Chapter 9

Searching and Sorting

9.1 Searching

Frequently, it becomes necessary to search a list of records to identify a particular record. Usually each record contains a field whose value is unique for each record. Such a field is often called the "key" field. Identifying a particular record refers to finding a record whose key value is equal to a given value. This operation is called "searching". If search results in locating the desired record then the search is said to be successful. Otherwise, the search operation is said to be unsuccessful. For simplicity, each record is assumed to consist of a key field only. Further, the type of the key field is considered to be integer.

9.1.1 Sequential Search

It was discussed in Chapter 2 that a simple way to search for a key value "k" in an array "A" is to compare the values of the elements in "A" with "k". The process starts with a comparison between the first element and "k". As long as a comparison does not result in a success, the algorithm proceeds to compare the next element of "A" and "k". The process terminates when the list is exhausted or a comparison results in a success. This method of searching is known as sequential search or linear search. The following function employs this technique to search an element whose value equal to "k" within an array "A" of "n" elements defined as A[0:n-1]. The function returns an integer which holds the index of the element of "A" in the case of a successful search. When the search is unsuccessful, the function returns (-1).

```
int sequentialSearch(int k, int A[], int n)
{
    int i = 0;
```

```
while (i < n)
{
        if (k == A[i])
                break;
        else
                i++;
}
if (i < n)
        return (i);
else
        return (-1);
}
```

When the values in the array "A" are not distinct then the function will return the first index, "i", of the array "A" which is equal to "k".

The analysis of the above algorithm is based on the assumption that "A" contains distinct values. First, note that the number of comparisons required for a successful search is not fixed. It depends on the position, which "k" occupies within the array. If it is the first element then only one comparison is necessary and if it is the last element then "n" comparisons are required. In general, "i" comparisons are necessary to search the i-th element. Now if "p_i" is the probability that the i-th element will be searched then the expected number of comparisons for a successful search is given by

$$C_s = 1.p_0 + 2.p_1 + \ldots + n.p_{n-1} = \sum_{i=0,(n-1)} i.p_i \quad \ldots\ldots(1)$$

If it is assumed that any element is an equally likely candidate for searching then $p_0 = p_1 = \ldots = p_{n-1} = 1/n$. In this case, we have

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

In case of an unsuccessful search, all elements must be examined unconditionally. Therefore, the expected number of comparisons in case of an unsuccessful search is given by

$$C_u = n$$

Equation (1) shows that "C_s" can be minimized if the probabilities "p_i" ($0 \le i < n$) are known and if it is possible to organize the array in any desired order. Since $p_0 + p_1 + \ldots + p_{n-1} = 1$, "$C_s$" is minimum when $p_0 \ge p_1 \ge p_2 \ge \ldots \ge p_{n-1}$. In other words, "$C_s$" is very small when the most frequently searched elements are placed towards the beginning of the array. This gives a basis for organizing an array to aid sequential searching.

A common experience is that in most of the cases, search is limited to the most active 20% elements of the array. If these 20% elements can be identified then the array can be better organized for linear search by placing the active elements near the beginning of the array.

9.1.2 Linear Search on a sorted array

A reduction in the expected number of comparisons required for an unsuccessful search is possible when the linear search technique is applied to a sorted array. Assuming that the values appear in ascending order, the search can be terminated as soon as an element with a value greater than or equal to "k" is found. When the array element is equal to "k" then the search is successful. When the value of an element is greater than "k", it can be concluded that "k" is not present in the array and the search ends unsuccessfully. The function to search sequentially in an array (sorted in an ascending manner) is presented in the following.

```
int searchSequentialSorted(int k, int A[], int n)
{
    int u = 0;
    int j = -1;
    while (u < n)
    {
            if (k ==A[u])
            {
                    j = u;
                    break;
            }
            else if (A[u] > k)
                    break;
            else
                    u = u + 1;
    }
    return (j);
}
```

In this case, there is no change in the expected number of comparisons for a successful search. The number of comparisons required for an unsuccessful search depends on the given value of "k". The number of comparisons for an unsuccessful search may be "1" in the best case and "n+1" in the worst case. In fact, this value may lie in any one of the (n+1) intervals created by the "n" elements. If "k" lies in any one of these intervals with equal probability, the expected number of comparisons is given by

$$C_u = (1 + 2 + .. + n + n)/(n+1) = n(n + 3)/2(n + 1)$$
$$= (n/2) + n/(n + 1) - (n/2 + 1) \text{ for large "n".}$$

This clearly justifies the intuitive expectation that there is a reduction of the expected number of comparisons required for an unsuccessful search.

If the array is assumed to be sorted, several searching techniques using the basic principle of linear search is possible. One such technique is described here. Suppose every r-th element of the array is compared with the given value, starting from the first element. The comparison proceeds until an element with greater than or equal to the given value is located or the last element of the array is reached. This enables to identify a contiguous portion of the array (not exceeding "r" elements) where the requested value is likely to be present. This portion is now linearly searched in the reverse order until a match is found or it is established that an element with the desired value is not present.

To see that the method is an improvement over the usual sequential search, the worst case behaviour of the algorithm given can be analyzed. The maximum number of comparisons required when every r-th element is examined is given by (n/r). Again, when the portion consisting of "r" or fewer elements are examined in the reverse order then the maximum number of comparisons that may be required is "r". Now, if "C" is the expected number of comparisons, then

$$C \leq n/r + r$$

Considering that the usual sequential search requires (n+1)/ 2 comparisons on an average, it may be mentioned that the present technique requires fewer comparisons when n>2r (approximately).

9.1.3 Binary Search

When the given array is sorted then a considerably better method of searching is possible. This method, known as binary search, makes a comparison between "k" and the middle element of the array. Since the array is sorted, this comparison results either in a match between "k" and the middle element of "A" or identifying the left half or the right half of the array to which the desired element may belong. In the case when the current element is not equal to the middle element of "A", the procedure is repeated on the half in which the desired element is likely to be present. Proceeding in this way, either the element is detected or the final division leads to a half consisting of no element. In this case, it is ascertained that the array does not contain "k".

This is a very efficient method of searching because each comparison enables one to eliminate half of the elements from further consideration. Because of this basic principle of elimination, the method is also known as dichotomous search.

There may be an ambiguity over the determination of the middle element of an array when the number of elements is even. In this case, there are two elements in the middle. However, an arbitrary choice of any one of these as the middle element will serve the purpose.

Suppose the array "A" consists of the following elements.

