CHAPTER 7

Sorting

7.1 MOTIVATION

In this chapter, we use the term *list* to mean a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as *keys*. Since the same list may be used for several different applications, the key fields for record identification depend on the particular application. For instance, we may regard a telephone directory as a list, each record having three fields: name, address, and phone number. The key is usually the person's name. However, we may wish to locate the record corresponding to a given number, in which case the phone number field would be the key. In yet another application we may desire the phone number at a particular address, so the address field could also be the key.

One way to search for a record with the specified key is to examine the list of records in left-to-right or right-to-left order. Such a search is known as a sequential search. We assume that the list of records is stored in positions 1 through n of an array. We use array indexes 1 through n for our records rather than 0 through $n-1$ because one of the sort methods we develop, heap sort, employs the the array representation of a heap. This representation (see Section 5.6), begins at position 1 of an array. However,

all the sort methods and examples of this chapter are adapted easily to work with record indexes that begin at 0. The datatype of each record is *element* and each record is assumed to have an integer field *key*. Program 7.1 gives a sequential search function that examines the records in the list $a[1{:}n]$ in left-to-right order.

```
int seqSearch(element a[], int k, int n)
{/* search a[1:n]; return the least i such that
    a[i].key = k; return 0, if k is not in the array */
   int i;
   for (i = 1; i <= n && a[i].key != k; i++)
      ;
   if (i > n) return 0;
   return i;
}
```

Program 7.1 Sequential search

If $a[1{:}n]$ does not contain a record with key k, the search is *unsuccessful*. Program 7.1 makes n key comparisons when the search is unsuccessful. For a successful search, the number of key comparisons depends on the position of the search key in the array a. When all keys are distinct and $a[i]$ is being searched for, i key comparisons are made. So, the average number of comparisons for a successful search is

$$(\sum_{1 \le i \le n} i)/n = (n + 1)/2.$$

It is possible to do much better than this when looking up phone numbers. The fact that the entries in the list (i.e., the telephone directory) are in lexicographic order (on the name key) enables one to look up a number while examining only a very few entries in the list. Binary search (see Chapter 1) is one of the better-known methods for searching an ordered, sequential list. A binary search takes only $O(\log n)$ time to search a list with n records. This is considerably better than the $O(n)$ time required by a sequential search. We note that when a sequential search is performed on an ordered list, the conditional of the **for** loop of *seqSearch* can be changed to $i <= n$ && $a[i].key < k$. This change must be accompanied by a change of the conditional $i > n$ to $i > n$ || $a[i].key != k$. These changes improve the performance of Program 7.1 for unsuccessful searches.

Getting back to our example of the telephone directory, we notice that neither a sequential nor a binary search strategy corresponds to the search method actually employed by humans. If we are looking for a name that begins with the letter *W*, we start the search toward the end of the directory rather than at the middle. A search method based on this interpolation scheme would begin by comparing k with $a[i].key$, where

$i = ((k - a[1].key)/(a[n].key - a[1].key)) * n$, and $a[1].key$ and $a[n].key$ are the smallest and largest keys in the list. An interpolation search can be used only when the list is ordered. The behavior of such a search depends on the distribution of the keys in the list.

Let us now look at another example in which the use of ordered lists greatly reduces the computational effort. The problem we are now concerned with is that of comparing two lists of records containing data that are essentially the same but have been obtained from two different sources. Such a problem could arise, for instance, in the case of the United States Internal Revenue Service (IRS), which might receive millions of forms from various employers stating how much they paid their employees and then another set of forms from individual employees stating how much they received. So we have two lists of records, and we wish to verify that there is no discrepancy between the two. Since the forms arrive at the IRS in a random order, we may assume a random arrangement of the records in the lists. The keys here are the social security numbers of the employees.

Let *list*1 be the employer list and *list*2 the employee list. Let *list*1[*i*].*key* and *list*2[*i*].*key*, respectively, denote the key of the *i*th record in *list* 1 and *list* 2. We make the following assumptions about the required verification:

(1) If there is no record in the employee list corresponding to a key in the employer list, a message is to be sent to the employee.

(2) If the reverse is true, then a message is to be sent to the employer.

(3) If there is a discrepancy between two records with the same key, a message to this effect is to be output.

Function *verify*1 (Program 7.2) solves the verification problem by directly comparing the two unsorted lists. The data type of the records in each list is *element* and we assume that the keys are integer. The complexity of *verify*1 is $O(mn)$, where n amd m are, respectively, the number of records in the employer and employee lists. On the other hand, if we first sort the two lists and then do the comparison, we can carry out the verification task in time $O(t_{Sort}(n)+t_{Sort}(m) + n + m)$, where $t_{Sort}(n)$ is the time needed to sort a list of n records. As we shall see, it is possible to sort n records in $O(n\log n)$ time, so the computing time becomes $O(\max\{n\log n, m\log m\})$. Function *verify*2 (Program 7.3) achieves this time.

We have seen two important uses of sorting: (1) as an aid in searching and (2) as a means for matching entries in lists. Sorting also finds application in the solution of many other more complex problems from areas such as optimization, graph theory and job scheduling. Consequently, the problem of sorting has great relevance in the study of computing. Unfortunately, no one sorting method is the best for all applications. We shall therefore study several methods, indicating when one is superior to the others.

First let us formally state the problem we are about to consider. We are given a list of records $(R_1, R_2, \cdots, R_n)$. Each record, R_i, has key value K_i. In addition, we assume an ordering relation $(<)$ on the keys so that for any two key values x and y, $x = y$

```
void verify1(element list1[], element list2[], int n, int m)
{/* compare two unordered lists list1[1:n] and list2[1:m] */
   int i,j, marked[MAX_SIZE];

   for (i = 1; i <= m; i++)
      marked[i] = FALSE;
   for (i = 1; i <= n; i++)
      if ((j = seqSearch(list2,m,list1[i].key)) == 0)
         printf("%d is not in list 2\n",list1[i].key);
      else
      /* check each of the other fields from list1[i] and
         list2[j], and print out any discrepancies */
         marked[j] = TRUE;
   for (i = 1; i <= m; i++)
      if (!marked[i])
         printf("%d is not in list 1\n",list2[i].key);
}
```

Program 7.2: Verifying two unsorted lists using a sequential search

or $x < y$ or $y < x$. The ordering relation (<) is assumed to be transitive (i.e., for any three values x, y, and z, $x < y$ and $y < z$ implies $x < z$). The sorting problem then is that of finding a permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$. The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \cdots, R_{\sigma(n)})$.

Note that when the list has several key values that are identical, the permutation, σ, is not unique. We shall distinguish one permutation, σ_s, from the others that also order the list. Let σ_s be the permutation with the following properties:

(1) $K_{\sigma_s(i)} \leq K_{\sigma_s(i+1)}$, $1 \leq i \leq n-1$.

(2) If $i<j$ and $K_i == K_j$ in the input list, then R_i precedes R_j in the sorted list.

A sorting method that generates the permutation σ_s is *stable*.

We characterize sorting methods into two broad categories: (1) internal methods (i.e., methods to be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory) and (2) external methods (i.e., methods to be used on larger lists). The following internal sorting methods will be developed: insertion sort, quick sort, merge sort, heap sort, and radix sort. This development will be followed by a discussion of external sorting. Throughout, we assume that relational operators have been overloaded so that record comparison is done by comparing their keys.

```
void verify2(element list1[], element list2[], int n, int m)
{/* same as verify1, but we sort list1 and list2 first */
   int i,j;
   sort(list1,n); sort(list2,m);
   i = j = 1;
   while (i <= n && j <= m)
      if (list1[i].key < list2[j].key) {
         printf("%d is not in list 2\n",list1[i].key);
         i++;
      }
      else if (list1[i].key == list2[j].key) {
      /* compare list1[i] and list2[j] on each of the other
         fields and report any discrepancies */
         i++; j++;
      }
      else {
         printf("%d is not in list 1\n", list2[j].key);
         j++;
      }
   for(; i <= n; i++)
      printf("%d is not in list 2\n",list1[i].key);
   for (; j <= m; j++)
      printf("%d is not in list 1\n",list2[j].key);
}
```

Program 7.3: Fast verification of two sorted lists

7.2 INSERTION SORT

The basic step in this method is to insert a new record into a sorted sequence of i records in such a way that the resulting sequence of size $i + 1$ is also ordered. Function *insert* (Program 7.4) accomplishes this insertion.

The use of $a[0]$ enables us to simplify the **while** loop, avoiding a test for end of list (i.e., $i < 1$). In insertion sort, we begin with the ordered sequence $a[1]$ and successively insert the records $a[2]$, $a[3]$, $\cdots$, $a[n]$. Since each insertion leaves the resultant sequence ordered, the list with n records can be ordered making $n - 1$ insertions. The details are given in function *insertionSort* (Program 7.5).

```
void insert(element e, element a[], int i)
{/* insert e into the ordered list a[1:i] such that the
    resulting list a[1:i+1] is also ordered, the array a
    must have space allocated for at least i+2 elements */
   a[0] = e;
   while (e.key < a[i].key)
   {
      a[i+1] = a[i];
      i--;
   }
   a[i+1] = e;
}
```

Program 7.4: Insertion into a sorted list

```
void insertionSort(element a[], int n)
{/* sort a[1:n] into nondecreasing order */
   int j;
   for (j = 2; j <= n; j++) {
      element temp = a[j];
      insert(temp, a, j-1);
   }
}
```

Program 7.5: Insertion sort

Analysis of *insertionSort:* In the worst case *insert* (e, a, i) makes $i + 1$ comparisons before making the insertion. Hence the complexity of *Insert* is $O(i)$. Function *insertion-Sort* invokes *insert* for $i = j - 1 = 1, 2, \cdots, n - 1$. So, the complexity of *insertionSort* is

$$O(\sum_{i=1}^{n-1} (i+1)) = O(n^2).$$

We can also obtain an estimate of the computing time of insertion sort based on the relative disorder in the input list. Record R_i is *left out of order* (LOO) iff $R_i < \max_{1 \le j < i}\{R_j\}$. The insertion step has to be carried out only for those records that are LOO. If k is the number of LOO records, the computing time is $O((k + 1)n) = O(kn)$. We can show that the average time for *insertionSort* is $O(n^2)$ as well. □

Example 7.1: Assume that $n = 5$ and the input key sequence is 5, 4, 3, 2, 1. After each insertion we have

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

For convenience, only the key field of each record is displayed, and the sorted part of the list is shown in bold. Since the input list is in reverse order, as each new record is inserted into the sorted part of the list, the entire sorted part is shifted right by one position. Thus, this input sequence exhibits the worst-case behavior of insertion sort. □

Example 7.2: Assume that $n = 5$ and the input key sequence is 2, 3, 4, 5, 1. After each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

In this example, only record 5 is LOO, and the time for each $j = 2$, 3, and 4 is O(1), whereas for $j = 5$ it is $O(n)$. □

It should be fairly obvious that *insertionSort* is stable. The fact that the computing time is $O(kn)$ makes this method very desirable in sorting sequences in which only a very few records are LOO (i.e., $k<<n$). The simplicity of this scheme makes it about the fastest sorting method for small n (say, $n \leq 30$).

Variations

1. ***Binary Insertion Sort:*** We can reduce the number of comparisons made in an insertion sort by replacing the sequential searching technique used in *insert* (Program 7.4) with binary search. The number of record moves remains unchanged.

2. ***Linked Insertion Sort:*** The elements of the list are represented as a linked list rather than as an array. The number of record moves becomes zero because only the link fields

require adjustment. However, we must retain the sequential search used in *insert*.

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each iteration of the **for** loop of *insertionSort* (Program 7.5).
2. Write a function that implements binary insertion sort. What is the worst-case number of comparisons made by your sort function? What is the worst-case number of record moves made? How do these compare with the corresponding numbers for Program 7.5?
3. Write a function that implements linked insertion sort. What is the worst-case number of comparisons made by your sort function? What is the worst-case number of record moves made? How do these compare with the corresponding numbers for Program 7.5?

7.3 QUICK SORT

We now turn our attention to a sorting scheme with very good average behavior. The quick sort scheme developed by C. A. R. Hoare has the best average behavior among the sorting methods we shall be studying. In quick sort, we select a pivot record from among the records to be sorted. Next, the records to be sorted are reordered so that the keys of records to the left of the pivot are less than or equal to that of the pivot and those of the records to the right of the pivot are greater than or equal to that of the pivot. Finally, the records to the left of the pivot and those to its right are sorted independently (using the quick sort method recursively).

Program 7.6 gives the resulting quick sort function. To sort $a[1{:}n]$, the function invocation is *quickSort* (a, 1, n). Function *quickSort* assumes that $a[n + 1]$ has been set to have a key at least as large as the remaining keys.

Example 7.3: Suppose we are to sort a list of 10 records with keys (26, 5, 37, 1, 61, 11, 59, 15, 48, 19). Figure 7.1 gives the status of the list at each call of *quickSort*. Square brackets indicate sublists yet to be sorted. □

Analysis of *quickSort*: The worst-case behavior of *quickSort* is examined in Exercise 2 and shown to be $O(n^2)$. However, if we are lucky, then each time a record is correctly positioned, the sublist to its left will be of the same size as that to its right. This would leave us with the sorting of two sublists, each of size roughly $n/2$. The time required to position a record in a list of size n is $O(n)$. If $T(n)$ is the time taken to sort a list of n records, then when the list splits roughly into two equal parts each time a record is positioned correctly, we have

```
void quickSort(element a[], int left, int right)
{/* sort a[left:right] into nondecreasing order
    on the key field; a[left].key is arbitrarily
    chosen as the pivot key;  it is assumed that
    a[left].key <= a[right+1].key */
   int pivot,i,j;
   element temp;
   if (left < right) {
      i = left; j = right + 1;
      pivot = a[left].key;
      do {/* search for keys from the left and right
             sublists, swapping out-of-order elements until
             the left and right boundaries cross or meet */
         do i++; while (a[i].key < pivot);
         do j--; while (a[j].key > pivot);
         if (i < j) SWAP(a[i],a[j],temp);
      } while (i < j);
      SWAP(a[left],a[j],temp);
      quickSort(a,left,j-1);
      quickSort(a,j+1,right);
   }
}
```

Program 7.6: Quick sort

$$
\begin{aligned}
T(n) &\le cn + 2T(n/2), \text{ for some constant } c \\
&\le cn + 2(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\quad . \\
&\quad . \\
&\quad . \\
&\le cn \log_2 n + nT(1) = O(n \log n)
\end{aligned}
$$

Lemma 7.1 shows that the average computing time for function *quickSort* is $O(n \log n)$. Moreover, experimental results show that as far as average computing time is concerned, Quick sort is the best of the internal sorting methods we shall be studying.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Figure 7.1: Quick sort example

Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function *quickSort* to sort a list with n records. Then there exists a constant k such that $T_{avg}(n) \leq kn\log_e n$ for $n \geq 2$.

Proof: In the call to *quickSort* (*list* , 1, n), the pivot gets placed at position j. This leaves us with the problem of sorting two sublists of size $j-1$ and $n-j$. The expected time for this is $T_{avg}(j-1) + T_{avg}(n-j)$. The remainder of the function clearly takes at most cn time for some constant c. Since j may take on any of the values 1 to n with equal probability, we have

$$T_{avg}(n) \leq cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j)) = cn + \frac{2}{n}\sum_{j=0}^{n-1}T_{avg}(j) \qquad (7.1)$$

for $n \geq 2$. We may assume $T_{avg}(0) \leq b$ and $T_{avg}(1) \leq b$ for some constant b. We shall now show $T_{avg}(n) \leq kn\log_e n$ for $n \geq 2$ and $k = 2(b + c)$. The proof is by induction on n.

Induction base: For $n = 2$, Eq. (7.1) yields $T_{avg}(2) \leq 2c + 2b \leq kn\log_e 2$.

Induction hypothesis: Assume $T_{avg}(n) \leq kn\log_e n$ for $1 \leq n < m$.

Induction step: From Eq. (7.1) and the induction hypothesis we have

$$T_{avg}(m) \leq cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1}T_{avg}(j) \leq cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1}j\log_e j \qquad (7.2)$$

Since $j\log_e j$ is an increasing function of j, Eq. (7.2) yields

$$T_{avg}(m) \leq cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x\,dx = cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \leq km\log_e m, \text{ for } m \geq 2 \ \square$$

Unlike insertion sort, where the only additional space needed was for one record, quick sort needs stack space to implement the recursion. If the lists split evenly, as in the above analysis, the maximum recursion depth would be log n, requiring a stack space of O(log n). The worst case occurs when the list is split into a left sublist of size $n - 1$ and a right sublist of size 0 at each level of recursion. In this case, the depth of recursion becomes n, requiring stack space of O(n). The worst-case stack space can be reduced by a factor of 4 by realizing that right sublists of size less than 2 need not be stacked. An asymptotic reduction in stack space can be achieved by sorting smaller sublists first. In this case the additional stack space is at most O(log n).

Variation—Quick sort using a median-of-three: Our version of quick sort always picked the key of the first record in the current sublist as the pivot. A better choice for this pivot is the median of the first, middle, and last keys in the current sublist. Thus, *pivot* = *median* $\{K_l, K_{(l+r)/2}, K_r\}$. For example, *median*{10, 5, 7} = 7 and *median*{10, 7, 7} = 7.

EXERCISES

1. Draw a figure similar to Figure 7.1 starting with the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18).
2. (a) Show that *quickSort* takes O(n^2) time when the input list is already in sorted order.
   (b) Show that the worst-case time complexity of *quickSort* is O(n^2).
   (c) Why is *list* [*left*] ≤ *list* [*right* + 1] required in *quickSort*?
3. (a) Write a nonrecursive version of *quickSort* incorporating the median-of-three rule to determine the pivot key.
   (b) Show that this function takes O(n log n) time on an already sorted list.
4. Show that if smaller sublists are sorted first, then the recursion in *quickSort* can be simulated by a stack of depth O(log n).
5. Quick sort is an unstable sorting method. Give an example of an input list in which the order of records with equal keys is not preserved.

7.4 HOW FAST CAN WE SORT?

Both of the sorting methods we have seen so far have a worst-case behavior of O(n^2). It is natural at this point to ask the question, What is the best computing time for sorting

that we can hope for? The theorem we shall prove shows that if we restrict our question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n \log n)$ is the best possible time.

The method we use is to consider a tree that describes the sorting process. Each vertex of the tree represents a key comparison, and the branches indicate the result. Such a tree is called a *decision tree*. A path through a decision tree represents a sequence of computations that an algorithm could produce.

Example 7.4: Let us look at the decision tree obtained for insertion sort working on a list with three records (Figure 7.2). The input sequence is R_1, R_2, and R_3, so the root of the tree is labeled [1, 2, 3]. Depending on the outcome of the comparison between keys K_1 and K_2, this sequence may or may not change. If $K_2 < K_1$, then the sequence becomes [2, 1, 3]; otherwise it stays [1, 2, 3]. The full tree resulting from these comparisons is given in Figure 7.2.

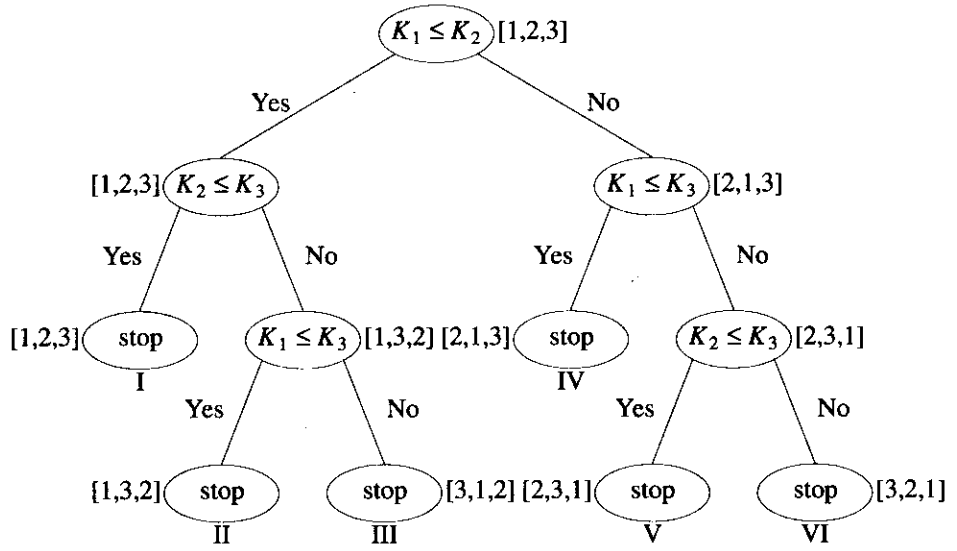


Figure 7.2: Decision tree for insertion sort

The leaf nodes are labeled I to VI. These are the only points at which the algorithm may terminate. Hence, only six permutations of the input sequence are obtainable from this algorithm. Since all six of these are different, and $3! = 6$, it follows that this algorithm has enough leaves to constitute a valid sorting algorithm for three records. The maximum depth of this tree is 3. Figure 7.3 gives six different orderings of the key

values 7, 9, and 10, which show that all six permutations are possible. □

leaf	permutation	sample input key values that give the permutation
I	1 2 3	[7, 9, 10]
II	1 3 2	[7, 10, 9]
III	3 1 2	[9, 10, 7]
IV	2 1 3	[9, 7, 10]
V	2 3 1	[10, 7, 9]
VI	3 2 1	[10, 9, 7]

Figure 7.3: Sample input permutations

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Proof: When sorting n elements, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves. But a decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is k. Therefore, the height must be at least $\log_2 n! + 1$. □

Corollary: Any algorithm that sorts only by comparisons must have a worst-case computing time of $\Omega(n \log n)$.

