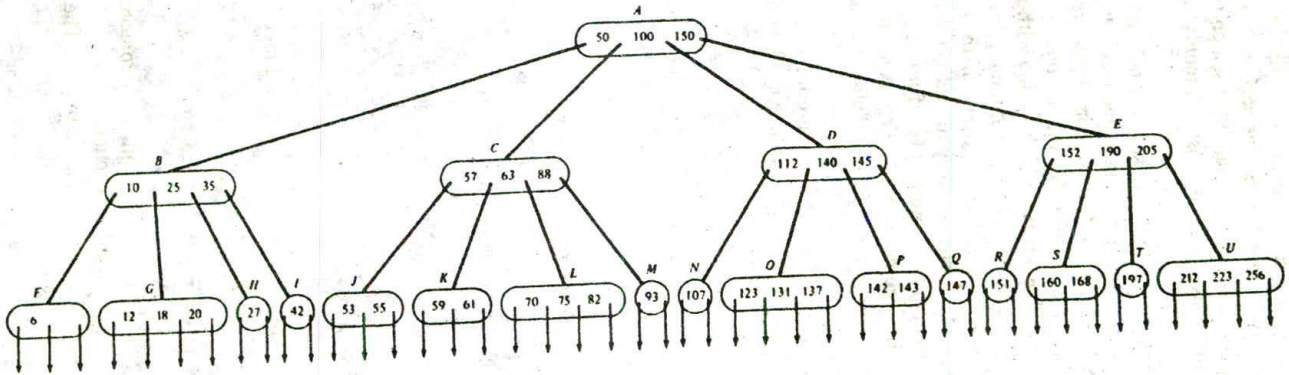

A
50 100 150
B
10 25 35
C
57 63 88
D
112 140 145
E
152 190 205
F
6
G
12 18 20
H
27
I
42
J
53 55
K
59 61
L
70 75 82
M
93
N
107
O
123 131 137
P
142 143
Q
147
R
151
S
160 168
T
197
U
212 223 256
key = 63
search = C
position = 2
found = true
(a)
key = 137
search = O
position = 3
found = true
(b)
key = 71
search = L
position = 2
found = false
(c)
key = 102
search = N
position = 1
found = false
(d)
key = 22
search = G
position = 4
found = false
(e)
key = 148
search = Q
position = 2
found = false
(f)

Figure 7.3.2

of the tree). In Figure 7.3.2, this applies to cases d and f only. The second step consists of inserting the new key (and record) into *node*(*s*). Note that if the tree is top-down or balanced, a nonfull semileaf discovered by *find* is always a leaf. Let *insrec*(*p*, *i*, *rec*) be a routine to insert the record *rec* in position *i* of *node*(*p*) as appropriate. Then the second step of the insertion process may be described as follows:

```
nt = numtrees(s);
numtrees(s) = nt + 1;
for (i = nt - 1; i > position; i--)
   k(s, i) = k(s, i - 1);
k(s, position) = key;
insrec(s, position, rec);
```

We call this function *insleaf*(*s*, *position*, *key*, *rec*).

Figure 7.3.3a illustrates the nodes located by the *find* procedure in Figure 7.3.2d and f with the new keys inserted. Note that it is unnecessary to copy the son pointers associated with the keys being moved, since the node is a leaf, so that all pointers are null. We assume that they were initialized to null when the node was initially added to the tree.

If step 2 is appropriate (that is, if a nonfull leaf node where the key can be inserted has been found), both insertion routines terminate. The two techniques differ only in the third step, which is invoked when the find procedure locates a full semileaf.

The first insertion technique, which results in top-down multiway search trees, mimics the actions of the binary search tree insertion algorithm. That is, it allocates a new node, inserts the key and record into the new node, and places the new node as the appropriate son of *node*(*s*). It uses the routine *maketree*(*key*, *rec*) to allocate a node, set the *n* pointers in it to *null*, its *numtrees* field to 2, and its first key field to *key*. *maketree* then calls *insrec* to insert the record as appropriate and finally returns a pointer to the newly allocated node. Using *maketree*, the routine *insfull* to insert the key when the appropriate semileaf is full may be implemented trivially as

```
p = maketree(key, rec);
son(s, position) = p;
```

If *father* and *index* fields are maintained in each node, the operations

```
father(p) = s;
index(p) = position;
```

are required as well.

Figure 7.3.3b illustrates the result of inserting keys 71 and 22, respectively, into the nodes located by *find* in Figure 7.3.2c and e. Figure 7.3.3c illustrates subsequent insertions of keys 86, 77, 87, 84, 85, and 73, in that order. Note that the order in which keys are inserted very much affects where the keys are placed. For example, consider what would happen if the keys were inserted in the order 85, 77, 86, 87, 73, 84.

Note also that this insertion technique can transform a leaf into a nonleaf (although it remains a semileaf) and therefore unbalances the multiway tree. It is therefore possi-

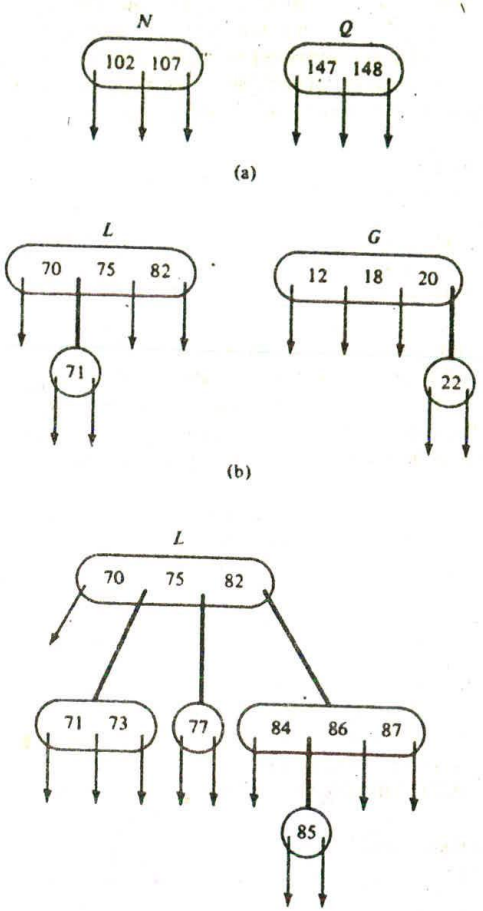


Figure 7.3.3

ble for successive insertions to produce a tree that is heavily unbalanced and in which an inordinate number of nodes must be accessed to locate certain keys. In practical situations, however, multiway search trees created by this insertion technique, although not completely balanced, are not too greatly unbalanced, so that too many nodes are not accessed in searching for a key in a leaf. However, the technique does have one major drawback. Since leaves are created containing only one key, and other leaves may be created before previously created leaves are filled, multiway trees created by successive insertions in this manner waste much space with leaf nodes that are nearly empty.

Although this insertion method does not guarantee balanced trees, it does guarantee top-down trees. To see this, note that a new node is not created unless its father

is full. Thus any nonfull node has no descendants and is therefore a leaf, which by definition implies that the tree is top-down. The advantage of a top-down tree is that the upper nodes are full, so that as many keys as possible are found on short paths.

Before examining the second insertion technique, we put all the pieces of the first technique together to form a complete search and insertion algorithm for top-down multiway search trees.

```
if (tree == null) {
   tree = maketree(key, rec);
   position = 0;
   return (tree);
} /* end if */
s = find(tree, key, position, found);
if (found == TRUE)
   return (s);
if (numtrees(s) < n) {
   insleaf(s, position, key, rec);
   return(s);
} /* end if */
p = maketree(key, rec);
son(s, position) = p;
position = 0;
return (p);
```

B-Trees

The second insertion technique for multiway search trees is more complex. Compensating for this complexity, however, is the fact that it creates balanced trees, so that the maximum number of nodes accessed to find any particular key is kept small. In addition, the technique yields one further bonus, in that all nodes (except for the root) in a tree created by this technique are at least half full, so that very little storage space is wasted. This last advantage is the primary reason that the second insertion technique (or a variation thereof) is used so frequently in actual file systems.

A balanced order-n multiway search tree in which each nonroot node contains at least $(n-1)/2$ keys is called a ***B-tree of order n***. (Note that the slash denotes integer division so that a B-tree of order 12 contains at least 5 keys in each nonroot node, as does a B-tree of order 11.) A B-tree of order n is also called an ***n-(n − 1) tree*** or an ***(n − 1)-n tree***. (The dash outside the parentheses is a hyphen while the dash inside the parentheses is a minus sign.) This reflects the fact that each node in the tree has a maximum of $n - 1$ keys and n sons. Thus, a 4-5 tree is a B-tree of order 5, as is a 5-4 tree. In particular, a 2-3 (or 3-2) tree is the most elementary nontrivial (that is, nonbinary) B-tree, with one or two keys per node and two or three sons per node.

(At this point, we should say something about terminology. In discussing B-trees, the word "order" is used differently by different authors. It is common to find the ***order*** of a B-tree defined as the minimum number of keys in a nonroot node [that is, $(n-1)/2$], and the ***degree*** of a B-tree to mean the maximum number of sons [that is, n]. Still other authors use "order" to mean the maximum number of keys in a node [that is, $n - 1$].

We use ***order*** consistently for all multiway search trees to mean the maximum number of sons.)

The first two steps of the insertion technique are the same for B-trees as for top-down trees. First, use *find* to locate the leaf into which the key should be inserted, and second, if the located leaf is not full, add the key using *insleaf*. It is in the third step, when the located leaf is found to be full, that the methods differ. Instead of creating a new node with only one key, split the full leaf in two: a left leaf and a right leaf. For simplicity, assume that n is odd. The n keys consisting of the $n - 1$ keys in the full leaf and the new key to be inserted are divided into three groups: the lowest $n/2$ keys are placed into the left leaf, the highest $n/2$ keys are placed into the right leaf, and the middle key [there must be a middle key since $2 * (n/2)$ equals $n - 1$ if n is odd] is placed into the father node if possible (that is, if the father node is not full). The two pointers on either side of the key inserted into the father are set to the newly created left and right leaves, respectively.

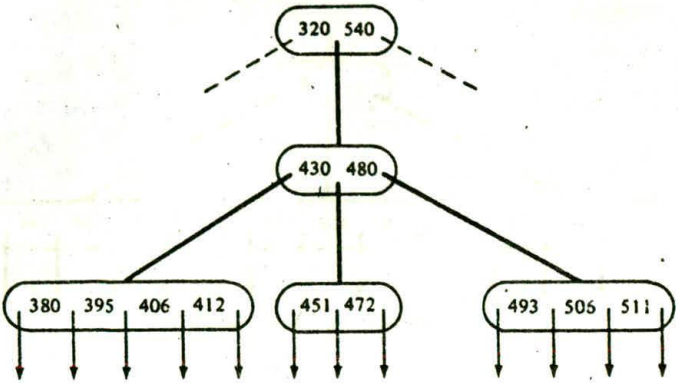
Figure 7.3.4 illustrates this process on a B-tree of order 5. Figure 7.3.4a shows a subtree of a B-tree, and Figure 7.3.4b shows part of the same subtree as it is altered by the insertion of 382. The leftmost leaf was already full, so that the five keys 380, 382, 395, 406, and 412 are divided so that 380 and 382 are placed in a new left leaf, 406 and 412 are placed in a new right leaf, and the middle key, 395, is advanced to the father node with pointers to the left and right leaves on either side. There is no problem placing 395 in the father node, since it contained only two keys and has room for four.

Figure 7.3.4c shows the same subtree with first 518 and then 508 inserted. (The same result would be achieved if they were inserted in reverse order.) It is possible to insert 518 directly in the rightmost leaf, since there is room for one additional key. However, when 508 arrives, the leaf is already full. The five keys 493, 506, 508, 511, and 518 are divided so that the lower two (493 and 506) are placed in a new left leaf, the higher two (511 and 518) in a new right leaf, and the middle key (508) is advanced to the father, which still has room to accommodate it. Note that the key advanced to the father is always the middle key, regardless of whether it arrived before or after the other keys.

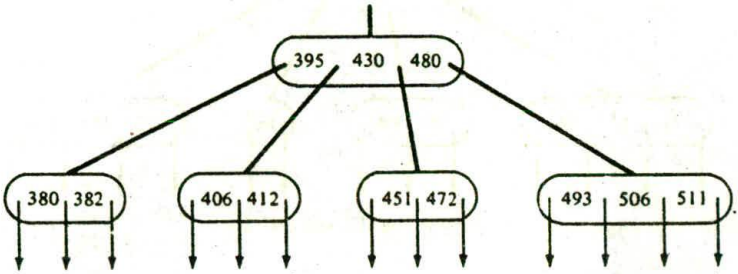
If the order of the B-tree is even, the $n - 1$ keys (excluding the middle key) must be divided into two unequal-sized groups: one of size $n/2$ and the other of size $(n - 1)/2$. [The second group is always of size $(n - 1)/2$, regardless of whether n is odd or even, since when n is odd, $(n - 1)/2$ equals $n/2$.] For example, if n equals 10, 10/2 (or 5) keys are in one group, 9/2 (or 4) keys are in the other group, and one key is advanced, for a total of 10 keys. These may be divided so that the larger-sized group is always in the left leaf or the right leaf, or divisions may be alternated so that on one split, the right leaf contains more keys and on the next split, the left leaf contains more keys. In practice, it makes little difference which technique is used.

Figure 7.3.5 illustrates both left and right biases in a B-tree of order 4. Note that whether a left or right bias is chosen determines which key is to be advanced to the father.

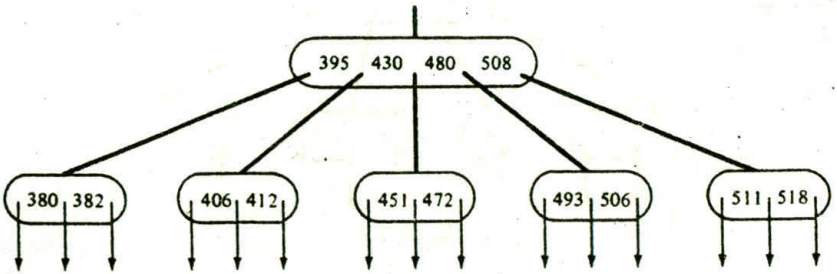
One basis on which the decision can be made as to whether to leave more keys in the left or right leaf is to examine the key ranges under both possibilities. In Figure 7.3.5b, utilizing a left bias, the key range of the left node is 87 to 102, or 15, and the key range of the right node is 102 to 140, or 38. In Figure 7.3.5c, utilizing a right bias,



(a) Initial portion of a B-tree

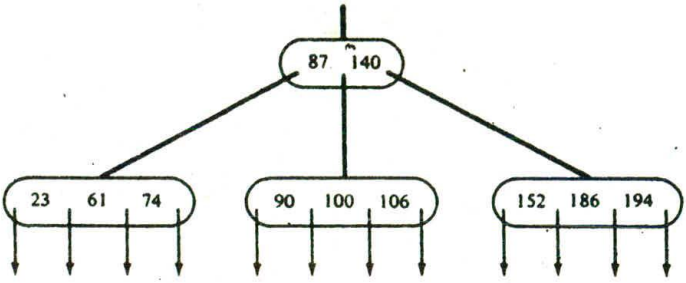


(b) After inserting 382

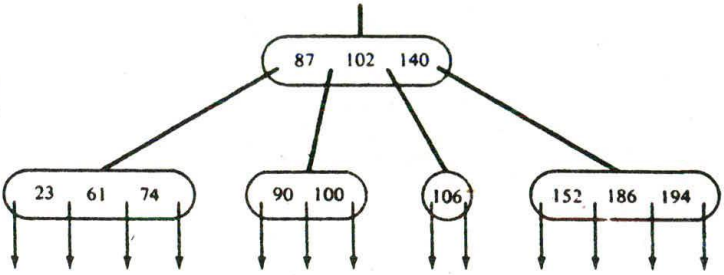


(c) After inserting 518 and 508

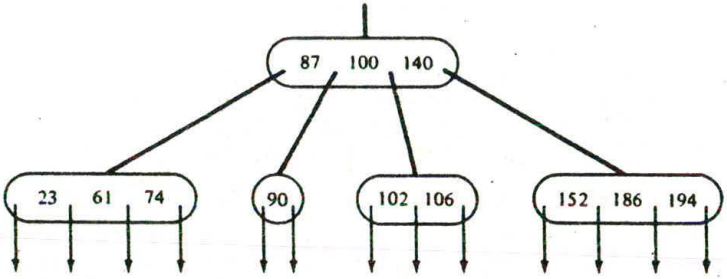
Figure 7.3.4



(a) An initial B-tree twig



(b) Inserting 102 with a left bias



(c) Inserting 102 with a right bias

Figure 7.3.5

the key ranges are 13 (87 to 100) and 40 (100 to 140). Thus we would select a left bias in this case, since it more nearly equalizes the probability of a new key going into the left and right, assuming a uniform distribution of keys.

The entire discussion thus far has assumed that there is room in the father for the middle node to be inserted. What if the father node, too, is full? For example, what

happens with the insertion of Figure 7.3.2c, where 71 must be inserted into the full node *L*, and *C*, the father of *L*, is full as well? The solution is quite simple. The father node is also split in the same way, and its middle node is advanced to its father. This process continues until a key is inserted in a node with room, or the root node, *A*, itself is split. When that happens, a new root node *NR* is created containing the key advanced from the splitting of *A*, and with the two halves of *A* as sons.

Figures 7.3.6 and 7.3.7 illustrate this process with the insertions of Figure 7.3.2c and e. In Figure 7.3.6a, node *L* is split. (We assume a left bias throughout this illustration.) The middle element (75) should be advanced to *C*, but *C* is full. Thus, in Figure 7.3.6b, we see *C* being split as well. The two halves of *L* are now made sons of the appropriate halves of *C*. 25, which had been advanced to *C*, must now be advanced to root node *A*, which also has no room. Thus, *A* itself must be split, as shown in Figure 7.3.6c. Finally, Figure 7.3.6d shows a new root node, *NR*, established, containing the key advanced from *A* and two pointers to the two halves of *A*.

Figure 7.3.7 illustrates the subsequent insertion of 22, as in Figure 7.3.2e. In that figure, 22 would have caused a split of nodes *G*, *B*, and *A*. But, in the meantime, *A* has already been split by the insertion of 71, so the insertion of 22 proceeds as in Figure 7.3.7. First *G* is split and 20 is advanced to *B* (Figure 7.3.7a), which is split in turn (Figure 7.3.7b). 25 is then advanced to *A*1, which is the new father of *B*. But since *A*1 has room, no furthur splits are necessary. 25 is inserted into *A*1 and the insertion is complete (Figure 7.3.7c).

As a final illustration, Figure 7.3.8 shows the insertion of several keys into the order-5 B-tree of Figure 7.3.4c. It would be worthwhile for you to generate a list of keys and continually insert them into an order-5 B-tree to see how it develops.

Note that a B-tree grows in depth through the splitting of the root and the creation of a new root, and in width by the splitting of nodes. Thus B-tree insertion into a balanced tree keeps the tree balanced. However, a B-tree is rarely top-down, since when a full nonleaf node splits, the two nonleaves created are not full. Thus, while the maximum number of accesses to find a key is low (since the tree is balanced), the average number of such accesses may be higher than in a top-down tree in which the upper levels are always full. In simulations, the average number of accesses in searching a random top-down tree has indeed been found slightly lower than in searching a random B-tree because random top-down trees are generally fairly balanced.

One other point to note in a B-tree is that older keys (those inserted first) tend to be closer to the root than younger keys, since they have had more opportunity to be advanced. However, it is possible for a key to remain in a leaf forever even if a large number of locally lower and higher keys are subsequently inserted. This is unlike a top-down tree in which a key in an ancestor node must be older than any key in a descendant node.

Algorithms for B-Tree Insertion

As you might imagine, the algorithm for B-tree insertion is fairly involved. To simplify matters temporarily, let us assume that we can access a pointer to the father of *node*(*nd*) by referring to *father*(*nd*) and the position of the pointer *nd* in *node*(*father*(*nd*)) by *index*(*nd*), so that *son*(*father*(*nd*), *index*(*nd*)) equals *nd*. (This can be implemented most directly by adding *father* and *index* fields to each node, but there

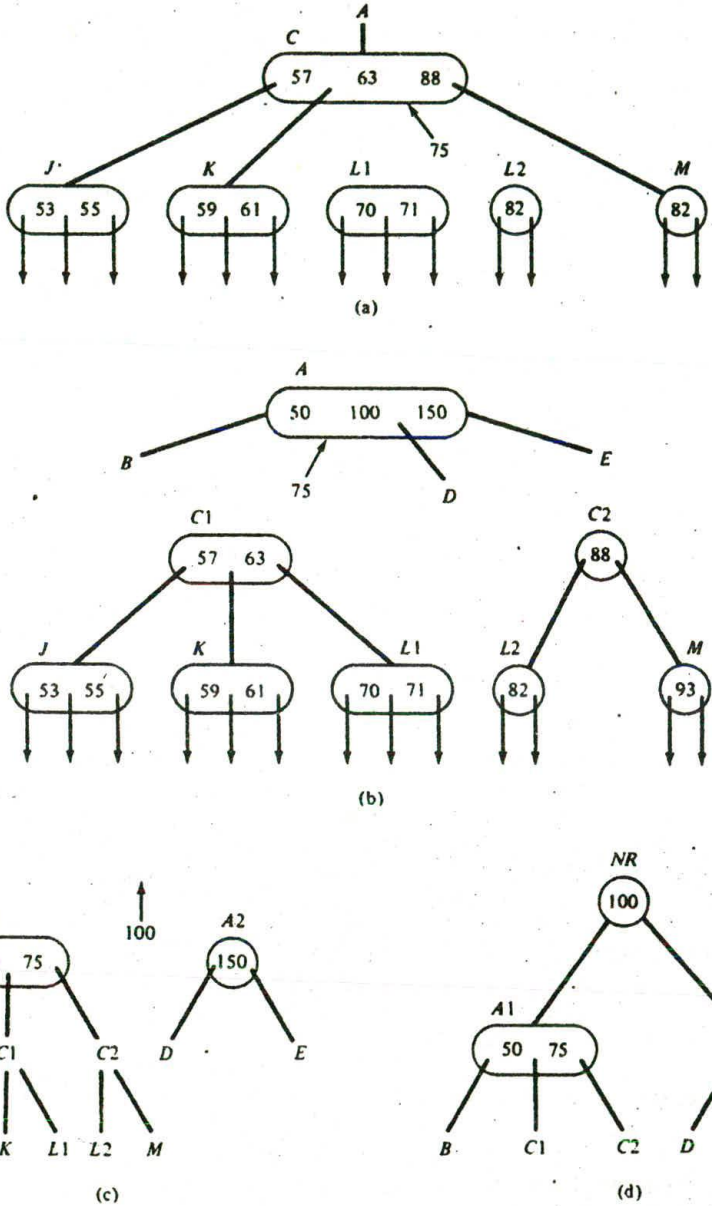
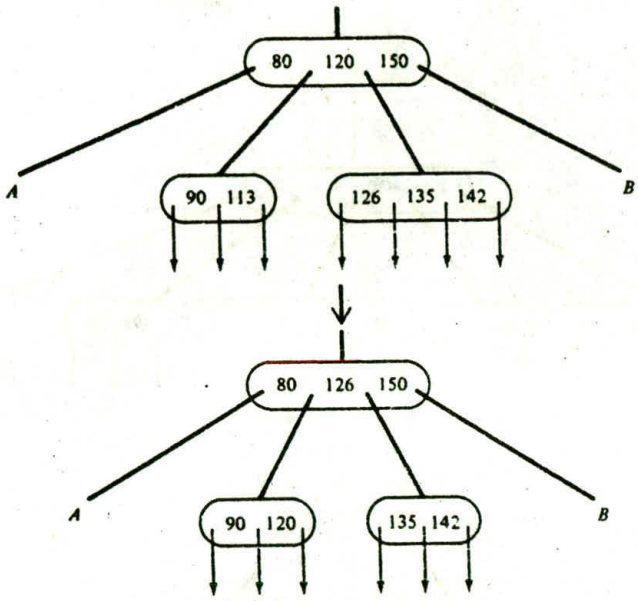
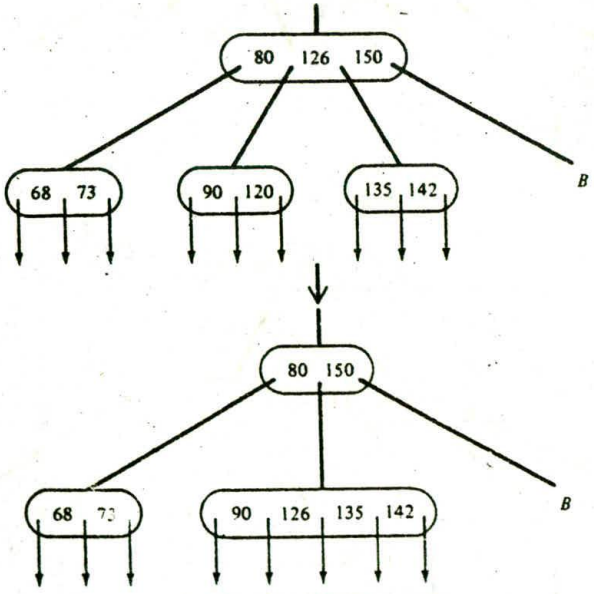

A
C
57 63 88
75
J
53 55
K
59 61
L1
70 71
L2
82
M
82
(a)
A
50 100 150
B
75
D
E
C1
57 63
C2
88
J
53 55
K
59 61
L1
70 71
L2
82
M
93
(b)
A1
50 75
100
A2
150
B
C1
C2
D
E
J
K
L1
L2
M
(c)
NR
100
A1
50 75
A2
150
B
C1
C2
D
E
(d)

Figure 7.3.6

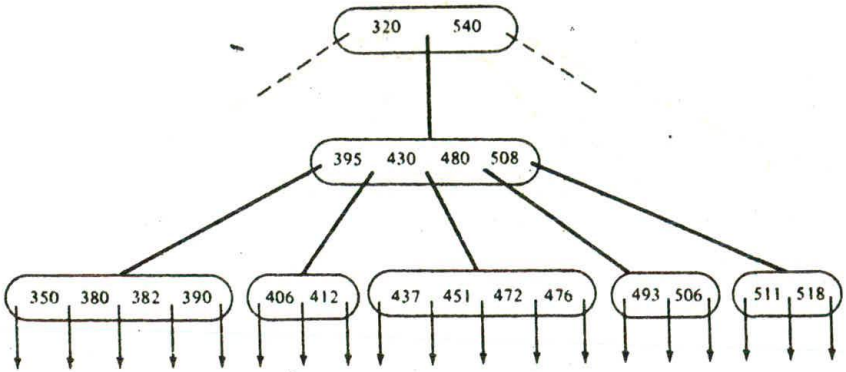


(a) Deleting key 113

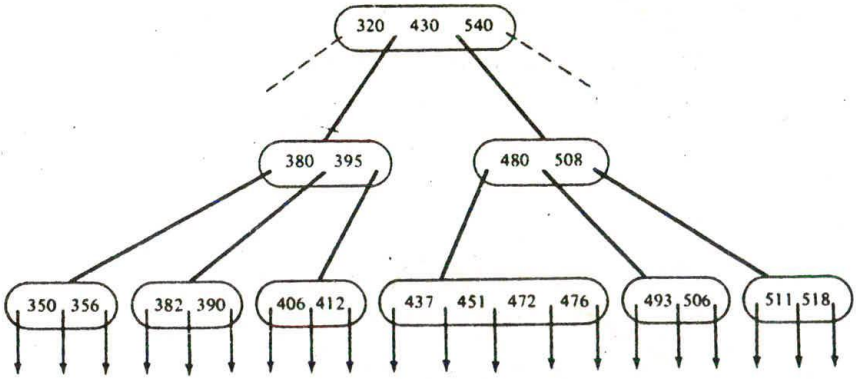


(b) Deleting key 120 and consolidating

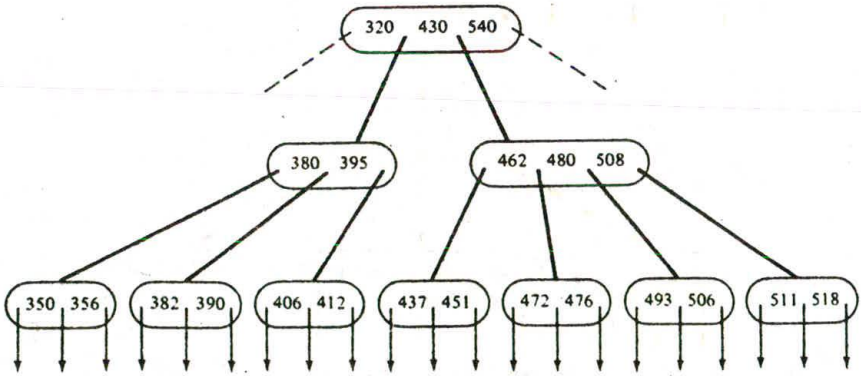
Figure 7.3.7



(a) After inserting 390, 476, 437, and 350



(b) After inserting 356



(c) After inserting 462

Figure 7.3.8

are complications to such an approach that we discuss shortly.) We also assume that *r(p,i)* is a pointer to the record associated with the key *k(p,i)*. Recall that the routine *find* returns a pointer to the leaf in which the key should be inserted and sets the variable *position* to the position in the leaf at which the key should be inserted. Recall also that *key* and *rec* are the argument key and record to be inserted.

The function *insert(key, rec, s, position)* inserts a record into a B-tree. It is called following a call to *find* if the output parameter *found* of that routine is *false* (that is, the key is not already in the tree), where the parameter *s* has been set to the node pointer returned by *find* (that is, the node where *key* and *rec* should be inserted). The routine uses two additional auxiliary routines, which will be presented shortly. The first routine, *split* accepts five input parameters: *nd*, a pointer to a node to be split; *pos*, the position in *node(nd)* where a key and record are to be inserted; *newkey* and *newrec*, the key and record being inserted (this key and record might be the ones being advanced from a previously split node or might be the new key and record being inserted into the tree); and *newnode*, a pointer to the subtree that contains the keys greater than *newkey* (that is, a pointer to the right half of a node previously split), which must be inserted into the node currently being split. To maintain the proper number of sons within a node, each time that a new key and record are inserted into a node, a new son pointer must be inserted as well. When a new key and record are inserted into a leaf, the son pointer inserted is *null*. Because a key and record are inserted into one of the upper levels only when a node is split at a lower level, the new son pointer to be inserted (*newnode*) will be to the right half of the node that was split at the lower level. The left half remains intact within the previously allocated lower-level node. *split* arranges *newkey* and the keys of *node(nd)* so that the group of *n*/2 smallest keys remain in *node(nd)*, the middle key and record are assigned to the output parameters *midkey* and *midrec*, and the remaining keys are inserted into a new node, *node(nd2)*, where *nd2* is also an output parameter.

The second routine, *insnode*, inserts the key *newkey* and the record *newrec* and the subtree pointed to by *newnode* into *node(nd)* at position *pos* if there is room. Recall that the routine *maketree(key, rec)* creates a new node containing the single key *key* and the record *rec* and all pointers *null*. *maketree* returns a pointer to the newly created node. We present the algorithm for *insert* using the routines *split*, *insnode*, and *maketree*.

```
nd = s;
pos = position;
newnode = null;  /*  pointer to right half of split node  */
newrec = rec;    /*       the record to be inserted       */
newkey = key;    /*         the key to be inserted        */
f = father(nd);
while (f != null && numtree(nd) == n) {
   split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
   newnode = nd2;
   pos = index(nd);
   nd = f;
   f = father(nd);
   newkey = midkey;
   newrec = midrec;
} /* end while */
```

```
if (numtrees(nd) < n) {
   insnode(nd, pos, newkey, newrec, newnode);
   return;
} /* end if */
/* f equals null and numtrees(nd) equals n so that nd is */
/*       a full root; split it and create a new root      */
split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
tree = maketree(midkey, midrec);
son(tree, 0) = nd;
son(tree, 1) = nd2;
```

The heart of this algorithm is the routine *split* that actually splits a node. *split* itself uses an auxiliary routine *copy*(*nd*1, *first*, *last*, *nd*2), which sets a local variable *numkeys* to *last* − *first* + 1 and copies fields *k*(*nd*1, *first*) through *k*(*nd*1, *last*) into *k*(*nd*2,0) through *k*(*nd*2, *numkeys*), fields *r*(*nd*1, *first*) through *r*(*nd*1, *last*) (which contains pointers to the actual records) into *r*(*nd*2,0) through *r*(*nd*2, *numkeys*), and fields *son*(*nd*1, *first*) through *son*(*nd*1, *last* + 1) into *son*(*nd*2,0) through *son*(*nd*2, *numkeys* + 1). *copy* also sets *numtrees*(*nd*2) to *numtrees* + 1. If *last* < *first*, *copy* sets *numtrees*(*nd*2) to 1 but does not change any *k*, *r*, or *son* fields. *split* also uses *getnode* to create a new node and *insnode* to insert a new key in a nonfull node.

The following is an algorithm for *split*. The *n* keys contained in *node*(*nd*) and *newkey* must be distributed so that the smallest *n*/2 remain in *node*(*nd*), the highest (*n* − 1)/2 (which equals *n*/2 if *n* is odd) are placed in a new node, *node*(*nd*2), and the middle key is placed in *midkey*. To avoid recomputing *n*/2 on each occasion, assume that its value has been assigned to the global variable *ndiv*2. The input value *pos* is the position in *node*(*nd*) in which *newkey* would be placed if there were room.

```
/* create a new node for the right half; keep */
/*       the first half in node(nd)           */
nd2 = getnode();
if (pos > ndiv2) {
   /* newkey belongs to node(nd2) */
   copy(nd, ndiv2 + 1, n - 2, nd2);
   insnode(nd2, pos - ndiv2 - 1, newkey, newrec, newnode);
   numtrees(nd) = ndiv2 + 1;
   midkey = k(nd, ndiv2);
   midrec = r(nd, ndiv2);
   return;
} /* end if */
if (pos == ndiv2) {
   /* newkey is the middle key */
   copy(nd, ndiv2, n - 2, nd2);
   numtrees(nd) = ndiv2 + 1;
   son(nd2, 0) = newnode;
   midkey = newkey;
   midrec = newrec;
   return;
} /* end if */
```

```
if (pos < ndiv
   /* newkey belongs in node(nd) */
   copy(nd, ndiv2, n - 2, nd2);
   numtrees(nd) = ndiv2;
   insnode(nd, pos, newkey, newrec, newnode);
   midkey = k(nd, ndiv2 - 1);
   midrec = r(nd, ndiv2 - 1);
   return;
} /* end if */
```

The routine *insnode*(*nd*, *pos*, *newkey*, *newrec*, *newnode*) inserts a new record *newrec* with key *newkey* into position *pos* of a nonfull node, *node*(*nd*). *newnode* points to a subtree to be inserted to the right of the new record. The remaining keys and subtrees in positions *pos* or greater are moved up one position. The value of *numtrees*(*nd*) is increased by 1. An algorithm for *insnode* follows:

```
for (i = numtrees(nd) - 1; i >= pos + 1; i--) {
   son(nd, i + 1) = son(nd, i);
   k(nd, i) = k(nd, i - 1);
   r(nd, i) = r(nd, i - 1);
} /* end for */
son(nd, pos + 1) = newnode;
k(nd, pos) = newkey;
r(nd, pos) = newrec;
numtrees(nd) += 1;
```

Computing *father* and *index*

Before examining the efficiency of the insertion procedure, we must clear up one outstanding issue: the matter of the *father* and *index* functions. You may have noticed that, although these functions are utilized in the *insert* procedure and we suggested that they could be implemented most directly by adding *father* and *index* fields to each node, those fields are not updated by the insertion algorithm. Let us examine how this update could be achieved and why we chose to omit that operation. We then examine alternative methods of implementing the two functions that do not require the update.

father and *index* fields would have to be updated each time that *copy* or *insnode* were called. In the case of *copy*, both fields in each son whose pointer is copied must be modified. In the case of *insnode*, the *index* field of each son whose pointer is moved must be modified, as well as both fields in the son being inserted. (In addition, the fields must be updated in the two halves of a root node being split in the *insert* routine.) However, this would impact the efficiency of the insertion algorithm in an unacceptable manner, especially when dealing with nodes in external storage. In the entire B-tree search and insertion process (excluding the update of *father* and *index* fields), at most two nodes at each level of the tr are accessed. In most cases, when a split does not occur on a level, only one node at the level is accessed. The *copy* and *insnode* operations, although they move nodes from one subtree to another, do so by moving pointers within one or two father nodes and thus do not actually require access to the son nodes being moved. Requiring an update to *father* and *index* fields in those sons would require accessing

and modifying all the son nodes themselves. But reading and writing a node from and to external storage are the most expensive operations in the entire B-tree management process. When one considers that, in a practical information storage system, a node can have several hundred sons, it becomes apparent that maintaining *father* and *index* fields could result in a hundredfold decrease in system efficiency.

How, then, can we obtain the *father* and *index* data required for the insertion process without maintaining separate fields? First, recall that the function *nodesearch*(*p*, *key*) returns the position of the smallest key in *node*(*p*) greater than or equal to *key*, so that *index*(*nd*) equals *nodesearch*(*father*(*nd*), *key*). Therefore once *father* is available, *index* can be obtained without a separate field.

To understand how we can obtain *father*, let us look at a related problem. No B-tree insertion can take place without a prior search to locate the leaf where the new key must be inserted. This search proceeds from the root and accesses one node at each level until it reaches the appropriate leaf. That is, it proceeds along a single path from the root to a leaf. The insertion then backs up along that same path, splitting all full nodes in the path from the leaf toward the root, until it reaches a nonfull node into which it can insert a key without splitting. Once that insertion is performed, the insertion process terminates.

The insertion process accesses the same nodes as the search process. Since we have seen that accessing a node from external storage is quite expensive, it would make sense for the search process to store the nodes on its path together with their external addresses in internal memory, where the insertion process can access them without an expensive second read operation. But once all the nodes in a path are stored in internal memory, a node's father can be located by simply examining the previous node in the path. Thus there is no need to maintain and update a *father* field.

Let us then present modified versions of *find* and *insert* to locate and insert a key in a B-tree. Let *pathnode*(*i*) be a copy of the *i*th node in the path from the root to a leaf, let *location*(*i*) be its location (either a pointer if the tree is in internal memory, or an external storage address if it is in external storage), and let *index*(*i*) be the position of the node among the sons of its father (note that *index* can be determined during the search process and retained for use during insertion). We refer to *son*(*i*,*j*), *k*(*i*,*j*) and *r*(*i*,*j*) as the *j*th *son*, *key*, and *record* field, respectively, in *pathnode*(*i*). Similarly, *numtrees*(*i*) is the *numtrees* field in *pathnode*(*i*).

The following *find* algorithm utilizes the operation *access*(*i*, *loc*) to copy a node from location *loc* (either from internal or external memory) into *pathnode*(*i*) and *loc* itself into *location*(*i*). If the tree is stored internally, this operation consists of

```
pathnode(i) = node(loc);
location(i) = loc;
```

If the tree is stored externally, the operation consists of

```
directread(loc, pathnode(i));
location(i) = loc;
```

where *directread* reads a block of storage at a particular external address (*loc*) into an internal memory buffer (*pathnode*(*i*)). We also assume that *nodesearch*(*i*, *key*) searches *pathnode*(*i*) rather than *node*(*i*). We present the algorithm *find*:

```
q = null;
p = tree;
j = -1;
i = -1;
while (p != null) {
   index(++j) = i;
   access(j, p);
   i = nodesearch(j, key);
   q = p
   if (i < numtrees(j) - 1  && key == k(j, i))
      break;
   p = son(j, i);
} /* end while */
position = i;
return(j);  /*  key is in pathnode(j) or belongs there  */
```

The insertion process is modified in several places. First, *insnode* and *copy* access *pathnode*(*nd*) rather than *node*(*nd*). That is, *nd* is now an array index rather than a pointer, so that all references to *k*, *son*, *p*, and *numtrees* are to fields within an element of *pathnode*. The algorithms for *insnode* and *copy* do not have to be changed.

Second, *split* must be modified to write out the two halves of a split node. It assumes a routine *replace*(*i*), which replaces the node at *location*(*i*) with the contents of *pathnode*(*i*). This routine is the reverse of *access*. If the tree is stored internally, it may be implemented by

```
node(location(i)) = pathnode(i);
```

and if externally, by

```
directwrite(location(i), pathnode(i));
```

where *directwrite* writes a buffer in memory (*pathnode*(*i*)) into a block of external storage at a particular external address (*location*(*i*)). *split* also uses a function *makenode*(*i*) that obtains a new block of storage at location *x*, places *pathnode*(*i*) in that block, and returns *x*. The following is a revised version of *split*:

```
if (pos > ndiv2) {
   copy(nd, ndiv2 + 1, n - 2, nd + 1);
   insnode(nd + 1, pos - ndiv2 - 1, newkey, newrec, newnode);
   numtrees(nd) = ndiv2 + 1;
   midkey = k(nd, ndiv2);
   midrec = r(nd, ndiv2);
   return;
} /* end if */
```

```
if (pos == ndiv2) {
   copy(nd, ndiv2, n - 2, nd + 1);
   numtrees(nd) = ndiv2;
   son(nd + 1, 0) = newnode;
   midkey = newkey;
   midrec = newrec;
   return;
} /* end if */
if (pos < ndiv2) {
   copy(nd, ndiv2, n - 2, nd + 1);
   numtrees(nd) = ndiv2;
   insnode(nd, pos, newkey, newrec, newnode);
   midkey = newkey;
   midrec = newrec;
} /* end if */
replace(nd);
nd2 = makenode(nd + 1);
```

Notice that *nd* is now a position in *pathnode* rather than a node pointer, and that *pathnode*(*nd* + 1) rather than *node*(*nd*2) is used to build the second half of the split node. This can be done since the node at level *nd* + 1 (if any) of the path has already been updated by the time *split* is called on *nd*, so that *pathnode*(*nd* + 1) can be reused. *nd*2 remains the actual location of the new node (allocated by *makenode*). (We should note that it may be desirable to retain a path to the newly inserted key in *pathnode* if, for example, we wish to perform a sequential traversal or sequential insertions beginning at that point. In that case the algorithm must be suitably adjusted to place the appropriate left or right half of the split node at the appropriate position in *pathnode*. We also could then not use *pathnode*(*i* + 1) to build the right half but must use an additional auxiliary internal memory node instead. We leave the details to the reader.)

The routine *insert* itself is also modified in that it uses *nd* − 1 rather than *father*(*nd*). It also calls upon *replace* and *makenode*. When the root must be split, *maketree* builds a new tree root node in internal storage. This node is placed in *pathnode*(*i*) (which is no longer needed, since the old root node has been updated by *split*) and written out using *makenode*. The following is the revised algorithm for *insert*.

```
nd = s;
pos = position;
newnode = null;
newrec = rec;
newkey = key;
while (nd != 0 && numtrees(nd) == n) {
   split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
   newnode = nd2;
   pos = index(nd);
   nd--;
   newkey = midkey;
   newrec = midrec;
} /* end while */
```

```
if (numtrees(nd) < n) {
   insnode(nd, pos, newkey, newrec, newnode);
   replace(nd);
   return;
} /* end if */
split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
pathnode(0) = maketree(midkey, midrec);
son(0, 0) = nd;
son(0, 1) = nd2;
tree = makenode(0);
```

Deletion in Multiway Search Trees

The simplest method for deleting a record from a multiway search tree is to retain the key in the tree but mark it in some way as representing a deleted record. This could be accomplished by setting the pointer to the record corresponding to the key to *null* or by allocating an extra flag field for each key to indicate whether or not it has been deleted. The space occupied by the record itself can, of course, be reclaimed. In this way the key remains in the tree as a guidepost to the subtrees but does not represent a record within the file.

The disadvantage of this approach is that the space occupied by the key itself is wasted, possibly leading to unnecessary nodes in the tree when a large number of records have been deleted. Extra "deleted" bits require still more space.

Of course, if a record with a deleted key is subsequently inserted, the space for the key can be recycled. In a nonleaf node, only the same key would be able to reuse the space, since it is too difficult to determine dynamically that the newly inserted key is between the predecessor and successor of the deleted key. However, in a leaf node (or, in certain situations, in a semileaf), the deleted key's space can be reused by a neighboring key, since it is relatively easy to determine proximity. Since a large portion of keys are in leaves or semileaves, if insertions and deletions occur with equal frequency (or if there are more insertions than deletions) and are uniformly distributed (that is, the deletions are not bunched together to significantly reduce the total number of keys in the tree temporarily), the space penalty is tolerable in exchange for the advantage of ease of deletion. There is also a small time penalty in subsequent searches, since some keys will require more nodes to be examined than if the deleted key had never been inserted in the first place.

If we are unwilling to pay the space/search-time penalty of simplified deletion, there are more expensive deletion techniques that eliminate the penalty. In an unrestricted multiway search tree, a technique similar to deletion from a binary search tree can be employed:

1. If the key to be deleted has an empty left or right subtree, simply delete the key and compact the node. If it was the only key in the node, free the node.
2. If the key to be deleted has nonempty left and right subtrees, find its successor key (which must have an empty left subtree); let the successor key take its place and compact the node that had contained that successor. If the successor was the only key in the node, free the node.

We leave the development of a detailed algorithm and program to the reader.

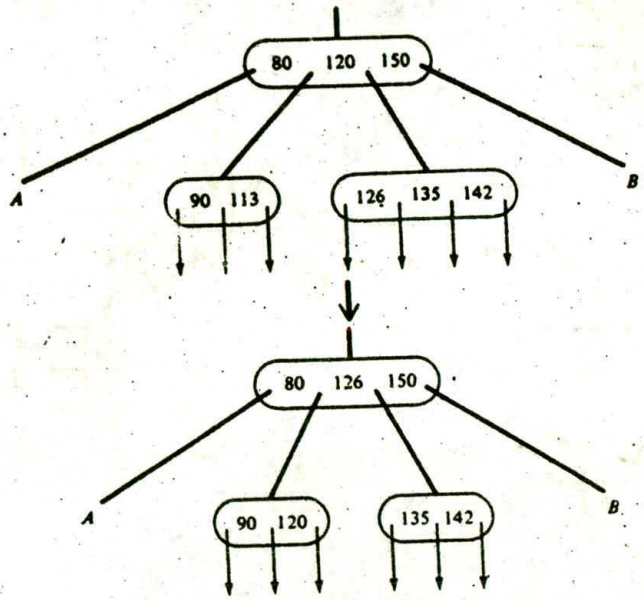
However, this procedure may result in a tree that does not satisfy the requirements for either a top-down tree or a B-tree, even if the initial tree did satisfy those requirements. In a top-down tree, if the key being deleted is from a semileaf that is not a leaf and the key has empty right and left subtrees, the semileaf will be left with fewer than $n - 1$ keys, even though it is not a leaf. This violates the top-down requirement. In that case it is necessary to choose a random nonempty subtree of the node and move the largest or smallest key from that subtree into the semileaf from which the key was deleted. This process must be repeated until the semileaf from which a key is taken is a leaf. This leaf can then be compacted or freed. In the worst case, this might require rewriting one node at each level of the tree.

In a strict B-tree, we must preserve the requirement that each node contains at least $(n - 1)/2$ keys. As noted previously, if a key is being deleted from a nonleaf node, its successor (which must be in a leaf) is moved to the deleted position and the deletion proceeds as if the successor were deleted from the leaf node. When a key (either the key to be deleted or its successor) is removed from a leaf node and the number of keys in the node drops below $(n - 1)/2$, remedial action must be taken. This situation is called an ***underflow***. When an underflow occurs, the simplest solution is to examine the leaf's younger or older brother. If the brother contains more than $(n - 1)/2$ keys, the key *ks* in the father node that separates between the two brothers can be added to the underflow node and the last or first key of the brother (last if the brother is older; first if younger) added to the father in place of *ks*. Figure 7.3.9a illustrates this process on an order-5 B-tree. (We should note that once a brother is being accessed, we could distribute the keys evenly between the two brothers rather than simply shifting one key. For example, if the underflow node $n1$ contains 106 and 112, the separating key in the father f is 120, and the brother $n2$ contains 123, 128, 134, 139, 142, and 146 in an order-7 B-tree, we can rearrange them so that $n1$ contains 106, 112, 120, and 123. 128 moves up to f as the separator, and 134, 139, 142, and 146 remain in $n2$.)

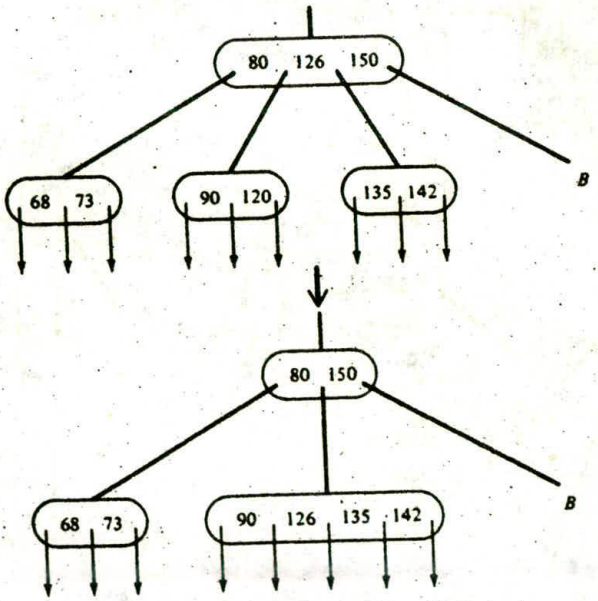
If both brothers contain exactly $(n - 1)/2$ keys, no keys can be shifted. In that case the underflow node and one of its brothers are ***concatenated***, or ***consolidated***, into a single node that also contains the separator key from their father. This is illustrated in Figure 7.3.9b, where we combine the underflow node with its younger brother.

Of course, it is possible that the father contains only $(n - 1)/2$ keys, so that it also has no extra key to spare. In that case it can borrow from its father and brother as in Figure 7.3.10a. In the worst case, when the father's brothers also have no spare keys, the father and its brother may also be consolidated and a key taken from the grandfather. This is illustrated in Figure 7.3.10b. Potentially, if all nonroot ancestors of a node and their brothers contain exactly $(n - 1)/2$ keys, a key will be taken from the root, as consolidations take place at each level from the leaves to the level just below the root. If the root had more than one key, this ends the process, since the root of a B-tree need have only one key. If, however, the root contained only a single key, that key is used in the consolidation of the two nodes below the root, the root is freed, the consolidated node becomes the new root of the tree, and the depth of the B-tree is reduced. We leave to the reader the development of an actual algorithm for B-tree deletion from this description.

You should note, however, that it is foolish to form a consolidated node with $n - 1$ keys if a subsequent insertion will immediately split the node in two. In a B-tree of large order, it may make sense to leave an underflow node with fewer than $(n - 1)/2$ keys

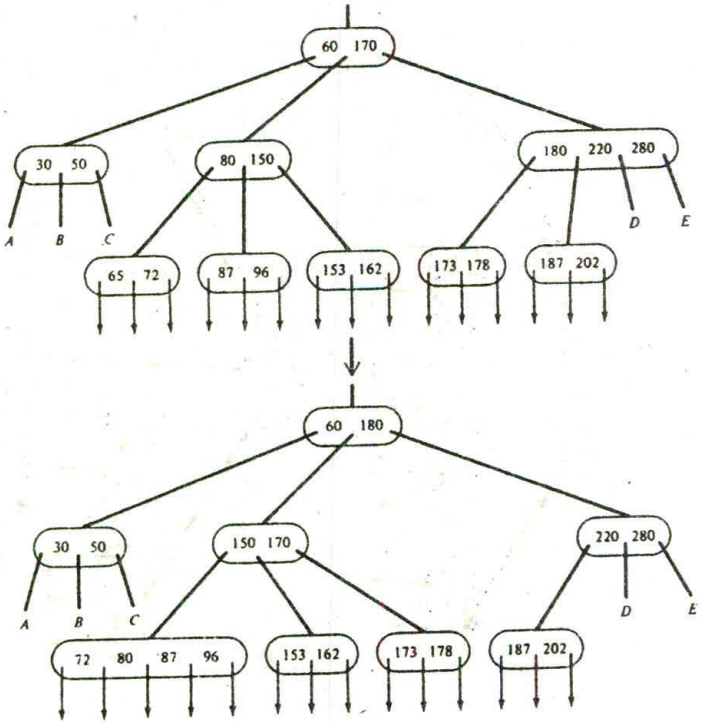


(a) Deleting key 113

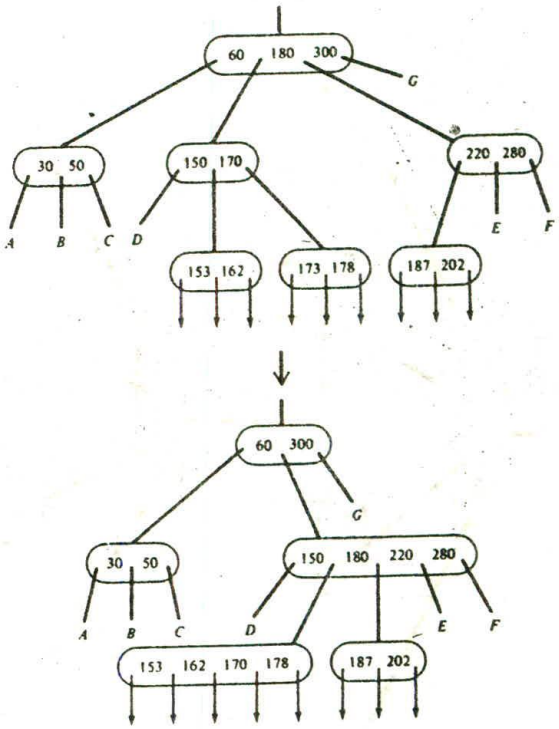


(b) Deleting key 120 and consolidating

Figure 7.3.9



(a) Deleting 65, consolidating and borrowing



(b) Deleting 173 and a double consolidation

Figure 7.3.10

(even though this violates the formal B-tree requirements) so that future insertions can take place without splitting. Typically, a minimum number (*min* less than $(n - 1)/2$) of keys is defined so that consolidation takes place only if fewer than *min* keys remain in a leaf node.

Efficiency of Multiway Search Trees

The primary considerations in evaluating the efficiency of multiway search trees, as for all data structures, are time and space. Time is measured by the number of nodes accessed or modified in an operation, rather than by the number of key comparisons. The reason for this, as mentioned earlier, is that accessing a node usually involves reading from external storage and modifying a node involves writing to external storage. These operations are far more time-consuming than internal memory operations and therefore dominate the time required.

Similarly, space is measured by the number of nodes in the tree and the size of the nodes rather than by the number of keys actually contained in the nodes, since the same space is allocated for a node regardless of the number of keys it actually contains. Of course, if the records themselves are stored outside the tree nodes, the space requirement for the records is determined by how the record storage is organized rather than by how the tree itself is organized. The storage requirements for the records generally overwhelm the requirements for the key tree, so that the actual tree space may not be significant.

First, let us examine top-down multiway search trees. Assuming an order-m tree and n records, there are two extreme possibilities. In the worst case for search time, the tree is totally unbalanced. Every node except one is a semileaf with one son and contains $m - 1$ keys. The single leaf node contains $((n - 1) \% (m - 1)) + 1$ keys. The tree contains $((n - 1)/(m - 1)) + 1$ nodes, one on each level. A search or an insertion accesses half that many nodes on the average and every node in the worst case. An insertion also requires writing one or two nodes (one if the key is inserted in the leaf, two if a new leaf must be created). A deletion always accesses every node and can modify as few as one node, but can potentially modify every node (unless a key can be simply marked as deleted).

In the best case for search time, the tree is almost balanced, each node except one contains $m - 1$ keys, and each nonleaf except one has m sons. There are still $((n - 1)/(m - 1)) + 1$ nodes, but there are fewer than $\log_m (n - 1) + 1$ levels. Thus the number of nodes accessed in a search, insertion, or deletion is less than this number. (In such a tree, more than half the keys are in a semileaf or leaf, so that the average search time is not much better than the maximum.) Fortunately, as is the case for binary trees, fairly balanced trees occur far more frequently than unbalanced trees, so that the average search time using multiway search trees is $O(\log n)$.

However, a general multiway tree and even a top-down multiway tree use an inordinate amount of storage. To see why this is so, see Figure 7.3.11, which illustrates a typical top-down multiway search tree of order 11 with 100 keys. The tree is fairly balanced and average search cost is approximately 2.19 [10 keys at level 0 require accessing one node, 61 at level 1 require accessing two nodes, and 29 at level 2 require accessing three nodes: $(10*1 + 61*2 + 29*3)/100 = 2.19$], which is reasonable.

981
969
958
950
946
940
935
921
887
877
859
841
837
821
811
768
757
749
736
733
730
721
709
705
692
691
369
356
326
322
320
475
450
423
415
413
440
433
412
407
389
386
373
306
294
261
293
290
289
286
241
401
384
121
107
102
88
82
118
111
97
91
81
63
47
46
29
5
7
937
763
687
656
652
639
611
607
593
591
585
581
641
568
562
537
533
511
482
481
239
235
229
226
218
197
185
147
146
142
128

Figure 7.3.11

However, to accommodate the 100 keys, the tree uses 23 nodes or 4.35 keys per node, representing a space utilization of only 43.5 percent. The reason for this is that many leaves contain only one or two keys, and the vast majority of the nodes are leaves. As the order and the number of keys increase, the utilization becomes worse, so that an order-11 tree with thousands of keys can expect 27 percent utilization, and an order-21 tree can expect only 17 percent utilization. As the order grows even larger, the utilization drops toward 0. Since high orders are required to produce reasonable search costs for large numbers of keys, top-down multiway trees are an unreasonable alternative for data storage.

Every B-tree is balanced and each node contains at least $(m - 1)/2$ keys. Figure 7.3.12 illustrates the minimum and maximum number of nodes and keys at levels 0, 1, 2, and an arbitrary level i, as well as the minimum and maximum number of total nodes and keys in a B-tree of order m and maximum level d. In that figure, q equals $(m - 1)/2$. Note that the maximum level is 1 less than the number of levels (since the root is at level 0), so that $d + 1$ equals the maximum number of node accesses needed to find an element. From the minimum total number of keys in Figure 7.3.12, we can deduce that the maximum number of node accesses for one of n keys in an order-m B-tree is $1 + \log_{q+1}(n/2)$. Thus, unlike top-down multiway trees, the maximum number of node accesses grows only logarithmically as the number of keys. Nevertheless, as we pointed out earlier, average search time is competitive between top-down multiway trees and B-trees, since top-down trees are usually fairly balanced.

Insertion into a B-tree requires reading one node per level and writing one node at minimum plus two nodes for every split that occurs. If s splits occur, $2s + 1$ nodes are written (two halves of each split plus the father of the last node split). Deletion requires reading one node per level to find a leaf, writing one node if the deleted key is in a leaf and the deletion does not cause an underflow, and writing two nodes if the deleted key is in a nonleaf and removing the replacement key from a leaf does not cause

	Minimum		Maximum	
Level	Nodes	Keys	Nodes	Keys
0	1	1	1	$m - 1$
1	2	$2q$	m	$(m - 1)m$
2	$2(q + 1)$	$2q(q + 1)$	m^2	$(m - 1)m^2$
i	$2(q + 1)^{i-1}$	$2q(q + 1)^{i-1}$	m^i	$(m - 1)m^i$
Total	$1 + \frac{2(q + 1)^d - 1)}{q}$	$2(q + 1)^d$	$\frac{m^{d+1} - 1}{m - 1}$	$m^{d+1} - 1$

Figure 7.3.12

that leaf to underflow. If an underflow does occur, one additional read (of the brother of each underflowed node) per underflow, one additional write for every consolidation except the last, and three additional writes for the final underflow if no consolidation is necessary (the underflow node, its brother, and its father) or two additional writes if a consolidation is necessary (the consolidated node and its father) are required. All these operations are $O(\log n)$.

As is the case with a heap (Section 6.3) and a balanced binary tree (Section 7.2), insertion and deletion of the minimum or maximum element are both $O(\log n)$ in a B-tree; therefore the structure can be used to implement an (ascending or descending) priority queue efficiently. In fact, a 3-2 tree (a B-tree of order 3) is probably the most efficient practical method for implementing a priority queue in internal memory.

Since each node in a B-tree (except the root) must be at least approximately half full, the worst case storage utilization approaches 50 percent. In practice, average storage utilization in a B-tree approaches 69 percent. For example, Figure 7.3.13 illustrates a B-tree of order 11 with the same 100 keys as the multiway tree of Figure 7.3.11. Average search time is 2.88 (1 key requiring 1 node access, 10 keys requiring 2 accesses, and 89 keys requiring 2 accesses), which is greater than the corresponding multiway tree. Indeed, a fairly balanced top-down multiway tree will have lower search cost than a B-tree, since all its upper nodes are always completely full. However, the B-tree contains only 15 nodes, yielding a storage utilization of 66.7 percent, far higher than the 43.5 percent utilization of the multiway tree.

Improving the B-Tree

There are a number of ways of improving the storage utilization of a B-tree. One method is to delay splitting a node when it overflows. Instead, the keys in the node and one of its adjacent brothers, as well as the key in the father that separates between the two nodes, are redistributed evenly. This is illustrated in Figure 7.3.14 on a B-tree of order 7. When both a node and its brother are full, the two nodes are split into 3. This guarantees a minimum storage utilization of almost 67 percent, and the storage utilization is higher in actual practice. Such a tree is called a B*-tree. Indeed, this technique can be extended even further by redistributing keys among all the brothers and the father of a full node. Unfortunately, this method exacts its own price, since it requires expensive additional accesses upon overflow insertions, while the marginal additional space utilization achieved by considering each extra brother becomes smaller and smaller.