10, 25, 48, 69, 92, 101

Further, suppose that the array is searched for a given value of "25". Since the array consists of "6" elements, A[3] ("69") is chosen as the middle element. Since "25" is less than "69", it may be concluded that if "25" is at all present in "A" then it must be present in the half consisting of the first three elements. In this half, the number of elements is three. So, A[1] becomes the middle element. A comparison with this element results in the match. This algorithm leads us to the following function.

```
int binarySearch(int k, int A[], int n)
{
   int lower = 0, mid;
   int i = -1;
   int upper;
   upper = n-1;
   while (upper >= lower)
   {
          mid = (upper + lower )/2;
          if (k == A[mid])
          {
                 i = mid;
                 break;
          }
          else
          {
                 if (k > A[mid])
                        lower = mid + 1;
                 else
                        upper = mid - 1;
          }
   }
   return (i);
}
```

To analyze the performance of binary search, the maximum number of comparisons required for a successful search will be computed first. Let "j" be the smallest integer such that

$$2^j \geq (n+1) \qquad \ldots\ldots\ldots(2)$$

The maximum number of elements that are left after the first comparison is $2^{(j-1)} - 1$

and in general, the maximum number of elements left after "k" comparisons is $2^{(j-k)} - 1$. The desired element will definitely be found after "j" comparisons in which case we are left with no elements to be compared. Thus, C_{max}, the maximum number of comparisons for a successful search is given by $C_{max} = j$ where "j" is the smallest integer satisfying (2). Alternatively,

$$C_{max} = \log_2(n + 1)$$

Note that C_u, the number of comparisons required by an unsuccessful search, is equal to C_{max}. This is because, search can be declared as unsuccessful only when there are no elements to be probed.

Now, find the average number of comparisons under the assumption that each key is an equally likely candidate for a search. In other words, the probability that a particular element will be requested in a search is (1/n). For simplicity, we assume $n = 2^j - 1$ for some "j". Now, the element in the middle position requires only one comparison to be searched. Similarly, the elements in the middle positions of the two halves require two comparisons each when searched. In this way we see that the total number of comparisons required to search every array element is given by

$$C = 1.2^0 + 2.2^1 + 3.2^2 + \ldots\ldots\ldots\ldots + j.2^{j-1} \ldots\ldots\ldots \quad (3)$$

To find an expression for "C", multiply both sides of (3) by 2 to get

$$2.C = 1.2^1 + 2.2^2 + 3.2^3 + \ldots + j.2^j \ldots\ldots\ldots \quad (4)$$

Subtracting (3) from (4), we get

$$C = j.2^j - (2^0 + 2^1 + 2^2 + \ldots 2^{(j-1)})$$
$$= 1 + 2^j(j-1)$$

Now, C_s, the expected number of comparisons for a successful search, is given by

$$C_s = C/n = \{1 + 2^j.(j-1)\}/n$$
$$= [1 + (n + 1).\{\log_2 (n + 1) - 1\}]/n$$
$$= (n + 1).\log_2 (n + 1)/n - 1$$
$$= \log_2 n \text{ for large "n".}$$

Although the above result has been derived under the assumption that "n", the number of elements, is of the form $2^j - 1$, the approximate result is also valid for any value of "n". This is because there is no significant difference in the values of $\log_2 n$ for different values of n in the range $2^{(j-1)} \le n \le 2^j - 1$.

Thus, it is shown that the expected numbers of comparison for a successful as well as for an unsuccessful search are approximately given by

$$C_u = C_s = O(\log_2 n)$$

Comparison with corresponding expressions for sequential search shows that binary search is more efficient than sequential search when any element is equally likely to be searched. It must be mentioned here that binary search relies heavily on random access to the elements of the list. Thus, a binary search implementation on

Straight Insertion Sort

Shuttle sort can be improved considerably by replacing each of its exchange operation to a transfer operation. Note that in the i-th pass of shuttle sort, A[i] is inserted into a sorted segment A[0:i-1] producing a longer sorted segment in A[0:i]. This insertion is performed through a sequence of exchanges which places the element in A[i] to its correct position and moves all elements greater than this element to the right by one position each. The same objective can also be achieved in the following manner. In the beginning of the i-th pass, A[i] is assigned to a variable "x". Now "x" is successively compared with A[i-1], A[i-2], etc. until either an element smaller than "x" is found or the beginning of the array is reached. Elements that are found to be greater than "x", are moved right by one position each to make room for "x". When comparisons and movements are interleaved, the insertion process becomes identical to that of shuttle sort except that the sequence of exchanges becomes a sequence of transfers. In fact, insertion through transfer is more logical and efficient than insertion through exchanges.

An implementation of the straight insertion sort to sort an array A[0:n-1] is given in the following.

```
void straightInsertion (float A[], int n)
{
    int i = 1, j, k = 0;
    float x;
    while (i < n)
    {
            j = i - 1;
            x = A[j+1];
            k=i;
            while (j >= 0)
            {
                    if (x < A[j])
                    {
                            k = j;
                            j--;
                    }
                    else
                    break;
            }
            j = i;
            while (j > k)
            {
```

```
                A[j] = A[j-1];
                j—;
            }
            A[k] = x;
            i++;
        }
    }
```

As mentioned above, there is no basic difference between shuttle sort and straight insertion sort. As such, the number of comparisons and other attributes remain same. Taking into account the assignment A[i] to "x" and the placement of "x" in the correct position, the number of transfers is slightly more than the number of exchanges required for shuttle sort.

Binary Insertion Sort

Like shuttle sort or straight insertion sort, this method also inserts A[i] in its proper place in the i-th pass. However, the process of insertion is somewhat different. Binary search is applied on the first "i" elements to determine where exactly A[i] is to be inserted. Note that the first "i" elements are sorted and there is no problem in applying binary search to this segment. Suppose that binary search reveals that A[i] should be placed in the r-th position. Except when "r" is equal to (i+1), all elements stored in the r-th to the i-th positions are to be moved right by one position each to make room for the element to be inserted at the r-th position. Through the use of binary search, the number of comparisons is reduced considerably but the number of transfers remains the same as that in case of straight insertion sort. Moreover, comparisons and transfers cannot be interleaved. As such there should be two inner loops - one for the binary search and the other for the transfers. The actual implementation of the method is left as an exercise.