Proof: We must show that for every decision tree with $n!$ leaves, there is a path of length $cn\log_2 n$, where c is a constant. By the theorem, there is a path of length $\log_2 n!$. Now

$$n! = n(n-1)(n-2) \cdots (3)(2)(1) \geq (n/2)^{n/2}$$

So, $\log_2 n! \geq (n/2)\log_2(n/2) = \Omega(n \log n)$. □

Using a similar argument and the fact that binary trees with 2^n leaves must have an average root-to-leaf path length of $\Omega(n \log n)$, we can show that the average complexity of comparison-based sorting methods is $\Omega(n \log n)$.

7.5.1 Merging

Before looking at the merge sort method to sort n records, let us see how one may merge two sorted lists to get a single sorted list. Program 7.7 gives the code for this. The two lists to be merged are *initList* [i:m] and *initList* [m+1:n]. The resulting merged list is *mergedList* [i:n].

```
void merge(element initList[], element mergedList[],
           int i, int m, int n)
{/*  the sorted lists initList[i:m] and initList[m+1:n] are
     merged to obtain the sorted list mergedList[i:n] */
   int j,k,t;
   j = m+1;        /* index for the second sublist */
   k = i;          /* index for the merged list */

   while (i <= m && j <= n) {
      if (initList[i].key <= initList[j].key)
         mergedList[k++] = initList[i++];
      else
         mergedList[k++] = initList[j++];
   }
   if (i > m)
   /* mergedList[k:n] = initList[j:n] */
      for (t = j; t <= n; t++)
         mergedList[t] = initList[t];
      else
      /* mergedList[k:n] = initList[i:m] */
         for (t = i; t <= m; t++)
            mergedList[k+t-i] = initList[t];
}
```

Program 7.7: Merging two sorted lists

Analysis of *merge:* In each iteration of the **while** loop, k increases by 1. The total increment in k is at most $n - i + 1$. Hence, the **while** loop is iterated at most $n - i + 1$ times. The **for** statements copy at most $n - i + 1$ records. The total time is therefore $O(n - i + 1)$.

If each record has a size s, then the time is $O(s(n - l + 1))$. When s is greater than

1, we can use linked lists instead of arrays and obtain a new sorted linked list containing these $n - l + 1$ records. Now, we will not need the additional space for $n - l + 1$ records as needed in *merge* for the array *mergedList*. Instead, space for $n - l + 1$ links is needed. The merge time becomes $O(n - i + 1)$ and is independent of s. Note that $n - l + 1$ is the number of records being merged. □

7.5.2 Iterative Merge Sort

This version of merge sort begins by interpreting the input list as comprised of n sorted sublists, each of size 1. In the first merge pass, these sublists are merged by pairs to obtain $n/2$ sublists, each of size 2 (if n is odd, then one sublist is of size 1). In the second merge pass, these $n/2$ sublists are then merged by pairs to obtain $n/4$ sublists. Each merge pass reduces the number of sublists by half. Merge passes are continued until we are left with only one sublist. The example below illustrates the process.

Example 7.5: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). The tree of Figure 7.4 illustrates the sublists being merged at each pass. □

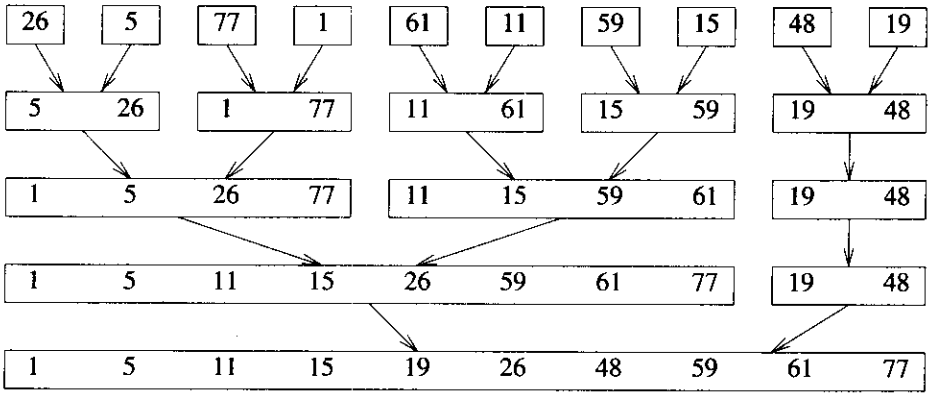


Figure 7.4: Merge tree

Since a merge sort consists of several merge passes, it is convenient first to write a function (Program 7.8) for a merge pass. Now the sort can be done by repeatedly invoking the merge-pass function as in Program 7.9.

```
void mergePass(element initList[], element mergedList[],
               int n, int s)
{/* perform one pass of the merge sort, merge adjacent
    pairs of sorted segments from initList[] into mergedList[],
    n is the number of elements in the list, s is
    the size of each sorted segment */
  int i,j;
  for (i = 1; i <= n - 2 * s + 1; i += 2 * s)
    merge(initList,mergedList,i,i + s - 1,i + 2 * s - 1);
 if (i + s - 1 < n)
    merge(initList,mergedList,i,i + s - 1,n);
 else
    for (j = i; j <= n; j++)
       mergedList[j] = initList[j];
}
```

Program 7.8: A merge pass

```
void mergeSort(element a[], int n)
{/* sort a[1:n] using the merge sort method */
   int s = 1; /* current segment size */
   element extra[MAX_SIZE];

   while (s < n) {
      mergePass(a, extra, n, s);
      s *= 2;
      mergePass(extra, a, n, s);
      s *= 2;
   }
}
```

Program 7.9: Merge sort

Analysis of *mergeSort*: A merge sort consists of several passes over the input. The first pass merges segments of size 1, the second merges segments of size 2, and the ith pass merges segments of size 2^{i-1}. Thus, the total number of passes is $\lceil \log_2 n \rceil$. As *merge* showed, we can merge two sorted segments in linear time, which means that each pass

takes $O(n)$ time. Since there are $\lceil \log_2 n \rceil$ passes, the total computing time is $O(n \log n)$. □

You may verify that *mergeSort* is a stable sorting function.

7.5.3 Recursive Merge Sort

In the recursive formulation we divide the list to be sorted into two roughly equal parts called the left and the right sublists. These sublists are sorted recursively, and the sorted sublists are merged.

Example 7.6: The input list (26, 5, 77, 1, 61, 11, 59, 15, 49, 19) is to be sorted using the recursive formulation of merge sort. If the sublist from *left* to *right* is currently to be sorted, then its two sublists are indexed from *left* to $\lfloor (left + right)/2 \rfloor$ and from $\lfloor (left + right)/2 \rfloor + 1$ to *right*. The sublist partitioning that takes place is described by the binary tree of Figure 7.5. Note that the sublists being merged are different from those being merged in *mergeSort*. □

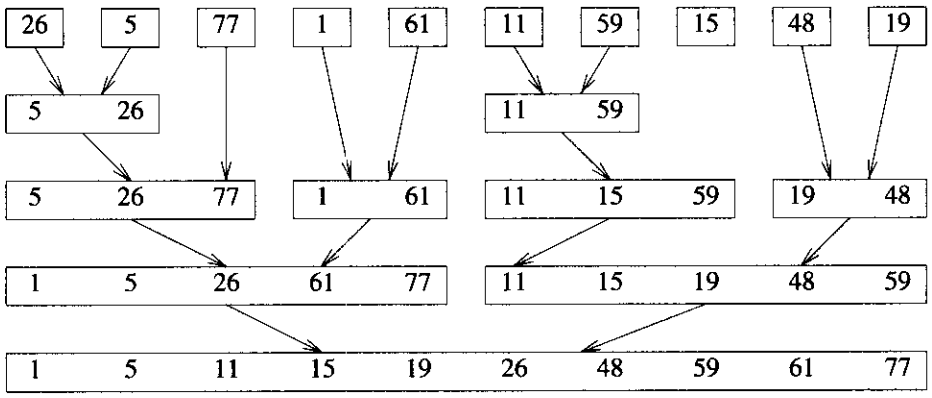


Figure 7.5: Sublist partitioning for recursive merge sort

To eliminate the record copying that takes place when *merge* (Program 7.7) is used to merge sorted sublists we associate an integer pointer with each record. For this purpose, we employ an integer array *link*[1:*n*] such that *link*[*i*] gives the record that follows record *i* in the sorted sublist. In case *link*[*i*] = 0, there is no next record. With the addition of this array of links, record copying is replaced by link changes and the

runtime of our sort function becomes independent of the size s of a record. Also the additional space required is $O(n)$. By comparison, the iterative merge sort described earlier takes $O(sn\log n)$ time and $O(sn)$ additional space. On the down side, the use of an array of links yields a sorted chain of records and we must have a follow up process to physically rearrange the records into the sorted order dictated by the final chain. We describe the algorithm for this physical rearrangement in Section 7.8.

We assume that initially $link[i] = 0$, $1 \le i \le n$. Thus, each record is initially in a chain containing only itself. Let *start*1 and *start*2 be pointers to two chains of records. The records on each chain are in nondecreasing order. Let *listMerge* (*a*, *link*, *start*1, *start*2) be a function that merges two chains *start*1 and *start*2 in array *a* and returns the first position of the resulting chain that is linked in nondecreasing order of key values. The recursive version of merge sort is given by function *rmergeSort* (Program 7.10). To sort the array $a[1{:}n]$ this function is invoked as *rmergeSort* (*a* , *link*, 1, *n*). The start of the chain ordered as described earlier is returned. Function *listMerge* is given in Program 7.11.

```
int rmergeSort(element a[], int link[], int left, int right)
{/* a[left:right] is to be sorted, link[i] is initially 0
    for all i, returns the index of the first element in the
    sorted chain */
   if (left >= right) return left;
   int mid = (left + right) / 2;
   return listMerge(a, link,
                    rmergeSort(a, link, left, mid),
                           /* sort left half */
                    rmergeSort(a, link, mid + 1, right));
                           /* sort right half */
}
```

Program 7.10: Recursive merge sort

Analysis of *rmergeSort*: It is easy to see that recursive merge sort is stable, and its computing time is $O(n \log n)$. □

Variation—Natural Merge Sort: We may modify *mergeSort* to take into account the prevailing order within the input list. In this implementation we make an initial pass over the data to determine the sublists of records that are in order. Merge sort then uses these initially ordered sublists for the remainder of the passes. Figure 7.6 shows natural merge sort using the input sequence of Example 7.6.

```
int listMerge(element a[], int link[], int start1, int start2)
{/* sorted chains beginning at start1 and start2,
    respectively, are merged; link[0] is used as a
    temporary header; returns start of merged chain */
   int last1, last2, lastResult = 0;
   for (last1 = start1, last2 = start2; last1 && last2;)
      if (a[last1] <= a[last2]) {
         link[lastResult] = last1;
         lastResult = last1; last1 = link[last1];
      }
      else {
         link[lastResult] = last2;
         lastResult = last2; last2 = link[last2];
      }

   /* attach remaining records to result chain */
   if (last1 == 0) link[lastResult] = last2;
   else link[lastResult] = last1;
   return link[0];
}
```

Program 7.11: Merging sorted chains

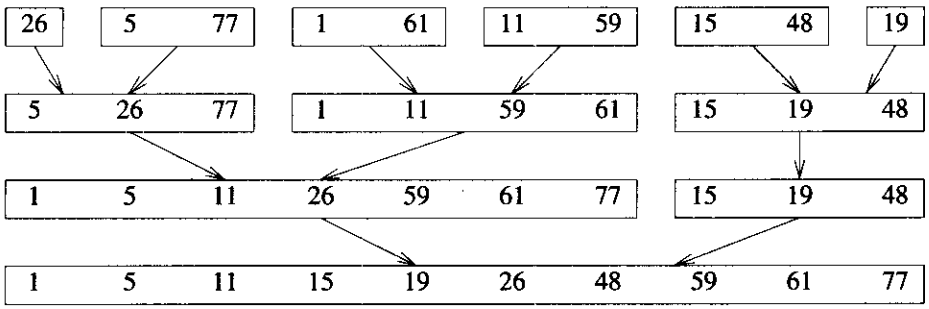


Figure 7.6: Natural merge sort

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each phase of *mergeSort* (Program 7.9).
2. Prove that *mergeSort* is stable.
3. Write an iterative natural merge sort function using arrays as in function *mergeSort*. How much time does this function take on an initially sorted list? Note that *mergeSort* takes $O(n \log n)$ on such an input list. What is the worst-case computing time of the new function? How much additional space is needed?
4. Do the previous exercise using chains.

7.6 HEAP SORT

Although the merge sort scheme discussed in the previous section has a computing time of $O(n \log n)$, both in the worst case and as average behavior, it requires additional storage proportional to the number of records to be sorted. The sorting method we are about to study, heap sort, requires only a fixed amount of additional storage and at the same time has as its worst-case and average computing time $O(n \log n)$. However, heap sort is slightly slower than merge sort.

In heap sort, we utilize the max-heap structure introduced in Chapter 5. The deletion and insertion functions associated with max heaps directly yield an $O(n \log n)$ sorting method. The n records are first inserted into an initially empty max heap. Next, the records are extracted from the max heap one at a time. It is possible to create the max heap of n records faster than by inserting the records one by one into an initially empty heap. For this, we use the function *adjust* (Program 7.12), which starts with a binary tree whose left and right subtrees are max heaps and rearranges records so that the entire binary tree is a max heap. The binary tree is embedded within an array using the standard mapping. If the depth of the tree is d, then the **for** loop is executed at most d times. Hence the computing time of *adjust* is $O(d)$.

To sort the list, first we create a max heap by using *adjust* repeatedly, as in the first **for** loop of function *heapSort* (Program 7.13). Next, we swap the first and last records in the heap. Since the first record has the maximum key, the swap moves the record with maximum key into its correct position in the sorted array. We then decrement the heap size and readjust the heap. This swap, decrement heap size, readjust heap process is repeated $n-1$ times to sort the entire array $a[1{:}n]$. Each repetition of the process is called a pass. For example, on the first pass, we place the record with the highest key in the nth position; on the second pass, we place the record with the second highest key in position $n-1$; and on the ith pass, we place the record with the ith highest key in position $n-i+1$.

Example 7.7: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). If we interpret this list as a binary tree, we get the tree of Figure 7.7(a). Figure 7.7(b) depicts the max heap

```
void adjust(element a[], int root, int n)
{/* adjust the binary tree to establish the heap */
   int child,rootkey;
   element temp;
   temp = a[root];
   rootkey = a[root].key;
   child = 2 * root;                          /* left child */
   while (child <= n) {
      if ((child < n) &&
      (a[child].key < a[child+1].key))
         child++;
      if (rootkey > a[child].key) /* compare root and
                                     max. child */
         break;
      else {
         a[child / 2] = a[child]; /* move to parent */
         child *= 2;
      }
   }
   a[child/2] = temp;
}
```

Program 7.12: Adjusting a max heap

after the first **for** loop of *heapSort*. Figure 7.8 shows the array of records following each of the first seven iterations of the second **for** loop. The portion of the array that still represents a max heap is shown as a binary tree; the sorted part of the array is shown as an array. □

Analysis of *heapSort:* Suppose $2^{k-1} \le n < 2^k$, so the tree has k levels and the number of nodes on level i is $\le 2^{i-1}$. In the first **for** loop, *adjust* (Program 7.12) is called once for each node that has a child. Hence, the time required for this loop is the sum, over each level, of the number of nodes on a level multiplied by the maximum distance the node can move. This is no more than

$$\sum_{1 \le i \le k} 2^{i-1}(k-i) = \sum_{1 \le i \le k-1} 2^{k-i-1} i \le n \sum_{1 \le i \le k-1} i/2^i < 2n = O(n)$$

In the next **for** loop, $n-1$ applications of *adjust* are made with maximum tree-depth $k = \lceil \log_2 (n+1) \rceil$ and *SWAP* is invoked $n-1$ times. Hence, the computing time for this loop is $O(n \log n)$. Consequently, the total computing time is $O(n \log n)$. Note that apart

```
void heapSort(element a[], int n)
{/* perform a heap sort on a[1:n] */
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(a,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(a[1],a[i+1],temp);
      adjust(a,1,i);
   }
}
```

Program 7.13: Heap sort

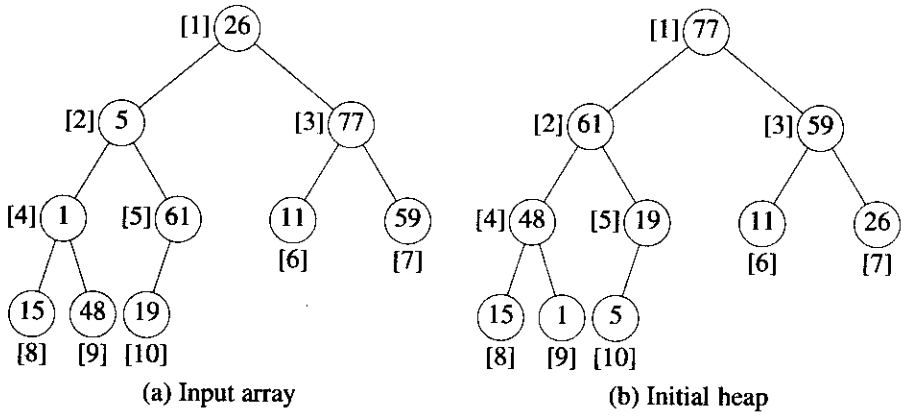


Figure 7.7: Array interpreted as a binary tree

from some simple variables, the only additional space needed is space for one record to carry out the swap in the second **for** loop. □

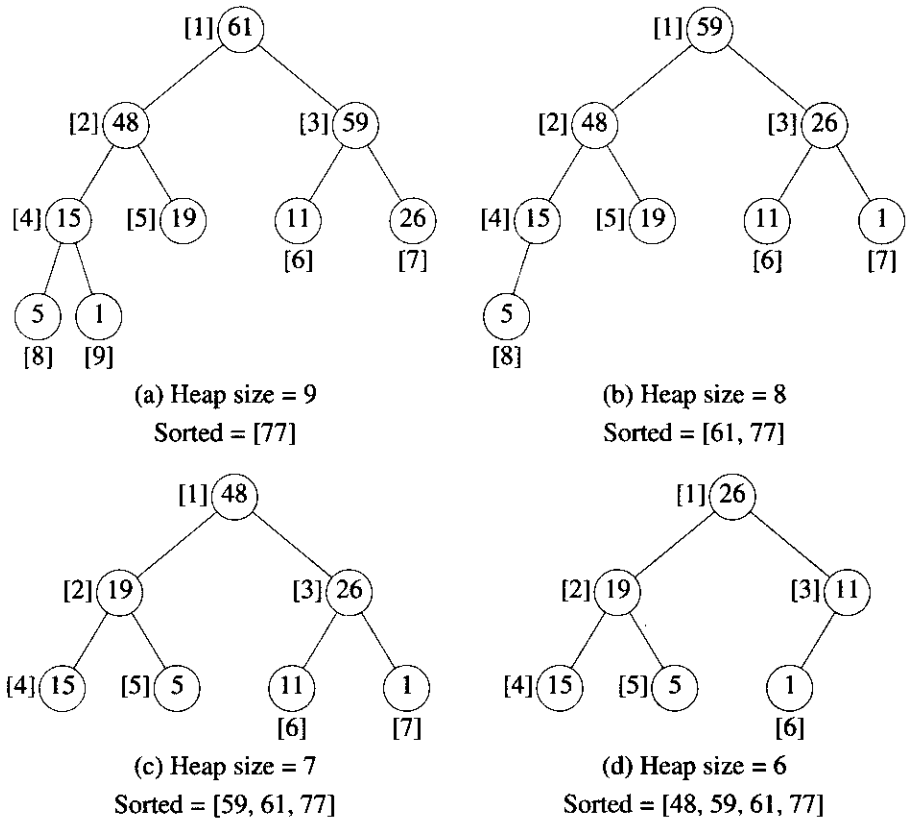
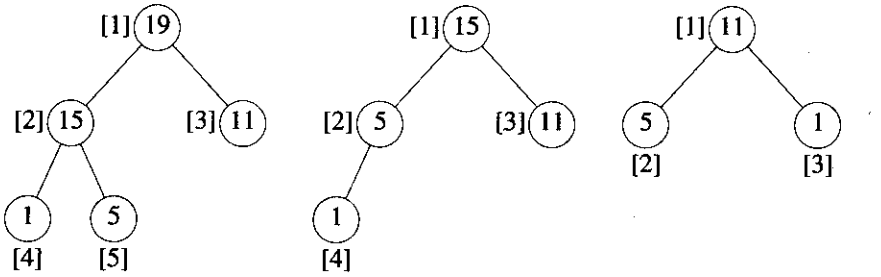


Figure 7.8: Heap sort example (continued on next page)

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of the first **for** loop as well as at the end of each iteration of the second **for** loop of *heapSort* (Program 7.13).
2. Heap sort is unstable. Give an example of an input list in which the order of records with equal keys is not preserved.



(e) Heap size = 5 [26, 48, 59, 61, 77]

(f) Heap size = 4 [19, 26, 48, 59, 61, 77]

(g) Heap size = 3 [15, 19, 26, 48, 59, 61, 77]

Figure 7.8: Heap sort example

7.7 SORTING ON SEVERAL KEYS

We now look at the problem of sorting records on several keys, $K^1, K^2, \cdots, K^r$ (K^1 is the most significant key and K^r the least). A list of records $R_1, \cdots, R_n$ is said to be sorted with respect to the keys $K^1, K^2, \cdots, K^r$ iff for every pair of records i and j, $i < j$ and $(K_i^1, \cdots, K_i^r) \leq (K_j^1, \cdots, K_j^r)$. The r-tuple $(x_1, \cdots, x_r)$ is less than or equal to the r-tuple $(y_1, \cdots, y_r)$ iff either $x_i = y_i$, $1 \leq i \leq j$, and $x_{j+1} < y_{j+1}$ for some $j < r$, or $x_i = y_i$, $1 \leq i \leq r$.

