Chapte 8

Sorting

8.1 INTRODUCTION

This chapter presents several algorithms and C programs that solve the classic problem called **sorting**. The sorting problem has an input sequence of n numbers typically in an array $a[1{:}n]$ and the output is a sorted sequence either in ascending order or descending order. For instance,

input: $[a_1, a_2, a_3, \ldots, a_n]$
output: $a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_n$

In practice, the numbers to be sorted are distinct values called as **keys**. Each key may be associated with a set of attributes carried around the key. For example, we may wish to sort a set of employee records consisting of employee *id* (a unique identifier), name, address, salary, etc. Here, if we sort the records based on the employee *id* then we can assume that the other fields are also sorted. How? Each key will point to the remaining data of the record and so when we sort the keys (or *id* numbers) the pointers would still point to the rest of the data. This is to minimize the unnecessary data movement.

We shall discuss the following sorting methods and their time and space efficiencies.

- Bubble sort
- Quick sort
- Merge sort
- Heap sort
- Selection sort
- Insertion sort

Consider another set of algorithms with computing time $10^4 n$ and n^2 respectively. You can not immediately say that the first one is superior, because for all values of $n < 10^4$, the second algorithm is faster.

That is, $10^4 \times 10 = 10^5$ versus 10^2 (with $n = 10$)

So, we cannot decide that, which of the two algorithms is better unless we know some thing about the constants associated with their basic operations.

This section gives several asymptotic notations for analyzing the algorithms. We will study, about O(Big Oh), Ω(Omega), and Θ(Theta) notations briefly.

8.3.1 Big Oh Notation (O)

We use O-notation to give an upper bound on a function $f(n)$, to with in a constant factor. The upper bound on $f(n)$ indicates that the function $f(n)$ will be the worst-case that it does not consume more than this computing time.

Definition: $f(n) = O(g(n))$ (read as "f of n is equal to the big Oh of g of n" such that there exists two positive constants c and n_0 with the constraint that $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$.

We can also say that "$f(n)$ is of the order of $g(n)$" and $f(n)$ grows no more than $g(n)$. When we say that an algorithm has computing time $O(g(n))$ we mean that if the algorithm is run on some computer on the same type of data but for increasing values of n, the resulting time will always be less then $c\,g(n)$.

The most common computing times fall under one of the categories as shown in Table 8.1.

Table 8.1 Asymptotic functions

Sl. No.	Function	Name
1	O(1)	Constant
2	O(log n)	Logarithmic
3	O(n)	Linear
4	O(n log n)	n log n
5	O(n^2)	Quadratic
6	O(n^3)	Cubic
7	O(2^n)	exponential

The seven functions shown in Table 8.1 is in the increasing order of computing time that is,

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

We shall understand how these functions grow with the increase in values of n. The Table 8.2 and the plot shown in Figure 8.1 illustrate the computing times for six of the typical functions grow for different values of n.

Table 8.2 Values for computing functions

n	$\log n$	$n \log n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296

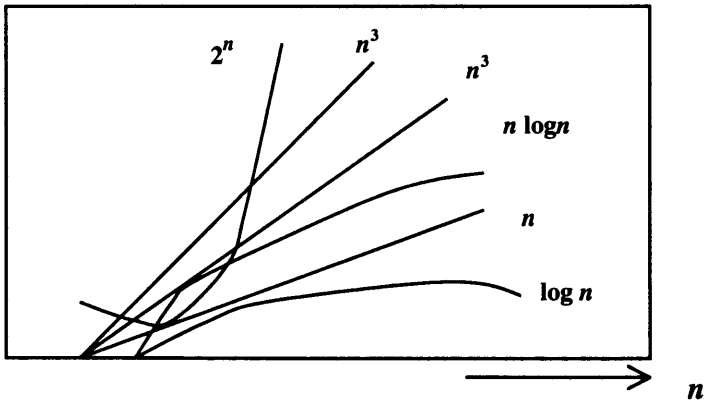


Fig. 8.1 Growth rate of common functions, $c = 1$

For large values of n, the function 2^n grows very rapidly and some times it is impractical also. We will consider few examples that will make you understand further the linear, quadratic, exponential function better. The $g(n)$ functions will be derived for a given $f(n)$ based upon the definition of big Oh notation.

8.3.2 Omega Notation (Ω)

So far we have seen O-notation, which described the algorithms upper bound performance. Some times we wish to find the lower bound behavior of $f(n)$ also. The lower bound would imply that below this time the algorithm can not perform better. The algorithm will take at least this much of time (lower bound). It is represented mathematically using the notation Ω.

Definition: $f(n) = \Omega(g(n))$ (read as "f of n is equal to omega of g of n") iff there exists positive constants c and n_0 such that for all $n > n_0$, $|f(n)| \geq c\,|g(n)|$

An example

To understand the meaning of lower bound, consider the problem of finding maximum (or minimum) element in a given array of n elements. To do this job, you must iterate through the entire loop, because with out comparing the first element with all (n-1) elements you can not obtain maximum or minimum. Therefore, the lower bound for this problem is n, that is $\Omega(n)$. Also the upper bound is n, that is $O(n)$. You don't need more than n steps for the computing time.