The number of comparisons required by binary search in the case of the i-th pass is $\log_2 i$ approximately. Thus, the sorting method requires "C" number comparisons where "C" is given by

$$C = \sum_{i=1,(n-1)} \log_2 i = \log_2 (n-1)! = \log_2 n! - \log_2 n$$

Approximating (n!) by Stirling's formula for large "n", we have

$$C = \log_2 (\sqrt{(2\pi)} n^{(n+1/2)} e^{-n}) - \log_2 n$$

$$\approx \log_2 (\sqrt{(2\pi/n)} + n\log_2 (n/e)$$

Although "C" is quite low, the time complexity of the method is still O(n) because the average number of transfers required by this method is the same as that in the case of straight insertion and is given by the expression

$$T = (n-1)(n+4) / 4$$

Binary insertion sort is not stable because when there are duplicate keys, binary insertion may place a latter key before its duplicate encountered earlier. The storage ratio is 1.

The choice of increments is the most important issue in shell sort. However, the question of finding an ideal sequence of increments is still open. The original suggestion of Shell is found to be unsatisfactory when the binary representation of "n" contains a long sequence of zeros. For example, suppose $n = 2^r$ and the given array is

$$(2^{r-1}+1, 1, 2^{r-1}+2, 2, \ldots\ldots).$$

Now, it can be easily verified that with Shell's choice of increments, not a single element will be moved in any pass other than the last one and actual sorting will take place only in the last pass. This defficiency of Shell's choice of increments was first pointed out by Frank and Lazrus (1960). They suggested that while calculating increments as per shell's formula, whenever necessary, a "1" should be added so as to make the increments odd. Hibberd (1963) proposed that the increments be taken as

$$d_1 = 2^k-1 \text{ and } d_{i+1} = d_i \text{ div } 2 \text{ where } 2^k < n \leq 2^{k+1}$$

Several other choices of increments have been suggested in literature but, as yet, there is no "best" sequence of increments, Hibberd's sequence of increments is usually recommended for use with shell sort. Another choice of increments is as follows:

$$d_p = 1, d_{i-1} = 1 + 3*d_i \text{ and } d_1 \leq (n/9).$$

So far, it has not been possible to evaluate the time complexity of Shell's method for an arbitrary array and for any general choice of increments. It is needless to mention that several attempts with fascinating results for specific cases have been made. In fact it can be shown that shell sort never does more than "n" comparisons when $d_p = 1$, $d_{i-1} = 1 + 3*d_i$ and $d_1 \leq (n/9)$. There have also been several experimental studies on the method. The average performance of Shell's technique is found to be $O(n\log_2 n)$. Storage rate for shell sort is 1. The method is not stable.

9.4 Sorting by Selection

As mentioned in section 9.2, every member of this family of sorting method works by successively selecting the smallest element, the second smallest element, the third smallest element and so on. The selected elements are placed sequentially in the output array in the order of their selections. It is obvious that such a method will correctly sort a given array. A selection sort can also work by selecting the largest element, then selecting the second largest element and so on. In this case the elements are to be placed in the output array in a reverse manner. Three methods of this family are described below.

9.4.1 Simple Selection Sort

The simplest possible technique, based on the principle of repeated selection, makes

use of "n" passes over a given array. In the i-th pass, the i-th smallest element is selected and it is placed in the i-th position of a separate output array. In order to ensure that an element once selected is not selected again, the position of the selected element in the original array is filled up with a value (say, "H") which exceeds any of the key values of the given array. The method is explained with a suitable example. Let an array "A" of six numbers be (40, 45, 23, 15, 18, 95). In the first pass, the smallest element is found in A[3]. The element B[0] of an output array "B" is assigned the value of A[3] and then A[3] is set to "H". In the second pass, the smallest element is searched in the array which is A[4]. B[1] gets the value of A[4] and A[4] is assigned "H". This process continues for six passes. Figure 9.4 illustrates the working of simple selection sort.

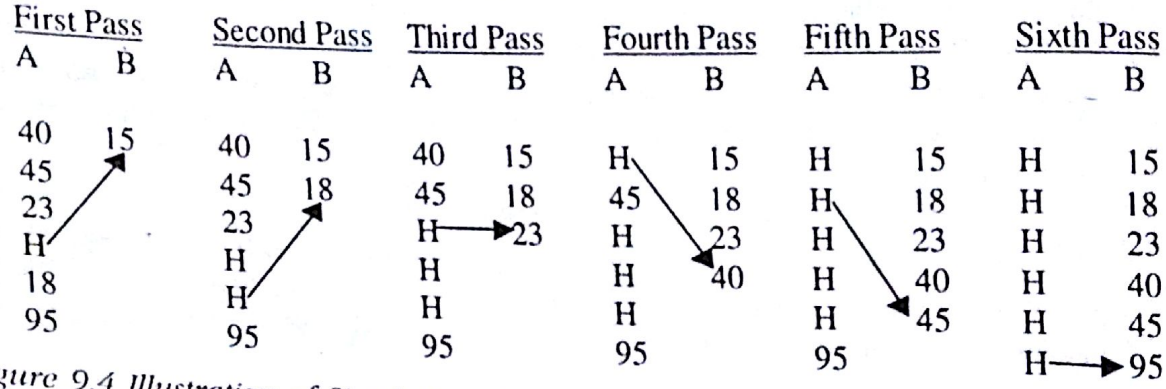

Figure 9.4 Illustration of Simple Selection Sort. The lines with arrow indicate that the value of an element from the original array is assigned to an element in the output array.

An implementation of the method is simple and is presented next. Here A[0:n-1] is the original unsorted array and B[0:n-1] is the final output sorted array. The number "100" is assumed to be the value of "H" which, in this case, exceeds any of the key values of the given array.

```
void simpleSelectionSort (float A[], float B[], int n)
{
    int i = 0, k, j;
    float T;
    T = 11;
    k=0;
    while (i < n)
    {
        for (j = 0; j < n; j++)
        {
            if (A[j] < T)
            {
                T = A[j];
                k = j;
```

```
                }
            }
            A[k] = H;
            B[i] = T;
            T = H;
            i++;
        }
    }
```

It may be observed that the method is not data sensitive. In every pass, the entire array is searched to find the smallest element. This requires (n-1) comparisons in each pass. Since there are "n" passes, a total of n(n-1) comparisons are required. Thus, the time complexity of the above algorithm is $O(n^2)$. In the above algorithm, the smallest element has been searched in such a way that if there exists more than one smallest element then the one that occurs first will be selected. Because of this, the method is stable. The storage ratio is 2.