For example, the problem of sorting a deck of cards may be regarded as a sort on two keys, the suit and face values, with the following ordering relations:

K^1 [Suits]: $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$

K^2 [Face values]: $2 < 3 < 4 \cdots < 10 < \mathrm{J} < \mathrm{Q} < \mathrm{K} < \mathrm{A}$

A sorted deck of cards therefore has the following ordering:

$$2\clubsuit, \ldots, \mathrm{A}\clubsuit, \ldots, 2\spadesuit, \ldots, \mathrm{A}\spadesuit$$

There are two popular ways to sort on multiple keys. In the first, we begin by sorting on the most significant key K^1, obtaining several "piles" of records, each having the same value for K^1. Then each of these piles is independently sorted on the key K^2 into "subpiles" such that all the records in the same subpile have the same values for K^1 and K^2. The subpiles are then sorted on K^3, and so on, and the piles are combined. Using this method on our card deck example, we would first sort the 52 cards into four piles, one for each of the suit values, then sort each pile on the face value. Then we would

place the piles on top of each other to obtain the desired ordering.

A sort proceeding in this fashion is referred to as a most-significant-digit-first (MSD) sort. The second way, quite naturally, is to sort on the least significant digit first (LSD). An LSD sort would mean sorting the cards first into 13 piles corresponding to their face values (key K^2). Then, we would place the 3's on top of the 2's, $\cdots$, the kings on top of the queens, the aces on top of the kings; we would turn the deck upside down and sort on the suit (K^1) using a stable sorting method to obtain four piles, each orderd on K^2; and we would combine the piles to obtain the required ordering on the cards.

Comparing the two functions outlined here (MSD and LSD), we see that LSD is simpler, as the piles and subpiles obtained do not have to be sorted independently (provided the sorting scheme used for sorting on the keys K^i, $1 \le i < r$, is stable). This in turn implies less overhead.

The terms LSD and MSD specify only the order in which the keys are to be sorted. They do not specify how each key is to be sorted. When sorting a card deck manually, we generally use an MSD sort. The sorting on suit is done by a *bin sort* (i.e., four "bins" are set up, one for each suit value and the cards are placed into their corresponding bins). Next, the cards in each bin are sorted using an algorithm similar to insertion sort. However, there is another way to do this. First use a bin sort on the face value. To do this we need 13 bins, one for each distinct face value. Then collect all the cards together as described above and perform a bin sort on the suits using four bins. Note that a bin sort requires only $O(n)$ time if the spread in key values is $O(n)$.

LSD or MSD sorting can be used to sort even when the records have only one key. For this, we interpret the key as being composed of several subkeys. For example, if the keys are numeric, then each decimal digit may be regarded as a subkey. So, if the keys are in the range $0 \le K \le 999$, we can use either the LSD or MSD sorts for three keys (K^1, K^2, K^3), where K^1 is the digit in the hundredths place, K^2 the digit in the tens place, and K^3 the digit in the units place. Since $0 \le K^i \le 9$ for each key K^i, the sort on each key can be carried out using a bin sort with 10 bins.

In a *radix sort*, we decompose the sort key using some radix r. When r is 10, we get the decimal decomposition described above. When $r = 2$, we get binary decomposition of the keys. In a Radix-r Sort, the number of bins required is r.

Assume that the records to be sorted are $R_1, \cdots, R_n$. The record keys are decomposed using a radix of r. This results in each key having d digits in the range 0 through $r - 1$. Thus, we shall need r bins. The records in each bin will be linked together into a chain with $front[i]$, $0 \le i < r$, a pointer to the first record in bin i and $rear[i]$, a pointer to the last record in bin i. These chains will operate as queues. Function *radixSort* (Program 7.14) formally presents the LSD radix-r method.

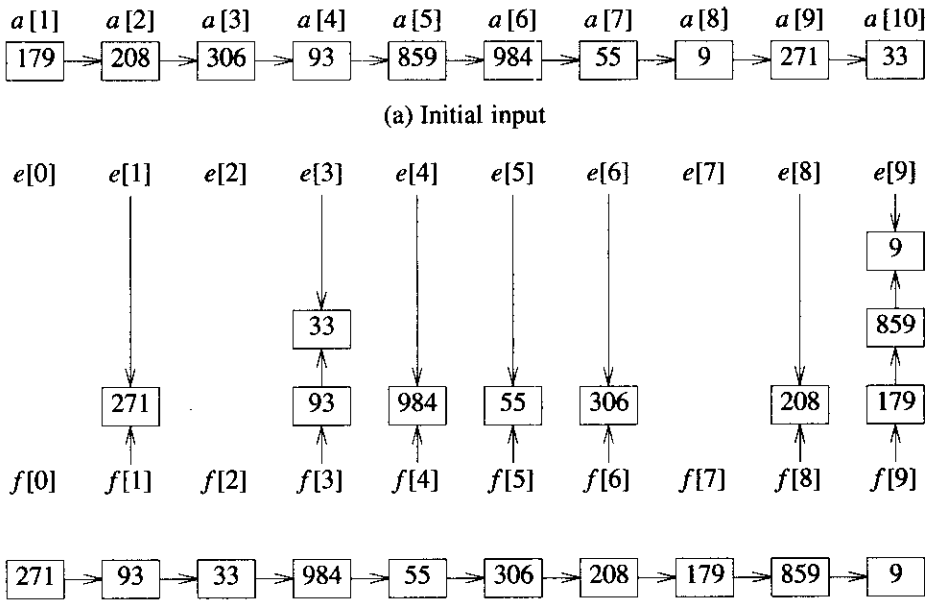
Analysis of *radixSort:* *radixSort* makes d passes over the data, each pass taking $O(n + r)$ time. Hence, the total computing time is $O(d(n + r))$. The value of d will depend on the choice of the radix r and also on the largest key. Different choices of r

```
int radixSort(element a[], int link[], int d, int r, int n)
{/* sort a[1:n]) using a d-digit radix-r sort, digit(a[i],j,r)
    returns the jth radix-r digit (from the left) of a[i]'s key;
    each digit is in the range is [0,r); sorting within a digit
    is done using a bin sort */
   int front[r], rear[r]; /* queue front and rear pointers */
   int i, bin, current, first, last;
   /* create initial chain of records starting at first */
   first = 1;
   for (i = 1; i < n; i++) link[i] = i + 1;
   link[n] = 0;

   for (i = d-1; i >= 0; i--)
   {/* sort on digit i */
      /* initialize bins to empty queues */
      for (bin = 0; bin < r; bin++) front[bin] = 0;

      for (current = first; current; current = link[current])
      {/* put records into queues/bins */
         bin = digit(a[current],i,r);
         if (front[bin] == 0) front[bin] = current;
         else link[rear[bin]] = current;
         rear[bin] = current;
      }
      /* find first nonempty queue/bin */
      for (bin = 0; !front[bin]; bin++);
      first = front[bin]; last = rear[bin];

      /* concatenate remaining queues */
      for (bin++; bin < r; bin++)
         if (front[bin])
            {link[last] = front[bin]; last = rear[bin];}
      link[last] = 0;
   }
   return first;
}
```

Program 7.14: LSD radix sort

will yield different computing times. □

Example 7.8: Suppose we are to sort 10 numbers in the range [0, 999]. For this example, we use $r = 10$ (though other choices are possible). Hence, $d = 3$. The input list is linked and has the form given in Figure 7.9(a). The nodes are labeled $R_1, \cdots, R_{10}$. Figure 7.9 shows the queues formed when sorting on each of the digits, as well as the lists after the queues have been collected from the 10 bins. □



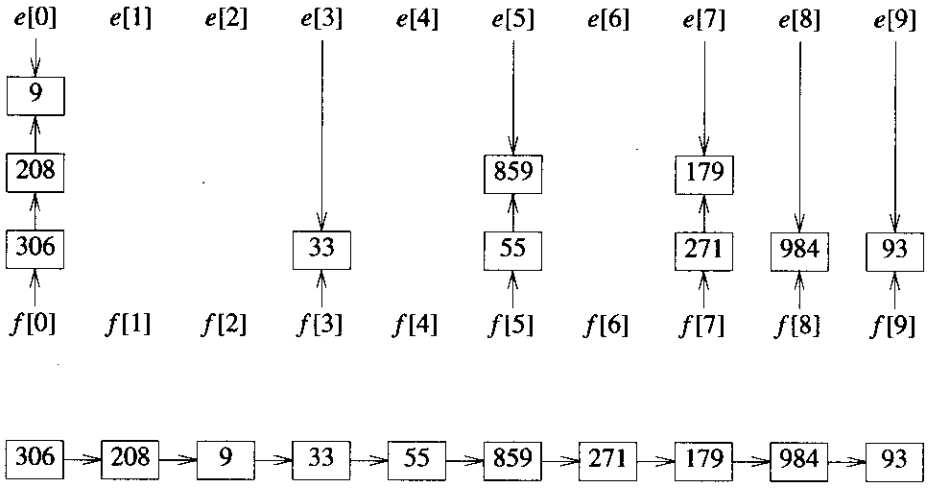
(a) Initial input

(b) First-pass queues and resulting chain

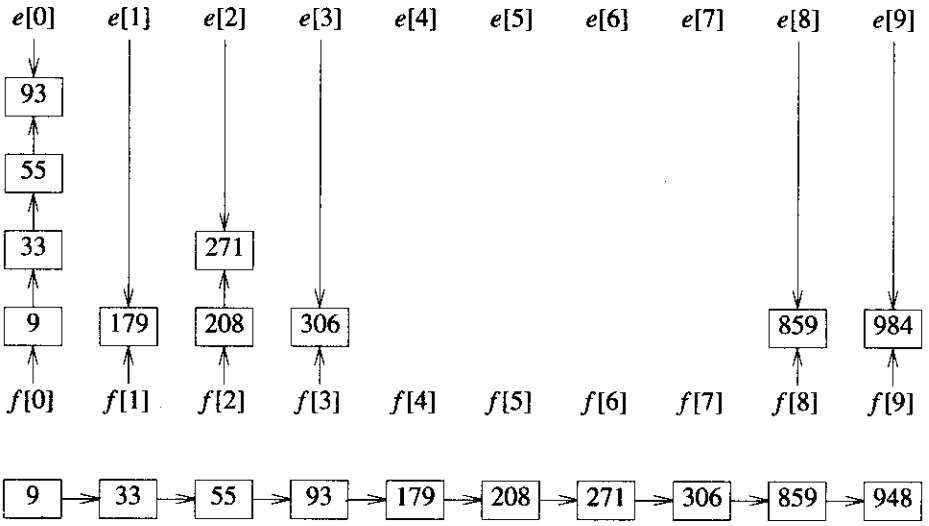
Figure 7.9: Radix sort example (continued on next page)

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each pass of *radixSort* (Program 7.14). Use $r = 10$.
2. Under what conditions would an MSD radix sort be more efficient than an LSD radix sort?
3. Does *radixSort* result in a stable sort when used to sort numbers as in Example 7.8?



(c) Second-pass queues and resulting chain



(d) Third-pass queues and resulting chain

Figure 7.9: Radix sort example

4. Write a sort function to sort records $R_1, \cdots, R_n$ lexically on keys $(K^1, \cdots, K^r)$ for the case when the range of each key is much larger than n. In this case, the bin-sort scheme used in *radixSort* to sort within each key becomes inefficient (why?). What scheme would you use to sort within a key if we desired a function with (a) good worst-case behavior, (b) good average behavior, (c) small n, say <15?
5. If we have n records with integer keys in the range $[0, n^2)$, then they may be sorted in $O(n \log n)$ time using heap sort or merge sort. Radix sort on a single key (i.e., $d = 1$ and $r = n^2$) takes $O(n^2)$ time. Show how to interpret the keys as two subkeys so that radix sort will take only $O(n)$ time to sort n records. (Hint: Each key, K_i, may be written as $K_i = K_i^1 n + K_i^2$ with K_i^1 and K_i^2 integers in the range $[0, n)$.)
6. Generalize the method of the previous exercise to the case of integer keys in the range $(0, n^p)$ obtaining an $O(pn)$ sorting method.
7. Experiment with *radixSort* to see how it performs relative to the comparison-based sort methods discussed in earlier sections.

7.8 LIST AND TABLE SORTS

Apart from radix sort and recursive merge sort, all the sorting methods we have looked at require excessive data movement. That is, as the result of a comparison, records may be physically moved. This tends to slow down the sorting process when records are large. When sorting lists with large records, it is necessary to modify the sorting methods so as to minimize data movement. Methods such as insertion sort and our iterative merge sort can be modified to work with a linked list rather than a sequential list. In this case each record will require an additional link field. Instead of physically moving the record, we change its link field to reflect the change in the position of the record in the list. At the end of the sorting process, the records are linked together in the required order. In many applications (e.g., when we just want to sort lists and then output them record by record on some external media in the sorted order), this is sufficient. However, in some applications it is necessary to physically rearrange the records *in place* so that they are in the required order. Even in such cases, considerable savings can be achieved by first performing a linked-list sort and then physically rearranging the records according to the order specified in the list. This rearranging can be accomplished in linear time using some additional space.

If the list has been sorted so that at the end of the sort, *first* is a pointer to the first record in a linked list of records, then each record in this list will have a key that is greater than or equal to the key of the previous record (if there is a previous record). To physically rearrange these records into the order specified by the list, we begin by interchanging records R_1 and R_{first}. Now, the record in the position R_1 has the smallest key.

If *first* $\neq 1$, then there is some record in the list whose link field is 1. If we could change this link field to indicate the new position of the record previously at position 1, then we would be left with records $R_2, \cdots, R_n$ linked together in nondecreasing order. Repeating the above process will, after $n-1$ iterations, result in the desired rearrangement. The snag, however, is that in a singly linked list we do not know the predecessor of a node. To overcome this difficulty, our first rearrangement function, *listSort*1 (Program 7.15), begins by converting the singly linked list *first* into a doubly linked list and then proceeds to move records into their correct places. This function assumes that links are stored in an integer array as in the case of our radix sort and recursive merge sort functions.

```
void listSort1(element a[], int linka[], int n, int first)
{/* rearrange the sorted chain beginning at first so that
    the records a[1:n] are in sorted order */
   int linkb[MAX_SIZE]; /* array for backward links */
   int i, current, prev = 0;
   element temp;
   for (current = first; current; current = linka[current])
   {/* convert chain into a doubly linked list */
      linkb[current] = prev;
      prev = current;
   }

   for (i = 1; i < n ; i++) /* move a[first] to position i
                               while maintaining the list */
   {
      if (first != i) {
         if (linka[i]) linkb[linka[i]] = first;
         linka[linkb[i]] = first;
         SWAP(a[first], a[i], temp);
         SWAP(linka[first], linka[i], temp);
         SWAP(linkb[first], linkb[i], temp);
      }
      first = linka[i];
   }
}
```

Program 7.15: Rearranging records using a doubly linked list

Example 7.9: After a list sort on the input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) has been made, the list is linked as in Figure 7.10(a) (only the key and link fields of each

record are shown).

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1

(a) Linked list following a list sort, *first* = 4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1
linkb	10	4	5	0	7	2	9	6	1	8

(b) Corresponding doubly linked list, *first* = 4

Figure 7.10: Sorted linked lists

Following the links starting at *first*, we obtain the logical sequence of records $R_4, R_2, R_6, R_8, R_{10}, R_1, R_9, R_7, R_5$, and R_3. This sequence corresponds to the key sequence 1, 5, 11, 15, 19, 26, 48, 59, 61, 33. Filling in the backward links, we get the doubly linked list of Figure 7.10(b). Figure 7.11 shows the list following the first four iterations of the second **for** loop of *listSort*1. The changes made in each iteration are shown in boldface. □

Analysis of *listSort1:* If there are n records in the list, then the time required to convert the chain *first* into a doubly linked list is $O(n)$. The second **for** loop is iterated $n - 1$ times. In each iteration, at most two records are interchanged. This requires three record moves. If each record is m words long, then the cost per record swap is $O(m)$. The total time is therefore $O(nm)$.

The worst case of $3(n - 1)$ record moves (note that each swap requires 3 record moves) is achievable. For example, consider the input key sequence $R_1, R_2, \cdots, R_n$, with $R_2 < R_3 < \cdots < R_n$ and $R_1 > R_n$. □

Although several modifications to *list*1 are possible, one of particular interest was given by M. D. MacLaren. This modification results in a rearrangement function, *listSort*2, in which no additional link fields are necessary. In this function (Program 7.16), after the record R_{first} is swapped with R_i, the link field of the new R_i is set to *first* to indicate that the original record was moved. This, together with the observation that *first* must always be $\geq i$, permits a correct reordering of the records.

i	$\mathbf{R_1}$	R_2	R_3	$\mathbf{R_4}$	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
linka	**2**	6	0	**9**	3	8	5	10	7	4
linkb	**0**	4	5	**10**	7	2	9	6	4	8

(a) Configuration after first iteration of the **for** loop of *listSort*1, *first* = 2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_9	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

(b) Configuration after second iteration, *first* = 6

i	R_1	R_2	$\mathbf{R_3}$	R_4	R_5	$\mathbf{R_6}$	R_7	R_8	R_9	R_{10}
key	1	5	**11**	26	61	**77**	59	15	48	19
linka	2	6	**8**	9	6	**0**	5	10	7	4
linkb	0	4	**2**	10	7	**5**	9	6	4	8

(c) Configuration after third iteration, *first* = 8

i	R_1	R_2	R_3	$\mathbf{R_4}$	R_5	R_6	R_7	$\mathbf{R_8}$	R_9	R_{10}
key	1	5	11	**15**	61	77	59	**26**	48	19
linka	2	6	8	**10**	6	0	5	**9**	7	8
linkb	0	4	2	**6**	7	5	9	**10**	8	8

(d) Configuration after fourth iteration, *first* = 10

Figure 7.11: Example for *listSort*1 (Program 7.15)

Example 7.10: The data are the same as in Example 7.9. After the list sort we have the configuration of Figure 7.10(a). The configuration after each of the first five iterations of the **for** loop of *listSort*2 is shown in Figure 7.12. □

Analysis of *listSort2*: The sequence of record moves for *listSort*2 is identical to that for *listSort*1. Hence, in the worst case $3(n-1)$ record moves for a total cost of $O(nm)$ are

```
void listSort2(element a[], int link[], int n, int first)
{/* same function as list1 except that a second link array
    linkb is not required. */
{
   int i;
   element temp;
   for (i = 1; i < n; i++)
   {/* find correct record for ith position,  its index is
       ≥ i as records in positions 1, 2, ..., i - 1 are
       already correctly positioned */
      while (first < i) first = link[first];
      int q = link[first]; /* a[q] is next in sorted order */
      if (first != i)
      {/* a[first] has ith smallest key, swap with a[i] and
          set link from old position of a[i] to new one */
         SWAP(a[i], a[first], temp);
         link[first] = link[i];
         link[i] = first;
      }
      first = q;
   }
}
```

Program 7.16: Rearranging records using only one link field

made. No node is examined more than once in the **while** loop. So, the total time for the **while** loop is $O(n)$. □

Although the asymptotic computing time for both *listSort*1 and *listSort*2 is the same, and the same number of record moves is made in either case, we expect *listSort*2 to be slightly faster than *listSort*1 because each time two records are swapped, *listSort*1 does more work than *listSort*2 does. *listSort*1 is inferior to *listSort*2 in both space and time considerations.

The list sort technique is not well suited for quick sort and heap sort. The sequential representation of the heap is essential to heap sort. For these sort methods, as well as for methods suited to list sort, one can maintain an auxiliary table, t, with one entry per record. The entries in this table serve as an indirect reference to the records.

At the start of the sort, $t[i] = i$, $1 \le i \le n$. If the sorting function requires a swap of $a[i]$ and $a[j]$, then only the table entries (i.e., $t[i]$ and $t[j]$) need to be swapped. At the end of the sort, the record with the smallest key is $a[t[1]]$ and that with the largest

i	**R_1**	R_2	R_3	**R_4**	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
link	**4**	6	0	**9**	3	8	5	10	7	1

(a) Configuration after first iteration of the **for** loop of *listSort*2, *first* = 2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

(b) Configuration after second iteration, *first* = 6

i	R_1	R_2	**R_3**	R_4	R_5	**R_6**	R_7	R_8	R_9	R_{10}
key	**1**	5	**11**	26	61	**77**	59	15	48	19
link	4	6	**6**	9	3	**0**	5	10	7	1

(c) Configuration after third iteration, *first* = 8

i	R_1	R_2	R_3	**R_4**	R_5	R_6	R_7	**R_8**	R_9	R_{10}
key	**1**	5	11	**15**	61	77	59	**26**	48	19
link	4	6	6	**8**	3	0	5	**9**	7	1

(d) Configuration after fourth iteration, *first* = 10

i	R_1	R_2	R_3	R_4	**R_5**	R_6	R_7	R_8	R_9	**R_{10}**
key	1	5	11	15	**19**	77	59	26	48	**61**
link	4	6	6	8	**10**	0	5	9	7	**3**

(e) Configuration after fifth iteration, *first* = 1

Figure 7.12: Example for *listSort*2 (Program 7.16)

$a[t[n]]$. The required permutation on the records is $a[t[1]], a[t[2]], \cdots, a[t[n]]$ (see Figure 7.13). This table is adequate even in situations such as binary search, where a sequentially ordered list is needed. In other situations, it may be necessary to physically rearrange the records according to the permutation specified by t.