Another technique is to use a ***compact B-tree***. Such a B-tree has maximum storage utilization for a given order and number of keys. It can be shown that this maximum storage utilization for a B-tree of a given order and a given number of keys is attained when nodes toward the bottom of the tree contain as many keys as possible. Figure 7.3.15 illustrates a compact B-tree for the 100 keys of Figures 7.3.11 and 7.3.13. It can be shown that the average search cost for a compact B-tree is never more than 1 more than the minimum average search cost among all B-trees with the given order and number of keys. Thus, although a compact B-tree achieves a maximum storage utilization, it also achieves reasonable search cost. For example, the search cost for the tree of Figure 7.3.14 is only 1.91 (9 keys at level 0, requiring one access, and 91 keys at level 1, requiring two accesses: $9*1+91*2 = 191/100 = 1.91$), which is very close

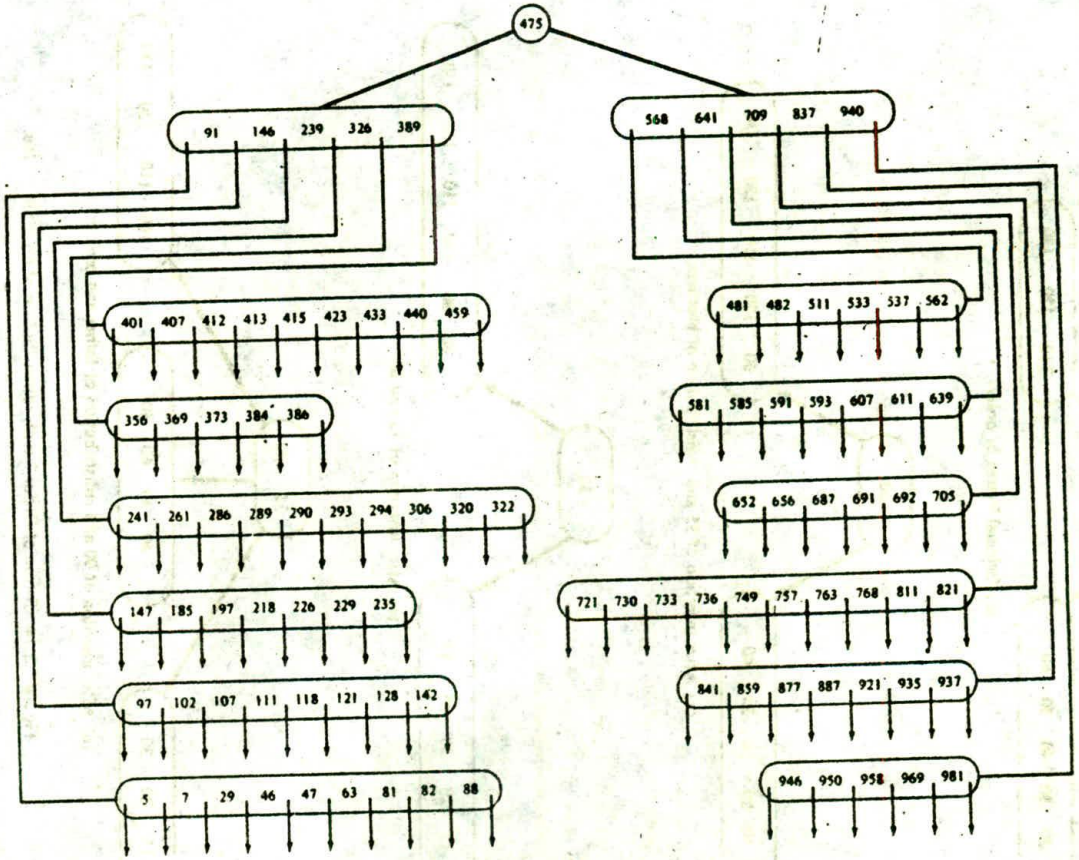
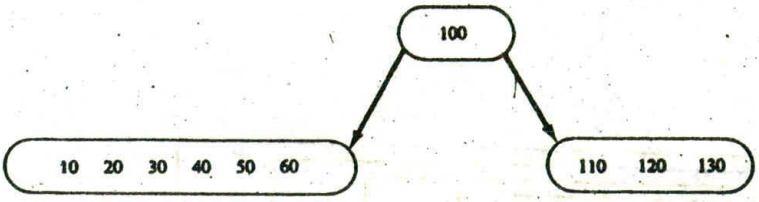
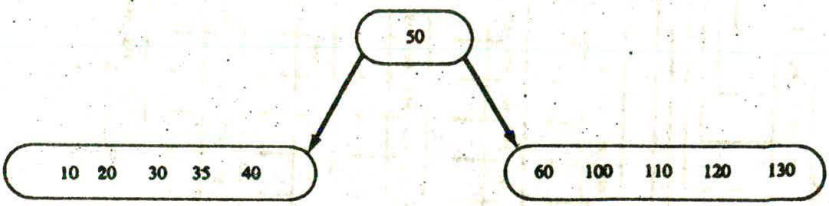

475
91 146 239 326 389
568 641 709 837 940
401 407 412 413 415 423 433 440 459
481 482 511 533 537 562
356 369 373 384 386
581 585 591 593 607 611 639
241 261 286 289 290 293 294 306 320 322
652 656 687 691 692 705
147 185 197 218 226 229 235
721 730 733 736 749 757 763 768 811 821
97 102 107 111 118 121 128 142
841 859 877 887 921 935 937
5 7 29 46 47 63 81 82 88
946 950 958 969 981

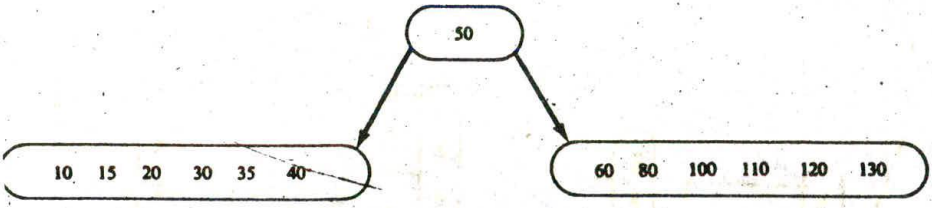
Figure 7.3.13



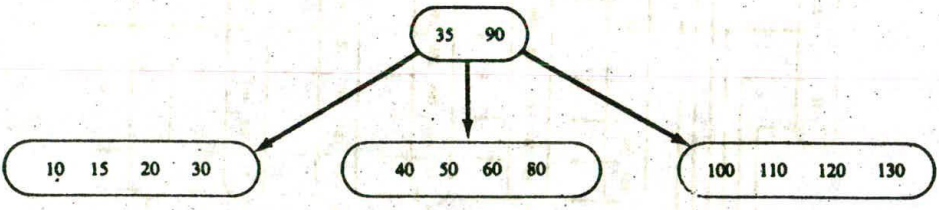
(a) Original B-tree of order7



(b) After insertion of 35 and redistribution of brothers.



(c) After insertion of 15 and 80.



(d) After insertion of 90 and redistribution of father and brothers.

Figure 7.3.14 B-tree of Order 7 with redistribution of multiple nodes.

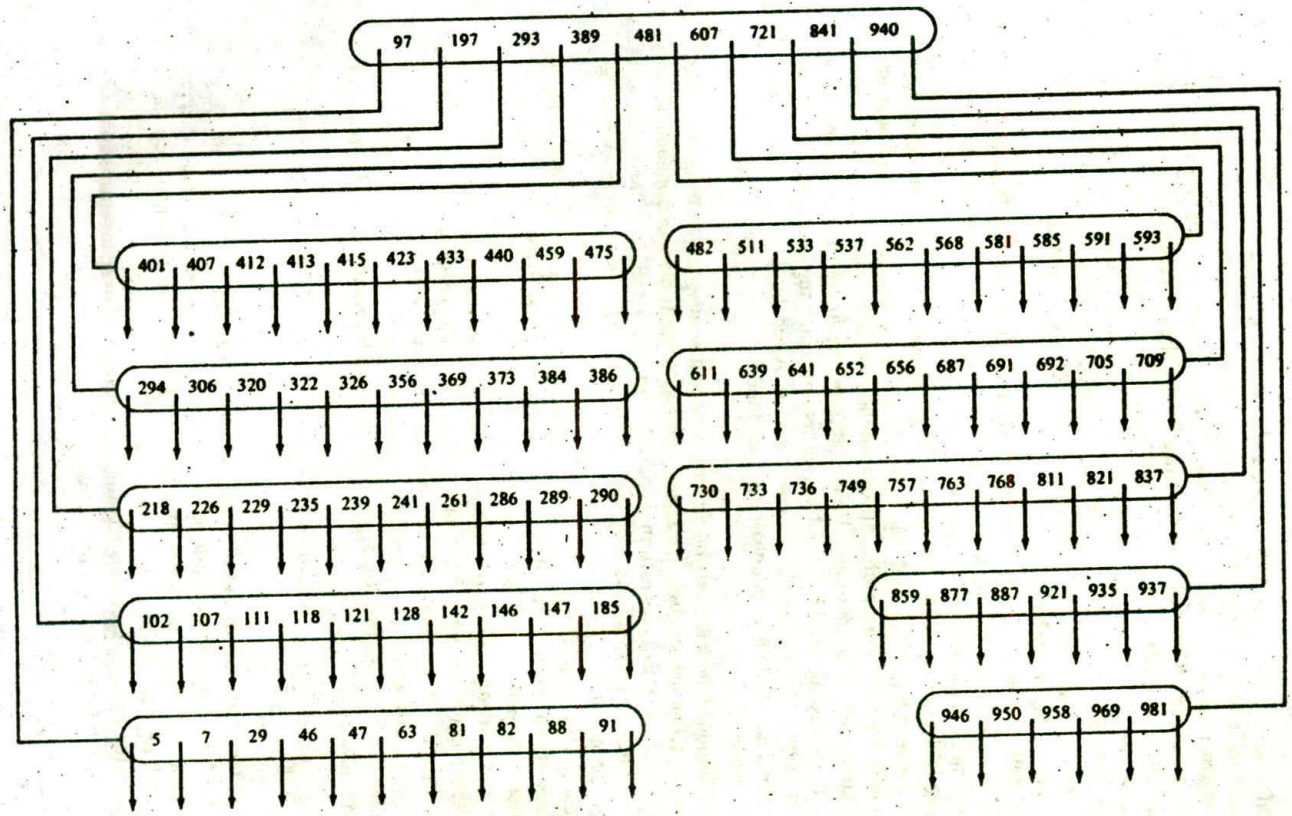

97 197 293 389 481 607 721 841 940
401 407 412 413 415 423 433 440 459 475
482 511 533 537 562 568 581 585 591 593
294 306 320 322 326 356 369 373 384 386
611 639 641 652 656 687 691 692 705 709
218 226 229 235 239 241 261 286 289 290
730 733 736 749 757 763 768 811 821 837
102 107 111 118 121 128 142 146 147 185
859 877 887 921 935 937
5 7 29 46 47 63 81 82 88 91
946 950 958 969 981

Figure 7.3.15

to optimal. Yet the tree uses only 11 nodes, for a storage utilization of 90.9 percent. With more keys, storage utilization in compact B-trees reaches 98 percent or even 99 percent.

Unfortunately, there is no known efficient algorithm to insert a key into a compact B-tree and maintain compactness. Instead, insertion proceeds as in an ordinary B-tree and compactness is not retained. Periodically (for example, at night when the file is not used), a compact B-tree can be constructed from the noncompact tree. However, a compact B-tree degrades so rapidly with insertions that, for high orders, storage utilization drops below that of a random B-tree after fewer than 2 percent additional keys have been inserted. Also, the number of splits required for an insertion is higher on the average than for a random B-tree. Thus, a compact B-tree should only be used when the set of keys is highly stable.

A widely used technique for reducing both space and time requirements in a B-tree is to use various compression techniques on the keys. Since all keys in a given node are fairly similar to each other, the initial bits of those keys are likely to be the same. Thus, those initial bits can be stored once for the entire node (or they can be determined as part of the search process from the root of the tree by noticing that if two adjacent keys in a node have the same prefix, all keys in the subtree between the two keys also have the same prefix). In addition, all bits preceding the first bit that distinguishes a key from its preceding neighbor need not be retained (although an indication of its position must be). This is called ***front compression***. A second technique, called ***rear compression***, maintains only enough of the rear of the key to distinguish between a key and its successor.

For example, if three keys are *anchor*, *andrew*, and *antoin*, *andrew* can be encoded as 2*d*, indicating that the first two characters are the same as its predecessor and that the next character, *d*, distinguishes it from its predecessor and successor. If the successors of *andrew* within the node were *andule*, *antoin*, *append*, and *apples*, *andrew* would be encoded as 2*d*, *andule* as 3*u*, *antoin* as simply 2, and *append* as 1*ppe*.

If rear compression is used, it is necessary to access the record itself to determine if a key is present in a file, since the entire key cannot be reconstructed from the encoding. Also, under both methods, the key code that is retained is of variable length, so that the maximum number of keys in a node is no longer fixed. Another disadvantage of variable-length key encoding is that binary search can no longer be used to locate a key in a node. In addition, the key code for some existing keys may have to be changed when a new key is inserted. The advantage of compression is that it enables more keys to be retained in a node, so that the depth of the tree and the number of nodes required can be reduced.

B^+-Trees

One of the major drawbacks of the B-tree is the difficulty of traversing the keys sequentially. A variation of the basic B-tree structure, the B^+-tree, retains the rapid random access property of the B-tree, while also allowing rapid sequential access. In the B^+-tree, all keys are maintained in leaves, and keys are replicated in nonleaf nodes to define paths for locating individual records. The leaves are linked together to provide a sequential path for traversing the keys in the tree.

Figure 7.3.16 illustrates a B^+-tree. To locate the record associated with key 53 (random access), the key is first compared with 98 (the first key in the root). Since it is less, proceed to node *B*. Fifty-three is then compared with 36 and then 53 in node *B*. Since it is less than or equal to 53, proceed to node *E*.

Note that the search does not halt when the key is found as is the case in a B-tree. In a B-tree, a pointer to the record corresponding to a key is contained with each key in the tree, whether in a leaf or a nonleaf node. Thus once the key is found, the record can be accessed. In a B^+-tree, pointers to records are only associated with keys in leaf nodes; consequently the search is not complete until the key is located in a leaf. Therefore, when equality is obtained in a nonleaf, the search continues. In node *E* (a leaf), key 53 is located, and from it, the record associated with that key. If we now wish to traverse the keys in the tree sequentially beginning with key 53, we need only follow the pointers in the leaf nodes.

The linked list of leaves is called a ***sequence set***. In actual implementations, the nodes of the sequence set frequently do not contain all the keys in the file. Rather, each sequence set node serves as an index to a large data area where a large number of records are kept. A search involves traversing a path in the B^+-tree, reading a block from the data area associated with the leaf node that is finally accessed, and then searching the block sequentially for the required record.

The B^+-tree may be considered to be a natural extension of the indexed sequential file of Section 7.1. Each level of the tree is an index to the succeeding level, and the lowest level, the sequence set, is an index to the file itself.

Insertion into a B^+-tree proceeds much as in a B-tree except that when a node is split, the middle key is retained in the left half-node as well as being promoted to the father. When a key is deleted from a leaf, it can be retained in the nonleaves, since it is still a valid separator between the keys in the nodes below.

The B^+-tree retains the search and insertion efficiencies of the B-tree but increases the efficiency of finding the next record in the tree from $O(\log n)$ (in a B-tree, where finding the successor involves climbing up or down the tree) to $O(1)$ (in a B^+-tree, where it involves accessing one additional leaf at most). An additional advantage of the B^+-tree is that no record pointers need be kept in the nonleaf nodes, which increases the potential order of the tree.

Digital Search Trees

Another method of using trees to expedite searching is to form a general tree based on the symbols of which the keys are composed. For example, if the keys are integers, each digit position determines one of ten possible sons of a given node. A forest representing one such set of keys is illustrated in Figure 7.3.17. If the keys consist of alphabetic characters, each letter of the alphabet determines a branch in the tree. Note that every leaf node contains the special symbol *eok*, which represents the end of a key. Such a leaf node must also contain a pointer to the record that is being stored.

If a forest is represented by a binary tree, as in Section 5.5, each node of the binary tree contains three fields: *symbol*, which contains a symbol of the key; *son*, which is a pointer to the node's oldest son in the original tree; and *brother*, which is a pointer to the node's next younger brother in the original tree. The first tree in the forest is pointed

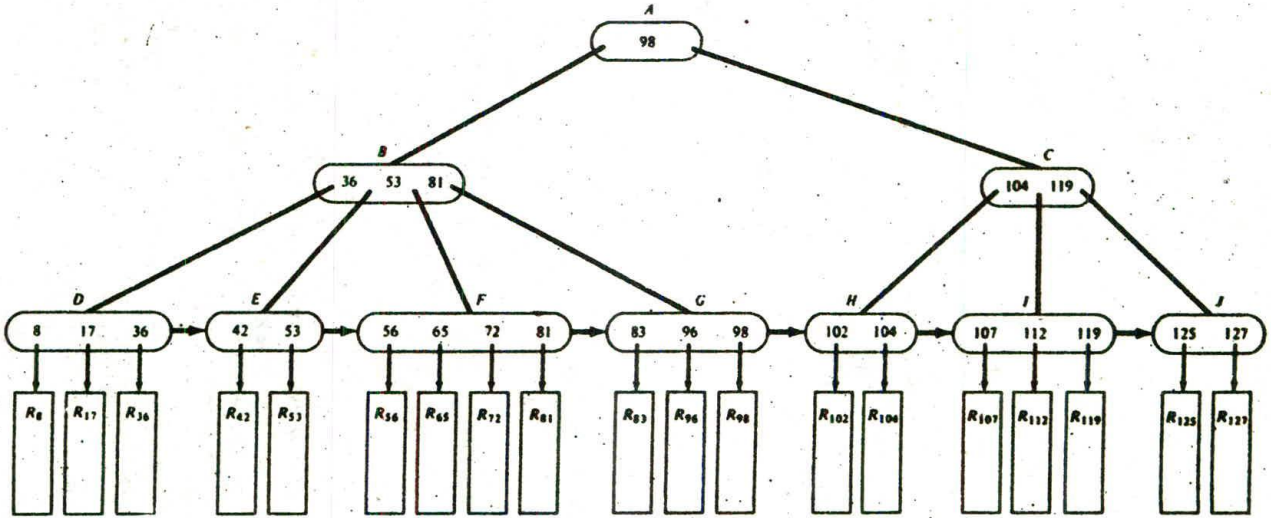

A
98
B
36 53 81
C
104 119
D
8 17 36
E
42 53
F
56 65 72 81
G
83 96 98
H
102 104
I
107 112 119
J
125 127
R8 R17 R36 R42 R53 R56 R65 R72 R81 R83 R96 R98 R102 R104 R107 R112 R119 R125 R127

Figure 7.3.16

Keys
180
185
1867
195
207
217
2174
21749
217493
226
27
274
278
279
2796
281
284
285
286
287
288
294
307
768

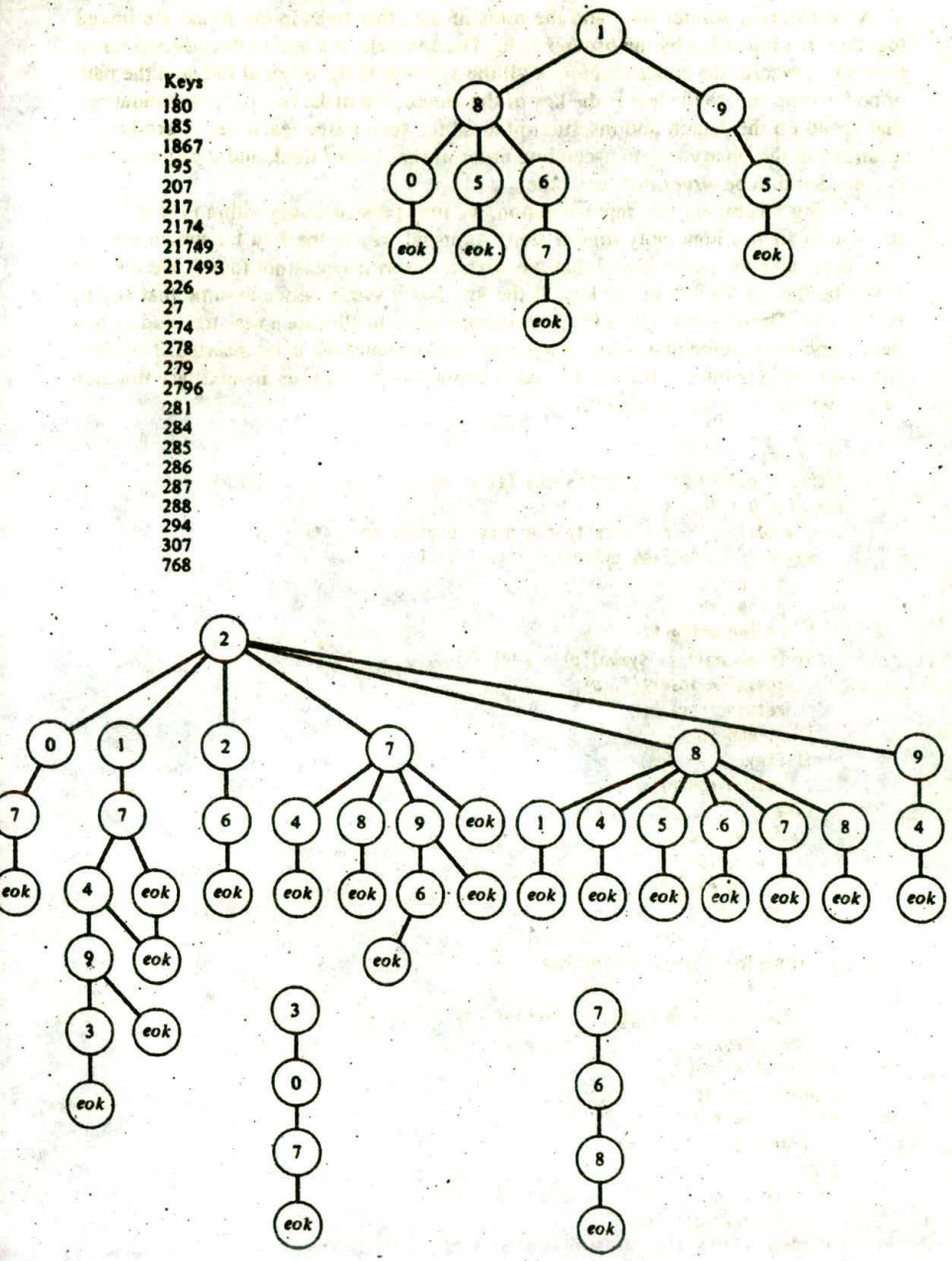


Figure 7.3.17 Forest representing a table of keys.

to by an external pointer *tree*, and the roots of the other trees in the forest are linked together in a linear list by the *brother* field. The *son* field of a leaf in the original forest points to a record; the concatenation of all the symbols in the original forest in the path of nodes from root to the leaf is the key of the record. We make two further stipulations that speed up the search and insertion process for such a tree: each list of brothers is arranged in the binary tree in ascending order of the *symbol* field, and the symbol *eok* is considered to be larger than any other.

Using this binary tree representation, we may present an algorithm to search and insert into such a nonempty ***digital tree***. As usual, *key* is the key for which we are searching, and *rec* is the record that we wish to insert if *key* is not found. We also let *key(i)* be the *i*th symbol of the key. If the key has *n* symbols, we assume that *key(n)* equals *eok*. The algorithm uses the *getnode* operation to allocate a new tree node when necessary. We assume that *recptr* is a pointer to the record *rec* to be inserted. The algorithm returns a pointer to the record that is being sought and uses an auxiliary function *insert*, whose algorithm is also given.

```
p = tree;
father = null;  /*  father is the father of p  */
for (i = 0;; i++) {
   q = null;  /*  q points to the other brother of p  */
   while (p != null && symbol(p) < key(i)) {
      q = p;
      p = brother(p);
   }  /* end while */
   if (p == null || symbol(p) > key(i)) {
      insval = insert(i, p);
      return(insval);
   }  /* end if */
   if (key(i) == eok)
      return(son(p));
   else {
      father = p;
      p = son(p);
   }  /* end else */
}  /* end for */
```

The algorithm for *insert* is as follows:

```
/*  insert the ith symbol of the key  */
s = getnode();
symbol(s) = key(i);
brother(s) = p;
if (tree == null)
   tree = s;
else
   if (q != null)
      brother(q) = s;
   else
      (father == null) ? tree = s : son(father) = s;
```

```
/* insert the remaining symbols of the key */
for (j = i; key(j) != eok; j++) {
   father = s;
   s = getnode();
   symbol(s) = key(j + 1);
   son(father) = s;
   brother(s) = null;
} /* end for */
son(s) = addr(rec);
return(son(s));
```

Note that by keeping the table of keys as a general tree, we need search only a small list of sons to find whether a given symbol appears at a given position within the keys of the table. However, it is possible to make the tree even smaller by eliminating those nodes from which only a single leaf can be reached. For example, in the keys of Figure 7.3.17, once the symbol '7' is recognized, the only key that can possibly match is 768. Similarly, upon recognizing the two symbols '1' and '9', the only matching key is 195. Thus the forest of Figure 7.3.17 can be abbreviated to the one of Figure 7.3.18. In that figure a box indicates a key and a circle indicates a tree node. A dashed line is used to indicate a pointer from a tree node to a key.

There are some significant differences between the trees of Figures 7.3.17 and 7.3.18. In Figure 7.3.17, a path from a root to a leaf represents an entire key; thus there is no need to repeat the key itself. In Figure 7.3.18, however, a key may be recognized only by its first few symbols. In those cases in which the search is made for a key that is known to be in the table, upon finding a leaf the record corresponding to that key can be accessed. If, however, as is more likely, it is not known whether the key is present in the table, it must be confirmed that the key is indeed correct. Thus the entire key must be kept in the record as well. Furthermore, a leaf node in the tree of Figure 7.3.17 can be recognized because its contents are *eok*. Thus its *son* pointer can be used to point to the record that that leaf represents. However, a leaf node of Figure 7.3.18 may contain any symbol. Thus, to use the *son* pointer of a leaf to point to the record, an extra field is required in each node to indicate whether or not the node is a leaf. We leave the representation of the forest of Figure 7.3.18 and the implementation of a search-and-insert algorithm for it as an exercise for the reader.

The binary tree representation of a digital search tree is efficient when each node has relatively few sons. For example, in Figure 7.3.18 only one node has as many as six (out of a possible ten) sons, whereas most nodes have only one, two, or three sons. Thus, the process of searching through the list of sons to match the next symbol in the key is relatively efficient. However, if the set of keys is dense within the set of all possible keys (that is, if almost any possible combination of symbols actually appears as a key), most nodes will have a large number of sons, and the cost of the search process becomes prohibitive.

Tries

A digital search tree need not be implemented as a binary tree. Instead, each node in the tree can contain m pointers, corresponding to the m possible symbols in

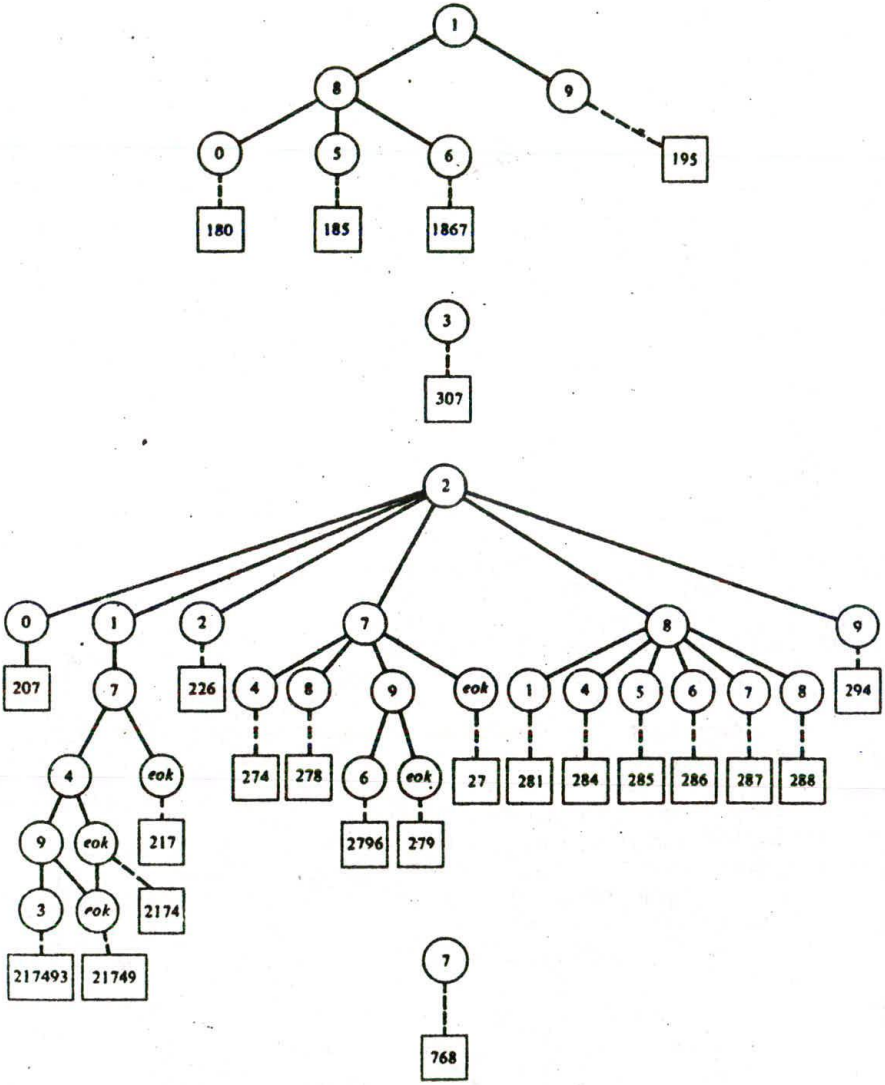


Figure 7.3.18 Condensed forest representing a table of keys.

each position of the key. Thus, if the keys were numeric, there would be 10 pointers in a node, and if strictly alphabetic, there would be 26. (There might also be an extra pointer corresponding to *eok*, or a flag with each pointer indicating that it pointed to a record rather than to a tree node.) A pointer in a node is associated with a particular symbol value based on its position in the node; that is, the first pointer corresponds to the lowest symbol value, the second pointer to the second lowest, and so forth. It is therefore unnecessary to keep the symbol values themselves in the tree. The number of nodes that must be accessed to find a particular key is log *mn*. A digital search tree implemented in this way is called a ***trie*** (from the word re*trie*val).

A trie is useful when the set of keys is dense, so that most of the pointers in each node are used. When the key set is sparse, a trie wastes a large amount of space with large nodes that are mostly empty. If the set of keys in a trie is known in advance and does not change, there are a number of techniques for minimizing the space requirements. One technique is to establish a different order in which the symbols of a key are used for searching (so that, for example, the third symbol of the argument key might be used to access the appropriate pointer in the trie root, the first symbol in level 1 nodes, and so forth). Another technique is to allow trie nodes to overlap each other, so that occupied pointers of one node overlay empty pointers of another.

EXERCISES

7.3.1. Show how a B-tree and a B^+-tree can be used to implement a priority queue (see Sections 4.1 and 6.3). Show that any sequence of n insertion and minimum-deletion operations can be performed in $O(n \log n)$ steps. Write C routines to insert and delete from a priority queue implemented by a 2-3 tree.

7.3.2. Choose any large paragraph from a book. Insert each word of the paragraph, in order, into an initially empty top-down multiway search tree of order 5, omitting any duplicates. Do the same for a B-tree of order 5, a B^+-tree of order 5, and a digital search tree.

7.3.3. Write C routines to implement the B-tree successor insertion operations if the B-tree is maintained

(a) In internal memory

(b) In external direct access storage

7.3.4. Write an algorithm and a C routine to delete a record from a top-down multiway search tree of order n.

7.3.5. Write an algorithm and a C routine to delete a record from a B-tree of order n.

7.3.6. Write an algorithm to create a compact B-tree from input in sorted order. Use the algorithm to write a C routine to produce a compact B-tree from an ordinary B-tree.

7.3.7. Write an algorithm and a C routine to search in a B-tree.

7.3.8. Write an algorithm and a C routine to

(a) Insert into a B^+-tree

(b) Insert into a B^*-tree

(c) Delete from a B^+-tree

(d) Delete from a B^*-tree

7.3.9. How many different 2-3 trees containing the integers 1 through 10 can you construct? How many permutations of these integers result in each tree if they are inserted into an initially empty tree in permutation order?

7.3.10. Develop algorithms to search and insert into a B-tree that uses front and rear compression.

7.3.11. Write a search-and-insert algorithm and C routine for the digital search forest of Figure 7.3.18.

7.3.12. Show how to implement a trie in external storage. Write a C search-and-insert routine for a trie.

7.4 HASHING

In the preceding two sections we assumed that the record being sought is stored in a table and that it is necessary to pass through some number of keys before finding the desired one. The organization of the file (sequential, indexed sequential, binary tree, and so forth) and the order in which the keys are inserted affect the number of keys that must be inspected before obtaining the desired one. Obviously, the efficient search techniques are those that minimize the number of these comparisons. Optimally, we would like to have a table organization and search technique in which there are no unnecessary comparisons. Let us see if this is feasible.

If each key is to be retrieved in a single access, the location of the record within the table can depend only on the key; it may not depend on the locations of other keys, as in a tree. The most efficient way to organize such a table is as an array (that is, each record is stored at a specific offset from the base address of the table). If the record keys are integers, the keys themselves can serve as indices to the array.

Let us consider an example of such a system. Suppose that a manufacturing company has an inventory file consisting of 100 parts, each part having a unique two-digit part number. Then the obvious way to store this file is to declare an array

```
parttype part[100];
```

where *part*[*i*] represents the record whose part number is *i*. In this situation, the part numbers are keys that are used as indices to the array. Even if the company stocks fewer than 100 parts, the same structure can be used to maintain the inventory file. Although many locations in *part* may correspond to nonexistent keys, this waste is offset by the advantage of direct access to each of the existent parts.

Unfortunately, however, such a system is not always practical. For example, suppose that the company has an inventory file of more than 100 items and the key to each record is a seven-digit part number. To use direct indexing using the entire seven-digit key, an array of 10 million elements would be required. This clearly wastes an unacceptably large amount of space because it is extremely unlikely that a company stocks more than a few thousand parts.

What is necessary is some method of converting a key [illegible] integer within a limited range. Ideally, no two keys should be converted into the same integer. Unfortunately, such an ideal method usually does not exist. Let us attempt to develop methods that come close to the ideal, and determine what action to take when the ideal is not achieved.

Let us reconsider the example of a company with an inventory file in which each record is keyed by a seven-digit part number. Suppose that the company has fewer than 1000 parts and that there is only a single record for each part. Then an array of 1000 elements is sufficient to contain the entire file. The array is indexed by an integer between 0 and 999 inclusive. The last three digits of the part number are used as the index for the part's record in the array. This is illustrated in Figure 7.4.1. Note that two keys that are relativeiy close to each other numericaliy, such as 4618396 and 4618996, may be farther from each other in the table than two keys that are widely separated numerically, such as 0000991 and 9846995. Only the last three digits of the key are used in determining the position of a record.

A function that transforms a key into a table index is called a ***hash function***. If *h* is a hash function and *key* is a key, *h*(*key*) is called the ***hash of key*** and is the index at which a record with the key *key* should be placed. If *r* is a record whose key hashes into *hr*, *hr* is called the ***hash key*** of *r*. The hash function in the preceding example is *h*(*k*) = *key* % 1000. The values that *h* produces should cover the entire set of indices in the table. For example, the function *x* % 1000 can produce any integer between 0 and 999, depending on the value of *x*. As we shall see shortly, it is a good idea for the table

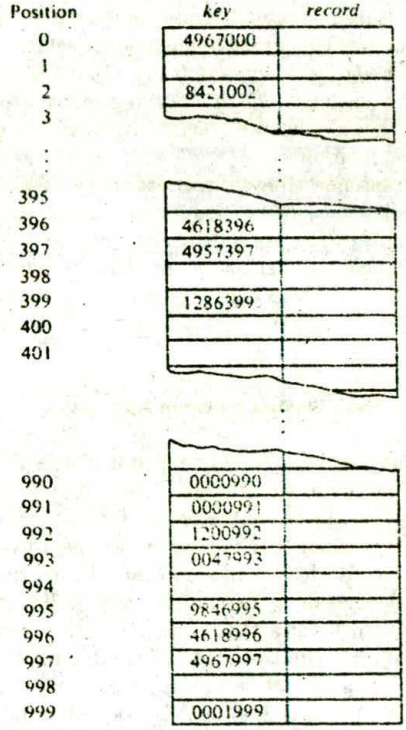


Figure 7.4.1

size to be somewhat larger than the number of records that are to be inserted. This is illustrated in Figure 7.4.1, where several positions in the table are unused.

The foregoing method has one flaw. Suppose that two keys $k1$ and $k2$ are such that $h(k1)$ equals $h(k2)$. Then when a record with key $k1$ is entered into the table, it is inserted at position $h(k1)$. But when $k2$ is hashed, because its hash key is the same as that of $k2$, an attempt may be made to insert the record into the same position where the record with key $k1$ is stored. Clearly, two records cannot occupy the same position. Such a situation is called a ***hash collision*** or a ***hash clash***. A hash clash occurs in the inventory example of Figure 7 4.1 if a record with key 0596397 is added to the table.

There are two basic methods of dealing with a hash clash. We explore both in detail in the remainder of this section. Briefly, the first technique, called ***rehashing***, involves using a secondary hash function on the hash key of the item. The rehash function is applied successively until an empty position is found where the item can be inserted. If the hash position of the item is found to be occupied during a search, the rehash function is again used to locate the item. The second technique, called ***chaining***, builds a linked list of all items whose keys hash to the same values. During search, this short linked list is traversed sequentially for the desired key. This technique involves adding an extra link field to each table position.

However, it should be noted that a good hash function is one that minimizes collisions and spreads the records uniformly throughout the table. That is why it is desirable to have the array size larger than the number of actual records. The larger the range of the hash function, the less likely it is that two keys yield the same hash value. Of course, this involves a space/time trade-off. Leaving empty spaces in the array is inefficient in terms of space; but it reduces the necessity of resolving hash clashes and is therefore more efficient in terms of time.

Although hashing allows direct access to a table and is therefore preferable to other search techniques, the method has one serious flaw. Items in a hash table are not stored sequentially by key, nor is there any generally practical method for traversing the items in key sequence. ***Order-preserving hash functions***, in which $h(key1) > h(key2)$ whenever $key1 > key2$, are usually nonuniform; that is, they do not minimize hash collisions and so do not serve the basic purpose of hashing: rapid access to any record directly from its key.

Resolving Hash Clashes by Open Addressing

Let us consider what would happen if it were desired to enter a new part number 0596397 into the table of Figure 7.4.1. Using the hash function $key \% 1000$, $h(0596397) = 397$; therefore the record for that part belongs in position 397 of the array. However, position 397 is already occupied by the record with key 4957397. Therefore the record with key 0596397 must be inserted elsewhere in the table.

The simplest method of resolving hash clashes is to place the record in the next available position in the array. In Figure 7.4.1, for example, since position 397 is already occupied, the record with key 0596397 is placed in location 398, which is still open. Once that record has been inserted, another record that hashes to either 397 (such as 8764397) or 398 (such as 2194398) is inserted at the next available position, which is 400.

This technique is called ***linear probing*** and is an example of a general method for resolving hash clashes called ***rehashing*** or ***open addressing***. In general, a rehash function, *rh*, accepts one array index and produces another. If array location *h*(*key*) is already occupied by a record with a different key, *rh* is applied to the value of *h*(*key*) to find another location where the record may be placed. If position *rh*(*h*(*key*)) is also occupied, it too is rehashed to see if *rh*(*rh*(*h*(*key*))) is available. This process continues until an empty location is found. Thus we may write a search and insertion function using hashing as follows. We assume the following declarations:

```
#define TABLESIZE...
typedef KEYTYPE ...
typedef RECTYPE ...
struct record {
   KEYTYPE k;
   RECTYPE r;
} table[TABLESIZE];
```

We also assume a hash function *h*(*key*) and a rehash function *rh*(*i*). The special value *nullkey* is used to indicate an empty record.

```
int search(KEYTYPE key, RECTYPE rec)
{
   int i;
   i = h(key);  /*  hash the key  */
   while (table[i].k != key && table[i].k != nullkey)
      i = rh(i);  /*  rehash  */
   if (table[i].k == nullkey) {
      /*  insert the record into the empty position  */
      table[i].k = key;
      table[i].r = rec;
   } /* end if */
   return(i);
} /* end search */
```

In the example of Figure 7.4.1, *h*(*key*) is the function *key* % 1000, and *rh*(*i*) is the function (*i* + 1) % 1000 (that is, the rehash of any index is the next sequential position in the array, except that the rehash of 999 is 0).

Let us examine the algorithm more closely to see if we can determine the properties of a "good" rehash function. In particular, we focus our attention on the loop, because the number of iterations determines the efficiency of the search. The loop can be exited in one of two ways: either *i* is set to a value such that *table*[*i*].*k* equals *key* (in which case the record is found), or *i* is set to a value such that *table*[*i*].*k* equals *nullkey* (in which case an empty position is found and the record may be inserted).

It may happen, however, that the loop executes forever. There are two possible reasons for this. First, the table may be full, so that it is impossible to insert any new records. This situation can be detected by keeping a count of the number of records in the table. When the count equals the table size, no additional insertions are attempted.

However, it is possible for the algorithm to loop indefinitely even if there are some (or even many) empty positions. Suppose, for example, that the function $rh(i) = (i + 2) \% 1000$ is used as a rehash function. Then any key that hashes into an odd integer rehashes into successive odd integers, and any key that hashes into an even integer rehashes into successive even integers. Consider the situation in which all the odd positions of the table are occupied and all the even ones are empty. Despite the fact that half the positions of the array are empty, it is impossible to insert a new record whose key hashes into an odd integer. Of course, it is unlikely that all the odd positions are occupied, while none of the even positions are. However, if the rehash function $rh(i) = (i + 200) \% 1000$ is used, each key can be placed in only one of five positions [since $x \% 1000 = (x + 1000) \% 1000$], and it is quite possible for these five places to be full while much of the table is empty.

One property of a good rehash function is that for any index i, the successive rehashes $rh(i)$, $rh(rh(i))$, ... cover as many of the integers between 0 and *tablesize* − 1 as possible (ideally, all of them). The rehash function $rh(i) = (i + 1) \% 1000$ has this property. In fact, any function $rh(i) = (i + c) \% tablesize$, where c is a constant value such that c and *tablesize* are relatively prime (that is, they cannot both be divided evenly by a single integer other than 1), produces successive values that cover the entire table. You are invited to confirm this fact by choosing some examples; the proof is left as an exercise. In general, however, there is no reason to choose a value of C other than 1. If the hash table is stored in external storage, it is desirable to have successive references as close to each other as possible (this minimizes seek delay on disks and may eliminate an *I/O* if the two references are on the same page).

There is another measure of the suitability of a rehash function. Consider the case of the linear rehash. Assuming that the hash function produces indices that are uniformly distributed over the interval 0 through *tablesize* − 1 [that is, it is equally likely that $h(key)$ is any particular integer in that range], then initially, when the entire array is empty, it is equally likely that a random record will be placed at any given (empty) position within the array. However, once entries have been inserted and several hash clashes have been resolved, this is no longer true. For example, in Figure 7.4.1 it is five times as likely for a record to be inserted at position 994 than at position 401. This is because any record whose key hashes into 990, 991, 992, 993, or 994 will be placed in 994, whereas only a record whose key hashes into 401 will be placed in that location. This phenomenon, where two keys that hash into different values compete with each other in successive rehashes, is called ***primary clustering***.

The same phenomenon occurs in the case of the rehash function $rh(i) = (i + c) \% tablesize$. For example, if *tablesize* = 1000, $c = 21$, and positions 10, 31, 52, 73, and 94 are all occupied, any record whose key is any one of these five integers will be placed at location 115. In fact, any rehash function that depends solely on the index to be rehashed causes primary clustering.

One way of eliminating primary clustering is to allow the rehash function to depend on the number of times that the function is applied to a particular hash value. In this approach the function rh is a function of two arguments. $rh(i,j)$ yields the rehash of the integer i if the key is being rehashed for the jth time. One example is $rh(i,j) = (i + j) \% tablesize$. The first rehash yields $rh1 = rh(h(key),1) = (h(key) + 1) \% table-$

size, the second yields *rh*2 = (*rh*1 + 2) % *tablesize*, the third yields *rh*3 = (*rh*2 + 3) % *tablesize*, and so on.

Another approach is to use a random permutation of the numbers between 1 and *t*, (where *t* equals *tablesize* − 1, the largest index of the table), *p*1, *p*2, . . . , *pt*, and to let the *j*th rehash of *h*(*key*) be (*h*(*key*) + *pj*) % *tablesize*. This has the advantage of ensuring that no two rehashes of the same key conflict. Still a third approach is to let the *j*th rehash of *h*(*key*) be (*h*(*key*) + *sqr*(*j*)) % *tablesize*. This is called the ***quadratic rehash***. . . another method of eliminating primary clustering is to allow the rehash to depend on the hash value, as in *rh*(*i*,*key*) = (*i* + *hkey*) % *tablesize*, where *hkey* = 1 + *h*(*key*) % *t*. (We cannot use *hkey* equal to *h*(*key*), which might be 0, or to *h*(*key*) + 1, which might be *tablesize*. Either of these cases would result in *rh*(*i*, *key*) equaling *i*, which is unacceptable). All these methods allow keys that hash into different locations to follow separate rehash paths.

However, although these methods eliminate primary clustering, they do not eliminate another phenomenon, known as ***secondary clustering***, in which different keys that hash to the same value follow the same rehash path. One way to eliminate all clustering is ***double hashing***, which involves the use of two hash functions, *h*1(*key*) and *h*2(*key*). *h*1, which is known as the ***primary hash function***, is first used to determine the position at which the record should be placed. If that position is occupied, the rehash function *rh*(*i*, *key*) = (*i* + *h*2(*key*)) % *tablesize* is used successively until an empty location is found. As long as *h*2(*key*1) does not equal *h*2(*key*2), records with keys *key*1 and *key*2 do not compete for the same set of locations. This is true despite the possibility that *h*1(*key*1) may indeed equal *h*1(*key*2). The rehash function depends not only on the index to be rehashed but also on the original key. Note that the value *h*2(*key*) does not have to be recomputed for each rehash: it need be computed only once for each key that must be rehashed. Optimally, one should choose functions *h*1 and *h*2 that distribute the hashes and rehashes uniformly over the interval 0 to *tablesize* − 1 and also minimize clustering. Such functions are not always easy to find.

An example of double hashing functions is *h*1(*key*) = *key* % *tablesize* and *h*2(*key*) = 1 + *key* % *t*, where *tablesize* is a prime number and *t* equals *tablesize* − 1. Another example is *h*1(*key*) as defined above and *h*2(*key*) = 1 + (*key*/*tablesize*) % *t*.

Deleting Items from a Hash Table

Unfortunately, it is difficult to delete items from a hash table that uses rehashing for search and insertion. For example, suppose that record *r*1 is at position *p*. To add a record *r*2 whose key *k*2 hashes into *p*, it must be inserted into the first free position from among *rh*(*p*), *rh*(*rh*(*p*)), ... Suppose that *r*1 is then deleted, so that position *p* becomes empty. A subsequent search for record *r*2 begins at position *h*(*k*2), which is *p*. But since that position is now empty, the search process may erroneously conclude that record *r*2 is absent from the table.

One possible solution to this problem is to mark a deleted record as "deleted" rather than "empty" and to continue searching whenever a "deleted" position is encountered in the course of a search. But this is possible only if there are a small number of deletions; otherwise, an unsuccessful search would require a search through the entire table because most positions will be marked "deleted" rather than "empty." Ideally,

we would prefer a deletion mechanism in which retrieval time is the same whenever n records are in the table, regardless of whether the n records are a result of n insertions or w insertions and $w - n$ subsequent deletions. Later in this section we examine alternatives to rehashing that allow us to accomplish this.

Efficiency of Rehashing Methods

The efficiency of a hashing method is usually measured by the average number of table positions that must be examined in searching for a particular item. This is called the number of ***probes*** required by the method. Note that in the algorithms we have presented, the number of key comparisons equals twice the number of probes, since the key at each probe position is compared with both the search argument and *nullkey*. However, the comparison with *nullkey* may be less expensive than the general key comparison. An additional field can also be used in each table position to indicate whether it is empty to avoid an extra key comparison.

Under rehashing, the average number of probes depends on both the hash function and the rehash method. The hash function is assumed to be uniform. That is, it is assumed that an arbitrary key is equally likely to hash into any table index as any other. Mathematical analysis of the average number of probes required to find an element in a hash table if the table had been constructed using a particular hash and rehash method can be quite involved. Let n be the number of items currently in the hash table, and let *tablesize* be the number of positions in the table. Then for large *tablesize*, it has been proved that the average number of probes required for a successful retrieval in a table organized using linear rehashing is approximately

$$\frac{2 * tablesize - n + 1}{2 * tablesize - 2 * n + 2}$$

If we set $x = (n - 1)/tablesize$, this equals $(2 - x)/(2 - 2x)$. Define the ***load factor*** of a hash table, *lf*, as $n/tablesize$, the fraction of the table that is occupied. Since *lf* is approximately equal to x for large *tablesize*, we may approximate the number of probes for a successful search under linear rehashing by $(2 - lf)/(2 - 2 * lf)$ or $0.5/(1 - lf) + 0.5$. When *lf* approximates 1 (that is, when the table is almost full), this formula is not useful. Instead, it can be shown that the average number of key comparisons for a successful search in an almost full table may be approximated by $sqrt(\pi * tablesize/8) + 0.33$.

For an unsuccessful search, the average number of probes in a table organized using linear rehashing is approximately equal to $0.5/(1 - lf)^2 + 0.5$ for large *tablesize*. When the table is full (that is, when $n = tablesize - 1$, since one position must be left open to detect that the key is not present), the average number of probes for an unsuccessful comparison under linear rehashing is $(tablesize + 1)/2$, which is the same as the average number of comparisons required to find a single empty slot among *tablesize* slots by sequential search.

For tables with low load factors, this performance is not unreasonable, but for high load factors it can be improved considerably. Eliminating primary clustering by setting $rh(i, key)$ to $(i + hkey)$ % *tablesize* as defined previously or by using quadratic rehashing sets the average number of probes to approximately $1 - \log(1 - lf) - lf/2$ for

successful retrievals and $1/(1 - lf) - lf - \log(1 - lf)$ for unsuccessful searches. (Here log is the natural logarithm as defined in the standard library *math.h*.) For full tables, successful search time approximates $\log(tablesize + 1)$, and unsuccessful search time remains at $(tablesize + 1)/2$.

Double hashing improves efficiency even further by eliminating both primary and secondary clustering. Uniform hashing is defined as any hashing scheme in which any newly inserted element is equally likely to be placed at any of the empty positions of the hash table. For such a theoretical scheme, it can be proved that successful search time is approximately $-\log(1 - lf)/lf$ for large *tablesize*, and that unsuccessful searching requires $(tablesize + 1)/(tablesize + 1 - n)$, or approximately $1/(1 - lf)$, probes for large *tablesize*. Experience with good double hashing functions shows that the average number of comparisons equals these theoretical values. For full tables, successful search time is approximately $\log(tablesize + 1) - 0.5$, and unsuccessful search time is again $(tablesize + 1)/2$.

The following table lists approximate number of probes for each of the three methods for various load factors. Recall that these approximations are generally only valid for large table sizes.

Load factor	Successful			Unsuccessful		
%	Linear	*i + hkey*	Double	Linear	*i + hkey*	Double
25	1.17	1.16	1.15	1.39	1.37	1.33
50	1.50	1.44	1.39	2.50	2.19	2.00
75	2.50	2.01	1.85	7.50	4.64	4.00
90	5.50	2.85	2.56	50.50	11.40	10.00
95	10.50	3.52	3.15	200.50	22.04	20.00

For full tables (in the successful case, where n equals *tablesize*; in the unsuccessful case, where n equals $tablesize - 1$), the following are some approximations of the average number of probes. We have also included the value of $\log_2(tablesize)$ for comparison with binary search and tree searching.

	Successful				
Tablesize	Linear	*i + hkey*	Double	Unsuccessful	$\log_2$(tablesize)
100	6.60	4.62	4.12	50.50	6.64
500	14.35	6.22	5.72	250.50	7.97
1,000	20.15	6.91	6.41	500.50	7.97
5,000	44.64	7.52	7.02	2,500.50	12.29
10,000	63.00	7.21	7.71	5,000.50	13.29

These data indicate that linear hashing should be strongly avoided for tables that become more than 75 percent full, especially if unsuccessful searches are common, since primary clustering does have a significant effect on search time for large load factors. The effects of secondary clustering, however, never add more than 0.5 probe to the average number required. Given the fact that double hashing requires an expensive

additional computation to determine $h2(key)$, it may be preferable to accept the extra half probe and use $rh(i, key) = (i + hkey) \% tablesize$.

One technique that can be used to improve the performance of linear rehashing is ***split sequence linear rehashing***. Under this technique, when $h(key)$ is found to be occupied, we compare key with the key kh located in position $h(key)$. If $kh < h(key)$, the rehash function $i + c1$ is used; if $kh > h(key)$, another rehash function, $i + c2$, is used. This splits the rehashes from a particular slot into two separate sequences and reduces clustering without requiring additional space or reordering the hash table. For tables with a load factor of 95 percent, the split sequence technique reduces the number of probes in a successful search by more than 50 percent and the number of probes in an unsuccessful search by more than 80 percent. However, nonlinear rehash methods are still better. A similar technique yields some, but not significant, improvement for the nonlinear rehash methods.

Another point to note regarding efficiency is that, in the context of rehashing, the modulus operation should not be obtained by using the system % operator, which involves division, but rather by a comparison and possibly a subtraction. Thus $rh(i, key) = (i + hkey) \% tablesize$ should be computed as follows:

```
x = i + hkey;
rh = x < tablesize  ?  x : x - tablesize;
```

The foregoing tables also indicate the great expense of an unsuccessful search in a nearly full table. Insertion also requires the same number of comparisons as unsuccessful search. When the table is nearly full, the insertion efficiency of hashing approaches that of sequential search and is far worse than tree insertion.

Hash Table Reordering

When a hash table is nearly full, many items in the table are not at the locations given by their hash keys. Many key comparisons must be made before finding some items. If an item is not in the table, an entire list of rehash positions must be examined before that fact is determined. There are several techniques for remedying this situation.

In the first technique, discovered by Amble and Knuth, the set of items that hash into the same location are maintained in descending order of the key. (We assume that *NULLKEY* is less than any key possibly occupying the table.) When searching for an item, it is not necessary to rehash repeatedly until an empty slot is found; as soon as an item in the table whose key is less than the search key is found, we know that the search key is not in the table. When inserting a key key, if a rehash accesses a key smaller than key, key and its associated record replace the smaller key in the table and the insertion process continues with the displaced key. A hash table organized in this way is called an ***ordered hash table***. The following is a search and insertion function for an ordered hash table. (Recall that *NULLKEY* is less than any other key.)

```
int search(KEYTYPE key, RECTYPE rec)
{
   int first, i, j;
   RECTYPE newentry, tempentry;
   KEYTYPE tk;
```

```
    i = h(key);
    newentry.k = key;
    newentry.r = rec;
    first = TRUE;
    while (table[i].k > newentry.k)
       i = rh(i);
    tk = table[i].k;
    while (tk != NULLKEY && tk != newentry.k) {
       /*  insert the new entry and displace  */
       /*        the entry at position i      */
       tempentry = table[i];
       table[i] = newentry;
       newentry = tempentry;
       if (first == TRUE) {
         j = i; /*  j is the position in which  */
                /*  the new record is inserted  */
         first = FALSE;
       } /* end if */
       i = rh(i);
       tk = table[i].k;
    } /*  end while  */
    if (tk == NULLKEY) {
       table[i] = newentry;
    if (first == TRUE)
       return(i);
    else
       return(j);
} /* end search */
```

The ordered hash table method can be used with any rehashing technique in which a rehash depends only on the index and the key; it cannot be used with a rehash function that depends on the number of times the item is rehashed (unless that number is kept in the table).

Using an ordered hash table does not change the average number of key comparisons required to find a key that is in the table, but it reduces significantly the number of key comparisons necessary to determine that a key does not exist in the table. It can be shown that unsuccessful search in an ordered hash table requires the same average number of probes as successful search (in an ordered or unordered table). This is a significant improvement. Unfortunately, however, the average number of probes for insertion is not reduced in an ordered hash table and equals the number required for an unsuccessful search in an unordered table. Ordered hash table insertions also require a significant number of hash table modifications.

Brent's Method

A different reordering scheme, attributable to Brent, can be used to improve the average search time for successful retrievals when double hashing is used. The technique involves rehashing the search argument until an empty slot is found. Then each of the keys in the rehash path is itself rehashed to determine if placing one of those

keys in an empty slot would require fewer rehashes. (Recall that, under double hashing, the rehash paths for two keys that rehash to the same slot will diverge.) If this is the case, the search argument replaces the existing key in the table and the existing key is inserted in its empty rehash slot.

The following is a routine to implement Brent's search and insertion algorithm. It uses auxiliary routines *setempty*, which initializes a queue of table indexes to empty: *insert*, which inserts an index onto a queue: *remove*, which returns an index removed from a queue: and *freequeue*, which frees all nodes of a queue.

```
int search(KEYTYPE key, RECTYPE rec)
{
   struct queue qq;  /*  of table indexes  */
   int i, j, jj, minoldpos, minnewpos;
   int count, mincount, rehashcount, displacecount;
   KEYTYPE displacekey;


   setempty(qq);
   /*   rehash repeatedly, placing each successive index   */
   /*    in the queue and keeping a count of the number    */
   /*                  of rehashes required                */
   i = h1(key);
   for (count = 0; table[i].k != key && table[i].k != nullkey;
                                                    count++) {
      insert(qq, i);
      i = rh(i, key);
   } /* end for */
   /*  minoldpos and minnewpos hold the initial and final  */
   /*   indexes of the key on the rehash path of key that  */
   /*     can be displaced with a minimum of rehashing.    */
   /*    Initially, assume no displacement and set them    */
   /*       both to i, the first empty index for key       */
   minoldpos = i;
   minnewpos = i;
   /*   mincount is the minimum number of rehashes of key  */
   /*    plus rehashes of the displaced key, displacekey.  */
   /*      rehashcount is the number of rehashes of key    */
   /*       needed to reach the index of the key being     */
   /*     displaced.  Initially, assume no displacement.   */
   mincount = count;
   rehashcount = 0;
   /*    The following loop determines if displacement of  */
   /*     the key at the next rehash of key will yield a   */
   /*    lower total number of rehashes.  If key was found */
   /*            in the table, then skip the loop          */
   if (table[i].k == nullkey)
      while (!empty(qq) && rehashcount+1 < mincount) {
         j = remove(qq);
         /*  the candidate key for displacement  */
```

```
            displacekey = table[j].k;
            jj = rh(j, displacekey);
            /*  displacecount is the number of rehashes  */
            /*      required to displace displacekey.    */
            for (displacecount = 1; table[jj].k != nullkey;
                                              displacecount++)
               jj = rh(jj, displacekey);
            if (rehashcount+displacecount <.mincount) {
               mincount = rehashcount + displacecount;
               minoldpos = j;
               minnewpos = jj;
            } /* end if */
            rehashcount++;
         } /* end while */
      /*  free any extra items on the queue  */
      freequeue(qq);
      /*    At this point, if no displacement is necessary    */
      /*    minoldpos equals minnewpos.  minoldpos is the     */
      /*  position where key was found or should be inserted. */
      /*    minnewpos (if not equal to minoldpos) is the      */
      /*  position where the key displaced from minoldpos     */
      /*                should be placed,                     */
      if (minoldpos != minnewpos)
         table[minnewpos] = table[minoldpos];
      if (minoldpos != minnewpos || table[minoldpos].k == nullkey) {
         table[minoldpos].k = key;
         table[minoldpos].r = rec;
      } /* end if */
      return(minoldpos);
   } /* end search */
```

Brent's method reduces the average number of comparisons for successful retrievals but has no effect on the number of comparisons for unsuccessful searches. Also, the effort required to insert a new item is increased substantially.

An extension of Brent's method that yields even greater improvements in retrieval times at the expense of correspondingly greater insertion time involves recursive insertion of items displaced inp the table. That is, in determining the minimum number of rehashes required to displace an item on a rehash path, all the items on that item's subsequent rehash path are considered for displacement as well, and so on. However, the recursion cannot be allowed to proceed to its natural conclusion, since insertion time would then become so large as to become impractical, even though insertion is infrequent. A maximum recursion depth of 4, plus an additional modification by which tentative rehash paths longer than 5 are penalized excessively, has been found to yield average retrievals very close to optimal with reasonable efficiency.

The following table shows the average number of probes required for retrieval and insertion under the unmodified Brent algorithm. The last column shows the number of retrievals per item required to make Brent's algorithm worthwhile (that is, so that the cumulative advantage on retrievals outweighs the disadvantage on insertion).

Load factor %	Probes/ retrieval	Probes/ insertion	Breakeven numbers of retrieval/item
20	1.10	1.15	2.85
60	1.37	1.92	2.48
80	1.60	2.97	2.32
90	1.80	4.27	2.26
95	1.97	5.84	2.26

As the table becomes full, approximately 2.5 probes per retrieval are required on the average, regardless of the table size. This compares very favorably with ordinary double hashing, in which retrieval from a full table requires $O(\log n)$ probes.

Binary Tree Hashing

Another method of improving Brent's algorithm is attributable to Gonnet and Munro and is called ***binary tree hashing***. Again, we assume the use of double hashing. Every time a key is to be inserted into the table, an almost complete binary tree is constructed. Figure 7.4.2 illustrates an example of such a tree, in which the nodes are numbered according to the array representation of an almost complete binary tree, as outlined in Section 5.2 (that is, *node*(0) is the root and *node*($2 * i + 1$) and *node*($2 * i + 2$) are the left and right sons of *node*(i)). (The details of this figure will be explained shortly.) Each node of the tree contains an index into the hash table. For purposes of this discussion, the hash table index contained in *node*(i) will be referred to as *index*(i), and the key at that position (that is, *table*[*index*(i)].*k*) as $k(i)$. *key* is referred to as $k(-1)$.

To explain how the tree is constructed, we first define the ***youngest right ancestor*** of *node*[i], or *yra*(i), as the node number of the father of the youngest ancestor of *node*(i) that is a right son. For example, in Figure 7.4.2, *yra*(11) is 0, since *node*(11) (containing *l*) is a left son and its father *node*(5) (containing *f*) is also a left son. Thus the youngest ancestor of *node*(11) that is a right son is *node*(2) (containing *c*), and its father is *node*(0). Similarly, *yra*(19) is 1 and *yra*(17) is 3. If *node*(i) is a right son, *yra*(i) is defined as the node number of its father, $(i - 1)/2$. Thus *yra*(14) in Figure 7.4.2 is 6. If *node*(i) has no ancestor that is a right son (as, for example, nodes 0, 1, 3, 7, and 15 of Figure 7.4.2), *yra*(i) is defined as -1.

The binary tree is constructed in node number order. *index*(0) is set to *h*(*key*). *index*(i), for each subsequent i, is set to *rh*(*index*($(i - 1)/2$), k(*yra*(i))). This process continues until $k(i)$ (that is, *table*[*index*(i)].*k*) equals *NULLKEY* and an empty position is found in the table.

For example, in Figure 7.4.2, *key* is hashed to obtain a = *h*(*key*), which is established as the index in the root node. Its left son is b = *rh*(a, *key*), and its right son is c = *rh*(a, *table*(a).*k*) = *rh*(a,*k*(0)). Similarly, the left son of b is d = *rh*(b,*key*), the right son of b is e = *rh*(b, *table*(b).*k*) = *rh*(b,*k*(1)), the left son of c is f = *rh*(c, *table*(a).*k*) = *rh*(c,*k*(0)) and the right son of c is g = *rh*(c, *table*(c).*k*) = *rh*(c,*k*(2)). This continues until t = *rh*(j, *table*(b).*k*) = *rh*(j,*k*(1)) is placed in *node*(19) and u = *rh*(j, *table*(j).*k*) = *rh*(j,*k*(9)) is placed in *node*(20). Since *table*[u].*k* (which is k(20)) is *NULLKEY*, an empty position in the hash table has been found, and the

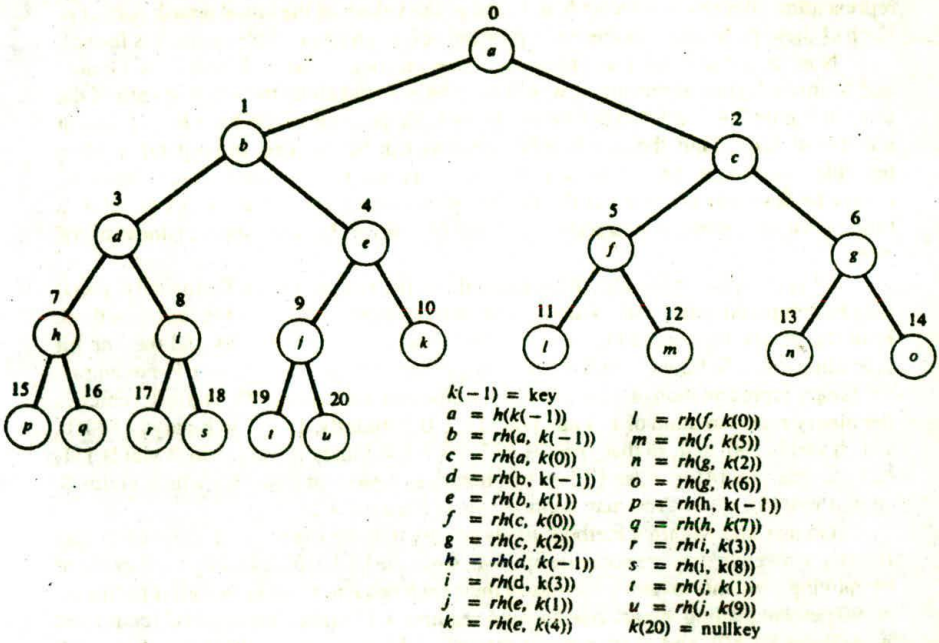


Figure 7.4.2

tree construction is completed. Note that any path following a series of left pointers through the tree consists of successive rehashes of a particular key and that a right pointer indicates that a new key is being rehashed.

Once the tree has been constructed, the keys along the path from the root to the last node are reordered in the hash table. Let i be initialized to the position of the last node of the tree. Then if $yra(i)$ is nonzero, $k(yra(i))$ and its associated record are moved from $table[index(yra(i))]$ to $table[index(i)]$ and i is reset to $yra(i)$. This process is repeated until $yra(i)$ is -1, at which point *key* and *rec* are inserted into $table[index(i)]$ and the insertion is complete. For example, in Figure 7.4.2, $yra(20) = 9$ and $index(9) = j$; consequently the key and record from position j of the hash table are moved to the previously empty position u. Then, since $yra(9) = 1$ and $index(1) = b$, the key and record from position b are moved to position j. Finally, since $yra(1) = -1$, *key* is inserted into position b.

When subsequently searching for *key*, two table positions are probed: a and b. When searching for the former $table[b].k$ (now at $table[j]$), two additional probes are required. When searching for the former $table[j].k$ (now at $table[u]$), one additional probe is required. Thus a total of five extra positions over the entire hash table contents must be probed as a result of the insertion of *key*, whereas at least six would have been required if *key* were inserted directly along its rehash path (consisting of a, b, d, h, p, and at least one more position). Similarly, under Brent's method, no path shorter than

length 6 would have been found (considering paths *abejt*, *abdir*, *abdhg*, and *abdhp*, representing attempts to relocate *b*, *d*, *h*, and *p*, the values on the initial rehash path of *a*. Each of these paths requires one more position before an empty table element is found).