9.4.2 Straight Selection Sort

This is merely an improvement of the simple selection sort. Instead of replacing the selected element by "H" in the i-th pass, the selected element is exchanged with the i-th element, A[i-1]. Thus, at the beginning of the i-th pass, the first (i-1) elements of the array are those that have been selected in the previous passes. The smallest element is now searched in the remaining (n-i+1) elements. After (n-1) passes, the sorted array is completely developed in the space occupied by the original array. The working of this method is shown in Figure 9.5. In the first pass, A[0] is swapped with "15", the smallest element in the entire array. In the second pass, A[1] is swapped with "18", the smallest element in the array A[1:5]. In the third pass, A[2] is swapped with "23", the smallest element in the array A[2:5]. In the fourth pass, A[3] is swapped with "38", the smallest element in the array A[3:5]. No more exchanges are necessary in the fifth pass.

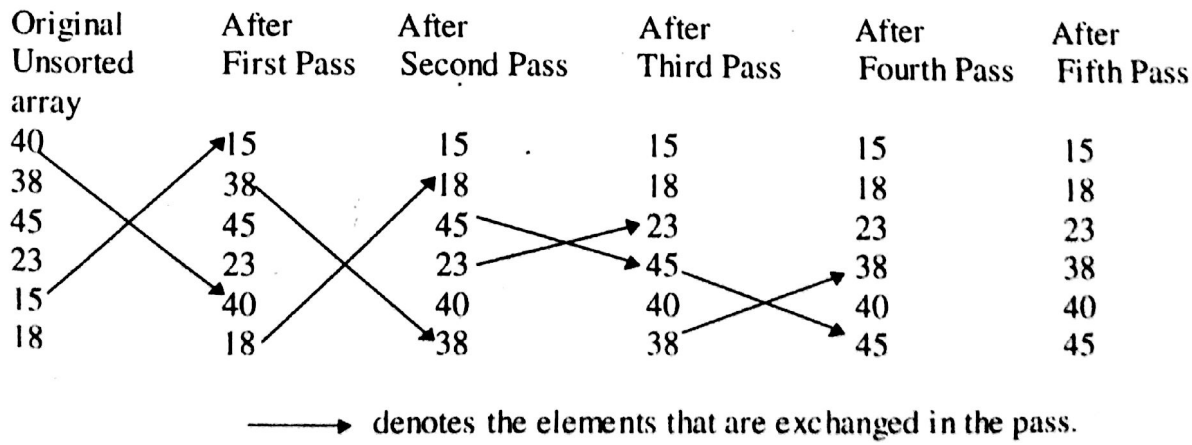

Figure 9.5 Illustration of Straight Selection Sort

An implementation of the straight selection sort is given below. Here, A[0:n-1] holds the unsorted array in the beginning and sorted array at the end.

```
void straightSelectionSort(float A[], int n)
{
    int i, k, j;
    float T;
    i = 0;
    while(i < n)
    {
            k = i;
            for(j = i+1; j < n; j++)
            {
                    if (A[j] < A[k])
                    k = j;
            }
            T = A[i];
            A[i] = A[k];
            A[k] = T;
            i++;
    }
}
```

This method is not data sensitive. There are "n" passes and in the i-th pass, the smallest element is searched among (n-i+1) elements. Therefore, the number of comparisons required to sort an array of "n" elements by this method is given by

$$C = \Sigma_{i=1,(n-1)}(n-i) = n(n-1)/2$$

Thus, although this method requires only half the number of comparisons required by simple selection sort, the time complexity is still $O(n^2)$. The method is stable as per the implementation given here. The storage ratio is 1.

9.4.2 Bubble Sort

Bubble sort is one of the most popular sorting methods. Bubble sort can be viewed as a selection sort because this method is also based on successively selecting the smallest element, second smallest element, etc. as in the case of any other selection sort. In order to find the successive smallest elements, the method relies heavily on the exchange of adjacent elements. Because of this, the method is often classed as "sorting by exchange".

Let "n" be the number of elements in an array "A". The first pass begins with a

comparison of the keys of the n-th and (n-1)-th element. If the n-th key is smaller, the two elements are exchanged. The smaller key, now in the (n-1)-th position, is compared with the key of the (n-2)-th element and if necessary, the elements are exchanged to place the smaller one in the (n-2)-th position. Comparisons progress in this manner and the first pass ends with the comparison and possible exchange of the elements A[1] and A[0]. It is easy to observe that this exercise in the first pass will "bubble up" the smallest element to A[0]. The second pass is an exact repetition of the first pass except that this time, the pass ends with the comparison and possible exchange of A[1] and A[2]. Thus, the second pass will "bubble up" the second smallest element to A[1]. Each subsequent pass is similar to the previous pass except that it excludes one more position (in the beginning of the array) from its consideration. The sorting method terminates after (n-1) passes by which time the array is sorted. The method may even be terminated earlier if no exchange is found necessary in an earlier pass. Absence of any exchange in a pass ensures that the elements are in correct order and there is no point in continuing the method.