The function to rearrange records corresponding to the permutation $t[1], t[2], \cdots, t[n]$ is a rather interesting application of a theorem from mathematics: Every

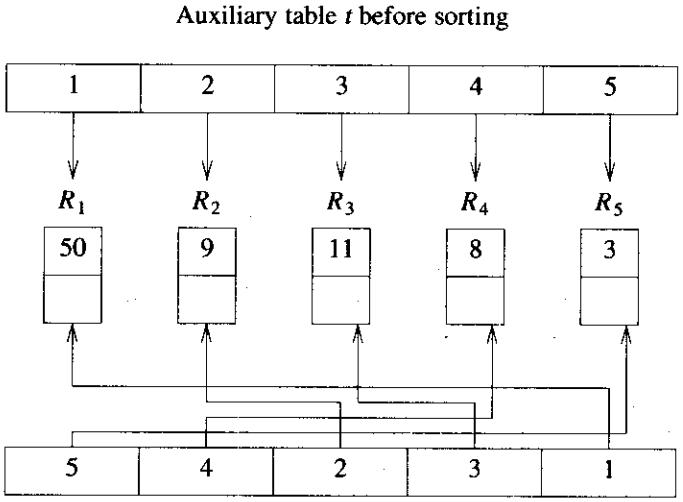


Figure 7.13: Table sort

permutation is made up of disjoint cycles. The cycle for any element i is made up of i, $t[i]$, $t^2[i]$, $\cdots$, $t^k[i]$, where $t^j[i] = t[t^{j-1}[i]]$, $t^0[i] = i$, and $t^k[i] = i$. Thus, the permutation t of Figure 7.13 has two cycles, the first involving R_1 and R_5 and the second involving R_4, R_3, and R_2. Function *table* (Program 7.17) utilizes this cyclic decomposition of a permutation. First, the cycle containing R_1 is followed, and all records are moved to their correct positions. The cycle containing R_2 is the next one examined unless this cycle has already been examined. The cycles for R_3, R_4, $\cdots$, R_{n-1} are followed in this order. The result is a physically sorted list.

When processing a trivial cycle for R_i (i.e., $t[i] = i$), no rearrangement involving record R_i is required, since the condition $t[i] = i$ means that the record with the ith smallest key is R_i. In processing a nontrivial cycle for record R_i (i.e., $t[i] \neq i$), R_i is moved to a temporary position *temp*, then the record at $t[i]$ is moved to i; next the record at $t[t[i]]$ is moved to $t[i]$, and so on until the end of the cycle $t^k[i]$ is reached and the record at *temp* is moved to $t^{k-1}[i]$.

Example 7.11: Suppose we start with the table t of Figure 7.14(a). This figure also shows the record keys. The table configuration is that following a Table Sort. There are two nontrivial cycles in the permutation specified by t. The first is R_1, R_3, R_8, R_6, R_1. The second is R_4, R_5, R_7, R_4. During the first iteration ($i = 1$) of the **for** loop of *tableSort* (Program 7.17), the cycle R_1, $R_{t[1]}$, $R_{t^2[1]}$, $R_{t^3[1]}$, R_1 is followed. Record R_1 is moved to a temporary spot *temp*; $R_{t[1]}$ (i.e., R_3) is moved to the position R_1; $R_{t^2[1]}$ (i.e.,

```
void tableSort(element a[], int n, int t[])
{/* rearrange a[1:n] to correspond to the sequence
    a[t[1]], ... , a[t[n]] */
   int i, current, next;
   element temp;
   for (i = 1; i < n; i++)
      if (t[i] != i)  {/* nontrivial cycle starting at i */
         temp = a[i]; current = i;
         do {
            next = t[current]; a[current] = a[next];
            t[current] = current; current = next;
         } while (t[current] != i);
         a[current] = temp;
         t[current] = current;
      }
}
```

Program 7.17: Table sort

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

(a) Initial configuration

key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) Configuration after rearrangement of first cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

(c) Configuration after rearrangement of second cycle

Figure 7.14: Table sort example

R_8) is moved to R_3; R_6 is moved to R_8; and finally, the record in *temp* is moved to R_6. Thus, at the end of the first iteration we have the table configuration of Figure 7.14(b).

For $i = 2$ or 3, $t[i] = i$, indicating that these records are already in their correct positions. When $i = 4$, the next nontrivial cycle is discovered, and the records on this cycle (R_4, R_5, R_7, R_4) are moved to their correct positions. Following this we have the table configuration of Figure 7.14(c).

For the remaining values of i (i = 5, 6, and 7), $t[i] = i$, and no more nontrivial cycles are found. □

Analysis of *tableSort*: If each record uses m words of storage, then the additional space needed is m words for *temp* plus a few more for variables such as i, *current*, and *next*. For the computing time, we observe that the **for** loop is executed $n - 1$ times. If for some value of i, $t[i] \neq i$, then there is a nontrivial cycle including $k > 1$ distinct records $R_i, R_{t[i]}, \cdots, R_{t^{k-1}[i]}$. Rearranging these records requires $k + 1$ record moves. Following this, the records involved in this cycle are not moved again at any time in the algorithm, as $t[j] = j$ for all such records R_j. Hence, no record can be in two different nontrivial cycles. Let k_l be the number of records on a nontrivial cycle starting at R_l when $i = l$ in the **for** loop. Let $k_l = 0$ for a trivial cycle. The total number of record moves is

$$\sum_{l=0, k_l \neq 0}^{n-1} (k_l + 1)$$

Since the records on nontrivial cycles must be different, $\sum k_l \leq n$. The total number of record moves is maximum when $\sum k_l = n$ and there are $\lfloor n/2 \rfloor$ cycles. When n is even, each cycle contains two records. Otherwise, one cycle contains three and the others two each. In either case the number of record moves is $\lfloor 3n/2 \rfloor$. One record move costs $O(m)$ time. The total computing time is therefore $O(mn)$. □

Comparing *listSort2* (Program 7.16) and *tableSort*, we see that in the worst case, *listSort2* makes $3(n-1)$ record moves, whereas *tableSort* makes only $\lfloor 3n/2 \rfloor$ record moves. For larger values of m it is worthwhile to make one pass over the sorted list of records, creating a table t corresponding to a table sort. This would take $O(n)$ time. Then *tableSort* could be used to rearrange the records in the order specified by t.

EXERCISES

1. Complete Example 7.9.
2. Complete Example 7.10.
3. Write a version of selection sort (see Chapter 1) that works on a chain of records.
4. Write a table sort version of quick sort. Now during the sort, records are not physically moved. Instead, $t[i]$ is the index of the record that would have been in position i if records were physically moved around as in *quickSort* (Program 7.6). Begin with $t[i] = i$, $1 \leq i \leq n$. At the end of the sort, $t[i]$ is the index of the record

that should be in the ith position in the sorted list. So now function *table* may be used to rearrange the records into the sorted order specified by t. Note that this reduces the amount of data movement taking place when compared to *quickSort* for the case of large records.

5. Do Exercise 4 for the case of insertion sort.
6. Do Exercise 4 for the case of merge sort.
7. Do Exercise 4 for the case of heap sort.

7.9 SUMMARY OF INTERNAL SORTING

Of the several sorting methods we have studied, no one method is best under all circumstances. Some methods are good for small n, others for large n. Insertion sort is good when the list is already partially ordered. Because of the low overhead of the method, it is also the best sorting method for "small" n. Merge sort has the best worst-case behavior but requires more storage than heap sort. Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$. The behavior of radix sort depends on the size of the keys and the choice of r. Figure 7.15 summarizes the asymptotic complexity of the first four of these sort methods.

Method	Worst	Average
Insertion sort	n^2	n^2
Heap sort	$n\log n$	$n\log n$
Merge sort	$n\log n$	$n\log n$
Quick sort	n^2	$n\log n$

Figure 7.15: Comparison of sort methods

Figures 7.16 and 7.17 give the average runtimes for the four sort methods of Figure 7.15. These times were obtained on a 1.7GHz Intel Pentium 4 PC with 512 MB RAM and Microsoft Visual Studio .NET 2003. For each n at least 100 randomly generated integer instances were run. These random instances were constructed by making repeated calls to the C function *rand*. If the time taken to sort these instances was less than 1 second then additional random instances were sorted until the total time taken was at least this much. The times reported in Figure 7.16 include the time taken to set up the random data. For each n the time taken to set up the data and the time for the remaining overheads included in the reported numbers is the same for all sort methods. As a result, the data of Figure 7.16 are useful for comparative purposes.

n	*Insert*	*Heap*	*Merge*	*Quick*
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Times are in milliseconds

Figure 7.16: Average times for sort methods

As Figure 7.18 shows, quick sort outperforms the other sort methods for suitably large n. We see that the break-even point between insertion and quick sort is between 50 and 100. The exact break-even point can be found experimentally by obtaining run-time data for n between 50 and 100. Let the exact break-even point be *nBreak*. For average performance, insertion sort is the best sort method (of those tested) to use when $n <$ *nBreak*, and quick sort is the best when $n >$ *nBreak*. We can improve on the performance of quick sort for $n >$ *nBreak* by combining insertion and quick sort into a single sort function by replacing the following statement in Program 7.6

```
if (left < right) {code to partition and make recursive calls}
```

with the code

```
if (left+nBreak < right) {
    code to partition and make recursive calls
}
else {
       sort a [left:right] using insertion sort;
       return;
}
```

For worst-case behavior most implementations will show merge sort to be best for $n > c$ where c is some constant. For $n \leq c$ insertion sort has the best worst-case behavior.

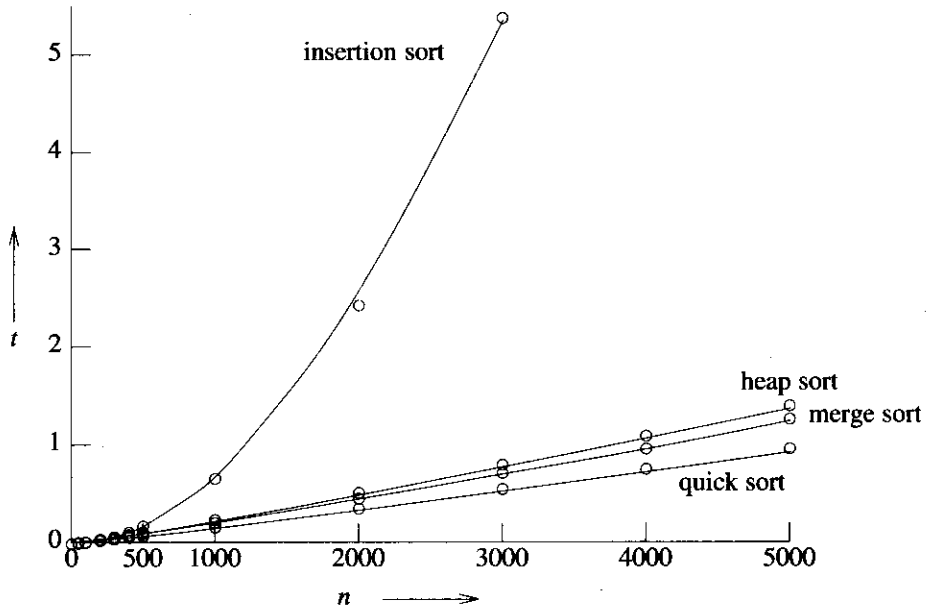


Figure 7.18: Plot of average times (milliseconds)

The performance of merge sort can be improved by combining insertion sort and merge sort in a manner similar to that described above for combining insertion sort and quick sort.

The run-time results for the sort methods point out some of the limitations of asymptotic complexity analysis. Asymptotic analysis is not a good predictor of performance for small instances—insertion sort with its $O(n^2)$ complexity is better than all of the $O(n \log n)$ methods for small instances. Programs that have the same asymptotic complexity often have different actual runtimes.

EXERCISES

1. [***Count Sort***] The simplest known sorting method arises from the observation that the position of a record in a sorted list depends on the number of records with smaller keys. Associated with each record there is a *count* field used to determine the number of records that must precede this one in the sorted list. Write a function to determine the count of each record in an unordered list. Show that if the list has n records, then all the counts can be determined by making at most

$n(n-1)/2$ key comparisons.

2. Write a function similar to *table* (Program 7.17) to rearrange the records of a list if, with each record, we have a count of the number of records preceding it in the sorted list (see Exercise 1).
3. Obtain Figures 7.16 and 7.18 for the worst-case runtime.
4. [***Programming Project***] The objective of this assignment is to come up with a composite sorting function that is good on the worst-time criterion. The candidate sort methods are (a) Insertion sort, (b) Quick sort, (c) Merge sort, (d) Heap sort.

   To begin with, program these sort methods in C. In each case, assume that n integers are to be sorted. In the case of quick sort, use the median-of-three method. In the case of merge sort, use the iterative method (as a separate exercise, you might wish to compare the runtimes of the iterative and recursive versions of merge sort and determine what the recursion penalty is in your favorite language using your favorite compiler). Check out the correctness of the programs using some test data. Since quite detailed and working functions are given in the book, this part of the assignment should take little effort. In any case, no points are earned until after this step.

   To obtain reasonably accurate runtimes, you need to know the accuracy of the clock or timer you are using. Determine this by reading the appropriate manual. Let the clock accuracy be δ. Now, run a pilot test to determine ballpark times for your four sorting functions for n = 500, 1000, 2000, 3000, 4000, and 5000. You will notice times of 0 for many of these values of n. The other times may not be much larger than the clock accuracy.

   To time an event that is smaller than or near the clock accuracy, repeat it many times and divide the overall time by the number of repetitions. You should obtain times that are accurate to within 1%.

   We need worst-case data for each of the four sort methods. The worst-case data for insertion sort are easy to generate. Just use the sequence n, $n-1$, $n-2$, $\cdots$, 1. Worst-case data for merge sort can be obtained by working backward. Begin with the last merge your function will perform and make this work hardest. Then look at the second-to-last merge, and so on. Use this logic to obtain a program that will generate worst-case data for merge sort for each of the above values of n.

   Generating worst-case data for heap sort is the hardest, so, here we shall use a random permutation generator (one is provided in Program 7.18). We shall generate random permutations of the desired size, clock heap sort on each of these, and use the max of these times to approximate to the worst-case time. You will be able to use more random permutations for smaller values of n than for larger. For no value of n should you use fewer than 10 permutations. Use the same technique to obtain worst-case times for quick sort.

   Having settled on the test data, we are ready to perform our experiment.

```
void permute(element a[], int n)
{/* random permutation generator */
   int i, j;
   element temp;
   for (i = n; i >= 2; i--)
   {
      j = rand() % i + 1;
         /* j = random integer in the range [1, i] */
      SWAP(a[j], a[i], temp);
   }
}
```

Program 7.18: Random permutation generator

Obtain the worst-case times. From these times you will get a rough idea when one function performs better than the other. Now, narrow the scope of your experiments and determine the exact value of n when one sort method outperforms another. For some methods, this value may be 0. For instance, each of the other three methods may be faster than quick sort for all values of n.

Plot your findings on a single sheet of graph paper. Do you see the n^2 behavior of insertion sort and quick sort and the $n\log n$ behavior of the other two methods for suitably large n (about $n > 20$)? If not, there is something wrong with your test or your clock or with both. For each value of n determine the sort function that is fastest (simply look at your graph). Write a composite function with the best possible performance for all n. Clock this function and plot the times on the same graph sheet you used earlier.

WHAT TO TURN IN

You are required to submit a report that states the clock accuracy, the number of random permutations tried for heap sort, the worst-case data for merge sort and how you generated it, a table of times for the above values of n, the times for the narrowed ranges, the graph, and a table of times for the composite function. In addition, your report must be accompanied by a complete listing of the program used by you (this includes the sorting functions and the main program for timing and test-data generation).

5. Repeat the previous exercise for the case of average runtimes. Average-case data are usually very difficult to create, so use random permutations. This time, however, do not repeat a permutation many times to overcome clock inaccuracies. Instead, use each permutation once and clock the overall time (for a fixed n).

6. Assume you are given a list of five-letter English words and are faced with the problem of listing these words in sequences such that the words in each sequence are anagrams (i.e., if x and y are in the same sequence, then word x is a permutation of word y). You are required to list out the fewest such sequences. With this restriction, show that no word can appear in more than one sequence. How would you go about solving this problem?
7. Assume you are working in the census department of a small town where the number of records, about 3000, is small enough to fit into the internal memory of a computer. All the people currently living in this town were born in the United States. There is one record for each person in this town. Each record contains (a) the state in which the person was born, (b) county of birth, and (c) name of person. How would you produce a list of all persons living in this town? The list is to be ordered by state. Within each state the persons are to be listed by their counties, the counties being arranged in alphabetical order. Within each county, the names are also listed in alphabetical order. Justify any assumptions you make.
8. [***Bubble Sort***] In a bubble sort several left-to-right passes are made over the array of records to be sorted. In each pass, pairs of adjacent records are compared and exchanged if necessary. The sort terminates following a pass in which no records are exchanged.
   (a) Write a C function for bubble sort.
   (b) What is the worst-case complexity of your function?
   (c) How much time does your function take on a sorted array of records?
   (d) How much time does your function take on an array of records that are in the reverse of sorted order?
9. Redo the preceding exercise beginning with an unsorted chain of records and ending with a sorted chain.
10. [***Programming Project***] The objective of this exercise is to study the effect of the size of an array element on the computational time of various sorting algorithms.
    (a) Use insertion sort, quick sort, iterative merge sort, and heap sort to sort arrays of (*i*) characters (**char**), (*ii*) integers (**int**), (*iii*) floating point numbers (**float**), and (*iv*) rectangles (Assume that a rectangle is represented by the coordinates of its bottom left point and its height and width, all of which are of type **float**. Assume, also, that rectangles are to be sorted in nondecreasing order of their areas.)
    (b) Obtain a set of runtimes for each algorithm-data type pair specified above. (There should be sixteen such pairs.) To obtain a set of runtimes of an algorithm-data type pair, you should run the algorithm on at least four arrays of different sizes containing elements of the appropriate data type. The elements in an array should be generated using a random number generator

(c) Draw tables and graphs displaying your experimental results. What do you conclude from the experiments?

7.10 EXTERNAL SORTING

7.10.1 Introduction

In this section, we assume that the lists to be sorted are so large that an entire list cannot be contained in the internal memory of a computer, making an internal sort impossible. We shall assume that the list (or file) to be sorted resides on a disk. The term *block* refers to the unit of data that is read from or written to a disk at one time. A block generally consists of several records. For a disk, there are three factors contributing to the read/write time:

(1) *Seek time:* time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

(2) *Latency time:* time until the right sector of the track is under the read/write head.

(3) *Transmission time:* time to transmit the block of data to/from the disk.

The most popular method for sorting on external storage devices is merge sort. This method consists of two distinct phases. First, segments of the input list are sorted using a good internal sort method. These sorted segments, known as *runs*, are written onto external storage as they are generated. Second, the runs generated in phase one are merged together following the merge-tree pattern of Figure 7.4, until only one run is left. Because the simple merge function *merge* (Program 7.7) requires only the leading records of the two runs being merged to be present in memory at one time, it is possible to merge large runs together. It is more difficult to adapt the other internal sort methods considered in this chapter to external sorting.

Example 7.12: A list containing 4500 records is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input list is maintained on disk and has a block length of 250 records. We have available another disk that may be used as a scratch pad. The input disk is not to be written on. One way to accomplish the sort using the general function outlined above is to

(1) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R_1 to R_6. A method such as heap sort, merge sort, or quick sort could be used. These six runs are written onto the scratch disk (Figure 7.19).

(2) Set aside three blocks of internal memory, each capable of holding 250 records. Two of these blocks will be used as input buffers and the third as an output buffer. Merge runs R_1 and R_2. This merge is carried out by first reading one block of each of these runs into input buffers. Blocks of runs are merged from the input buffers

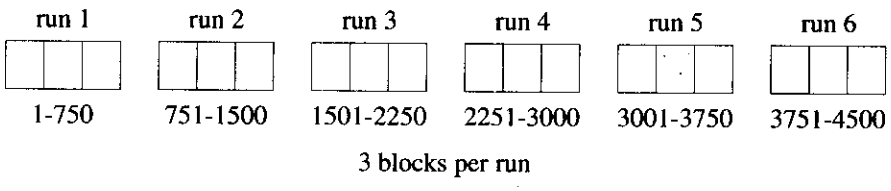


Figure 7.19: Blocked runs obtained after internal sorting

into the output buffer. When the output buffer gets full, it is written onto the disk. If an input buffer gets empty, it is refilled with another block from the same run. After runs R_1 and R_2 are merged, R_3 and R_4 and finally R_5 and R_6 are merged. The result of this pass is three runs, each containing 1500 sorted records or six blocks. Two of these runs are now merged using the input/output buffers set up as above to obtain a run of size 3000. Finally, this run is merged with the remaining run of size 1500 to obtain the desired sorted list (Figure 7.20). □

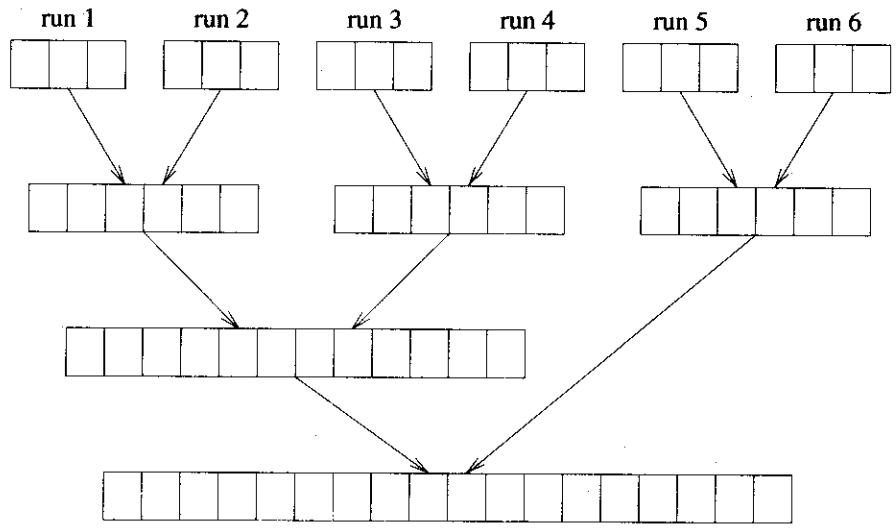


Figure 7.20: Merging the six runs

To analyze the complexity of external sort, we use the following notation:

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

t_{IO} = time to input or output one block

$= t_s + t_l + t_{rw}$

t_{IS} = time to internally sort 750 records

nt_m = time to merge n records from input buffers to the output buffer

We shall assume that each time a block is read from or written onto the disk, the maximum seek and latency times are experienced. Although this is not true in general, it will simplify the analysis. The computing times for the various operations in our 4500-record example are given in Figure 7.21.

	operation	time
(1)	read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2)	merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(3)	merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4)	merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
	total time	$132t_{IO} + 12{,}000t_m + 6t_{IS}$

Figure 7.21: Computing times for disk sort example

The contribution of seek time can be reduced by writing blocks on the same cylinder or on adjacent cylinders. A close look at the final computing time indicates that it depends chiefly on the number of passes made over the data. In addition to the initial input pass made over the data for the internal sort, the merging of the runs requires 2-2/3 passes over the data (one pass to merge 6 runs of length 750 records, two-thirds of a pass to merge two runs of length 1500, and one pass to merge one run of length 3000 and one of length 1500). Since one full pass covers 18 blocks, the input and output time is $2 \times (2\text{-}2/3 + 1) \times 18\, t_{IO} = 132t_{IO}$. The leading factor of 2 appears because each record

that is read is also written out again. The merge time is $2\text{-}2/3 \times 4500t_m = 12{,}000t_m$. Because of this close relationship between the overall computing time and the number of passes made over the data, future analysis will be concerned mainly with counting the number of passes being made. Another point to note regarding the above sort is that no attempt was made to use the computer's ability to carry out input/output and CPU operation in parallel and thus overlap some of the time. In the ideal situation we would overlap almost all the input/output time with CPU processing so that the real time would be approximately $132\, t_{IO} \approx 12{,}000\, t_m + 6t_{IS}$.