Note also that if *key* had previously been inserted in the table using the Gonnet and Munro insertion algorithm, it would have been found along the leftmost path of the tree (in Figure 7.4.2, *abdhp*) before an empty table position would have been found at a different node. Thus the tree-building process can be initiated, in preparation for a possible insertion, as part of the search process. However, if insertions are infrequent, it may be desirable to construct the full left path of the tree until an empty position is found (that is, to perform a straight search for *key*) before building the remainder of the tree.

Of course, the entire algorithm depends on the routine *yra*(*i*). Fortunately, *yra*(*i*) may be computed quite easily. *yra*(*i*) can be derived directly from the following method: Find the binary representation of $i + 1$. Delete any trailing zero bits and the one bit preceding them. Subtract 1 from the resulting binary number to get *yra*(*i*). For example, the binary representation of 11 + 1 is 1100. Removing the trailing 100 yields 1, which is the binary representation of 1. Thus *yra*(11) = 0. Similarly, 17 + 1 in binary is 10010, which yields 100, or 4, so that *yra*(17) = 3; 14 + 1 in binary is 1111, which yields 111, or 7, so that *yra*(14) = 6; and 15 + 1 in binary is 10000 (or 010000), which yields 0, so that *yra*(15) = −1. You may confirm this in Figure 7.4.2.

Gonnet and Munro's method yields results that are even closer to optimal than Brent's. However, they are not quite optimal, since the hash table can only be rearranged by moving elements to later positions in their hash sequence, never to earlier positions. At 90 percent loading, binary tree hashing requires 1.75 probes per retrieval (compared with Brent's 1.80), and at 95 percent requires 1.88 (compared with 1.97). For a full table, 2.13 probes are required on average, compared with Brent's 2.5. The maximum number of probes required to access an element under Brent's method is $O(sqrt(n))$, whereas under binary hashing it is $O(\log n)$. Note that the queue of Brent's method and the tree of Gonnet and Munro's can be reused for each insertion. If all insertions take place initially, and the table is subsequently required only for searches, the space for these data structures may be freed.

If the hash table is static (that is, if elements are initially inserted and the table remains unchanged for a long series of searches), another reordering strategy is to perform all the insertions initially and then to rearrange the elements of the table to absolutely minimize the expected retrieval costs. Experiments show that the minimum expected retrieval cost is 1.4 probes per retrieval with a load factor of .5, 1.5 for a load factor of .8, 1.7 for a load factor of .95, and 1.83 for a full table. Unfortunately, algorithms to optimally reorder a table to achieve this minimum are $O(n * tablesize^2)$ and are therefore impractical for large numbers of keys.

Improvements with Additional Memory

Thus far we have assumed that no additional memory is available in each table element. If additional memory is available, we can maintain some information in each entry to reduce the number of probes required to find a record or to determine that the desired record is absent.

Before looking at specific techniques, we should make one observation. The most obvious use to which additional memory can be put is to expand the size of the hash table. This reduces the load factor and immediately improves efficiency. Therefore in evaluating any efficiency improvements caused by adding more information to each table entry, one must consider whether the improvement outweighs utilizing the memory to expand the table.

On the other hand, the benefit of expanding each table entry by one or two bytes may indeed be worthwhile. Each table item (including space for the key and record) may require 10, 50, 100, or even 1000 bytes, so that utilizing the space to expand the table may not buy as much as utilizing the space for small increments in each table element. (In reality, long records would not be kept within a hash table, since empty table entries waste too much space. Instead, each table entry would contain the key and a pointer to the record. This could still require 30 or 40 bytes if the key were large and 10 to 15 bytes for typical key sizes.) In addition, for technical reasons (for example, word size), not all of the space in a table entry may actually be used, so that there may be some extra space available that cannot be used for additional table entries. In that case, whatever use can be made of the storage is beneficial.

The first improvement that we consider reduces the time required for an unsuccessful search, but not that for a retrieval. It involves keeping with each table element a one-bit field whose value is initialized to 0 and is set to 1 whenever a key to be inserted hashes or rehashes to that position but the position is found occupied. When hashing or rehashing a key during a search and finding the bit still set to 0, we immediately know that the key is not in the table, since if it were, it would either be found in that position or the bit would have been reset to 1 when it or some other key had been inserted. Use of this technique together with the ordered hash table algorithm of Amble and Knuth reduces the average number of probes for an unsuccessful search in a table with a load factor of 95 percent from 10.5 to 10.3 using linear rehashing and from 3.15 to 2.2 using double hashing. This method is called the ***pass-bit method***, since the additional bit indicates whether a table element has been passed over while inserting an item.

The next method can be used with both linear rehashing and quadratic rehashing. In both cases we can define a function *prb*(*j*, *key*) that directly computes the position of the *j*th rehash of *key*, which is the position of the *j*th probe in searching for *key*. *prb*(0, *key*) is defined as *h*(*key*). For linear rehashing [*rh*(*i*) = (*i* + *c*) % *tablesize*, where *c* is a constant], *prb*(*j*, *key*) is defined as (*h*(*key*) + *j***c*) % *tablesize*. For quadratic rehashing, *prb*(*j*, *key*) is defined as (*h*(*key*) + *sqr*(*j*)) % *tablesize*. Note that no such routine can be defined for double hashing; therefore the method is not applicable to that technique.

The method uses an additional integer field, called a *predictor*, in each table position. Let *prd*(*i*) be the predictor field in table position *i*. Initially, all predictor fields are 0. Under linear rehashing, the predictor field is reset as follows. Suppose that key *k*1 is being inserted and that *j* is the smallest integer such that *prb*(*j*,*k*1) is a probe position whose predictor field *prd*(*prb*(*j*,*k*1)) is 0. Then, after *k*1 is rehashed several more times and is inserted in position *prb*(*p*,*k*1), *prd*(*prb*(*j*,*k*1)) is reset from 0 to *p* − *j*. Then, during a search, when position *prb*(*j*,*k*1) is found not to contain *k*1, the next position examined is *prb*(*j* + *prd*(*prb*(*j*,*k*1)), *k*1) or *prb*(*p*,*k*1) rather than *prb*(*j* + 1,*k*1). This eliminates *p* − *j* − 1 probes.

An advantage of this approach is that it can be adapted quite easily when only a few extra bits are available in each table position. Since the predictor field contains only the number of additional rehashes needed, in most cases this number is low and can fit in the available space. It would be very rare for a predictor value to be greater than the largest integer expressible in four or five bits. If only b bits are available for the *prd* field and the predictor field cannot fit, the field value can be set to $2^b - 1$ (the largest integer representable by b bits). Then we would skip at least $2^b - 2$ probes after reaching such a position.

Under linear rehashing, suppose two keys, $k1$ and $k2$, hash into different values but the mth probe of one equals the nth probe of the other. [That is, $prb(n,k1) = prb(m,k2)$, where n and m are unequal.] Suppose that $k1$ is inserted first into position $i = prb(m,k1)$. Then when $k2$ is placed in position $prb(m + x, k2)$, $prd(i)$ is set to x. This presents no problem, since $prb(n + x,k1)$ and $prb(m + x,k2)$ both equal $(i + x * c)$ % *tablesize*. Thus anything that hashes into either $h(k1)$ or $h(k2)$ and rehashes into i can be referred to $i + prd(i)$ for the next probe. This reflects the fact that linear rehashing involves primary clustering in which keys hashing into different locations follow the same rehash paths once those paths intersect.

Under quadratic rehashing, however, primary clustering is eliminated. Thus, the paths followed by $k1$ and $k2$ differ if $h(k1)$ does not equal $h(k2)$, even after those paths intersect. Thus $prb(n + x,k1)$, which equals $(h(k1) + sqr(n + x))$ % *tablesize*, does not equal $prb(m + x,k2)$, which equals $(h(k2) + sqr(m + x))$ % *tablesize*, even though $prb(n,k1)$ does happen to equal $prb(m,k2)$. Thus, if $prd(prb(i,k1))$ is set to x, and if we are searching for $k2$ at $prb(j,k2)$, which happens to equal $prb(i,k1)$, we cannot directly go to $prb(j + x,k2)$ unless $h(k1)$ equals $h(k2)$ (that is, $k1$ and $k2$ are in the same secondary cluster) and i equals j. Therefore under quadratic rehashing or any other rehashing method that involves only secondary clustering, we must ensure that $h(k(i))$ equals $h(key)$ before using or setting $prd(i)$ during a search or insertion of *key*. If the two hash values are unequal, rehashing continues in the usual fashion until a location j is reached where $h(k(j))$ equals $h(key)$, where use of the *prd* field can be resumed.

Unfortunately, the predictor method cannot be used at all under double hashing. The reason for this is that even secondary clustering is eliminated, so that there is no guarantee that $prb(n + x,k1)$ equals $prb(n + x,k2)$ even if $h(k1)$ equals $h(k2)$ and $prb(n,k1)$ equals $prb(n,k2)$.

An extension of the predictor method is the ***multiple predictor method.*** Under this technique, np predictor fields are maintained in each table position. A predictor hash routine $ph(key)$, whose value is between 0 and $np - 1$, determines which predictor is used for a particular key. The jth predictor in table position i is referenced as $prd(i,j)$. When a key probes an occupied slot i that equals $prb(j, key)$ such that $ph(k(i))$ equals $ph(key)$, the next position probed is $prb(j + prd(i,ph(key)), key)$. Similarly, if $ph(k(i))$ equals $ph(key)$ and $prd(i,ph(key))$ is 0, we know that *key* is not in the table. If *key* is inserted at $prb(i + x, key)$, $prd(i,ph(key))$ is set to x.

The advantage of the multiple predictor method is similar to the advantages of double hashing; it eliminates the effects of secondary clustering by dividing the list of elements that hash or rehash into a particular location into np separate and shorter lists.

Simulation results of a slightly modified version of the predictor method using the quadratic rehash method are shown in the following table, which lists the average

number of probes required for a successful search under various load factors, with various numbers of predictor fields and numbers of bits in each predictor. By comparison, recall that quadratic hashing without predictors required 1.44 average probes for a 50 percent load factor and 2.85 for 90 percent, and that double hashing required 1.39 and 2.56 probes, respectively.

Number of predictors	Load factor %	Bits in each predictor		
		3	4	5
1	50	1.25	1.25	1.25
	70	1.39	1.35	1.35
	90	1.83	1.55	1.46
2	50	1.24	1.23	1.23
	70	1.35	1.32	1.31
	90	1.79	1.50	1.41
4	50	1.23	1.23	1.23
	70	1.33	1.30	1.30
	90	1.74	1.47	1.38
8	50	1.22	1.22	1.22
	70	1.32	1.29	1.29
	90	1.72	1.46	1.37

As the number of bits in each predictor and the number of predictors grow very large, the average number of probes required for a successful search with a load factor *lf* becomes $2 - (1 - exp(-lf))/lf$. For a single full-integer predictor, the average number of probes is $1 + lf/2$. The predictor method also reduces the average number of probes for unsuccessful searches.

Coalesced Hashing

Perhaps the simplest use of additional memory to reduce retrieval time is to add a link field to each table entry. This field contains the next position of the table to examine in searching for a particular item. In fact, under this method a rehash function is not required at all; the technique is therefore our first example of the second major method of collision resolution, called ***chaining***. This method uses links rather than a rehash function to resolve hash clashes.

The simplest of the chaining methods is called ***standard coalesced hashing***. A search and insertion algorithm for this method can be presented as follows. Assume that each table entry contains a key field *k* initialized to *NULLKEY* and a *next* field initialized to −1. The algorithm uses an auxiliary function *getempty*, which returns the index of an empty table location.

```
i = h(key);
while (k(i) != key && next(i) >= 0)
   i = next(i);
if (k(i) == key)
   return(i);
```

```
/*  set j to the position where the new  */
/*        record is to be inserted       */
if (k(i) == NULLKEY)
   /*  the hash position is empty  */
   j = i;
else {
   j = getempty();
   next(i) = j;
} /* end if */
k(j) = key;
r(j) = rec;
return(j);
```

The function *getempty* can use any technique to locate an empty position. The simplest method is to use a variable *avail* initialized to *tablesize* − 1 and to execute

```
while (k(avail) != NULLKEY)
   avail--;
return(avail);
```

each time that *getempty* is called. When *getempty* is called, all table positions between *avail* and *tablesize* − 1 have been already allocated. *getempty* sequentially examines positions less than *avail* to locate the first empty position. *avail* is reset to that position, which is then allocated. Of course, it may be desirable to avoid the *nullkey* comparisons in *getempty*. This can be done in a number of ways. An additional one-bit *empty* field can be added to each table position, or the *next* fields can be initialized to − 2 and modified to − 1 when keys are inserted in the table. An alternative solution is to link the free positions together in a list that acts as a stack.

Figure 7.4.3 illustrates a table of ten elements that has been filled using standard coalesced hashing using the hash function *key* % 10. The keys were inserted in the order 14, 29, 34, 28, 42, 39, 84, and 38. Note that items hashing into both 4 and 8 have coalesced into a single list (in positions 4, 8, 7, 5, and 3, containing items 14, 34, 28, 84, and 38). This is how the method gets its name.

There are several advantages to standard coalesced hashing. First, it reduces the average number of probes to approximately exp $(2 * lf)/4 - lf/2 + 0.75$ for an un-

avail =

	k	*next*
0	nullkey	−1
1	nullkey	−1
2	42	−1
3	38	−1
4	14	8
5	84	3
6	39	−1
7	28	5
8	34	7
9	29	6

Figure 7.4.3

successful search, and to approximately (exp (2 * *lf*) − 1)/(8 * *lf*) + *lf*/4 + 0.75 for a successful search. (exp (*x*), available in the standard library *math.h*, computes the value of e^x.) This compares favorably with all previous methods other than the predictor method. A full table requires only an average of 1.8 probes to locate an item, and 2.1 probes to determine that an item is not in the table. A fraction of approximately 1 − *lf*/2 of items in the table can be found on the first probe.

Another major advantage of chaining methods is that they permit efficient deletion without penalizing the efficiency of subsequent retrievals. An item being deleted can be removed from its list, its position in the table freed, and *avail* reset to the following position (unless *avail* already points to a position later in the table). This may slow down the second subsequent insertion somewhat by forcing *avail* to be repeatedly decremented through a long series of occupied positions, but that is not very significant. If the free table positions are kept in a linked list, this penalty also disappears.

A variation of standard coalesced hashing inserts a new element into its chain immediately following the item at its hash location rather than at the end of the chain. This technique, called ***early insertion standard coalesced hashing***, or ***EISCH***, requires the same average number of probes as ordinary standard coalesced hashing for an unsuccessful search, but fewer probes (approximately (*exp* (*lf*) − 1)/*lf*) for a successful search. In a full table, EISCH requires approximately 5 percent fewer probes for a successful search.

A generalization of the standard coalesced hashing method, which we call ***general coalesced hashing***, adds extra positions to the hash table that can be used for list nodes in the case of collisions, but not for initial hash locations. Thus the table would consist of t entries (numbered 0 to $t - 1$), but keys would hash only into one of $m < t$ values (0 to $m - 1$). The extra $t - m$ positions are called the ***cellar*** and are available for storing items whose hash positions are full.

Using a cellar results in less conflict between lists of items with different hash values and therefore reduces the lengths of the lists. However, a cellar that is too large could increase list lengths relative to what they might be if the cellar positions were permitted as hash locations. For full tables, lowest average successful search time is achieved if the ratio *m*/*t* is 0.853 (that is, approximately 15 percent of the table is used as a cellar). In that case the average number of probes is only 1.69. Lowest average unsuccessful search time is attained if the ratio is 0.782, in which case only 1.79 probes are required for the average unsuccessful search. Higher values of *m*/*t* produce lowest successful and unsuccessful search times for lower load factors (when the table is not full).

Unlike the situation with standard coalesced hashing, the early insertion method yields worse retrieval times than if elements are added at the end of the chain in general coalesced hashing. A combination of the two techniques, called ***varied insertion coalesced hashing***, seems to yield the best results. Under this method a colliding element is ordinarily inserted in the list immediately following its hash position, as in the early insertion method, unless the list emanating from that position contains a cellar element. When that occurs, the varied insertion method inserts the collider after the last cellar position in the chain. However, the extra overhead of varied insertion and early insertion often make them less useful in practice.

Separate Chaining

Both rehashing and coalesced hashing assume fixed table sizes determined in advance. If the number of records grows beyond the number of table positions, it is impossible to insert them without allocating a larger table and recomputing the hash values of the keys of all records already in the table using a new hash function. (In general coalesced hashing, the old table can be copied into the first half of the new table and the remaining portion of the new table used to enlarge the cellar, so that items do not have to be rehashed.) To avoid the possibility of running out of room, too many locations may be initially allocated for a hash table, resulting in much wasted space.

Another method of resolving hash clashes is called ***separate chaining*** and involves keeping a distinct linked list for all records whose keys hash into a particular value. Suppose that the hash routine produces values between 0 and *tablesize* − 1. Then an array bucket of header nodes of size *tablesize* is declared. This array is called the ***hash table***. *bucket*[*i*] points to the list of all records whose keys hash into *i*. In searching for a record, the list head *bucket*[*i*] is accessed and the list that it initiates is traversed. If the record is not found, it is inserted at the end of the list. Figure 7.4.4 illustrates separate chaining. We assume a ten-element array and the hash routine *key* % 10. The keys in that figure are presented in the order

75 66 42 192 91 40 49 87 67 16 417 130 372 227

We may write a search and insertion function for separate chaining using a hash function *h*, an array *bucket*, and nodes that contain three fields: *k* for the key, *r* for the record, and *next* as a pointer to the next node in the list. *getnode* is used to allocate a new list node.

```
struct nodetype *search(KEYTYPE key, RECTYPE rec)
{
   struct nodetype *p, *q, *s;

   i = h(key);
   q = NULL;
   p = bucket[i];
   while (p != NULL && p->k != key) {
      q = p;
      p = p->next;
   } /* end while */
   if (p->k == key)
      return(p);
   /*  insert a new record  */
   s = getnode();
   s->k = key;
   s->r = rec;
   s->next = NULL;
   if (q == NULL)
      bucket[i] = s;
   else
      q->next = s;
   return(s);
} /* end search  */
```

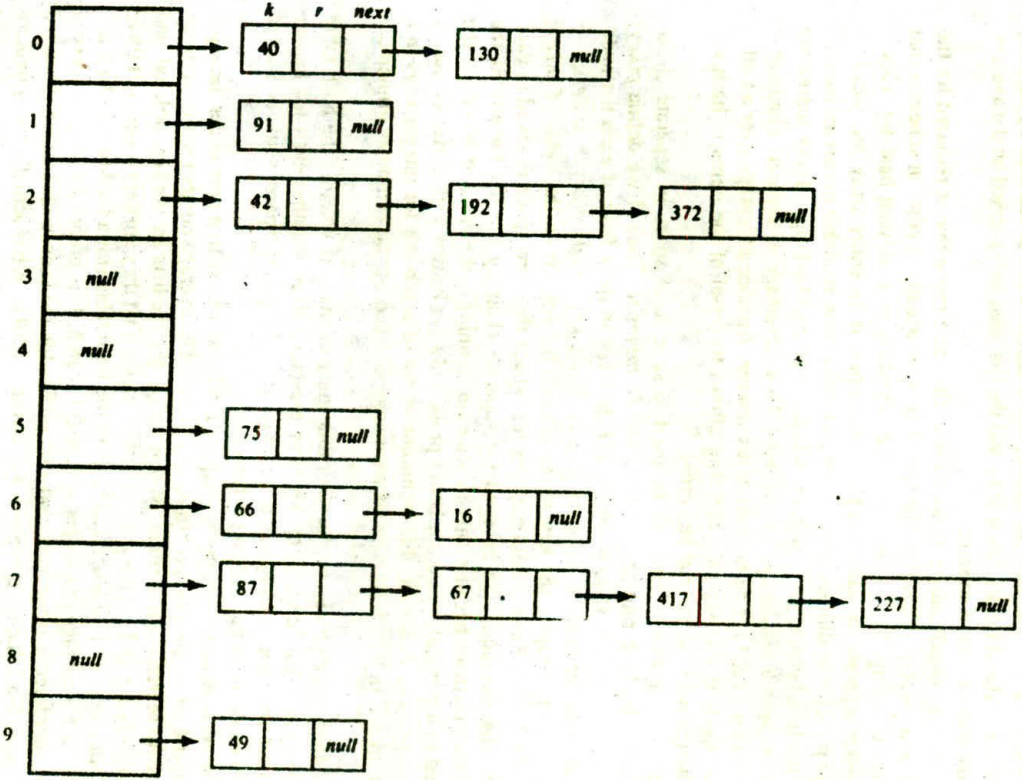

k
r
next
0
40
130
null
1
91
null
2
42
192
372
null
3
null
4
null
5
75
null
6
66
16
null
7
87
67
417
227
null
8
null
9
49
null

Figure 7.4.4

Note that the lists may be reordered dynamically for more efficient searching by the methods of Section 7.1. The time for unsuccessful searches can be reduced by keeping each of the lists ordered by *key*. Then only half the list need be traversed on the average to determine that an item is missing.

The primary disadvantage of chaining is the extra space that is required for the hash table and pointers. However, the initial array is usually smaller in schemes that use separate chaining than in those that use rehashing or coalesced hashing. This is because under separate chaining it is less catastrophic if the entire array becomes full; it is always possible to allocate more nodes and make them available to various lists. Of course, if the lists become very long, the whole purpose of hashing—direct addressing and resultant search efficiency—is defeated. One advantage of separate chaining is that the list items need not be in contiguous storage. Another advantage over all other hashing methods is that separate chaining allows traversal of the items in hash-key order, although not in sequential key order.

There is a technique that can be used to save some space in separate chaining. Consider the hash routine $h(key) = key \% tablesize$. Then if we define $p(key) = key/tablesize$, we can store $p(key)$ rather than *key* in the k field of each list node. When searching for *key*, we compute $h(key)$. Since *key* equals $p(key) * tablesize + h(key)$, we can recompute the key value in each list node from the value of $p(key)$ stored in the k field. Since $p(key)$ requires less space than *key*, space is saved in each list node. The technique can be used for separate chaining but not for any of the other hashing methods because only in separate chaining is the value of $h(key)$ for the key stored in a particular position being probed always known with certainty during the search process. However, this technique is not of much use for integer keys in a language like C in which the size of an integer is fixed by the computer implementation.

Another space-efficiency decision that must be made is whether the hash table should be simply list headers (as we have presented it) or whether each list header should itself also contain a key and a record (or record pointer). Space would be wasted for empty array items but gained for full ones.

The average number of probes required to locate an existing item under the separate chaining technique is approximately $1 + lf/2$. The average number required for an unsuccessful search is approximately $exp(-lf) + lf$ if the lists are not kept ordered, and approximately $1 + lf/2 - (1 - exp(-lf))/lf + exp(-lf)$ if they are kept ordered. If there are *tablesize* keys and *tablesize* elements in the hash table (for a load factor of 1), this translates to 1.5 average probes per successful search, 1.27 probes for an unsuccessful search if unordered lists are used, and 1.05 probes for an unsuccessful search if ordered lists are used. Note that the average for an unsuccessful search is actually lower than for a successful search, since when an array element is empty an unsuccessful search to that element involves zero probes, whereas a successful search requires at least one probe. (This assumes that a key is not kept in each element, since an unsuccessful search to an empty element would then require a key comparison with *NULLKEY*.)

One technique that can be used to reduce the number of probes in separate chaining is to maintain the records that hash into the same value as a binary search tree emanating from the hash bucket rather than as a linked list. However, this requires two pointers to be kept with each record. Since chains are usually small (otherwise, the init-

ial bucket table should be larger), the added space and programming complexity do not seem to be warranted.

Although the average number of probes for separate chaining appears to be quite low, the numbers are deceiving. The reason for this is that the ***load factor*** is defined as the number of keys in the table divided by the number of positions in the table. But, in separate chaining, the number of positions in the hash table is not a valid measure of space utilization, since the keys are not stored in the hash table but in the list nodes. Indeed, the formulas for search time remain valid even if *lf* is greater than 1. For example, if there are five times as many keys as buckets, $lf = 5$, and the average number of probes for a successful search is $1 + lf/2$ or 3.5. Thus the total space allocated to the hash table should be adjusted to include the list nodes. When this is done, coalesced hashing is quite competitive with separate chaining. Note also that rehashing with multiple predictors also performs better than separate chaining.

Hashing in External Storage

If a hash table is maintained in external storage on a disk or some other direct-access device, time rather than space is the critical factor. Most systems have sufficient external storage to allow the luxury of unused allocated space for growth but cannot afford the time needed to perform an I/O operation for every element on a linked list. In such a situation the table in external storage is divided into a number of blocks called ***buckets***. Each bucket consists of a useful physical segment of external storage such as a page or a disk track or track fraction. The buckets are usually contiguous and can be accessed by bucket offsets from 0 to *tablesize* − 1 that serve as hash values, much like indexes of an array in internal storage.

Alternatively, one or more contiguous storage blocks can be used as a hash table containing pointers to buckets distributed noncontiguously. In that situation the hash table is most likely read into memory as soon as the file is opened (or upon the first record read) and remains in memory until the file is closed. When a record is requested, its key is hashed and the hash table (now in internal memory) is used to locate the external storage address of the appropriate bucket. Such a hash table is often called an ***index*** (not to be confused with the term "index" as used to refer to a particular table position).

Each external memory bucket contains room for a moderate number of records (in practical situations, from 10 to 100). An entire bucket is read into memory at once and sequentially searched for the appropriate record. (Of course, a binary search or some other appropriate search mechanism based on the internal organization of the records within the bucket can be used, but the number of records in a bucket is usually small enough that no significant advantage is gained.)

We should note that when dealing with external storage, the computational efficiency of a hash function is not as important as its success at avoiding hash clashes. It is more efficient to spend microseconds computing a complex hash function at internal CPU speeds than milliseconds or longer accessing additional buckets at I/O speeds when a bucket overflows. We also note that external storage space is usually inexpensive. Thus, the number of contiguous initial buckets or the size of the hash table should be chosen so that it is unlikely that any of the buckets become full, even though this entails allocating unused space. Then when a new record must be inserted, there usually is room in the appropriate bucket, and an additional expensive I/O is not required.

If a bucket is full and a record must be inserted, any of the rehash or chaining techniques discussed previously can be used. Of course, additional I/O operations are required when searching for records that are not in the buckets directly corresponding to the hash value. The size of the hash table (which equals the number of buckets that are accessible in one I/O operation) is crucial. A hash table that is too large implies that most buckets will be empty, and a great deal of space is wasted. A hash table that is too small implies that buckets will be full, and large numbers of I/O operations will be required to access many records. If a file is very volatile, growing and shrinking rapidly and unpredictably, this simple hashing technique is inefficient in either space or time. We will see how to deal with this situation shortly.

The following table indicates the expected number of external storage accesses per successful search under linear rehashing, double rehashing, and separate chaining for various bucket sizes and load factors (the load factor is defined as the number of records in the file divided by the product of the number of buckets and the bucket size).

Bucket size	Load factor	Linear rehashing	Double hashing	Separate chaining
1	.5	1.500	1.386	1.250
	.8	3.000	2.012	1.400
	.95	10.5	3.153	1.5
5	.5	1.031	1.028	1.036
	.8	1.289	1.184	1.186
	.95	2.7	1.529	1.3
10	.5	1.005	1.005	1.007
	.8	1.110	1.079	1.115
	.95	1.8	1.292	1.3
50	.5	1.000	1.000	1.000
	.8	1.005	1.005	1.015
	.95	1.1	1.067	1.2

This table indicates that double hashing is the preferred method for moderate or large bucket sizes.

However, when dealing with external storage such as a disk, the number of buckets that have to be read from external storage is not the only determinant of access efficiency. Another important factor is the dispersal of the buckets accessed—that is, how far apart the buckets accessed are from each other. In general, a major factor in the time it takes to read a block from a disk is the ***seek time***. This is the time it takes for the disk head to move to the location of the desired data on the disk. If two buckets accessed one after the other are far apart, more time is required than if they are close together. Given this fact, it would seem that linear rehashing is the most effective technique because although it may require accessing more buckets, the buckets it accesses are contiguous (assuming that $c = 1$ in the linear rehash so that if a record is not in a full bucket, the next sequential bucket is checked). Surprisingly, the table indicates that fewer buckets are accessed under linear rehashing than under separate chaining for large bucket sizes.

If separate chaining is used, it is desirable to reserve an overflow area in each cylinder of the file so that full buckets in that cylinder can link to overflow records in the same cylinder, thus minimizing seek time and essentially eliminating the dispersal penalty. It should be noted that the overflow area need not be organized into buckets and should be organized as individual records with links. In general, few records overflow, and there is only a small chance that sufficiently many will overflow from a single bucket to fill an additional complete bucket. Thus, by keeping individual overflow records, more buckets can overflow into the same cylinder. Since space is reserved within the file for overflow records, the load factor does not represent a true picture of storage utilization for this version of separate chaining. The number of accesses in separate chaining is therefore higher for a given amount of external storage than the numbers in the foregoing table would indicate.

Although double hashing requires fewer accesses than linear rehashing, it disperses the buckets that must be accessed to a degree that may overwhelm this advantage. However, in systems where dispersal is not a factor, double hashing is preferred. This is true of modern large multiuser systems in which many users may be requesting access to a disk simultaneously, and the requests are scheduled by the operating system based on the way the data is arranged on the disk. In such situations, waiting time for disk access is required in any case, so that dispersal is not a significant factor.

The major drawback in using hashing for external file storage is that sequential access (in ascending key order) is not possible, since a good hash function disperses keys without regard to order. Access to records in key-sequential order is particularly important in external file systems.

Separator Method

One technique for reducing access time in external hash tables at the expense of increasing insertion time is attributable to Gonnet and Larson. We will call the technique the ***separator method***. The method uses rehashing (either linear rehashing or double hashing) to resolve collisions but also uses an additional hash routine, s, called the ***signature function***. Given a key key, let $h(key, i)$ be the ith rehash of key, and let $s(key, i)$ be the ith signature of key. If a record with key key is stored in bucket number $h(key, j)$, the ***current signature*** of the record and the key, $sig(key)$, is defined as $s(key, j)$. That is, if a record is placed in a bucket corresponding to its key's jth rehash, its current signature is its key's jth signature.

A separator table, sep, is maintained in internal memory. If b is a bucket number, $sep(b)$ contains a signature value greater than the current signature of every record in $bucket(b)$. To access the record with key key, repeatedly hash key until obtaining a value j such that $sep(h(key, j)) > s(key, j)$. At that point, if the record is in the file it must be in $bucket(h(key, j))$. This ensures the ability to access any record in the file with only a single external memory access.

If m is the number of bits allowed in each item of the separator table (so that it can hold values between 0 and $2^m - 1$), the signature function, s, is restricted to producing values between 0 and $2^m - 2$. Initially, before any overflows have occurred in bucket b, the value of $sep(b)$ is set to $2^m - 1$, so that any record whose key hashes to b can be inserted directly into $bucket(b)$ regardless of its signature. Now, suppose that $bucket(b)$

is full and a new record to be inserted hashes into b. [That is, $h(key, j)$ equals b, and j is the smallest integer such that $sep(h(key, j)) > s(key, j)$.] Then the records in b with the largest current signature *lcs* must be removed from *bucket*(b) to make room for the new record. The new record is then inserted into *bucket*(b) and the old records that had current signature *lcs* and were removed from bucket b are rehashed and relocated into new buckets (with new current signatures, of course). *sep*(b) is then reset to *lcs*, since the current signatures of all records in *bucket*(b) are less than *lcs*. Future keys are directed to *bucket*(b) only if their signatures are less than *lcs*. Notice that more than one record may have to be removed from a bucket if they have equal maximal current signature values. This may leave a bucket with some remaining space after an insertion causes it to overflow.

Records that overflow from a bucket during an insertion may cause cascading overflows in other buckets when attempting to relocate them. This means that an insertion may cause an indefinite number of additional external storage reads and writes. In practice, a limit is placed on the number of such cascading overflows beyond which the insertion fails. If the insertion does fail, it is necessary to restore the file to the status it was in before inserting the new record that caused the original overflow. This is usually done by delaying writing modified buckets to external storage, keeping the modified versions in internal memory until it is determined that the insertion can be completed successfully. If the insertion is aborted because the cascade limit is reached, no writes are done, leaving the file in its original state.

With 40 buckets per record, four-bit signature values and a load factor of 90 percent, an average of no more than two pages need be modified per insertion under this method. However, the number of modified pages per insertion rises rapidly as the load factor is increased, so that the technique is impractical with a load factor greater than 95 percent. Larger signature values and larger bucket sizes permit the method to be used with larger load factors.

Dynamic Hashing and Extendible Hashing

One of the most serious drawbacks of hashing for external storage is that it is insufficiently flexible. Unlike internal data structures, files and databases are semipermanent structures that are not usually created and destroyed within the lifetime of a single program. Further, the contents of an external storage structure tend to grow and shrink unpredictably. All the hash table structuring methods that we have examined have a sharp space/time trade-off. Either the table uses a large amount of space for efficient access, resulting in much wasted space when the structure shrinks, or it uses a small amount of space and accommodates growth very poorly by sharply increasing the access time for overflow elements. We would like to develop a scheme that does not utilize too much extra space when a file is small but permits efficient access when it grows larger. Two such schemes are called ***dynamic hashing***, attributable to Larson, and ***extendible hashing***, attributable to Fagin, Nievergelt, Pippenger, and Strong.

The basic concept under both methods is the same. Initially, m buckets and a hash table (or index) of size m are allocated. Assume that m equals 2^b, and assume a hash routine h that produces hash values that are $w > b$ bits in length. Let $hb(key)$ be

the integer between 0 and m represented by the first b bits of $h(key)$. Then, initially, hb is used as the hash routine, and records are inserted into the m buckets as in ordinary external storage hashing.

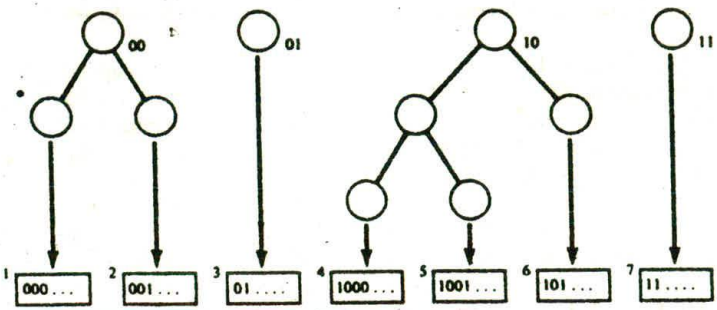
When a bucket overflows, the bucket is split in two and its records are assigned to the two new buckets based on the $(b + 1)$st bit of $h(key)$. If the bit is 0, the record is assigned to the first (or left) new bucket; if the bit is 1, the record is assigned to the second (or right) bucket. (Of course, the original bucket can be reused as one of the two new buckets.) The records in each of the two new buckets now all have the same first $b + 1$ bits in their hash keys, $h(key)$. Similarly, when a bucket representing i bits overflows (where $b <= i <= w$), the bucket is split and the $(i + 1)$st bit of $h(key)$ for each record in the bucket is used to place the record in the left or right new bucket. Both new buckets then represent $i + 1$ bits of the hash key. We call the bucket whose keys have 0 in their $(i + 1)$st bit the ***0-bucket*** and the other bucket the ***1-bucket***.

Dynamic hashing and extendible hashing differ as to how the index is modified when a bucket splits. Under dynamic hashing, each of the m original index entries represents the root of a binary tree each of whose leaves contains a pointer to a bucket. Initially each tree consists of only one node (a leaf node) that points to one of the m initially allocated buckets. When a bucket splits, two new leaf nodes are created to point to the two new buckets. The former leaf that had pointed to the bucket being split is transformed into a nonleaf node whose left son is the leaf pointing to the 0-bucket and whose right son is the leaf pointing to the 1-bucket. Dynamic hashing with $b = 2$ $(m = 4)$ is illustrated in Figure 7.4.5.

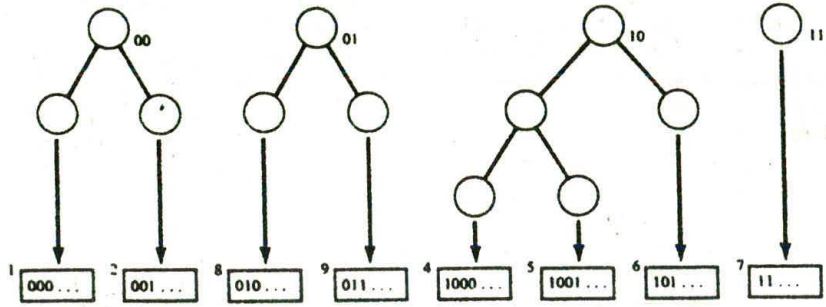
To locate a record under dynamic hashing, compute $h(key)$ and use the first b bits to locate a root node in the original index. Then use each successive bit of $h(key)$ to move down the tree, going left if the bit is 0 and right if the bit is 1, until a leaf is reached. Then use the pointer in the leaf to locate the bucket that contains the desired record, if it exists.

In extendible hashing, each bucket contains an indication of the number of bits of $h(key)$ that determine which records are in that bucket. This number is called the ***bucket depth***. Initially, this number is b for all bucket entries; it is increased by 1 each time a bucket splits. Associated with the index is the ***index depth***, d, which is the maximum of all the bucket depths. The size of the index is always 2^d (initially, 2^b).

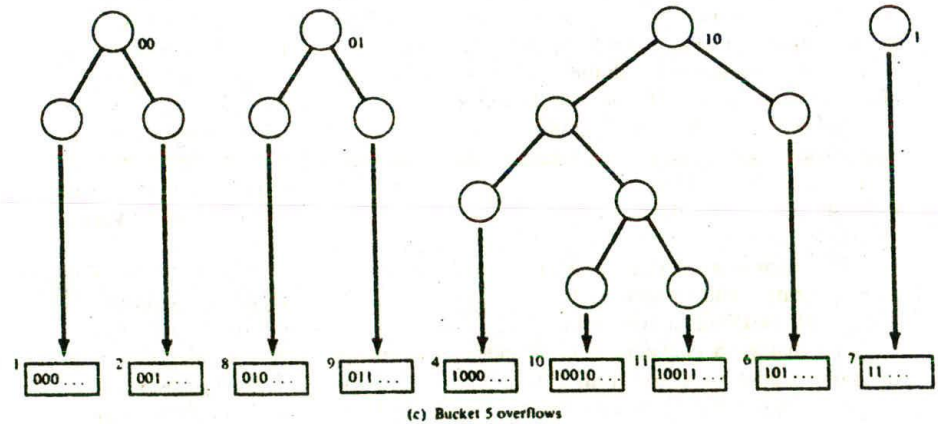
Suppose that a bucket of depth i is to be split. Let $a1, a2, \ldots, ai$ (where each aj is either 0 or 1) be the first i bits of $h(key)$ for the records in the bucket being split. There are two cases to consider: $i < d$ and $i = d$. If $i < d$ (so that the bucket depth is being increased to $i + 1$ but the index depth remains at d), all index positions with bit values $a1, a2, \ldots, ai00 \ldots 0$ (up to a bit size of d) through $a1, a2, \ldots, ai01 \ldots 1$ of the index (that is, all positions starting with $a1, \ldots, ai0$) are reset to point to the 0-bucket, and index positions with bit values $a1, a2, \ldots, ai10 \ldots 0$ through $a1, a2, \ldots, ai11, \ldots, 1$ (that is, all positions starting with $a1 \ldots ai1$) are reset to point to the 1-bucket. If $i = d$ (so that the bucket depth and the index depth are both being increased to $d + 1$), the index is doubled in size from 2^d to 2^{d+1}; the old contents of all index positions $x1 \ldots xd$ are copied into the new positions $x1 \ldots xd0$ and $x1 \ldots xd1$; the contents of index position $a1 \ldots ad0$ is set to point to the new 0-bucket, and the contents of index position $a1 \ldots ad1$ to point to the new 1-bucket. Extendible hashing is illustrated in Figure 7.4.6. Figure 7.4.6a illustrates a configuration with index depth 4. Figure 7.4.6b illustrates an



(a) A dynamic hashing configuration



(b) Bucket 3 overflows



(c) Bucket 5 overflows

Figure 7.4.5 Dynamic hashing with $b = 2$.

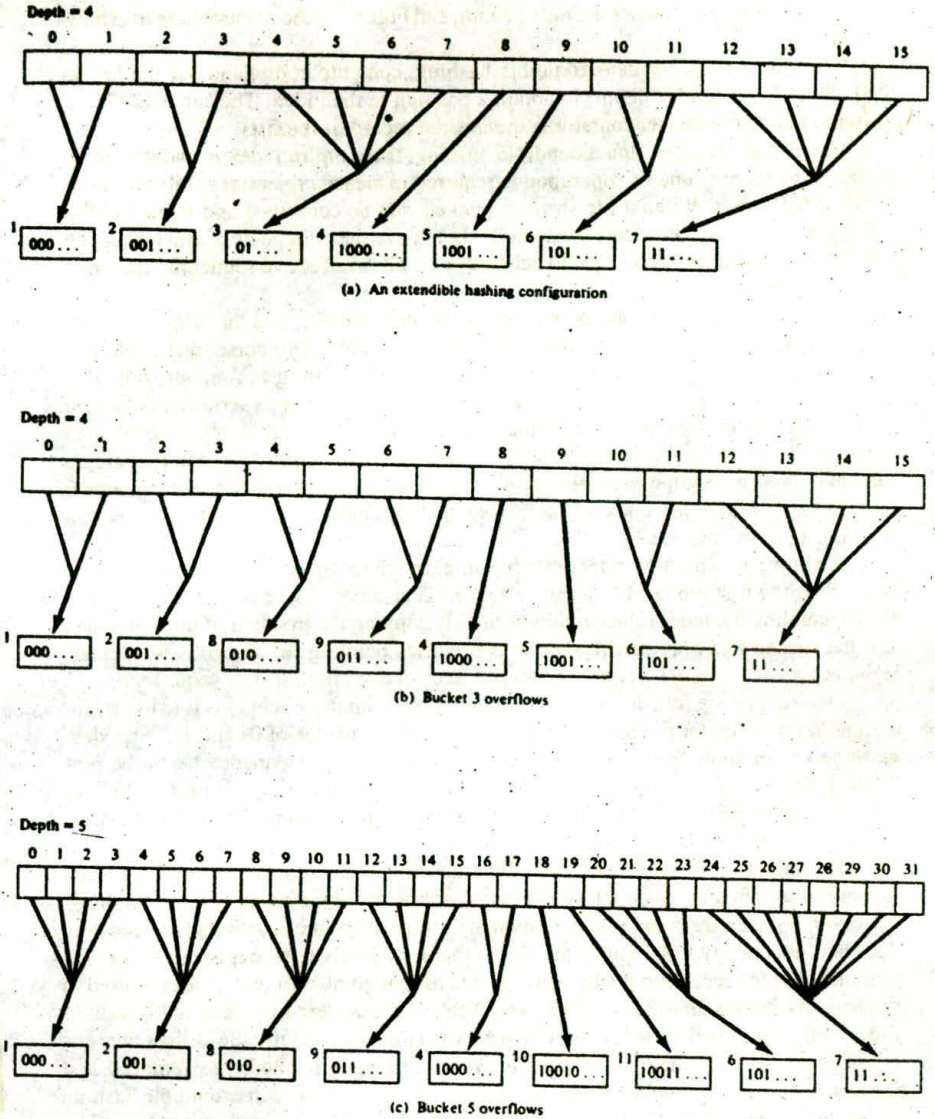


Figure 7.4.6 Extendible hashing.

overflow that does not increase the index depth, and Figure 7.4.6c illustrates an overflow that does.

To locate a record under extendible hashing, compute $h(key)$ and use the first d bits (where d is the index depth) to obtain a position in the index. The contents of this position point to the bucket containing the desired record, if it exists.

Under both dynamic and extendible hashing, if the entire index is maintained in internal storage, only one I/O operation is required to locate a record regardless of how large the file grows. When a file shrinks, buckets can be combined and freed and the index size can be reduced. Thus these methods achieve the twin goals of efficient space utilization and efficient access. Both schemes also allow effective sequential traversal of records in hash-key order.

However, neither method permits traversal in key order, and this often prevents the practical use of the techniques for file implementation. Of course, one could use the key itself as a hash value or some other order-preserving hash function, but such functions are usually nonuniform. The lack of uniformity is not as serious an obstacle under these methods as under static hashing methods, since any number of bits can be used. The more bits used, the less likely that two keys clash. Although too large a number of bits can result in too large an index for practical use, dynamic hashing, which does not use as large an index as extendible hashing, may indeed be practical with a nonuniform hash function.

A simple variation of extendible hashing, in which the last bits of the hash key rather than the first are used to locate a bucket, is also possible. Such a variation simplifies doubling the index, since it allows merely copying the first half of the new index into the second and only modifying the two entries pointing to the two new buckets. However, such a scheme would not permit traversal even in hash-key sequence.

One suggestion for a hashing technique for use with these methods is to use a random number generator to produce an arbitrarily long sequence of 0s and 1s as needed, with the key or some function thereof as the seed. The same sequence would be produced for the same key every time, but there is no limit to the extendibility. This has the advantage of allowing the file to grow arbitrarily large, and in the case of dynamic hashing, to ensure balanced trees.

In comparing dynamic and extendible hashing, we note that extendible hashing is more time efficient, since a tree path need not be traversed as in dynamic hashing. However, if the entire index is kept in memory, the time spent in traversing the tree path does not involve any I/Os. Traversal time is therefore likely to be negligible compared with the time for accessing the bucket. The maximum number of tree nodes required in dynamic hashing is $2n - 1$, assuming n buckets, whereas there may be as many as 2^{n-1} index entries required under extendible hashing. However, usually fewer than twice as many extendible hashing index entries as dynamic hashing tree nodes are required, and the tree nodes require two pointer fields compared with one for each extendible hashing index entry. Thus the two methods are comparable in average internal space utilization.

It is also possible to compress very large extendible hashing indexes by keeping only one copy of each bucket pointer and maintaining from/to indicators. Another point to note is that extendible hashing performs the same way regardless of the value of m, the initial number of index entries, whereas dynamic hashing requires longer tree paths

if m is smaller. In fact, there is no reason not to initialize m to 0 (that is, $b = 0$) with a single empty bucket in extendible hashing, other than for contiguity of the buckets in external storage.

External storage utilization of both dynamic and balanced hashing is approximately 69 percent on the average, which is the same as is achieved with B-trees. However, the storage utilization of the hashing methods oscillates far more sharply and persists longer than for B-trees, so that there is a period of low utilization (approximately 50 percent) after buckets are split as they begin filling up. This is followed by a period of high utilization (approaching 90–100 percent) as new records are uniformly distributed into the buckets and they all become full more or less simultaneously. Finally, a short period of intensive splitting is observed, after which utilization is again low.

The reason for this oscillation is that the hash routine is expected to be uniform so that all buckets are filled at approximately the same time. It may be desirable to minimize this oscillation by purposely introducing some nonuniformity in the hash function. However, if this is done in extendible hashing, the nonuniformity could cause extremely large indexes. This problem can be solved by keeping a compressed version of the index as noted.

To achieve storage utilization higher than 69 percent, it is possible to utilize overflow buckets. When a bucket becomes full and an additional record is to be inserted, an overflow bucket is allocated and linked to the full bucket. When the overflow bucket also fills up, both buckets' contents are redistributed into two nonoverflow buckets, with the overflow bucket linked to the new bucket with more than half of the records. This results in more than two full buckets of data distributed into three buckets (two regular, one overflow), yielding a minimum utilization of 67 percent, rather than 50 percent, once the initial buckets are filled. Of course, it is also possible to use overflow buckets that are smaller than regular buckets (in which case not every split results in an overflow bucket remaining allocated), as well as to allow an overflow bucket to contain records that have overflowed from more than one bucket. This latter technique, however, complicates traversal in hash-key order.

Another method of increasing space utilization is to allow overflow records to be placed in a full bucket's brother (under extendible hashing, the brother of a 0 bucket is its corresponding 1 bucket and vice versa). This method also complicates traversal in hash-key order. Both the use of overflow buckets and the use of a brother bucket for overflow records increase the average number of accesses required for a search. However, if brother buckets are kept contiguous, this penalty may not be great.

Linear Hashing

One drawback of dynamic and extendible hashing is the need for an index. Although the index may be kept in internal storage once the file is opened, this is not always possible if the index becomes very large. Also, the index does require external storage when the file is not in use. In addition, the external copy of the index may have to be constantly updated to guard against power failure or other interruption that would prevent rewriting the index when the file is closed.

Another technique, ***linear hashing*** (not to be confused with linear rehashing), proposed by Litwin and modified by Larson, permits a hash table to expand and shrink dynamically without requiring an index. However, the basic technique does require the use of overflow buckets, unlike dynamic and extendible hashing. The version that we present is called ***linear hashing with two partial expansions,*** or ***LH2P.***

Under LH2P the initial file consists of m buckets, where m is even, numbered 0 to $m - 1$. The file is considered divided into ng groups of buckets, numbered 0 to $ng - 1$. Initially, there are $m/2$ groups of buckets ($ng = m/2$), each consisting of two buckets. Group i initially consists of buckets i and $i + ng$. For example, if m equals 6, the file initially contains six buckets, numbered 0 to 5. There are three groups: group 0 contains buckets 0 and 3, group 1 contains buckets 1 and 4, and group 2 contains buckets 2 and 5.

The file grows in two ways: overflow growth and regular growth. Unlike B-trees, dynamic hashing, or extendible hashing, a bucket is not split when it overflows. Instead, an overflow mechanism is used to contain the overflowing records from a particular bucket. This mechanism can utilize any of the techniques discussed earlier.

Regular growth takes place by expanding the size of the file by one bucket at a time. Expansion of the file by one bucket is called a ***simple expansion.*** A simple expansion takes place whenever the load factor (defined as the total number of records in the file divided by the number of records that fit into regular, nonoverflow buckets) exceeds a threshold percentage. When a simple expansion takes place, the number of regular buckets in the file, *nb*, increases by 1. At any time, the file consists of buckets 0 through $nb - 1$ plus any overflow buckets or records. Initially, *nb* equals m.

Regular growth under LH2P takes place in a series of simple expansions, grouped into ***partial expansions*** and ***full expansions.*** Each full expansion doubles the number of regular buckets in the file and consists of two partial expansions. The first partial expansion increases the number of regular buckets by 50 percent, and the second increases the number by the same amount. Thus, after the first partial expansion, *nb* equals $3 * m/2$; after the first full expansion, *nb* equals $2 * m$; and after the second full expansion, *nb* equals $4 * m$.

Each simple expansion increases the size of a particular group of buckets by one bucket. The variable *nextgroup* always holds the number of the next group to be expanded (initially, *nextgroup* is 0). During the first partial expansion, two-bucket groups are expanded to three buckets by moving some records from buckets *nextgroup* and *nextgroup* + *ng* (and some of their associated overflow records) into bucket *nextgroup* $+2 * ng$. (Exactly which records are moved to the new bucket and which remain in place will be discussed shortly.) Note that the buckets are numbered from 0 to $nb - 1$, so that *nb* is always the number of the next bucket added to the file. During the first partial expansion *nb* always equals *nextgroup* $+2 * ng$. (Initially, *nextgroup* = 0 and $ng = m/2$, so $nb = m$.) After a group has been expanded, *nextgroup* and *nb* are both increased by 1, and the next group is ready for expansion.

After the first partial expansion, all *ng* groups have been expanded and *nextgroup* is reset to 0. Each group now contains three regular buckets rather than two, and the file size (in number of buckets) has grown by 50 percent.

During the second expansion, *nb* always equals *nextgroup* $+3 * ng$. (At the start of a second partial expansion, *nextgroup* is 0 and *nb* equals $3 * ng$.) Three-bucket groups are expanded to four by moving some records from buckets *nextgroup*, *nextgroup* + *ng*, and *nextgroup* $+2 * ng$ (and some of their associated overflow records) into bucket *nextgroup* $+3 * ng$ (which equals *nb* during the second partial expansion). Any overflow records not moved to bucket *nb* during an expansion are moved back into their home bucket, if there is room.

After the second partial expansion, the file size has doubled and a full expansion has taken place. In preparation for the next full expansion, the number of groups (*ng*) is doubled and the size of each groups is halved, from four to two. That is, group *i*, consisting of buckets $i, i + j, i + 2 * j$, and $i + 3 * j$ (where *j* is the old value of *ng*), is now viewed as two separate groups: group *i* consisting of buckets *i* and $i + 2 * j$ and group $i + j$ constisting of buckets $i + j$ and $i + 3 * j$. The first partial expansion of the next full expansion then begins.

Note that the group that is expanded is always the next sequential group and is independent of whether or not overflow has taken place in that group. Overflow is handled by a separate mechanism from expansion.

A key question is how a hash function is used to access a record directly. When the file contains *nb* buckets, the hash function must produce a value between 0 and $nb - 1$, but when the file size is increased by one bucket, it must produce a value between 0 and *nb*. Further, a record moved from bucket *i* to bucket *j* must have previously hashed into *i* and must henceforth hash into *j*. The hash function must also be used to determine whether or not a record should be moved during an expansion. The following method allows direct access to a record's bucket. Although it does involve many CPU operations, it only requires one I/O operation to obtain the appropriate bucket from a nonoverflow record.

A function $h1(key)$ that produces values in the range of 0 to $m - 1$ is used as a direct hashing function initially, before any expansions have taken place. The value $h1(key)$ is called the ***initial hash*** of *key*. We also assume a function $h2(key, i)$ for $i > 0$ that uniformly produces values in the range 1 to 4. The value of $h2(key, i)$ determines whether the record with key *key* is moved during the *i*th full expansion and, if so, whether it is moved to the first or the second expansion bucket of its group. If $h2(key, i)$ is 1 or 2, the record is not moved during the *i*th full expansion; if $h2(key, i)$ is 3, it is moved to the first expansion bucket of its group in the *i*th full expansion; if $h2(key, i)$ is 4, it is moved to the second expansion bucket. Thus a random record has a 50 percent chance of being moved during a full expansion and a 25 percent chance of being moved to either of the two expansion buckets.

During the first partial expansion of the *i*th full expansion, the hashing algorithm examines the values $h2(key, i), h2(key, i + 1), h2(key, i + 2)$, and so on, seeking the first value less than 4. If that value is 1 or 2, the record is not moved; if it is 3, it is moved to the single expansion bucket. It will stay there after the second partial expansion if $h(key, i)$ is 3; it will be moved to the second partial expansion bucket by the second partial expansion if $h2(key, i)$ is 4. Thus a random record has a one-third chance of being moved to the third bucket of a group in a first partial expansion and a one-quarter chance of being moved to the fourth bucket in a second partial expansion. This guarantees that the records are distributed uniformly throughout the file.

Define the ***level*** of an LH2P file as the number of full expansions that have taken place, and let the variable *level* contain the level of the file. If *pe* is the number of the current partial expansion (either 1 or 2), *pe* + 1 is the number of buckets in a group that has not yet been expanded in the current partial expansion, and *pe* + 2 is the number of buckets in a group that has been expanded. At all times,

$$ng = m * 2^{level-1}$$

and

$$nb = nextgroup + (pe + 1) * ng;$$

The values *m*, *level*, *nextgroup*, and *pe* therefore define the state of an LH2P file and must be kept with the file at all times.

To locate the address of a record, it is necessary to follow its relocations through the *level* full expansions, the *pe* − 1 completed partial expansions, and the *nextgroup* completed simple expansions of the current partial expansion. The hash of a key, *h*(*key*), is therefore computed by the following algorithm:

```
h = h1(key);              /* initial hash */
numgroups = m/2;          /* initial number of groups */
for (i = 1; i < level; i++) {
   /* trace the movement of the record */
   /*   through level full expansions  */
   bucketnum = h2(key, i);
   if (bucketnum > 2) {
      /* record was moved in ith full expansion */
      groupnum = h % numgroups;
      h = groupnum + (bucketnum - 1) * numgroups;
   } /* end if */
   numgroups = 2 * numgroups;   /* number of groups after */
                                /*   i full expansions    */
} /* end for */
/* At this point h holds the key's hash after level */
/*    full expansions and no partial expansions.    */
/* Now, compute the record's current location after */
/* possible movement in the current full expansion. */
groupnum = h % numgroups;        /* current group number */
i = level + 1;
bucketnum = h2(key, i);          /* eventual bucket number */
                                 /* at end of current full */
                                 /*       expansion        */
/* compute the size of the current group */
/*  pe holds the number of the current   */
/*          partial expansion            */
if (groupnum < nextgroup)
   groupsize = pe + 2;
```

```
else
   groupsize = pe + 1;
/* if the eventual bucket number is larger than */
/*  current group size, then continue hashing   */
/*   until the current bucket number is found   */
while (bucketnum > groupsize) {
   i++;
   bucketnum = h2(key, i);
} /* end while */
if (bucketnum > 2)
   /* the record was already moved  */
   /* in the current full expansion */
   h = groupnum + (bucketnum - 1) * numgroups;
```

The following example illustrates LH2P. Consider an LH2P file that initially contains six buckets, 0 through 5, consisting of three groups: (0, 3), (1, 4), and (2, 5). In this case, m is 6 and nb is initially 6. Consider six records $r0$ through $r5$ with keys $k0$ through $k5$ that initially hash into 0 through 5, respectively (that is, $h1(ki) = i$). Then $r0$ through $r5$ are placed in buckets 0 through 5 initially.

Assume that the following are the values of $h2(key, i)$ for key equal to $k0$ through $k5$ and i from 1 to 4:

Key	$h2(key, 1)$	$h2(key, 2)$	$h2(key, 3)$	$h2(key, 4)$
$k0$	1	4	2	3
$k1$	4	2	1	2
$k2$	3	1	2	1
$k3$	4	4	3	1
$k4$	1	2	1	2
$k5$	2	4	4	3

The following table illustrates the rearrangement of records during expansion of this LH2P file. Each simple expansion is specified in the first column by a status triple consisting of the number of full expansions that have taken place (*level*), the number of the current partial expansion(*pe*), and the number of the group currently being expanded (*nextgroup*). Initially, there are three groups ($ng = 3$), six buckets ($nb = 6$), and two buckets per group. The second column shows the existing buckets of the current group in parentheses, followed by the bucket being added to the group in the current expansion step. We use the notation bi to indicate bucket i. Below the buckets of the group in each second-column entry are the records contained in that group. The third column indicates the results of the simple expansion. Of course, for the expansions to take place, additional records must be added so that the load factor exceeds the threshold. However, we do not illustrate these other records here but merely illustrate how existing records move into expansion buckets.

Status	Group and records	Result of expansion
(0, 1, 0)	(*b*0, *b*3); *b*6	*h*2(*k*0, 1) = 1, so *r*0 remains in *b*0.
	(*r*0, *r*3)	*h*2(*k*3, 1) = 4; *h*2(*r*3, 2) = 4; *h*2(*k*3, 3) = 3, so *r*3 moves to *b*6
(0, 1, 1)	(*b*1, *b*4); *b*7	*h*2(*k*1, 1) = 4; *h*2(*k*1, 2) = 2, so *r*1 remains in *b*1. *h*2(*k*4, 1) = 1, so *r*4 remains in *b*4.
(0, 1, 2)	(*b*2, *b*5); *b*8	*h*2(*k*2, 1) = 3, so *r*2 moves to *b*8.
	(42, *r*5)	*h*2(*k*5, 1) = 2, so *r*5 remains in *b*5.

This ends the first partial expansion. There are still three groups, but each now contains three buckets, so $nb = 9$. The second partial expansion then begins:

Status	Group and records	Result of expansion
(0, 1, 0)	(*b*0, *b*3, *b*6); *b*9	*h*2(*k*0, 1) = 1, so *r*0 remains in *b*0.
	(*r*0, *r*3)	*h*2(*k*3, 1) = 4, so *r*3 moves to *b*9.
(0, 1, 1)	(*b*1, *b*4, *b*7); *b*10	*h*2(*k*1, 1) = 4, so *r*1 moves to *b*10.
	(*r*1, *r*4)	*h*2(*k*4, 1) = 1, so *r*4 remains in *b*4.
(0, 1, 2)	(*b*2, *b*5, *b*8); *b*11	*h*2(*k*2, 1) = 3, so *r*2 remains in *b*8.
	(*r*2, *r*5)	*h*2(*k*5, 1) = 1, so *r*5 remains in *b*5.

This ends the first full expansion. There are now three groups, and each contains four buckets, so $nb = 12$. To start the second full expansion, the number of groups is doubled ($ng = 6$), and the number of buckets in each is reset to 2. The second full expansion proceeds:

Status	Group and records	Result of expansion
(1, 1, 0)	(*b*0, *b*6); *b*12	*h*2(*k*0, 2) = 4; *h*2(*k*0, 3) = 2,
	(*r*0)	so *r*0 remains in *b*0.
(1, 1, 1)	(*b*1, *b*7,); *b*13	No records in this group.
(1, 1, 2)	(*b*2, *b*8); *b*14	*h*2(*k*2, 2) = 4, so *r*2 remains in *b*8.
	(*r*2)	
(1, 1, 3)	(*b*4, *b*10); *b*15	*h*2(*k*3, 2) = 4; *h*2(*k*3, 3) = 3,
	(*r*3)	so *r*3 moves to *b*15.
(1, 1, 4)	(*b*4, *b*10); *b*16	*h*2(*k*1, 2) = 2, so *r*1 remains in *b*10.
	(*r*1, *r*4)	*h*2(*k*4, 2) = 2, so *r*4 remains in *b*4.
(1, 1, 5)	(*b*5, *b*11); *b*17	*h*2(*k*5, 2) = 4; *h*2(*k*5, 3) = 4; *h*2(*k*5, 4) = 3,
	(*r*5)	so *r*5 moves to *b*17.

This ends the first partial expansion.

Status	Group and records	Result of expansion
(1, 2, 0)	(*b*0,*b*6,*b*12); *b*18 (*r*0)	*h*2(*k*0, 2) = 4, so *r*0 moves to *b*18.
(1, 2, 1)	(*b*1,*b*7,*b*13); *b*19	No records in this group.
(1, 2, 2)	(*b*2,*b*8,*b*14); *b*20 (*r*2)	*h*2(*k*2, 2) = 1, so *r*1 remains in *b*8.
(1, 2, 3)	(*b*3,*b*9,*b*15); *b*21 (*r*3)	*h*2(*k*3, 2) = 4, so *r*3 moves to *b*21.
(1, 2, 4)	(*b*4,*b*10,*b*16); *b*22 (*b*1,*b*4)	*h*2(*k*1, 2) = 2, so *r*1 remains in *b*10. *h*2(*k*4, 2) = 2, so *r*4 remains in *b*4.
(1, 2, 5)	(*b*5,*b*11,*b*17); *b*23 (*r*5)	*h*2(*k*5, 2) = 4, so *r*5 moves to *b*23.

This ends the second full expansion.

The techniques of LH2P can be generalized to allow n partial expansions in each full expansion. Such a scheme is called LHnP. Each partial expansion increases the file size by the fraction $1/n$. Although higher values of n reduce the average number of overflow records (since records hashing to a particular bucket are redistributed more frequently) and therefore the number of accesses for both search and insertion, more partial expansions require more frequent allocations of storage and more complex hash value computations. Thus, practical insertion costs and expansion costs are higher. The value $n = 2$, leading to the scheme LH2P, is a practical compromise.

Overflow can be handled by a variety of methods under linear hashing. Use of overflow buckets is the simplest technique but may require varying-sized buckets for efficiency. Such variation would complicate storage management. Ramamohanarao and Sacks–Davis suggest recursive linear hashing in which records that overflow the prime area are placed in a second linear hashing file, records that overflow that area are placed in a third, and so on. More than three areas are rarely needed.

There are several techniques that eliminate the need for separate, dedicated overflow areas. Mullin suggests using chaining within the linear hashing file, in which the most recently expanded group is used to contain overflow records (since that group is most likely to have empty space). Larson suggests that every kth bucket in the primary area be reserved for overflow records, with the hash function suitably modified to avoid the overflow buckets.

Larson also suggests the possibility of using linear rehashing to locate overflow records. When linear rehashing is used, it is more efficient to implement each partial expansion in several sweeps of step size $s > 1$ and to go backward among the groups $ng - 1, ng - 1 - s, ng - 1 - 2 * s$, and so on; the second step would expand groups $ng - 2, ng - 2 - s, ng - 2 - 2 * s$, and so on. Linear rehashing can also be combined with the separator method of Gonnet and Larson to allow one-access retrieval and eliminate the overhead of overflow.

Choosing a Hash Function

Let us now turn to the question of how to choose a good hash function. Clearly, the function should produce as few hash clashes as possible; that is, it should spread the

keys uniformly over the possible array indices. Of course, unless the keys are known in advance, it cannot be determined whether a particular hash function disperses them properly. However, although it is rare to know the keys before selecting a hash function, it is fairly common to know some properties of the keys that affect their dispersal.

In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or strewn throughout the key) hash into different values. Thus a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digit or character permutations of each other (such as 139 and 319 or *meal* and *lame*), they should also hash into different values. The reason for this is that key sets frequently have clusters or permutations that might otherwise result in collisions.