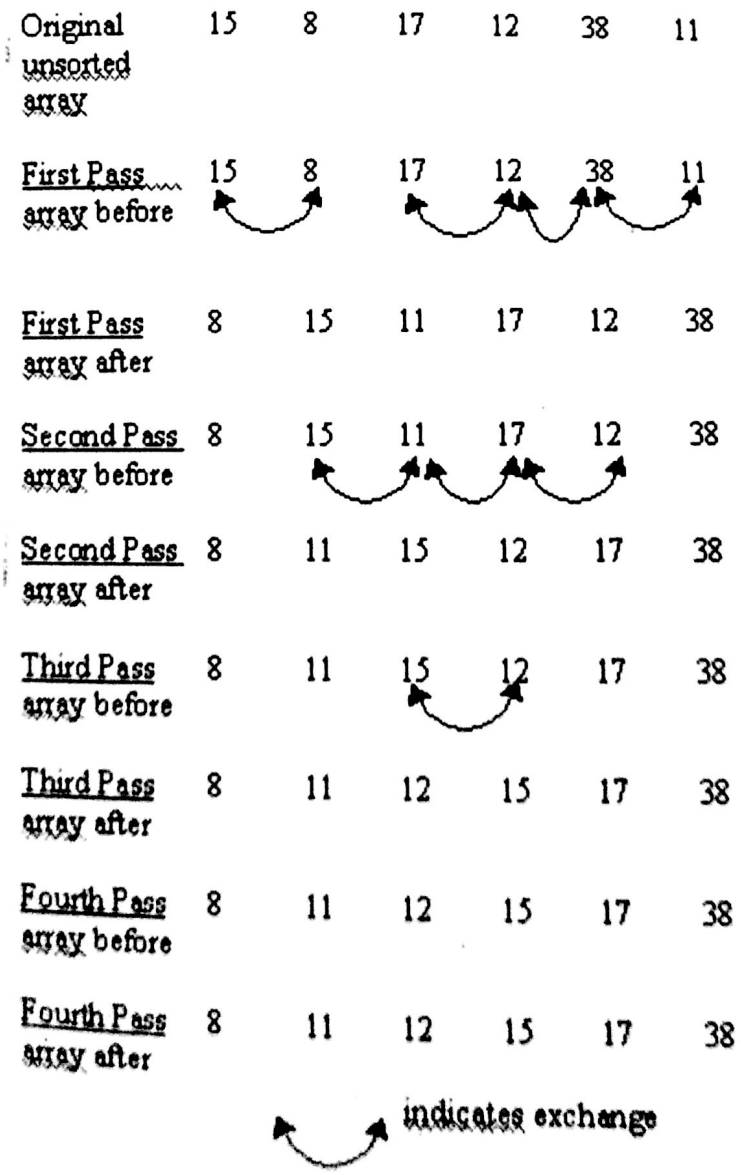

Figure 9.6 Illustration of Bubble Sort

present in any such tree. A tree having at least (n!) external nodes must have a height of log(n!) at the least. Any sorting algorithm must traverse from the root to the leaf node. So, the worst case complexity of any sorting algorithm can not be less than O(log(n!)) which is O(nlogn).

9.9 Hashing

Sequential search or binary search techniques described earlier in this chapter are based on comparing a given key with existing key values in a list. It has been observed that there are theoretical lower bounds on the average case complexity of search methods based on comparison. Therefore, one has to delve into new paradigms in order to find out a better mechanism for searching. "Hashing" provides a conceptually different mechanism to search a table for a given key value. In hashing, the record for a key value, "k", is directly referred by calculating the address from the key value.

Consider that there are "n" elements which are to be stored in a hash table of size "M", where (M ≥ n). An element with key value, "k", will be put in slot "j" of the hash table, if j = h (k); "h" is called the hash function. For example, h (k) may be defined as k mod M+1, where 1 ≤ h (k) ≤ M for any integer "k". Ideally, hash functions should result in a unique value when applied to any key. But, such a hash function is not practically available. So, it is quite possible that $h(k_1)$ may be the same as $h(k_2)$ for two distinct key values "k_1" and "k_2". As a result, while inserting a key "k_2" to the address given by $h(k_2)$, it is possible to find that the address is already used by another key "k_1", previously encountered. This indicates that two keys "k_1" and "k_2" collided. Often, "k_1" and "k_2" are called synonyms. Then, there has to be some collision resolution strategy to resolve this conflict.

It is easy to see that collision plays a major role in hashing methods. Thus, while the objective of a hash function is to minimize the collision, the objective of a collision resolution strategy is to locate a vacant position as efficiently as possible once collision occurs.

9.9.1 Hash Functions

The problem of finding a hash function can be stated as follows. There are "n" keys such that all key values are between "a" and "b". It is required to find a hash function f(key) that transforms a key into an address in the range "0" to (M-1). Moreover, M > n. Note that it is sufficient to map the addresses in the range "0" to (M-1). In case the storage locations start from an address "x" then the address computed by the hash function is to be added to "x" to get the effective address. The domain of the hashing function namely, the interval [a, b] is usually very wide. For example, suppose that there is an array of 800 elements and each key consists of 6 digits. If the array is housed in a storage area consisting of 1000 addresses, we have n=800 and m=1000. The domain of the hashing function is (0,999999) and its

range is (0, 999).

An ideal hash function should distribute the keys uniformly over the range (0, M-1). In other words, the probability of generating an address "x" (where $0 \leq x \leq (M-1)$) for a given key "k" should be (1/M). This is considered to be an "ideal" hash function because any key can occupy any address with equal probability. As such, it is not likely for a good hash function to overburden particular address with too many keys. Consequently, the number of collisions is expected to be low for a good hash function.

There are many methods for hashing. Some of the popular hash functions are discussed in the following. In all these cases, the keys will be assumed to be numeric integers. In case of alphanumeric keys, they are to be transformed into numeric values and then hash function is to be applied on the numeric equivalent of the keys. The numeric equivalent of the alphanumeric keys can be found by taking the ASCII values of the characters forming the keys.

The Division Method

In this method, the key is divided by "M" and the remainder is taken to be the address. Thus, the hash function, "h", is given by

$$h(k) = k \bmod M$$

An advantage of this method is that it produces addresses exactly in the range "0" to (M-1). However, "M" should be chosen carefully. Certain choices of "M" are not at all satisfactory. For example, if the keys are decimal integers and "M" is chosen to be a power of "10", say 1000, then all keys having identical last three digits will hash into the same address. Again, if "M" is chosen to be an even integer then all even keys will hash into odd addresses. Obviously, keys will not be hashed uniformly in the range (0, M-1). For example, if there are more odd keys than the even ones, the method will produce more odd addresses. Any hashing method that seeks to retain the effect of the original key sequence pattern should be avoided. Experience has shown that any number which is not divisible by 2, 3, 5, 10 is usually a good choice for "M". It has also been pointed out that non-prime odd integers having no divisors less than "19" can produce very good results when chosen as the value of "M". Another good choice of "M" is a prime number. Such a choice of "M" usually removes any pattern present in the key sequence from the generated addresses. However, a prime number related to the base of the number system used for the key by some simple relation should be avoided. For example, primes such as 4999, 7001 or 7999 are related to 10 by the relations $q.10^p + 1$ or $q.10^p - 1$. Use of such prime numbers would have effects of the last p-digits of the original key on the remainders. It is easy to see that if 4999 is chosen as "M", keys such as 00234, 05233, 10232, 15231 etc. all map into the address 234. Here, the quotient of the division of the first digits by 5, when added to the last three digits produces the address.

The Mid square Method

In this method, the key is squared and a portion of the squared value is selected from the middle which is chosen as the address. Suppose that a q-digit address is to be generated from a p-digit key. To do so, the p-digit key is first squared to yield a 2p-digit value. The q-digit address, then, is selected from the middle of this 2p-digit result. For example, let p = 4, q = 3 and the key be "3271". Square of "3271" is "10699441". Since 3 digits are to be extracted from the middle, the first three and the last two digits can be removed and "994" may be selected as the address. The method can also be modified to select "699" as the address.