If we have two disks, we can write on one, read from the other, and merge buffer loads already in memory in parallel. A proper choice of buffer lengths and buffer handling schemes will result in a time of almost $66t_{IO}$. This parallelism is an important consideration when sorting is being carried out in a nonmultiprogramming environment. In this situation, unless input/output and CPU processing is going on in parallel, the CPU is idle during input/output. In a multiprogramming environment, however, the need for the sorting program to carry out input/output and CPU processing in parallel may not be so critical, since the CPU can be busy working on another program (if there are other programs in the system at the time) while the sort program waits for the completion of its input/output. Indeed, in many multiprogramming environments it may not even be possible to achieve parallel input, output, and internal computing because of the structure of the operating system.

The number of merge passes over the runs can be reduced by using a higher-order merge than two-way merge. To provide for parallel input, output, and merging, we need an appropriate buffer-handling scheme. Further improvement in runtime can be obtained by generating fewer (or equivalently longer) runs than are generated by the strategy described above. This can be done using a loser tree. The loser-tree strategy to be discussed in Section 7.10.4 results in runs that are on the average almost twice as long as those obtained by the above strategy. However, the generated runs are of varying size. As a result, the order in which the runs are merged affects the time required to merge all runs into one. We consider these factors now.

7.10.2 *k*-Way Merging

The two-way merge function *merge* (Program 7.7) is almost identical to the merge function just described (Figure 7.20). In general, if we start with m runs, the merge tree corresponding to Figure 7.20 will have $\lceil \log_2 m \rceil + 1$ levels, for a total of $\lceil \log_2 m \rceil$ passes over the data list. The number of passes over the data can be reduced by using a higher-order merge (i.e., k-way merge for $k \geq 2$). In this case, we would simultaneously merge k runs together. Figure 7.22 illustrates a four-way merge of 16 runs. The number of passes over the data is now two, versus four passes in the case of a two-way merge. In general, a k-way merge on m runs requires $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher-order merge.

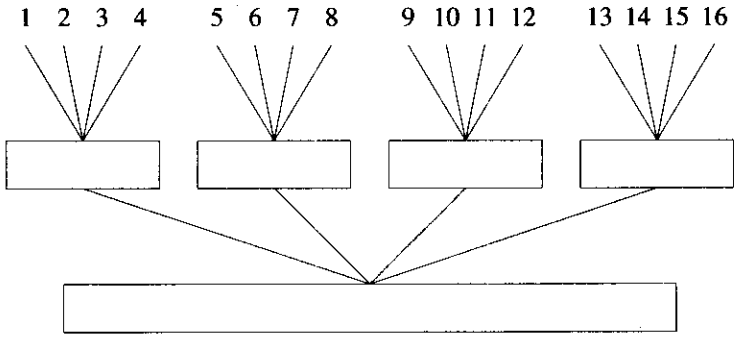


Figure 7.22: A four-way merge on 16 runs

The use of a higher-order merge, however, has some other effects on the sort. To begin with, k runs of size $s_1, s_2, s_3, \cdots, s_k$ can no longer be merged internally in $O(\sum_1^k s_i)$ time. In a k-way merge, as in a two-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make $k-1$ comparisons to determine the next record to output. The computing time for this is $O((k-1)\sum_1^k s_i)$. Since $\log_k m$ passes are being made, the total number of key comparisons is $n(k-1)\log_k m = n(k-1)\log_2 m/\log_2 k$, where n is the number of records in the list. Hence, $(k-1)/\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be outweighed by the resulting increase in CPU time needed to perform the k-way merge.

For large k (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using a loser tree with k leaves (see Chapter 5). In this case, the total time needed per level of the merge tree is $O(n \log_2 k)$. Since the number of levels in this tree is $O(\log_k m)$, the asymptotic internal processing time becomes $O(n \log_2 k \log_k m) = O(n \log_2 m)$. This is independent of k.

In going to a higher-order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes. This is so because the number of input buffers needed to carry out a k-way merge increases with k. Although $k+1$ buffers are sufficient, in the next section we shall see that the use of

$2k + 2$ buffers is more desirable. Since the internal memory available is fixed and independent of k, the buffer size must be reduced as k increases. This in turn implies a reduction in the block size on disk. With the reduced block size, each pass over the data results in a greater number of blocks being written or read. This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain k value the input/output time will increase despite the decrease in the number of passes being made. The optimal value for k depends on disk parameters and the amount of internal memory available for buffers.

7.10.3 Buffer Handling for Parallel Operation

If k runs are being merged together by a k-way merge, then we clearly need at least k input buffers and one output buffer to carry out the merge. This, however, is not enough if input, output, and internal merging are to be carried out in parallel. For instance, while the output buffer is being written out, internal merging has to be halted, since there is no place to collect the merged records. This can be overcome through the use of two output buffers. While one is being written out, records are merged into the second. If buffer sizes are chosen correctly, then the time to output one buffer will be the same as the CPU time needed to fill the second buffer. With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have $2k$ input buffers. These $2k$ input buffers have to be used cleverly to avoid reaching a situation in which processing has to be held up because of a lack of input records from any one run. Simply assigning two buffers per run does not solve the problem.

Example 7.13: Assume that a two-way merge is carried out using four input buffers, $in[i]$, $0 \le i \le 3$, and two output buffers, $ou[0]$ and $ou[1]$. Each buffer is capable of holding two records. The first few records of run 0 have key value 1, 3, 5, 7, 8, 9. The first few records of run 1 have key value 2, 4, 6, 15, 20, 25. Buffers $in[0]$ and $in[2]$ are assigned to run 0. The remaining two input buffers are assigned to run 1. We start the merge by reading in one buffer load from each of the two runs. At this time the buffers have the configuration of Figure 7.23(a). Now runs 0 and 1 are merged using records from $in[0]$ and $in[1]$. In parallel with this, the next buffer load from run 0 is input. If we assume that buffer lengths have been chosen such that the times to input, output, and generate an output buffer are all the same, then when $ou[0]$ is full, we have the situation of Figure 7.23(b). Next, we simultaneously output $ou[0]$, input into $in[3]$ from run 1, and merge into $ou[1]$. When $ou[1]$ is full, we have the situation of Figure 7.23(c). Continuing in this way, we reach the configuration of Figure 7.23(e). We now begin to output $ou[1]$, input from run 0 into $in[2]$, and merge into $ou[0]$. During the merge, all records from run 0 get used before $ou[0]$ gets full. Merging must now be delayed until

the inputting of another buffer load from run 0 is completed. □

Example 7.13 makes it clear that if $2k$ input buffers are to suffice, then we cannot assign two buffers per run. Instead, the buffer must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In the buffer assignment strategy we shall describe, there will at any time be at least one input buffer containing records from each run. The remaining buffers will be filled on a priority basis (i.e., the run for which the k-way merging algorithm will run out of records first is the one from which the next buffer will be filled). One may easily predict which run's records will be exhausted first by simply comparing the keys of the last record read from each of the k runs. The smallest such key determines this run. We shall assume that in the case of equal keys, the merge process first merges the record from the run with least index. This means that if the key of the last record read from run i is equal to the key of the last record read from run j, and $i < j$, then the records read from i will be exhausted before those from j. So, it is possible to have more than two bufferloads from a given run and only one partially full buffer from another run. All bufferloads from the same run are queued together. Before formally presenting the algorithm for buffer utilization, we make the following assumptions about the parallel processing capabilities of the computer system available:

(1) We have two disk drives and the input/output channel is such that we can simultaneously read from one disk and write onto the other.

(2) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate, only one output buffer would be needed. By the time the first record for the new output block is determined, the first record of the previous output block has been written out.

(3) To simplify the discussion we assume that input and output buffers are of the same size.

Keeping these assumptions in mind, we provide a high-level description of the algorithm obtained using the strategy outlined earlier and then illustrate how it works through an example. Our algorithm, Program 7.19, merges k-runs, $k \geq 2$, using a k-way merge. $2k$ input buffers and two output buffers are used. Each buffer is a continuous block of memory. Input buffers are queued in k queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are placed on a linked stack. The algorithm also assumes that the end of each run has a sentinel record with a very large key, say $+\infty$. It is assumed that all other records have key value less than that of the sentinel record. If block lengths, and hence buffer lengths, are chosen such that the time to merge one output buffer load equals the time to read a block, then almost all input, output, and computation will be carried out in

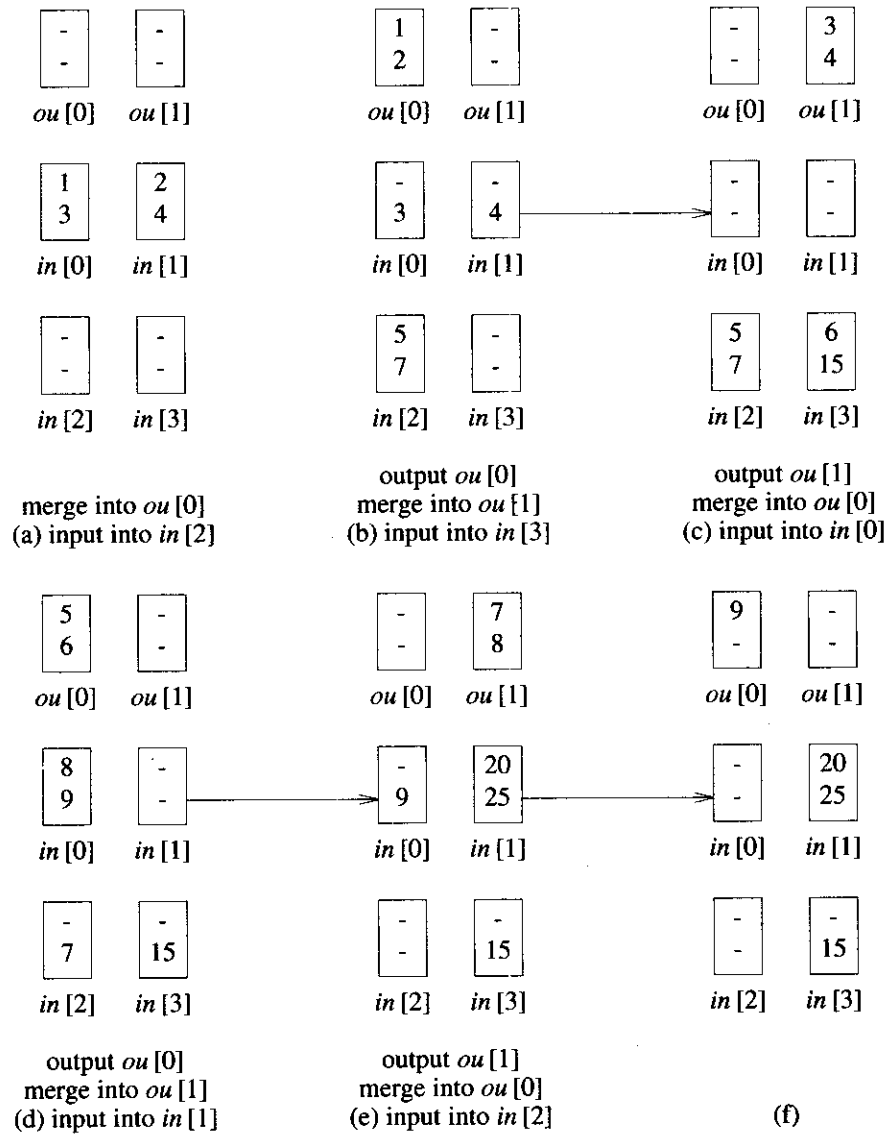


Figure 7.23: Example showing that two fixed buffers per run are not enough for continued parallel operation

parallel. It is also assumed that in the case of equal keys, the k-way merge algorithm first outputs the record from the run with the smallest index.

{Steps in buffering algorithm}

Step 1: Input the first block of each of the k runs, setting up k linked queues, each having one block of data. Put the remaining k input blocks into a linked stack of free input blocks. Set *ou* to 0.

Step 2: Let *lastKey* [i] be the last key input from run i. Let *nextRun* be the run for which *lastKey* is minimum. If *lastKey* [*nextRun*] $\neq +\infty$, then initiate the input of the next block from run *nextRun*.

Step 3: Use a function *kWayMerge* to merge records from the k input queues into the output buffer *ou*. Merging continues until either the output buffer gets full or a record with key $+\infty$ is merged into *ou*. If, during this merge, an input buffer becomes empty before the output buffer gets full or before $+\infty$ is merged into *ou*, the *kWayMerge* advances to the next buffer on the same queue and returns the empty buffer to the stack of empty buffers. However, if an input buffer becomes empty at the same time as the output buffer gets full or $+\infty$ is merged into *ou*, the empty buffer is left on the queue, and *kWayMerge* does not advance to the next buffer on the queue. Rather, the merge terminates.

Step 4: Wait for any ongoing disk input/output to complete.

Step 5: If an input buffer has been read, add it to the queue for the appropriate run. Determine the next run to read from by determining *NextRun* such that *lastKey* [*nextRun*] is minimum.

Step 6: If *lastKey* [*nextRun*] $\neq +\infty$, then initiate reading the next block from run *nextRun* into a free input buffer.

Step 7: Initiate the writing of output buffer *ou*. Set *ou* to $1 - ou$.

Step 8: If a record with key $+\infty$ has been not been merged into the output buffer, go back to Step 3. Otherwise, wait for the ongoing write to complete and then terminate.

Program 7.19: k-way merge with floating buffers

We make the following observations about Program 7.19:

(1) For large k, determination of the queue that will be exhausted first can be found in $\log_2 k$ comparisons by setting up a loser tree for *last* [i], $0 \leq i < k$, rather than making $k - 1$ comparisons each time a buffer load is to be read in. The change in computing time will not be significant, since this queue selection represents only a

very small fraction of the total time taken by the algorithm.

(2) For large k, function *kWayMerge* uses a tree of losers (see Chapter 5).

(3) All input and output except for the input of the initial k blocks and the output of the last block is done concurrently with computing. Since, after k runs have been merged, we would probably begin to merge another set of k runs, the input for the next set can commence during the final merge stages of the present set of runs. That is, when *lastKey* [*nextRun*] = $+\infty$ in Step 6, we begin reading one by one the first blocks from each of the next set of k runs to be merged. So, over the entire sorting of a file the only time that is not overlapped with the internal merging time is the time to input the first k blocks and that to output the last block.

(4) The algorithm assumes that all blocks are of the same length. Ensuring this may require inserting a few dummy records into the last block of each run following the sentinel record with key $+\infty$.

Example 7.14: To illustrate the algorithm of Program 7.19, let us trace through it while it performs a three-way merge on the three runs of Figure 7.24. Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is $+\infty$. We have six input buffers and two output buffers. Figure 7.25 shows the status of the input buffer queues, the run from which the next block is being read, and the output buffer being output at the beginning of each iteration of the loop of Steps 3 through 8 of the buffering algorithm.

Run 0	20 25	26 28	29 30	33 $+\infty$
Run 1	23 29	34 36	38 60	70 $+\infty$
Run 2	24 28	31 33	40 43	50 $+\infty$

Figure 7.24: Three runs

From line 5 of Figure 7.25 it is evident that during the k-way merge, the test for "output buffer full?" should be carried out before the test "input buffer empty?", as the next input buffer for that run may not have been read in yet, so there would be no next buffer in that queue. In lines 3 and 4 all six input buffers are in use, and the stack of free buffers is empty. □

We end our discussion of buffer handling by proving that Program 7.19 is correct.

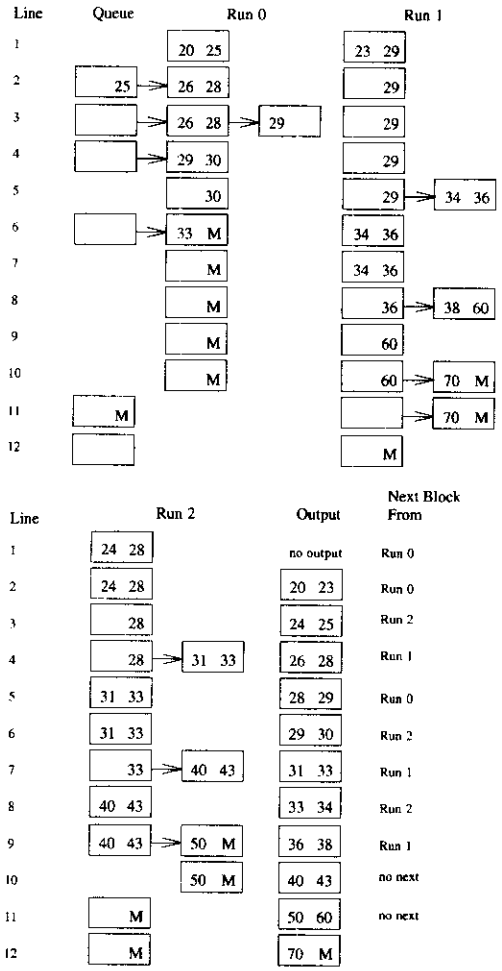


Figure 7.25: Buffering example

Theorem 7.2: The following are true for Program 7.19:

(1) In Step 6, there is always a buffer available in which to begin reading the next block.

(2) During the k-way merge of Step 3, the next block in the queue has been read in by the time it is needed.

Proof: (1) Each time we get to Step 6 of the algorithm, there are at most $k + 1$ buffer loads in memory, one of these being in an output buffer. For each queue there can be at most one buffer that is partially full. If no buffer is available for the next read, then the remaining k buffers must be full. This means that all the k partially full buffers are empty (as otherwise there will be more than $k+1$ buffer loads in memory). From the way the merge is set up, only one buffer can be both unavailable and empty. This may happen only if the output buffer gets full exactly when one input buffer becomes empty. But $k > 1$ contradicts this. So, there is always at least one buffer available when Step 6 is being executed.

(2) Assume this is false. Let run R_i be the one whose queue becomes empty during *kWayMerge*. We may assume that the last key merged was not $+\infty$, since otherwise *kWayMerge* would terminate the merge rather than get another buffer for R_i. This means that there are more blocks of records for run R_i on the input file, and $lastKey[i] \neq +\infty$. Consequently, up to this time whenever a block was output, another was simultaneously read in. Input and output therefore proceeded at the same rate, and the number of available blocks of data was always k. An additional block is being read in, but it does not get queued until Step 5. Since the queue for R_i has become empty first, the selection rule for choosing the next run to read from ensures that there is at most one block of records for each of the remaining $k - 1$ runs. Furthermore, the output buffer cannot be full at this time, as this condition is tested for before the input-buffer-empty condition. Thus, fewer than k blocks of data are in memory. This contradicts our earlier assertion that there must be exactly k such blocks of data. □

7.10.4 Run Generation

Using conventional internal sorting methods such as those discussed earlier in this chapter, it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time. Using a tree of losers, it is possible to do better than this. In fact, the algorithm we shall present will, on the average, generate runs that are twice as long as obtainable by conventional methods. This algorithm was devised by Walters, Painter, and Zalk. In addition to being capable of generating longer runs, this algorithm will allow for parallel input, output, and internal processing.

We assume that input/output buffers have been set up appropriately for maximum overlapping of input, output, and internal processing. Wherever there is an input/output instruction in the run-generation algorithm, it is assumed that the operation takes place through the input/output buffers. The run generation algorithm uses a tree of losers. We assume that there is enough space to construct such a tree for k records, $record[i]$, $0 \leq i < k$. Each node, i, in this tree has one field $loser[i]$. $loser[i]$, $1 \leq i < k$, represents the loser of the tournament played at node i. Each of the k record positions $record[i]$ has a run number $runNum[i]$, $0 \leq i < k$. This field enables us to determine whether or not $record[i]$ can be output as part of the run currently being generated. Whenever the

tournament winner is output, a new record (if there is one) is input, and the tournament is replayed as discussed in Chapter 5.