For example, the most common hash function (which we have used in the examples of this section) uses the ***division*** method, in which an integer key is divided by the table size and the remainder is taken as the hash value. This is the hash function *h(key) = key % tablesize*. Suppose, however, that *tablesize* equals 1000 and that all the keys end in the same three digits (for example, the last three digits of a part number might represent a plant number, and the program is being written for that plant). Then the remainder on dividing by 1000 yields the same value for all the keys, so that a hash clash occurs for each record except the first. Clearly, given such a collection of keys, a different hash function should be used.

It has been found that the best results with the division method are achieved when *tablesize* is prime (that is, it is not divisible by any positive integer other than 1 and itself). However, even if *tablesize* is prime, an additional restriction is called for. If r is the number of possible character codes on a particular computer (assuming an 8-bit byte, r is 256), and if *tablesize* is a prime such that *r % tablesize* equals 1, the hash function *key % tablesize* is simply the sum of the binary representation of the characters in the key modulo *tablesize*. For example, suppose that r equals 256 and that *tablesize* equals 17, in which case *r % tablesize* = 1. Then the key 37956, which equals 148 * 256 + 68 (so that the first byte of its representation is 148 and the second byte is 68), hashes into 37956 % 17, which equals 12, which equals (148 + 68) % 17. Thus two keys that are simply character permutations (such as *steam* and *mates*) will hash into the same value. This may promote collisions and should be avoided. Similar problems occur if *tablesize* is chosen so that r^k % *tablesize* is very small or very close to *tablesize* for some small value of k.

Another hash method is the *multiplicative method*. In this method a real number c between 0 and 1 is selected. *h(key)* is defined as *floor(m*frac(c*key))*, where the function *floor(x)*, available in the standard library *math.h*, yields the integer part of the real number x, and *frac(x)* yields the fractional part. [Note that *frac(x) = x − floor(x).*] That is, multiply the key by a real number between 0 and 1, take the fractional part of the product yielding a random number between 0 and 1 dependent on every bit of the key, and multiply by m to yield an index between 0 and $m - 1$. If the word size of the computer is b bits, c should be chosen so that $2^b * c$ is an integer relatively prime to 2^b, and c should not be too close to either 0 or 1. Also if r, as before, is the number of possible character codes, avoid values of c such that *frac*$((r^k) * c)$ is too close to 0 or 1 for some small value of k (these values yield similar hashes for keys with the same last

k characters) and of values c of the form $i/(r-1)$ or $i/(r^2-1)$ (these values yield similar hashes for keys that are character permutations). Values of c that yield good theoretical properties are 0.6180339887 [which equals $(sqrt(5)-1)/2$] or 0.3819660113 [which equals $1-(sqrt(5)-1)/2$]. If m is chosen as a power of 2 such as 2^p, the computation of $h(key)$ can be done quite efficiently by multiplying the one-word integer key by the one-word integer $c * 2^b$ to yield a two-word product. The integer represented by the most significant p bits of the integer in the second word of this product is then used as the value of $h(key)$.

In another hash function, known as the ***midsquare method,*** the key is multiplied by itself and the middle few digits (the exact number depends on the number of digits allowed in the index) of the square are used as the index. If the square is considered as a decimal number, the table size must be a power of 10, whereas if it is considered as a binary number, the table size must be a power of 2. Alternatively, the number represented by the middle digits can be divided by the table size and the remainder used as the hash value. Unfortunately, the midsquare method does not yield uniform hash values and does not perform as well as the previous two techniques.

The ***folding method*** breaks up a key into several segments that are added or exclusive *or*ed together to form a hash value. For example, suppose that the internal bit string representation of a key is 010111001010110 and that 5 bits are allowed in the index. The three bit strings 01011, 10010, and 10110 are *exclusive or*ed to produce 01111, which is 15 as a binary integer. (The ***exclusive or*** of two bits is 1 if the two bits are different, and 0 if they are the same. It is the same as the binary sum of the bits, ignoring the carry.) The disadvantage of the folding method is that two keys that are k-bit permutations of each other (that is, where both keys consist of the same groups of k bits in a different order) hash into the same k-bit value. Still another technique is to apply a multiplicative hash function to each segment individually before folding.

There are many other hash functions, each with its own advantages and disadvantages depending on the set of keys to be hashed. One consideration in choosing a hash function is efficiency of calculation; it does no good to be able to find an object on the first try if that try takes longer than several tries in an alternative method.

If the keys are not integers, they must be converted into integers before applying one of the foregoing hash functions. There are several ways to do this. For example, for a character string the internal bit representation of each character can be interpreted as a binary number. One disadvantage of this is that the bit representations of all the letters or digits tend to be very similar on most computers. If the keys consist of letters alone, the index of each letter in the alphabet can be used to create an integer. Thus the first letter of the alphabet (a) is represented by the digits 01 and the fourteenth (n) is represented by the digits 14. The key 'hello' is represented by the integer 0805121215. Once an integer representation of a character string exists, the folding method can be used to reduce it to manageable size. However, here too, every other digit is a 0, 1, or 2, which may result in nonuniform hashes. Another possibility is to view each letter as a digit in base-26 notation so that 'hello' is viewed as the integer $8*26^4+5*26^3+12*26^2+12*26+15$.

One of the drawbacks of all these hash functions is that they are not order preserving; that is, the hash values of the two keys are not necessarily in the same order as the keys themselves. It is therefore not possible to traverse the hash table in sequential order by key. An example of a hash function that is order preserving is $h(key) = key/c$,

where c is some constant chosen so that the highest possible key divided by c equals *tablesize* − 1. Unfortunately, order-preserving hash functions usually are severely nonuniform, leading to many hash clashes and a larger average number of probes to access an element. Note also that to enable sequential access to keys, the separate chaining method of resolving collisions must be used.

Perfect Hash Functions

Given a set of keys $K = \{k1,k2,\ldots,kn\}$, a ***perfect hash function*** is a hash function h such that $h(ki) != h(kj)$ for all distinct i and j. That is, no hash clashes occur under a perfect hash function. In general, it is difficult to find a perfect hash function for a particular set of keys. Further, once a few more keys are added to the set for which a perfect hash function has been found, the hash function generally ceases to be perfect for the expanded set. Thus, although it is desirable to find a perfect hash function to ensure immediate retrieval, it is not practical to do so unless the set of keys is static and is frequently searched. The most obvious example of such a situation is a compiler in which the set of reserved words of the programming language being compiled does not change and must be accessed repeatedly. In such a situation, the effort required to find a perfect hashing function is worthwhile because, once the function is determined, it can save a great deal of time in repeated applications.

Of course, the larger the hash table, the easier it is to find a perfect hash function for a given set of keys. If 10 keys must be placed in a table of 100 elements, 63 percent of the possible hash functions are perfect (although as soon as the number of keys reaches 13 in a 100-item table, the majority are no longer perfect). In the example given earlier, if the compiler symbol table is to contain all symbols used in any program so that a large table must be allocated to allow for a large number of user-declared identifiers, a perfect hash function can easily be found for the reserved symbols of the language. The table can be initialized with the reserved symbols already in the positions determined by that function, with the user-defined symbols inserted as they are encountered. Although hash clashes may occur for user symbols, we are guaranteed immediate lookup for the reserved symbols.

In general it is desirable to have a perfect hash function for a set of n keys in a table of only n positions. Such a perfect hash function is called ***minimal***. In practice this is difficult to achieve. Sprugnoli has developed a number of perfect hash function determination algorithms. The algorithms are fairly complex and are not presented here. One technique finds perfect hash functions of the form $h(key) = (key + s)/d$ for some integers s and d. These are called ***quotient reduction perfect hash functions***, and, once found, are quite easy to compute.

For the key set 17, 138, 173, 294, 306, 472, 540, 551, and 618, Sprugnoli's algorithm finds the quotient reduction hash function $(key + 25)/64$, which yields the hash values 0, 2, 3, 4, 5, 7, 8, 9, and 10. The function is not minimal, since it distributes the 9 keys to a table of 11 positions. Sprugnoli's algorithm does, however, find the quotient reduction perfect hash function with the smallest table size.

An improvement to the algorithm yields a minimal perfect hash function of the form

```
h(key) = (key + s)/d          if key <= t
h(key) = (key + s + r)/d      if key > t
```

where the values s, d, t, and r are determined by the algorithm. However, the algorithm to discover such a minimal perfect hash function is $O(n^3)$ with a large constant of proportionality so that it is not practical for even very small key sets. A slight modification yields a more efficient algorithm that produces a near-minimal perfect hashing function of this form for small key sets. In the foregoing example, such a function is

```
h(key) = (key - 7)/72         if key <= 306
h(key) = (key - 42)/72        if key > 306
```

which yields the hash values 0, 1, 2, 3, 4, 5, 6, 7, and 8 and happens to be minimal. A major advantage of quotient reduction hash functions and their variants is that they are order preserving.

Sprugnoli also presents another group of hashing functions, called ***remainder reduction perfect hash functions***, which are of the form

```
h(key) = ((r + s * key) % x)/d
```

and an algorithm to produce values r, s, x, and d that yield such a perfect hash function for a given key set and a desired minimum load factor. If the minimum load factor is set to 1, a minimal perfect hash function results. However, the algorithm does not guarantee that a perfect remainder reduction hash function can be found in reasonable time for high load factors. Nevertheless, the algorithm can often be used to find minimal perfect hash functions for small key sets in reasonable time.

Unfortunately, Sprugnoli's algorithms are all at least $O(n^2)$ and are therefore only practical for small sets of keys (12 or fewer). Given a larger set of keys, k, a perfect hash function can be developed by a technique called ***segmentation***. This technique involves dividing k into a number of small sets, $k_0, k_2, \ldots, k_p$, and finding a perfect hash function h_i for each small set k_i. Assume a grouping function *set* such that *key* is in the set $k_{set(key)}$. If m_i is the maximum value of h_i on k_i and b_i is defined as $i + m_0 + m_1 + \cdots + m_{i-1}$, we can define the segmented hash function h as $h(key) = b_{set(key)} + h_{set(key)}(key)$. Of course, the function *set* that determines the grouping must be chosen with care to disperse the keys reasonably.

Jaeschke presents a method for generating minimal perfect hash functions using a technique called ***reciprocal hashing***. The reciprocal hash functions generated by Jaeschke's algorithm are of the form

```
h(key) = (c/(d * key + e)) % tablesize
```

for some constants c, d, and e, and *tablesize* equal to the number of keys. Indeed, if the keys are all relatively prime integers, a constant c can be found that yields a minimal perfect hash function of the form

```
(c/key) % tablesize
```

by the following algorithm. Assume that the keys are initially in a sorted array $k(0)$ through $k(n-1)$ and that $f(c, key)$ is the function $(c/key) \% n$.

```
c = ((n - 2) * k(0) * k(n - 1))/(k(n - 1) - k(0));
while (TRUE) {
   /* check if c yields a perfect hash function */
   bigi = -1;  /* these will be set to the largest values */
   bigj = -1;  /* such that f(c, k(bigi)) = f(c, k(bigj)) */
   for (i = 0; i < n; i++)
      val(i) = f(c, k(i));
   for (i = n - 1; bigi < 0 && i >= 0; i--) {
      vi = val(i);
      j = i - 1;
      while (bigi < 0 && j >= 0)
         if (vi == val(j)) {
            bigi = i;
            bigj = j;
         }
         else
            j--;
   } /* end for */
   if (bigi < 0)
      return;
   /* increment c */
   x = k(bigj) - (c % k(bigj));
   y = k(bigi) - (c % k(bigi));
   (x < y)  ?  c += x : c += y;
} /* end while */
```

Applying this algorithm to the key set 3, 5, 11, 14 yields $c = 11$ and the minimal perfect hash function $(11/key) \% 4$. For the key set 3, 5, 11, 13, 14 the algorithm produces $c = 66$. In practice one would set an upper limit on the value of c to ensure that the algorithm does not go on indefinitely.

If the keys are not relatively prime, Jaeschke presents another algorithm to compute values d and e so that the values of $d * k(i) + e$ are relatively prime, so that the algorithm can be used on those values.

For low values of n, approximately 1.82^n values of c are examined by this algorithm, which is tolerable for $n <= 20$. For values of n up to 40, we can divide the keys into two sets $s1$ and $s2$ of size $n1$ and $n2$, where all keys in $s1$ are smaller than those of $s2$. Then we can find values $c1$, $d1$, $e1$ and $c2$, $d2$, $e2$ for each of the sets individually and use

```
h(key) = (c1/(d1 * key + e1)) % n1
```

for keys in $s1$ and

```
h(key) = n1 + (c2/(d2 * key + e2)) % n2
```

for keys in $s2$. For larger key sets, the segmentation technique of Sprugnoli can be used.

Chang presents an order-preserving minimal perfect hash function that depends on the existence of a ***prime number function,*** $p(key)$, for the set of keys. Such a function always produces a prime number corresponding to a given key and has the additional property that if $key1$ is less than $key2$, $p(key1)$ is less than $p(key2)$. An example of such a prime number function is

$$p(x) = x^2 - x + 41 \quad \text{for} \quad 1 <= x <= 40$$

If such a prime number function has been found, Chang presents an efficient algorithm to produce a value c such that the function $h(key) = c \% p(key)$ is an order-preserving minimal hash function. However, prime number functions are difficult to find, and the value c is too large to be practically useful.

Cichelli presents a very simple method that often produces a minimal or near-minimal perfect hash function for a set of character strings. The hash function produced is of the form

```
h(key) = val(key[0]) + val(key[length(key) - 1]) + length(key)
```

where $val(c)$ is an integer value associated with the character c and $key[i]$ is the ith character of key. That is, add integer values associated with the first and last characters of the key to the length of the key. The integer values associated with particular characters are determined in two steps as follows.

The first step is to order the keys so that the sum of the occurrence frequencies of the first and last characters of the keys are in decreasing order. Thus if e occurs ten times as a last or first character, g occurs six times, t occurs nine times, and o occurs four times, the keys *gate*, *goat*, and *ego* have occurrence frequencies 16 (6 + 10), 15 (6 + 9), and 14 (10 + 4), respectively, and are therefore ordered properly.

Once the keys have been ordered, attempt to assign integer values. Each key is examined in turn. If the key's first or last character has not been assigned values, attempt to assign one or two values between 0 and some predetermined limit. If appropriate values can be assigned to produce a hash value that does not clash with the hash value of a previous key, tentatively assign those values. If not, or if both characters have been assigned values that result in a conflicting hash value, backtrack to modify tentative assignments made for a previous key. To find a minimal perfect hash function, the predetermined limit for each character is set to the number of distinct first and last character occurrences.

Cichelli perfect hash functions may not exist for some key sets. For example, if two keys of the same length have the same or reversed first and last characters, no such hash function can exist. In that case, different character positions may be used to develop the hash function. However, in other cases no such hash function can be found regardless of what character positions are used. In practice it is often useful to attempt to find a Cichelli perfect hash function before trying other methods. If the predetermined limit is set high enough, so that minimality is not required, Cichelli's algorithm can be quite practical for up to 50 keys. Cook and Oldehoeft present several improvements on the basic Cichelli method.

Sager presents an important generalization and extension of Cichelli's method that efficiently finds perfect hash functions for as many as 512 keys. The method is

fairly complex and is not presented here; the interested reader is referred to Sager's paper listed in the Bibliography.

An additional technique for generating minimal perfect hash functions is attributable to Du, Hsieh, Jea, and Shieh. The technique uses a number of nonperfect random hash functions $h_1, \ldots, h_j$ and a separate ***hash indicator table*** (or ***hit***) of size n. The table is initialized as follows. First, set all its entries to 0. Next, apply h_1 to all the keys. For all values x between 0 and $n - 1$ such that only one key hashes to x using $h1$, reset $hit[x]$ from 0 to 1. Remove all keys that hash to unique values using h_1 from the key set and apply h_2 to the remaining keys. For all values x between 0 and $n - 1$ such that $hit[x] = 0$ and only one key hashes to x using h_2, reset $hit[x]$ from 0 to 2. This process continues until either the key set is empty (in which case *hit* has been initialized and any remaining unused hash functions are unnecessary) or until all the hash functions have been applied (in which case, if there are remaining keys, a perfect hash function cannot be found using this method and the given random hash functions).

Once *hit* has been fully initialized, the hashing algorithm is as follows

```
for (i = 0; ; i++) {
   x = hi(key);
   if (hit(x) == i)
      return(x);
} /* end for */
```

The probability that a perfect hash function results rises very slowly as additional random hash functions are added. Therefore a segmentation technique, with distinct *hit* tables, should be used for large sets of keys.

Universal Classes of Hash Functions

As we have seen, it is difficult to obtain a perfect hash function for a large set of keys. It is also not possible to guarantee that a specific hash function minimizes collisions without knowing the precise set of keys to be hashed. If a particular hash function is found not to work well in practice in a particular application, it is difficult to come up with another hash function that does better.

Carter and Wegman have introduced the concept of a ***universal class of hash functions.*** Such a class consists of a set of hash functions *hi(key)*. Although an individual function in the class may work poorly on a particular input key set, enough of the functions work well for any random input set that if one function is chosen randomly from the class, it is likely to perform well on any input set that is actually presented.

Given a hash table of size m, and a set a of possible keys, a class of nh hash functions h is ***universal***$_2$ if there are no two keys in a on which more than nh/m of the functions in h result in collision. This means that no pair of distinct keys clash under more than $1/m$th of the functions. It can be shown that if k items have been inserted into a hash table of size m using a random member of a universal$_2$ class of hash functions with separate chaining, the expected number of probes for an unsuccessful search is less than $1 + k/m$ (the number for a successful search is even lower).

Carter and Wegman present several examples of such universal$_2$ classes. One example of such a class is for keys that can be represented as positive integers between 0 and $w - 1$ ($w - 1$ is usually the maximum value that fits into one computer word). Let p be a prime number larger than w, let s be an integer between 1 and $p - 1$, and let t be an integer between 0 and $p - 1$. Then define $h_{s,t}(key)$ as $((s*key + t) \% p) \% m$. The set of all such functions $h_{s,t}$ for given w and p is universal$_2$.

A second example is for keys consisting of i bits and a table size $m = 2^j$ for some j. Let a be an i-element array of table indexes (between 0 and $m - 1$). Then define $h_a(key)$ as the exclusive or of the indexes $a[k]$ such that the kth bit of key is 1. For example, if $m = 128$, $i = 16$, a is an array containing the values 47, 91, 35, 42, 16, 81, 113, 91, 12, 6, 47, 31, 106, 87, 95, and 11, and key is 15381 (which is 0011110000010101 as a 16-bit number), $h_a(key)$ is the exclusive or of $a[3]$, $a[4]$, $a[5]$, $a[6]$, $a[12]$, $a[14]$, and $a[16]$ (these are 35, 42, 16, 47, 31, 87 and 11), which is 01110101 or 117. The set of functions h_a for all such array values a is universal$_2$.

If the hash table is maintained internally and is not required between program runs (as in a compiler, for example), the hash function used may be generated by the program from a universal$_2$ class to guarantee reasonable average running time (although any particular run may be slow). In the preceding examples a random number generator might be used to select s, t, and the elements of a. If the hash table remains between program runs, as in a file or data base, then a random hash function from the universal$_2$ class might be selected initially, and if poor program behavior is observed (although this is unlikely), a new random function could be selected and the entire hash table reorganized at a convenient time.

Sarwate has introduced an even better category of hash functions classes, called ***optimally universal$_2$*** (***OU$_2$***) classes. If there are nk possible keys and m table entries, a set H containing nh hash functions is OU$_2$ if any two keys collide under exactly $nh*(nk - m)/(m*(nk - 1))$ functions in OU$_2$ and if, for any function h in H, every key collides with exactly $nk/m - 1$ other keys. Sarwate provides several examples of such OU$_2$ classes. Unfortunately, hash functions in OU$_2$ classes are difficult to compute and may not be practically useful.

EXERCISES

7.4.1. Write a C function *search*(*table*, *key*) that searches a hash table for a record with key *key*. The function accepts an integer key and a table declared by

```
struct record {
   KEYTYPE k;
   RECTYPE r;
   int flag;
} array[TABLESIZE];
```

table[*i*].*k* and *table*[*i*].*r* are the *i*th key and record, respectively. *table*[*i*].*flag* equals *FALSE* if the *i*th table position is empty and *TRUE* if it is occupied. The routine returns an integer in the range 0 to *tablesize* − 1 if a record with key *key* is present in the table. If no such record exists, the function returns −1. Assume the existence of a hashing

routine, *h(key)*, and a rehashing routine *rh(index)* that both produce integers in the range 0 to *tablesize* − 1.

7.4.2. Write a C function *sinsert(table, key, rec)* to search and insert into a hash table as in Exercise 7.4.1.

7.4.3. Develop a mechanism for detecting when all possible rehash positions of a given key have been searched. Incorporate this method into the C routines *search* and *sinsert* of the previous exercises.

7.4.4. Consider a double hashing method using primary hash function *h1(key)* and rehash function *rh(i)* = *tablesize* % (*i* + *h2(key)*, *tablesize*). Assume that *h2(key)* is relatively prime to *tablesize*, for any key *key*. Develop a search algorithm and an algorithm to insert a record whose key is known not to exist in the table so that the keys at sucessive rehashes of a single key are in ascending order. The insertion algorithm may rearrange records previously inserted into the table. Can you extend these algorithms to a search and insertion algorithm?

7.4.5. Suppose that a key is equally likely to be any integer between *a* and *b*. Suppose the midsquare hash method is used to produce an integer between 0 and 2^{k-1}. Is the result equally likely to be any integer within that range? Why?

7.4.6. Given a hash function *h(key)*, write a C simulation program to determine each of the following quantities after 0.8 * *tablesize* random keys have been generated. The keys should be random integers.

1. the percentage of integers between 0 and *tablesize* − 1 that do not equal *h(key)* for some generated key
2. the percentage of integers between 0 and *tablesize* − 1 that equal *h(key)* for more than one generated key
3. the maximum number of keys that hash into a single value between 0 and *tablesize* − 1
4. the average number of keys that hash into values between 0 and *tablesize* − 1, not including those values into which no key hashes

Run the program to test the uniformity of each of the following hash functions.

(a) *h(key)* = *key* % *tablesize* for *tablesize* a prime
(b) *h(key)* = *key* % *tablesize* for *tablesize* a power of 2
(c) The folding method using *exclusive or* to produce five-bit indices, where *tablesize* = 32
(d) The mid-square method using decimal arithmetic to produce four-digit indexes, where *tablesize* = 10,000

7.4.7. If a hash table contains *tablesize* positions, and *n* records currently occupy the table, the ***load factor*** is defined as *n*/*tablesize*. Show that if a hash function uniformly distributes keys over the *tablesize* positions of the table and if *lf* is the load factor of the table, (*n* − 1)**lf*/2 of the *n* keys in the table collided upon insertion with a previously entered key.

7.4.8. Assume that *n* random positions of a *tablesize*-element hash table are occupied, using hash and rehash functions that are equally likely to produce any index in the table. Show that the average number of comparisons needed to insert a new element is (*tablesize* + 1)/(*tablesize* − *n* + 1). Explain why linear probing does not satisfy this condition.

8

Graphs and Their Applications

In this chapter we consider a new data structure: the graph. We define some of the terms associated with graphs and show how to implement them in C. We also present several applications of graphs.

8.1 GRAPHS

A ***graph*** consists of a set of ***nodes*** (or ***vertices***) and a set of ***arcs*** (or ***edges***). Each arc in a graph is specified by a pair of nodes. Figure 8.1.1a illustrates a graph. The set of nodes is $\{A,B,C,D,E,F,G,H\}$, and the set of arcs is $\{(A,B),(A,D),(A,C),(C,D),(C,F),(E,G),(A,A)\}$. If the pairs of nodes that make up the arcs are ordered pairs, the graph is said to be a ***directed graph*** (or ***digraph***). Figure 8.1.1b, c, and d illustrate three digraphs. The arrows between nodes represent arcs. The head of each arrow represents the second node in the ordered pair of nodes making up an arc, and the tail of each arrow represents the first node in the pair. The set of arcs for the graph of Figure 8.1.1b is $\{<A,B>,<A,C>,<A,D>,<C,D>,<F,C>,<E,G>,<A,A>\}$. We use parentheses to indicate an unordered pair and angled brackets to indicate an ordered pair. In the first three sections of this chapter, we restrict our attention to digraphs. We consider undirected graphs again in Section 8.4.

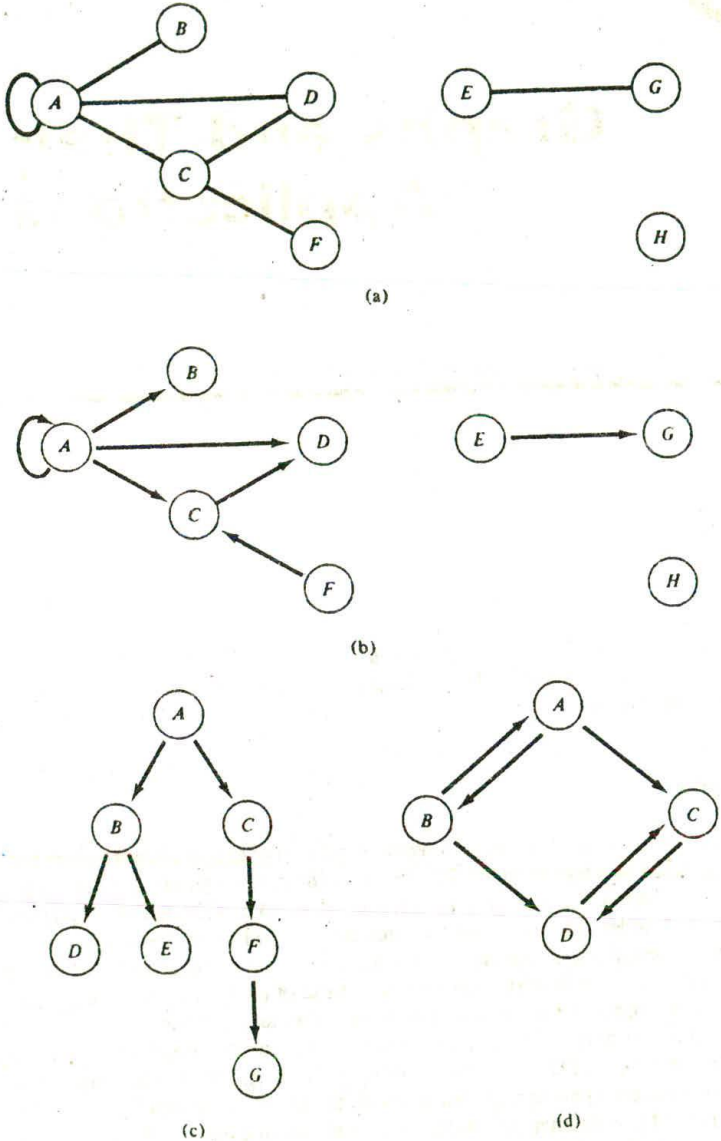


Figure 8.1.1 Examples of graphs.

Note that a graph need not be a tree (Figure 8.1.1a, b, and d) but that a tree must be a graph (Figure 8.1.1c). Note also that a node need not have any arcs associated with it (node *H* in Figure 8.1.1a and b).

A node *n* is ***incident*** to an arc *x* if *n* is one of the two nodes in the ordered pair of nodes that constitute *x*. (We also say that *x* is incident to *n*.) The ***degree*** of a node is the number of arcs incident to it. The ***indegree*** of a node *n* is the number of arcs that have *n* as the head, and the ***outdegree*** of *n* is the number of arcs that have *n* as the tail. For example, node *A* in Figure 8.1.1d has indegree 1, outdegree 2, and degree 3. A node *n* is ***adjacent*** to a node *m* if there is an arc from *m* to *n*. If *n* is adjacent to *m*, *n* is called a ***successor*** of *m*, and *m* a ***predecessor*** of *n*.

A ***relation*** *R* on a set *A* is a set of ordered pairs of elements of *A*. For example, if $A = \{3,5,6,8,10,17\}$, the set $R = \{<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>\}$ is a relation. If $<x,y>$ is a member of a relation *R*, *x* is said to be ***related*** to *y* in *R*. The above relation *R* may be described by saying that *x* is related to *y* if *x* is less than *y* and the remainder obtained by dividing *y* by *x* is odd. <8,17> is a member of this relation, since 8 is smaller than 17 and the remainder on dividing 17 by 8 is 1, which is odd.

A relation may be represented by a graph in which the nodes represent the underlying set and the arcs represent the ordered pairs of the relation. Figure 8.1.2a illustrates the graph representing the foregoing relation. A number may be associated with each arc of a graph as in Figure 8.1.2b. In that figure, the number associated with each arc is the remainder obtained by dividing the integer at the head of the arc by the integer at the tail. Such a graph, in which a number is associated with each arc, is called a ***weighted graph*** or a ***network***. The number associated with an arc is called its ***weight***.

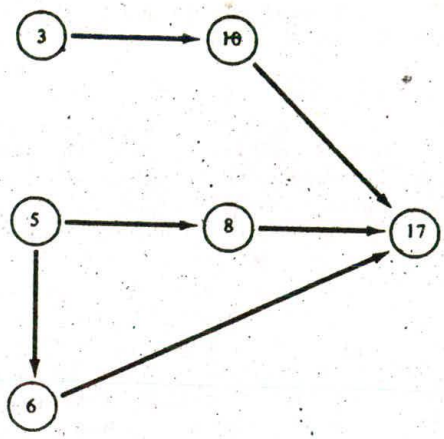
We identify several primitive operations that are useful in dealing with graphs. The operation *join(a,b)* adds an arc from node *a* to node *b* if one does not already exist. *joinwt(a,b,x)* adds an arc from *a* to *b* with weight *x* in a weighted graph. *remv(a,b)* and *remvwt(a,b,x)* remove an arc from *a* to *b* if one exists (*remvwt* also sets *x* to its weight). Although we may also want to add or delete nodes from a graph, we postpone a discussion of these possibilities until a later section. The function *adjacent(a,b)* returns *true* if *b* is adjacent to *a*, and *false* otherwise.

A ***path of length k*** from node *a* to node *b* is defined as a sequence of $k + 1$ nodes $n_1, n_2, \ldots, n_{k+1}$ such that $n_1 = a$, $n_{k+1} = b$ and *adjacent*(n_i, n_{i+1}) is *true* for all *i* between 1 and *k*. If for some integer *k*, a path of length *k* exists between *a* and *b*, there is a ***path*** from *a* to *b*. A path from a node to itself is called a ***cycle***. If a graph contains a cycle, it is ***cyclic***; otherwise it is ***acyclic***. A directed acyclic graph is called a ***dag*** from its acronym.

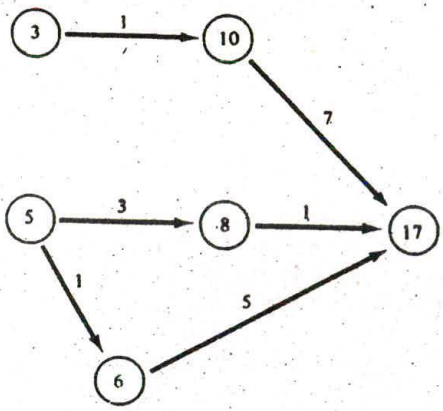
Consider the graph of Figure 8.1.3. There is a path of length 1 from *A* to *C*, two paths of length 2 from *B* to *G*, and a path of length 3 from *A* to *F*. There is no path from *B* to *C*. There are cycles from *B* to *B*, from *F* to *F*, and from *H* to *H*. Be sure that you can find all paths of length less than 9 and all cycles in the figure.

Application of Graphs

We now consider an example. Assume one input line containing four integers followed by any number of input lines with two integers each. The first integer on the



(a)



(b)

Figure 8.1.2 Relations and graphs.

first line, n, represents a number of cities, which for simplicity are numbered from 0 to $n-1$. The second and third integers on that line are between 0 and $n-1$ and represent two cities. It is desired to travel from the first city to the second using exactly nr roads, where nr is the fourth integer on the first input line. Each subsequent input line contains two integers representing two cities, indicating that there is a road from the first city to the second. The problem is to determine whether there is a path of required length by which one can travel from the first of the given cities to the second.

A plan for solution is the following: Create a graph with the cities as nodes and the roads as arcs. To find a path of length nr from node A to node B, look for a node C such that an arc exists from A to C and a path of length $nr-1$ exists from C to B. If

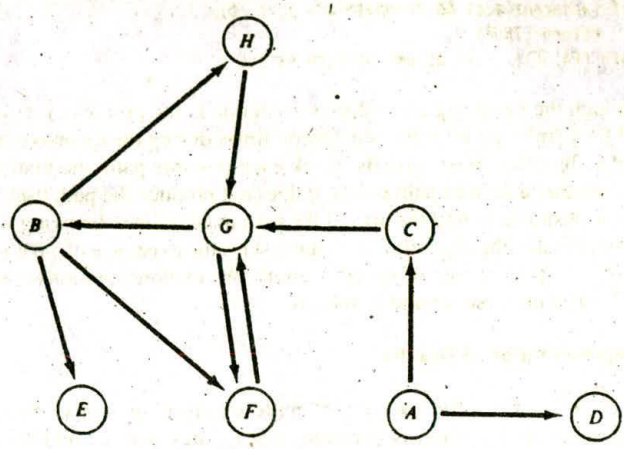


Figure 8.1.3

these conditions are satisfied for some node C, the desired path exists. If the conditions are not satisfied for any node C, the desired path does not exist. The algorithm uses an auxiliary recursive function *findpath*(k,a,b), whose algorithm we also present shortly. This function returns *true* if there is a path of length k from A to B and *false* otherwise. The algorithms for the program and the function follow:

```
scanf ("%d", &n);                    /*  number of cities      */
create n nodes and label them from 0 to n - 1;
scanf ("%d %d", &a, &b);          /* seek path from a to b  */
scanf ("%d", &nr);                   /*  desired number of     */
                                     /*    roads to take       */
while(scanf("%d %d", &city1, &city2) != EOF)
   join(city1, city2);
if (findpath(nr,a,b))
   printf("a path exists from %d to %d in %d steps",
                                              a, b, nr);
else
   printf("no path exists from %d to %d in %d steps",
                                                 a, b, nr);
```

The algorithm for the function *findpath*(k,a,b) follows:

```
if (k == 1)
   /*  search for a path of length 1 */
   return (adjacent(a,b));
/* determine if there is a path through c  */
```

```
for (c = 0; c < n; ++c)
   if (adjacent(a,c) && findpath(k - 1, c, b))
      return (TRUE);
return (FALSE);    /* assume no path exists */
```

Although the foregoing algorithm is a solution to the problem, it has several deficiencies. Many paths are investigated several times during the recursive process. Also, although the algorithm must actually check each possible path, the final result merely ascertains whether a desired path exists; it does not produce the path itself. More likely than not, it is desirable to find the arcs of the path in addition to knowing whether or not a path exists. Finally, the algorithm does not test for the existence of a path regardless of length; it only tests for a path of specific length. We explore solutions to some of these problems later in this chapter and in the exercises.

C Representation of Graphs

Let us now turn to the question of representing graphs in C. Suppose that the number of nodes in the graph is constant: that is, arcs may be added or deleted but nodes may not. A graph with 50 nodes could then be declared as follows:

```
#define  MAXNODES  50

struct node {
   /* information associated with each node */
};
struct arc {
   int adj;
   /* information associated with each arc  */
};
struct graph {
   struct node nodes[MAXNODES];
   struct arc arcs[MAXNODES][MAXNODES];
};
struct graph g;
```

Each node of the graph is represented by an integer between 0 and *MAXNODES* − 1, and the array field *nodes* represents the appropriate information assigned to each node. The array field *arcs* is a two-dimensional array representing every possible ordered pair of nodes. The value of *g.arcs*[*i*][*j*].*adj* is either *TRUE* or *FALSE* depending on whether or not node *j* is adjacent to node *i*. The two-dimensional array *g.arcs*[ ][ ].*adj* is called an ***adjacency matrix***. In the case of a weighted graph, each arc can also be assigned information.

Frequently the nodes of a graph are numbered from 0 to *MAXNODES* − 1 and no information is assigned to them. Also, we may be interested in the existence of arcs but not in any weights or other information about them. In such cases the graph could be declared simply by

```
int  adj[MAXNODES][MAXNODES];
```

In effect, the graph is totally described by its adjacency matrix. We present the code for the primitive operations just described in the case where a graph is described by its adjacency matrix.

```
void join (int adj[][MAXNODES], int node1, int node2)
{
   /* add an arc from node1 to node2 */
   adj[node1][node2] = TRUE;
} /* end join */

void remv(int adj[][MAXNODES], int node1, int node2)
{
   /* delete arc from node1 to node2 if one exists */
   adj[node1][node2] = FALSE;
} /* end remv */

int adjacent(int adj[][MAXNODES], int node1, int node2)
{
   return((adj[node1][node2] == TRUE)? TRUE: FALSE);
} /* end adjacent */
```

A weighted graph with a fixed number of nodes may be declared by

```
struct arc {
   int adj;
   int weight;
};
struct arc g[MAXNODES][MAXNODES];
```

The routine *joinwt*, which adds an arc from *node*1 to *node*2 with a given weight *wt*, may be coded as follows:

```
void joinwt (struct arc g[][MAXNODES], int node1, int node2, int wt)
{
   g[node1][node2].adj = TRUE;
   g[node1][node2].weight = wt;
} /* end joinwt */
```

The routine *remvwt* is left to the reader as an exercise.

Transitive Closure

Let us assume that a graph is completely described by its adjacency matrix, *adj* (that is, no data is associated with the nodes and the graph is not weighted). Consider the logical expression $adj[i][k]$ && $adj[k][j]$. Its value is *TRUE* if and only if the values of both $adj[i][k]$ and $adj[k][j]$ are *TRUE*, which implies that there is an arc from node

i to node k and an arc from node k to node j. Thus $adj[i][k]$ && $adj[k][j]$ equals *TRUE* if and only if there is a path of length 2 from i to j passing through k.

Now consider the expression

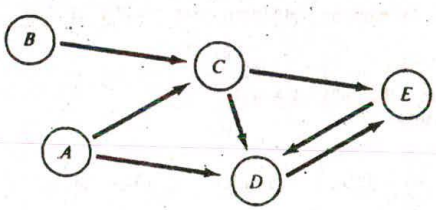
```
(adj[i][0] &&  adj[0][j]) || (adj[i][1] &&  adj[1][j])
   ||...|| (adj[i][MAXNODES - 1] && adj[MAXNODES - 1][j])
```

The value of this expression is *TRUE* only if there is a path of length 2 from node i to node j either through node 0 or through node 1, ... or through node *MAXNODES* − 1. This is the same as saying that the expression evaluates to *TRUE* if and only if there is some path of length 2 from node i to node j.

Consider an array adj_2 such that $adj_2[i][j]$ is the value of the foregoing expression. adj_2 is called the ***path matrix of length 2***. $adj_2[i][j]$ indicates whether or not there is a path of length 2 between i and j. (If you are familiar with matrix multiplication, you should realize that adj_2 is the product of adj with itself, with numerical multiplication replaced by conjunction (the && operation) and addition replaced by disjunction (the || operation).) adj_2 is said to be the ***Boolean product*** of adj with itself.

Figure 8.1.4 illustrates this process. Figure 8.1.4a depicts a graph and its adjacency matrix in which *true* is represented by 1 and *false* is represented by 0. Figure 8.1.4b is the Boolean product of that matrix with itself, and thus is the path matrix of length 2 for the graph. Convince yourself that a 1 appears in row i, column j of the matrix of Figure 8.1.4b if and only if there is a path of length 2 from node i to node j in the graph.

Similarly, define adj_3, the path matrix of length three, as the Boolean product of adj_2 with adj. $adj_3[i][j]$ equals *true* if and only if there is a path of length 3 from i to j. In general, to compute the path matrix of length l, form the Boolean product of the path matrix of length $l - 1$ with the adjacency matrix. Figure 8.1.5 illustrates the matrices adj_3 and adj_4 of the graph in Figure 8.1.4a.



	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	1	0	0

(a) adj

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

(b) adj_2

Figure 8.1.4

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

(a) adj_3

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

(b) adj_4

Figure 8.1.5

Assume that we want to know whether a path of length 3 or less exists between two nodes of a graph. If such a path exists between nodes i and j, it must be of length 1, 2, or 3. If there is a path of length 3 or less between nodes i and j the value of

```
adj[i][j] || adj2[i][j] || adj3[i][j]
```

must be *true*. Figure 8.1.6 shows the matrix formed by "or-ing" the matrices *adj*, adj_2, and adj_3. This matrix contains the value *TRUE* (represented by the value 1 in the figure) in row i, column j, if and only if there is a path of length 3 or less from node i to node j.

Suppose that we wish to construct a matrix *path* such that *path*[i][j] is *TRUE* if and only if there is some path from node i to node j (of any length). Clearly,

```
path[i][j] == adj[i][j] || adj2[i][j] || ...
```

However, the preceding equation cannot be used in computing *path*, since the process that it describes is an infinite one. However, if the graph has n nodes, it must be true that

```
path[i][j] == adj[i][j] || adj2[i][j] || ... || adjn[i][j]
```

This is because if there is a path of length $m > n$ from i to j such as $i, i_2, i_3, \ldots, i_m, j$, there must be another path from i to j of length less than or equal to n. To see this, note that since there are only n nodes in the graph, at least one node k must appear in the path twice. The path from i to j can be shortened by removing the cycle from k to k. This process is repeated until no two nodes in the path (except possibly i and j) are equal and therefore the path is of length n or less. Figure 8.1.7 illustrates the matrix *path* for the graph of Figure 8.1.4a. The matrix *path* is often called the ***transitive closure*** of the matrix *adj*.

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

Figure 8.1.6

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

Figure 8.1.7 *path* = *adj* or adj_2 or adj_3 or adj_4 or adj_5.

We may write a C routine that accepts an adjacency matrix *adj* and computes its transitive closure *path*. This routine uses an auxiliary routine *prod*(*a*,*b*,*c*), which sets the array *c* equal to the Boolean product of *a* and *b*.

```
void transclose (int adj[][MAXNODES], int path[][MAXNODES])
{
   int i, j, k;
   int newprod[MAXNODES][MAXNODES],
       adjprod[MAXNODES][MAXNODES];

   for (i = 0; i < MAXNODES; ++i)
      for (j = 0; j < MAXNODES; ++j)
         adjprod[i][j] = path[i][j] = adj[i][j];
   for (i = 1; i < MAXNODES; ++i) {
      /* i represents the number of times adj has  */
      /* been multiplied by itself to obtain       */
      /* adjprod.  At this point path represents   */
      /*     all paths of length i or less         */
      prod (adjprod, adj, newprod);
      for (j = 0; j < MAXNODES; ++j)
         for (k = 0; k < MAXNODES; ++k)
            path[j][k] = path[j][k] || newprod[j][k];
      for (j = 0; j < MAXNODES; ++j)
         for (k = 0; k < MAXNODES; ++k)
            adjprod[j][k] = newprod[j][k];
   } /* end for */
} /* end transclose */
```

The routine *prod* may be written as follows:

```
void prod (int a[][MAXNODES], int b[][MAXNODES], int c[][MAXNODES])
{
   int i, j, k, val;

   for (i = 0; i < MAXNODES; ++i)       /*  pass through rows     */
      for (j = 0; j < MAXNODES; ++j) { /*  pass through columns  */
         val = FALSE;
         for (k = 0; k < MAXNODES; ++k)
            val = val || (a[i][k] && b[k][j]);
         c[i][j] = val;
      } /* end for j */
} /* end prod */
```

To analyze the efficiency (or inefficiency) of this routine, note that finding the boolean product by the method we have presented is $O(n^3)$, where n is the number of graph nodes (that is, *MAXNODES*). In *transclose*, this process (the call to *prod*) is embedded in a loop that is repeated $n - 1$ times, so that the entire transitive closure routine is $O(n^4)$.

Warshall's Algorithm

The foregoing method is quite inefficient. Let us see if a more efficient method to compute *path* can be produced. Let us define the matrix $path_k$ such that $path_k[i][j]$ is *true* if and only if there is a path from node i to node j that does not pass through any nodes numbered higher than k (except, possibly, for i and j themselves). How can the value of $path_{k+1}[i][j]$ be obtained from $path_k$? Clearly for any i and j such that $path_k[i][j] = TRUE$, $path_{k+1}[i][j]$ must be *TRUE* (why?). The only situation in which $path_{k+1}[i][j]$ can be *TRUE* while $path_k[i][j]$ equals *FALSE* is if there is a path from i to j passing through node $k + 1$, but there is no path from i to j passing through only nodes 1 through k. But this means that there must be a path from i to $k + 1$ passing through only nodes 1 through k and a similar path from $k + 1$ to j. Thus $path_{k+1}[i][j]$ equals *TRUE* if and only if one of the following two conditions holds:

1. $path_k[i][j] == TRUE$
2. $path_k[i][k + 1] == TRUE$ and $path_k[k + 1][j] == TRUE$

This means that $path_{k+1}[i][j]$ equals $path_k[i][j] \parallel (path_k[i][k + 1] \&\& path_k[k + 1][j])$. An algorithm to obtain the matrix $path_k$ from the matrix $path_{k-1}$ based on this observation follows:

```
for (i = 0; i < MAXNODES; ++i)
   for (j = 0; j < MAXNODES; ++j)
      path_k [i][j] = path_k-1 [i][j] || (path_k-1 [i][k] && path_k-1 [k][j]);
```

This may be logically simplified and made more efficient as follows:

```
for (i = 0; i < MAXNODES; ++i)
   for (j = 0; j < MAXNODES; ++j)
      path_k [i][j] = path_k-1 [i][j];
for (i = 0; i < MAXNODES; ++i)
   if (path_k-1 [i][k] == TRUE)
      for (j = 0; j < MAXNODES; ++j)
         path_k [i][j] = path_k-1 [i][j] || path_k-1 [k][j];
```

Clearly, $path_0[i][j] = adj$, since the only way to go from node i to node j without passing through any other nodes is to go directly from i to j. Further, $path_{MAXNODES-1}[i][j] = path[i][j]$, since if a path may pass through any nodes numbered from 0 to $MAXNODES - 1$, any path from node i to node j may be selected. The following C routine may therefore be used to compute the transitive closure:

```
void transclose (int adj[][MAXNODES], int path[][MAXNODES])
{
   int i, j, k;
   for (i = 0; i < MAXNODES; ++i)
      for (j = 0; j < MAXNODES; ++j)
         path[i][j] = adj[i][j];  /* path starts off as adj */
```

```
    for (k = 0; k < MAXNODES; ++k)
      for (i = 0; i <.MAXNODES; ++i)
        if (path [i][k] == TRUE)
          for (j = 0; j < MAXNODES; ++j)
            path [i][j] = path [i][j] || path [k][j];
}/*  end transclose */
```

This technique increases the efficiency of finding the transitive closure to $O(n^3)$. The method is often called ***Warshall's algorithm***, after its discoverer.

Shortest-Path Algorithm

In a weighted graph, or network, it is frequently desired to find the shortest path between two nodes, *s* and *t*. The shortest path is defined as a path from *s* to *t* such that the sum of the weights of the arcs on the path is minimized. To represent the network, we assume a weight function, such that *weight* (*i*,*j*) is the weight of the arc from *i* to *j*. If there is no arc from *i* to *j*, *weight*(*i*,*j*) is set to an arbitrarily large value to indicate the infinite cost (that is, the impossibility) of going directly from *i* to *j*.

If all weights are positive, the following algorithm, attributable to Dijkstra, determines the shortest path from *s* to *t*. Let the variable *infinity* hold the largest possible integer. *distance*[*i*] keeps the cost of the shortest path known thus far from *s* to *i*. Initially, *distance*[*s*] = 0 and *distance*[*i*] = *infinity* for all *i* != *s*. A set *perm* contains all nodes whose minimal distance from *s* is known—that is, those nodes whose distance value is permanent and will not change. If a node *i* is a member of *perm*, *distance*[*i*] is the minimal distance from *s* to *i*. Initially, the only member of *perm* is *s*. Once *t* becomes a member of *perm*, *distance*[*t*] is known to be the shortest distance from *s* to *t*, and the algorithm terminates.

The algorithm maintains a variable, *current*, that is the node that has been added to *perm* most recently. Initially, *current* = *s*. Whenever a node *current* is added to *perm*, *distance* must be recomputed for all successors of *current*. For every successor *i* of *current*, if *distance*[*current*] + *weight*(*current*, *i*) is less than *distance*[*i*], the distance from *s* to *i* through *current* is smaller than any other distance from *s* to *i* found thus far. Thus *distance*[*i*] must be reset to this smaller value.

Once *distance* has been recomputed for every successor of *current*, *distance*[*j*] (for any *j*) represents the shortest path from *s* to *j* that includes only members of *perm* (except for *j* itself). This means that for the node *k*, not in *perm*, for which *distance*[*k*] is smallest, there is no path from *s* to *k* whose length is shorter than *distance*[*k*]. (*distance*[*k*] is already the shortest distance to *k* that includes only nodes in *perm*, and any path to *k* that includes a node *nd* as its first node not in *perm* must be longer, since *distance*[*nd*] is greater than *distance*[*k*].) Thus *k* can be added to *perm*. *current* is then reset to *k* and the process is repeated.

The following is a C routine to implement this algorithm. In addition to calculating distances, the program finds the shortest path itself by maintaining an array *precede* such that *precede*[*i*] is the node that precedes node *i* on the shortest path found thus far. An array *perm* is used to keep track of the corresponding set. *Perm*[*i*] is 1 if *i* is a

member of the set and 0 if not. The routine accepts a weight matrix (with nonadjacent arcs having a weight of *infinity*) and two nodes, *s* and *t*, and calculates the minimum distance *pd* from *s* to *t* as well as the array *precede* to define the path. The routine assumes the following definitions and declarations:

```
#define INFINITY ...
#define MAXNODES ...
#define MEMBER  1
#define NONMEMBER  0

void shortpath (int weight[][MAXNODES], int s, int t, int *pd, int precede[])
{
   int distance[MAXNODES], perm[MAXNODES];
   int current, i, k, dc;
   int smalldist, newdist;

   /* initialization */
   for (i = 0; i < MAXNODES; ++i) {
      perm[i] = NONMEMBER;
      distance[i] = INFINITY;
   } /*  end for */
   perm[s] = MEMBER;
   distance[s] = 0;
   current = s;
   while (current != t) {
      smalldist = INFINITY;
      dc = distance[current];
      for (i = 0; i < MAXNODES; i++)
         if (perm[i] == NONMEMBER) {
            newdist = dc + weight[current][i];
            if (newdist < distance[i]) {
               /* distance from s to i through current is */
               /*      smaller than  distance[i]          */
               distance[i] = newdist;
               precede[i] = current;
            } /* end  if */
            /* determine the smallest distance */
            if (distance[i] < smalldist) {
               smalldist = distance[i];
               k = i;
            } /* end if */
         } /* end for ... if  */
      current = k;
      perm[current] = MEMBER;
   }/*  end while  */
   *pd = distance[t];
} /* end shortpath  */
```

An alternative implementation that maintains the set of "permanent" nodes as a linked list instead of the array *perm* is left as an exercise for the reader.

Assuming that a function *all*(x) has been defined to return *TRUE* if every element of array x is 1 and *FALSE* otherwise, Dijkstra's algorithm can be modified to find the shortest path from a node s to every other node in the graph by modifying the ***while*** header to

```
while (all(perm) == FALSE)
```

To analyze the efficiency of this implementation of Dijkstra's algorithm, note that one node is added to *perm* in each iteration of the ***while*** loop so that, potentially, the loop must be repeated n times (where n = *MAXNODES*, the number of nodes in the graph). Each iteration involves examining every node [**for** ($i = 0$; $i <$ *MAXNODES*; $++i$)], so the entire algorithm is $O(n^2)$. We examine a more efficient implementation of Dijkstra's algorithm in Section 8.3.

EXERCISES

8.1.1. For the graph of Figure 8.1.1b:
- **(a)** Find its adjacency matrix.
- **(b)** Find its path matrix using powers of the adjacency matrix.
- **(c)** Find its path matrix using Warshall's algorithm.

8.1.2. Draw a digraph to correspond to each of the following relations on the integers from 1 to 12.
- **(a)** x is related to y if $x - y$ is evenly divisible by 3.
- **(b)** x is related to y if $x + 10 * y < x * y$.
- **(c)** x is related to y if the remainder on division of x by y is 2.

Compute the adjacency and path matrices for each of these relations.

8.1.3. A node $n1$ is ***reachable*** from a node $n2$ in a graph if $n1$ equals $n2$ or there is a path from $n2$ to $n1$. Write a C function *reach*(*adj*,*i*,*j*) that accepts an adjacency matrix and two integers and determines if the jth node in the digraph is reachable from the ith node.

8.1.4. Write C routines which, given an adjacency matrix and two nodes of a graph, compute:
- **(a)** The number of paths of a given length existing between them
- **(b)** The number of total paths existing between them

8.1.5. A relation on a set S (and its corresponding digraph) is ***symmetric*** if for any two elements x and y in S such that x is related to y, y is also related to x.
- **(a)** What must be true of a digraph if it represents a symmetric relation?
- **(b)** Give an example of a symmetric relation and draw its digraph.
- **(c)** What must be true of the adjacency matrix of a symmetric digraph?
- **(d)** Write a C routine that accepts an adjacency matrix and determines if the digraph it represents is symmetric.

8.1.6. A relation on a set S (and its corresponding digraph and adjacency matrix) is ***transitive*** if for any three elements x, y, and z in S, if x is related to y and y is related to z, x is related to z.

(a) What must be true of a digraph if it represents a transitive relation?
(b) Give an example of a transitive relation and draw its digraph.
(c) What must be true of the Boolean product of the adjacency matrix of a transitive digraph with itself?
(d) Write a C routine that accepts an adjacency matrix and determines if the digraph it represents is transitive.
(e) Prove that the transitive closure of any digraph is transitive.
(f) Prove that the smallest transitive digraph that includes all nodes and arcs of a given digraph is the transitive closure of that digraph.

8.1.7. Given a digraph, prove that it is possible to renumber its nodes so that the resultant adjacency matrix is lower triangular (see Exercise 1.2.8) if and only if the digraph is acyclic. Write a C function *lowtri(adj, ltadj, perm)* that accepts an adjacency matrix *adj* of an acyclic graph and creates a lower triangular adjacency matrix *ltadj* that represents the same graph. *perm* is a one-dimensional array such that *perm*[*i*] is set to the new number assigned to the node that was numbered *i* in the matrix *adj*.

8.1.8. Rewrite the routine *shortpath* to implement the set of "permanent" nodes as a linked list. Show that the efficiency of the method remains $O(n^2)$.

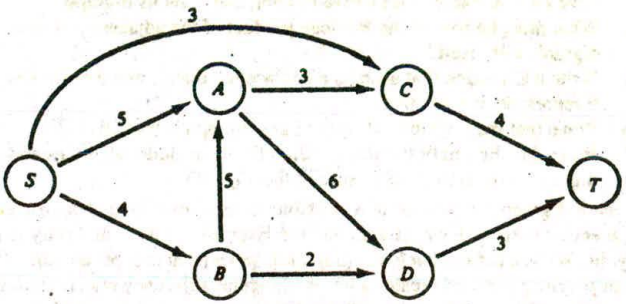
8.2 FLOW PROBLEM

In this section we consider a real-world problem and illustrate a solution that uses a weighted graph. There are a number of formulations of this problem whose solutions carry over to a wide range of applications. We present one such formulation here and refer the reader to the literature for alternate versions.

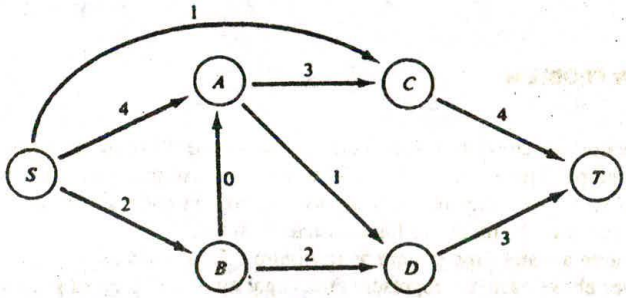
Assume a water pipe system as in Figure 8.2.1a. Each arc represents a pipe and the number above each arc represents the capacity of that pipe in gallons per minute. The nodes represent points at which pipes are joined and water is transferred from one pipe to another. Two nodes, *S* and *T*, are designated as a ***source*** of water and a ***user*** of water (or a ***sink***), respectively. This means that water originating at *S* must be carried through the pipe system to *T*. Water may flow through a pipe in only one direction (pressure sensitive valves may be used to prevent water from flowing backward), and there are no pipes entering *S* or leaving *T*. Thus, a weighted directed graph, as in Figure 8.2.1a, is an ideal data structure to model the situation.

We would like to maximize the amount of water flowing from the source to the sink. Although the source may be able to produce water at a prodigious rate and the sink may be able to consume water at a comparable rate, the pipe system may not have the capacity to carry it all from the source to the sink. Thus the limiting factor of the entire system is the pipe capacity. Many other real-world problems are similar in nature. The system could be an electrical network, a railway system, a communications network, or any other distribution system in which one wants to maximize the amount of an item being delivered from one point to another.

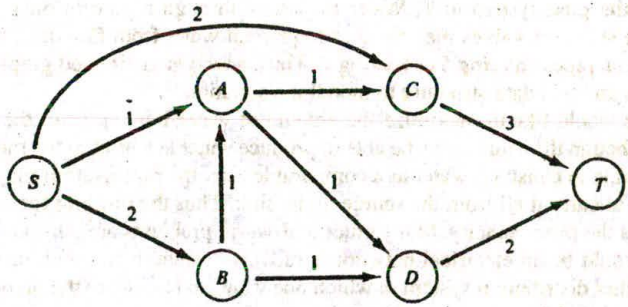
Define a ***capacity function,*** $c(a,b)$, where a and b are nodes, as follows: If *adjacent*(a,b) is *true* (that is, if there is a pipe from a to b), $c(a,b)$ is the capacity of the pipe from a to b. If there is no pipe from a to b, $c(a,b) = 0$. At any point in the operation



(a) A flow problem.



(b) A flow function.



(c) A flow function.

Figure 8.2.1

of the system, a given amount of water (possibly 0) flows through each pipe. Define a ***flow function***, $f(a,b)$, where a and b are nodes, as 0 if b is not adjacent to a, and as the amount of water flowing through the pipe from a to b otherwise. Clearly, $f(a,b) >= 0$ for all nodes a and b. Furthermore, $f(a,b) <= c(a,b)$ for all nodes a and b, since a pipe may not carry more water than its capacity. Let v be the amount of water that flows through the system from S to T. Then the amount of water leaving S through all pipes equals the amount of water entering T through all pipes and both these amounts equal v. This can be stated by the equality:

$$\sum_{x \in nodes} f(S, x) = v = \sum_{x \in nodes} f(x, T)$$

No node other than S can produce water and no node other than T can absorb water. Thus the amount of water leaving any node other than S or T is equal to the amount of water entering that node. This can be stated by

$$\sum_{y \in nodes} f(x, y) = \sum_{y \in nodes} f(y, x) \text{ for all nodes } x != S, T$$

Define the ***inflow*** of a node x as the total flow entering x and the ***outflow*** as the total flow leaving x. The foregoing conditions may be rewritten as

$$outflow\ (S) = inflow\ (T) = v$$
$$inflow\ (x) = outflow\ (x) \text{ for all } x != S, T$$

Several flow functions may exist for a given graph and capacity function. Figures 8.2.1b and c illustrate two possible flow functions for the graph of Figure 8.2.1a. Make sure that you understand why both of them are valid flow functions and why both satisfy the foregoing equations and inequalities.

We wish to find a flow function that maximizes the value of v, the amount of water going from S to T. Such a flow function is called ***optimal***. Clearly, the flow function of Figure 8.2.1b is better than the one of Figure 8.2.1c, since v equals 7 in the former but only 5 in the latter. See if you can find a flow function which is better than the one of Figure 8.2.1b.

One valid flow function can be achieved by setting $f(a,b)$ to 0 for all nodes a and b. Of course this flow function is least optimal, since no water flows from S to T. Given a flow function, it can be improved so that the flow from S to T is increased. However, the improved version must satisfy all the conditions for a valid flow function. In particular, if the flow entering any node (except for S or T) is increased or decreased, the flow leaving that node must be increased or decreased correspondingly. The strategy for producing an optimal flow function is to begin with the zero flow function and to improve upon it successively until an optimal flow function is produced.

Improving a Flow Function

Given a flow function f there are two ways to improve upon it. One way consists of finding a path $S = x_1, x_2, \ldots, x_n = T$ from S to T such that the flow along each arc

in the path is strictly less than the capacity (that is, $f(x_{k-1}, x_k) < c(x_{k-1}, x_k)$ for all k between 1 and $n-1$). The flow can be increased on each arc in such a path by the minimum value of $c(x_{k-1}, x_k) - f(x_{k-1}, x_k)$ for all k between 1 and $n-1$ (so that when the flow has been increased along the entire path there is at least one arc $<x_{k-1}, x_k>$ in the path for which $f(x_{k-1}, x_k) = c(x_{k-1}, x_k)$ and through which the flow may not be increased).

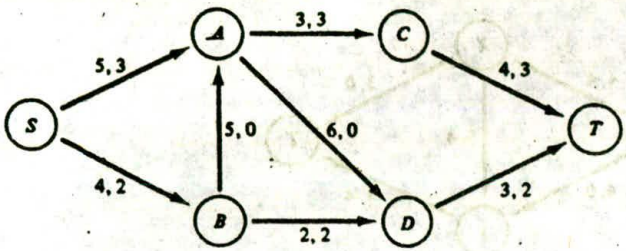
This may be illustrated by the graph of Figure 8.2.2a which gives the capacity and the current flow respectively for each arc. There are two paths from S to T with positive flow ((S,A,C,T) and (S,B,D,T)). However each of these paths contains one arc ($<A,C>$ and $<B,D>$) in which the flow equals the capacity. Thus the flow along these paths may not be improved. However, the path (S,A,D,T) is such that the capacity of each arc in the path is greater than its current flow. The maximum amount by which the flow can be increased along this path is 1, since the flow along arc $<D,T>$ cannot exceed 3. The resulting flow function is shown in Figure 8.2.2b. The total flow from S to T has been increased from 5 to 6. To see that the result is still a valid flow function note that for each node (except T) whose inflow is increased, the outflow is increased by the same amount.

Are there any other paths whose flow can be improved? In this example, you should satisfy yourself that there are not. However, given the graph of Figure 8.2.2a we could have chosen to improve the path (S,B,A,D,T). The resulting flow function is illustrated in Figure 8.2.2c. This function also provides for a net flow of 6 from S to T and is therefore neither better nor worse than the flow function of Figure 8.2.2b.

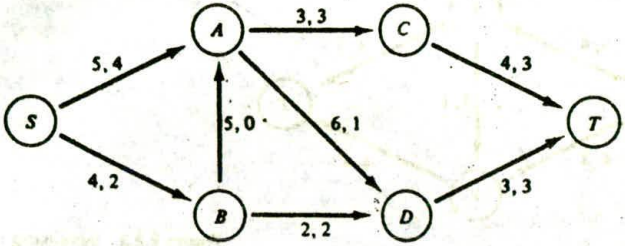
Even if there is no path whose flow may be improved, there may be another method of improving the net flow from the source to the sink. This is illustrated by Figure 8.2.3. In Figure 8.2.3a there is no path from S to T whose flow may be improved. But if the flow from X to Y is reduced, the flow from X to T can be increased. To compensate for the decrease in the inflow of Y the flow from S to Y could be increased, thereby increasing the net flow from S to T. The flow from X to Y can be redirected to T as shown in Figure 8.2.3b and the net flow from S to T can thereby be increased from 4 to 7.

We may generalize this second method as follows. Suppose that there is a path from S to some node Y, a path from some node X to T and a path from X to Y with positive flow. Then the flow along the path from X to Y may be reduced and the flows from X to T and from S to Y may be increased by the same amount. This amount is the minimum of the flow from X to Y and the differences between capacity and flow in the paths from S to Y and X to T.