However, consider the case of linear search where the key element may be found in the first position itself and so the time-complexity is 1 i.e., $\Omega(1)$. However, if the key is at the last position then the upper bound (or worst-case) is $O(n)$. Similarly, for not found case, it is $\Omega(n)$ and $O(n)$.

8.3.3 Theta Notation (Θ)

For some functions the lower and upper bound may be same i.e., Ω and O will have the same function. For example, finding the minimum (or maximum) in a given array takes the computing time $\Omega(n)$ and $O(n)$. We have a special notation to denote for functions having the same time-complexity for lower and upper found and this notation is called as the **theta notation** Θ.

Definition: $f(n) = \Theta(g(n))$ iff there exist positive constants c_1 and c_2 such that for all $n > n_0$, $c_1 | g(n) | \leq | f(n) | \leq c_2 | g(n) |$

To explain the theta notation we shall try to prove that $\frac{1}{2} n^2 - 3n = \Theta(n^2)$ by using the formal definition of Θ-notation.

8.4 BUBBLE SORT

Out of all the sorting methods listed in Section 8.1, the simplest one is the **Bubble sort** algorithm. Bubble sort works on the **bubbling** strategy. At the end of each *pass* the largest element is moved to the right most position of the array, assuming that we wish to sort in an ascending order.

The technique employed in bubbling is to compare the adjacent elements starting from the beginning of the array. If the element on the left is greater than the element on the right, then these two elements are swapped. If the left-side element is less than the right-side, then no swapping is done. Since, pairs of elements are taken at any point of time, the number of iterations required for the bubbling is $n - 1$ (n is the number of elements to be sorted).

Let us try an input sequence with $n = 6$ to shown the bubbling of largest element at (n-1) the position.

$a[0:5] = [7, 5, 9, 2, 3, 1]$

$= [5, 7, 9, 2, 3, 1]$ {7 and 5 - swapped}

$= [5, 7, 2, 9, 3, 1]$ {7 and 9 - no swap, 9 and 2 - swapped}

$= [5, 7, 2, 3, 9, 1]$ {9 and 3 - swapped}

$= [5, 7, 2, 3, 1, 9]$ {9 and 1 - swapped}

You can see that largest element 9 occupies the 5th position of the array and we need to sort now the remaining $(n - 1)$ elements using the same technique. Algorithm 8.1 shows the bubbling action.

Algorithm 8.1

```
Algorithm Bubble (a, n)
{
     // a[0:n-1] : array to be sorted
     // n: number of elements
     for(i = 0 to n-1) do
     {
          if(a[i] > a[i+1])
               swap(a[i], a[i+1]);
     }
}
```

When the first two elements 7 and 5 are compared, the left element is greater than the right element and so they are swapped. Next, 7 and 9 are compared and no swap is required and so on:

The algorithm Bubble is called $n - 1$ times to bring the largest element to the right in every pass. We will show the remaining passes by continuing the same example array considered earlier.

Pass 2: $[5, 7, 2, 3, 1, 9] \Rightarrow [5, 2, 3, 1, 7, 9]$

Pass 3: $[5, 2, 3, 1, 7, 9] \Rightarrow [2, 3, 1, 5, 7, 9]$

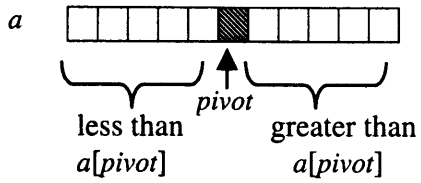
Pass 4: $[2, 3, 1, 5, 7, 9] \Rightarrow [2, 1, 3, 5, 7, 9]$

Pass 5: $[2, 1, 3, 5, 7, 9] \Rightarrow [1, 2, 3, 5, 7, 9]$

Sorted sequence is $a = [1, 2, 3, 5, 7, 9]$

The input to Pass 2 is [5, 7, 2, 3, 1, 9] which is the output of Pass 1, and it yields [5, 2, 3, 1, 7, 9] and so on. Though we have shown all the elements in each pass, the bubbling algorithm need not do its task on all elements but $n - 1$, $n - 2$, $n - 3$, . . .,1. Program 8.1 shows the C code for bubble sort algorithm.

long as $i < j$. After all the elements are compared you would notice that the elements that are left of pivot will be less than the pivot element and the elements to the right of the pivot will be greater than the pivot element.



The left segment and right segment are sorted by calling the function `QuickSort()` recursively using the method just described.

8.5.1 An Example

Let us now consider a data set and see how quick sort works. All the steps are shown in Table 8.3.