Function *runGeneration* (Program 7.20) is an implementation of the loser tree strategy just discussed. The variables used in this function have the following significance:

$record[i]$, $0 \le i < k$	...	the k records in the tournament tree
$loser[i]$, $1 \le i < k$	...	loser of the tournament played at node i
$loser[0]$	...	winner of the tournament
$runNum[i]$, $0 \le i < k$	...	the run number to which $record[i]$ belongs
currentRun	...	run number of current run
winner	...	overall tournament winner
winnerRun	...	run number for $record[winner]$
maxRun	...	number of runs that will be generated
lastKey	...	key of last record output

The loop of lines 10 to 37 repeatedly plays the tournament outputting records. The variable *lastKey* is made use of in line 21 to determine whether or not the new record input, $record[winner]$, can be output as part of the current run. If $key[winner] < lastKey$ then $record[winner]$ cannot be output as part of the current run *currentRun*, as a record with larger key value has already been output in this run. When the tree is being readjusted (lines 27 to 36), a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in nondecreasing order of their run numbers. Within the same run, records come out of the tree in nondecreasing order of their key values. *maxRun* is used to terminate the function. In line 18, when we run out of input, a record with run number *maxRun* + 1 is introduced. When this record is ready for output, the function terminates from line 13.

Analysis of *runGeneration:* When the input list is already sorted, only one run is generated. On the average, the run size is almost $2k$. The time required to generate all the runs for an n run list is $O(n \log k)$, as it takes $O(\log k)$ time to adjust the loser tree each time a record is output. □

7.10.5 Optimal Merging of Runs

The runs generated by function *runs* may not be of the same size. When runs are of different size, the run merging strategy employed so far (i.e., make complete passes over the collection of runs) does not yield minimum runtimes. For example, suppose we have four runs of length 2, 4, 5, and 15, respectively. Figure 7.26 shows two ways to merge these using a series of two-way merges. The circular nodes represent a two-way merge using as input the data of the children nodes. The square nodes represent the initial runs. We shall refer to the circular nodes as *internal nodes* and the square ones as *external*

```
 1 void runGeneration(int k)
 2 {/* run generation using a k-player loser tree,
 3     variable declarations have been omitted */
 4   for (i = 0; i < k; i++) {/* input records */
 5     readRecord(record[i]); runNum[i] = 1;
 6   }
 7   initializeLoserTree();
 8   winner = loser[0]; winnerRun = 1;
 9   currentRun = 1; maxRun = 1;
10   while(1) {/* output runs */
11     if (winnerRun != currentRun) {/* end of run */
12       output end of run marker;
13       if (winnerRun > maxRun) return;
14       else currentRun = winnerRun;
15     }
16     writeRecord(record[winner]);
17     lastKey = record[winner].key;
18     if (end of input) runNum[winner] = maxRun + 1;
19     else { /* input new record into tree */
20       readRecord(record[winner]);
21       if (record[winner].key < lastKey)
22         /* new record is in next run */
23         runNum[winner] = maxRun = winnerRun + 1;
24       else runNum[winner] = currentRun;
25     }
26     winnerRun = runNum[winner];
27     /* adjust losers */
28     for (parent = (k+winner)/2; parent; parent /= 2;)
29       if ((runNum[loser[parent]] < winnerRun) ||
30            ((runNum[loser[parent]] == winnerRun)
31                && (record[loser[parent]].key <
32                    record[winner].key)))
33       {/* parent is the winner */
34         SWAP(winner, loser[parent], temp);
35         winnerRun = runNum[winner];
36       }
37   }
38 }
```

Program 7.20: Run generation using a loser tree

nodes. Each figure is a *merge tree*.

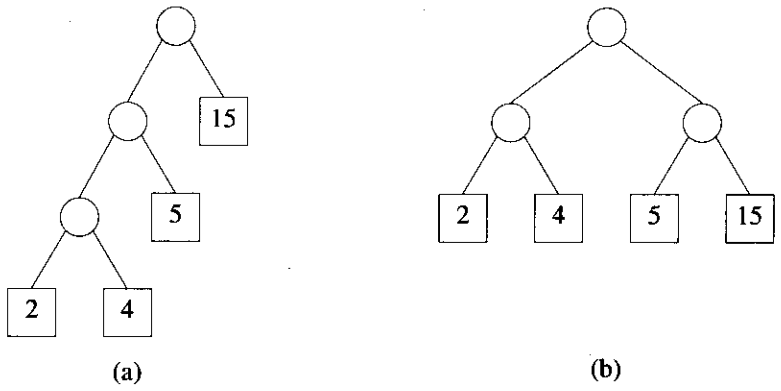


Figure 7.26: Possible two-way merges

In the first merge tree, we begin by merging the runs of size 2 and 4 to get one of size 6; next this is merged with the run of size 5 to get a run of size 11; finally this run of size 11 is merged with the run of size 15 to get the desired sorted run of size 26. When merging is done using the first merge tree, some records are involved in only one merge, and others are involved in up to three merges. In the second merge tree, each record is involved in exactly two merges. This corresponds to the strategy in which complete merge passes are repeatedly made over the data.

The number of merges that an individual record is involved in is given by the distance of the corresponding external node from the root. So, the records of the run with 15 records are involved in one merge when the first merge tree of Figure 7.26 is used and in two merges when the second tree is used. Since the time for a merge is linear in the number of records being merged, the total merge time is obtained by summing the products of the run lengths and the distance from the root of the corresponding external nodes. This sum is called the *weighted external path length*. For the two trees of Figure 7.26, the respective weighted external path lengths are

$$2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1 = 43$$

and

$$2 \cdot 2 + 4 \cdot 2 + 2 + 5 \cdot 2 + 15 \cdot 2 = 52$$

The cost of a k-way merge of n runs of length q_i, $1 \le i \le n$, is minimized by using a merge tree of degree k that has minimum weighted external path length. We shall consider the case $k = 2$ only. The discussion is easily generalized to the case $k > 2$ (see the exercises).

We briefly describe another application for binary trees with minimum weighted external path length. Suppose we wish to obtain an optimal set of codes for messages $M_1, \cdots, M_{n+1}$. Each code is a binary string that will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a *decode tree*. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 7.27 corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3, and M_4, respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decoding time is

$$\sum_{1 \le i \le n+1} q_i d_i$$

where d_i is the distance of the external node for message M_i from the root node. The expected decoding time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length.

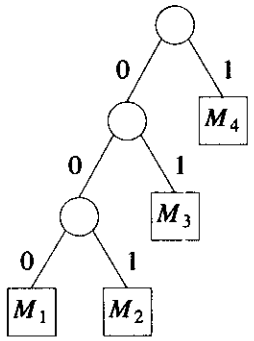


Figure 7.27: A decode tree

A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman. We simply state his algorithm and leave the correctness proof as an exercise. The following type declarations are assumed:

```
typedef struct treeNode *tree_pointer;
typedef struct {
        treePointer leftChild;
        int         weight;
        treePointer rightChild;
        } treeNode;
```

The *huffman* function (Program 7.21) begins with n extended binary trees, each containing one node. These are in the array *heap* []. Each node in a tree has three fields: *weight*, *left_child*, and *right_child*. The single node in each of the initial extended binary trees has as weight of one of the q_i's. During the course of the algorithm, for any tree in *heap* with root node *tree* and depth greater than 1, *tree* $\rightarrow$ *weight* is the sum of the weights of all external nodes in the tree rooted at *tree*. The *huffman* function uses the min heap functions *push*, *pop*, and *initialize*; *push* adds a new element to the min heap, *pop* deletes and returns the element with minimum weight, and *initialize* initializes the min heap. As discussed in Section 7.6, a heap can be initialized in linear time.

```
void huffman(treePointer heap[], int n)
{/* heap[1:n] is a list of single-node binary trees */
   treePointer tree;
   int i;
   /* initialize min heap */
   initialize(heap, n);
   /* create a new tree by combining the trees with the
      smallest weights until one tree remains */

   for (i = 1; i < n; i++) {
      MALLOC(tree, sizeof(*tree));
      tree →leftChild = pop(&n);
      tree→rightChild = pop(&n);
      tree→weight = tree→leftChild→weight +
                    tree→rightChild→weight;
      push(tree,&n); /* add to min heap */
   }
}
```

Program 7.21: Finding a binary tree with minimum weighted external path length

Example 7.15: Suppose we have the weights $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$, and $q_6 = 13$. Then the sequence of trees we would get is given in Figure 7.28 (the

number in a circular node represents the sum of the weights of external nodes in that subtree).

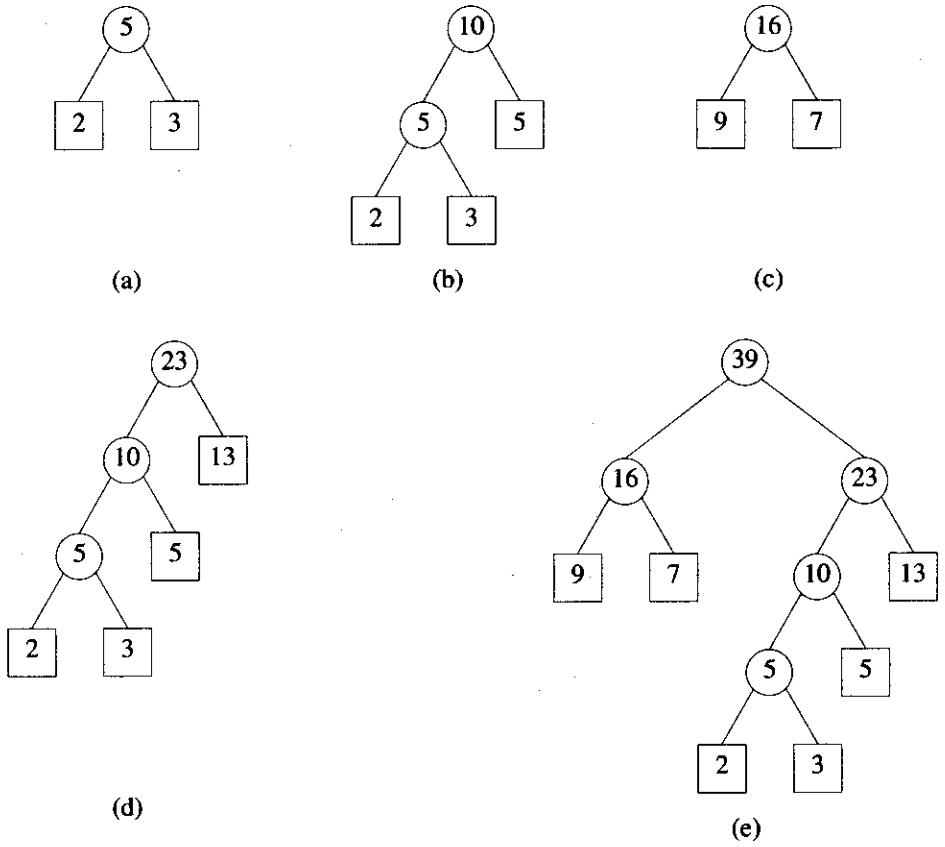


Figure 7.28: Construction of a Huffman tree

The weighted external path length of this tree is

$$2 \cdot 4 + 3 \cdot 4 + 5 \cdot 3 + 13 \cdot 2 + 7 \cdot 2 + 9 \cdot 2 = 93$$

By comparison, the best complete binary tree has weighted path length 95. □

Analysis of *huffman*: Heap initialization takes O(n) time. The main **for** loop is executed $n - 1$ times. Each call to *push* and *pop* requires only O(log n) time. Hence, the asymptotic computing time for the algorithm is O(n log n). □

EXERCISES

1. (a) n records are to be sorted on a computer with a memory capacity of S records ($S << n$). Assume that the entire S-record capacity may be used for input/output buffers. The input is on disk and consists of m runs. Assume that each time a disk access is made, the seek time is t_s and the latency time is t_l. The transmission time is t_t per record transmitted. What is the total input time for phase two of external sorting if a k-way merge is used with internal memory partitioned into input/output buffers to permit overlap of input, output, and CPU processing as in *buffering* (Program 7.19)?

   (b) Let the CPU time needed to merge all the runs together be t_{CPU} (we may assume it is independent of k and hence constant). Let $t_s = 80\ ms$, $t_l = 20ms$, $n = 200{,}000$, $m = 64$, $t_t = 10^{-3}$ sec/record, and $S = 2000$. Obtain a rough plot of the total input time, t_{input}, versus k. Will there always be a value of k for which $t_{CPU} \approx t_{input}$?

2. (a) Show that function *huffman* (Program 7.21) correctly generates a binary tree of minimal weighted external path length.

   (b) When n runs are to be merged together using an m-way merge, Huffman's method can be generalized to the following rule: "First add $(1 - n) \bmod (m - 1)$ runs of length zero to the set of runs. Then, repeatedly merge the m shortest remaining runs until only one run is left." Show that this rule yields an optimal merge pattern for m-way merging.

7.11 REFERENCES AND SELECTED READINGS

A comprehensive discussion of sorting and searching may be found in *The Art of Computer Programming: Sorting and Searching*, by D. Knuth, Vol. 3, Second Edition, Addison-Wesley, Reading, MA, 1998.

CHAPTER 8

Hashing

8.1 INTRODUCTION

In this chapter, we again consider the ADT dictionary that was introduced in Chapter 5 (ADT 5.3). Examples of dictionaries are found in many applications, including the spelling checker, the thesaurus, the index for a database, and the symbol tables generated by loaders, assemblers, and compilers. When a dictionary with n entries is represented as a binary search tree as in Chapter 5, the dictionary operations *search*, *insert* and *delete* take $O(n)$ time. These dictionary operations may be performed in $O(\log n)$ time using a balanced binary search tree (Chapter 10). In this chapter, we examine a technique, called hashing, that enables us to perform the dictionary operations *search*, *insert* and *delete* in $O(1)$ expected time. We divide our discussion of hashing into two parts: *static hashing* and *dynamic hashing*.

8.2.1 Hash Tables

In *static hashing* the dictionary pairs are stored in a table, ht, called the *hash table*. The hash table is partitioned into b buckets, $ht[0], \cdots, ht[b-1]$. Each bucket is capable of holding s dictionary pairs (or pointers to this many pairs). Thus, a bucket is said to consist of s slots, each slot being large enough to hold one dictionary pair. Usually $s = 1$, and each bucket can hold exactly one pair. The address or location of a pair whose key is k is determined by a hash function, h, which maps keys into buckets. Thus, for any key k, $h(k)$ is an integer in the range 0 through $b-1$. $h(k)$ is the hash or home address of k. Under ideal conditions, dictionary pairs are stored in their home buckets.

Definition: The *key density* of a hash table is the ratio n/T, where n is the number of pairs in the table and T is the total number of possible keys. The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$. □

Suppose our keys are at most six characters long, where a character may be a decimal digit or an uppercase letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \le i \le 5} 26 \times 36^i > 1.6 \times 10^9$. Any reasonable application, however, uses only a very small fraction of these. So, the key density, n/T, is usually very small. Consequently, the number of buckets, b, which is usually of the same magnitude as the number of keys, in the hash table is also much less than T. Therefore, the hash function h maps several different keys into the same bucket. Two keys, k_1, and k_2, are said to be *synonyms* with respect to h if $h(k_1) = h(k_2)$.

As indicated earlier, under ideal conditions, dictionary pairs are stored in their home buckets. Since many keys typically have the same home bucket, it is possible that the home bucket for a new dictionary pair is full at the time we wish to insert this pair into the dictionary. When this situation arises, we say that an *overflow* has occurred. A *collision* occurs when the home bucket for the new pair is not empty at the time of insertion. When each bucket has 1 slot (i.e., $s = 1$), collisions and overflows occur simultaneously.

Example 8.1: Consider the hash table ht with $b = 26$ buckets and $s = 2$. We have $n = 10$ distinct identifiers, each representing a C library function. This table has a loading factor, α, of $10/52 = 0.19$. The hash function must map each of the possible identifiers onto one of the numbers, 0–25. We can construct a fairly simple hash function by associating the letters, a–z, with the numbers, 0–25, respectively, and then defining the hash function, $f(x)$, as the first character of x. Using this scheme, the library functions **acos**, **define**, **float**, **exp**, **char**, **atan**, **ceil**, **floor**, **clock**, and **ctime** hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively. Figure 8.1 shows the first 8

identifiers entered into the hash table.

	Slot 0	Slot 1
0	**acos**	**atan**
1		
2	**char**	**ceil**
3	**define**	
4	**exp**	
5	**float**	**floor**
6		
...		
25		

Figure 8.1: Hash table with 26 buckets and two slots per bucket

The identifiers **acos** and **atan** are synonyms, as are **float** and **floor**, and **ceil** and **char**. The next identifier, **clock**, hashes into the bucket $ht[2]$. Since this bucket is full, we have an overflow. Where in the table should we place **clock** so that we may retrieve it when necessary? We consider various solutions to the overflow problem in Section 8.2.3. □

When no overflows occur, the time required to insert, delete or search using hashing depends only on the time required to compute the hash function and the time to search one bucket. Hence, the insert, delete and search times are independent of n, the number of entries in the dictionary. Since the bucket size, s, is usually small (for internal-memory tables s is usually 1) the search within a bucket is carried out using a sequential search.

The hash function of Example 8.1 is not well suited for most practical applications because of the very large number of collisions and resulting overflows that occur. This is so because it is not unusual to find dictionaries in which many of the keys begin with the same letter. Ideally, we would like to choose a hash function that is both easy to compute and results in very few collisions. Since the ratio b/T is usually very small, it is impossible to avoid collisions altogether.

In summary, hashing schemes use a hash function to map keys into hash-table buckets. It is desirable to use a hash function that is both easy to compute and minimizes the number of collisions. Since the size of the key space is usually several orders of magnitude larger than the number of buckets and since the number of slots in a bucket is small, overflows necessarily occur. Hence, a mechanism to handle overflows is needed.

8.2.2 Hash Functions

A hash function maps a key into a bucket in the hash table. As mentioned earlier, the desired properties of such a function are that it be easy to compute and that it minimize the number of collisions. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs; that is, if k is a key chosen at random from the key space, then we want the probability that $h(k) = i$ to be $1/b$ for all buckets i. With this stipulation, a random key has an equal chance of hashing into any of the buckets. A hash function satisfying this property is called a *uniform hash function*.

Several kinds of uniform hash functions are in use in practice. Some of these compute the home bucket by performing arithmetic (e.g., multiplication and division) on the key. Since, in many applications, the data type of the key is not one for which arithmetic operations are defined (e.g., string), it is necessary to first convert the key into an integer (say) and then perform arithmetic on the obtained integer. In the following subsections, we describe four popular hash functions as well as ways to convert strings into integers.

8.2.2.1 Division

This hash function, which is the most widely used hash function in practice, assumes the keys are non-negative integers. The home bucket is obtained by using the modulo (%) operator. The key k is divided by some number D, and the remainder is used as the home bucket for k. More formally,

$$h(k) = k \% D$$

This function gives bucket addresses in the range 0 through $D - 1$, so the hash table must have at least $b = D$ buckets. Although for most key spaces, every choice of D makes h a uniform hash function, the number of overflows on real-world dictionaries is criticaly dependent on the choice of D. If D is divisible by two, then odd keys are mapped to odd buckets (as the remainder is odd), and even keys are mapped to even buckets. Since real-world dictionaries tend to have a bias toward either odd or even keys, the use of an even divisor D results in a corresponding bias in the distribution of home buckets. In practice, it has been found that for real-world dictionaries, the distribution of home buckets is biased whenever D has small prime factors such as 2, 3, 5, 7 and so on. However, the degree of bias decreases as the smallest prime factor of D increases. Hence, for best performance over a variety of dictionaries, you should select D so that it is a prime number. With this selction, the smallest prime factor of D is D itself. For most practical dictionaries, a very uniform distribution of keys to buckets is seen even when we choose D such that it has no prime factor smaller than 20.

When you write hash table functions for general use, the size of the dictionary to be accommodated in the hash table is not known. This makes it impractical to choose D

as suggested above. So, we relax the requirement on D even further and require only that D be odd to avoid the bias caused by an even D. In addition, we set b equal to the divisor D. As the size of the dictionary grows, it will be necessary to increase the size of the hash table ht dynamically. To satisfy the relaxed requirement on D, array doubling results in increasing the number of buckets (and hence the divisor D) from b to $2b + 1$.

8.2.2.2 Mid-Square

The mid-square hash function determines the home bucket for a key by squaring the key and then using an appropriate number of bits from the middle of the square to obtain the bucket address; the key is assumed to be an integer. Since the middle bits of the square usually depend on all bits of the key, different keys are expected to result in different hash addresses with high probability, even when some of the digits are the same. The number of bits to be used to obtain the bucket address depends on the table size. If r bits are used, the range of values is 0 through 2^r-1. So the size of hash tables is chosen to be a power of two when the mid-square function is used.

8.2.2.3 Folding

In this method the key k is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for k. There are two ways of carrying out this addition. In the first, all but the last partition are shifted to the right so that the least significant digit of each lines up with the corresponding digit of the last partition. The different partitions are now added together to get $h(k)$. This method is known as *shift folding*. In the second method, *folding at the boundaries*, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain $h(k)$. This is equivalent to reversing every other partition and then adding.