This mid square method has the advantage that it is a flexible method and it can be suitably modified if needed. For example, when the key is large then only a selected part of the keys may be squared. Usually, mid square method is found to produce addresses that are uniformly distributed over the range of the hashing function.

Folding

Suppose we have p-digit keys from which q-digit addresses are to be generated. In this method, the digits of a key are partitioned into groups of q-digits from the right. These groups are then added and the rightmost q-digits of the sum is selected as the address. The following example will show the working of the method. Suppose we have an 8-digit key namely, 39427829 and we have to generate a 3-digit address from it. When partitioned we have three groups as shown below.

39/427/829

Adding 39, 427 and 829, we get 1295. Selecting the last three digits we get the desired address as 295. This method is also flexible like the 'Mid square' method and can be modified as desired. In case the key is string of characters, this can be converted into a number consisting of string of ASCII values of the characters.

For example, let the key value be "DOEACC". Then taking the ASCII values of the characters we get k = 687969656767. When partitioned we have four groups as shown in the following

687/969/656/767

Now adding these groups we get 3079. Thus, the generated address becomes 79.

Analysis for an ideal Hash Function

It was mentioned earlier that an ideal hash function should distribute "n" keys uniformly into "M" addresses. Such hash functions are called uniform hash functions. Therefore, "p", the probability that a particular address "a" ($0 \leq a \leq M-1$) will be generated for a key value, "k" is given by

$p = (1/M)$

Call the event of generating some address "a" by one trial of the hashing function to

be a "success". When the addresses for all keys have been generated then there have been "n" trials and each such trial is independent. The random variable, "X", denoting the number of success in "n" trials follows a "Binomial distribution". Therefore, p(x), the probability of "x" successes in "n" trials, is given by

$p(x) = ({}^{n}C_{x}) p^{x} (1 - p)^{n-x}$

Again, Binomial distribution can be approximated by Poisson's distribution for large "n". Thus, for large "n",

$p(x) = ((np)^{x} e^{-np})/ x!$

$= (\alpha^{x} e^{\alpha})/ x!$ ....(5)

where $\alpha = np = n/M$. "α" is also called the load factor.

Now, p(x) can also be interpreted as the probability that "x" keys will be mapped to a single address, "a", by an ideal hash function. It is interesting to note that this probability depends only on the load factor "α". This is why, the load factor is so important in hashing.

Using (5) as a working formula, one can find certain other attributes for an ideal hash function. One such attribute is "r", the expected number of distinct addresses that will be generated. Obviously, "r" is less than "n". An expression for "r" can be computed in the following manner. The probability that the address "a" will not be generated at all is given by p(0). Therefore, (1 – p(0)) gives the probability that some address "a" will be generated at least once. This gives

$r = (1 - p(0)).m$

$=(1-e^{-\alpha}).m$

Expressions for two other attributes may be computed by using "r". These are β, the expected number of home keys expressed as percentage of "n", and γ, the expected number of synonyms expressed as percentage of "n". The home keys are the ones which are hashed to a particular slot in the hash table for the first time. Synonyms are keys which are hashed to the same address giving rise to collision.

$\beta = 100 (r/n) = 100 (1 - e^{-\alpha}) / \alpha$

$\gamma = 100 - \beta = 100 (\alpha + e^{-\alpha} - 1) / \alpha$

The values of β and γ for various values of α are shown in the following table.

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
β	95.2	90.6	86.4	82.4	78.7	75.2	71.9	68.8	65.9	63.2
γ	4.8	9.4	13.6	17.6	21.3	24.8	28.1	31.2	34.1	36.8

It may be mentioned from the table that a value around 0.5 is a good choice for "α". A choice of $\alpha = 0.5$ signifies that if the number of entries of the hash table is twice the number of key values to be stored then the average number of synonyms or collisions may be restricted to the level of 21%.

Although this analysis is related directly to ideal hash functions only, it provides a method for testing the effectiveness of a given hash function. One can perform trial runs with the given hash function and can compare the number of synonyms obtained with the expected number of synonyms given by the corresponding equation obtained for an ideal hash function. A more elaborate and thorough test would be to find the distribution of the synonyms experimentally. It can now be compared with the theoretical distribution of the synonyms obtained for an ideal hash function.

9.9.2 Conflict resolution techniques

Every hash function is expected to generate the same address for distinct key values. The key values which are transformed to the same address are often referred to as synonyms. Therefore, one must know how to cope with such situations. Methods to deal with synonyms are known as collision resolution techniques. These techniques fall into two broad classes called open addressing and chaining. If a conflict occurs while inserting a key and if the conflict is resolved using "open addressing" methods then an empty position is found out within the hash table itself and the new key is inserted in that empty position. So, in "open addressing" methods, no space outside the hash table is used for storing the key values. In "chaining" methods, synonyms are placed in linked lists. Nodes of linked lists may be allocated from some space outside the hash table. There are several methods in each class. Some important and widely used methods from each class are described below.

Linear and Quadratic Probe

In this open addressing method, if the hashed address (typically an index to the hash table) is found to be already occupied by an element then the indices following the hashed index are sequentially examined to locate the first empty position. The concerned element is stored in the index so located. For the purpose of sequential examination, the hash table is considered to be circular. That is, the first index in the table is taken as the position next to the last index. Linear probe technique requires a means to identify whether a position in the hash table is empty or not.

In order to present algorithms, certain assumptions are made. As before, it is assumed that an element of the array consists only of the key field. The array, "A", is assumed to be of size "m". Another array, "T", is used to indicate whether an element of "A" is empty or not. It is assumed that if the i-th element of "T" is false then the i-th element of "A" is empty; otherwise, it is not empty. Algorithms to insert an element "k" and to search an element "k" using "linear probe" method are presented in the following.

```
int insertLinearProbe(int k)
{
    int i = 0, j;
```

```
    j = hash(k);
    do
    {
        if (T[j] == false) /* no key occupies A[j] */
        {
            A[j] = k;
            T[j] = true;
            return j;
        }
        i++;
        j = (j + i)%m;
    } while (i < m);
    return (–1);
}
```

Note that the existence of a function called "hash" has been assumed. This function returns an address in the range of "0" to (m-1) for a key. If A[j] ("j" is the index generated by the hash function) already contains some key then A[j+1], A[j+2], etc. are sequentially examined to find out an empty slot in "A". To produce the wrap-around effect, modulo-m additions are performed while incrementing "j". The function returns (-1) when no space is available for insertion. The "insertLinearProbe" function stores only one element at a time. It has to be repeatedly called to store more than one element. All the elements of the array, "T", should be initialized to "false" at the beginning.