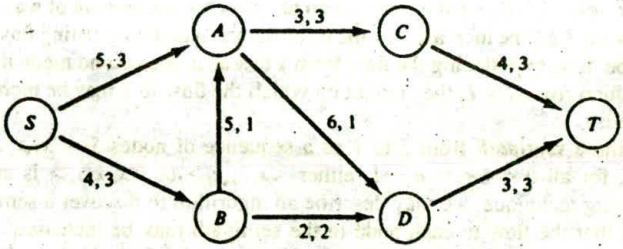
These two methods may be combined by proceeding through the graph from S to T as follows: The amount of water emanating from S toward T can be increased by any amount (since we have assumed no limit on the amount that can be produced by the source) only if the pipes from S to T can carry the increase. Suppose that the pipe capacity from S to x allows the amount of water entering x to be increased by an amount a. If the pipe capacity to carry the increase from x to T exists then the increase can be made. Then if a node y is adjacent to x (that is, there is an arc $<x,y>$), the amount of water emanating from y toward T can be increased by the minimum of a and the unused capacity of arc $<x,y>$. This is an application of the first method. Similarly, if node x is



(a)

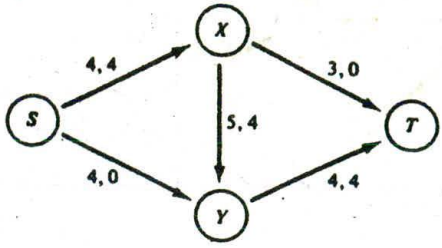


(b)

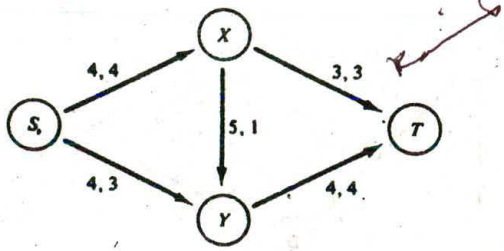


(c)

Figure 8.2.2 Increasing the flow in a graph.



(a)



(b)

Figure 8.2.3 Increasing the flow in a graph.

adjacent to some node y (that is, there is an arc $<y,x>$), the amount of water emanating from y toward T can be increased by the minimum of a and the existing flow from y to x. This can be done by reducing the flow from y to x as in the second method. Proceeding in this fashion from S to T, the amount by which the flow to T may be increased can be determined.

Define a ***semipath*** from S to T as a sequence of nodes $S = x_1,x_2, \ldots ,x_n = T$ such that, for all $0 < i <= n - 1$, either $<x_{i-1},x_i>$ or $<x_i,x_{i-1}>$ is an arc. Using the foregoing technique, we may describe an algorithm to discover a semipath from S to T such that the flow to each node in the semipath may be increased. This is done by building upon already discovered partial semipaths from S. If the last node in a discovered partial semipath from S is a, the algorithm considers extending it to any node b such that either $<a,b>$ or $<b,a>$ is an arc. The partial semipath is extended to b only if the extension can be made in such a way that the inflow to b can be increased. Once a partial semipath has been extended to a node b, that node is removed from consideration as an extension of some other partial semipath. (This is because at this point we are trying to discover a single semipath from S to T.) The algorithm of course keeps track of the amount by which the inflow to b may be increased and whether its increase is due to consideration of the arc $<a,b>$ or $<b,a>$.

This process continues until some partial semipath from S has been completed by extending it to T. The algorithm then proceeds backward along the semipath adjusting all flows until S is reached. (This will be illustrated shortly with an example.) The entire process is then repeated in an attempt to discover yet another such semipath from S to T. When no partial semipath may be successfully extended, the flow cannot be increased and the existing flow is optimal. (You are asked to prove this as an exercise.)

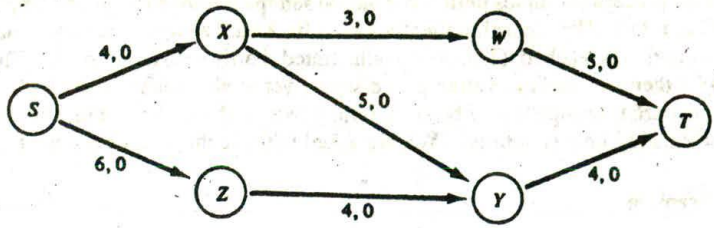
Example

Let us illustrate this process with an example. Consider the arcs and capacities of the weighted graph of Figure 8.2.4. We begin by assuming a flow of 0 and attempt to discover an optimal flow. Figure 8.2.4a illustrates the initial situation. The two numbers above each arc represent the capacity and current flow respectively. We may extend a semipath from S to (S,X) and (S,Z), respectively. The flow from S to X may be increased by 4, and the flow from S to Z may be increased by 6. The semipath (S,X) may be extended to (S,X,W) and (S,X,Y) with corresponding increases of flow to W and Y of 3 and 4, respectively. The semipath (S,X,Y) may be extended to (S,X,Y,T) with an increase of flow to T of 4. (Note that at this point we could have chosen to extend (S,X,W) to (S,X,W,T). Similarly we could have extended (S,Z) to (S,Z,Y) rather than (S,X) to (S,X,W) and (S,X,Y). These decisions are arbitrary.)

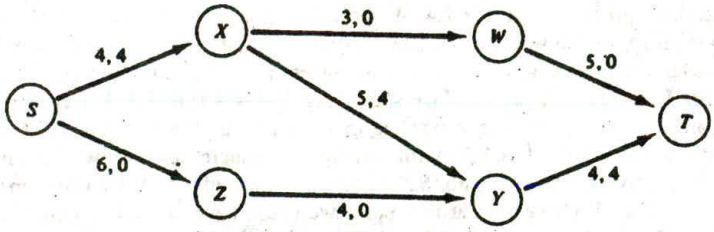
Since we have reached T by the semipath (S,X,Y,T) with a net increase of 4, we increase the flow along each forward arc of the semipath by this amount. The results are depicted in Figure 8.2.4b.

We now repeat the foregoing process with the flow of Figure 8.2.4b. (S) may be extended to (S,Z) only, since the flow in arc $<S,X>$ is already at capacity. The net increase to Z through this semipath is 6. (S,Z) may be extended to (S,Z,Y), yielding a net increase of 4 to Y. (S,Z,Y) cannot be extended to (S,Z,Y,T), since the flow in arc $<Y,T>$ is at capacity. However, it can be extended to (S,Z,Y,X) with a net increase to node X of 4. Note that since this semipath includes a backward arc $<Y,X>$, it implies a reduction in the flow from X to Y of 4. The semipath (S,Z,Y,X) may be extended to (S,Z,Y,X,W) with a net increase of 3 (the unused capacity of $<X,W>$) to W. This semipath may then be extended to (S,Z,Y,X,W,T) with a net increase of 3 in the flow to T. Since we have reached T with an increase of 3, we proceed backward along this semipath. Since $<W,T>$ and $<X,W>$ are forward arcs, their flow may each be increased by 3. Since $<Y,X>$ is a backward arc, the flow along $<X,Y>$ is reduced by 3. Since $<Z,Y>$ and $<S,Z>$ are forward arcs, their flow may be increased by 3. This results in the flow shown in Figure 8.2.4c.

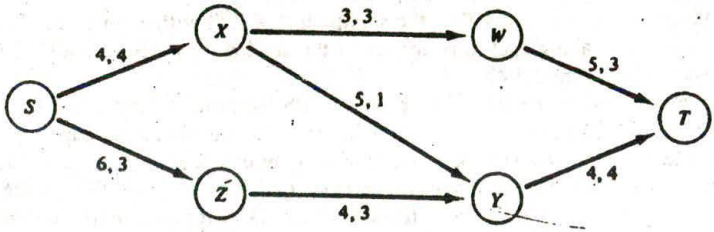
We then attempt to repeat the process. (S) may be extended to (S,Z) with an increase of 3 to Z, (S,Z) may be extended to (S,Z,Y) with an increase of 1 to Y, and (S,Z,Y) may be extended to (S,Z,Y,X) with an increase of 1 to X. However, since arcs $<S,X>$, $<Y,T>$ and $<X,W>$ are at capacity, no semipath may be extended further and an optimum flow has been found. Note that this optimum flow need not be unique. Figure 8.2.4d illustrates another optimum flow for the same graph which was obtained from Figure 8.2.4a by considering the semipaths (S,X,W,T) and (S,Z,Y,T).



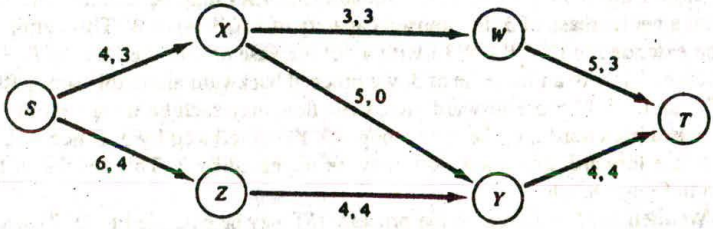
(a)



(b)



(c)



(d)

Figure 8.2.4 Producing an optimum flow.

Algorithm and Program

Given a weighted graph (an adjacency matrix and a capacity matrix) with a source S and a sink T, the algorithm to produce an optimum flow function for that graph may be outlined as follows:

```
1     initialize the flow function to 0 at each arc;
2     canimprove = TRUE;
3     do {
4        attempt to find a semipath from S to T that
                          increases the flow to T by x > 0;
5        if (a semipath cannot be found)
            canimprove = FALSE;
6        else
            increase the flow to each node (except S)
                                        in the semipath by x;
7     } while (canimprove == TRUE);
```

Of course, the heart of the algorithm lies in line 4. Once a node has been placed on a partial semipath, it can no longer be used to extend a different semipath. Thus the algorithm uses an array of flags *onpath* such that *onpath*[*node*] indicates whether or not *node* is on some semipath. It also needs an indication of which nodes are at the ends of partial semipaths so that such partial semipaths can be extended by adding adjacent nodes. *endpath*[*node*] indicates whether or not *node* is at the end of a partial semipath. For each node on a semipath the algorithm must keep track of what node precedes it on that semipath and the direction of the arc. *precede*[*node*] points to the node that precedes *node* on its semipath, and *forward*[*node*] has the value *TRUE* if and only if the arc is from *precede*[*node*] to *node*. *improve*[*node*] indicates the amount by which the flow to *node* may be increased along its semipath. The algorithm that attempts to find a semipath from S to T along which the flow may be increased may be written as follows. (We assume that $c[a][b]$ is the capacity of the pipe from a to b and that $f[a][b]$ is the current flow from a to b.)

```
set endpath[node], onpath[node] to FALSE for all nodes;
endpath[S] = TRUE;
onpath[S] = TRUE;
/* compute maximum flow from S that pipes can carry */
improve[S] = sum of c[S][node] over all nodes node;
while ((onpath[T] == FALSE)
      && (there exists a node nd such that endpath[nd] == TRUE)) {
   endpath[nd] = FALSE;
   while (there exists a node i such that
         (onpath[i] == FALSE) && (adjacent(nd,i) == TRUE)
         && (f[nd][i] < c[nd][i])) {
      /*  the flow from nd to i may be increased  */
      /*          place i on the semipath         */
```

```
        onpath[i] = TRUE;
        endpath[i] = TRUE;
        precede[i] = nd;
        forward[i] = TRUE;
        x = c[nd][i] - f[nd][i];
        improve[i] = (improve[nd] < x) ? improve[nd] : x;
      } /* end while there exists... */
      while (there exists a node i such that (onpath[i] == FALSE)
            && (adjacent(i,nd) == TRUE) && (f[i][nd] > 0)) {
        /*  the flow from i to nd may be decreased  */
        /*          place i on the semipath          */
        onpath[i] = TRUE;
        endpath[i] = TRUE;
        precede[i] = nd;
        forward[i] = FALSE;
        improve[i] = (improve[nd] < f[i][nd]) ? improve[nd] :
                                                f[i][nd];
      } /* end while there exists... */
    } /* end while (onpath[T] == FALSE) */
    if (onpath(T) == TRUE)
      we have found a semipath from S to T;
    else
      the flow is already optimum;
```

Once a semipath from S to T has been found, the flow may be increased along that semipath (line 6 above) by the following algorithm:

```
x = improve[T];
nd = T;
while (nd != S) {
   pred = precede[ND];
   (forward[nd] == TRUE) ? (f[pred, nd] += x) : (f[nd, pred] -= x);
   nd = pred;
} /* end while */
```

This method of solving the flow problem is known as the ***Ford–Fulkerson algorithm*** after its discoverers.

Let us now convert these algorithms into a C routine *maxflow*(*cap, s, t, flow, totflow*), where *cap* is an input parameter representing a capacity function defined on a weighted graph. *s* and *t* are input parameters representing the source and sink, *flow* is an output parameter representing the maximum flow function, and *totflow* is the amount of flow from *s* to *t* under the flow function *flow*.

The previous algorithms may be converted easily into C programs. Five arrays *endpath, forward, onpath, improve*, and *precede* are required. The question of whether j is adjacent to i can be answered by checking whether or not $cap[i][j] == 0$.

We present the routine here as a straightforward implementation of the algorithms. *any* is a function that accepts an array of logical values and returns *TRUE* if any element of the array is *TRUE*. If none of the elements of the array is *TRUE*, *any* returns *FALSE*. We leave its coding as an exercise.

```
#define  MAXNODES  50
#define  INFINITY ....

int any(int []);

void maxflow (int cap[][MAXNODES], int s, int t,
                        int flow[][MAXNODES], int *ptotflow)
{
   int pred, nd, i, x;
   int precede[MAXNODES], improve[MAXNODES];
   int endpath[MAXNODES], forward[MAXNODES], onpath[MAXNODES];

   for (nd = 0; nd < MAXNODES; ++nd)
      for (i = 0; i < MAXNODES; ++i)
         flow[nd][i] = 0;
   *ptotflow = 0;
   do {
      /* attempt to find a semipath from s to t */
      for (nd = 0; nd < MAXNODES; ++nd) {
         endpath[nd] = FALSE;
         onpath[nd] = FALSE;
      } /* end for */
      endpath[s] = TRUE;
      onpath[s] = TRUE;
      improve[s] = INFINITY;
      /* we assume that s can provide infinite flow */
      while ((onpath[t] == FALSE) && (any(endpath) == TRUE)) {
         /* attempt to extend an existing path */
         for (nd = 0; endpath[nd] == FALSE; nd++)
               ;
         endpath[nd] = FALSE;
         for (i = 0; i < MAXNODES; ++i) {
            if ((flow[nd][i] < cap[nd][i]) && (onpath[i] == FALSE)) {
               onpath[i] = TRUE;
               endpath[i] = TRUE;
               precede[i] = nd;
               forward[nd] = TRUE;
               x = cap[nd][i] - flow[nd][i];
               improve[i] = (improve[nd] < x) ? improve[nd] : x;
            } /* end if */
            if ((flow[i][nd] > 0) && (onpath[i] == FALSE)) {
               onpath[i] = TRUE;
               endpath[i] = TRUE;
```

```
            precede[i] = nd;
            forward[nd] = FALSE;
            improve[i] = (improve[nd] < flow[i][nd]) ?
                                        improve[nd] : flow[i][nd];
         } /* end if */
      } /* end for */
   } /* end while */

   if (onpath[t] == TRUE) {
      /* flow on semipath to t can be increased */
      x = improve[t];
      *ptotflow += x;
      nd = t;
      while (nd != s) {
         /* travel back along path */
         pred = precede[nd];
         /* increase or decrease flow from pred */
         (forward[pred] == TRUE) ? (flow[pred][nd] += x):
                                   (flow[nd][pred] -= x);
         nd = pred;
      } /* end while */
   } /* end if */
} while (onpath[t] == TRUE);   /* end do */
} /* end maxflow */
```

Note that although we have maintained the arrays as they were specified in the algorithm, we could have eliminated the array *forward* by setting *precede*[*nd*] to a positive number in the case of a forward arc and to a negative number in the case of a backward arc. You are asked to pursue this possibility as an exercise.

For large graphs with many nodes, the arrays *improve* and *endpath* may be prohibitively expensive in terms of space. Furthermore, a search through all nodes to find a node *nd* such that *endpath*[*nd*] = *TRUE* may be very inefficient in terms of time. An alternate solution might be to note that the value of *improve* is required only for those nodes *nd* such that *endpath*[*nd*] = *TRUE*. Those graph nodes which are at the end of semipaths may be kept in a list whose nodes are declared by

```
struct listnode {
   int graphnode;
   int improve;
   int next;
};
```

When a node which is at the end of a semipath is required, remove the first element from the list. We can similarly dispense with the array *precede* by maintaining a separate list of nodes for each semipath. However, this suggestion is of dubious value since almost all nodes will be on some semipath. You are invited to write the routine *maxflow* as an exercise using these suggestions to save time and space.

EXERCISES

8.2.1. Find the maximum flows for the graphs in Figure 8.2.1 using the Ford–Fulkerson method (the capacities are shown next to the arcs).

8.2.2. Given a graph and a capacity function as in this section, define a ***cut*** as any set of nodes x containing S but not T. Define the ***capacity of the cut*** x as the sum of the capacities of all the arcs leaving the set x.

(a) Show that for any flow function f, the value of the total flow v is less than or equal to the capacity of any cut.

(b) Show that equality in (a) is achieved when the flow is maximum and the cut has minimum capacity.

8.2.3. Prove that the Ford–Fulkerson algorithm produces an optimum flow function.

8.2.4. Rewrite the routine *maxflow* using a linked list to contain nodes at the end of semipaths, as suggested in the text.

8.2.5. Assume that in addition to a capacity function for every arc, there is also a cost function, *cost*. *cost*(*a*,*b*) is the cost of each unit of flow from node *a* to node *b*. Modify the program of the text to produce the flow function which maximizes the total flow from source to sink at the lowest cost (that is, if there are two flow functions, both of which produce the same maximum flow, choose the one with the least cost).

8.2.6. Assuming a cost function as in the previous exercise, write a program to produce the maximum cheapest flow—that is, a flow function such that the total flow divided by the cost of the flow is greatest.

8.2.7. A ***probabilistic*** directed graph is one in which a probability function associates a probability with each arc. The sum of the probabilities of all arcs emanating from any node is 1. Consider an acyclic probabilistic digraph representing a tunnel system. A man is placed at one node in the tunnel. At each node he chooses to take a particular arc to another node with probability given by the probability function. Write a program to compute the probability that the man passes through each node of the graph. What if the graph were cyclic?

8.2.8. Write a C program that reads the following information about an electrical network:

1. n, the number of wires in the network
2. The amount of current entering through the first wire and leaving through the nth
3. The resistance of each of the wires 2 through $n - 1$
4. A set of ordered pairs $<i,j>$ indicating that wire i is connected to wire j and that electricity flows through wire i to wire j

The program should compute the amount of current flowing through each of wires 2 though $n - 1$ by applying Kirchhoff's law and Ohm's law. Kirchhoff's law states that the amount of current flowing into a junction equals the amount leaving a junction. Ohm's law states that if two paths exist between two junctions, the sums of the currents times the resistances over all wires in the two paths are equal.

8.3 LINKED REPRESENTATION OF GRAPHS

The adjacency matrix representation of a graph is frequently inadequate because it requires advance knowledge of the number of nodes. If a graph must be constructed in the course of solving a problem, or if it must be updated dynamically as the program

proceeds, a new matrix must be created for each addition or deletion of a node. This is prohibitively inefficient, especially in a real-world situation where a graph may have a hundred or more nodes. Further, even if a graph has very few arcs so that the adjacency matrix (and the weight matrix for a weighted graph) is sparse, space must be reserved for every possible arc between two nodes, whether or not such an arc exists. If the graph contains n nodes, a total of n^2 locations must be used.

As you might expect, the remedy is to use a linked structure, allocating and freeing nodes from an available pool. This is similar to the methods used to represent dynamic binary and general trees. In the linked representation of trees, each allocated node corresponds to a tree node. This is possible because each tree node is the son of only one other tree node and is therefore contained in only a single list of sons. However, in a graph an arc may exist between any two graph nodes. It is possible to keep an adjacency list for every node in a graph (such a list contains all nodes adjacent to a given node) and a node might find itself on many different adjacency lists (one for each node to which it is adjacent). But this requires that each allocated node contain a variable number of pointers, depending on the number of nodes to which it is adjacent. This solution is clearly impractical as we saw in attempting to represent general trees with nodes containing pointers to each of its sons.

An alternative is to construct a multilinked structure in the following way. The nodes of the graph (hereafter referred to as ***graph nodes***) are represented by a linked list of ***header nodes***. Each such header node contains three fields: *info*, *nextnode*, and *arcptr*. If p points to a header node representing a graph node a, *info*(p) contains any information associated with graph node a. *nextnode*(p) is a pointer to the header node representing the next graph node, if any. Each header node is at the head of a list of nodes of a second type called ***list nodes***. This list is called the ***adjacency list***. Each node on an adjacency list represents an arc of the graph. *arcptr*(p) points to the adjacency list of nodes representing the arcs emanating from the graph node a.

Each adjacency list node contains two fields: *ndptr* and *nextarc*. If q points to a list node representing an arc $<A,B>$, *ndptr*(q) is a pointer to the header node representing the graph node B. *Nextarc*(q) points to a list node representing the next arc emanating from graph node A, if any. Each list node is contained in a single adjacency list representing all arcs emanating from a given graph node. The term ***allocated node*** is used to refer to either a header or a list node of a multilinked structure representing a graph. We also refer to an adjacency list node as an ***arc node***.

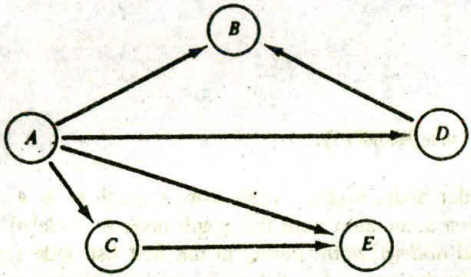
Figure 8.3.1 illustrates this representation. If each graph node carries some information but (since the graph is not weighted) the arcs do not, two types of allocated nodes are needed: one for header nodes (graph nodes) and the other for adjacency list nodes (arcs). These are illustrated in Figure 8.3.1a. Each header node contains an *info* field and two pointers. The first of these is to the adjacency list of arcs emanating from the graph node, and the second is to the next header node in the graph. Each arc node contains two pointers, one to the next arc node in the adjacency list and the other to the header node representing the graph node that terminates the arc. Figure 8.3.1b depicts a graph and Figure 8.3.1c its linked representation.

Note that header nodes and list nodes have different formats and must be represented by different structures. This necessitates either keeping two distinct available lists or defining a union. Even in the case of a weighted graph in which each list node

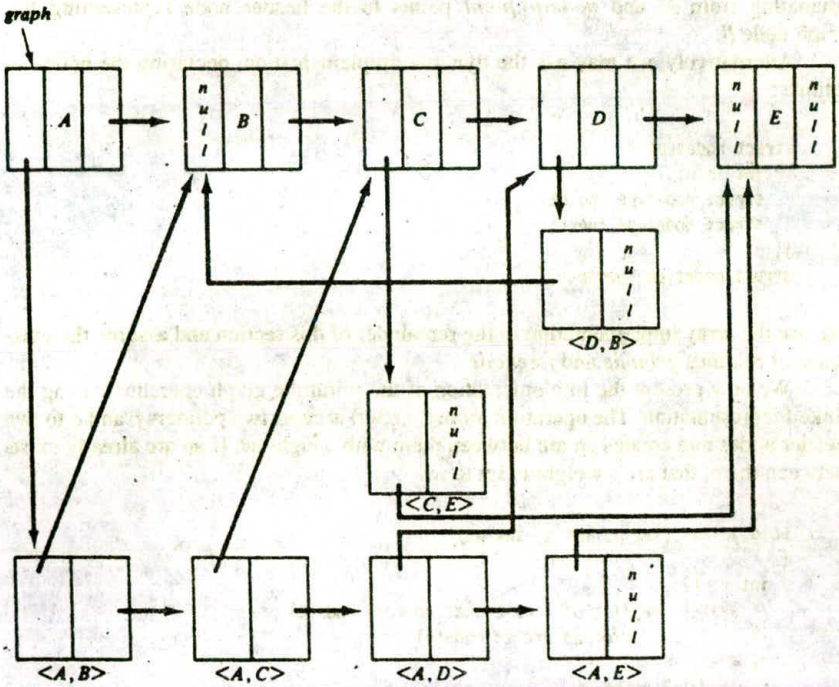
A sample header node representing a graph node.

A sample list node representing an arc.

(a)



(b) A graph.



(c) Linked representation of a graph.

Figure 8.3.1 Linked representation of a graph.

contains an *info* field to hold the weight of an arc, two different structures may be necessary if the information in the header nodes is not an integer. However, for simplicity we make the assumption that both header and list nodes have the same format and contain two pointers and a single integer information field. These nodes are declared using the array implementation as

```
#define MAXNODES 500

struct nodetype {
   int info;
   int point;
   int next;
};
struct nodetype node[MAXNODES];
```

In the case of a header node, *node*[*p*] represents a graph node *A*, *node*[*p*].*info* represents the information associated with the graph node *A*, *node*[*p*].*next* points to the next graph node, and *node*[*p*].*point* points to the first list node representing an arc emanating from *A*. In the case of a list node, *node*[*p*] represents an arc <*A*,*B*>, *node*[*p*].*info* represents the weight of the arc, *node*[*p*].*next* points to the next arc emanating from *A*, and *node*[*p*].*point* points to the header node representing the graph node *B*.

Alternatively, we may use the dynamic implementation, declaring the nodes as follows:

```
struct nodetype {
   int info;
   struct nodetype *point;
   struct nodetype *next;
};
struct nodetype *nodeptr;
```

We use the array implementation in the remainder of this section and assume the existence of routines *getnode* and *freenode*.

We now present the implementation of the primitive graph operations using the linked representation. The operation *joinwt*(*p*,*q*,*wt*) accepts two pointers *p* and *q* to two header nodes and creates an arc between them with weight *wt*. If an arc already exists between them, that arc's weight is set to *wt*.

```
void joinwt (int p, int q, int wt)
{
  int r, r2;
  /* search the list of arcs emanating from node[p]  */
  /*              for an arc to node[q]              */
  r2 = -1;
  r = node[p].point;
```

```
  while (r >= 0 && node[r].point != q) {
    r2 = r;
    r = node[r].next;
  } /* end while */
  if (r >= 0) {
     /* node[r] represents an arc from */
     /*       node[p] to node[q]       */
     node[r].info = wt;
     return;
  } /* end if */
  /* an arc from node[p] to node[q] does not     */
  /*     exist. Such an arc must be created.     */
  r = getnode();
  node[r].point = q;
  node[r].next = -1;
  node[r].info = wt;
  (r2 < 0) ? (node[p].point = r) : (node[r2].next = r);
} /* end joinwt */
```

We leave the implementation of the operation *join* for an unweighted graph as an exercise for the reader. The operation *remv(p,q)* accepts pointers to two header nodes and removes the arc between them, if one exists.

```
void remv (int p, int q)
{
   int r, r2;

   r2 = -1;
   r = node[p].point;
   while (r >= 0 && node[r].point != q) {
      r2 = r;
      r = node[r].next;
   } /* end while */
   if (r >= 0) {
      /* r points to an arc from node[p]  */
      /*         to node[q]               */
      (r2 < 0) ? (node[p].point = node[r].next):
                 (node[r2].next = node[r].next);
      freenode(r);
      return;
   } /* end if */
   /*  if no arc has been found, then no action */
   /*           need be taken                   */
} /* end remv */
```

We leave the implementation of the operation *remvwt(p,q,x)*, which sets *x* to the weight of the arc <*p,q*> in a weighted graph and then removes the arc from the graph, as an exercise for the reader.

The function *adjacent(p,q)* accepts pointers to two header nodes and determines whether *node(q)* is adjacent to *node(p)*.

```
int adjacent (int p, int q)
{
   int r;

   r = node[p].point;
   while (r >= 0)
      if (node[r].point == q)
         return(TRUE);
      else
         r = node[r].next;
   return (FALSE);
} /* end adjacent */
```

Another useful function is *findnode(graph, x)* which returns a pointer to a header node with information field *x* if such a header node exists, and returns the null pointer otherwise.

```
int findnode (int graph, int x)
{
   int p;

   p = graph;
   while (p >= 0)
      if (node[p].info == x)
         return(p);
      else
         p = node[p].next;
   return (-1);
} /* end findnode */
```

The function *addnode(&graph, x)* adds a node with information field *x* to a graph and returns a pointer to that node.

```
int addnode (int *pgraph, int x)
{
   int p;

   p = getnode();
   node[p].info = x;
   node[p].point = -1;
   node[p].next = *pgraph;
   *pgraph = p;
   return (p);
  /* end addnode */
```

The reader should be aware of another important difference between the adjacency matrix representation and the linked representation of graphs. Implicit in the matrix representation is the ability to traverse a row or column of the matrix. Traversing a row is equivalent to identifying all arcs emanating from a given node. This can be done efficiently in the linked representation by traversing the list of arc nodes starting at a given header node. Traversing a column of an adjacency matrix, however, is equivalent to identifying all arcs that terminate at a given node; there is no corresponding method for accomplishing this under the linked representation. Of course, the linked representation could be modified to include two lists emanating from each header node: one for the arcs emanating from the graph node and the other for the arcs terminating at the graph node. However, this would require allocating two nodes for each arc, thus increasing the complexity of adding or deleting an arc.

Alternatively, each arc node could be placed on two lists. In this case, an arc node would contain four pointers: one to the next arc emanating from the same node, one to the next arc terminating at the same node, one to the header node at which it terminates and one to the header node from which it emanates. A header node would contain three pointers: one to the next header node, one to the list of arcs emanating from it and one to the list of arcs terminating at it. The programmer must, of course, choose from among these representations by examining the needs of the specific problem and considering both time and storage efficiency.

We invite the reader to write a routine *remvnode*(*graph*, *p*) that removes a header node pointed to by *p* from a graph pointed to by *graph* using the various graph representations outlined in the foregoing. Of course, when a node is removed from a graph all arcs emanating and terminating at that node must also be removed. In the linked representation which we have presented there is no easy way of removing a node from a graph since the arcs terminating at the node cannot be obtained directly.

Dijkstra's Algorithm Revisited

In Section 8.1 we presented an implementation of Dijkstra's algorithm for finding the shortest path between two nodes in a weighted graph represented by a weight matrix. That implementation was $O(n^2)$, where n is the number of nodes in the graph. We now show how the algorithm can be implemented more efficiently in most cases if the graph is implemented using adjacency lists.

We suggest review of the algorithm described in Section 8.1. This algorithm may be outlined as follows. We seek a shortest path from *s* to *t*. **pd* is to be set to the shortest distance; *precede*[*i*] to the node preceding node *i* in the shortest path:

```
1   for (all nodes i) {
2      distance[i] = INFINITY;
3      perm[i] = NONMEMBER;
4   }
5   perm[s] = MEMBER;
6   distance[s] = 0;
7   current = s;
```

```
8    while (current != t) {
9       dc = distance[current];
10      for (all nodes i that are successors of current)  {
11         newdist = dc + weight[current][i];
12         if (newdist < distance[i]) {
13            distance[i] = newdist;
14            precede[i] = current;
15         } /* end if */
16      } /*  end for    */
17      k = the node k such that perm[k] == NONMEMBER and
                          such that distance[k] is smallest;
18      current = k;
19      perm[k] = MEMBER;
20   } /* end  while */
21   *pd = distance[t];
```

Review how this algorithm is implemented in Section 8.1. Note especially how finding the minimum distance (line 17) is incorporated into the ***for*** loop and how that loop is implemented.

The keys to an efficient implementation are lines 10 and 17. In Section 8.1, where we had access only to a weight matrix, there is no way to limit the access to the successors of *current* as specified in line 10. It is necessary to traverse all n nodes of the graph each time the inner loop is repeated. We are able to increase efficiency by looking only at elements not in *perm*, but that cannot speed things up by more than a constant factor. Once an $O(n)$ inner loop is required, we may as well use it to compute the minimum as well (line 17).

However, given an adjacency list representation of the graph, it is possible to traverse directly all nodes adjacent to *current* without examining all graph nodes. Therefore, the total number of nodes i examined in the loop headed by line 10 is $O(e)$, where e is the number of edges in the graph. [Note that we are not saying that each execution of the inner loop is $O(e)$ but that the total of all repetitions of all passes of the inner loop is $O(e)$.] In most graphs, e is far smaller than n^2, so this is quite an improvement.

However, we are not yet done. Since we are eliminating a traversal through all nodes, we must find an alternative way of implementing line 17 to find the node with the smallest distance. If the best we can do in finding this minimum distance is $O(n)$, the entire process remains $O(n^2)$.

Fortunately, there is a solution. Suppose that, instead of maintaining the array *perm*, we maintained its complement, *notperm*. Then line 3 would become

```
3          notperm[i] = MEMBER;
```

line 5 would become

```
5          notperm[s] = NONMEMBER;
```

line 17 would become

```
17          k = the node k such that notperm[k] == MEMBER and
                        such that distance[k] is smallest;
```

and line 19 would become

```
19          notperm[k] = NONMEMBER;
```

The operations performed on the array *notperm* are creation [line 5; this may be $O(n)$ but it is outside the ***while*** loop and therefore does not hurt the overall efficiency], finding the minimum element (line 17), and deleting the minimum element (line 19). But these latter two operations can be combined into the single *pqmindelete* operation of an ascending priority queue and, by now, we have a number of ways of implementing that operation in less than $O(n)$. In fact, we can implement *pqmindelete* in $O(\log n)$ by using an ascending heap, a balanced binary tree, or a 2-3 tree. If the set *notperm* is implemented as a priority queue using one of these techniques, the efficiency of n such operations is $O(n \log n)$. If a priority queue ordered by the value of distance is used to implement *notperm*, the position of i must be adjusted in the priority queue whenever *distance*[i] is modified in line 13. Fortunately, this can also be done in $O(\log n)$ steps.

Thus Dijkstra's algorithm can be implemented using $O((e + n)\log n)$ operations, which is significantly better than $O(n^2)$ for sparse graphs (that is, graphs with very few edges as opposed to dense graphs that have an edge between almost every pair of nodes). We leave an actual C implementation as an exercise for the reader.

Organizing the Set of Graph Nodes

In many applications, the set of graph nodes (as implemented by header nodes) need not be organized as a simple linked list. The linked list organization is suitable only when the entire set of graph nodes must be traversed and when graph nodes are being dynamically inserted. Both of these operations are highly efficient on a linked list.

If graph nodes must also be deleted, the list must be doubly linked. In addition, as noted earlier, there is the need to ensure that no arcs emanate or terminate at a deleted node or that all such arcs are deleted as part of the node deletion routine. If we choose merely to ensure that no arcs terminate in a node being deleted rather than to delete any such arcs, it is not necessary to keep with each node a list of arcs terminating at the node. It is only necessary to maintain a count field in the node to hold the number of arcs terminating at the node; when count becomes 0 (and no arcs terminate at the node), the node may be deleted.

If graph nodes are not being added or deleted, the nodes can be kept in a simple array, where each array element contains any necessary information about the node plus a pointer to an adjacency list of arcs. Each arc need contain only an array index to indicate the position of its terminating node in the array.

In many applications, graph nodes must be accessed by their contents. For example, in a graph whose nodes represent cities, an application must find the appropriate node given the name of the city. If a linked list is used to represent the graph nodes, the entire list must be traversed to find the node associated with a particular name.

The problem of finding a particular element in a set based on its contents, or value, is one that we have already studied in great detail: it is simply the searching problem. And we know a great many possible solutions; binary search trees, multiway search trees, and hash tables are all ways of organizing sets to permit rapid searching.

The set of graph nodes can be organized in any of these ways. The particular organization chosen depends on the detailed needs of the application. In Dijkstra's algorithm, for example, we have just seen an illustration where the set of graph nodes could be organized as an array that implements an ascending heap used as a priority queue. Let us now look at a different application. We introduce it with a frivolous example, but the application itself is quite important.

Application to Scheduling

Suppose a chef in a diner receives an order for a fried egg. The job of frying an egg can be decomposed into a number of distinct subtasks:

Get egg	Crack egg	Get grease
Grease pan	Heat grease	Pour egg into pan
Wait until egg is done		Remove egg

Some of these tasks must precede others (for example, "get egg" must precede "crack egg"). Others may be done simultaneously (for example, "get egg" and "heat grease"). The chef wishes to provide the quickest service possible and is assumed to have an unlimited number of assistants. The problem is to assign tasks to the assistants so as to complete the job in the least possible time.

Although this example may seem frivolous, it is typical of many real-world scheduling problems. A computer system may wish to schedule jobs to minimize turnaround time; a compiler may wish to schedule machine language operations to minimize execution time; or a plant manager may wish to organize an assembly line to minimize production time. All these problems are closely related and can be solved by the use of graphs.

Let us represent the above problem as a graph. Each node of the graph represents a subtask and each arc $<x,y>$ represents the requirement that subtask y cannot be performed until subtask x has been completed. This graph G is shown in Figure 8.3.2.

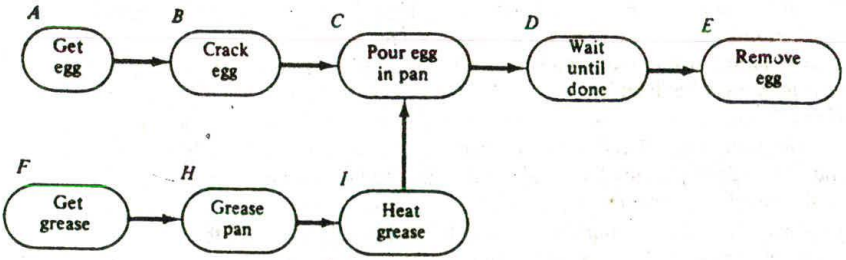


Figure 8.3.2 Graph *G*.

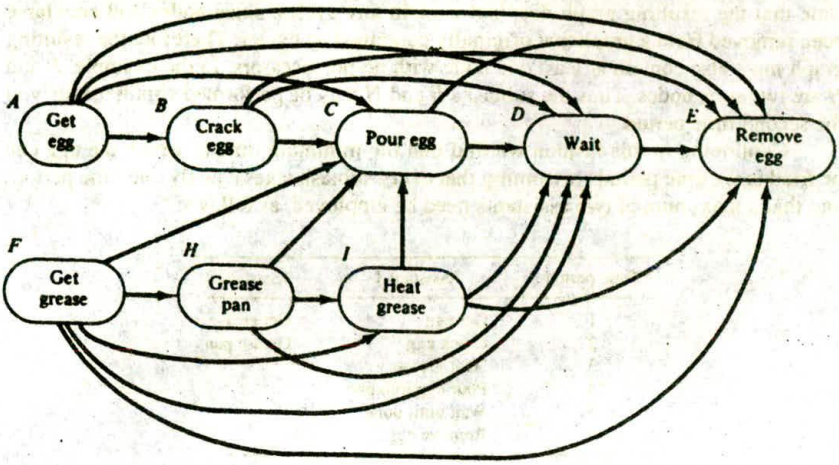


Figure 8.3.3 Graph *T*.

Consider the transitive closure of G. The transitive closure is the graph T such that $<x,y>$ is an arc of T if and only if there is a path from x to y in G. This transitive closure is shown in Figure 8.3.3.

In the graph T, an arc exists from node x to node y if and only if subtask x must be performed before subtask y. Note that neither G nor T can contain a cycle, since if a cycle from node x to itself existed, subtask x could not be performed until after subtask x had been completed. This is clearly an impossible situation in the context of the problem. Thus G is a **dag**, a directed acyclic graph.

Since G does not contain a cycle, there must be at least one node in G which has no predecessors. To see this suppose that every node in the graph did have a predecessor. In particular, let us choose a node z that has a predecessor y. y cannot equal z or the graph would have a cycle from z to itself. Since every node has a predecessor, y must also have a predecessor x that is not equal to either y or z. Continuing in this fashion, a sequence of distinct nodes

$$z,\ y,\ x,\ w,\ v,\ u,\ \ldots$$

is obtained. If any two nodes in this sequence were equal, a cycle would exist from that node to itself. However, the graph contains only a finite number of nodes so that eventually, two of the nodes must be equal. This is a contradiction. Thus there must be at least one node without a predecessor.

In the graphs of Figures 8.3.2 and 8.3.3, the nodes A and F do not have predecessors. Since they have no predecessors the subtasks that they represent may be performed immediately and simultaneously without waiting for any other subtasks to be completed. Every other subtask must wait until at least one of these is completed. Once these two subtasks have been performed, their nodes can be removed from the graph.

Note that the resulting graph does not contain any cycles, since nodes and arcs have been removed from a graph that originally contained no cycles. Therefore the resulting graph must also contain at least one node with no predecessors. In the example, *B* and *H* are two such nodes. Thus the subtasks *B* and *H* may be performed simultaneously in the second time period.

Continuing in this fashion we find that the minimum time in which the egg can be fried is six time periods (assuming that every subtask takes exactly one time period) and that a maximum of two assistants need be employed, as follows:

Time period	Assistant 1	Assistant 2
1	Get egg	Get grease
2	Crack egg	Grease pan
3	Heat grease	
4	Pour egg into pan	
5	Wait until done	
6	Remove egg	

The above process can be outlined as follows:

1. Read the precedences and construct the graph.
2. Use the graph to determine subtasks that can be done simultaneously.

Let us refine each of these two steps. Two crucial decisions must be made in refining step 1. The first is to decide the format of the input; the second is to decide on the representation of the graph. Clearly, the input must contain indications of which subtasks must precede others. The most convenient way to represent these requirements is by ordered pairs of subtasks; each input line contains the names of two subtasks where the first subtask on a line must precede the second. Of course, the data must be valid in the sense that no subtask may precede itself (no cycles are permitted in the graph). Only those precedences that are implied by the data and the transitive closure of the resulting graph are assumed to hold. A subtask may be represented by a character string such as "*get egg*" or by a number. We choose to represent subtasks by character strings in order that the input data reflect the real-world situation as closely as possible.

What information should be kept with each node of the graph? Clearly, the name of the subtask that the node represents is needed to locate the node associated with a particular task and for output purposes. This name will be kept as an array of single characters. The remaining information depends on how the graph is used. This will become apparent only after step 2 is refined. Here is a good example of how the various parts of a program outline interact with each other to produce a single unit.

Step 2 can be refined into the following algorithm:

```
while (the graph is not empty) {
   determine which nodes have no predecessors;
   output this group of nodes with an indication that they
      can be performed simultaneously in the next time period;
```

```
      remove these nodes and their incident arcs from the graph;
   } /* end while */
```

How can it be determined which nodes have no predecessors? One method is to maintain a *count* field in each node containing the number of nodes that precede it. Note that we are not interested in which nodes precede a given node—only in how many.

Initially, after the graph has been constructed, we examine all the graph nodes and place those with zero count on an output list. Then, during each simulated time period, the output list is traversed, each graph node on the list is output, and the adjacency list of arcs emanating from that graph node is traversed. For each arc, the count in the graph node that terminates the arc is reduced by 1, and if the count thereby becomes 0, that terminating graph node is placed on the output list of the next time period. At the same time, the arc node is freed.

The refinement of step 2 may then be rewritten as follows:

```
   /* traverse the set of graph nodes and place all those */
   /*    nodes with 0 count on the initial output list    */
1  outp = NULL;
2  for (all node(p) in the graph)
3     if (count(p) == 0) {
4        remove node(p) from the graph;
5        place node(p) on the output list;
6     } /* end if */
   /* simulate the time periods */
7  period = 0;
8  while (outp != NULL) {
9     ++period;
10    printf ("%d\n", period);
      /* initialize the next period's output list */
11    nextout = NULL;
      /* traverse the output list */
12    p = outp;
13    while (p != NULL) {
14       printf("%s", info(p));
         for (all arcs a emanating from node(p)) {
            /* reduce count in terminating */
            /*          node               */
16          t = the pointer to the node that terminates a;
17          count(t)--;
18          if (count(t) == 0) {
19             remove node(t) from the graph;
20             add node(t) to the nextout list;
21          } /* end if */
22          free arc (a)
23       } /* end for */
24       q = next(p);
25       free node(p);
```

```
26          p = q;
27       } /* end while p */
28       outp = nextout;
29    } /* end while */
30    if (any nodes remain in the graph)
31       error - there is a cycle in the graph;
```

We have been purposely vague in this algorithm about how the graph is implemented. Clearly, to efficiently process all arcs emanating from a node (lines 15 through 23), an adjacency list implementation is desired. But what of the set of graph nodes? Only a single traversal is required (lines 3 through 7) to initialize the output list. Thus the efficiency of this operation is not very crucial to the efficiency of the program.

It is necessary in step 1 to be able to access each graph node from the character string that specifies the task the node represents. For this reason, it makes sense to organize the set of graph nodes in a hash table. Although the initial traversal will require accessing some extra table positions, this is more than offset by the ability to access a node directly from its task name. The only impediment is the need (in line 19) to delete nodes from the graph.

However, further analysis reveals that the only reason to delete a node is to be able to check whether any nodes remain when the output list is empty (line 30) so that a cycle may be detected. If we maintain a counter of the number of nodes and implement the deletion by reducing this counter by 1, we can check for remaining nodes by comparing the counter with zero. (This is similar to using a count field rather than requiring a list of arcs terminating in a given node.) Having determined the data structures required, we are ready to transform the algorithm into a C program.

C Program

Let us first indicate the structure of the nodes that are required. The header nodes that represent graph nodes contain the following fields:

info	the name of the subtask represented by this node
count	the number of predecessors of this graph node
arcptr	a pointer to the list of arcs emanating from this node
nextnode	a pointer to the next node in the output list

Each list node representing an arc contains two pointers:

nodeptr	a pointer to its terminating node
nextarc	a pointer to the next arc in the adjacency list

Thus two types of nodes are required: one to represent graph nodes and one to represent arcs. These may be declared by

```
#define  MAXGRAPH  ...
#define  MAXARC    ...
```

```
struct graphtype {
   char info[20];
   int count;
   int arcpointer;
   int nextnode;
};
struct arctype {
   int nodeptr;
   int nextarc;
};
struct graphtype graphnode[MAXGRAPH];
struct arctype arc[MAXARC];
```

The array *graphnode* is a hash table, with rehashing used to resolve collisions. The array *arc* is a list of available arc nodes allocated by a routine *getarc* and freed by *freearc*. These manipulate an available pointer *availarc*.

We also assume the existence of a function *find*(*inf*) that searches *graphnode* for the presence of an element *nd* (that is, a graph node) such that *graphnode*[*nd*].*info* equals *inf*. If no such graph node exists, *find* allocates a previously empty position *nd*, sets *graphnode*[*nd*].*info* to *inf*, *graphnode*[*nd*].*count* to 0 and *graphnode*[*nd*].*arcptr* to -1, and increases the count of the number of nodes in the graph (which is maintained in a variable *numnodes*) by 1. In either case, *find* returns *nd*. Of course, *nd* is determined within *find* via functions *hash* and *rehash* applied to *inf*.

A routine *join* is also used. This routine accepts pointers to two graph nodes, *n*1 and *n*2, and allocates an arc node (using *getarc*) that is established as an arc from *graphnode*[*n*1] to *graphnode*[*n*2]. *join* is responsible for adding the arc node to the list of arcs emanating from *graphnode*[*n*1] as well as for increasing *graphnode*[*n*2].*count* by 1. Finally, the routines *strcpy* and *strcmp* are used to copy strings and compare them for equality.

We may now write a C scheduling program:

```
#include <string.h>
#define  MAXGRAPH  ...
#define  MAXARC    ...
#define  NULLTASK  ""
#define  TRUE      1
#define  FALSE     0
struct graphtype {
   char info[20];
   int count;
   int arcptr;
   int nextnode;
};

struct arctype  {
   int nodeptr;
   int nextarc;
};
```

```
struct graphtype graphnode[MAXGRAPH];
struct arctype arc[MAXARC];
int  availarc;
int  numnodes = 0;  /* number of graph nodes */
int find(char *);
int getarc(int, int);
int hash(int);
int rehash(int);
void join(int, int);
void freearc(int);
main()
{
   int p, q, r, s, t, outp, nextout, period;
   char inf1[20], inf2[20];
   /* initialize graph nodes and available list of arcs */
   for (p = 0; p < MAXGRAPH; ++p)
      strcpy(graphnode[p].info, NULLTASK);
   for (s = 0; s < MAXARC -1 ; ++s)
      arc[s].nextarc = s + 1;
   arc[MAXARC - 1].nextarc = -1;
   availarc = 0;
   while(scanf("%s %s", inf1, inf2) != EOF) {
      p = find(inf1);
      q = find(inf2);
      join(p,q);
   } /* end while */
   /* The graph has been constructed. Traverse the hash table and  */
   /*  place all graph nodes with zero count on the output list.   */
   outp = -1;
   for (p = 0; p < MAXGRAPH; ++p)
      if ((strcmp(graphnode[p].info, NULLTASK) == FALSE) &&
                                    (graphnode[p].count == 0)) {
         graphnode[p].nextnode = outp;
         outp = p;
      } /* end if */
   /* simulate the time periods */
   period = 0;
   while (outp != -1)   {
      ++period;
      printf("%d\n", period);
      /*   initialize output list for next period  */
      nextout = -1;
      /*   traverse the output list  */
      p = outp;
      while (p != -1)  {
         printf("%s\n", graphnode[p].info);
         r = graphnode[p].arcptr;
         /*   traverse the list of arcs  */
         while (r != -1) {
            s = arc[r].nextarc;
```

```
            t = arc[r].nodeptr;
            --graphnode[t].count;
            if (graphnode[t].count == 0) {
               /*   place graphnode[t] on the next  */
               /*      period's output list         */
               graphnode[t].nextnode = nextout;
               nextout = t;
            } /* end if */
            freearc(r);
            r = s;
         } /* end while */
         /*  delete the graph node  */
         strcpy(graphnode[p].info,NULLTASK);
         --numnodes;
         /*  continue traversing the output list  */
         p = graphnode[p].nextnode;
      } /* end while (p != -1) */
      /*  reset output list for the next period  */
      outp = nextout;
   } /* end while (outp != -1)  */
   if (numnodes != 0)
      error("error in input - graph contains a cycle\n");
} /* end schedule  */
```

EXERCISES

8.3.1. Implement a graph using linked lists so that each header node heads two lists: one containing the arcs emanating from the graph node and the other containing the arcs terminating at the graph node.

8.3.2. Implement a graph so that the lists of header nodes and arc nodes are circular.

8.3.3. Implement a graph using an adjacency matrix represented by the sparse matrix techniques of Section 8.1.

8.3.4. Implement a graph using an array of adjacency lists. Under this representation, a graph of n nodes consists of n header nodes, each containing an integer from 0 to $n - 1$ and a pointer. The pointer is to a list of list nodes each of which contains the node number of a node adjacent to the node represented by the header node. Implement Dijkstra's algorithm using this graph representation with the array formed into an ascending heap.

8.3.5. There may be more than one way to organize a set of subtasks in a minimum number of time periods. For example, the subtasks in Figure 8.3.2 may be completed in six time periods in one of three different methods:

Period	Method 1	Method 2	Method 3
1	*A,F*	*F*	*A,F*
2	*B,H*	*A,H*	*B*
3	*I*	*B,I*	*E*
4	*C*	*C*	*C*
5	*D*	*D*	*I*
6	*E*	*E*	*E*

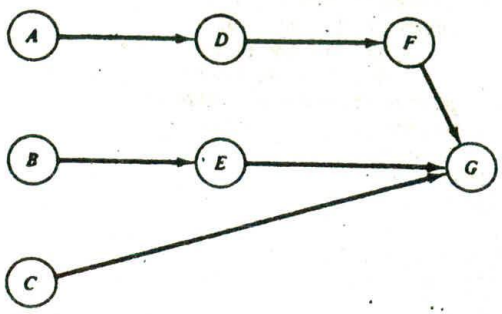


Figure 8.3.4

Write a program to generate all possible methods of organizing the subtasks in the minimum number of time periods.

8.3.6. Consider the graph of Figure 8.3.4. The program *schedule* outputs the following organization of tasks:

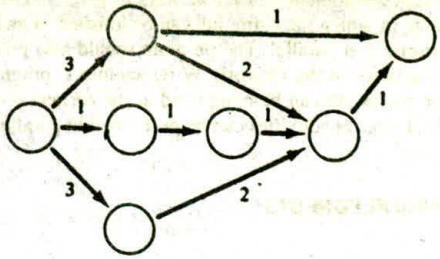
Time	Subtasks
1	*A,B,C*
2	*D,E*
3	*F*
4	*G*

This requires three assistants (for time period 1). Can you find a method of organizing the subtasks so that only two assistants are required at any time period, yet the entire job can be accomplished in the same four time periods? Write a program that organizes subtasks so that a minimum number of assistants are needed to complete the entire job in the minimum number of time periods.

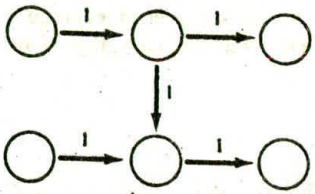
8.3.7. If there is only one worker available, it will take k time periods to complete the entire job, where k is the number of subtasks. Write a program to list a valid order in which the worker can perform the tasks. Note that this program is simpler than *schedule*, since an output list is not needed; as soon as the *count* field reaches 0 the task may be output. The process of converting a set of precedences into a single linear list in which no later element precedes an earlier one is called a ***topological sort***.

8.3.8. A ***PERT network*** is a weighted acyclic directed graph in which each arc represents an activity and its weight represents the time needed to perform that activity. If arcs $<a,b>$ and $<b,c>$ exist in the network, the activity represented by arc $<a,b>$ must be completed before the activity represented by $<b,c>$ can be started. Each node x of the network represents a time at which all activities represented by arcs terminating at x can be completed.

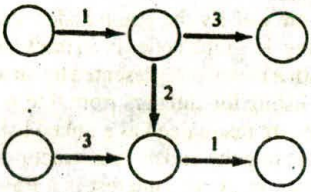
(a) Write a C routine that accepts a representation of such a network and assigns to each node x the earliest time that all activities terminating in that node can be completed. Call this quantity $et(x)$. [*Hint*: Assign time 0 to all nodes with no predecessors. If all predecessors of a node x have been assigned times, $et(x)$ is the maximum over



(a)



(b)



(c)

Figure 8.3.5 Some PERT networks.

all predecessors of the sum of the time assigned to a predecessor and the weight of the arc from that predecessor to x.]

(b) Given the assignment of times in part (a), write a C routine that assigns to each node x the latest time that all activities terminating in x can be completed without delaying the completion of all the activities. Call this quantity $lt(x)$. (*Hint*: Assign time $et(x)$ to all nodes x with no successors. If all successors of a node x have been assigned times, $lt(x)$ is the minimum over all successors of the difference between the time assigned to a successor and the weight of the arc from x to the successor.)

(c) Prove that there is at least one path in the graph from a node with no predecessors to a node with no successors such that $et(x) = lt(x)$ for every node x on the path. Such a path is called a ***critical path***.

(d) Explain the significance of a critical path by showing that reducing the time of the activities along every critical path reduces the earliest time by which the entire job can be completed.

(e) Write a C routine to find all critical paths in a PERT network.

(f) Find the critical paths in the networks of Figure 8.3.5.

8.3.9. Write a C program that accepts a repesentation of a PERT network as given in Exercise 8.3.8 and computes the earliest time in which the entire job can be finished, if as many activities as possible may be performed in parallel. The program should also print the starting and ending time of each activity in the network. Write another C program to schedule the activities so that the entire job can be completed at the earliest possible time subject to the constraint that at most m activities can be performed in parallel.

8.4 GRAPH TRAVERSAL AND SPANNING FORESTS

A great many algorithms depend on being able to traverse a graph. In this section we examine techniques for systematically accessing all the nodes of a graph and present several useful algorithms that implement and use those traversal techniques. We also look at ways of creating a general forest that is a subgraph of given graph G and contains all the nodes of G.

Traversal Methods for Graphs

It is often desirable to ***traverse*** a data structure, that is, to visit each of its elements in a systematic manner. We have already seen traversal techniques for lists and trees; we now examine traversal techniques for graphs.

The elements of the graph to be visited are usually the graph nodes. It is always possible to traverse a graph efficiently by visiting the graph nodes in an implementation-dependent manner. For example, if a graph with n nodes is represented by an adjacency matrix or an array of adjacency lists, simply listing the integers from 0 to $n - 1$ "traverses" the graph. Similarly, if the graph nodes are maintained as a linked list, a search tree, a hash table or some other structure, traversing the underlying structure might be considered a "traversal" of the graph. However, of greater interest is a traversal that corresponds to the graph structure of the object, not one for the underlying implementation structure. That is, the sequence in which the nodes are visited should relate to the adjacency structure of the graph.

Defining a traversal that relates to the structure of a graph is more complex than for a list or a tree for three reasons:

1. In general, there is no natural "first" node in a graph from which the traversal should start, as there is a first node in a list or a root in a tree. Further, once a starting node has been determined and all nodes reachable from that node have been visited, there may remain other nodes in the graph that have not been visited because they are not reachable from the starting node. This is again unlike a list or tree where every node is reachable from the header or the root. Thus, once all reachable nodes in a graph have been visited, the traversal algorithm again faces the problem of selecting another starting node.
2. There is no natural order among the successors of a particular node. Thus there is no a priori order in which the successors of a particular node should be visited.

3. Unlike a node of a list or a tree, a node of a graph may have more than one predecessor. If node x is a successor of both nodes y and z, x may be visited after y but before z. It is therefore possible for a node to be visited before one of its predecessors. In fact, if a graph is cyclic, every possible traversal must include some node that is visited before one of its predecessors.

To deal with these three complications, any graph traversal method incorporates the following three features:

1. The algorithm is either presented with a starting node for the traversal or chooses a random node at which to start. The same traversal algorithm produces a different ordering of the nodes depending on the node at which it starts. In the following discussion, s denotes the starting node.

We also assume a function *select* with no parameters that chooses an arbitrary unvisited node. The *select* operation is usually dependent on the graph representation. If the graph nodes are represented by the integers 0 to $n - 1$, *select* maintains a global variable *last* (initialized to -1) that keeps track of the last node selected by *select* and utilizes a flag *visited*(i) that is *true* only if *node*(i) has been visited. The following is an algorithm for *select*:

```
for (i = last + 1; i < n && visited(i); i++)
;
if (i == n)
   return(-1)
last = i;
return(i);
```

A similar *select* routine can be implemented if the graph nodes are organized as a linked list, with *last* being a pointer to the last header node selected.

2. Generally, the implementation of the graph determines the order in which the successors of a node are visited. For example, if the adjacency matrix implementation is used, the node numbering (from 0 to $n - 1$) determines the order; if the adjacency list implementation is used, the order of the arcs on the adjacency list determines the order in which the successors are visited. Alternatively, and much less commonly, the algorithm may choose a random ordering among the successors of a node. We consider two operations: *firstsucc*(x), which returns a pointer to the "first" successor of *node*(x), and *nextsucc*(x,y), where *node*(y) is a successor of *node*(x), which returns a pointer to the "next" successor of *node*(x) following *node*(y). Let us examine how to implement these functions under both the adjacency matrix and linked representations of a graph.

In the adjacency matrix representation, if x and y are indices such that *node*(y) is a successor of *node*(x), the next successor of x following y can be computed as the lowest index i greater than y such that *adj*(x,i) is *true*. Unfortunately, things are not so simple for the linked representation. If x and y represent two graph nodes in a graph representation that uses adjacency lists (x and y can be either array indices or pointers to header nodes), there is no way to access the "next" successor of *node*(x) following *node*(y). This is

because, in the adjacency list representation, the ordering of successors is based on the ordering of arc nodes. It is therefore necessary to locate the arc node following the arc node that points to *node*(*y*). But there is no reference from *node*(*y*) to the arc nodes that point to it, and therefore no way to get to the next arc node. It is therefore necessary for *y* to point to an arc node rather than a graph node, although the pointer actually represents the graph node terminating that arc [that is, *node*(*ndptr*(*y*))]. The next successor of *node*(*x*) following that graph node can then be found as *node*(*ndptr*(*nextarc*(*y*))), that is, the node that terminates the arc that follows the arc node *node*(*y*) on the adjacency list emanating from *node*(*x*).

To employ a uniform calling technique for *firstsucc* and *nextsucc* under all graph implementations, we present them as subroutines rather than functions:

> *firstsucc*(*x*, *yptr*, *ynode*) sets both *yptr* and *ynode* to the index of the first successor of *node*(*x*) under the adjacency matrix representation. Under the linked representation, *ynode* is set to a pointer to the header node (or a node number) of the first successor of *node*(*x*), and *yptr* is set to point to the arc node representing the arc from *node*(*x*) to *node*(*ynode*).
>
> *nextsucc*(*x*, *yptr*, *ynode*) accepts two array indices (*x* and *yptr*) in the adjacency matrix representation and sets both *yptr* and *ynode* to the array index of the successor of *node*(*x*) that follows *node*(*yptr*). In the linked representation, *x* is an array index or a pointer to a header node, *yptr* is a pointer to an arc node and is reset to point to the arc node that follows *node*(*yptr*) on the adjacency list, and *ynode* is set to point to the header node that terminates the arc node pointed to by the modified value of *yptr*.

Given these conventions, an algorithm to visit all successors of *node*(*x*) can be written as follows:

```
firstsucc(x,yptr,ynode);
while (yptr != NULL) {
   visit(ynode);
   nextsucc(x,yptr,ynode);
} /* end while */
```

This algorithm will operate correctly under both implementations.

Let us now present algorithms for *firstsucc* and *nextsucc*. If the adjacency matrix implementation is used, *nextsucc*(*x*, *yptr*, *ynode*) is implemented as follows:

```
for (i = yptr + 1; i < n; i++)
   if (adj(x,i)) {
      yptr = ynode = i;
      return;
   } /* end for ... if */
yptr = ynode = null;
return;
```

firstsucc(x, yptr, ynode) is implemented by

```
nextsucc(x,-1,ynode);
yptr = ynode;
```

Note that traversing all of a node's successors in a graph of n nodes is $O(n)$ using the adjacency matrix representation.

If the linked representation is used, *nextsucc* is implemented quite simply as follows. (We assume an *arcptr* field in each header node and *ndptr* and *nextarc* fields in each arc node.)

```
yptr = nextarc(yptr);
ynode = (yptr == NULL) ? NULL : ndptr(yptr);
```

firstsucc is implemented by

```
yptr = arcptr(x);
ynode = (yptr == NULL) ? NULL : ndptr(yptr);
```

Note that if e is the number of edges (arcs) in the graph and n the number of graph nodes, e/n is the average number of arcs emanating from a given node. Traversing the successors of a particular node by this method is therefore $O(e/n)$ on the average. If the graph is sparse (that is, very few of the n^2 possible edges exist), this is a significant advantage of the adjacency list representation.

3. If a node has more than one predecessor, it is necessarily encountered more than once during a traversal. Therefore, to ensure termination and to ensure that each node is visited only once, a traversal algorithm must check that a node being encountered has not been visited previously. There are two ways to do this. One is to maintain a set of visited nodes. The set would be maintained for efficient lookup and insertion as a search tree or a hash table. Whenever a node is encountered, the table is searched to see if the node has already been visited. If it has, the node is ignored; if it has not, the node is visited and added to the table. Of course, the lookup and insertion add to the traversal overhead.

The second technique is to keep a flag *visited(nd)* in each node. Initially, all flags are set off (*false*) via a quick nongraph traversal through the list of graph nodes. The visit routine turns the flag on (*true*) in the node being visited. When a node is encountered, its flag is examined. If it is on, the node is ignored; if it is off, the node is visited and the flag is set on. The flagging technique is used more commonly, since the flag initialization overhead is less than the table lookup and maintenance overhead.

Spanning Forests

A *forest* may be defined as an acyclic graph in which every node has one or no predecessors. A *tree* may be defined as a forest in which only a single node (called the

root) has no predecessors. Any forest consists of a collection of trees. An ***ordered forest*** is one whose component trees are ordered. Given a graph G, F is a ***spanning forest*** of G if

1. F is a subgraph of G containing all the nodes of G.
2. F is an ordered forest containing trees $T_1, T_2, \ldots, T_n$.
3. T_i contains all nodes that are reachable in G from the root of T_i and are not contained in T_j for some $j < i$.

F is a spanning tree of G if it is a spanning forest of G and consists of a single tree.

Figure 8.4.1 illustrates four spanning forests for the graph of Figure 8.1.3. In each forest the arcs of the graph that are not included in the forest are shown as dotted arrows, and the arcs included in the forest are solid arrows. The spanning forests of Figure 8.4.1a and b are spanning trees, whereas those of Figure 8.4.1c and d are not.