Example 8.2: Suppose that $k = 12320324111220$, and we partition it into parts that are three decimal digits long. The partitions are $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$, and $P_5 = 20$. Using shift folding, we obtain

$$h(k) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

When folding at the boundaries is used, we first reverse P_2 and P_4 to obtain 302 and 211, respectively. Next, the five partitions are added to obtain $h(k) = 123 + 302 + 241 + 211 + 20 = 897$. □

8.2.2.4 Digit Analysis

This method is particularly useful in the case of a static file where all the keys in the table are known in advance. Each key is interpreted as a number using some radix r. The same radix is used for all the keys in the table. Using this radix, the digits of each key are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

8.2.2.5 Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to nonnegative integers. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers. It is ok for us to convert the strings *data*, *structures*, and *algorithms* into the same integer (say, 199). In this section, we consider only the conversion of strings into non-negative integers. Similar methods may be used to convert other data types into non-negative integers to which the described hash functions may be applied.

Example 8.3: ***[Converting Strings to Integers]*** Since it is not necessary to convert strings into unique nonnegative integers, we can map every string, no matter how long, into an integer. Programs 8.1 and 8.2 show you two ways to do this.

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
     number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

Program 8.1 converts each character into a unique integer and sums these unique integers. Since each character maps to an integer in the range 0 through 255, the integer returned by the function is not much more than 8 bits long. For example, strings that are eight characters long would produce integers up to 11 bits long.

Program 8.2 shifts the integer corresponding to every other character by 8 bits and then sums. This results in a larger range for the integer returned by the function. □

```
unsigned int stringToInt(char *key)
{/* alternative additive approach to create a natural number
    that is within the integer range */
  int number = 0;
  while (*key)
  {
    number += *key++;
    if (*key) number += ((int) *key++) << 8;
  }
  return number;
}
```

Program 8.2: Alternative way to convert a string into a non-negative integer

8.2.3 Overflow Handling

8.2.3.1 Open Addressing

There are two popular ways to handle overflows: *open addressing* and *chaining*. In this section, we describe four open addressing methods—linear probing, which also is known as linear open addressing, quadratic probing, rehashing and random probing. In linear probing, when inserting a new pair whose key is k, we search the hash table buckets in the order, $ht[h(k)+i]$ % b, $0 \le i \le b-1$, where h is the hash function and b is the number of buckets. This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket. In case no such bucket is found, the hash table is full and it is necessary to increase the table size. In practice, to ensure good performance, table size is increased when the loading density exceeds a prespecified threshold such as 0.75 rather than when the table is full. Notice that when we resize the hash table, we must change the hash function as well. For example, when the division hash function is used, the divisor equals the number of buckets. This change in the hash function potentially changes the home bucket for each key in the hash table. So, all dictionary entries need to be remapped into the new larger table.

Example 8.4: Assume we have a 13-bucket table with one slot per bucket. As our data we use the words **for**, **do**, **while**, **if**, **else**, and **function**. Figure 8.2 shows the hash value for each word using the simplified scheme of Program 8.1 and the division hash function. Inserting the first five words into the table poses no problem since they have different hash addresses. However, the last identifier, **function**, hashes to the same bucket as **if**. Using a circular rotation, the next available bucket is at $ht[0]$, which is where we place **function** (Figure 8.3). □

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + [illegible] 105 + 111 + 110	870	12

Figure 8.2: Additive transformation

[0]	**nction**
	do
[4]	**while**
[5]	
[6]	
[7]	
[8]	
[9]	**else**
[10]	
[11]	
[12]	**if**

Figure 8.3: Hash table with linear probing (13 buckets, one slot pe[illegible]

When $s = 1$ and linear probing is used to handle overflows, a [illegible] the pair with key k proceeds as follows:

(1) Compute $h(k)$.

(2) Examine the hash table buckets in the order $ht[h(k)]$, $ht[(h(k) + 1) \% b]$, $\cdots$, $ht[(h(k) + j) \% b]$ until one of the following happens:

   (a) The bucket $ht[(h(k) + j) \% b]$ has a pair whose key is k; in this case, the desired pair has been found.

(b) $ht[h(k)+j]$ is empty; k is not in the table.

(c) We return to the starting position $ht[h(k)]$; the table is full and k is not in the table.

Program 8.3 is the resulting search function. This function assumes that the hash table *ht* stores pointers to dictionary pairs. The data type of a dictionary pair is *element* and data of this type has two componenets *item* and *key*.

```
element* search(int k)
{/* search the linear probing hash table ht (each bucket has
    exactly one slot) for k, if a pair with key k is found,
    return a pointer to this pair; otherwise, return NULL */
   int homeBucket, currentBucket;
   homeBucket = h(k);
   for (currentBucket = homeBucket; ht[currentBucket]
                && ht[currentBucket]->key != k;) {
      currentBucket = (currentBucket + 1) % b;
                     /* treat the table as circular */
      if (currentBucket == homeBuket)
         return NULL; /* back to start point */
   }
   if (ht[currentBucket]->key == k)
      return ht[currentBucket];
   return NULL;
}
```

Program 8.3: Linear probing

When linear probing is used to resolve overflows, keys tend to cluster together. Moreover, adjacent clusters tend to coalesce, thus increasing the search time. For example, suppose we enter the C built-in functions **acos**, **atoi**, **char**, **define**, **exp**, **ceil**, **cos**, **float**, **atol**, **floor**, and **ctime** into a 26-bucket hash table in that order. For illustrative purposes, we assume that the hash function uses the first character in each function name. Figure 8.4 shows the bucket number, the identifier contained in the bucket, and the number of comparisons required to insert the identifier. Notice that before we can insert **atol**, we must examine $ht[0], \ldots, ht[8]$, a total of nine comparisons. This is far worse than the worst case behavior of the search trees we will study in Chapter 10. If we retrieved each of the identifiers in *ht* exactly once, the average number of buckets examined would be 35/11 = 3.18 per identifier.

When linear probing is used together with a uniform hash hash function, the expected average number of key comparisons to look up a key is approximately

bucket	x	buckets searched
0	**acos**	1
1	**atoi**	2
2	**char**	1
3	**define**	1
4	**exp**	1
5	**ceil**	4
6	**cos**	5
7	**float**	3
8	**atol**	9
9	**floor**	5
10	**ctime**	9
...		
25		

Figure 8.4: Hash table with linear probing (26 buckets, one slot per bucket)

$(2 - \alpha)/(2 - 2\alpha)$, where α is the loading density. This is the average over all possible sets of keys yielding the given loading density and using a uniform function h. In the example of Figure 8.4, $\alpha = 11/26 = .42$ and $p = 1.36$. This indicates that the expected average number of comparisons to search a table with a loading density of .42 is 1.36. Even though the average number of comparisons is small, the worst case can be quite large.

Some improvement in the growth of clusters and hence in the average number of comparisons needed for searching can be obtained by *quadratic probing*. Linear probing was characterized by searching the buckets $(h(k) + i) \% b$, $0 \le i \le b - 1$, where b is the number of buckets in the table. In quadratic probing, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets $h(k)$, $(h(k) + i^2) \% b$, and $(h(k) - i^2) \% b$ for $1 \le i \le (b - 1)/2$. When b is a prime number of the form $4j + 3$, for j an integer, the quadratic search described above examines every bucket in the table. Figure 8.5 lists some primes of the form $4j + 3$.

An alternative method to retard the growth of clusters is to use a series of hash functions $h_1, h_2, \cdots, h_m$. This method is known as *rehashing*. Buckets $h_i(k)$, $1 \le i \le m$ are examined in that order. Yet another alternative, random probing, is explored in the exercises.

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Figure 8.5: Some primes of the form $4j + 3$

8.2.3.2 Chaining

Linear probing and its variations perform poorly because the search for a key involves comparison with keys that have different hash values. In the hash table of Figure 8.4, for instance, searching for the key **atol** involves comparisons with the buckets $ht[0]$ through $ht[8]$, even though only the keys in $ht[0]$ and $ht[1]$ had a collision with with **atol**; the remainder cannot possibly be **atol**. Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address $h(k)$ and examining only those keys in the list for $h(k)$. Although the list for $h(k)$ may be maintained using any data structure that supports the search, insert and delete operations (e.g., arrays, chains, search trees), chains are most frequently used. We typically use an array $ht[0{:}b-1]$ with $ht[i]$ pointing to the first node of the chain for bucket i. Program 8.4 gives the search algorithm for chained hash tables.

Figure 8.6 shows the chained hash table corresponding to the linear table found in Figure 8.4. The number of comparisons needed to search for any of the identifiers is now one each for **acos**, **char**, **define**, **exp** and **float**; two each for **atoi**, **ceil**, and **float**; three each for **atol** and **cos**; and four for **ctime**. The average number of comparisons is now 21/11 = 1.91.

To insert a new key, k, into a chain, we must first verify that it is not currently on the chain. Following this, k may be inserted at any position of the chain. Deletion from a chained hash table can be done by removing the appropriate node from its chain.

When chaining is used along with a uniform hash function, the expected average number of key comparisons for a successful search is $\approx 1 + \alpha/2$, where α is the loading density n/b (b = number of buckets). For $\alpha = 0.5$ this number is 1.25, and for $\alpha = 1$ it is 1.5. The corresponding numbers for linear probing are 1.5 and b, the table size.

The performance results cited in this section tend to imply that provided we use a uniform hash function, performance depends only on the method used to handle

```
element* search(int k)
{/* search the chained hash table ht for k, if a pair with
    this key is found, return a pointer to this pair;
    otherwise, return NULL.
   nodePointer current;
   int homeBucket = h(k);
   /* search the chain ht[homeBucket] */
   for (current = ht[homeBucket]; current;
                                  current = current→link)
      if (current→data.key == k) return &current→data;
   return NULL;
}
```

Program 8.4: Chain search

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
...
[25] → *NULL*

Figure 8.6: Hash chains corresponding to Figure 8.4

overflows. Although this is true when the keys are selected at random from the key space, it is not true in practice. In practice, there is a tendency to make a biased use of keys. Hence, in practice, different hash functions result in different performance. Generally, the division hash function coupled with chaining yields best performance.

The worst-case number of comparisons needed for a successful search remains $O(n)$ regardless of whether we use open addressing or chaining. The worst-case number of comparisons may be reduced to $O(\log n)$ by storing synonyms in a balanced search tree (see Chapter 10) rather than in a chain.

8.2.4 Theoretical Evaluation of Overflow Techniques

The experimental evaluation of hashing techniques indicates a very good performance over conventional techniques such as balanced trees. The worst-case performance for hashing can, however, be very bad. In the worst case, an insertion or a search in a hash table with n keys may take $O(n)$ time. In this section, we present a probabilistic analysis for the expected performance of the chaining method and state without proof the results of similar analyses for the other overflow handling methods. First, we formalize what we mean by expected performance.

Let $ht[0:b-1]$ be a hash table with b buckets, each bucket having one slot. Let h be a uniform hash function with range $[0, b-1]$. If n keys $k_1, k_2, \cdots, k_n$ are entered into the hash table, then there are b^n distinct hash sequences $h(k_1), h(k_2), \cdots, h(k_n)$. Assume that each of these is equally likely to occur. Let S_n denote the expected number of key comparisons needed to locate a randomly chosen k_i, $1 \le i \le n$. Then, S_n is the average number of comparisons needed to find the jth key k_j, averaged over $1 \le j \le n$, with each j equally likely, and averaged over all b^n hash sequences, assuming each of these also to be equally likely. Let U_n be the expected number of key comparisons when a search is made for a key not in the hash table. This hash table contains n keys. The quantity U_n may be defined in a manner analogous to that used for S_n.

Theorem 8.1: Let $\alpha = n/b$ be the loading density of a hash table using a uniform hashing function h. Then

(1) for linear open addressing

$$U_n \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

$$S_n \approx \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha)$$

$$S_n \approx -\left[\frac{1}{\alpha}\right]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha$$

$$S_n \approx 1 + \alpha/2$$

Proof: Exact derivations of U_n and S_n are fairly involved and can be found in Knuth's book *The Art of Computer Programming: Sorting and Searching* (see the References and

Selected Readings section). Here we present a derivation of the approximate formulas for chaining. First, we must make clear our count for U_n and S_n. If the key k being sought has $h(k) = i$, and chain i has q nodes on it, then q comparisons are needed if k is not on the chain. If k is in the jth node of the chain, $1 \le j \le q$, then j comparisons are needed.

When the n keys are distributed uniformly over the b possible chains, the expected number in each chain is $n/b = \alpha$. Since U_n equals the expected number of keys on a chain, we get $U_n = \alpha$.

When the ith key, k_i, is being entered into the table, the expected number of keys on any chain is $(i - 1)/b$. Hence, the expected number of comparisons needed to search for k_i after all n keys have been entered is $1 + (i - 1)/b$ (this assumes that new entries will be made at the end of the chain). Thus,

$$S_n = \frac{1}{n}\sum_{i=1}^{n}\{1 + (i-1)/b\} = 1 + \frac{n-1}{2b} \approx 1 + \frac{\alpha}{2} \quad \square$$

EXERCISES

1. Show that the hash function $h(k) = k \% 17$ does not satisfy the one-way property, weak collision resistance, or strong collision resistance.
2. Consider a hash function $h(k)=k \% D$, where D is not given. We want to figure out what value of D is being used. We wish to achieve this using as few attempts as possible, where an attempt consists of supplying the function with k and observing $h(k)$. Indicate how this may be achieved in the following two cases.
   (a) D is known to be a prime number in the range [10,20].
   (b) D is of the form 2^k, where k is an integer in [1,5].
3. Write a function to delete the pair with key k from a hash table that uses linear probing. Show that simply setting the slot previously occupied by the deleted pair to empty does not solve the problem. How must *Get* (Program 8.3) be modified so that a correct search is made in the situation when deletions are permitted? Where can a new key be inserted?
4. (a) Show that if quadratic searching is carried out in the sequence $(h(k) + q^2)$, $(h(k) + (q-1)^2)$, $\cdots$, $(h(k) + 1)$, $h(k)$, $(h(k) - 1)$, $\cdots$, $(h(k) - q^2)$ with $q = (b - 1)/2$, then the address difference $\% b$ between successive buckets being examined is

$$b-2, b-4, b-6, \ldots, 5, 3, 1, 1, 3, 5, \ldots, b-6, b-4, b-2$$

   (b) Write a function to search a hash table *ht* of size b for the key k. Use h as the hash function and the quadratic probing scheme discussed in the text to resolve overflows. Use the results of part (a) to reduce the computations.

5. [***Morris 1968***] In random probing, the search for a key, k, in a hash table with b buckets is carried out by examining the buckets in the order $h(k)$, $(h(k) + s(i)) \% b$, $1 \leq i \leq b-1$ where $s(i)$ is a pseudo random number. The random number generator must satisfy the property that every number from 1 to $b-1$ must be generated exactly once as i ranges from 1 to $b-1$.

   (a) Show that for a table of size 2^r, the following sequence of computations generates numbers with this property:

   Initialize q to 1 each time the search routine is called.
   On successive calls for a random number do the following:
   q *= 5
   q = low order r + 2 bits of q
   $s(i) = q/4$

   (b) Write search and insert functions for a hash table using random probing and the mid-square hash function. Use the random number generator of (a).

   It can be shown that for this method, the expected value for the average number of comparisons needed to search for a dictionary pair is $-(1/\alpha)\log(1-\alpha)$ for large tables (α is the loading factor).

6. Develop a hash table implementation in which overflows are resolved using a binary search tree. Use the division hash function with an odd divisor D and array doubling whenever the loading density exceeds a prespecified amount. Recall that in this context, array doubling actually increases the size of the hash table from its current size $b = D$ to $2b + 1$.

7. Write a function to list all the keys in a hash table in lexicographic order. Assume that linear probing is used. How much time does your function take?

8. Let the binary representation of key k be k_1k_2. Let $|t|$ denote the number of bits in k and let the first bit of k_1 be 1. Let $|k_1| = \lceil |k|/2 \rceil$ and $|k_2| = \lfloor |k|/2 \rfloor$. Consider the following hash function

$$h(k) = \text{middle } q \text{ bits of } (k_1 \oplus k_2)$$

   where $\oplus$ is the exclusive-or operator. Is this a uniform hash function if keys are drawn at random from the space of integers? What can you say about the behavior of this hash function in actual dictionary usage?

9. [***T. Gonzalez***] Design a dictionary representation that allows you to search, insert, and delete in O(1) time. Assume that the keys are integer and in the range [0, m) and that $m + n$ units of space are available, where n is the number of insertions to be made. (Hint: Use two arrays, $a[n]$ and $b[m]$, where $a[i]$ will be the $(i+1)$th pair inserted into the table. If k is the ith key inserted, then $b[k] = i$.) Write C++ functions to search, insert, and delete. Note that you cannot initialize the arrays a

and b as this would take $O(n + m)$ time.

10. [*T. Gonzalez*] Let $s = \{s_1, s_2, \cdots, s_n\}$ and $t = \{t_1, t_2, \cdots, t_r\}$ be two sets. Assume $1 \le s_i \le m$, $1 \le i \le n$, and $1 \le t_i \le m$, $1 \le i \le r$. Using the idea of Exercise 9, write a function to determine if $s \subseteq t$. Your function should work in $O(r + n)$ time. Since $s \equiv t$ iff $s \subseteq t$ and $t \subseteq s$, one can determine in linear time whether two sets are the same. How much space is needed by your function?

11. [*T. Gonzalez*] Using the idea of Exercise 9, write an $O(n + m)$ time function to carry out the task of *Verify2* (Program 7.3). How much space does your function need?

12. Using the notation of Section 8.2.4, show that when linear probing is used

$$S_n = \frac{1}{n}\sum_{i=0}^{n-1} U_i$$

Using this equation and the approximate equality

$$U_n \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)^2}\right] \quad \text{where } \alpha = \frac{n}{b}$$

show that

$$S_n \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)}\right]$$

8.3 DYNAMIC HASHING

8.3.1 Motivation for Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prespecified threshold. So, for example, if we currently have b buckets in our hash table and are using the division hash function with divisor $D = b$, then, when an insert causes the loading density to exceed the prespecified threshold, we use array doubling to increase the number of buckets to $2b + 1$. At the same time, the hash function divisor changes to $2b + 1$. This change in divisor requires us to rebuild the hash table by collecting all dictionary pairs in the original smaller size table and reinserting these into the new larger table. We cannot simply copy dictionary entries from the smaller table into corresponding buckets of the bigger table as the home bucket for each entry has potentially changed. For very large dictionaries that must be accessible on a 24/7 basis, the required rebuild means that dictionary operations must be suspended for unacceptably long periods while the rebuild is in progress. Dynamic

hashing, which also is known as extendible hashing, aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket. In other words, although table doubling increases the total time for a sequence of n dictionary operations by only $O(n)$, the time required to complete an insert that triggers the doubling is excessive in the context of a large dictionary that is required to respond quickly on a per operation basis. The objective of dynamic hashing is to provide acceptable hash table performance on a per operation basis.

We consider two forms of dynamic hashing—one uses a directory and the other does not—in this section. For both forms, we use a hash function h that maps keys into non-negative integers. The range of h is assumed to be sufficiently large and we use $h(k,p)$ to denote the integer formed by the p least significant bits of $h(k)$.

For the examples of this section, we use a hash function $h(k)$ that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation. Figure 8.7 shows 8 possible 2 character keys together with the binary representation of $h(k)$ for each. For our example hash function, $h(A\,0,1) = 0$, $h(A\,1,3) = 1$, $h(B\,1,4) = 1001 = 9$, and $h(C\,1,6) = 110\,001 = 49$.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure 8.7: An example hash function

8.3.2 Dynamic Hashing Using Directories

We employ a directory, d, of pointers to buckets. The size of the directory depends on the number of bits of $h(k)$ used to index into the directory. When indexing is done using, say, $h(k, 2)$, the directory size is $2^2 = 4$; when $h(k, 5)$ is used, the directory size is 32. The number of bits of $h(k)$ used to index the directory is called the *directory depth*. The

size of the directory is 2^t, where t is the directory depth and the number of buckets is at most equal to the directory size. Figure 8.8 (a) shows a dynamic hash table that contains the keys A0, B0, A1, B1, C2, and C3. This hash table uses a directory whose depth is 2 and uses buckets that have 2 slots each. In Figure 8.8, the directory is shaded while the buckets are not. In practice, the bucket size is often chosen to match some physical characteristic of the storage media. For example, when the dictionary pairs reside on disk, a bucket may correspond to a disk track or sector.

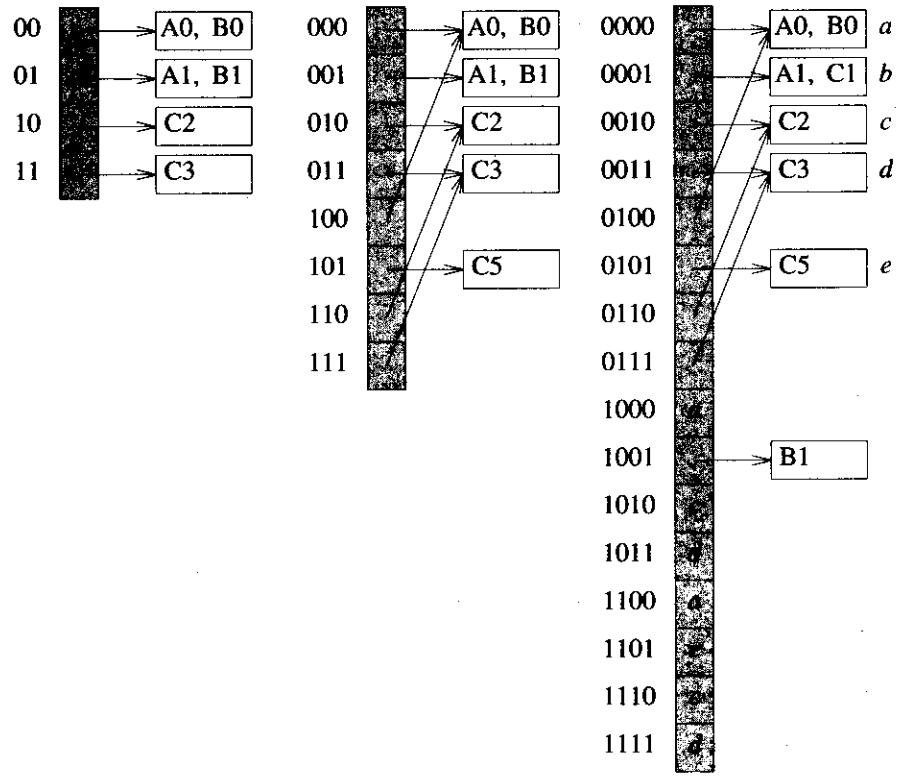


Figure 8.8: Dynamic hash tables with directories

To search for a key k, we merely examine the bucket pointed to by $d[h(k,t)]$, where t is the directory depth.