The procedure for searching looks for the presence of a key value "k" in a hash table "A". The function returns an index of "A" where "k" is found. In the case of an unsuccessful search, the function returns (-1). The function is described next.

```
int searchLinearProbe(int k)
{
    int i = 0, j;
    j = hash(k);
    do
    {
        if (A[j] == k)
            return i;
        i++;
```

```
            j = (j + i)%m;
        } while ((T[j] == True) && (i < m));
        return –1;
}
```

Note that the search terminates either upon detection of the key or upon detection of an empty location (i.e. when "T" is false). This means that the number of keys to be stored in the array "A" must be less than "m" so that there will be at least one empty location to terminate the algorithm in the case of an unsuccessful search.

Linear probing is easy to implement but it suffers from "primary clustering". A collision at address "i" indicates that many keys are mapped to the same address "i". Linear probing would not distribute these keys in the hash table. Instead, all keys mapped to "i" will be clustered around that slot building up long runs of occupied slots. This would increase the search and insertion times. Thus, linear probing is not a good approximation to uniform hashing.

One way to reduce primary clustering is to use quadratic probing to resolve collision. Suppose, "k" is to be inserted in a hash table and h[k] = j where "h" is the hash function. If A[j] is already occupied then "k" has to be inserted at some other location. In quadratic probing, the locations "j", (j+1), (j+4), (j+9), ... are examined to find out the first empty slot where "k" may be inserted. Thus, the increment in this case is i^2 for i = 1, 2, 3, ...All additions are modulo-m where "m" is the size of the hash table. Although this technique reduces primary clustering, it does not ensure that all slots in the hash table would be examined to find out an empty slot. Thus, it is possible that a key could not be inserted even when the hash table is not full.

Double Hashing

Double hashing provides an easy way to virtually eliminate the problem of clustering. This method requires two hashing functions $f_1(k)$ and $f_2(k)$. The function $f_1(k)$ is used as a primary hash function. In case the address determined by $(1 + f_1(k))$, say "a", is already occupied by a key, the function $f_2(k)$ is evaluated. The second hash function is used to compute the increment to be added to the address obtained by the first hash function in case of collision. The resultant sum becomes the new address for search or insertion operation. Let "b" be the value of $f_2(k)$. The search for an empty location to house the new key is made successively at the addresses a + b, a + 2b, a + 3b etc. until one empty slot is found. As in the case of linear probe, the storage space is considered to be circular for the purpose of calculating addresses like a + b, a+ 2b, a+ 3b etc. Note that this increment was "1" in case of linear probe technique which gave rise to clustering. The second hash function is to be chosen carefully. While $f_1(k)$ generates an address in the range "0" to (m-1), $f_2(k)$ generates a value in the range "1" to (m-1) that is relatively prime to "m". In other words, there should not be any common factor between "m" and the values generated by

$f_2(k)$. In case "m" is a prime number, any value between "1" to (m-1) could be generated by $f_2(k)$.

The program to insert a new key "k" using double hashing is shown below. The assumptions made in the linear probe algorithm are also used here. Two hash functions "hash1" and "hash2" are assumed to be available.

```
int storeDoubleHashing(int k)
{
    int j, k, u, i = 0;
    j = hash1(k);
    u = hash2(k);
    do
    {
            if (T[j] == false)
            {
                    A[j] = k;
                    T[j] = true;
                    return i;
            }
            i++;
            j = (j + i*u)%m;
    } while (i < m);
    return –1;
}
```

The search for an element in the array should follow the same path used to store the element. The search algorithm is simple and left as an exercise.

It can be mentioned here that the division method can be used for hash functions $f_1(k)$ and $f_2(k)$. The divisor should be a prime number. Then, $f_1(k)$ and $f_2(k)$ are given by

$$f_1(k) = k \bmod m$$

$$f_2(k) = | k \bmod (m - 1)| + 1$$

However, if "m" is a prime number then (m–1) is even. To avoid division method with an even divisor, $f_2(k)$ can be taken as

$$f_2(k) = | k \bmod (m - 2)| + 1$$

This will generate values in the range 1 to (m–2). But, since these values are relatively prime to "m", there will not be any problem.

In all open addressing methods, deletion of a key is not straight-forward. If there are many keys, which are hashed to the same address then removal of a key from the middle implies that all synonyms must be packed. Otherwise, handling insertion and search operations would become difficult. Moreover, packing is a time-consuming operation. Another approach for deletion is that when a key is deleted the corresponding entry in the hash table is marked with a special symbol. But this approach leads to inefficient search operations.

Separate Chaining

In this method, linked lists are maintained to store all elements that hash to the same address. Let the hash table "A" contains "m" entries or slots. Each slot contains a pointer to a linked list and the list stores the elements hashed to this slot. Each node in such a list is a self-referential structure containing a key value and a pointer to the same structure. A typical "C" declaration for a node may be done in the following manner.

```
typedef struct node
{
   int key;
   struct node *next;
}node;
```

Then, the hash table "A" containing "m" pointers may be defined as follows.

```
node *A[50];
```

It is assumed that the maximum value of "m" is 50. This declaration implies that each entry in the array is a pointer to a list. For all "i", A[i] should be initialized to NULL. While inserting a key "k" using a hash function "h", first h(k) is computed. Let, h(k) be "i", $0 \le i \le m-1$. Then "k" is inserted to the list pointed to by A[i]. Therefore, each slot in the hash table maintains a separate chain (i.e., a linked list) to store all elements hashed to that slot. This is why, this method is known as "separate chaining" method. Also note that in this method, "k" is not stored in the table "A". Instead, a node is created to store "k" and that node is added to the list pointed to by A[i].

As an example, let us consider the list

L=(351, 493, 251, 71, 706, 146)

Suppose that an array A[0:6] is used as the hash table. Let the hash function used be defined as h(k) = k mod 7. The hashed addresses generated for the elements in "L" are 1, 3, 6, 1, 6, 6 respectively. If the keys are inserted in the order they appear in the list then the hash table along with the chains will be as shown in Figure 9.10.

their original positions because they are loaded later. In chaining method, such an element is also placed in the same chain in which the synonyms occupying its original position appears. The chains are, therefore, really coalesced chains containing synonyms as well as elements deprived of their home positions for the above mentioned reasons. This does not create any problem except that the chains may become longer. However, the average length of such chains is not very large.