Any spanning tree divides the edges (arcs) of a graph into four distinct groups: ***tree edges, forward edges, cross edges***, and ***back edges***. Tree edges are arcs of the graph that are included in the spanning forest. Forward edges are arcs of the graph from a node to a spanning forest nonson descendant. A cross edge is an arc from one node to another node that is not the first node's descendant or ancestor in the spanning forest. Back edges are arcs from a node to a spanning forest ancestor. The following table classifies the arcs of the graph of Figure 8.1.3 in relation to each of the spanning trees of Figure 8.4.1:

Arc	(a)	(b)	(c)	(d)
$< A, C >$	tree	tree	cross	cross
$< A, D >$	tree	tree	tree	tree
$< B, E >$	tree	tree	tree	tree
$< B, F >$	tree	cross	tree	cross
$< B, H >$	tree	tree	tree	tree
$< C, G >$	tree	tree	cross	tree
$< F, G >$	back	back	tree	back
$< G, B >$	tree	tree	back	tree
$< G, F >$	forward	tree	back	tree
$< H, G >$	back	back	cross	back

Consider a traversal method that visits all nodes reachable from a previously visited node before visiting any node not reachable from a previously visited node. In such a traversal a node is visited either arbitrarily or as the successor of a previously visited node. The traversal defines a spanning forest in which an arbitrarily selected node is the root of a tree in the spanning forest, and in which a node $n1$ se-

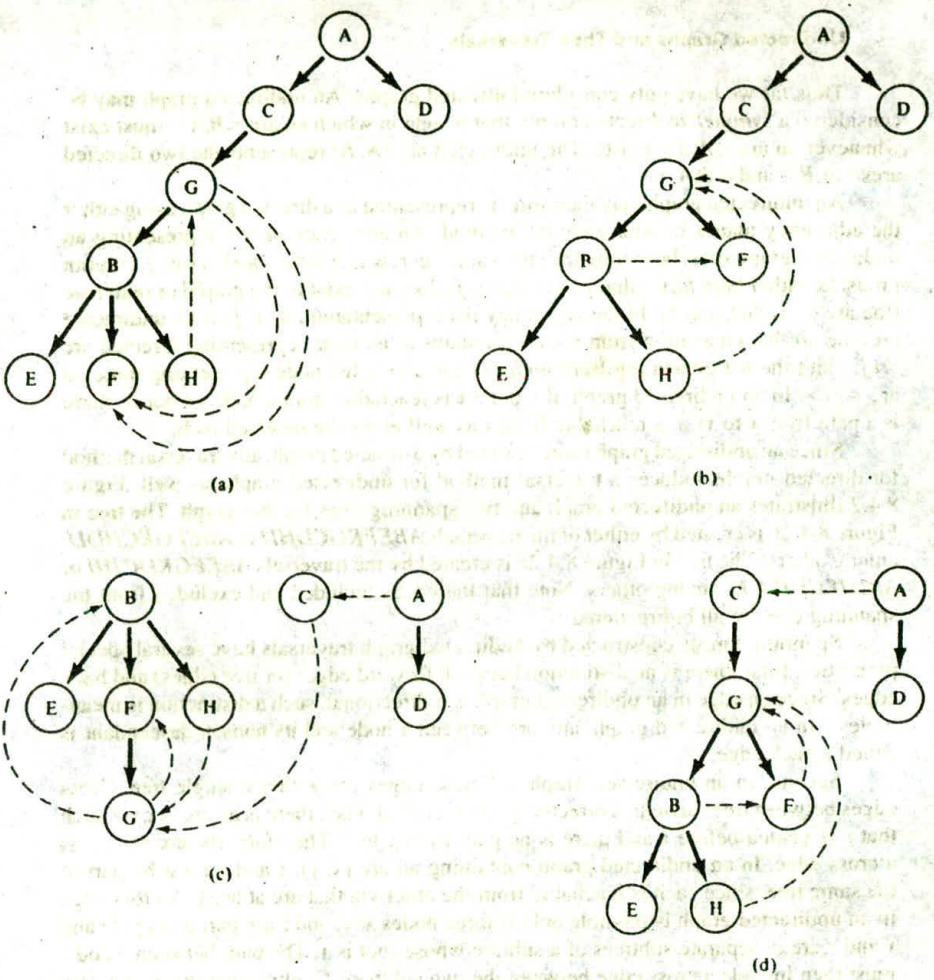


Figure 8.4.1

lected as the successor of $n2$ is a son of $n2$ in the spanning forest. For example, the traversal *ACGBEFHD* defines the forest of Figure 8.4.1a, and the traversal *BEFGHCAD* defines that of Figure 8.4.1c. Although a particular traversal defines a single spanning forest, a number of traversals may define the same forest. For example, *ACDGBEFH* also defines the spanning forest of Figure 8.4.1a.

Undirected Graphs and Their Traversals

Thus far we have only considered directed graphs. An undirected graph may be considered a ***symmetric*** directed graph, that is, one in which an arc <*B,A*> must exist whenever an arc <*A,B*> exists. The undirected arc (*A, B*) represents the two directed arcs <*A,B*> and <*B,A*>.

An undirected graph may therefore be represented as a directed graph using either the adjacency matrix or adjacency list method. An adjacency matrix representing an undirected graph must be symmetric; the values in row *i*, column *j* and in row *j*, column *i* must be either both *false* [that is, the arc (*i, j*) does not exist in the graph] or both *true* [the arc (*i, j*) does exist]. In the adjacency list representation, if (*i, j*) is an undirected arc, the arc list emanating from *node*(*i*) contains a list node representing directed arc <*i,j*> and the list emanating from *node*(*j*) contains a list node representing directed arc <*j,i*>. In an undirected graph, if a node *x* is reachable from a node *y* (that is, there is a path from *y* to *x*), *y* is reachable from *x* as well along the reversed path.

Since an undirected graph is represented by a directed graph, any traversal method for directed graphs induces a traversal method for undirected graphs as well. Figure 8.4.2 illustrates an undirected graph and two spanning trees for that graph. The tree in Figure 8.4.2b is created by either of the traversals *ABEFKGCDHIJ* or *ABEFGKCHDIJ*, among others. The tree in Figure 8.4.2c is created by the traversals *ABEFGKDCJHI* or *ABDJECHIFGK*, among others. Note that the edges included and excluded from the spanning tree are all bidirectional.

Spanning forests constructed by undirected graph traversals have several special properties. First, there is no distinction between forward edges (or tree edges) and back edges. Since an edge in an undirected graph is bidirectional, such a distinction is meaningless. In an undirected graph, any arc between a node and its nonson descendant is called a back edge.

Second, in an undirected graph, all cross edges are within a single tree. Cross edges between trees arise in a directed graph traversal when there is an arc <*x,y*> such that *y* is visited before *x* and there is no path from *y* to *x*. Therefore, the arc <*x,y*> is a cross edge. In an undirected graph containing an arc (*x, y*), *x* and *y* must be part of the same tree, since each is reachable from the other via that arc at least. A cross edge in an undirected graph is possible only if three nodes *x*, *y*, and *z* are part of a cycle and *y* and *z* are in separate subtrees of a subtree whose root is *x*. The path between *y* and *z* must then include a cross edge between the two subtrees. Confirm that this is the case with all cross edges of Figure 8.4.2c.

Because undirected graphs have "double" the edges of directed graphs, their spanning forests tend to have fewer, but larger, trees.

An undirected graph is termed ***connected*** if every node in it is reachable from every other. Pictorially, a connected graph has only one segment. For example, the graph of Figure 8.4.2a is a connected graph. The graph of Figure 8.1.1a is not connected, since node *E* is not reachable from node *C*, for example. A ***connected component*** of an undirected graph is a connected subgraph containing all arcs incident to any of its nodes such that no graph node outside the subgraph is reachable from any node in the subgraph. For example, the subgraph of Figure 8.1.1a has three connected components:

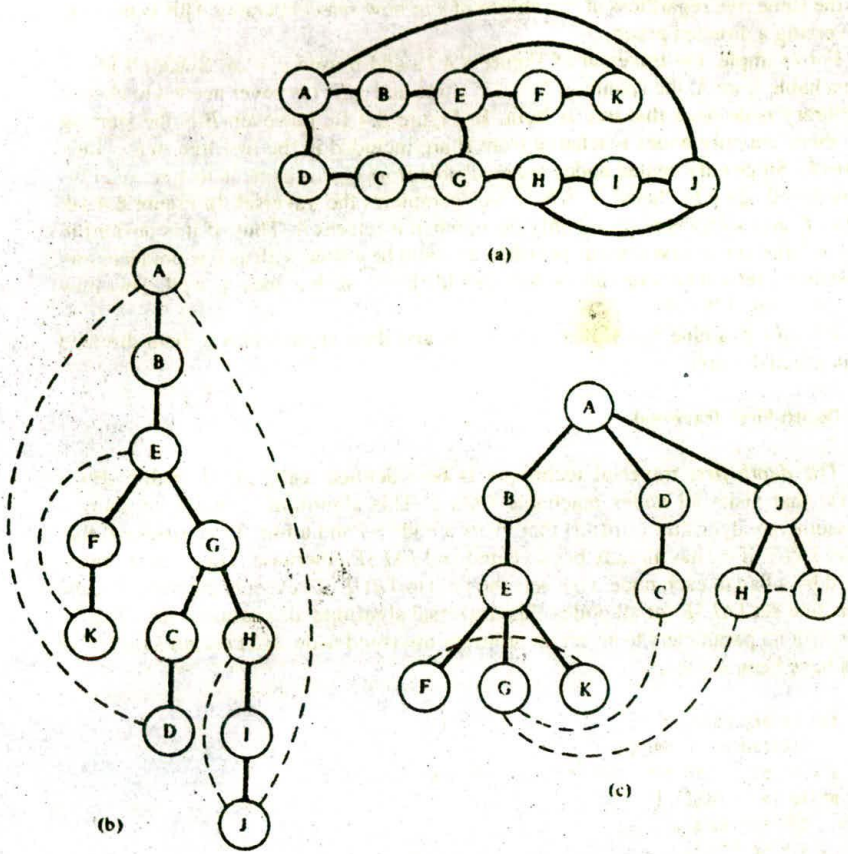


Figure 8.4.2

nodes *A,B,C,D,F*; nodes *E* and *G*; and node *H*. A connected graph has a single connected component.

The spanning forest of a connected graph is a spanning tree. Each tree in the spanning forest of an undirected graph contains all the nodes in a single connected component of the graph. Thus any traversal method that creates a spanning forest (that is, one that visits all nodes reachable from visited nodes before visiting any other nodes) can be used to determine whether an undirected graph is connected and to identify an undirected graph's connected components.

In traversing an undirected graph, it is not very important which node *s* is used as the starting node or how *select* chooses an arbitrary node (except perhaps in terms of

the efficiency of *select*.) This is because all nodes of a connected component will wind up in the same tree regardless of the choice of s or how *select* operates. This is not true in traversing a directed graph.

For example, the traversal of Figure 8.4.1a and b used $s = A$. Since all nodes are reachable from A, the spanning forest is a tree and *select* is never needed to choose an arbitrary node once that tree is built. In Figure 8.4.1c, however, B is the starting node; therefore only nodes reachable from B are included in the first tree. *select* then chooses C. Since only visited nodes are reachable from C, it is alone in its tree. *select* is then required again to choose A, whose tree completes the traversal. In Figure 8.4.1d, s equals C and *select* is required only once, when it returns A. Thus, if it is desired to create as large and as few trees as possible, s should be a node with as few predecessors as possible (preferably none) and *select* should choose such a node as well. This may make *select* less efficient.

We now examine two traversal methods and their applications to both directed and undirected graphs.

Depth-First Traversal

The ***depth-first*** traversal technique is best defined using an algorithm *dftraverse(s)* that visits all nodes reachable from s. This algorithm is presented shortly. We assume an algorithm *visit(nd)* that visits a node *nd* and a function *visited(nd)* that returns *TRUE* if *nd* has already been visited and *FALSE* otherwise. This is best implemented by a flag in each node. *visit* sets the field to *TRUE*. To execute the traversal, the field is first set *FALSE* for all nodes. The traversal algorithm also assumes the function *select* with no parameters to select an arbitrary unvisited node. *select* returns *null* if all nodes have been visited.

```
for (every node nd)
   visited(nd) = FALSE;
s = a pointer to the starting node for the traversal;
while (s != NULL) {
   dftraverse(s);
   s = select();
} /* end while */
```

Note that a starting node s is specified for the traversal. This node becomes the root of the first tree in the spanning forest. The following is a recursive algorithm for *dftraverse(s)*, using the routines *firstsucc* and *nextsucc* presented earlier:

```
/* visit all nodes reachable from s */
visit(s);
/* traverse all unvisited successors of s */
firstsucc(s,yptr,nd);
while (yptr != NULL) {
   if (visited(nd) == FALSE)
      dftraverse(nd);
```

```
    nextsucc(s,yptr,nd);
} /* end while */
```

If it is known that every node in the graph is reachable from the starting node s (as in the case of the graph of Figure 8.1.3 starting from node A or in the case of a connected undirected graph such as that of Figure 8.4.2a), the spanning forest is a single spanning tree and the ***while*** loop and *select* are not required in the traversal algorithm, since every node is visited in a single call to *dftraverse*.

A depth-first traversal, as its name indicates, traverses a single path of the graph as far as it can go (that is, until it visits a node with no successors or a node all of whose successors have already been visited). It then resumes at the last node on the path just traversed that has an unvisited successor and begins traversing a new path emanating from that node. Spanning trees created by a depth-first traversal tend to be very deep. Depth-first traversal is also sometimes called ***depth-first search***.

Figure 8.4.1a and c are both depth-first spanning trees of the graph of Figure 8.1.3. In Figure 8.4.1a, the traversal started at A and proceeded as follows: *ACGBEFHD*. Note that this is the preorder traversal of the spanning tree. In fact, the depth-first traversal of a tree is its preorder traversal. In Figure 8.4.1c, the traversal starts at B and proceeds as follows: *BEFGH*. At that point, all nodes reachable from B have been visited; consequently *select* is called to find an arbitrary unvisited node. Figure 8.4.1c assumes that *select* returned a pointer to C. But no unvisited nodes are successors of C (G has already been visited); therefore *select* is called again and returns A. D is an unvisited successor of A and is visited to complete the traversal. Thus Figure 8.4.1c corresponds to the complete depth-first traversal *BEFGHCAD*.

This illustrates that there may be several depth-first traversals and depth-first spanning trees for a particular directed graph. The traversal depends very much on how the graph is represented (adjacency matrix or adjacency list), on how the nodes are numbered, on the starting node, and on how the basic depth-first traversal is implemented (in particular, the implementation of *firstsucc*, *nextsucc*, and *select*). The essential feature of a depth-first traversal is that, after a node is visited, all descendants of the node are visited before its unvisited brothers. Figure 8.4.2b represents the depth-first traversal *ABEFKGCDHIJ* of the undirected graph of Figure 8.4.2a.

As usual, a stack can be used to eliminate the recursion in depth-first traversal. The following is a complete nonrecursive depth-first traversal algorithm:

```
for (every node nd)
    visited(nd) = FALSE;
s = a pointer to the starting node for the traversal;
ndstack = the empty stack;
while (s != NULL) {
    visit(s);
    /* find first unvisited successor */
    firstsucc(s,yptr,nd);
    while ((nd != NULL) && (visited(nd) == TRUE))
        nextsucc(s,yptr,nd);
    /* if no unvisited successors, simulate return */
```

```
   /*          from recursive call                    */
   while ((nd == NULL) && (empty(ndstack) == FALSE)) {
      popsub(ndstack,s,yptr);
      /*  find next unvisited successor  */
      nextsucc(s,yptr,nd);
      while ((nd != NULL) && (visited(nd) == TRUE))
         nextsucc(s,yptr,nd);
   } /*  end while ((nd == NULL) && ...) */
   if (nd != NULL) {
      /* simulate the recursive call  */
      push(ndstack,s,yptr);
      s = nd;
   } /* end if  */
   else
      s = select;
} /* end while (s != NULL) */
```

Note that each stack element contains pointers to both a father node (s) and an incident arc or its son ($yptr$) to allow continuation of the traversal of the successors.

To use this algorithm to construct a spanning tree, it is necessary to keep track of a node's father when it is visited, as follows. First, change the *if* statement at the end of the algorithm to

```
if (nd != NULL) {
   /*  simulate the recursive call */
   push(ndstack,s,yptr);
   f = s;              /* this statement is added */
   s = nd;
} /*  end  if  */
else {
   s = select();
   f = NULL;           /* this statement is added  */
} /*  end if  */
```

Second, initialize f to *NULL* at the beginning of the algorithm. Third, change $visit(s)$ to $addson(f,s)$: $visited(s) = TRUE$, where *addson* adds $node(s)$ to the tree as the next son of $node(f)$. [Recall that $visit(s)$ was defined to set $visited(s)$ to *TRUE*.] If f is *NULL*, $addson(f,s)$ adds $node(s)$ as a new tree in the forest (for example, it calls *maketree*. It is assumed that the tree roots are kept in a linked list managed by *maketree* using two global variables pointing to the first and last trees in the forest.)

You are invited to apply this modified algorithm to the graph of Figure 8.1.3, with s initialized to A and the successors of any node ordered alphabetically, to obtain the spanning tree of Figure 8.4.1a. Similarly, applying the modified algorithm to the same graph, with s initialized to B and assuming that *select* chooses C before A and D, and A before D, yields the spanning forest of Figure 8.4.1c. Applying the algorithm to the graph of Figure 8.4.2a produces the tree of Figure 8.2.4b.

As illustrated by Figure 8.4.2b, a depth-first spanning forest of an undirected graph may contain tree edges and back edges but cannot contain any cross edges. To see why, assume that (x, y) is an edge in the graph and that x is visited before y. In a depth-first traversal, y must be visited as a descendant of x before any nodes that are not reachable from x. Thus the arc (x, y) is either a tree edge or a back edge. The same is true in reverse if y is visited first, since the undirected arc (x, y) is equivalent to (y, x). In a directed graph, however, the arc $<x,y>$ but not $<y,x>$ may be in the graph. If y is visited first, since x may not be reachable from y, x may not be in a subtree rooted at y, so the arc $<x,y>$ may be a cross edge even in a depth-first spanning tree. This is illustrated by the arcs $<A,C>$ and $<C,G>$ in Figure 8.4.1c.

Applications of Depth-First Traversal

Depth-first traversal, like any other traversal method that creates a spanning forest, can be used to determine if an undirected graph is connected and to identify the connected components of an undirected graph. Whenever *select* is called, a new connected component of the graph is being traversed. If *select* is never called, the graph is connected.

Depth-first traversal can also be used to determine if a graph is acyclic. In both directed and undirected graphs, a cycle exists if and only if a back edge exists in a depth-first spanning forest. It is obvious that if a back edge exists, the graph contains a cycle formed by the back edge itself and the tree path starting at the ancestor head of the back edge and ending at the descendant tail of the back edge. To prove that a back edge must exist in a cyclic graph, consider the node *nd* of a cycle that is the first node in its cycle visited by a depth-first traversal. There must exist a node x such that the arc (x, nd) or $<x,nd>$ is in the cycle. Since x is in the cycle, it is reachable from *nd*, so that x must be a descendant of *nd* in the spanning forest. Thus the arc (x, nd) or $<x,nd>$ is a back edge by definition.

Therefore to determine if a graph is acyclic it is only necessary to determine that an edge encountered during a depth-first traversal is not a back edge. When considering an edge (s, nd) or $<s,nd>$ in the depth-first traversal algorithm, the edge can be a back edge only if *visited*(*nd*) is *true*. In an undirected graph, where there are no cross edges in a depth-first traversal, (s, nd) is a back edge if and only if *visited*(*nd*) is *true* and nd ! = *father*(s) in the spanning forest.

In a directed graph, $<s,nd>$ can be a back edge even if nd == *father*(s), since $<s,nd>$ and $<nd,s>$ are distinct arcs. Thus in a directed graph a cycle may consist of only two nodes (such as s and *nd*), whereas in an undirected graph, at least three are required. However, since a directed graph's spanning tree may contain cross edges as well as back edges, *visited*(*nd*) equaling true is not enough to detect a cycle. For example, cross edges $<A,C>$, $<H,G>$ and $<C,G>$ in Figure 8.4.1c are not part of a cycle, although C has been visited by the time $<A,C>$ is considered, and G has been visited by the time $<H,G>$ and $<C,G>$ are considered. To determine that an arc $<s,nd>$ is not a back edge when *visited*(*nd*) is *true*, it is necessary to consider each ancestor of s in turn to ensure that it does not equal *nd*. We leave the details of an algorithm to determine if a directed graph is acyclic (that is, a dag) as an exercise for the reader.

In the preceding section we examined an algorithm to schedule tasks given a series of required precedences among those tasks. We saw that the precedence relations among the tasks can be represented by a dag. The algorithm presented in that section can be used to specify a linear ordering of the nodes in which no node comes before a preceding node. Such a linear ordering is called a ***topological sort*** of the nodes.

A depth-first traversal can be used to produce a reverse topological ordering of the nodes. Consider the inorder traversal of the spanning forest formed by a depth-first traversal of a dag. We now prove that such an inorder traversal produces a reverse topological ordering.

To repeat the recursive definition of inorder traversal of a forest from Section 5.5:

1. Traverse the forest formed by the subtrees of the first tree in the forest, if any.
2. Visit the root of the first tree.
3. Traverse the forest formed by the remaining trees of the forest, if any.

To differentiate in the following discussion between the depth-first traversal that creates the forest and the inorder traversal of the forest, we refer to DF-visits (and a DF-traversal) and IO-visits (and an IO-traversal), respectively.

An IO-traversal of the depth-first spanning tree of a dag must be in reverse topological order. That is, if x precedes y, x is IO-visited after y. To see why this is so, consider the arc $<x,y>$. We show that y is IO-visited before x. Since the graph is acyclic, $<x,y>$ cannot be a back arc. If it is a tree arc or a forward arc, so that y is a descendant of x in the spanning forest, y is IO-visited before x because an inorder traversal IO-visits the root of a subtree after traversing all its subtrees. If $<x,y>$ is a cross edge, y must have been IO-visited before x (otherwise, y would have been a descendant of x). Consider the smallest subtrees, $S(x)$ and $S(y)$, containing x and y respectively whose roots are brothers. (Roots of trees of the spanning forest are also considered brothers in this context.) Then since y was DF-visited before x, $S(y)$ precedes $S(x)$ in their subtree ordering. Thus $S(y)$ is IO-traversed before $S(x)$, which means that y is IO-visited before x.

Thus an algorithm to determine a reverse topological ordering of the nodes of a dag consists of a depth-first search of the dag followed by an inorder traversal of the resulting spanning forest. Fortunately, it is unnecessary to make a separate traversal of the spanning tree since an inorder traversal can be incorporated directly into the recursive depth-first traversal algorithm. To do this, simply push a node onto a stack when it is DF-visited. Whenever *dftraverse* returns, pop the stack and IO-visit the popped node. Since *dftraverse* DF-traverses all the subtrees of a tree before completing the tree's DF-traversal and traverses the first subtree of a set of brothers before DF-traversing the others, this routine yields an IO-traversal. The reader is invited to implement this algorithm nonrecursively.

Efficiency of Depth-First Traversal

The depth-first traversal routine visits every node of a graph and traverses all the successors of each node. We have already seen that, for the adjacency matrix implementation, traversing all successors of a node using *firstsucc* and *nextsucc* is $O(n)$,

where n is the number of graph nodes. Thus traversing the successors of all the nodes is $O(n^2)$. For this reason, depth-first search using the adjacency matrix representation is $O(n + n^2)$ (n node visits and n^2 possible successor examinations), which is the same as $O(n^2)$.

If the adjacency list representation is used, traversing all successors of all nodes is $O(e)$, where e is the number of edges in the graph. Assuming that the graph nodes are organized as an array or a linked list, visiting all n nodes is $O(n)$, so that the efficiency of depth-first traversal using adjacency lists is $O(n + e)$. Since e is usually much smaller than n^2, the adjacency list representation yields more efficient traversals. (The difference, however, is somewhat offset by the fact that in an adjacency matrix, traversal of successors involves merely counting from 1 to n, whereas in an adjacency list, it involves successively accessing fields in nodes.) Depth-first traversal is often considered $O(e)$, since e is usually larger than n.

Breadth-First Traversal

An alternative traversal method, ***breadth-first traversal*** (or ***breadth-first search***), visits all successors of a visited node before visiting any successors of any of those successors. This is in contradistinction to depth-first traversal, which visits the successors of a visited node before visiting any of its "brothers." Whereas depth-first traversal tends to create very long, narrow trees, breadth-first traversal tends to create very wide, short trees. Figure 8.4.1b represents a breadth-first traversal of the graph of Figure 8.1.3, and Figure 8.4.2c represents a breadth-first traversal of the graph of Figure 8.4.2a.

In implementing depth-first traversal, each visited node is placed on a stack (either implicitly via recursion or explicitly), reflecting the fact that the last node visited is the first node whose successors will be visited. Breadth-first traversal is implemented using a queue, representing the fact that the first node visited is the first node whose successors are visited. The following is an algorithm *bftraverse*(s) to traverse a graph using breadth-first traversal beginning at *node*(s):

```
ndqueue = the empty queue;
while (s != NULL) {
   visit(s);
   insert(ndqueue,s);
   while (empty(ndqueue) == FALSE) {
      x = remove(ndqueue);
      /* visit all successors of x */
      firstsucc(x,yptr,nd);
      while (nd != NULL) {
         if (visited(nd) == FALSE) {
            visit(nd);
            insert(ndqueue,nd);
         } /* end if */
         nextsucc(x,yptr,nd);
      } /* end while */
   } /* end while */
```

```
    s = select();
} /* end while */
```

We leave the modification of the algorithm to produce a breadth-first spanning forest as an exercise for the reader. Figure 8.4.1b illustrates a breadth-first spanning tree for the graph of Figure 8.1.3, representing the breadth-first traversal *ACDGBFEH*. Note that although the traversal differs significantly from the depth-first traversal ACGBEFHD that produced the spanning tree of Figure 8.4.1a, the two spanning trees themselves do not differ except for the position of node *F*. This reflects the fact that the graph of Figure 8.1.3 has relatively few arcs (ten) compared with the total number of potential arcs ($n^2 = 64$). In a graph with more arcs, the difference in spanning forests is more pronounced.

A breadth-first spanning tree does not have any forward edges, since all nodes adjacent to a visited node *nd* have already been visited or are spanning tree sons of *nd*. For the same reason, for a directed graph, all cross edges within the same tree are to nodes on the same or higher levels of the tree. For an undirected graph, a breadth-first spanning forest contains no back edges, since every back edge is also a forward edge.

Breadth-first traversal can be used for some of the same applications as depth-first traversal. In particular, breadth-first traversal can be used to determine if an undirected graph is connected and to identify the graph's connected components. Breadth-first traversal can also be used to determine if a graph is cyclic. For a directed graph, this is detected when a back edge is found; for an undirected graph, it is detected when a cross edge within the same tree is found.

For an unweighted graph, breadth-first traversal can also be used to find the shortest path (fewest arcs) from one node to another. Simply begin the traversal at the first node and stop when the target node has been reached. The breadth-first spanning tree path from the root to the target is the shortest path between the two nodes.

The efficiency of breadth-first traversal is the same as that of depth-first traversal: each node is visited once and all arcs emanating from every node are considered. Thus its efficiency is $O(n^2)$ for the adjacency matrix graph representation and $O(n + e)$ for the adjacency list graph representation.

Minimum Spanning Trees

Given a connected weighted graph *G*, it is often desired to create a spanning tree *T* for *G* such that the sum of the weights of the tree edges in *T* is as small as possible. Such a tree is called a ***minimum spanning tree*** and represents the "cheapest" way of connecting all the nodes in *G*.

There are a number of techniques for creating a minimum spanning tree for a weighted graph. The first of these, ***Prim's algorithm***, discovered independently by Prim and Dijkstra, is very much like Dijkstra's algorithm for finding shortest paths. An arbitrary node is chosen initially as the tree root (note that in an undirected graph and its spanning tree, any node can be considered the tree root and the nodes adjacent to it as its sons). The nodes of the graph are then appended to the tree one at a time until all nodes of the graph are included.

The node of the graph added to the tree at each point is that node adjacent to a node of the tree by an arc of minimum weight. The arc of minimum weight becomes a tree arc connecting the new node to the tree. When all the nodes of the graph have been added to the tree, a minimum spanning tree has been constructed for the graph.

To see that this technique creates a minimum spanning tree, consider a minimum spanning tree T for the graph and consider the partial tree PT built by Prim's algorithm at any point. Suppose that (a,b) is the minimum-cost arc from nodes in PT to nodes not in PT, and suppose that (a,b) is not in T. Then, since there is a path between any two graph nodes in a spanning tree, there must be an alternate path between a and b in T that does not include arc (a,b). This alternate path P must include an arc (x,y) from a node in PT to a node outside of PT. Let us assume that P contains subpaths between a and x and between y and b.

Now consider what would happen if we replaced arc (x, y) in T with (a, b) to create NT. We claim that NT is also a spanning tree. To prove this, we need to show two things: that any two nodes of the graph are connected in NT and that NT does not contain a cycle—that is, that there is only one path between any two nodes in NT.

Since T is a spanning tree, any two nodes, m and n, are connected in T. If the path between them in T does not contain (x,y), the same path connects them in NT. If the path between them in T does contain (x,y), consider the path in NT formed by the subpath in T from m to x, the subpath in P (which is in T) from x to a, the arc (a,b), the subpath in P from b to y, and the subpath in T from y to n. This is a path from m to n in NT. Thus any two nodes of the graph are connected in NT.

To show that NT does not contain a cycle, suppose that it did. If the cycle does not contain (a,b), the same cycle would exist in T. But that is impossible, since T is a spanning tree. Thus the cycle must contain (a,b). Now consider the same cycle with arc (a,b) replaced with the subpath of P between a and x, the arc (x,y), and the subpath P between y and b. The resulting path must also be a cycle and is a path entirely in But, again, T cannot contain a cycle. Therefore NT also does not contain a cycle.

NT has thus been shown to be a spanning tree. But NT must have lower cost than T, since (a,b) was chosen to have lower cost than (x,y). Thus T is not a minimum spanning tree unless it includes the lowest weight arc from PT to nodes outside PT. Therefore any arc added by Prim's algorithm must be part of a minimum spanning tree.

The crux of the algorithm is a method for efficient determination of the "closest" node to a partial spanning tree. Initially, when the partial tree consists of a single root node, the distance of any other node *nd* from the tree, *distance*[*nd*], is equal to *weight*(*root*,*nd*). When a new node, *current*, is added to the tree, *distance*[*nd*] is modified to the minimum of *distance*[*nd*] and *weight*(*current*,*nd*). The node added to the tree at each point is the node whose distance is lowest. For nodes *tnd* in the tree, *distance*[*tnd*] is set to *infinity*, so that a node outside the tree is chosen as closest. An additional array *closest*[*nd*] points to the node in the tree such that *distance*[*nd*] = *weight*(*closest*[*nd*],*nd*): that is, the node in the tree closest to *nd*. If two nodes, x and y, are not adjacent, *weight*(x,y) is also *infinity*.

Prim's algorithm may therefore be implemented as follows:

```
root = an arbitrary node chosen as root;
for (every node nd in the graph)  {
```

```
      distance[nd] = weight(root,nd);
      closest[nd] = root;
   } /* end for */
   distance[root] = INFINITY;
   current = root;
   for (i = 1; i < number of nodes in the graph; ++i) {
      /* find the node closest to the tree */
      mindist = INFINITY;
      for (every node nd in the graph)
         if (distance[nd] < mindist) {
            current = nd;
            mindist = distance[nd];
         } /* end if */
      /* add the closest node to the tree */
      /*       and adjust distances       */
      addson(closest[current],current);
      distance[current] = INFINITY;
      for (every node nd adjacent to current)
         if ((distance[nd] < INFINITY)
               && (weight(current,nd) < distance[nd])) {
            distance[nd] = weight(current,nd);
            closest[nd] = current;
         } /* end if */
   } /* end for */
```

If the graph is represented by an adjacency matrix, each for loop in Prim's algorithm must examine $O(n)$ nodes. Since the algorithm contains a nested for loop, it is $O(n^2)$.

However, just as in Dijkstra's algorithm, Prim's algorithm can be made more efficient by maintaining the graph using adjacency lists and keeping a priority queue of the nodes not in the partial tree. The first inner loop [*for (every node nd in the graph)* . . .] can then be replaced by removing the minimum-distance node from the priority queue and adjusting the priority queue. The second inner loop simply traverses an adjacency list and adjusts the position of any nodes whose distance is modified in the priority queue. Under this implementation, Prim's algorithm is $O((n + e) \log n)$.

Kruskal's Algorithm

Another algorithm to create a minimum spanning tree is attributable to Kruskal. The nodes of the graph are initially considered as n distinct partial trees with one node each. At each step of the algorithm, two distinct partial trees are connected into a single partial tree by an edge of the graph. When only one partial tree exists (after $n - 1$ such steps), it is a minimum spanning tree.

The issue of course is what connecting arc to use at each step. The answer is to use the arc of minimum cost that connects two distinct trees. To do this, the arcs can be placed in a priority queue based on weight. The arc of lowest weight is then examined to see if it connects two distinct trees.

To determine if an arc (x,y) connects distinct trees, we can implement the trees with a *father* field in each node. Then we can traverse all ancestors of x and y to obtain the roots of the trees containing them. If the roots of the two trees are the same node, x and y are already in the same tree, arc (x,y) is discarded, and the arc of next lowest weight is examined. Combining two trees simply involves setting the *father* of the root of one to the root of the other.

We leave the actual algorithm and its C implementation for the reader. Forming the initial priority queue is $O(e \log e)$. Removing the minimum-weight arc and adjusting the priority queue is $O(\log e)$. Locating the root of a tree is $O(\log n)$. Initial formation of the n trees is $O(n)$. Thus, assuming that $n < e$, as is true of most graphs, Kruskal's algorithm is $O(e \log e)$.

Round-Robin Algorithm

Still another algorithm, attributable to Tarjan and Cheriton, provides even better performance when the number of edges is low. The algorithm is similar to Kruskal's except that there is a priority queue of arcs associated with each partial tree, rather than one global priority queue of all unexamined arcs.

All partial trees are maintained in a queue, Q. Associated with each partial tree, T, is a priority queue, $P(T)$, of all arcs with exactly one incident node in the tree, ordered by the weights of the arcs. Initially, as in Kruskal's algorithm, each node is a partial tree. A priority queue of all arcs incident to *nd* is created for each node *nd*, and the single-node trees are inserted into Q in arbitrary order. The algorithm proceeds by removing a partial tree, T_1, from the front of Q; finding the minimum-weight arc a in $P(T1)$; deleting from Q the tree, $T2$, at the other end of arc a; combining $T1$ and $T2$ into a single new tree $T3$ [and at the same time combining $P(T1)$ and $P(T2)$, with a deleted, into $P(T3)$]; and adding $T3$ to the rear of Q. This continues until Q contains a single tree: the minimum spanning tree.

It can be shown that this round-robin algorithm requires only $O(e \log \log n)$ operations if an appropriate implementation of the priority queues is used.

EXERCISES

8.4.1. Consider the following nonrecursive depth-first traversal algorithm:

```
for (every node nd)
   visited(i) = FALSE;
s = a pointer to the starting node of the traversal;
ndstack = the empty stack;
while (s != NULL) {
   push(ndstack,s);
   while (empty(ndstack) == FALSE) {
      x = pop(ndstack);
      if (visited(x) == FALSE) {
         visit(x);
```

```
            firstsucc(x,yptr,nd);
            while(nd != NULL) {
               if(visited(nd) == FALSE)
                  push(ndstack,nd);
               nextsucc(x,yptr,nd);
            } /* end while */
         } /* end if */
      } /* end while */
      s = select();
   } /* end while */
```

(a) Apply the algorithm to the graphs of Figures 8.1.3 and 8.4.2a to determine the order in which the nodes are visited, if the successors of a node are assumed ordered in alphabetical order.

(b) Draw the spanning trees induced by the traversal on each of the graphs. Modify the algorithm to construct a depth-first spanning forest.

(c) Would a modified ordering of successors produce the spanning trees of Figures 8.4.1a and 8.4.2b using this algorithm? Would a modified ordering produce the spanning trees in (b) using the algorithm of the text?

(d) Is either algorithm preferable?

8.4.2. Write an algorithm and a C program to determine if a directed graph is a dag.

8.4.3. Write a recursive C program to print the nodes of a dag in reverse topological order.

8.4.4. Write a nonrecursive C program to print the nodes of a dag in reverse topological order

8.4.5. A node *nd* in a connected graph is an ***articulation point*** if removing *nd* and all arcs adjacent to *nd* results in an unconnected graph. Thus the "connectedness" of the graph depends on *nd*. A graph with no articulation points is called ***biconnected***.

(a) Show that the root of the depth-first spanning tree of a biconnected graph has only a single son.

(b) Show that if *nd* is not the root of a depth-first spanning tree *t*, *nd* is an articulation point if and only if *t* does not contain a back edge from a descendant of *nd* to an ancestor of *nd*.

(c) Modify the recursive depth-first traversal algorithm to determine if a connected graph is biconnected.

8.4.6. Write a C routine to create a breadth-first spanning forest of a graph.

8.4.7. Write C routines that use a breadth-first traversal to determine if a directed and an undirected graph are cyclic.

8.4.8. Write a C routine to produce the shortest path from node x to node y in an unweighted graph, if a path exists, or an indication that no path exists between the two nodes.

8.4.9. Show that the algorithms to find a cycle using depth-first or breadth-first search must be $O(n)$.

8.4.10. Implement Prim's algorithm using an adjacency matrix and using adjacency lists and a priority queue.

8.4.11. Implement Kruskal's algorithm as a C routine.

9

Storage Management

A programming language that incorporates a large number of data structures must contain mechanisms for managing those structures and for controlling how storage is assigned to them. The previous chapters of this book illustrated some of those management techniques. As data structures become more complex and provide greater capabilities to the user, the management techniques grow in complexity as well.

In this chapter we look at several techniques for implementing dynamic allocation and freeing of storage. Most of these methods are used in some form by operating systems to grant or deny user program requests. Others are used directly by individual language processors. We begin by expanding the concept of a list.

9.1 GENERAL LISTS

In Chapter 4 and in Section 7.1 we examined linked lists as a concrete data structure and as a method of implementation for such abstract data types as the stack, the queue, the priority queue, and the table. In those implementations a list always contained elements of the same type.

It is also possible to view a list as an abstract data type in its own right. As an abstract data type, a list is simply a sequence of objects called ***elements***. Associated with each list element is a ***value***. We make a very specific distinction between an ***element***, which is an object as part of a list, and the element's ***value***, which is the object

considered individually. For example, the number 5 may appear on a list twice. Each appearance is a distinct element of the list, but the values of the two elements—the number 5—are the same. An element may be viewed as corresponding to a node in the linked list implementation, whereas a value corresponds to the node's contents. Note that the phrase "linked list" refers to the linked implementation of the abstract data type "list."

There is also no reason to assume that the elements of a list must be of the same type. Figure 9.1.1 illustrates a linked list implementation of an abstract list *list*1 that contains both integers and characters. The elements of that list are 5, 12, 's', 147, and 'a'. The pointer *list*1 is called an ***external pointer*** to the list (because it is not contained within a list node), whereas the other pointers in the list are ***internal pointers*** (because they are contained within list nodes). We often reference a linked list by an external pointer to it.

It is also not necessary that a list contain only "simple" elements (for example, integers or characters); it is possible for one or more of the elements of a list to themselves be lists. The simplest way to implement a list element e whose value is itself a list l, is by representing the element by a node containing a pointer to the linked list implementation of l.

For example, consider the list *list*2 of Figure 9.1.2. This list contains four elements. Two of these are integers (the first element is the integer 5; the third element is the integer 2) and the other two are lists. The list that is the second element of *list*2 contains five elements, three of which are integers (the first, second and fifth elements) and two of which are lists (the third element is a list containing the integers 14, 9, and 3 and the fourth element is the null list [the list with no elements]). The fourth element of *list*2 is a list containing the three integers 6, 3, and 10.

There is a convenient notation for specifying abstract general lists. A list may be denoted by a parenthesized enumeration of its elements separated by commas. For example, the abstract list represented by Figure 9.1.1 may be denoted by

$$list1 = (5, 12, 's', 147, 'a')$$

The null list is denoted by an empty parenthesis pair [such as ()]. Thus the list of Figure 9.1.2 may be denoted by

$$list2 = (5, (3, 2, (14, 9, 3), (), 4), 2, (6, 3, 10))$$

We now define a number of abstract operations on lists. For now, we are concerned only with the definition and logical properties of the operations; we consider methods of implementing the operations later in this section. If *list* is a nonempty list, ***head(list)*** is defined as the value of the first element of *list*. If *list* is a nonempty list, ***tail(list)*** is defined as the list obtained by removing the first element of *list*. If *list* is the empty list,

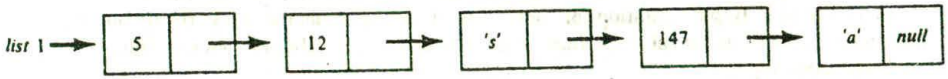


Figure 9.1.1 List of integers and characters.

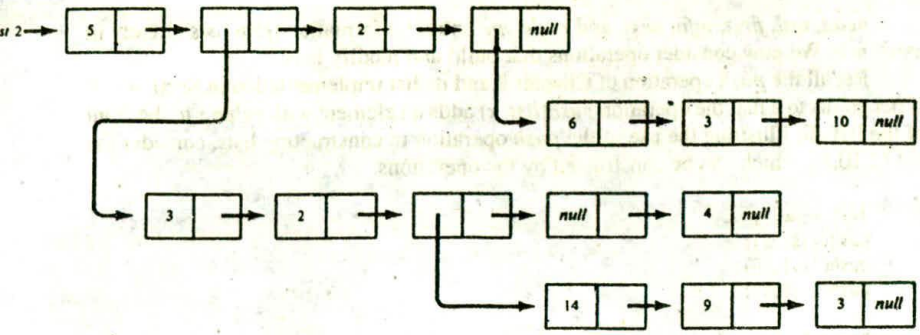


Figure 9.1.2

head(*list*) and *tail*(*list*) are not defined. For a general list, *head*(*list*) may be either a list (if the value of the first list element is itself a list) or a simple data item; *tail*(*list*) must be a (possibly null) list.

For example, if *list*1 and *list*2 are as in Figures 9.1.1 and 9.1.2,

*list*1 = (5, 12, 's', 147, 'a')

head(*list*1) = 5

tail(*list*1) = (12, 's', 147, 'a')

head(*tail*(*list*1)) = 12

tail(*tail*(*list*1)) = ('s', 147, 'a')

*list*2 = (5,(3,2,(14,9,3),(),4),2,(6,3,10))

tail(*list*2) = ((3,2,(14,9,3),(),4),2,(6,3,10))

head(*tail*(*list*2)) = (3,2,(14,9,3),(),4)

head(*head*(*tail*(*list*2))) = 3

The *head* and *tail* operations are not defined if their argument is not a list. A ***sublist*** of a list *l* is a list that results from the application of zero or more *tail* operations to *l*.

The operation ***first*(*list*)** returns the first element of list *list*. (If *list* is empty, *first*(*list*) is a special ***null element***, which we denote by *nullelt*.) The operation ***info*(*elt*)** returns the value of the list element *elt*. The *head* operation produces a value, whereas the *first* operation produces an element. In fact, *head*(*list*) equals *info*(*first*(*list*)). Finally, the operation ***next*(*elt*)** returns the element that follows the element *elt* on its list. This definition presupposes that an element can have only one follower.

The operation ***nodetype*(*elt*)** accepts a list element *elt* and returns an indication of the type of the element's value. Recall that, in the linked list implementation, an element is represented by a node. Thus if the enumerated constants *ch*, *intgr*, and *lst* represent the types character, integer, and list, respectively, *nodetype*(*first*(*list*1)) equals *intgr*, *nodetype*(*first*(*tail*(*tail*(*list*1))))) equals *ch*, and *nodetype*(*first*(*tail*(*list*2))) equals *lst*.

Operations That Modify a List

head, *tail*, *first*, *info*, *next*, and *nodetype* extract information from lists already in existence. We now consider operations that build and modify lists.

Recall the *push* operation of Chapter 2 and its list implementation in Section 4.2. If *list* points to a list, the operation *push*(*list*, *x*) adds an element with value *x* to the front of the list. To illustrate the use of the *push* operation in constructing lists, consider the list (5,10,8), which can be constructed by the operations:

```
list = null;
push(list,8);
push(list,10);
push(list,5);
```

Note that the abstract *push* operation changes the value of its first parameter to the newly created list. We introduce as a new operation the function

```
addon(list,x)
```

which returns a new list that has *x* as its head and *list* as its tail. For example, if *l*1 = (3,4,7), the operation

```
l2 = addon(l1,5);
```

creates a new list *l*2 equal to (5,3,4,7). The crucial difference between *push* and *addon* is that *push* changes the value of its first parameter, while *addon* does not. Thus, in the foregoing example, *l*1 retains the value (3,4,7). The operation *push*(*list*, *x*) is equivalent to *list* = *addon*(*list*, *x*). Since *addon* is more flexible than *push*, and since *push* is usually used only in connection with stacks, we henceforth use *addon* exclusively.

Two other operations used to modify lists are ***setinfo*** and ***setnext***. *setinfo*(*elt*, *x*) changes the value of a list element *elt* to the value *x*. Thus we may write *setinfo*(*first*(*list*),*x*) to reset the value of the first element of the list *list* to *x*. This operation is often abbreviated *sethead*(*list*, *x*). For example, if *list* equals (5,10,8), the operation

```
sethead(list,18)
```

changes *list* to (18, 10, 8), and the operation *sethead*(*list*,(5,7,3,4)) changes *list* to ((5,7,3,4),10,8).

sethead is called the "inverse *head* operation" for an obvious reason. After performing the operation *sethead*(*list*, *x*), the value of *head*(*list*) is *x*. Note that *sethead* (*list*, *x*) is equivalent to

```
list = addon(tail(list),x);
```

The operation *setnext*(*elt*1, *elt*2) is somewhat more complex. It modifies the list containing *elt*1, so that *next*(*elt*1) = *elt*2. *elt*1 cannot be the null element. *next*(*elt*2) is unchanged. Also if *next*(*elt*3) had been equal to *elt*2 before execution of

setnext(*elt*1,*elt*2), *next*(*elt*3) still equals *elt*2 after its execution, so that both *next*(*elt*1) and *next*(*elt*3) equal *elt*2. In effect, *elt*2 has become an element of two lists. The operation *settail*(*list*1,*list*2) is defined as *setnext*(*first*(*list*1), *first*(*list*2)) and sets the tail of *list*1 to *list*2. *settail* is sometimes called the inverse *tail* operation.

For example, if *list* = (5,9,3,7,8,6), *settail*(*list*, (8)) changes the value of *list* to (5,8), and *settail*(*list*, (4,2,7)) changes its value to (5,4,2,7). Note that the operation *settail* (*list*, *l*) is equivalent to *list* = *addon*(*l*, *head*(*list*)).

Examples

Let us look at some simple examples of algorithms that use these operations.

The first example is an algorithm to add 1 to every integer that is an element of a list *list*. Character or list elements remain unchanged.

```
p = first(list);
while (p != nullelt) {
   if (nodetype(p) == INTGR)
      setinfo(p, info(p) + 1);
   p = next(p);
} /* end while */
```

The second example involves deletions. We wish to delete from a list *list* any character element whose value is 'w'. (Compare this example with the routine of Section 4.2). One possible solution is as follows:

```
q = nullelt;
p = first(list);
while (p != nullelt)
   if (info(p) == 'w') {
      /* remove node(p) from the list */
      p = next(p);
      if (q == nullelt)
         list = tail(list);
      else
         setnext(q,p);
   } /* end if */
   else {
      q = p;
      p = next(p);
   } /* end else */
```

Before looking at a more complex example, we define a new term. An element (or a node) *n* is ***accessible*** from a list (or an external pointer) *l* if there is a sequence of *head* and *tail* operations that, if applied to *l*, yields a list with *n* as its first element. For example, in Figure 9.1.2 the node containing 14 is accessible from *list*2, since it is the first element of *tail*(*tail*(*head*(*tail*(*list*2)))). In fact, all the nodes shown in that figure are accessible from *list*2. When a node is removed from a list, it becomes inaccessible from the external pointer to that list.

Now suppose that we wish to increase by 1 the value in every integer node accessible from a given list pointer *list*. We cannot simply traverse *list*, since it is also necessary to traverse all lists that are elements of *list*, as well as all lists that may be elements of elements of *list*, and so forth. One tentative solution is the following recursive algorithm *addone2*(*list*).

```
p = first(list);
while (p != nullelt) {
   if (nodetype(p) == INTGR)
      setinfo(p, info(p) + 1);
   else
      if (nodetype(p) == lst)
         addone2(info(p));
   p = next(p);
} /* end while */
```

It is simple to remove the recursion and use a stack explicitly.

Linked List Representation of a List

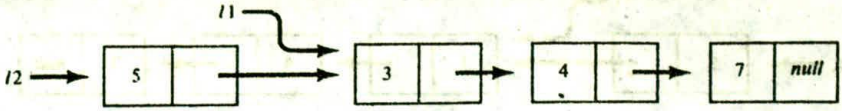
As previously noted, the abstract concept of a list is usually implemented by a linked list of nodes. Each element of the abstract list corresponds to a node of the linked list. Each node contains fields *info* and *next*, whose contents correspond to the abstract list operations *info* and *next*. The abstract concepts of a "list" and an "element" are both represented by a pointer: a list by an external pointer to the first node of a linked list and an element by a pointer to a node. Thus, a pointer to a node *nd* in a list, which represents a list element, also represents the sublist formed by the elements represented by the nodes from *nd* to the end of the list. The value of an element corresponds to the contents of the *info* field of a node.

Under this implementation the abstract operation *first*(*list*), which returns the first element of a list, is meaningless. If *list* is a pointer that represents a list, it points to the first node of a linked list. That pointer therefore also represents the first element of the list. Since *first*(*list*) and *list* are equivalent, *head*(*list*), which is equivalent to *info*(*first*(*list*)), is equivalent to *info*(*list*). *sethead*(*list*,*x*), defined as *setinfo*(*first*(*list*),*x*), is equivalent to *setinfo*(*list*,*x*). Similarly, *settail*(*list*1,*list*2), defined as *setnext*(*first*(*list*1),*first*(*list*2)), is equivalent to *setnext*(*list*1,*list*2) under the linked list representation.

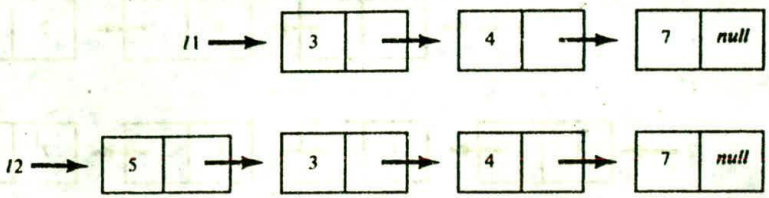
The operation *setinfo*(*elt*,*x*) is implemented by the assignment statement *info*(*elt*) = *x*, where *elt* is a pointer to a node, and the operation *setnext*(*elt*1,*elt*2) by the assignment *next*(*elt*1) = *elt*2. Since a list is represented by a pointer to its first node, an element of a list that is itself a list is represented by a pointer to the list in the *info* field of the node representing the element.

We defer a discussion of the implementation of *nodetype* until we present the C implementation of general lists later in this section.

There are two methods of implementing the *addon* and *tail* operations. Consider the list *l*1 = (3,4,7) and the operation *l*2 = *addon*(*l*1,5). The two possible ways of implementing this operation are illustrated in Figure 9.1.3a and b. In the first method,



(a) The Pointer Method



(b) The Copy Method

Figure 9.1.3

called the ***pointer method***, the list (3,4,7) is represented by a pointer to it, *l*1. To create the list *l*2, a node containing 5 is allocated and the value of *l*1 is placed in its *next* field. Thus the list *l*1 becomes a sublist of *l*2. The nodes of list *l*1 are used in two contexts: as part of list *l*1 and of list *l*2. In the second method, called the ***copy method***, the list (3,4,7) is copied before the new list element is added to it. *l*1 still points to the original version, and the new copy is made a sublist of *l*2. The copy method ensures that a node appears in only one context.

The difference between these methods becomes apparent when we attempt to perform the operation

sethead(*l*1,7)

The resulting lists are shown in Figure 9.1.4a and b. If the copy method is used, a change in list *l*1 does not affect list *l*2 (Figures 9.1.3b and 9.1.4b). If the pointer method is used, any subsequent change in list *l*1 also modifies *l*2 (Figures 9.1.3a and 9.1.4a).

The *tail* operation can also be implemented by either the pointer method or the copy method. Under the pointer method, *tail*(*list*) returns a pointer to the second node in the input list. After a statement such as *l* = *tail*(*list*), the pointer *list* still points to the first node, and all nodes from the second on are on both list *list* and list *l*. Under a strict copy method, a new list would be created containing copies of all nodes from the second list node onward, and *l* would point to that new list. Here, too, if the pointer method is used, then a subsequent change in either the input list (*list*) or the output list (*l*) causes a change in the other list. If the copy method is used, the two lists are independent. Note that, under the pointer method, the operation *tail*(*list*) is equivalent to *next*(*list*): both return the pointer in the *next* field of the node pointed to by the pointer *list*. Under the copy method however, an entirely new list is created by the *tail* operation.

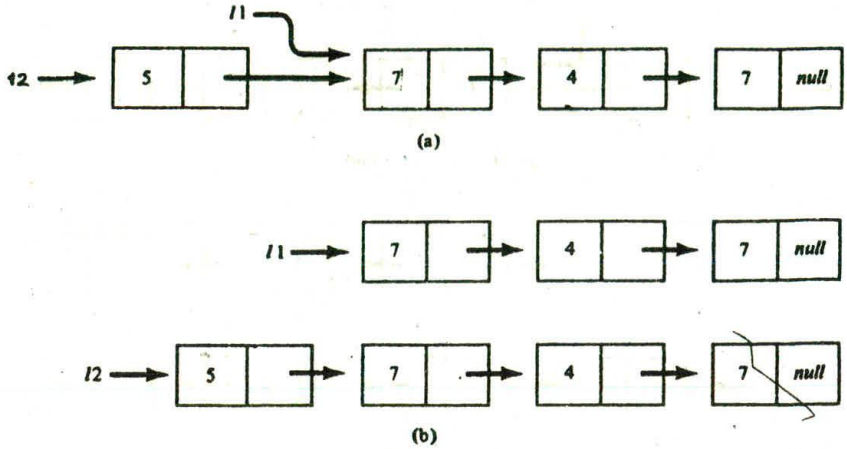


Figure 9.1.4

The operation of the copy method is similar to an operation of the assignment statement $a = b$ so that a subsequent change in b does not change a. This is because the assignment statement copies the contents of location b (the "value of b") into location a. A change in the value of b changes only the copy in location b. Similarly, in the copy method, although *l1* is a pointer, it really refers to the abstract list being represented. When a new list is formed from the old, the value of the old list is copied. The two lists are then entirely independent. In the pointer method, *l1* refers to the nodes themselves rather than the list that they collectively represent. A change in one list modifies the contents of nodes that are also part of another.

For reasons of efficiency, most list processing systems use the pointer method rather than the copy method. Imagine a 100-element list to which nodes are constantly being added (using *addon*) and from which nodes are being deleted (using *tail*). The overhead involved in both time and space in allocating and copying 100 list nodes (not to mention any list nodes on lists that appear as elements) each time that an operation is performed is prohibitive. Under the pointer method, the number of operations involved in adding or deleting an element is independent of the list size, since it involves only modification of a few pointers. However, in exchange for this efficiency, the user must be aware of possible changes to other lists. When the pointer method is used, it is common for list nodes to be used in more than one context.

In list processing systems that use the pointer method, an explicit copy operation is provided. The function

```
copy(list)
```

copies the list pointed to by *list* (including all list elements) and returns a pointer to the new copy. The user can use this operation to ensure that a subsequent modification to one list does not affect another.

Representation of Lists

So far we have ignored a number of important implementation questions: When may list nodes be freed? When must new list nodes be allocated? How are list nodes allocated and freed? For example, *settail* appends a new list to the first element of a list. But what happens to the previous tail of the list that was replaced?

These questions are related to another question: What happens when a node (or a list) is an element or a sublist of more than one list? For example, suppose the list (4,5,3,8) occurs twice as an element of a list (that is, it is the information field of two separate nodes of the list). One possibility is to maintain two copies of the list, as in Figure 9.1.5a. Or suppose that a list appears at the end of two lists as does (43,28) in Figure 9.1.5b. Although it is possible to duplicate each element whenever it appears, this often results in a needless waste of space. An alternative is to maintain the lists as in Figure 9.1.6. In this representation, a list appears only once, with all appropriate pointers pointing to the head of the single list. Under this method, a node is pointed to by more than one pointer.

If this possibility is allowed, the recursive algorithm *addone2* presented earlier to add 1 to each accessible element in a list does not work correctly. For example, if

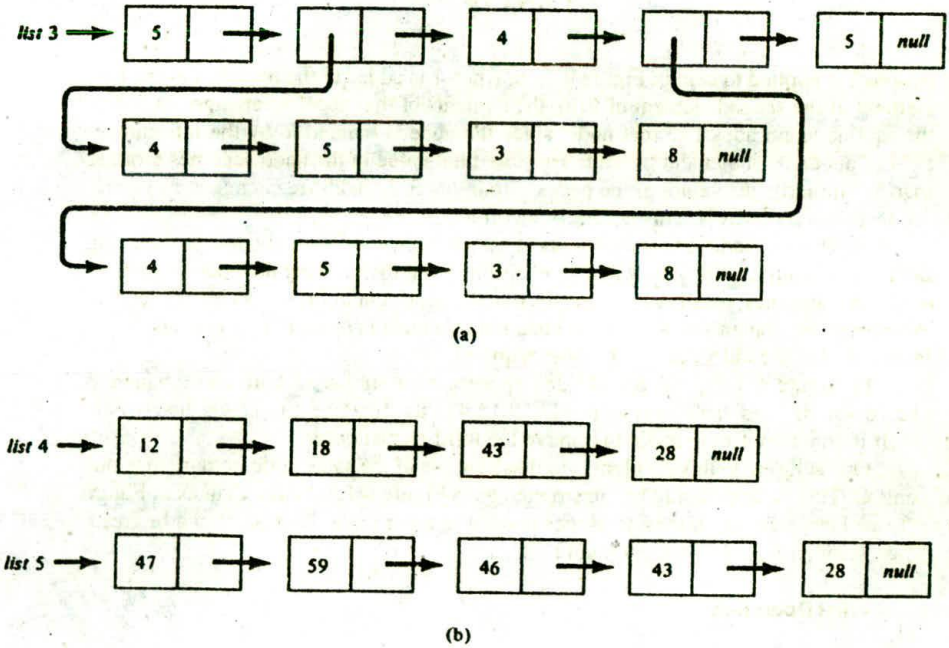


Figure 9.1.5

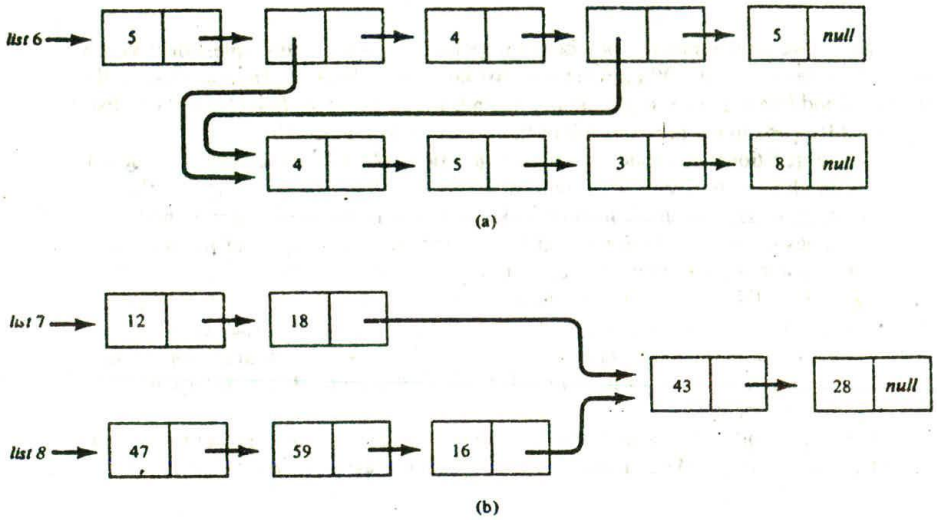


Figure 9.1.6

*addone*2 is applied to *list*6 in Figure 9.1.6a, when 1 is added to the integer 4 as the first element of the second element of *list*6, the contents of that node are changed to 5. But the routine again adds 1 to that node, since the node is pointed to by the information field of another list element as well. Thus the final value in the node becomes 6 rather than 5. Similarly, the values of the nodes containing 5, 3, and 8 are changed to 7, 5, and 10, respectively (why?). This is clearly incorrect.

In general, modifying the contents of a node from 4 to 5 is equivalent to replacing the node containing 4 by a new node containing 5 and then freeing the node containing 4. But this assumes, possibly erroneously, that the node containing 4 is no longer needed. Whenever the contents of a node are changed or a node is deleted, it is first necessary to ensure that the old value is no longer required.

In Figure 9.1.6b, the list (43,28) appears as a sublist of both *list*7, which is (12,18,43,28), and *list*8, which is (47,59,16,43,28). Imagine the chaos that would result if an attempt were made to remove the third element of *list*7.

One solution to this problem is to disallow use of the same node in more than one context. That is, lists should be constructed as in Figure 9.1.5a rather than as in Figure 9.1.6a. Then when a node is no longer needed in a particular context, it can be freed, since no other internal pointers point to it.

crlist Operation

Suppose that we want to create the list of Figure 9.1.6a. The following sequence of operations accomplishes this:

```
l = null;
l = addon(l,8);
l = addon(l,3);
l = addon(l,5);
l = addon(l,4);
list6 = null;
list6 = addon(list6,5);
list6 = addon(list6,l);
list6 = addon(list6,4);
list6 = addon(list6,l);
list6 = addon(list6,5);
```

Let us introduce the operation $l = crlist(a_1,a_2,\ldots,a_n)$, where each parameter is either a simple data item or a list pointer. This operation is defined as the sequence of statements:

```
l = null;
l = addon(l,an);
    ...
l = addon(l,a2);
l = addon(l,a1);
```

That is, $crlist(a_1,a_2,\ldots,a_n)$ creates the list $(a_1,a_2,\ldots,a_n)$. Then the foregoing sequence of operations can be rewritten as

```
l = crlist(4,5,3,8);
list6 = crlist(5,l,4,l,5);
```

Notice that this is not the same as the single operation

```
list6 = (5,crlist(4,5,3,8),4,crlist(4,5,3,8),5);
```

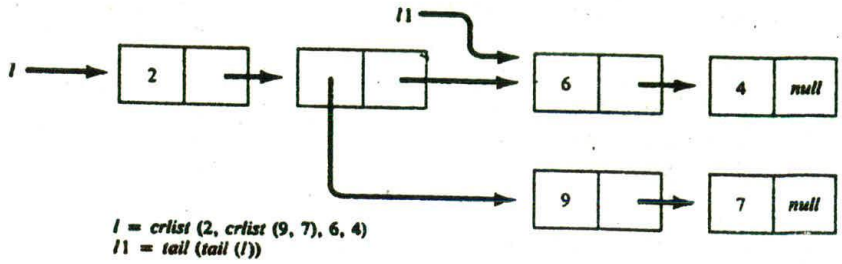
which creates two distinct copies of the list (4,5,3,8): one as its second element and one as its fourth.

We leave as an exercise for the reader the task of finding a sequence of list operations that creates the lists *list*7 and *list*8 of Figure 9.1.6b.

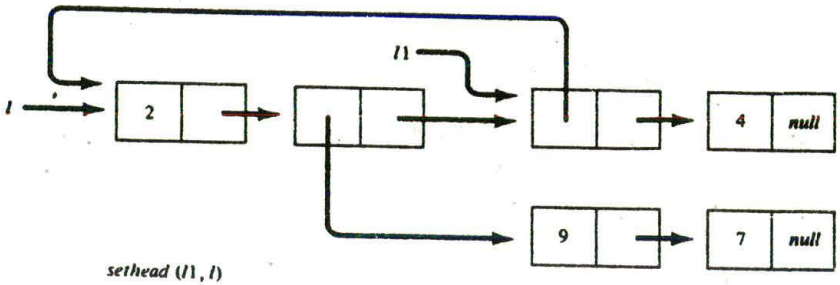
If the pointer method is used to implement list operations, it is possible to create ***recursive lists***. These are lists that contain themselves as elements. For example, suppose the following operations are performed:

```
l = crlist(2,crlist(9,7),6,4);
l1 = tail(tail(l));
sethead(l1,l);
l2 = head(tail(l));
l2 = tail(l2);
sethead(l2,l);
```

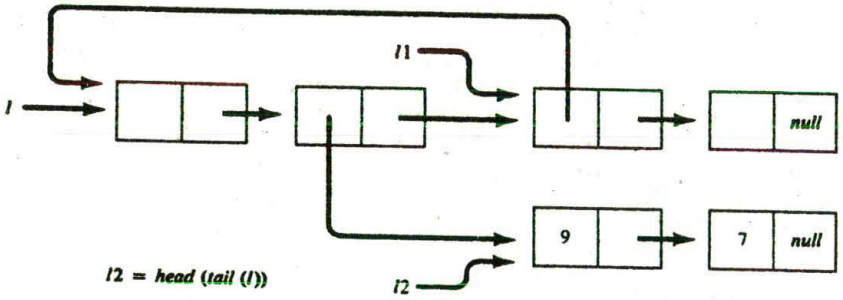
Figure 9.1.7 illustrates the effect of each of these operations. At the end of the sequence (Figure 9.1.7e) the list *l* contains itself as its third element. In addition, the second element of *l* is a list whose second element is *l* itself.