Suppose we insert C5 into the hash table of Figure 8.8 (a). Since, h(C5,2) = 01,

we follow the pointer, $d[01]$, in position 01 of the directory. This gets us to the bucket with A1 and B1. This bucket is full and we get a bucket overflow. To resolve the overflow, we determine the least u such that $h(k,u)$ is not the same for all keys in the overflowed bucket. In case the least u is greater than the directory depth, we increase the directory depth to this least u value. This requires us to increase the directory size but not the number of buckets. When the directory size doubles, the pointers in the original directory are duplicated so that the pointers in each half of the directory are the same. A quadrupling of the directory size may be handled as two doublings and so on. For our example, the least u for which $h(k,u)$ is not the same for A1, B1, and C5 is 3. So, the directory is expanded to have depth 3 and size 8. Following the expansion, $d[i] = d[i+4], 0 \le i < 4$.

Following the resizing (if any) of the directory, we split the overflowed bucket using $h(k,u)$. In our case, the overflowed bucket is split using $h(k, 3)$. For A1 and B1, $h(k, 3) = 001$ and for C5, $h(k, 3) = 101$. So, we create a new bucket with C5 and place a pointer to this bucket in $d[101]$. Figure 8.8 (b) shows the result. Notice that each dictionary entry is in the bucket pointed at by the directory position $h(k, 3)$, although, in some cases the dictionary entry is also pointed at by other buckets. For example, bucket 100 also points to A0 and B0, even though $h(A0,3) = h(B0,3) \neq 000$.

Suppose that instead of C5, we were to insert C1. The pointer in position $h(C1,2) = 01$ of the directory of Figure 8.8 (a) gets us to the same bucket as when we were inserting C5. This bucket overflows. The least u for which $h(k,u)$ isn't the same for A1, B1 and C1 is 4. So, the new directory depth is 4 and its new size is 16. The directory size is quadrupled and the pointers $d[0:3]$ are replicated 3 times to fill the new directory. When the overflowed bucket is split, A1 and C1 are placed into a bucket that is pointed at by $d[0001]$ and B1 into a bucket pointed at by $d[1001]$.

When the current directory depth is greater than or equal to u, some of the other pointers to the split bucket also must be updated to point to the new bucket. Specifically, the pointers in positions that agree with the last u bits of the new bucket need to be updated. The following example illustrates this. Consider inserting A4 ($h(A4) = 100$ 100) into Figure 8.8 (b). Bucket $d[100]$ overflows. The least u is 3, which equals the directory depth. So, the size of the directory is not changed. Using $h(k, 3)$, A0 and B0 hash to 000 while A4 hashes to 100. So, we create a new bucket for A4 and set $d[100]$ to point to this new bucket.

As a final insert example, consider inserting C1 into Figure 8.8 (b). $h(C1,3) = 001$. This time, bucket $d[001]$ overflows. The minumum u is 4 and so it is necessary to double the directory size and increase the directory depth to 4. When the directory is doubled, we replicate the pointers in the first half into the second half. Next we split the overflowed bucket using $h(k, 4)$. Since $h(k, 4) = 0001$ for A1 and C1 and 1001 for B1, we create a new bucket with B1 and put C1 into the slot previously occupied by B1. A pointer to the new bucket is placed in $d[1001]$. Figure 8.8 (c) shows the resulting configuration. For clarity, several of the bucket pointers have been replaced by lowercase letters indicating the bucket pointed to.

Deletion from a dynamic hash table with a directory is similar to insertion. Although dynamic hashing employs array doubling, the time for this array doubling is considerably less than that for the array doubling used in static hashing. This is so because, in dynamic hashing, we need to rehash only the entries in the bucket that overflows rather than all entries in the table. Further, savings result when the directory resides in memory while the buckets are on disk. A search requires only 1 disk access; an insert makes 1 read and 2 write accesses to the disk, the array doubling requires no disk access.

8.3.3 Directoryless Dynamic Hashing

As the name suggests, in this method, which also is known as linear dynamic hashing, we dispense with the directory, d, of bucket pointers used in the method of Section 8.3.2. Instead, an array, ht, of buckets is used. We assume that this array is as large as possible and so there is no possibility of increasing its size dynamically. To avoid initializing such a large array, we use two variables q and r, $0 \leq q < 2^r$, to keep track of the *active buckets*. At any time, only buckets 0 through $2^r + q - 1$ are active. Each active bucket is the start of a chain of buckets. The remaining buckets on a chain are called *overflow buckets*. Informally, r is the number of bits of $h(k)$ used to index into the hash table and q is the bucket that will split next. More accurately, buckets 0 through $q - 1$ as well as buckets 2^r through $2^r + q - 1$ are indexed using $h(k, r + 1)$ while the remaining active buckets are indexed using $h(k, r)$. Each dictionary pair is either in an active or an overflow bucket.

Figure 8.9 (a) shows a directoryless hash table ht with $r = 2$ and $q = 0$. The hash function is that of Figure 8.7, $h(\text{B4}) = 101\ 100$, and $h(\text{B5}) = 101\ 101$. The number of active buckets is 4 (indexed 00, 01, 10, and 11). The index of an active bucket identifies its chain. Each active bucket has 2 slots and bucket 00 contains B4 and A0. There are 4 bucket chains, each chain begins at one of the 4 active buckets and comprises only that active bucket (i.e., there are no overflow buckets). In Figure 8.9 (a), all keys have been mapped into chains using $h(k, 2)$. In Figure 8.9 (b), $r = 2$ and $q = 1$; $h(k, 3)$ has been used for chains 000 and 100 while $h(k, 2)$ has been used for chains 001, 010, and 011. Chain 001 has an overflow bucket; the capacity of an overflow bucket may or may not be the same as that of an active bucket.

To search for k, we first compute $h(k,r)$. If $h(k,r) < q$, then k, if present, is in a chain indexed using $h(k, r + 1)$. Otherwise, the chain to examine is given by $h(k, r)$. Program 8.5 gives the algorithm to search a directoryless dynamic hash table.

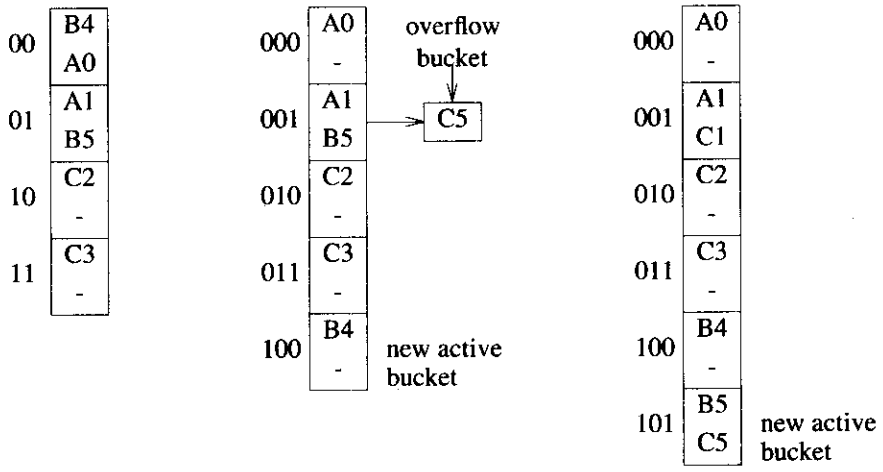
To insert C5 into the table of Figure 8.9 (a), we use the search algorithm of Program 8.5 to determine whether or not C5 is in the table already. Chain 01 is examined and we verify that C5 is not present. Since the active bucket for the searched chain is full, we get an overflow. An overflow is handled by activating bucket $2^r + q$; reallocating the entries in the chain q between q and the newly activated bucket (or chain) $2^r + q$,

if ($h(k,r) < q$) search the chain that begins at bucket $h(k,r+1)$;
else search the chain that begins at bucket $h(k,r)$;

Program 8.5: Searching a directoryless hash table

and incrementing q by 1. In case q now becomes 2^r, we increment r by 1 and reset q to 0. The reallocation is done using $h(k, r + 1)$. Finally, the new pair is inserted into the chain where it would be searched for by Program 8.5 using the new r and q values.

For our example, bucket 4 = 100 is activated and the entries in chain 00 ($q = 0$) are rehashed using $r + 1 = 3$ bits. B4 hashes to the new bucket 100 and A0 to bucket 000. Following this, $q = 1$ and $r = 2$. A search for C5 would examine chain 1 and so C5 is added to this chain using an overflow bucket (see Figure 8.9 (b)). Notice that at this time, the keys in buckets 001, 010 and 011 are hashed using $h(k, 2)$ while those in buckets 000 and 100 are hashed using $h(k, 3)$.



(a) $r = 2$, $q = 0$ (b) Insert C5, $r = 2$, $q = 1$ (c) Insert C1, $r = 2$, $q = 2$

Figure 8.9: Inserting into a directoryless dynamic hash table

Let us now insert C1 into the table of Figure 8.9 (b). Since, $h(C1,2) = 01 = q$, chain 01 = 1 is examined by our search algorithm (Program 8.5). The search verifies that C1 is not in the dictionary. Since the active bucket 01 is full, we get an overflow. We activate bucket $2^r + q = 5 = 101$ and rehash the keys A1, B5, and C5 that are in chain q. The rehashing is done using 3 bits. A1 is hashed into bucket 001 while B5 and C5 hash into bucket 101. q is incremented by 1 and the new key C1 is inserted into bucket 001. Figure 8.9 (c) shows the result.

EXERCISES

1. Write an algorithm to insert a dictionary pair into a dynamic hash table that uses a directory.
2. Write an algorithm to delete a dictionary pair from a dynamic hash table that uses a directory.
3. Write an algorithm to insert a dictionary pair into a directoryless dynamic hash table.
4. Write an algorithm to delete a dictionary pair from a directoryless dynamic hash table.

8.4 BLOOM FILTERS

8.4.1 An Application—Differential Files

Consider an application where we are maintaining an indexed file. For simplicity, assume that there is only one index and hence just a single key. Further assume that this is a dense index (i.e., one that has an entry for each record in the file) and that updates to the file (inserts, deletes, and changes to an existing record) are permitted. It is necessary to keep a backup copy of the index and file so that we can recover from accidental loss or failure of the working copy. This loss or failure may occur for a variety of reasons, which include corruption of the working copy due to a malfunction of the hardware or software. We shall refer to the working copies of the index and file as the *master index* and *master file*, respectively.

Since updates to the file and index are permitted, the backup copies of these generally differ from the working copies at the time of failure. So, it is possible to recover from the failure only if, in addition to the backup copies, we have a log of all updates made since the backup copies were created. We shall call this log the *transaction log*. To recover from the failure, it is necessary to process the backup copies and the transaction log to reproduce an index and file that correspond to the working copies at the time of failure. The time needed to recover is therefore a function of the sizes of the backup index and file and the size of the transaction log. The recovery time can be reduced by making more frequent backups. This results in a smaller transaction log. Making sufficiently frequent backups of the master index and file is not practical when these are

very large and when the update rate is very high.

When only the file (but not the index) is very large, a reduction in the recovery time may be obtained by keeping updated records in a separate file called the *differential file*. The master file is unchanged. The master index is, however, changed to reflect the position of the most current version of the record with a given key. We assume that the addresses for differential-file records and master-file records are different. As a result, by examining the address obtained from a search of the master index, we can tell whether the most current version of the record we are seeking is in the master file or in the differential file. The steps to follow when accessing a record with a given key are given in Program 8.6(b). Program 8.6(a) gives the steps when a differential file is not used.

Notice that when a differential file is used, the backup file is an exact replica of the master file. Hence, it is necessary to backup only the master index and differential file frequently. Since these are relatively small, it is feasible to do this. To recover from a failure of the master index or differential file, the transactions in the transaction log need to be processed using the backup copies of the master file, index, and differential file. The transaction log can be expected to be relatively small, as backups are done more frequently. To recover from a failure of the master file, we need merely make a new copy of its backup. When the differential file becomes too large, it is necessary to create a new version of the master file by merging the old master file and the differential file. This also results in a new index and an empty differential file. It is interesting to note that using a differential file as suggested does not affect the number of disk accesses needed to perform a file operation (see Program 8.6(a,b)).

Suppose that both the index and the file are very large. In this case the differential-file scheme discussed above does not work as well, as it is not feasible to backup the master index as frequently as is necessary to keep the transaction log sufficiently small. We can get around this difficulty by using a differential file and a differential index. The master index and master file remain unchanged as updates are performed. The differential file contains all newly inserted records and the current versions of all changed records. The differential index is an index to the differential file. This also has null address entries for deleted records. The steps needed to perform a file operation when both a differential index and file are used are given in Program 8.6(c). Comparing with Program 8.6(a), we see that additional disk accesses are frequently needed, as we will often first query the differential index and then the master index. Observe that the differential file is much smaller than the master file, so most requests are satisfied from the master file.

When a differential index and file are used, we must backup both of these with high frequency. This is possible, as both are relatively small. To recover from a loss of the differential index or file, we need to process the transactions in the transaction log using the available backup copies. To recover from a loss of the master index or master file, a copy of the appropriate backup needs to be made. When the differential index and/or file becomes too large, the master index and/or file is reorganized so that the differential index and/or file becomes empty.

Step 1: Search master index for record address.

Step 2: Access record from this master file address.

Step 3: If this is an update, then update master index, master file, and transaction log.

(a) No differential file

Step 1: Search master index for record address.

Step 2: Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3: If this is an update, then update master index, differential file, and transaction log.

(b) Differential file in use

Step 1: Search differential index for record address. If the search is unsuccessful, then search the master index.

Step 2: Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3: If this is an update, then update differential index, differential file, and transaction log.

(c) Differential index and file in use

Step 1: Query the Bloom filter. If the answer is "maybe," then search differential index for record address. If the answer is "no" or if the differential index search is unsuccessful, then search the master index.

Step 2: Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3: If this is an update, then update Bloom filter, differential index, differential file, and transaction log.

(d) Differential index and file and Bloom filter in use

Program 8.6: Access steps

8.4.2 Bloom Filter Design

The performance degradation that results from the use of a differential index can be considerably reduced by the use of a *Bloom filter*. This is a device that resides in internal memory and accepts queries of the following type: Is key k in the differential index? If

queries of this type can be answered accurately, then there will never be a need to search both the differential and master indexes for a record address. Clearly, the only way to answer queries of this type accurately is to have a list of all keys in the differential index. This is not possible for differential indexes of reasonable size.

A Bloom filter does not answer queries of the above type accurately. Instead of returning one of "yes" and "no" as its answer, it returns one of "maybe" and "no". When the answer is "no," then we are assured that the key k is not in the differential index. In this case, only the master index is to be searched, and the number of disk accesses is the same as when a differential index is not used. If the answer is "maybe," then the differential index is searched. The master index needs to be searched only if k is not found in the differential index. Program 8.6(d) gives the steps to follow when a Bloom filter is used in conjunction with a differential index.

A *filter error* occurs whenever the answer to the Bloom filter query is "maybe" and the key is not in the differential index. Both the differential and master indexes are searched only when a filter error occurs. To obtain a performance close to that when a differential index is not in use, we must ensure that the probability of a filter error is close to zero.

Let us take a look at a Bloom filter. Typically, it consists of m bits of memory and h uniform and independent hash functions $f_1, \cdots, f_h$. Each f_i hashes a key k to an integer in the range $[1,m]$. Initially all m filter bits are zero, and the differential index and file are empty. When key k is added to the differential index, bits $f_1(k), \cdots, f_h(k)$ of the filter are set to 1. When a query of the type "Is key k in the differential index?" is made, bits $f_1(k), \cdots, f_h(k)$ are examined. The query answer is "maybe" if all these bits are 1. Otherwise, the answer is "no." One may verify that whenever the answer is "no," the key cannot be in the differential index and that when the answer is "maybe," the key may or may not be in the differential index.

We can compute the probability of a filter error in the following way. Assume that initially there are n records and that u updates are made. Assume that none of these is an insert or a delete. Hence, the number of records remains unchanged. Further, assume that the record keys are uniformly distributed over the key space and that the probability that an update request is for record i is $1/n$, $1 \le i \le n$. From these assumptions, it follows that the probability that a particular update does not modify record i is $1 - 1/n$. So, the probability that none of the u updates modifies record i is $(1 - 1/n)^u$. Hence, the expected number of unmodified records is $n(1 - 1/n)^u$, and the probability that the $(u+1)$'st update is for an unmodified record is $(1 - 1/n)^u$.

Next, consider bit i of the Bloom filter and the hash function f_j, $1 \le j \le h$. Let k be the key corresponding to one of the u updates. Since f_j is a uniform hash function, the probability that $f_j(k) \neq i$ is $1 - 1/m$. As the h hash functions are independent, the probability that $f_j(k) \neq i$ for all h hash functions is $(1 - 1/m)^h$. If this is the only update, the probability that bit i of the filter is zero is $(1 - 1/m)^h$. From the assumption on update requests, it follows that the probability that bit i is zero following the u updates is $(1 - 1/m)^{uh}$. From this, we conclude that if after u updates we make a query for an

unmodified record, the probability of a filter error is $(1-(1-1/m)^{uh})^h$. The probability, $P(u)$, that an arbitrary query made after u updates results in a filter error is this quantity times the probability that the query is for an unmodified record. Hence,

$$P(u) = (1-1/n)^u(1-(1-1/m)^{uh})^h$$

Using the approximation

$$(1-1/x)^q \sim e^{-q/x}$$

for large x, we obtain

$$P(u) \sim e^{-u/n}(1-e^{-uh/m})^h$$

when n and m are large.

Suppose we wish to design a Bloom filter that minimizes the probability of a filter error. This probability is highest just before the master index is reorganized and the differential index becomes empty. Let u denote the number of updates done up to this time. In most applications, m is determined by the amount of memory available, and n is fixed. So, the only variable in design is h. Differentiating $P(u)$ with respect to h and setting the result to zero yields

$$h = (\log_e 2)m/u \sim 0.693m/u$$

We may verify that this h yields a minimum for $P(u)$. Actually, since h has to be an integer, the number of hash functions to use either is $\lceil 0.693m/u \rceil$ or $\lfloor 0.693m/u \rfloor$, depending on which one results in a smaller $P(u)$.

EXERCISES

1. By differentiating $P(u)$ with respect to h, show that $P(u)$ is minimized when $h = (\log_e 2)m/u$.
2. Suppose that you are to design a Bloom filter with minimum $P(u)$ and that $n = 100{,}000$, $m = 5000$, and $u = 1000$.
   (a) Using any of the results obtained in the text, compute the number, h, of hash functions to use. Show your computations.
   (b) What is the probability, $P(u)$, of a filter error when h has this value?

8.5 REFERENCES AND SELECTED READINGS

For more on hashing, see *The Art of Computer Programming: Sorting and Searching*, by D. Knuth, Vol. 3, Second Edition, Addison-Wesley, Reading, MA, 1998 and "Hash tables", by P. Morin, in *Handbook of data structures and algorithms*, edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.

Our development of differential files and Bloom filters parallels that of Severence

and Lohman in the paper "Differential files: Their application to the maintenance of large databases," by D. Severence and G. Lohman, *ACM Transactions on Database Systems*, 1:3, 1976, pp. 256-267. This paper also provides several advantages of using differential files. The assumptions of uniformity made in the filter error analysis are unrealistic, as, in practice, future accesses are more likely to be for records previously accessed. Several authors have attempted to take this into account. Two references are "A practical guide to the design of differential file architectures," by H. Aghili and D. Severance, *ACM Transactions on Database Systems*, 7:2, 1982, pp. 540-565; and "A regression approach to performance analysis for the differential file architecture," by T. Hill and A. Srinivasan, *Proceedings of the Third IEEE International Conference on Data Engineering*, 1987, pp. 157-164.

Bloom filters have found application in the solution to problems in a variety of domains. Some applications to network related problems may be found in "Space-code Bloom filter for efficient traffic flow measurement," by A. Kumar, J. Xu, L. Li and J. Wang, *ACM Internet Measurement Conference*, 2003; "Hash-based paging and location update using Bloom filters," by P. Mutaf and C. Castelluccia, *Mobile Networks and Applications*, Kluwer Academic, 9, 627-631, 2004; and "Approximate caches for packet classification," by F. Chang, W. Feng and K. Li, *IEEE INFOCOM*, 2004.

CHAPTER 9

Priority Queues

9.1 SINGLE- AND DOUBLE-ENDED PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has an associated priority. We study two varieties of priority queues—single- and double-ended—in this chapter. Single-ended priority queues, which were first studied in Section 5.6, may be further categorized as min and max priority queues. As noted in Section 5.6.1, the operations supported by a min priority queue are:

SP1: Return an element with minimum priority.

SP2: Insert an element with an arbitrary priority.

SP3: Delete an element with minimum priority.

The operations supported by a max priority queue are the same as those supported by a min priority queue except that in SP1 and SP3 we replace minimum by maximum. The heap structure of Section 5.6 is a classic data structure for the representation of a priority queue. Using a min (max) heap, the minimum (maximum) element can be found