In order to present the algorithm that an element using this technique, assume that an element consists only of the key field. The keys are stored in the array A[0: m-1] while the links are stored in link[0: m-1]. Null links are denoted by zeroes. A negative value in the link field denotes that the position is empty. In the beginning all the link fields are assumed to contain negative values (say –1).

When a key is to be inserted and a collision takes place then the coalesced chain is traversed first to find the last node of the chain. The addresses following this last node is then sequentially searched to locate an empty node. Satisfy yourself that this is actually the first empty node following the hashed address. One could also search in a straight forward manner (as done in the insert algorithm for linear probe). However, traversal of the chain is faster and this also enables one to insert the new node at the end of the chain without further examination.

9.9.3 Analysis of Collision resolution techniques

Statistical analysis of the collision resolution techniques has been carried out to find out their average case performances. The figures of merit that are chosen include the number of probes required for successful and unsuccessful search operations. There are other metrics to measure performance of the collision resolution techniques. For example, the complexity of inserting a new record or deleting an existing record is also very important. However, the search operation appears to be more important than insertion or deletion because hashing is generally used for lists that are searched frequently. Suppose that a hash table of "m" slots is used to store "n" keys. Let C_u (n, m) and C_s (n, m) denote the average number of probes required for an unsuccessful and for a successful search respectively. It is not easy to find even approximate expressions for C_u (n, m) and C_s (n, m). In some cases the calculations are quite involved. In this section, the final expressions for different metrics of several collision resolution techniques will be mostly quoted omitting the calculations for those results.

Open addressing with random probes

Given an open address hash table using an increment function that is random with a load factor α ($\alpha < 1$), the expected number of probes in an unsuccessful search, C_u (n, m), is at most $1/(1 - \alpha)$ assuming uniform hashing. Moreover, the expected number of probes in a successful search, C_s (n, m), is at most $(1/\alpha)\log(1/(1 - \alpha))$. The expression for C_u (n, m) is easy to obtain. If the load factor is "α" then the

probability that the first probe hits an occupied slot is "α". Therefore, the probability that the first probe hits an unoccupied slot is (1 - α). Thus, the probability that an unsuccessful search makes exactly one probe is (1 - α). If the slot in the first probe is occupied then the next slot (i.e., the slot located at an address equal to the address of the first slot plus some increment) needs to be examined. Assuming that the increments generated by the increment function are random and independent to one another, the probability that the slot in the second probe is unoccupied becomes (1 - α). The probability that the number of probes in an unsuccessful search is "2" may be given by the probability that the first probe hits an occupied slot multiplied by the probability that the second probe hits an unoccupied slot. Generalizing,

probability(number of probes in an unsuccessful search = k)

$= \alpha^{k-1} (1 - \alpha)$

Therefore, $C_u(n, m)$, the expected number of probes for an unsuccessful search is given by

$\Sigma_{k=1,\infty} k \alpha^{k-1} (1 - \alpha)$

$= (1 - \alpha)/(1 - \alpha)^2$

$= 1/(1 - \alpha)$

An element is inserted only if there is vacant space in the table (i.e., $\alpha < 1$). Insertion operation requires an unsuccessful search followed by placing the key in the first empty slot. Thus, inserting an element into an open address hash table with a load factor a requires at most $1/(1 - \alpha)$ probes on the average.

Linear probe

Analysis of this method shows that

$$C_u(n,m) = \tfrac{1}{2} (1+ 1/(1 - \alpha)^2)$$

$$C_s (n, m) = \tfrac{1}{2} (1+1/(1 - \alpha))$$

Where α is the load factor (i.e. $\alpha = n/m$)

Separate Chaining

Under the assumption of simple uniform hashing, any key "k" is equally likely to be hashed to any of the "m" slots. The average time for an unsuccessful search for a key "k" is the average time to search all elements in one of the "m" lists. Since a total of "n" keys are stored in "m" slots, the average number of elements in any slot is the load factor (n/m). Thus, the expected number of elements examined for an unsuccessful search is α and the total time required for this purpose (including the time to hash) is $O(1 + \alpha)$

The expected number of elements examined during a successful search for a key "k"is one more than the average number of elements examined when "k" had been

inserted. Let "k" be the i-th element to be inserted. Before that insertion, there were (i-1) elements in the hash table. So, there were an average of (i-1)/m number of elements in each slot of the hash table. The expected number of elements examined to insert "k' must be (i-1)/m. Hence, the expected number of elements examined to search for "k" is 1+ (i-1)/m. But the value of "i" could be any thing from "1" to "n" Thus the expected number of elements examined for a successful search is given by

$(1/n)\Sigma_{i=1,n}(1+(i-1)/m))$

$= 1 + (1/nm)\ \Sigma_{i=1,n}(i-1)$

$= 1 + (1/nm)\ ((n-1)n/2)$

$= 1 + \alpha/2 - 1/2m$

Thus, the total time required for a successful search is $O(1 + \alpha)$.

In fact, if "α" is nearly equal to "1" then insertion operation takes O(1) time in the worst case and deletion operation also takes O(1) worst-case time provided the lists are doubly linked. Thus, all three operations can be supported in O(1) time on the average.

Coalesced Chaining

The analysis of coalesced chaining method is too involved. We shall, therefore, simply mention the results. In this case

$$C_u(n, m) = 1 + (1/4).(e^{2\alpha}\alpha - 1 - 2\alpha)$$

$$C_s(n, m) = 1 + (1/(8\alpha)).(e^{2\alpha}\alpha - 1 - 2\alpha) + (1/4).\alpha$$

Comparison of the collision resolution techniques

The expressions of the two metrics corresponding to various collision resolution techniques have been evaluated for some selected values of "α". The results are presented in the following table.

Load Factor (α)		0.5	0.6	0.7	0.8	0.9
Linear Probe	Unsuccessful	2.50	3.625	6.055	13.00	50.50
	Successful	1.50	1.75	2.167	3.00	5.50
Random Probe	Unsuccessful	2.00	2.50	3.33	5.00	10.00
	Successful	1.40	1.53	1.72	2.00	2.60
Separate Chaining	Unsuccessful	1.25	1.30	1.35	1.40	1.45
	Successful	0.50	0.60	0.70	0.80	0.90