(a)

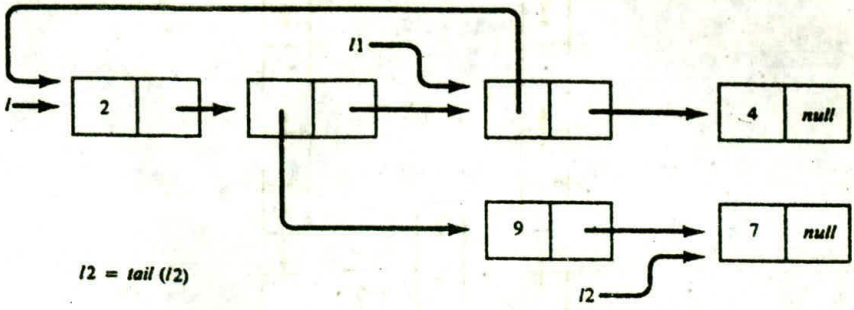


(b)

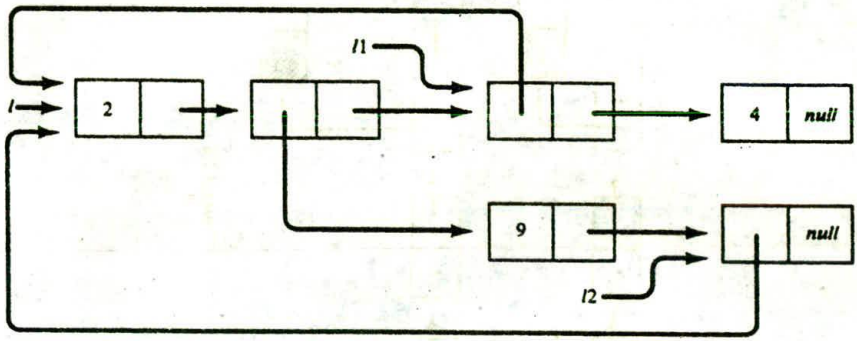


(c)

Figure 9.1.7



(d)



(e)

Figure 9.1.7 (*cont.*)

Use of List Headers

In Chapter 4 list headers were introduced as a place to store global information about an entire list. In many general list processing systems, header nodes are used for other purposes as well. We have already seen two ways of implementing general lists: the pointer method and the copy method. There is a third alternative, called the ***header method***, that is widely used in list processing systems. Under this method a header node is always placed at the beginning of any group of nodes that is to be considered a list. In particular, an external pointer always points to a header node. Similarly, if a list *l* is an element of another list, there is a header node at the front of *l*. Figure 9.1.8

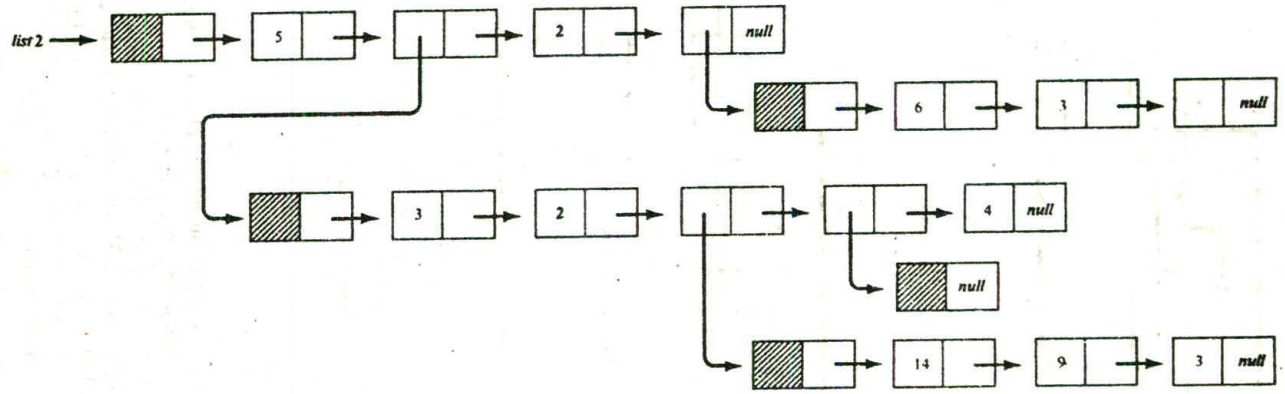


Figure 9.1.8

illustrates the list of Figure 9.1.2 using the header method. The information portion of a header node holds global information about the list (such as the number of nodes in it, or a pointer to its last node). In the figure this field is shaded. Note that a null list is now represented by a pointer to a header node containing a null pointer in its *next* field, rather than by the null pointer itself.

Any parameter that represents a list must be implemented as a pointer to a header node for that list. Any function that returns a list must be implemented so as to return a pointer to a header node.

The header method is similar to the pointer method in that a list is represented by a pointer to it. However, the presence of the header node causes significant differences (as we noted earlier in Section 4.5 when we discussed linear, circular, and doubly linked lists with headers). For example, we made a distinction between the *push* and the *addon* operations. If l is a list pointer, the function *addon*(l,x) adds a node containing x to the list pointed to by l, without changing the value of l, and returns a pointer to the new node. *push*(l,x) changes the value of its parameter l to point to the new node. Under the header method, adding an element to a list involves inserting a node between the header and the first list node. Thus despite the fact that the value of l is not changed, the list that l represents has been altered.

Freeing List Nodes

Earlier in this section we saw that a node or a set of nodes could be an element and/or a sublist of one or several lists. In such cases there is difficulty in determining when such a node can be modified or freed. Define a ***simple node*** as a node containing a simple data item (so that its *info* field does not contain a pointer). Generally, multiple use of simple nodes is not permitted. That is, operations on simple nodes are performed by the copy method rather than the pointer method. Thus any simple node deleted from a list can be freed immediately.

However, the copy method is highly inefficient when applied to nodes whose values are lists. The pointer method is the more commonly used technique when manipulating such nodes. Thus whenever a list is modified or deleted as an element or a sublist, it is necessary to consider the implications of the modification or freeing of the list on other lists that may contain it. The question of how to free a deleted list is compounded by the fact that lists may contain other lists as elements. If a particular list is freed, it may also be necessary to free all the lists that are elements of it; however if these lists are also elements of other lists, they cannot be freed.

As an illustration of the complexity of the problem, consider *list9* of Figure 9.1.9. The nodes in that figure are numbered arbitrarily so that we may refer to them easily in the text.

Consider the operation

```
list9 = null;
```

Which nodes can be freed and which must be retained? Clearly, the list nodes of *list9* (nodes 1,2,3,4) can be freed, since no other pointers reference them. Freeing node 1

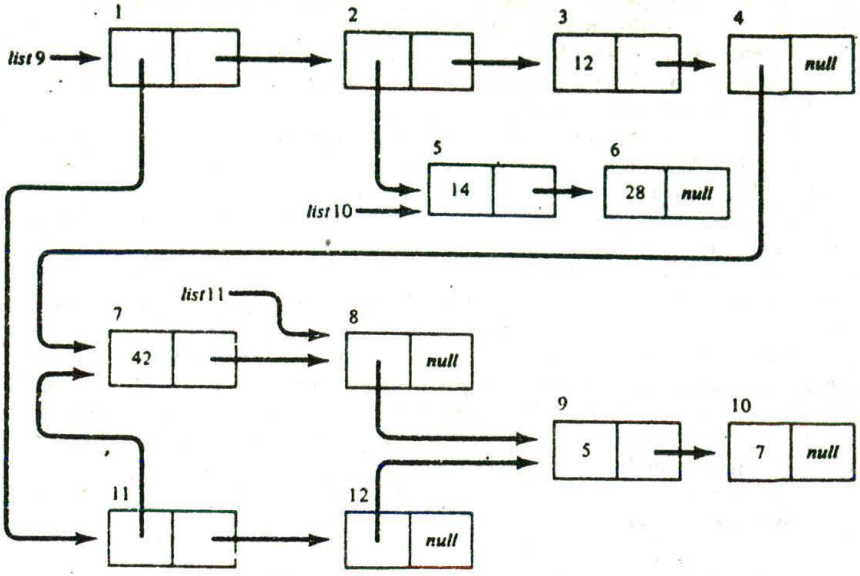


Figure 9.1.9

allows us to free nodes 11 and 12, since they too are accessed by no other pointers. Once node 11 is freed, can nodes 7 and 8 also be freed? Node 7 can be freed because each of the nodes containing a pointer to it (nodes 11 and 4) can be freed. However, node 8 cannot be freed, since *list*11 points to it. *list*11 is an external pointer; therefore the node to which it points may still be needed elsewhere in the program. Since node 8 is kept, nodes 9 and 10 must also be kept (even though node 12 is being freed). Finally, nodes 5 and 6 must be kept because of the external pointer *list*10.

The problem to be addressed in the next section is how to determine algorithmically which nodes are to be kept and which are to be freed. However, before considering possible solutions, let us consider how lists can be implemented in C and make some comments about list processing languages and their design.

General Lists in C

Because general list nodes can contain simple data elements of any type or pointers to other lists, the most direct way to declare list nodes is by using unions. One possible implementation is as follows:

```
#define INTGR 1
#define CH    2
#define LST   3
```

```
struct nodetype {
   int utype;       /* utype equals INTGR, CH,  or LST */
   union {
      int intgrinfo;                  /* utype = INTGR */
      char charinfo;                  /* utype = CH    */
      struct nodetype *lstinfo;       /* utype = LST   */
   } info;
   struct nodetype *next;
};
typedef struct nodetype *NODEPTR;
```

Each list node had three elementary fields: a flag (*utype*) to indicate the type of the information field, the actual information field and a pointer to the next node in the list. The operation *nodetype*(*p*) is implemented by simply referencing $p->utype$.

The actual implementation of any list operation depends on whether the system in question is implemented using the pointer method, the copy method, or the header method. We consider the pointer method.

The *tail* operation always produces a pointer to a list (possibly the *null* pointer), assuming that its argument points to a valid list. Thus this operation may be implemented as a simple C function:

```
NODEPTR tail(NODEPTR list)
{
   if (list == NULL) {
      printf("illegal tail operation");
      exit(1);
   }
   else
      return(list->next);
} /* end tail */
```

We might be tempted to implement the *head* operation in similar fashion, by simply returning the value in the *info* field of the node to which the parameter points. However, because some versions of C do not allow a function to return a structure, we implement the *head* operation as a function with two parameters: an input parameter that holds a pointer to the input list and an output parameter that points to a structure containing the information. Let us assume that a structure has been declared

```
struct {
   int utype;
   union {
      int intgrinfo;
      char charinfo;
      struct nodetype *listinfo;
   } info;
} infotype;
typedef struct infotype *INFOPTR;
```

Then the function *head*, which is invoked by a statement such as *head(list1, &item)*, is implemented as follows:

```
void head(NODEPTR list, INFOPTR pitem)
{
   if (list == NULL) {
      printf ("illegal head operation");
      exit(1);
   } /* end if */
   pitem = (INFOPTR) malloc(sizeof (struct infotype));
   pitem->utype = list->utype;
   pitem->info = list->info;
   return;
} /* end head */
```

In some applications it may not be necessary to return the value of the contents of the information portion of the node; it may be sufficient to identify a pointer to the desired node. In such a case the value of the pointer variable traversing the list may be used instead of the *head* routine. Note that neither the *head* nor *tail* operations change the original list in any way. All fields retain the same values that they had before the routines were called.

Once the basic forms of *head* and *tail* have been implemented, other list operations can be implemented either in terms of these operations or by accessing list nodes directly. For example, the *addon* operation may be implemented as follows:

```
NODEPTR addon(NODEPTR list, struct infotype *pitem)
{
   NODEPTR newptr;

   newptr = (INFOPTR) malloc (sizeof (struct nodetype));
   newptr->utype = pitem->utype;
   newptr->info = pitem->info;
   newptr->next = list;
   return (newptr);
} /* end addon */
```

Now that *addon* has been implemented, *sethead(list, item)* can be implemented by the statement *list = addon(tail (list), &item)*, as we mentioned earlier. Alternatively, the *sethead* operation can be implemented directly by the sequence of statements:

```
list->utype = item.utype;
list->info = item.info;
```

The *settail* operation may be implemented similarly as follows, using an auxiliary routine *freelist* to free the previous tail of *list*:

```
void settail (NODEPTR *plist, NODEPTR t1)
{
   NODEPTR p;
```

```
    p = *plist->next;
    *plist->next = tl;
    freelist(p);
} /* end settail */
```

Despite the fact that we are using the pointer method, more likely than not, there is no need for the previous tail of *list*. If some other pointer points to this portion of the list then the call to *freelist* should be omitted.

The implementation of *settail* highlights the problem of automatic list management and how to determine when a node should be freed if it may indeed appear in more than one context (as in the pointer method or if recursive lists are permitted). As mentioned, we examine these issues in Section 9.2.

Programming Languages and Lists

Throughout this text we have been treating a list as a compound data structure (a collection of nodes), rather than as a native data type (an elementary item such as *int* and *char*). The reason for this is that we have been working closely with the C language. In C, one cannot make a declaration such as

```
list x;
```

and apply such functions as *head* and *tail* to *x* directly. Rather, the programmer must implement lists by writing the necessary procedures and functions for their manipulation. Other languages, however, do contain lists as elementary data structures with the operations *crlist*, *head*, *tail*, *addon*, *sethead*, and *settail* already built into the language. (A good example of such a language is LISP.)

One consequence of the fact that C does not include list manipulation capabilities is that if a programmer programs a list manipulation application, it is the programmer's responsibility to allocate and free the necessary list nodes. As we have seen in this section, that problem is not at all trivial if lists are allowed in all their generality. However, any given application can usually be designed more easily using a specific type of list, tree or graph, as we have seen in Chapters 4, 5, and 8. Indeed, general list manipulation techniques are more expensive in terms of both time and space than techniques designed specifically for a particular application. (This is a corollary to the axiom that a price is always paid for generality.) Thus the C programmer will rarely have occasion to use general list manipulation techniques.

However, a general list processing system, in which the list is a native data type and list operations are built-in, must be able to deal with lists in all their generality. Since the fundamental objects are lists and data items rather than nodes, the programmer cannot be responsible for allocating and freeing individual nodes. Rather, when a program issues a statement such as

```
l1 = crlist(3,4,7);
```

the system is responsible for allocating sufficient list nodes and initializing the proper pointers. When the program later issues the command

```
l1 = NULL;
```

the system is responsible for identifying and freeing those nodes previously on list *l*1 that now become inaccessible. If such nodes are not freed, available space would rapidly become exhausted.

In some sense, languages that include lists as native data types are of "higher level" than C because the programmer is freed from so much of the bookkeeping activity associated with storage management. C may be thought of as a language of higher level than FORTRAN, in that C includes data structures such as structures and unions, whereas FORTRAN does not. So too, a list processing system is of higher level than C in that it includes lists, whereas C does not.

Another point that should be made concerns the implementation of lists. The implementation of lists as presented in this section is oriented toward C. Because C permits the use of unions, it was possible to define a type *infotype* to contain any of the legal data types in our list system. Some languages (for example, PL/I) do not support unions. In such languages, the type of a node (with certain limited exceptions) is fixed in advance. In such languages it would be necessary to separate a list system into ***list nodes*** and ***atomic nodes***. An atomic node is a node that contains no pointers—only a simple data item. Several different types of atomic nodes would exist, each with a single data item corresponding to one of the legal data types. A list node contains a pointer to an atomic node and a type indicator indicating the type of atomic node to which it points (as well as a pointer to the next node on the list, of course). When it is necessary to place a new node on a list, an atomic node of the appropriate type must be allocated, its value must be assigned, the list node information field must be set to point to the new atomic node, and the type field in the list node must be set to the proper type.

To understand how clumsy this situation is, suppose that there are ten different types of atomic nodes (there is no reason that an atomic node may not be an array or a stack or a queue or a program label, for example). Each of these must have a unique typecode. Further there must be a separate variable declared for each type of atomic node. Let us suppose that the typecodes used for the ten types are *t*1, *t*2, . . . , *t*10 and that the atomic node variables are *node*1, *node*2, . . . , *node*10. Then each time that an atomic node is processed, we would need code such as

```
switch (typecode) {
   case  t1: /*do something with node1*/
   case  t2: /*do something with node2*/
             ...
   case t10: /*do something with node10*/
} /* end switch */
```

This is a cumbersome organization, and one which we are able to avoid by using unions.

In the next section of this chapter we examine techniques incorporated into list processing systems to recover storage that is no longer needed. We retain the list structure conventions of this section, but it should be understood that they are not absolute.

EXERCISES

9.1.1. How would you implement a general stack and queue in C? Write all the routines necessary for doing so.

9.1.2. Implement the routines *addon*, *sethead*, *settail*, and *crlist* in C.

9.1.3. Write a C subroutine *freelist*(*list*) that frees all nodes accessible from a pointer *list*. If your solution is recursive, rewrite it nonrecursively.

9.1.4. Rewrite *addone*2 so that it is nonrecursive.

9.1.5. Write a C routine *dlt*(*list*, *n*) that deletes the *n*th element of a *list*. If this *n*th element is itself a list, all nodes accessible through that list should be freed. Assume that a list can appear in only one position.

9.1.6. Implement the function *copy*(*list*) in C. This routine accepts a pointer *list* to a general list and returns a pointer to a copy of that list. What if the list is recursive?

9.1.7. Write a C routine that accepts a list pointer and prints the parenthesized notation for that list. Assume that list nodes can appear only on a single list, and that recursive lists are prohibited.

9.1.8. What are the advantages and disadvantages of languages in which the type of variables need not be declared, as compared with languages such as C?

9.1.9. Write two sets of list operations to create the lists of Figures 9.1.4b and 9.1.9.

9.1.10. Redraw all the lists of this section that do not include header nodes so that they are now included.

9.1.11. Implement the routines *addon*, *head*, *tail*, *sethead*, and *settail* in C for lists using the following methods.
- (a) Copy method
- (b) Header method

9.1.12. Implement the list operations for a system that uses doubly linked lists.

9.2 AUTOMATIC LIST MANAGEMENT

In the last section we presented the need for algorithms to determine when a given list node is no longer accessible. In this section we investigate such algorithms. The philosophy behind incorporating such an algorithm into a programming system is that the programmer should not have to decide when a node should be allocated or freed. Instead, the programmer should code the solution to the problem with the assurance that the system will automatically allocate any list nodes that are necessary for the lists being created and that the system will make available for reuse any nodes that are no longer accessible.

There are two principal methods used in automatic list management: the ***reference count*** method and the ***garbage collection*** method. We proceed to a discussion of each.

Reference Count Method

Under this method each node has an additional *count* field that keeps a count (called the ***reference count***) of the number of pointers (both internal and external) to that node. Each time that the value of some pointer is set to point to a node, the reference

count in that node is increased by 1; each time that the value of some pointer that had been pointing to a node is changed, the reference count in that node is decreased by 1. When the reference count in any node becomes 0, that node can be returned to the available list of free nodes.

Each list operation of a system using the reference count method must make provision for updating the reference count of each node that it accesses and for freeing any node whose count becomes 0. For example, to execute the statement

```
l = tail(l);
```

the following operations must be performed:

```
p = l;
l = next(l);
next(p) = null;
reduce(p);
```

where the operation *reduce*(*p*) is defined recursively as follows:

```
if (p != null) {
   count(p)--;
   if (count(p) == 0) {
      r = next(p);
      reduce(r);
      if (nodetype(p) == lst)
         reduce(head(p));
      free node(p);
   } /* end if */
} /* end if */
```

reduce must be invoked whenever the value of a pointer to a list node is changed. Similarly, whenever a pointer variable is set to point to a list node, the *count* field of that node must be increased by 1. The *count* field of a free node is 0.

To illustrate the reference count method, consider again the list of Figure 9.1.9. The following set of statements creates that list:

```
list10 = crlist(14,28);
list11 = crlist(crlist(5,7));
l1 = addon(list11,42);
m = crlist(l1,head(list11));
list9 = crlist(m,list10,12,l1);
m = null;
l1 = null;
```

Figure 9.2.1 illustrates the creation of the list using the reference count method. Each part of that figure shows the list after an additional group of the foregoing statements has been executed. The reference count is shown as the leftmost field of each list node. Each node in that figure is numbered according to the numbering of the nodes in

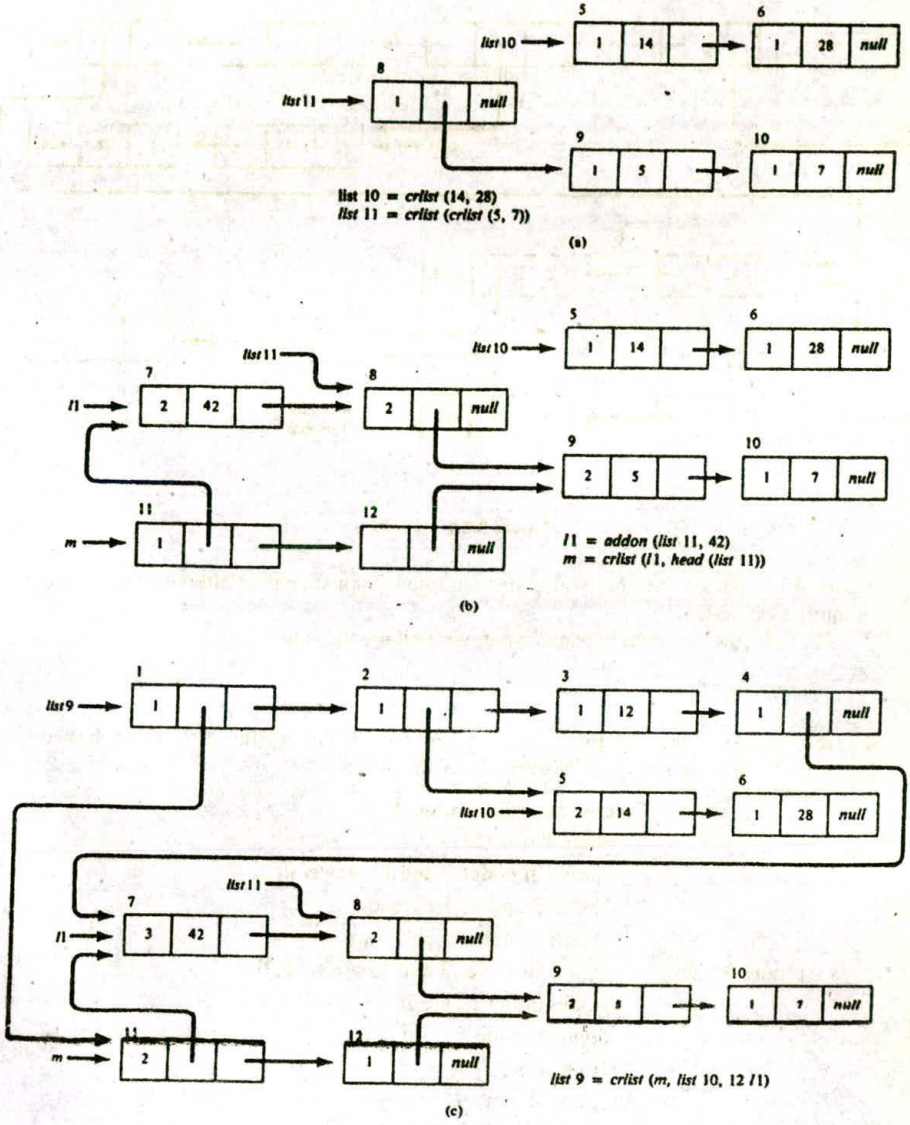

list10
list11
5
6
8
9
10
1
14
28
5
7
null
list 10 = crlist (14, 28)
list 11 = crlist (crlist (5, 7))
(a)
l1
m
7
11
12
2
42
addon (list 11, 42)
m = crlist (l1, head (list 11))
(b)
list9
1
2
3
4
12
(c)
list 9 = crlist (m, list 10, 12 l1)

Figure 9.2.1

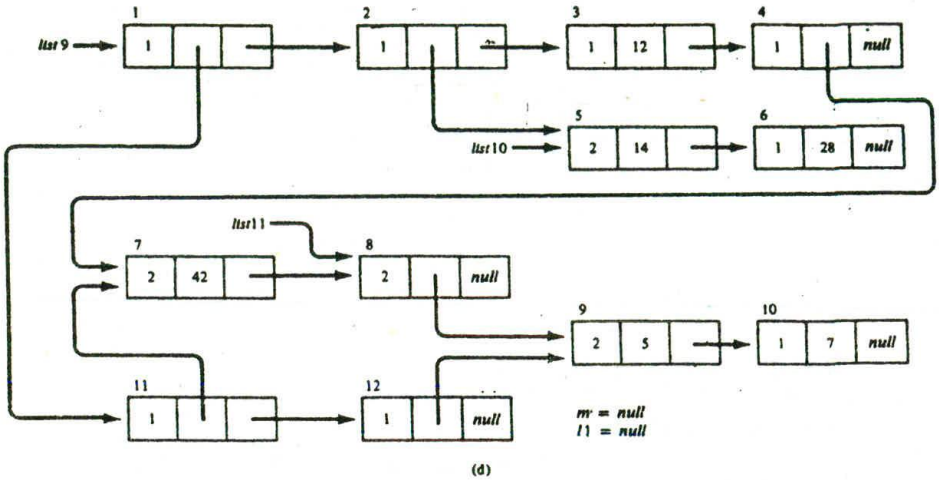


Figure 9.2.1 (*cont.*)

Figure 9.1.9. Make sure that you understand how each statement alters the reference count in each node.

Let us now see what happens when we execute the statement

```
list9 = null;
```

The results are illustrated in Figure 9.2.2, where freed nodes are illustrated using dashed lines. The following sequence of events may take place:

count of node 1 is set to 0.
Node 1 is freed.
counts of nodes 2 and 11 are set to 0.
Nodes 2 and 11 are freed.
counts of nodes 5 and 7 are set to 1.
(Figure 9.2.2a) *counts* of nodes 3 and 12 are set to 0.
Nodes 3 and 12 are freed.
count of node 4 is set to 0.
Node 4 is freed.
count of node 9 is set to 1.
count of node 7 is set to 0.
Node 7 is freed.
(Figure 9.2.2b) *count* of node 8 is set to 1.

Only those nodes accessible from the external pointers *list*10 and *list*11 remain allocated; all others are freed.

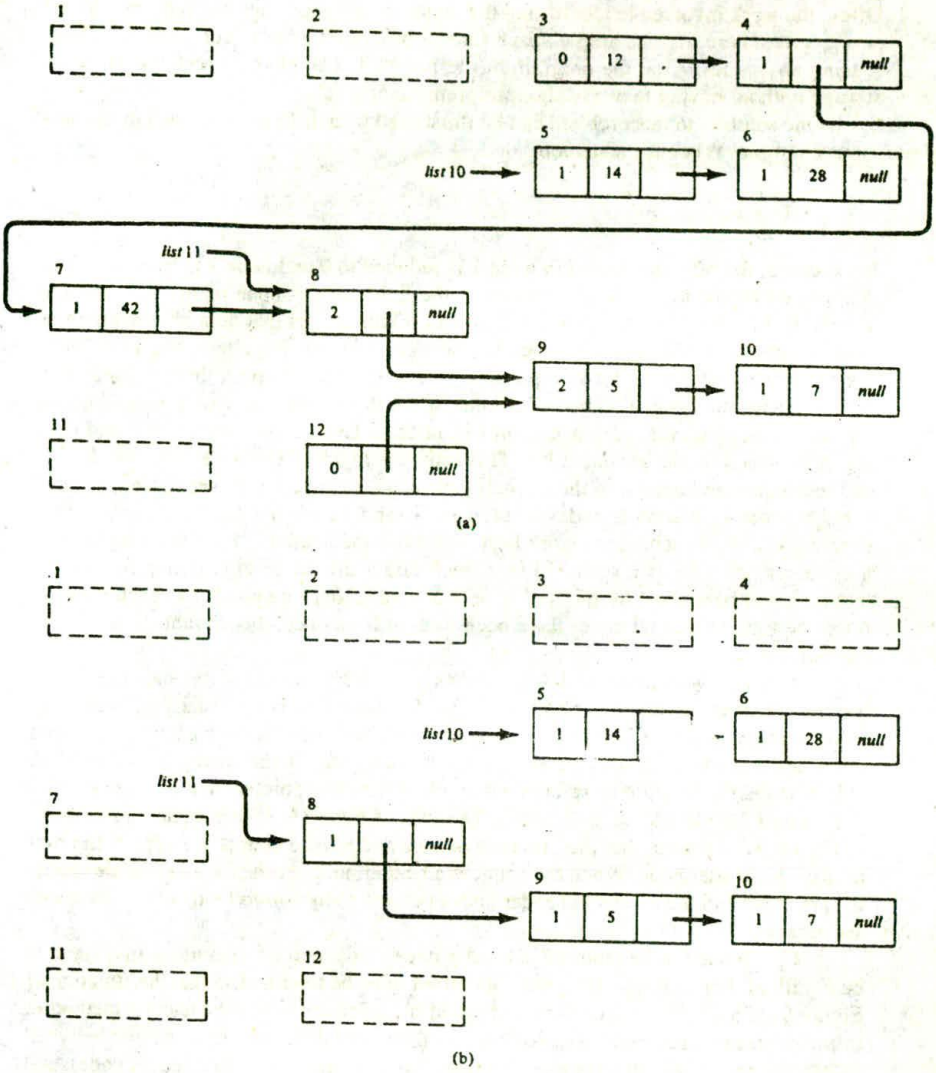


Figure 9.2.2

One drawback of the reference count method is illustrated by the foregoing example. The amount of work that must be performed by the system each time that a list manipulation statement is executed can be considerable. Whenever a pointer value is changed, all nodes previously accessible from that pointer can potentially be freed.

Often, the work involved in identifying the nodes to be freed is not worth the reclaimed space, since there may be ample space for the program to run to completion without reusing any nodes. After the program has terminated, a single pass reclaims all of its storage without having to worry about reference count values.

One solution to this problem can be illustrated by a different approach to the previous example. When the statement

```
list9 = null
```

is executed, the reference count in node 1 is reduced to 0 and node 1 is freed—that is, it is placed on the available list. However, the fields of this node retain their original values, so that it still points to nodes 2 and 11. (This means that an additional pointer field is necessary to link such nodes on the available list. An alternative is to reuse the reference count field for this purpose.) The reference count values in these two nodes remain unchanged. When additional space is needed and node 1 is reallocated for some other use, the reference counts in nodes 2 and 11 are reduced to 0 and they are then placed on the available list. This removes much of the work from the deallocation process and adds it to the allocation process. If node 1 is never reused because enough space is available, nodes 2, 11, 3, 4, 7, and 12 are not freed during program execution. For this scheme to work best, however, the available list should be kept as a queue rather than as a stack, so that freed nodes are never allocated before nodes that have not been used for the first time. (Of course, once a system has been running for some time so that all nodes have been used at least once, this advantage no longer exists.)

There are two additional disadvantages to the reference count method. The first is the additional space required in each node for the count. This is not usually an overriding consideration, however. The problem can be somewhat alleviated if each list is required to contain a header node and a reference count is kept only in the header. However, then only a header node could be referenced by more than one pointer (that is, a list such as in Figure 9.2.3b would be prohibited). The lists of Figure 9.2.3 are analogous to those of Figure 9.1.6 except that they include header nodes. The counts are kept in the first field of the header node. When the count in a header node reaches 0, all the nodes on its list are freed and the counts in header nodes pointed to by *lstinfo* fields in the list nodes are reduced.

If counts are to be retained in header nodes only, certain operations may have to be modified. For example, the *settail* operation must be modified so that the situation of Figure 9.2.3b does not occur. One method of modification is to use the copy method in implementing these operations. Another method is to differentiate somehow between external pointers, which represent lists (and therefore must point to a header node), and "temporary" external pointers, which are used for traversal (and can point directly to list nodes). When the count in a header node becomes 0, references to its list nodes through temporary pointers become illegal.

The other disadvantage of the reference count method is that the count in the first node of a recursive or circular list will never be reduced to 0. Of course, whenever a pointer within a list is set to point to a node on that list, the reference count can be maintained rather than increased, but detecting when this is so is often a difficult task.

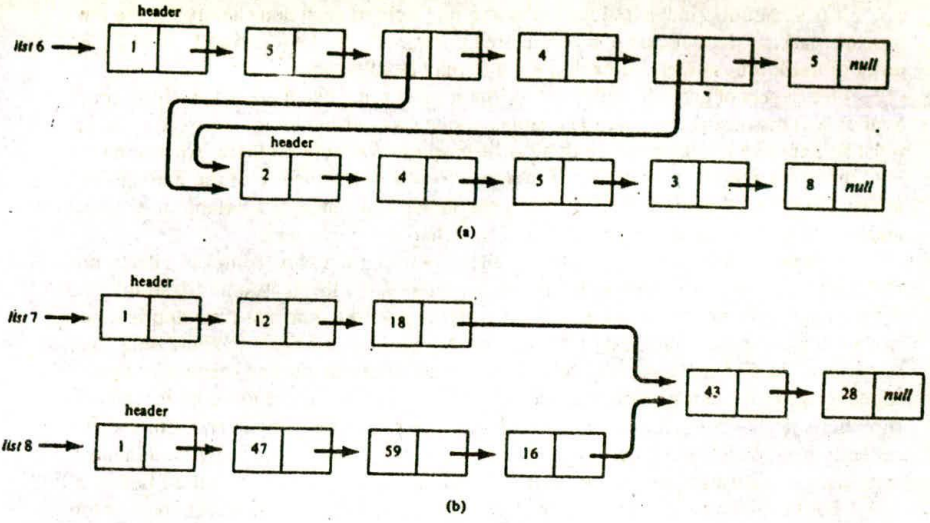


Figure 9.2.3

Garbage Collection

Under the reference count method, nodes are reclaimed when they become available for reuse (or under one version when they are needed). The other principal method of detecting and reclaiming free nodes is called ***garbage collection***. Under this method, nodes no longer in use remain allocated and undetected until all available storage has been allocated. A subsequent request for allocation cannot be satisfied until nodes that had been allocated but are no longer in use are recovered. When a request is made for additional nodes and there are none available, a system routine called the ***garbage collector*** is called. This routine searches through all of the nodes in the system, identifies those that are no longer accessible from an external pointer, and restores the inaccessible nodes to the available pool. The request for additional nodes is then fulfilled with some of the reclaimed nodes and the system continues processing user requests for more space. When available space is used up again, the garbage collector is called once more.

Garbage collection is usually done in two phases. The first phase, called the ***marking phase***, involves marking all nodes that are accessible from an external pointer. The second phase, called the ***collection phase***, involves proceeding sequentially through memory and freeing all nodes that have not been marked. We examine the marking phase first and then turn our attention to the collection phase.

One field must be set aside in each node to indicate whether a node has or has not been marked. The marking phase sets the mark field to *true* in each accessible node. As the collection phase proceeds, the mark field in each accessible node is reset to *false*. Thus, at the start and end of garbage collection, all mark fields are *false*. User programs do not affect the mark fields.

It is sometimes inconvenient to reserve one field in each node solely for the purpose of marking. In that case a separate area in memory can be reserved to hold a long array of mark bits, one bit for each node that may be allocated.

One aspect of garbage collection is that it must run when there is very little space available. This means that auxiliary tables and stacks needed by the garbage collector must be kept to a minimum since there is little space available for them. An alternative is to reserve a specific percentage of memory for the exclusive use of the garbage collector. However, this effectively reduces the amount of memory available to the user and means that the garbage collector will be called more frequently.

Whenever the garbage collector is called, all user processing comes to a halt while the algorithm examines all allocated nodes in memory. For this reason it is desirable that the garbage collector be called as infrequently as possible. For real-time applications, in which a computer must respond to a user request within a specific short time span, garbage collection has generally been considered an unsatisfactory method of storage management. We can picture a spaceship drifting off into the infinite as it waits for directions from a computer occupied with garbage collection. However, methods have recently been developed whereby garbage collection can be performed simultaneously with user processing. This means that the garbage collector must be called before all space has been exhausted so that user processing can continue in whatever space is left, while the garbage collector recovers additional space.

Another important consideration is that users must be careful to ensure that all lists are well formed and that all pointers are correct. Usually, the operations of a list processing system are carefully implemented so that if garbage collection does occur in the middle of one of them, the entire system still works correctly. However, some users try to outsmart the system and implement their own pointer manipulations. This requires great care so that garbage collection will work properly. In a real-time garbage collection system, we must ensure not only that user operations do not upset list structures that the garbage collector must have but also that the garbage collection algorithm itself does not unduly disturb the list structures that are being used concurrently by the user. As we shall see, some marking algorithms do disturb (temporarily) list structures and are therefore unsuitable for real-time use.

It is possible that, at the time the garbage collection program is called, users are actually using almost all the nodes that are allocated. Thus almost all nodes are accessible and the garbage collector recovers very little additional space. After the system runs for a short time, it will again be out of space; the garbage collector will again be called only to recover very few additional nodes, and the vicious cycle starts again. This phenomenon, in which system storage management routines such as garbage collection are executing almost all the time, is called ***thrashing***.

Clearly, thrashing is a situation to be avoided. One drastic solution is to impose the following condition. If the garbage collector is run and does not recover a specific percentage of the total space, the user who requested the extra space is terminated and removed from the system. All of that user's space is then recovered and made available to other users.

Algorithms for Garbage Collection

The simplest method for marking all accessible nodes is to mark initially all nodes that are immediately accessible (that is, those pointed to by external pointers) and then

repeatedly pass through all of memory sequentially. On each sequential pass, whenever a marked node *nd* is encountered, all nodes pointed to by a pointer within *nd* are marked. These sequential passes continue until no new nodes have been marked in an entire pass. Unfortunately, this method is as inefficient as it is simple. The number of sequential passes necessary is equal to the maximum path length to any accessible node (why?), and on each pass every list node in memory must be examined. However, this method requires almost no additional space.

A somewhat more efficient variation is the following: Suppose that a node $n1$ in the sequential pass has been previously marked and that $n1$ includes a pointer to an unmarked node, $n2$. Then node $n2$ is marked and the sequential pass would ordinarily continue with the node that follows $n1$ sequentially in memory. However, if the address of $n2$ is less than the address of $n1$, the sequential pass resumes from $n2$ rather than from $n1$. Under this modified technique, when the last node in memory is reached, all accessible nodes have been marked.

Let us present this method as an algorithm. Assume that all list nodes in memory are viewed as a sequential array.

```
#define NUMNODES ...;
struct nodetype {
   int mark;
   int utype;
   union {
      int intgrinfo;
      char charinfo;
      int lstinfo;
   } info;
   int next;
} node[NUMNODES];
```

An array *node* is used to convey the notion that we can step through all nodes sequentially. *node*[0] is used to represent a dummy node. We assume that *node*[0].*lstinfo* and *node*[0].*next* are initialized to 0, *node*[0].*mark* to *true*, and *node*[0].*utype* to *lst*, and that these values are never changed throughout the system's execution. The *mark* field in each node is initially *false* and is set to *true* by the marking algorithm when a node is found to be accessible.

Now that we have defined the format of our nodes, we turn to the actual algorithm. Assume that *acc* is an array containing external pointers to immediately accessible nodes, declared by

```
#define NUMACC = ...;
int acc[NUMACC];
```

The marking algorithm is as follows:

```
/* mark all immediately accessible nodes */
for (i = 0; i < NUMACC; i++)
  node[acc[i]].mark = TRUE;
```

```
/* begin a sequential pass through the array of nodes */
/*   i points to the node currently being examined    */
i = 1;
while (i < NUMNODES) {
   j = i + 1; /* j points to the node to be examined next */
   if (node[i].mark) {
      /* mark nodes to which i points */
      if (node[i].utype == LST &&
                     node[node[i].lstinfo].mark != TRUE) {
         /* the information portion of i */
         /* points to an unmarked node */
         node[node[i].lstinfo].mark = TRUE;
         if (node[i].lstinfo < j)
            j = node[i].lstinfo;
      } /* end if */
      if (node[node[i].next].mark != TRUE) {
         /* the list node following */
         /* node[i] is an unmarked node */
         node[node[i].next].mark = TRUE;
         if (node[i].next < j)
            j = node[i].next;
      } /* end if */
   } /* end if */
   i = j;
} /* end while */
```

In the exercises you are asked to trace the execution of this algorithm on a list distributed throughout memory such as *list*9 in Figure 9.1.9.

Although this method is better than successive sequential passes, it is still inefficient. Consider how many nodes must be examined if *node*[1] is immediately accessible and points to *node*[999], which points to *node*[2], and so on. Thus it is usually too slow to use in an actual system.

A more desirable method is one that is not based on traversing memory sequentially but rather traverses all accessible lists. Thus it examines only those nodes that are accessible, rather than all nodes.

The most obvious way to accomplish this is by use of an auxiliary stack and is very similar to depth-first traversal of a graph. As each list is traversed through the *next* fields of its constituent nodes, the *utype* field of each node is examined. If the *utype* field of a node is *lst*, the value of the *lstinfo* field of that node is placed on the stack. When the end of a list or a marked node is reached, the stack is popped and the list headed by the node at the top of the stack is traversed. In the algorithm that follows, we again assume that *node*[0].*mark* = *true*.

```
for (i = 0; i < NUMACC; i++) {
   /* mark the next immediately accessible */
   /* node and place it on the stack */
```

```
        node[acc[i]].mark = TRUE;
        push(stack, acc[i]);
        while (empty(stack) != TRUE) {
           p = pop(stack);
           while (p != 0) {
              if (node[p].utype == LST &&
                          node[node[p].lstinfo].mark != TRUE) {
                 node[node[p].lstinfo].mark = TRUE;
                 push(stack, node[p].lstinfo);
              } /* end if */
              if (node[node[p].next].mark == TRUE)
                 p = 0;
              else {
                 p = node[p].next;
                 node[p].mark = TRUE;
              } /* end if */
           } /* end while */
        } /* end while */
     } /* end for */
```

This algorithm is as efficient as we can hope for in terms of time, since each node to be marked is visited only once. However, it has a significant weakness because of its dependence on an auxiliary stack. A garbage collection algorithm is called when there is no extra space available, so where is the stack to be kept? Since the size of the stack is never greater than the depth of the list nesting and lists are rarely nested beyond some reasonable limit (such as 100), a specific number of nodes reserved for the garbage collection stack would suffice in most cases. However, there is always the possibility that some user would want to nest nodes more deeply.

One solution is to use a stack limited to some maximum size. If the stack is about to overflow, we can revert to the sequential method given in the previous algorithm. We ask the reader to work out the details in an exercise.

Another solution is to use the allocated list nodes themselves as the stack. Clearly, we do not want to add an additional field to each list node to hold a pointer to the next node on the stack, since the extra space could be better used for other purposes. Thus either the *lstinfo* field or *next* field of the list nodes must be used to link together the stack. But this means that the list structure is temporarily disturbed. Provision must be made for the lists to be restored properly.

In the foregoing algorithm, each list is traversed using the *next* fields of its nodes, and the value of each pointer *lstinfo* to a list node is pushed onto a stack. When either the end of a list or a section of the list which had already been marked is reached, the stack is popped and a new list is traversed. Therefore, when a pointer to a node *nd* is popped, there is no need to restore any of the fields within *nd*.

However, suppose that the stack is kept as a list, linked by the *next* fields. Then when a node is pushed onto the stack, its *next* field must be changed to point to the top node in the stack. This implies that the field must be restored to its original value when the node is popped. But that original value has not been saved anywhere. (It cannot be saved on the stack, since there is no extra storage available for it.)

A solution to this problem can be described by the following scheme. Let us first assume a list with no elements that are themselves lists. As each node in the list is visited, it is pushed onto the stack and its *next* field is used to link it onto the stack. Since each node preceding the current node on the list is also present on the stack (the top of the stack is the last encountered element on the list), the list can be reconstructed easily by simply popping the stack and restoring the *next* fields.

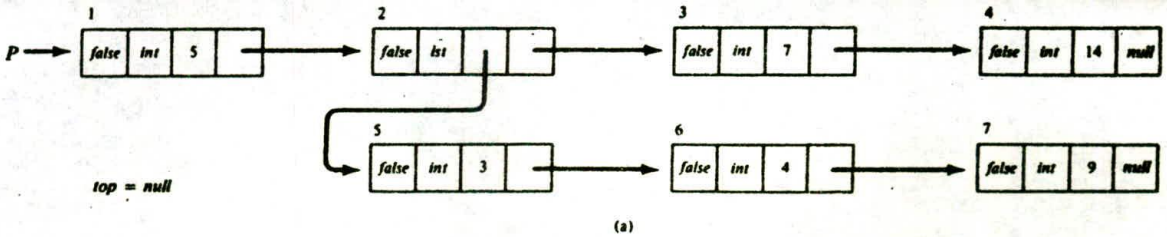
The situation is only slightly different in the case where one list is an element of another. Suppose that *nd*1 is a node on *list*1, that *nd*2 is a node on *list*2, and that *node*[*nd*1].*lstinfo* = *nd*2. That is, *nd*2 is the first node of *list*2, where *list*2 is an element of *list*1. The algorithm has been traversing *list*1 and is now about to begin traversing *list*2. In this case *node*[*nd*1].*next* cannot be used as a stack pointer because it is needed to link *nd*1 to the remainder of *list*1. However, the *listinfo* field of *nd*1 can be used to link *nd*1 onto the stack, since it is currently being used to link to *nd*2.

In general, when a node *nd* is pushed onto the stack, either its *lstinfo* field or its *next* field is used to point to the previous top element. If the next node to be examined is pointed to by *node*[*nd*].*lstinfo*, the *lstinfo* field is used to link *nd* onto the stack, whereas if the node is pointed to by *node*[*nd*].*next*, the *next* field is used to link *nd* onto the stack. The remaining problem is how to determine for a given node on the stack whether the *lstinfo* or *next* field is used to link the stack.

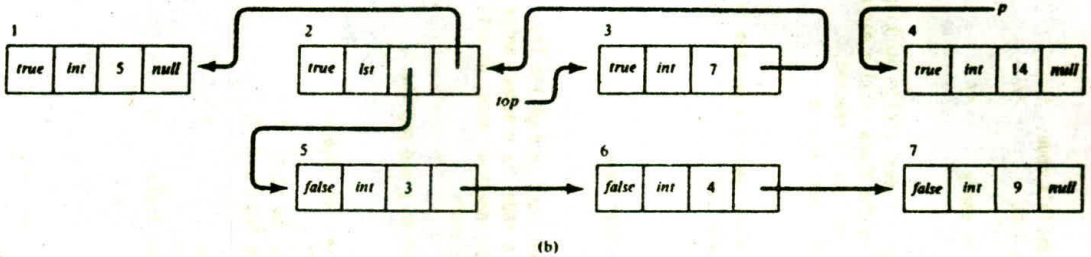
If the *utype* field of a node indicates that the node is a simple node, its *next* field must be in use as a stack pointer. (This is because the node ha no *lstinfo* field that must be traversed.) However, a node with a *utype* field of *lst* is not so easily handled. Suppose that each time the *lstinfo* field is used to advance to the next node, the *utype* field in the list node is changed from *lst* to some new code (say *stk* for stack) that is neither *lst* nor any of the codes that denote simple elements. Then when a node is popped from the stack, if its *utype* field is not *stk*, its *next* field must be restored, and if its *tag* field is *stk*, its *lstinfo* field must be restored and the *utype* field restored to *lst*.

Figure 9.2.4 illustrates how this stacking mechanism works. Figure 9.2.4a shows a list before the marking algorithm begins. The pointer *p* points to the node currently being processed, *top* points to the stack top, and *q* is an auxiliary pointer. The mark field is shown as the first field in each node. Figure 9.2.4b shows the same list immediately after node 4 has been marked. The path taken to node 4 is through the *next* fields of nodes 1, 2, and 3. This path can be retraced in reverse order, beginning at *top* and following along the *next* fields. Figure 9.2.4c shows the list after node 7 has been marked. The path to node 7 from the beginning of the list was from node 1, through *node*[1].*next* to node 2, through *node*[2].*lstinfo* to node 5, through *node*[5].*next* to node 6, and then from *node*[6].*next* to node 7. The same fields that link together the stack are used to restore the list to its original form. Note that the *utype* field in node 2 is *stk* rather than *lst* to indicate that its *lstinfo* field, not its *next* field, is being used as a stack pointer. The algorithm that incorporates these ideas is known as the Schorr–Waite algorithm, after its discoverer.

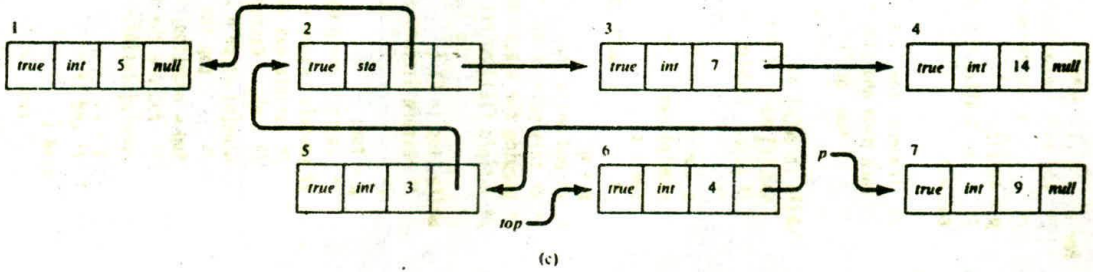
Now that we have described the temporary distortions that are made in the list structure by the Schorr–Waite algorithm, we present the algorithm itself. We invite the reader to trace through the effects of the algorithm on the lists of Figures 9.2.4a and 9.1.9.



(a)



(b)



(c)

Figure 9.2.4

```
    for (i = 0; i < NUMACC; i++) {
      /*  for each immediately accessible node,  */
      /*          trace through its list         */
      p = acc[i];
      /*     initialize the stack to empty       */
      top = 0;
again:
      /*  Traverse the list through its next fields, marking  */
      /* each node and placing it on the stack until a marked */
      /*       node or the end of the list is reached.        */
      /*                 Assume node[0].mark = true           */
      while (node[p].mark != TRUE) {
        node[p].mark = TRUE;
        /* place  node[p] on the stack, saving a pointer */
        /*                 to the next node              */
        q = node[p].next;
        node[p].next = top;
        top = p;
        /*          advance to the next node             */
        p = q;
      } /* end while  */;
      /* at this point trace the way back through the list,  */
      /* popping the stack  until a node  is reached  whose  */
      /* lstinfo field points to an unmarked  node, or until */
      /*                 the list is empty                   */
      while (top != 0) {
        /* restore lstinfo or next to p and pop the */
        /*                 stack                    */
        p = top;
        /*  restore the proper field of node[p]     */
        if (node[p].utype == STK) {
          /* lstinfo was used as the stack  */
          /* link.   Restore the tag field  */
          node[p].utype = LST;
          /*       pop the stack            */
          top = node[top].lstinfo;
          /*  restore the lstinfo field     */
          node[p].lstinfo = q;
          q = p;
        } /* end if */
        else {
          /* next was used as  the stack   */
          /*       link. Pop the stack.    */
          top = node[top].next;
          /*    restore the next field     */
          node[p].next = q;
          q = p;
          /* check if we must travel  down */
          /*      node[p].lstinfo          */
          if (node[p].utype == LST) {
```

```
            /* indicate that lstinfo is    */
            /* used as the stack link      */
            node[p].utype = STK;
            /* push node[p] on the stack   */
            node[p].lstinfo = top;
            top = p;
            /*   advance to next node      */
            p = q;
            goto again;
          } /* end if */
        } /* end if */
      } /* end while */
    } /* end for */
```

Although this algorithm is advantageous in terms of space since no auxiliary stack is necessary, it is disadvantageous in terms of time because each list must be traversed twice: once in pushing each node in the list on the stack and once in popping the stack. This can be contrasted with the relatively few nodes that must be stacked when an auxiliary stack is available.

Of course, several methods of garbage collection can be combined into a single algorithm. For example, an auxiliary stack of fixed size can be set aside for garbage collection and when the stack is about to overflow, the algorithm can switch to the Schorr–Waite method. We leave the details as an exercise.

Collection and Compaction

Once the memory locations of a given system have been marked appropriately, the collection phase may begin. The purpose of this phase is to return to available memory all those locations that were previously garbage (not used by any program but unavailable to any user). It is easy to pass through memory sequentially, examine each node in turn, and return unmarked nodes to available storage.

For example, given the type definitions and declarations presented above, the following algorithm could be used to return the unmarked nodes to an available list headed by *avail*:

```
for (p = 0; p < NUMNODES; p++) {
  if (node[p].mark != TRUE) {
    node[p].next = avail;
    avail = p;
  } /* end if */
  node[p].mark = FALSE;
} /* end for */
```

After this algorithm has completed, all unused nodes are on the available list, and all nodes that are in use by programs have their *mark* fields turned off (for the next call to garbage collection). Note that this algorithm places nodes on the available list in opposite order of their memory location. If it were desired to return nodes to

available memory in the order of increasing memory location, the ***for*** loop above could be reversed to read

```
for (p = NUMNODES - 1; p >= 1; p--)
```

Although at this point (following the marking and collection phases of the system) all nodes that are not in use are on the available list, the memory of the system may not be in an optimal state for future use. This is because the interleaving of the occupied nodes with available nodes may make much of the memory on the available list unusable. For example, memory is often required in blocks (groups of contiguous nodes) rather than as single discrete nodes one at a time. The memory request by a compiler for space in which to store an array would require the allocation of such a block. If, for example, all the odd locations in memory were occupied and all the even locations were on the available list, a request for even an array of size 2 could not be honored, despite the fact that half of memory is on the available list. Although this example is probably not very realistic, there certainly exist situations in which a request for a contiguous block of memory could not be honored, despite the fact that sufficient memory does indeed exist.

There are several approaches to this problem. Some methods allocate and free portions of memory in blocks (groups of contiguous nodes) rather than in units of individual nodes. This guarantees that when a block of storage is freed (returned to the available pool), a block will be available for subsequent allocation requests. The size of these blocks and the manner in which they are stored, allocated and freed are discussed in the next section.

However, even if storage is maintained as units of individual nodes rather than as blocks, it is still possible to provide the user with blocks of contiguous storage. The process of moving all used (marked) nodes to one end of memory and all the available memory to the other end is called ***compaction***, and an algorithm that performs such a process is called a ***compaction*** (or ***compacting***) ***algorithm***.

The basic problem in developing an algorithm that moves portions of memory from one location to another is to preserve the integrity of pointer values to the nodes being moved. For example, if *node*(p) in memory contains a pointer q, when *node*(p) and *node*(q) are moved, not only must the addresses of *node*(p) and *node*(q) be modified but the contents of *node*(p) (which contained the pointer q) must be modified to point to the new address of *node*(q). In addition to being able to change addresses of nodes, we must have a method of determining whether the contents of any node contain a pointer to some other node (in which case its value may have to be changed) or whether it contains some other data type (so that no change is necessary).

A number of compaction techniques have been developed. As in the case of marking algorithms, because the process is required at precisely the time that little additional space is available, methods that require substantial additional storage (for example, a stack) are not practical. Let us examine one compaction algorithm that does not need additional memory when it runs.

The compaction algorithm is executed after the marking phase and traverses memory sequentially. Each marked node, as it is encountered in the sequential traversal, is assigned to the next available memory location starting from the beginning of available memory. When examining a marked node *nd*1 that points to a node *nd*2, the

pointer in *nd*1 that now points to *nd*2 must be updated to the new location where *nd*2 will be moved. That location may not yet be known because *nd*2 might be at a later address than *nd*1. *nd*1 is therefore placed on a list emanating from *nd*2 of all nodes that contain pointers to *nd*2, so that when the new location of *nd*2 is determined, *nd*1 can be accessed and the pointer to *nd*2 contained in it modified.

For now let us assume that a new field *header* in each node *nd*2 points to the list of nodes that contain pointers to *nd*2. We call this list the ***adjustment list*** of *nd*2. We can reuse the field that pointed to *nd*2 (either *next* or *lstinfo*) as the link field for the adjustment list of *nd*2; we know that its "real" value is *nd*2 because the node is on the list emanating from *header*(*nd*2). Thus once its adjustment list has been formed, when *nd*2 is reached in a sequential traversal, that adjustment list can be traversed and the values in the fields used to link that list can be changed to the new location assigned to *nd*2. Then, once all nodes that point to *nd*2 have had their pointers adjusted, *nd*2 itself can be moved.

However, there is one additional piece of information that is required. The adjustment list of nodes pointing to *nd*2 can be linked via either the *next* pointer of a node *nd*1 (if *next*(*nd*1) = *nd*2) or the *lstinfo* pointer (if *lstinfo*(*nd*1) = *nd*2). How can we tell which it is? For this purpose, three additional fields in each node are necessary. The values of these fields can be either "N" for *none*, which indicates that a node is not on an adjustment list, "I" for *info*, which indicates that a node is linked onto the adjustment list using *lstinfo*, or "L" for *link*, which indicates that it is linked onto the adjustment list using *next*. The three fields are named *headptr*, *infoptr*, and *nextptr*. *headptr*(*nd*) defines the link field in the node pointed to by *header*(*nd*); *infoptr*(*nd*) defines the link field in the node pointed to by *lstinfo*(*nd*); and *nextptr*(*nd*) defines the link field in the node pointed to by *next*(*nd*).

Thus, we assume the following format for the nodes:

```
struct nodetype {
   int mark;
   int header;
   int next;
   char headptr;
   char infoptr;
   char nextptr;
   int utype;
   union {
      int intgrinfo;
      char charinfo;
      int lstinfo;
   } info;
};
```

Now consider a single sequential pass of the algorithm. If a node *nd*1 points to a node *nd*2 that appears later in memory, by the time the algorithm reaches *nd*2 sequentially, *nd*1 will have already been placed on the adjustment list of *nd*2. When the algorithm reaches *nd*2, therefore, the pointers in *nd*1 can be modified. But if *nd*2 appears earlier in memory, when *nd*2 is reached, it is not yet known that *nd*1 points to it; therefore the pointer in *nd*1 cannot be adjusted. For this reason the algorithm requires

two sequential passes. The first places nodes on adjustment lists and modifies pointers in nodes that it finds on adjustment lists. The second clears away adjustment lists remaining from the first pass and actually moves the nodes to their new locations. The first pass may be outlined as follows:

1. Update the memory location to be assigned to the next marked node, *nd*.
2. Traverse the list of nodes pointed to by *header(nd)* and change the appropriate pointer fields to point to the new location of *nd*.
3. If the *utype* field of *nd* is *lst* and *lstinfo(nd)* is not *null*, place *nd* on the list of the nodes headed by *header(lstinfo(nd))*.
4. If *next(nd)* is not *null*, place *nd* on the list of the nodes headed by *header(next)(nd)*.

Once this process has been completed for each marked node, a second pass through memory will perform the actual compaction. During the second pass we perform the following operations:

1. Update the memory location to be assigned to the next marked node, *nd*.
2. Traverse the list of nodes pointed to by *header(nd)* and change the appropriate pointer fields to point to the new location of *nd*.
3. Move *nd* to its new location.

The following algorithm performs the actual compaction. (We assume an auxiliary variable *source*, which will contain an "N", "I", or "L" as explained before for use in traversing the lists.)

```
/* initialize fields for compaction algorithm */
for (i = 1; i < MAXNODES; i++) {
   node[i].header = 0;
   node[i].headptr = 'N';
   node[i].infoptr = 'N';
   node[i].nextptr = 'N';
} /* end for */
/*                        Pass 1                                  */
/* Scan nodes sequentially. As each node nd is encountered */
/* perform the following operations:                              */
/* 1. Determine the new location of the node                      */
/* 2. For all nodes that were previously encountered on     */
/*    this pass that  point to nd, adjust the appropriate  */
/*    pointer to point to the new location of nd.          */
/* 3. If any of the fields of nd point to some other node, */
/*    p, place nd on the list headed by node[p].header     */

newloc = 0;
for (nd = 1; nd < MAXNODES; nd++)
  if (node[nd].mark == TRUE) {
   /* nodes that are not marked are   */
   /*         to be ignored.          */
```

```
        newloc++; /* operation 1                    */
        /* operation 2 */
        p = node[nd].header;
        source = node[nd].headptr;
        while (p != 0)
           /* traverse the list of nodes          */
           /* encountered thus far that           */
           /*          point to nd                */
           if (source == 'I') {
              q = node[p].lstinfo;
              source = node[p].infoptr;
              node[p].lstinfo = newloc;
              node[p].infoptr = 'N';
              p = q;
           } /* end if*/
           else {
              q = node[p].next;
              source = node[p].nextptr;
              node[p].next = newloc;
              node[p].nextptr = 'N';
              p = q;
           } /* end else */
           node[nd].headptr = 'N';
           node[nd].header = 0;
           /* operation 3 */
           if ((node[nd].utype == LST) &&
                                    (node[nd].lstinfo != 0)) {
              /* place node[nd] on a list          */
              /* linked by node[nd].lstinfo */
              p = node[nd].lstinfo;
              node[nd].lstinfo = node[p].header;
              node[nd].infoptr = node[p].headptr;
              node[p].header = nd;
              node[p].headptr = 'I';
           } /* end if */
        /* place node[nd] on a list linked by    */
        /*          node[nd].next                */

        p = node[nd].next;
        node[nd].next = node[p].header;
        node[nd].nextptr = node[p].headptr;
        if (p != 0) {
           node[p].header = nd;
           node[p].headptr = 'L';
        } /* end if */
     } /* end if node[nd].mark */

     /* Pass 2: This pass examines each node nd in turn,      */
     /* updates all nodes on the adjustment list of nd, and   */
     /* then moves the contents of nd to its new location.    */
```

```
newloc = 0;
for (nd = 1; nd < MAXNODES; nd++)
   if (node[nd].mark) {
      newloc++;
      p = node[nd].header;
      source = node[nd].headptr;
      while (p != 0)
         if (source == 'I') {
            q = node[p].lstinfo;
            source = node[p].infoptr;
            node[p].lstinfo = newloc;
            node[p].infoptr = 'N';
            p = q;
         }
         else {
            q = node[p].next;
            source = node[p].nextptr;
            node[p].next = newloc;
            node[p].nextptr = 'N';
            p = q;
         } /* end if */
      node[nd].headptr = 'N';
      node[nd].header = 0;
      node[nd].mark = false;
      node[newloc] = node[nd];
   } /* end if node[nd].mark */
```

Several points should be noted with respect to this algorithm. First, *node*[0] is suitably initialized so that the algorithm need not test for special cases. Second, the process of adjusting the pointers of all nodes on the list headed by the header field of a particular node is performed twice: once during the first pass and once during the second. This process could not be deferred entirely to the second pass, when all the pointers to a particular node are known. The reason for this is that when a field in a node *nd*2 in the adjustment list of node *nd* is changed to *nd*, it must be changed before *nd*2 is moved to a new location, since no record of the new location is maintained in *nd*2. Thus nodes on the adjustment list of *nd* that precede *nd* sequentially must have their fields modified before they have been moved. But since they are moved before we reach *nd* in the second pass, and they have already been placed on the adjustment list by the time we reach *nd* in the first pass, we must clear the adjustment lists and modify the pointer fields at that point. We also must modify pointer fields in the second pass for nodes on the adjustment list of *nd* that are sequentially after *nd* and were put on the adjustment list of *nd* during the first pass after having already passed *nd*.

The algorithm seems to require several additional fields for each node. In reality, these additional fields are not required. Most systems have at least one field in each node that cannot take on a pointer value during the ordinary course of processing. This field can be used to hold the *header* pointer to the adjustment list, so that an additional *header* field is not necessary. The value that was held in this field can be moved to the last node in the adjustment list, and placed in either the *next* or *lstinfo* field, depending

on which of the two held the pointer to the target node. We assume that it is possible to distinguish between a pointer and a nonpointer value so that we can detect when we reach the end of the adjustment list by the presence of a nonpointer value in the last node.

In addition, we used the fields *headptr*, *nextptr*, and *infoptr* to indicate which field (*next* or *lstinfo*) in the node pointed to by *header*, *next*, or *lstinfo*, respectively, held the target pointer. But since in C, *header*, *next*, and *lstinfo* may contain actual addresses, rather than just a pointer to a node, they could hold the address of the specific field within the node that held the target pointer rather than just the address of the node. Thus an additional field to identify the particular field within the node is unnecessary and we can eliminate *headptr*, *nextptr*, and *infoptr* from the nodes.

We therefore see that our compaction algorithm can be modified so that it does not require any additional storage in the nodes. Such an algorithm is called a ***bounded workspace algorithm***.

The time requirements of the algorithm are easy to analyze. There are two linear passes throughout the complete array of memory. Each pass through memory scans each node once and adjusts any pointer fields to which the nodes point. The time requirements are obviously $O(n)$.

With respect to the actual compaction itself, it is not necessary to invoke the compaction routine each time the garbage collection routine is called. The garbage collection routine is called when there is little (or no) space available. The amount of space reclaimed by the algorithm may or may not provide sufficient contiguous blocks. It is the compaction algorithm that assures that the space that is reclaimed is contiguous at one end of memory. The memory returned by the garbage collection algorithm may not be sufficiently fragmented to warrant a call to the compaction routine, so that only after several calls to the collection routine is it necessary to call the compaction algorithm.

On the other hand, if the compaction routine is not called sufficiently often, the system may indicate that insufficient space is available when in fact there is sufficient space but that space is not contiguous. Failure to invoke the compaction routine may then result in additional calls to the garbage collection routine. The decision of when to invoke the compaction algorithm in conjunction with the garbage collection algorithm is a difficult one. Yet, because compaction is usually more efficient than garbage collection, it is usually not too inefficient to invoke them at the same time.

Variations of Garbage Collection

There are a number of recently discovered variations of the garbage collection systems just presented. In the traditional schemes we have considered, the applications programs function as long as space availability of the system satisfies certain criteria. (These criteria may relate to the total amount of free space available, the number and size of contiguous memory locations available, the amount of memory requested since the last garbage collection phase, and so forth.) When these criteria are no longer met, all applications programs halt and the system directs its resources to garbage collection. Once the collection has completed, the applications programs may resume execution from the point at which they were interrupted.

In some situations, however, this is not satisfactory. Applications that are executing in real time (for example, computing the trajectory of a spaceship, or monitoring a

chemical reaction) cannot be halted while the system is performing garbage collection. In these circumstances it is usually necessary to dedicate a separate processor devoted exclusively to the job of garbage collection. When the system signals that garbage collection must be performed, the separate processor begins executing concurrently with the applications program. Because of this simultaneous execution, it is necessary to guarantee that nodes that are in the process of being acquired for use by an application program are not mistakenly returned to the available pool by the collector. Avoiding such problems is not a trivial process. Systems that allow the collection process to proceed simultaneously with the applications program use "on-the-fly" garbage collection.

Another subject of interest deals with minimizing the cost of reclaiming unused space. In the methods we have discussed, the cost of reclaiming any portion of storage is the same as the cost of reclaiming any other portion (of the same size). Recent attention has been directed toward designing a system in which the cost of reclaiming a portion of storage is proportional to its lifetime. It has been shown empirically that some portions of memory are required for smaller time intervals than are others and that requests for portions of memory with smaller lifetimes occur more frequently than do requests for portions of memory with longer lifetimes. Thus, by reducing the cost of retrieving portions of memory required for short time periods at the expense of the cost of retrieving portions of memory with longer lifespans, the overall cost of the garbage collection process will be reduced. Exactly how one classifies the lifetimes of portions of memory and algorithms for retrieving such portions of memory will not be considered further. The interested reader is referred to the references.

The process of garbage collection is also applied to reclaiming unused space in secondary devices (for example, a disk). Although the concept of allocation and freeing space is the same (that is, space may be requested or released by a program), algorithms that manage space on such devices often cannot be translated efficiently from their counterparts that manipulate main memory. The reason for this is that the cost of accessing any location in main memory is the same as that of accessing any other location in main memory. In secondary storage, on the other hand, the cost depends on the location of storage that is currently being accessed as well as the location we desire to access. It is very efficient to access a portion of secondary storage that is in the same block that is now being accessed; to access a location in a different block may involve expensive disk seeks. For this reason, device management systems for offline storage try to minimize the number of such accesses. The interested reader is referred to the literature for a discussion of the relevant techniques.

EXERCISES

9.2.1. Implement each of the following list operations of Section 9.1 in C assuming that the reference count method of list management is used.

(a) *head*
(b) *tail*
(c) *addon*
(d) *sethead*
(e) *settail*

9.2.2. Rewrite the routines of the previous exercise under the system in which the reference counter in a node *nd*1 is decremented when a node *nd*2 pointing to *nd*1 is reallocated, rather than when *nd*2 is freed.

9.2.3. Implement the list operations of Exercise 9.2.1 in C assuming the use of list headers, with reference counts in header nodes only. Ensure that illegal lists are never formed.

9.2.4. Write an algorithm to detect recursion in a list, that is, whether or not there is a path from some node on the list back to itself.

9.2.5. Write an algorithm to restore all nodes on a list *list* to the available list. Do the same using no additional storage.

9.2.6. In a multiuser environment where several users are running concurrently, it may be possible for one user to request additional storage and thus invoke the garbage collector while another user is in the middle of list manipulation. If garbage collection is allowed to proceed at that point (before the lists of the second user have been restored to legal form), the second user will find that many list nodes have been freed. Assume that there exist two system routines *nogarbage* and *okgarbage*. A call to the first inhibits the invocation of garbage collection until after the same user calls the second. Implement the list operations of Exercise 9.2.1, using calls to these two routines to ensure that garbage collection is not invoked at inopportune moments.

9.2.7. Trace the actions of the three garbage collection algorithms of the text on the lists of Figures 9.2.4a and 9.1.9, assuming that the integer above each list node is the index of that node in the array *node*. Trace through the algorithms on the list of Figure 9.1.9, after executing the statement

```
list9 = NULL;
```

9.2.8. Given pointers p and q to two list nodes, write an algorithm to determine whether *node*(q) is accessible from *node*(p).

9.2.9. Assume that each node contains an arbitrary number of pointers to other nodes rather than just two so that the lists now become graphs. Revise each of the marking algorithms presented in this section under this possibility.

9.2.10. Revise each of the marking algorithms presented in this section under the assumption that the lists are doubly linked, so that each list node contains a *prevptr* field to the previous node on the same list. How do each of the algorithms increase in efficiency? What restriction on the list structure does the presence of such a field imply?

9.2.11. Write two marking algorithms that use a finite, auxiliary stack of size *stksize*. The algorithms operate like the second marking algorithm presented in the text until the stack becomes full. At that point, the first of the two algorithms operates like the sequential algorithm presented in the text and the second operates like the Schorr–Waite algorithm.

9.2.12. Can you rewrite the Schorr–Waite algorithm to eliminate the ***goto*** statement?

9.3 DYNAMIC MEMORY MANAGEMENT

In the previous sections we assumed that storage is allocated and freed one node at a time. There are two characteristics of nodes that make the previous methods suitable. The first is that each node of a given type is of fixed size and the second is that the size of each node is fairly small. In some applications, however, these characteristics do not

apply. For example, a particular program might require a large amount of contiguous storage (for example, a large array). It would be impractical to attempt to obtain such a block one node at a time. Similarly, a program may require storage blocks in a large variety of sizes. In such cases a memory management system must be able to process requests for variable-length blocks. In this section we discuss some systems of this type.

As an example of this situation, consider a small memory of 1024 words. Suppose a request is made for three blocks of storage of 348, 110 and 212 words, respectively. Let us further suppose that these blocks are allocated sequentially, as shown in Figure 9.3.1a. Now suppose that the second block of size 110 is freed, resulting in the situation depicted in Figure 9.3.1b. There are now 464 words of free space; yet, because the free space is divided into noncontiguous blocks, a request for a block of 400 words could not be satisfied.

Suppose that block 3 were now freed. Clearly, it is not desirable to retain three free blocks of 110, 212, and 354 words. Rather the blocks should be combined into a single large block of 676 words so that further large requests can be satisfied. After combination, memory will appear as in Figure 9.3.1c.

This example illustrates the necessity to keep track of available space, to allocate portions of that space when allocation requests are presented, and to combine contiguous free spaces when a block is freed.

Compaction of Blocks of Storage

One scheme sometimes used involves compaction of storage as follows: Initially memory is one large block of available storage. As requests for storage arrive, blocks of memory are allocated sequentially starting from the first location in memory. This is illustrated in Figure 9.3.2a. A variable *freepoint* contains the address of the first location following the last block allocated. In Figure 9.3.2a, *freepoint* equals 950. Note that all memory locations between *freepoint* and the highest address in memory are free. When a block is freed, *freepoint* remains unchanged and no combinations of free spaces take place. When a block of size n is allocated, *freepoint* is increased by n. This continues until a block of size n is requested and *freepoint* $+ n - 1$ is larger than the highest address in memory. The request cannot be satisfied without further action being taken.

At that point user routines come to a halt and a system compaction routine is called. Although the algorithm of the previous section was designed to address uniform nodes, it could be modified to compact memory consisting of blocks of storage as well. Such a routine copies all allocated blocks into sequential memory locations starting from the lowest address in memory. Thus all free blocks that were interspersed with allocated blocks are eliminated, and *freepoint* is reset to the sum of the sizes of all the allocated blocks. One large free block is created at the upper end of memory and the user request may be filled if there is sufficient storage available. This process is illustrated in Figure 9.3.2 on a memory of 1024 words.

When allocated blocks are copied into lower portions of memory, special care must be taken so that pointer values remain correct. For example, the contents of memory location 420 in allocated block 2 of Figure 9.3.2a might contain the address 340. After block 2 is moved to locations 125 through 299, location 140 contains the previous contents of location 340. In moving the contents of 420 to 220, those contents must be

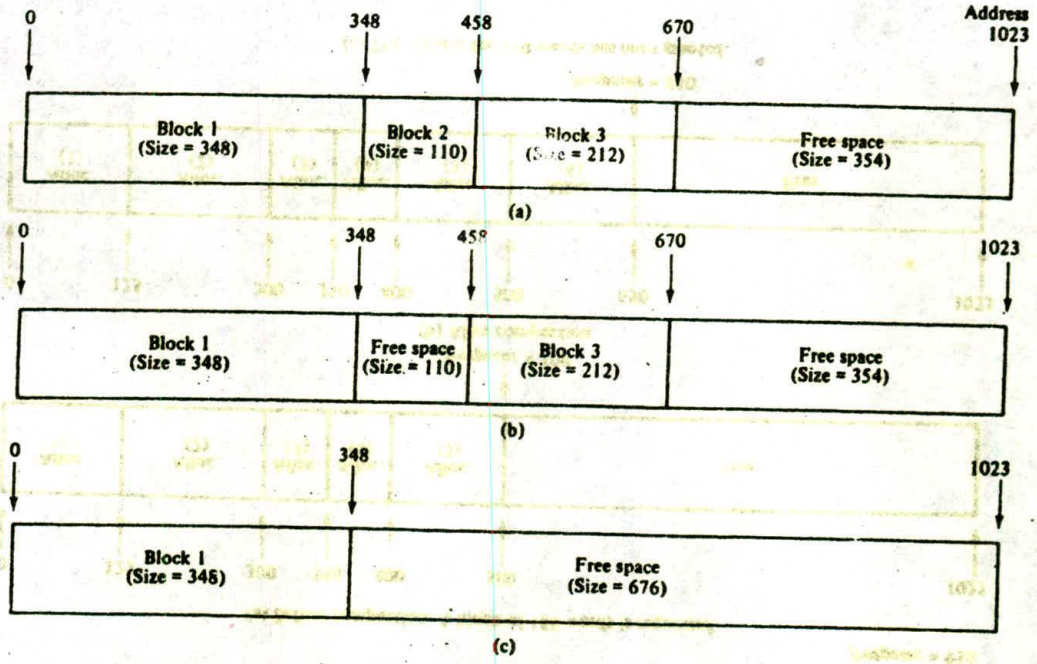
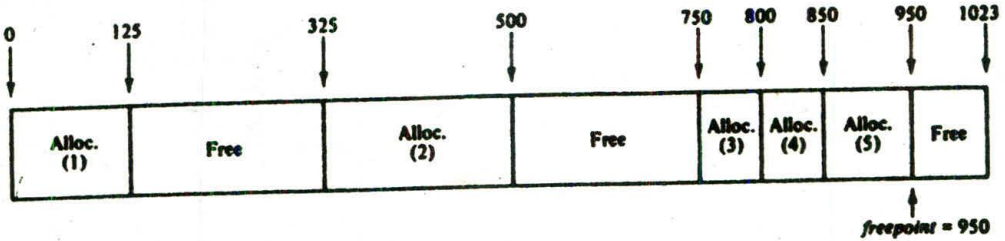
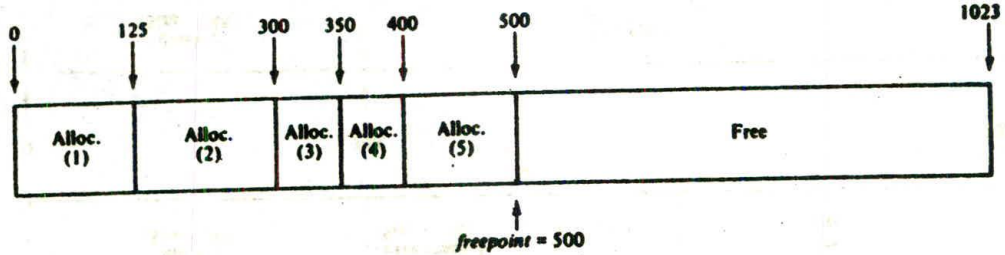

0
348
458
670
Address
1023
Block 1
(Size = 348)
Block 2
(Size = 110)
Block 3
(Size = 212)
Free space
(Size = 354)
(a)
0
348
458
670
1023
Block 1
(Size = 348)
Free space
(Size = 110)
Block 3
(Size = 212)
Free space
(Size = 354)
(b)
0
348
1023
Block 1
(Size = 348)
Free space
(Size = 676)
(c)

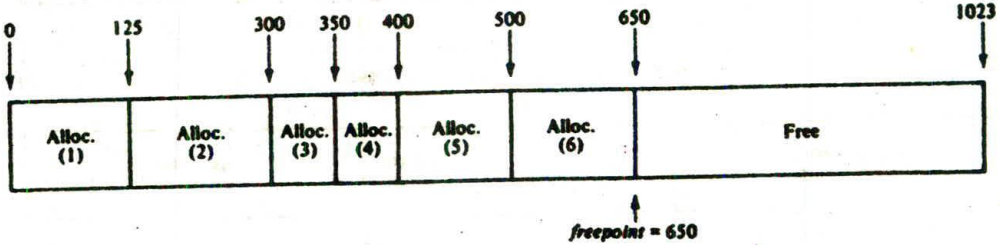
Figure 9.3.1



(a) Before compaction. A block of 150 words is requested.



(b) After compaction.



(c) The request for 150 words has been granted.

Figure 9.3.2

changed to 140. Thus in order for compaction to be successful, there must be a method to determine if the contents of a given location is an address.

An alternative is a system that computes addresses as offsets from some base address. In that case only the contents of the base address must be changed, and the offset in memory need not be altered. For example, in the previous instance, location 420 would contain the offset 15 before compaction, rather than the address 340. Since the base address of the block is 325, the address 340 would be computed as the base address 325 plus the offset 15. When the block is moved, its base address is changed to 125, while the offset 15 is moved from location 420 to location 220. Adding the new base address 125 to the offset 15 yields 140, which is the address to which the contents of 340 have been moved. Note that the offset 15 contained in memory has not been changed at all. However, such a technique is useful only for intrablock memory references; interblock references to locations in a different block must still be modified. A compaction routine requires a method by which the size of a block and its status (allocated or free) could be determined.

Compaction is similar to garbage collection in that all user processing must stop as the system takes time to clean up its storage. For this reason, and because of the pointer problem discussed in the foregoing, compaction is not used as frequently as the more complicated schemes that follow.

First Fit, Best Fit, and Worst Fit

If it is not desirable to move blocks of allocated storage from one area of memory to another, it must be possible to reallocate memory blocks that have been freed dynamically as user processing continues. For example, if memory is fragmented as shown in Figure 9.3.1b and a request is made for a block of 250 words of storage, locations 670 through 919 would be used. The result is shown in Figure 9.3.3a. If memory is as shown in Figure 9.3.1b, a request for a block of 50 words could be satisfied by either words 348 through 397 or words 670 through 719 (see Figure 9.3.3b and c). In each case part of a free block becomes allocated, leaving the remaining portion free.

Each time that a request is made for storage, a free area large enough to accommodate the size requested must be located. The most obvious method for keeping track of the free blocks is to use a linear linked list. Each free block contains a field containing the size of the block and a field containing a pointer to the next free block. These fields are in some uniform location (say, the first two words) in the block. If p is the address of a free block, the expressions *size*(p) and *next*(p) are used to refer to these two quantities. A global pointer *freeblock* points to the first free block on this list. Let us see how blocks are removed from the free list when storage is requested. We then examine how blocks are added onto this list when they are freed.

Consider the situation of Figure 9.3.1b, reproduced in Figure 9.3.4a to show the free list. There are several methods of selecting the free block to use when requesting storage. In the ***first-fit*** method, the free list is traversed sequentially to find the first free block whose size is larger than or equal to the amount requested. Once the block is found, it is removed from the list (if it is equal in size to the amount requested) or is split into two portions (if it is greater than the amount requested). The first of these portions remains on the list and the second is allocated. The reason for allocating the

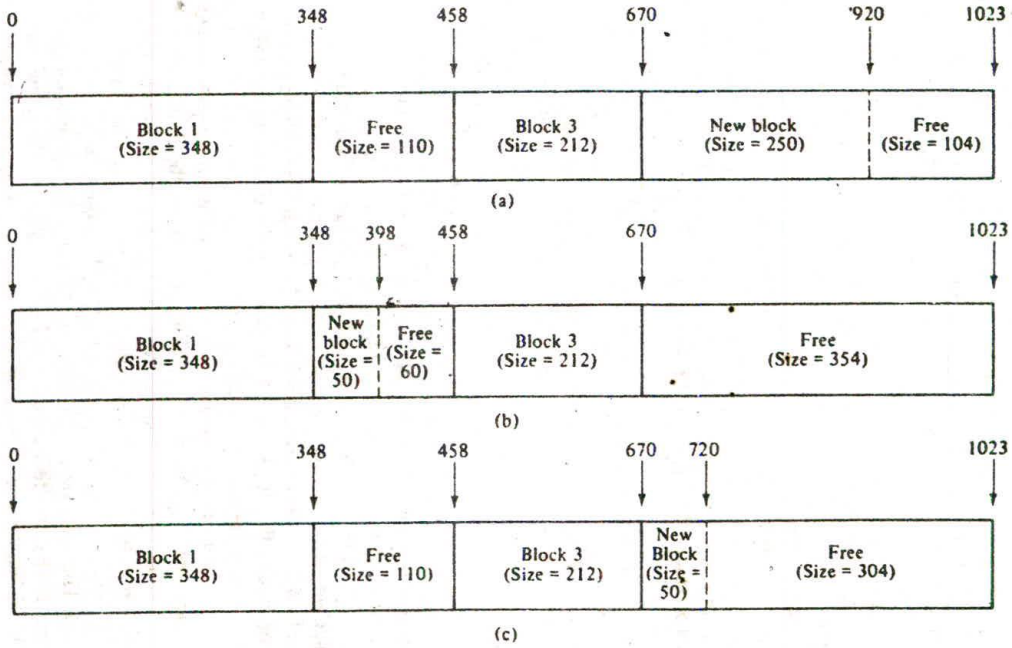

0
348
458
670
920
1023
Block 1 (Size = 348)
Free (Size = 110)
Block 3 (Size = 212)
New block (Size = 250)
Free (Size = 104)
(a)
0
348
398
458
670
1023
Block 1 (Size = 348)
New block (Size = 50)
Free (Size = 60)
Block 3 (Size = 212)
Free (Size = 354)
(b)
0
348
458
670
720
1023
Block 1 (Size = 348)
Free (Size = 110)
Block 3 (Size = 212)
New Block (Size = 50)
Free (Size = 304)
(c)

Figure 9.3.3

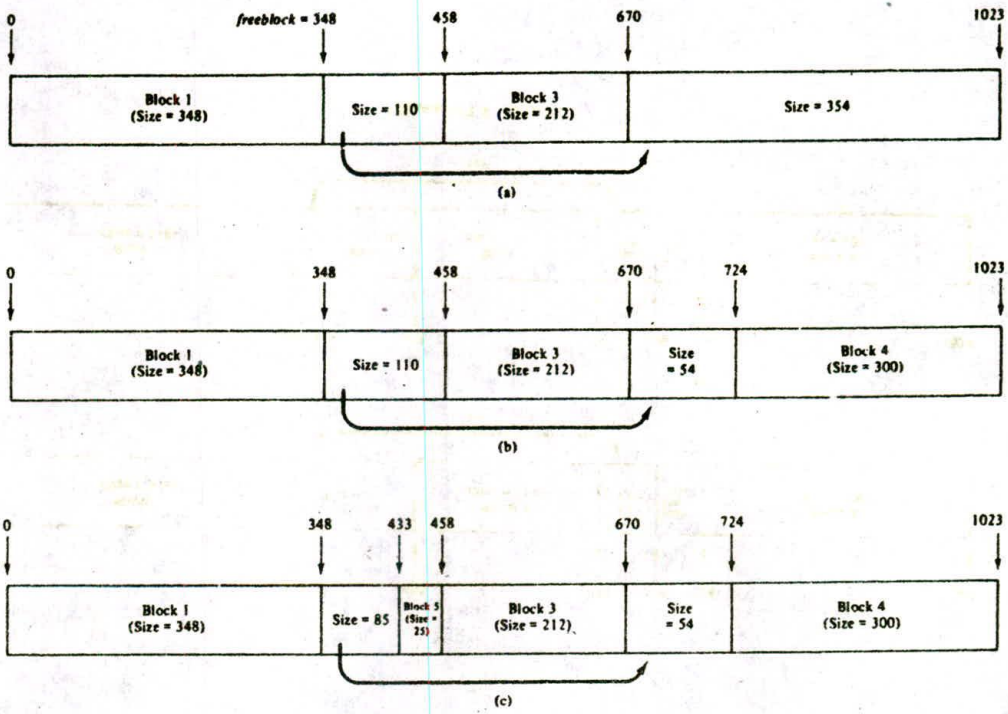

0
freeblock = 348
458
670
1023
Block 1
(Size = 348)
Size = 110
Block 3
(Size = 212)
Size = 354
(a)
0
348
458
670
724
1023
Block 1
(Size = 348)
Size = 110
Block 3
(Size = 212)
Size
= 54
Block 4
(Size = 300)
(b)
0
348
433
458
670
724
1023
Block 1
(Size = 348)
Size = 85
Block 5
(Size =
25)
Block 3
(Size = 212)
Size
= 54
Block 4
(Size = 300)
(c)

Figure 9.3.4

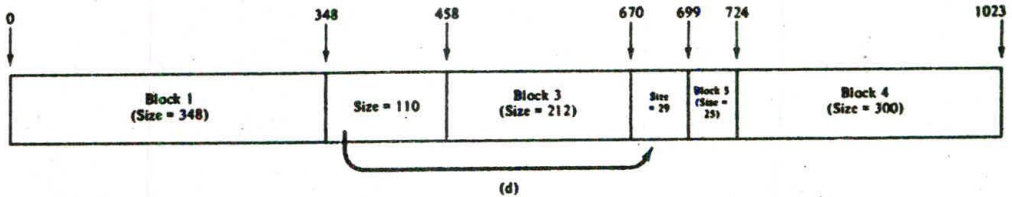


Figure 9.3.4 *(cont.)*

second portion rather than the first is that the free list *next* pointer is at the beginning of each free block. By leaving the first portion of the block on the free list, this pointer need not be copied into some other location, and the *next* field of the previous block in the list need not be changed.

The following first-fit allocation algorithm returns the address of a free block of storage of size n in the variable *alloc* if one is available and sets *alloc* to the null address if no such block is available.

```
p = freeblock;
alloc = null;
q = null;
while (p != null && size(p) < n) {
   q = p;
   p = next(p);
} /* end while */
if (p != null) {  /* there is a block large enough  */
   s = size(p);
   alloc = p + s - n; /* alloc contains the address */
                      /*    of the desired block    */
   if (s == n)
      /* remove the block from the free list */
      if (q == null)
         freeblock = next(p);
      else
         next(q) = next(p);
   else /* adjust the size of the remaining */
        /*          free block             */
      size(p) = s - n;
} /* end if */
```

The *best-fit* method obtains the smallest free block whose size is greater than or equal to n. An algorithm to obtain such a block by traversing the entire free list follows. We assume that *memsize* is the total number of words in memory.

```
p = freeblock;       /* p is used to traverse the free list   */
q = null;            /*       q is one block behind p         */
r = null;            /*    r points to the desired block      */
rq = null;           /*      rq is one block behind r         */
rsize = memsize + 1; /* rsize is the size of the block at r */
alloc = null;        /* alloc will point to the block selected */
while (p != null) {
   if (size(p) >= n && size(p) < rsize) {
      /* we have found a free block closer in size */
      r = p;
      rq = q;
      rsize = size(p);
   } /* end if */
   /* continue traversing the free list */
   q = p;
   p = next(p);
} /* end while */
```

```
if (r != null) {
   /* there is a block of sufficient size */
   alloc = r + rsize - n;
   if (rsize == n)
      /*remove the block from the free list*/
      if (rq == null)
         freeblock = next(r);
      else
         next(rq) = next(r);
   else
      size(r) = rsize - n;
} /* end if */
```

To see the difference between the first-fit and best-fit methods, consider the following examples. We begin with memory fragmented as in Figure 9.3.4a. There are two blocks of free storage, of sizes 110 and 354. If a request is made for a block of 300 words, the block of 354 is split as shown in Figure 9.3.4b under both the first-fit and best-fit methods. Suppose a block of size 25 is then requested. Under first-fit, the block of size, 110 is split (Figure 9.3.4c), whereas under best-fit the block of size 54 is split (Figure 9.3.4d). If a block of size 100 is then requested, the request can be fulfilled under best-fit, since the block of size 110 is available (Figure 9.3.4e), but it cannot be fulfilled under first-fit. This illustrates an advantage of the best-fit method in that very large free blocks remain unsplit so that requests for large blocks can be satisfied. In the first-fit method, a very large block of free storage at the beginning of the free list is nibbled away by small requests so that it is severely shrunken by the time a large request arrives.

However, it is also possible for the first-fit method to succeed where the best-fit method fails. As an example, consider the case in which the system begins with free blocks of size 110 and 54 and then makes successive requests for 25, 70, and 50 words. Figure 9.3.5 illustrates that the first-fit method succeeds in fulfilling these requests, whereas the best-fit method does not. The reason is that remaining unallocated portions of blocks are smaller under best-fit than under first-fit.

Yet another method of allocating blocks of storage is the ***worst-fit method***. In this method the system always allocates a portion of the largest free block in memory. The philosophy behind this method is that by using a small number of very large blocks repeatedly to satisfy the majority of requests, many moderately sized blocks will be left unfragmented. Thus, this method is likely to satisfy a larger number of requests than the

Request	Blocks remaining using First-fit	Best-fit
Initially	110, 54	110, 54
25	85, 54	110, 29
70	15, 54	40, 29
50	15, 4	cannot be fulfilled

Figure 9.3.5

other methods, unless most of the requests are for very large portions of memory. For example, if memory consists initially of blocks of sizes 200, 300, and 100, the sequence of requests 150, 100, 125, 100, 100 can be satisfied by the worst-fit method but not by either the first-fit or best-fit methods. (Convince yourself that this is the case.)

The major reason for choosing one method over the other is efficiency. In each of the methods, the search can be made more efficient. For example, a true first-fit method, which allocates the block at the lowest memory address first, will be most efficient if the available list is maintained in the order of increasing memory address (as it should be for reasons to be discussed shortly). On the other hand, if the available list is maintained in the order of increasing size, a best-fit search for a block becomes more efficient. And finally, if the list is maintained in the order of decreasing size, a worst-fit request requires no searching, as the largest size block is always the first on the list. However, for reasons we shall discuss shortly, it is not practical to maintain the list of available blocks ordered by size.

Each of the methods has certain characteristics that make it either desirable or undesirable for various request patterns. In the absence of any specific consideration to the contrary, the first-fit method is usually preferred.

Improvements in the First-Fit Method

There are several improvements that can be made in the first-fit method. If the size of a free block is only slightly larger than the size of the block to be allocated, the portion of the free block that remains free is very small. Very often this remaining portion is so small that there is little likelihood of its being used before the allocated portion is freed and the two portions are recombined. Thus there is little benefit achieved by leaving that small portion on the free list. Also recall that any free block must be of some minimum size (in our case, two words) so that it may contain *size* and *next* fields. What if the smaller portion of a free block is below this minimum size after the larger portion has been allocated?

The solution to these problems is to insist that no block may remain free if its size is below some reasonable minimum. If a free block is about to be split and the remaining portion is below this minimum size, the block is not split. Instead, the entire free block is allocated as though it were exactly the right size. This allows the system to remove the entire block from the free list and does not clutter up the list with very small blocks.

The phenomenon in which there are many small noncontiguous free blocks is called ***external fragmentation*** because free space is wasted outside allocated blocks. This contrasts with ***internal fragmentation***, in which free space is wasted within allocated blocks. The foregoing solution transforms external fragmentation into internal fragmentation. The choice of what minimum size to use depends on the pattern of allocation requests in the particular system. It is reasonable to use a minimum size such that only a small percentage (say 5 percent) of the allocation requests are less than or equal to that size. Note that the possibility of small slivers remaining is even greater under the best-fit method than under first-fit, so that the establishment of such a minimum size is of correspondingly greater importance under that method.

Another significant improvement in the first-fit method can be made. As time goes on, smaller free blocks will tend to accumulate near the front of the free list. This is

because a large block near the front of the list is reduced in size before a large block near the back of the list. Thus, in searching for a large or even a moderate-size block, the small blocks near the front cannot be used. The algorithm would be more efficient if the free list were organized as a circular list whose first element varies dynamically as blocks are allocated.

Two ways of implementing this dynamic variance suggest themselves. In the first, *freeblock* (which is the pointer to the first free block on the list) is set to *next(freeblock)*, so that the front of the list advances one block each time that a block is allocated. In the second, *freeblock* is set to *next(alloc)*, where *alloc* points to the block just chosen for allocation. Thus all blocks that were too small for this allocation request are in effect moved to the back of the list. The reader is invited to investigate the advantages and disadvantages of both techniques.

Freeing Storage Blocks

Thus far nothing has been said about how allocated blocks of storage are freed and how they are combined with contiguous free blocks to form larger blocks of free storage. Specifically, three questions arise:

1. When a block of storage is freed, where is it placed on the free list? The answer to this question determines how the free list is ordered.
2. When a block of storage is freed, how can it be determined whether the blocks of storage on either side of it are free (in which case the newly freed block should be combined with an already existing free block)?
3. What is the mechanism for combining a newly freed block with a previously free contiguous block?

The term ***liberation*** is used for the process of freeing an allocated block of storage; an algorithm to implement this process is called a ***liberation algorithm***. The free list should be organized to facilitate efficient allocation and liberation.

Suppose that the free list is organized arbitrarily, so that when a block is freed, it is placed at the front of the list. It may be that the block just freed is adjacent to a previously free block. To create a single large free block, the newly freed block should be combined with the adjacent free block. There is no way, short of traversing the entire free list, to determine if such an adjacent free block exists. Thus each liberation would involve a traversal of the free list. For this reason it is inefficient to maintain the free list this way.

An alternative is to keep the free list sorted in order of increasing memory location. Then, when a block is freed, the free list is traversed in a search for the first free block *fb* whose starting address is greater than the starting address of the block being freed. If a contiguous free block is not found in this search, no such contiguous block exists and the newly freed block can be inserted into the free list immediately before *fb*. If *fb* or the free block immediately preceding *fb* on the free list is contiguous to the newly freed block, it can be combined with that newly freed block. Under this method, the entire free list need not be traversed. Instead, only half of the list must be traversed on the average.

The following liberation algorithm implements this scheme, assuming that the free list is linear (not circular) and that *freeblock* points to the free block with the smallest address. The algorithm frees a block of size *n* beginning at address *alloc*.

```
q = null;
p = freeblock;
/* p traverses the free list. q remains one step behind p */
while (p != null && p < alloc) {
   q = p;
   p = next(p);
} /* end while */
/* At this point, either q = null or q < alloc and either  */
/*  p = null or alloc < p. Thus if p and q are not null,   */
/* the block must be combined with the blocks beginning at */
/* p or q or both, or must be inserted in the list between */
/* the two blocks.                                         */
if (q == null)
   freeblock = alloc;
else
   if (q + size(q) == alloc) {
      /* combine with previous block */
      alloc = q;
      n = size(q) + n;
   }
   else
      next(q) = alloc;
if (p != null && alloc + n == p) {
   /* combine with subsequent block */
   size(alloc) = n + size(p);
   next(alloc) = next(p);
}
else {
   size(alloc) = n;
   next(alloc) = p;
} /* end if */
```

If the free list is organized as a circular list, the first-fit allocation algorithm begins traversing the list from varying locations. However, to traverse the list from the lowest location during liberation, an additional external pointer, *lowblock*, to the free block with the lowest location is required. Ordinarily, traversal starts at *lowblock* during liberation. However, if it is found that *freeblock* < *alloc* when the block that starts at *alloc* is about to be freed, traversal starts at *freeblock*, so that even less search time is used during liberation. The reader is urged to implement this variation as an exercise.

Boundary Tag Method

It is desirable to eliminate all searching during liberation to make the process more efficient. One method of doing this comes at the expense of keeping extra information in all blocks (both free and allocated).

A search is necessary during liberation to determine if the newly freed block may be combined with some existing free block. There is no way of detecting whether such a block exists or which block it is without a search. However, if such a block exists, it must immediately precede or succeed the block being freed. The first address of the block that follows a block of size n at *alloc* is *alloc* + n. Suppose that every block contains a field *flag* that is *true* if the block is allocated and *false* if the block is free. Then by examining *flag*(*alloc* + n), it can be determined whether or not the block immediately following the block at *alloc* is free.

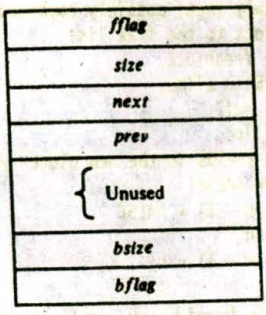
It is more difficult to determine the status of the block immediately preceding the block at *alloc*. The address of the last location of that preceding block is, of course, *alloc* − 1. But there is no way of finding the address of its first location without knowing its size. Suppose, however, that each block contains two flags: *fflag* and *bflag*, both of which are *true* if the block is allocated and *false* otherwise. *fflag* is at a specific offset from the front of the block, and *bflag* is at a specific negative offset from the back of the block.

Thus, to access *fflag*, the first location of the block must be known; to access *bflag*, the last location of the block must be known. The status of the block following the block at *alloc* can be determined from the value of *fflag*(*alloc* + n), and the status of the block preceding the block at *alloc* can be determined from the value of *bflag*(*alloc* − 1). Then, when a block is to be freed, it can be determined immediately whether it must be combined with either of its two neighboring blocks.

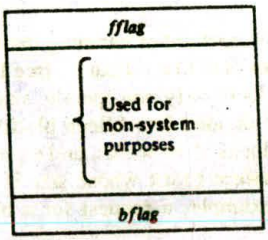
A list of free blocks is still needed for the allocation process. When a block is freed, its neighbors are examined. If both blocks are allocated, the block can simply be appended to the front of the free list. If one (or both) of its neighbors is free, the neighbor(s) can be removed from the free list, combined with the newly freed block, and the newly created large block can be placed at the head of the free list. Note that this would tend to reduce search times under first-fit allocation as well, since a previously allocated block (especially if it has been combined with other blocks) is likely to be large enough to satisfy the next allocation request. Since it is placed at the head of the free list, the search time is reduced sharply.

To remove an arbitrary block from the free list (to combine it with a newly freed block) without traversing the entire list, the free list must be doubly linked. Thus each free block must contain two pointers, *next* and *prev*, to the next and previous free blocks on the free list. It is also necessary to be able to access these two pointers from the last location of a free block. (This is needed when combining a newly freed block with a free block that immediately precedes it in memory.) Thus the front of free block must be accessible from its rear. One way to do this is to introduce a *bsize* field at a given negative offset from the last location of each free block. This field contains the same value as the *size* field at the front of the block. Figure 9.3.6 illustrates the structure of free and allocated blocks under this method, which is called the ***boundary tag method***. Each of the control fields, *fflag*, *size*, *next*, *prev*, *bsize*, and *bflag* is shown as occupying a complete word, although in practice they may be packed together, several fields to a word.

We now present the liberation algorithm using the boundary tag method. For clarity, we assume that *fflag* and *bflag* are logical flags and that *true* indicates an allocated block and *false* indicates a free block. [We assume that *bflag*(0) and *fflag*(m), where m



Free block



Allocated block

Figure 9.3.6

is the size of memory, are both *true*.] The algorithm frees a block of size n at location *alloc*. It makes use of an auxiliary routine *remove* that removes a block from the doubly linked list. The details of that routine are left as an exercise for the reader.

```
/*check the preceding block*/
if (bflag(alloc - 1) != TRUE) {
   /* the block must be combined with the preceding block      */
   start = alloc - bsize(alloc - 1); /* find the initial       */
                                     /* address of the block  */
   remove(start); /* remove the block from the free list      */
                  /* increase the size and combine the blocks */
   n = n + size(start);
   alloc = start;
} /* end if */
/* check the following block */
if (fflag(alloc + n) != TRUE) {
   /* the block must be combined with the following block */
   start = alloc + n;
   n = n + size(start);
   remove(start);
} /* end if */
```

```
/* add the newly free, possibly combined */
/*          block to the free list          */
next(alloc) = freeblock;
prev(freeblock) = alloc;
prev(alloc) = null;
freeblock = alloc;
/* adjust the fields in the new block */
fflag(alloc) = FALSE;
bflag(alloc + n - 1) = FALSE
size(alloc) = n;
bsize(alloc + n - 1) = n;
```

Of course, the newly freed block can be inserted into the list based on its size, so that one of the other methods (for example, best-fit or worst-fit) can also be used.

Buddy System

An alternative method of handling the storage management problem without frequent list traversals is to keep separate free lists for blocks of different sizes. Each list contains free blocks of only one specific size. For example, if memory contains 1024 words it might be divided into fifteen blocks: one block of 256 words, two blocks of 128 words, four blocks of 64 words, and eight blocks of 32 words. Whenever storage is requested, the smallest block whose size is greater than or equal to the size needed is reserved. For example, a request for a block of 97 words is filled by a block of size 128.

There are several drawbacks to this scheme. First, space is wasted due to internal fragmentation. (In the example, 31 words of the block are totally unusable). Second, and more serious, a request for a block of size 300 cannot be filled, since the largest size maintained is 256. Also, if two blocks of size 150 are needed, the requests cannot be filled even if sufficient contiguous space is available. Thus, the solution is impractical. The source of the impracticality is that free spaces are never combined. However, a variation of this scheme, called the ***buddy system***, is quite useful.

Several free lists consisting of various sized blocks are maintained. Adjacent free blocks of smaller size may be removed from their lists, combined into free blocks of larger size, and placed on the larger size free list. These larger blocks can then be used intact to satisfy a request for a large amount of memory or they can be split once more into their smaller constituent blocks to satisfy several smaller requests.

The following method works best on binary computers in which the memory size is an integral power of 2 and in which multiplication and division by 2 can be performed very efficiently by shifting. Initially, the entire memory of size 2^m is viewed as a single free block. For each power of two between 1 (which equals 2^0) and 2^m, a free list containing blocks of that size is maintained. A block of size 2^i is called an ***i-block*** and the free list containing i-blocks is called the ***i-list***. (In practice, it may be unreasonable to keep free blocks of sizes 1, 2, and 4, so that 8 is the smallest free block size allowed; we will ignore the possibility.) However, it may be (and usually is) the case that some of these free lists are empty. Indeed, initially all the lists except the m-list are empty.

Blocks may be allocated only in sizes 2^k for some integer k between 0 and m. If a request for a block of size n is made, an i-block is reserved where i is the smallest integer such that $n <= 2^i$. If no i-block is available (the i-list is empty), an $(i + 1)$-block is removed from the $(i + 1)$-list and is split into two equal size buddies. Each of these buddies is an i-block. One of the buddies is allocated, and the other remains free and is placed on the i-list. If an $(i + 1)$-block is also unavailable, an $(i + 2)$-block is split into two $(i + 1)$-block buddies, one of which is placed on the $(i + 1)$-list and the other of which is split into two i-blocks. One of these i-blocks is allocated and the other is placed onto the i-list. If no $(i + 2)$-block is free, this process continues until either an i-block has been allocated or an m-block is found to be unavailable. In the former case, the allocation attempt is successful; in the latter case a block of proper size is not available.

The buddy system allocation process can best be described as a recursive function *getblock*(n) that returns the address of the block to be allocated, or the null pointer if no block of size n is available. An outline of this function follows:

```
find the smallest integer i such that 2^i >= n;
if (the i-list is not empty) {
   p = the address of the first block on the i-list;
   remove the first block from the i-list;
   return(p);
} /* end if */
else /*the i-list is empty*/
   if (i == m)
      return(null);
   else {
      p = getblock(2^(i+1));
      if (p == null)
         return(null);
      else {
         put the i-block starting at location p on the i-list;
         return(p + 2^i);
      } /* end if */
   } /* end if */
```

In this outline, if an $(i + 1)$-block starts at location p, the two i-blocks into which it is split start at locations p and $p + 2^i$. The first of these remains on the free list, and the second is allocated. Each block is created by splitting a block of one size higher. If an $(i + 1)$-block is split into two i-blocks $b1$ and $b2$, $b1$ and $b2$ are ***buddies*** of each other. The buddy of an i-block at location p is called the ***i-buddy*** of p. Note that a block at location p can have several buddies but only one i-buddy.

If an i-block is freed and its i-buddy is already free, the two buddies are combined into the $(i + 1)$-block from which they were initially created. In this way a larger free block of storage is created to satisfy large requests. If the i-buddy of a newly freed i-block is not free, then the newly freed block is placed directly on the i-list.

Suppose that a newly freed i-block has been combined with its previously free i-buddy into an $(i + 1)$-block. It is possible that the $(i + 1)$-buddy of this recombined $(i + 1)$-block is also free. In that case the two $(i + 1)$-blocks can be recombined further

into an $(i + 2)$-block. This process continues until a recombined block is created whose buddy is not free or until the entire memory is combined into a single m-block.

The liberation algorithm can be outlined as a recursive routine *liberate*(*alloc*, *i*) that frees an *i*-block at location *alloc*:

```
if (i == m) or (the i-buddy of alloc is not free)
   add the i-block at alloc to the i-list
else {
   remove the i-buddy of alloc from the i-list;
   combine the i-block at alloc with its i-buddy;
   p = the address of the newly formed (i + 1)-block;
   liberate(p, i + 1);
} /* end if */
```

Let us refine the outline of *liberate*; we leave the refinement of *getblock* as an exercise for the reader.

There is one obvious question that must be answered. How can the free status of the *i*-buddy of *alloc* be established? Indeed, how can it be determined whether an *i*-buddy of *alloc* exists at all? It is quite possible that the *i*-buddy of *alloc* has been split and part (or all) of it is allocated. Additionally, how can the starting address of the *i*-buddy of *alloc* be determined? If the *i*-block at *alloc* is the first half of its containing $(i + 1)$-block, its *i*-buddy is at *alloc* $+ 2^i$; if the *i*-block is the second half of its containing block, its *i*-buddy is at *alloc* $- 2^i$. How can we determine which is the case?

At this point it would be instructive to look at some examples. For illustrative purposes, consider an absurdly small memory of 1024 $(= 2^{10})$ words. Figure 9.3.7a illustrates this memory after a request for a block of 100 words has been filled. The smallest power of 2 greater than 100 is 128 $(= 2^7)$. Thus, the entire memory is split into two blocks of size 512; the first is placed on the 9-list, and the second is split into two blocks of size 256. The first of these is placed on the 8-list and the second is split into two blocks of size 128, one of which is placed on the 7-list and the second of which is allocated (block *B*1). At the bottom of the figure, the starting addresses of the blocks on each nonempty *i*-list are indicated. Make sure that you follow the execution of the functions *getblock* and *liberate* on this and succeeding examples.

Figure 9.3.7b illustrates the sample memory after filling an additional request for 50 words. There is no free 6-block; therefore the free 7-block at location 768 is split into two 6-blocks. The first 6-block remains free, and the second is allocated as block *B*2. In Figure 9.3.7c three additional 6-blocks have been allocated in the order *B*3, *B*4, and *B*5. When the first request is made, a 6-block at location 768 is free, so that no splitting is necessary. The second request forces the 8-block at 512 to be split into two 7-blocks and the second 7-block at 640 to be split into two 6-blocks. The second of these is allocated as *B*4, and when the next request for a 6-block is made, the first is also allocated as *B*5.

Note that in Figure 9.3.7a the block beginning at 768 is a 7-block, whereas in Figure 9.3.7b it is a 6-block. Similarly, the block at 512 is an 8-block in Figures 9.3.7a and b, but a 7-block in Figure 9.3.7c. This illustrates that the size of a block cannot be determined from its starting address. However, as we shall soon see, a block of a given size can start only at certain addresses.

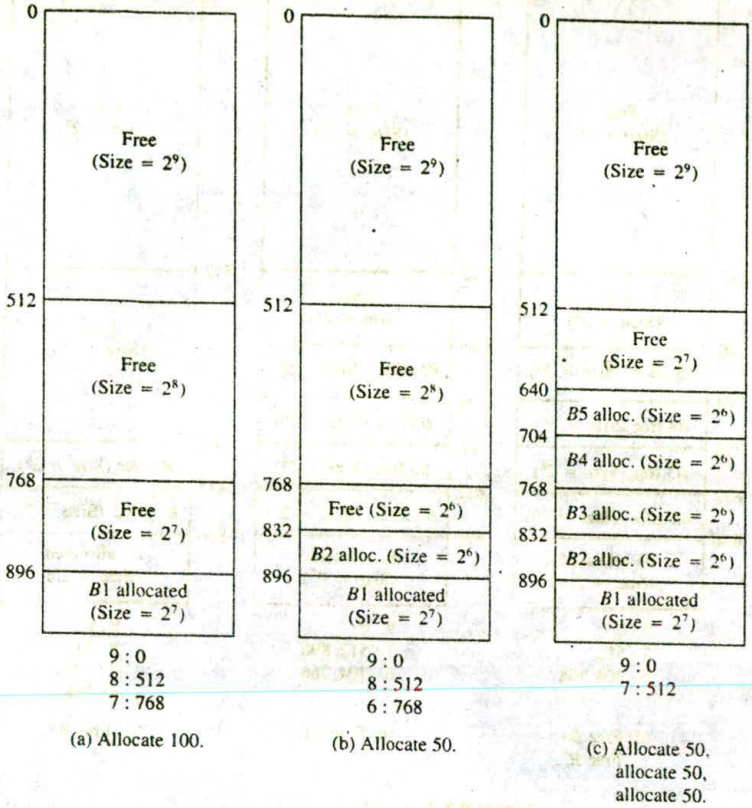


Figure 9.3.7

Figure 9.3.7d illustrates the situation after blocks $B4$ and $B3$ have been freed. When block $B4$ at location 704 is freed, its buddy is examined. Since $B4$ is a 6-block that is the second half of the 7-block from which it was split, its buddy is at location $704 - 2^6 = 640$. However, the 6-block at location 640 (which is $B5$) is not free; therefore no combination can take place. When $B3$ is freed, since it is a 6-block and was the first half of its containing 7-block, its 6-buddy at $768 + 2^6 = 832$ must be examined. However, that 6-buddy is allocated, so again, no combination can take place. Notice that two adjacent blocks of the same size (6-blocks $B4$ and $B3$ at 704 and 768) are free but are not combined into a single 7-block. This is because they are not buddies; that is, they were not originally split from the same 7-block. $B4$ can be combined only with its buddy $B5$, and $B3$ can be combined only with its buddy $B2$.

In Figure 9.3.7e, the 7-block $B1$ has been freed. $B1$ is the second half of its 8-block; therefore its 7-buddy is at $896 - 2^7 = 768$. Although block $B3$, which starts at that location is free, no combination can take place. This is because block $B3$ is not a 7-block, but only a 6-block. This means that the 7-block starting at 768 is split

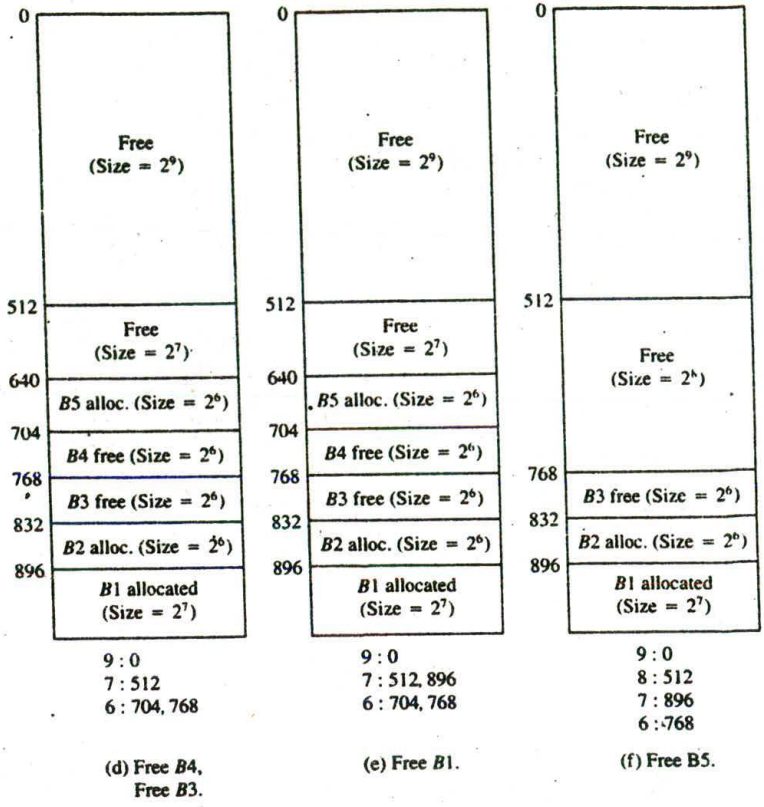


Figure 9.3.7 *(cont.)*

and therefore partially allocated. We see that it is not yet ready for combination. Both the address and the size of a given free block must be considered when making a decision as to whether or not to combine buddies.

In Figure 9.3.7f, the 6-block *B*5 at location 640 is freed. *B*5 is the first half of its containing 7-block; therefore its 6-buddy is at $640 + 2^6 = 704$. That 6-buddy (block *B*4) is already free; therefore the two can be combined into a single 7-block at 640. That 7-block is the second half of its containing 8-block; therefore its 7-buddy is at $640 - 2^7 = 512$. The 7-block at that location is free, so that the two 7-blocks can be combined into an 8-block at 512. That 8-block is the first half of its containing 9-block; therefore its 8-buddy is at $512 + 2^8 = 768$. But the block at location 768 is a 6-block rather than an 8-block; therefore no further combination can take place.

These examples illustrate that it is necessary to be able to determine whether a given i-block is the first or second half of its containing $(i + 1)$-block in order to compute the location of its i-buddy.

Clearly, there is only one m-block in memory, and its starting location is 0. When this block is split, it produces two $(m - 1)$-blocks starting at location 0 and 2^{m-1}. These split into four $(m - 2)$-blocks at locations 0, 2^{m-2}, 2^{m-1}, and $3 * 2^{m-2}$. In general, there are 2^{m-i} i-blocks starting at locations that are integer multiples of 2^i. For example, if $m = 10$ (memory size is 1024), there are $2^{10-6} = 16$ six-blocks starting at locations 0, 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 704, 768, 832, 896, and 960. Each of these addresses is an integral multiple of 64 (which is 2^6), from $0 * 64$ to $15 * 64$.

Notice that any address that is the starting location of an i-block is also the starting location of a k-block for all $0 <= k < i$. This is because the i-block can be split into two $(i - 1)$-blocks, the first of which begins at the same location as the i-block. This is consistent with the observation that an integral multiple of 2^i is also an integral multiple of 2^{i-1}. However, the reverse is not necessarily true. A location that is the starting address of an i-block is the starting address of an $(i + 1)$-block only if the i-block is the first half of the $(i + 1)$-block, but not if it is the second half. For example, in Figure 9.3.7, addresses 640 and 768 begin 7-blocks as well as 6-blocks, and 768 begins an 8-block as well. However, addresses 704 and 832 begin 6-blocks but not 7-blocks.

After making these observations, it is easy to determine whether a given i-block is the first or second half of the $(i + 1)$-block from which it was split. If the starting address p of the i-block is evenly divisible by 2^{i+1}, the block is the first half of an $(i + 1)$-block, and its i-buddy is at $p + 2^i$; otherwise it is the second half of an $(i + 1)$-block and its buddy is at $p - 2^i$.

We can therefore introduce a function *buddy*(p,i) that returns the address of the i-buddy of p [we use an auxiliary function *expon*(a,b) that computes a^b]:

```
if (p % expon(2,i + 1) == 0)
   return(p + expon(2,i));
else
   return(p - expon(2,i));
```

Now that the address of a newly freed block's i-buddy can be found, how can we determine whether or not that buddy is free? One way of making that determination is to traverse the i-list to see whether a block at the desired address is present. If it is, it can be removed and combined with its buddy. If it is not, the newly freed i-block can be added to the i-list. Since each i-list is generally quite small [because as soon as two i-buddies are free they are combined into an $(i + 1)$-buddy and removed from the i-list], this traversal is fairly efficient. Furthermore, to implement this scheme, each i-list need not be doubly linked, since a block is removed from the i-list only after the list is traversed, so that its list predecessor is known.

An alternative method that avoids list traversal is to have each block contain a flag to indicate whether or not it is allocated. Then when an i-block is freed it is possible to determine directly whether or not the block beginning at the address of its buddy is already free. However, this flag alone is insufficient. For example, in Figure 9.3.7e, when 7-block $B1$ at location 896 is freed, its buddy's starting address is calculated as 768. The block at 768 is free and its flag would indicate that fact. Yet the two blocks at 768 and 896 cannot be combined because the block at 768 is not a 7-block but a

6-block whose 6-buddy is allocated. Thus an additional *power* field is necessary in each block. The value of this integer field is the base-two logarithm of its size (for example, if the block is of size 2^i, the value of *power* is i). When an i-block is freed, its buddy's address is calculated. If the *power* field at that address is i and if the flag indicates that the buddy is free, the two blocks are combined.

Under this method, the i-lists are required only for the allocation algorithm so that a block of proper size can be found efficiently. However, because blocks are removed from the i-lists without traversing them, the lists must be doubly linked. Thus each free block must contain four fields: *free*, *power*, *prev*, and *next*. The last two are pointers to the previous and next blocks on the i-list. An allocated block need contain only the *flag* field.

We present the second method of liberation, leaving the first to the reader as an exercise. We assume an array of pointers *list*[$m+1$], where *list*[i] points to the first block on the i-list. We also replace the recursive call to *liberate* by a loop in which successively larger blocks are combined with their buddies until a block is formed whose buddy is not free. The algorithm *liberate*(*alloc*, i) frees an i-block at location *alloc*. For completeness, let us establish that *buddy*(p, m) equals 0. The flag *free* is *true* if the block is free, and *false* otherwise.

```
P = alloc;
bud = buddy(p,i);
while (i < m && free(bud) == TRUE &&  power(bud) == i) {
   /* remove i-buddy of p from the i-list */
   q = prev(bud);
   if (q == null)
      list[i] = next(bud);
   else
      next(q) = next(bud);
   if (next(bud) != null)
      prev(next(bud)) = q;
   /* combine the i-block at p with its buddy */
   if (p % expon(2,i + 1) != 0)
   /* the combined block begins at bud */
      p = bud;
   i++;
   bud = buddy(p,i);
   /* attempt to combine the larger block with its buddy */
} /* end while */
/* add the i-block at p to the i-list */
q = list[i];
prev(p) = null;
next(p) = q;
list[i] = p;
if (q != null)
   prev(q) = p;
/* adjust the fields on the i-block */
power(p) = i;
free(p) = TRUE;
```

Other Buddy Systems

The buddy system that we have just considered is called the ***binary buddy system***, based on the rule that when an i-block (of size 2^i) is split, two equal-sized $(i-1)$-blocks are created. Similarly, two i-blocks that are buddies can be joined into a single $(i+1)$-block.

There are, however, other buddy systems in which a large block is not necessarily split into two equal-sized smaller blocks. One such system is called the ***Fibonacci buddy system***. The sizes of the blocks in this system are based on the Fibonacci numbers first introduced in Section 3.1. Instead of blocks of size 1, 2, 4, 8, 16, ...as in the case of the binary buddy system, the Fibonacci buddy system uses blocks of size 1, 2, 3, 5, 8, 13, ...when an i-block (the size of an i-block in this system is the ith Fibonacci number) is split into two blocks, one of the blocks is an $(i-1)$-block, and the other is an $(i-2)$-block. Thus, for example, a 9-block (of size 34) may split into an 8-block (size 21) and a 7-block (size 13). Similarly, the buddy of an i-block may be either an $(i+1)$-block or an $(i-1)$-block. In the former case, recombination produces an $(i+2)$-block, and in the latter case, an $(i+1)$-block is produced.

Another alternative buddy system is the ***weighted buddy system***. In this scheme a block of size 2^k is split into two blocks, one of size 2^{k-2} and the other of size $3 * 2^{k-2}$. For example, a block of size 64 splits into two buddies of sizes 16 and 48. Rules for recombination are similar.

The philosophy behind such schemes, in which blocks are split into unequal subblocks, is that requests for storage are usually not for sizes that match those of the blocks in the system. Thus, the next larger size block must be used, with the result that space is wasted within the block. For example, in the binary buddy system, when a request is made for a block of size 10, a block of size 16 will be allocated (resulting in 6 wasted bytes); in the Fibonacci buddy system, however, a block of size 13 may be allocated (resulting in only 3 wasted bytes); in the weighted buddy system, a block of size 12 will be sufficient (resulting in only 2 wasted bytes). It is not always the case that the Fibonacci system results in less wasted space than the binary system (for example, a request for a block of size 15), but in general, allowing blocks of varying sizes will more likely produce a closer fit than will requiring groups of blocks to be of uniform size.

An alternative to the foregoing approach is to combine smaller blocks into larger ones only when necessary. In such a scheme, called a ***recombination delaying buddy system***, when a block is freed it is returned to the list of blocks of its size. When a block of a particular size is required, the list of blocks of that size is searched. If a block of the required size is found, the search halts successfully; otherwise a search is made for a pair of blocks of the next smaller size that are buddies. If such a pair exists, the two blocks are combined to form a single block of the required size. If no such pair exists, this process is repeated recursively with successively smaller blocks until either a block of the required size can be formed from smaller blocks, so that the search is successful, or until it is determined that the required block cannot be formed from smaller blocks. Blocks of larger sizes will also be searched to determine if a split is feasible. If a block

of the desired size cannot be found either by splitting larger blocks or by combining smaller blocks, the search ends in failure.

The philosophy behind this scheme is that smaller blocks are often returned to the available pool only to be called for again. Instead of recombining the smaller blocks into a larger block only to decompose it again, the smaller blocks are retained and are recombined into larger blocks only when blocks of the larger size are necessary. The disadvantage of this approach is that blocks of larger sizes may not be available when there is, in fact, enough memory to satisfy their requests. For example, there may be three i-blocks available, two of which are buddies. If a request for an i-block arrives and one of the i-buddies is used to satisfy this request, a subsequent request for an $(i+1)$-block cannot be satisfied. If, on the other hand, the two i-blocks were recombined into an $(i+1)$-block first, the request for an i-block would be satisfied from the isolated i-block before an attempt would be made to break up an $(i+1)$-block. (Of course, it is possible to place the i-block on the i-list in such a way that i-buddies are always at the rear of the list. This prevents the allocation of one of a pair of buddies before an isolated block of the required size is allocated. However, when it is necessary to allocate one of several pairs of buddies, it may be difficult to select the pair that will allow subsequently larger recombinations.)

Yet another variation of the buddy system is the ***tailored list buddy system***. In this system, instead of maintaining the lists in blocks as large as possible (the standard system) and instead of not combining buddies until blocks of a larger size are necessary (the recombination delaying system), the blocks are distributed on the various lists in preassigned proportions.

If the relative frequency of requests for blocks of the possible block sizes are known, memory may be divided initially into blocks of the different sizes according to the given distribution. As blocks are called for and returned to the pool, a record is kept of the actual number of blocks of each block size. When a block is returned to the pool and the number of blocks of that size is at or greater than the number specified by the distribution, an attempt is made to combine the block into a block of the next larger size. This process is repeated successively until no such recombination is possible or until the number of blocks of each size is not exceeded.

When a block of a particular size is requested and there is no block of the required size, a block may be formed either by splitting a block of larger size or by recombining several blocks of smaller size. Various allocation strategies can be used in this case. Very often the distribution of requests is not known in advance. In such a case it is possible to allow the distribution of blocks to stabilize slowly, by maintaining a record of the actual distribution of requests as they arrive. The desired distribution will probably never be achieved exactly, but it can be used as a guide in determining whether and when to recombine blocks.

There are two primary disadvantages to buddy systems. The first is internal fragmentation. For example, in the binary buddy system, only blocks whose sizes are integral powers of 2 can be allocated without waste. This means that a little less than half the storage in each block could be wasted. The other disadvantage is that adjacent free blocks are not combined if they are not buddies. However, simulations have shown that the buddy system does work well and that once the pattern of memory allocations and liberations stabilizes, splitting and combinations take place infrequently.

EXERCISES

9.3.1. Let s be the average size of an allocated block in a system that uses compaction. Let r be the average number of time units between block allocations. Let m be the memory size, f the average percentage of free space, and c the average number of time units between calls on the compaction algorithm. If the memory system is in equilibrium (over a period of time, equal numbers of blocks are allocated as are freed), derive a formula for c in terms of s, r, m, and f.

9.3.2. Implement the first-fit, best-fit, and worst-fit methods of storage allocation in C as follows: Write a function *getblock*(n) that returns the address of a block of size n that is available for allocation and modifies the free list appropriately. The function should utilize the following variables:

memsize, the number of locations in memory

memory[*memsize*], an array of integers representing the memory

freeblock, a pointer to the first location of the first free block on the list

The value of *size*(p) may be obtained by the expression *memory*[p], and the value of *next*(p) by the expression *memory*[p + 1].

9.3.3. Revise the first-fit and best-fit algorithms so that if a block on the free list is less than x units larger than a request, the entire block is allocated as is, unsplit. Revise the C implementations of Exercise 9.3.2 in a similar manner.

9.3.4. Revise the first-fit algorithm and its implementation (see Exercise 9.3.2) so that the free list is circular and is modified in each of the following ways.

(a) The front of the free list is moved up one block after each allocation request.

(b) The front of the free list is reset to the block following the block that satisfied the last allocation request.

(c) If a block is split in meeting an allocation request, the remaining portion of that block is placed at the rear of the free list.

What are the advantages and disadvantages of these methods over the method presented in the text? Which of the three methods yields the smallest average search time? Why?

9.3.5. Design two liberation algorithms in which a newly freed node is placed on the front of the free list when no combinations can be made. Do not use any additional fields other than *next* and *size*. In the first algorithm, when two blocks are combined the combined block is moved to the front of the free list; in the second, the combined block remains at the same position in the free list as its free portion was before the combination. What are the relative merits of the two methods?

9.3.6. Implement the liberation algorithm presented in the text in which the free list is ordered by increasing memory location. Write a C function *liberate*(*alloc*, n) that uses the variables presented in Exercise 9.3.2, where *alloc* is the address of the block to be freed and n is its size. The procedure should modify the free list appropriately.

9.3.7. Implement a storage management system by writing a C program that accepts inputs of two types: An allocation request contains an 'A', the amount of memory requested, and an integer that becomes the identifier of the block being allocated (that is, block 1, block 2, block 3, and so on). A liberation request contains an 'L' and the integer identifying the block to be liberated. The program should call the routines *getblock* and *liberate* programmed in Exercises 9.3.2 and 9.3.6.

9.3.8. Implement the boundary tag method of liberation as a C function, as in Exercises 9.3.2 and 9.3.6. The values of *size*(p) and *bsize*(p) should be obtained by the expression

abs(*memory*[*p*]), *fflag*(*p*) and *bflag*(*p*) by (*memory*[*p*] > 0), *next*(*p*) by *memory* [*p* + 1], and *prev*(*p*) by *memory* [*p* + 2].

9.3.9. How could the free list be organized to reduce the search time in the best-fit method? What liberation algorithm would be used for such a free list?

9.3.10. A storage management system is in ***equilibrium*** if as many blocks are allocated as are liberated in any given time period. Prove the following about a system in equilibrium.

(a) The fraction of total storage that is allocated is fairly constant.

(b) If adjacent free blocks are always combined, the number of allocated blocks is half the number of free blocks.

(c) If adjacent free blocks are always combined, and the average size of an allocated block is greater than some multiple k of the average size of a free block, the fraction of memory that is free is greater than $k/(k + 2)$.

9.3.11. Present allocation and liberation algorithms for the following systems.

(a) Fibonnaci buddy system

(b) Weighted buddy system

(c) Recombination delaying buddy system

(d) Tailored list buddy system

9.3.12. Refine the outline of *getblock*, which is responsible for allocation in the buddy system, into a nonrecursive algorithm that explicitly manipulates free lists.

9.3.13. Prove formally (using mathematical induction) that in the binary buddy system:

(a) There are 2^{m-i} possible i-blocks.

(b) The starting address of an i-block is an integer multiple of 2^i.

9.3.14. Implement the binary buddy system as a set of C programs.