Table 8.3 Trace of Quick Sort

0	1	2	3	4	5	6	7	*i*	*j*
25	10	72	18	40	11	32	9	1	7
25	10	72	18	40	11	32	9	2	7
25	10	9	18	40	11	32	72	2	7
25	10	9	18	40	11	32	72	3	7
25	10	9	18	40	11	32	72	4	6
25	10	9	18	40	11	32	72	4	5
25	10	9	18	11	40	32	72	4	5
25	10	9	18	11	40	32	72	5	4
11	10	9	18	**25**	40	32	72		

Segment 1 **Segment 2**

Now the Segment 1 and Segment 2 are recursively sorted.

Segment 1

0	1	2	3	*i*	*j*
11	10	9	18	1	4
11	10	9	18	2	3
11	10	9	18	2	2
9	10	**11**	18		

The sorted sequence of Segment 1 = [9, 10, 11, 18]

Segment 2

5	6	7	i	j
40	32	72	6	8
40	32	72	7	7
40	32	72	7	6
32	**40**	72		

The sorted sequence of Segment 2 is = [32, 40, 72]
Now the final list, which is fully sorted, is given below

[9, 10, 11, 18, 25, 32, 40, 72]

The algorithm pseudo code is shown in Figure 3.9 and the Program 8.2 shows the source code for the same.

Algorithm QuickSort

```
Algorithm QuickSort(a, lb, ub)
{
    // i and j are local variable
    if (lb < ub)
    {
        q = Partition(a, lb, ub);
        // partition the given list in to two segments and place
        // the pivot element at appropriate position
        QuickSort(a, lb, q) ;     // sort left segment
        QuickSort(a, q+1, ub);  // sort right segment.
    }
}
```

Fig. 8.2 Pseudo code for QuickSort()

Program 8.2
QuickSort

```
void QuickSort(int a[], int lb, int ub)
{
    int i, j, temp, key, flag = 1;
    if (lb < ub)
    {
        i = lb; j = ub + 1;
        key = a[lb]; /* mark first element as pivot */
        while (flag)
        {
            i++;
```

8.6.1 The Method

Given a sequence of n elements A[0], A[1], ...,A[n-1] are splitted in to two sub files

A(0), A(1),.....,A([n/2]) and A([n/2]+1), ...,A(n-1)

Each subfile is separately sorted and the resulting sequences are merged to produce a single sorted sequence. The Figure 8.4 shows the pseudo code for *Mergesort()* in which the list is divided using recursion.

Algorithm MergeSort

```
Algorithm MergeSort(A, low, high)
{
    // A [0 : n-1] array to be sorted.
    // low and high are the indices for the sorting of A.
    if ( low < high)          // at least two elements.
    {
        mid = (low + high) / 2;
        Mergesort (A, low, mid);
        Mergesort (A, mid+1, high);
        Merge (low, mid, high);
    }
}
```

Fig. 8.4 Pseudo code for MergeSort()

The function *Merge()* will be presented shortly and meanwhile, we shall show the concept of *dividing* with an example. Let, $n = 10$.

	0	1	2	3	4	5	6	7	8	9
A	7,	5,	9,	2,	3,	1,	4,	6,	10,	8

```
[7, 5, 9, 2, 3]          [1, 4, 6, 10, 8]
[7, 5, 9] [2, 3]         [1, 4, 6] [10, 8]
[7, 5] [9] [2] [3]       [1, 4] [6] [10] [8]
[7] [5] [9] [2] [3]      [1] [4] [6] [10] [8]
```

Merging

```
[5, 7]  [9]  [2, 3]      [1, 4] [6] [8, 10]
[5, 7, 9]  [2, 3]        [1, 4, 6]  [8, 10]
[2, 3, 5, 7, 9]          [1, 4, 6, 8, 10]
```

A [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

The splitting is done until the size of the subfile becomes 1. Now we are ready to present the *Merge()* algorithm and is shown in Figure 8.5.

Algorithm for Merge()

```
Algorithm Merge(A, low, mid, high)
{
        // B is a temporary array
        // B (low : high)
        // i, j, h are local variables

        n = low;
        i = low;            // i is for the first segment
        j = mid + 1;        // j is for the second segment

        while (h <= mid && j <= high )          // subfiles not exhausted yet
        {
                if (A[h] <= A[j])
                {
                        B[i] = A[h]; h++;
                }
                else
                {
                        B[i] = A[j]; j++;
                }
                i++;
        }
        if (h > mid)        // first subfile exhausted –copy remaking elements.
                for (k = j; k <= high; k++, i++)
                        B[i] = A[k];
        else                // second subfile exhausted-copy remaining elements
                for (k = h; k <= mid; k++, i++)
                        B[i] = a[k];

        // Copy B to A
        for (k = low; k < high ; k++)
                A[k] = B[k];
}
```

Fig. 8.5 Pseudo code for Merge()

Let us show the working of *MergeSort()* with a recursive tree as shown in Figure 8.6 Each node, in the figure, represents a recursive call and in every node the *low* and *mid*

values are shown. You would also see the elemental value(s) written adjacent to each node.

The program corresponding to algorithm *MergeSort()* and *Merge()* is shown in Program 8.3 and it can be invoked as,

```
MergeSort(a, 0, n-1);
```

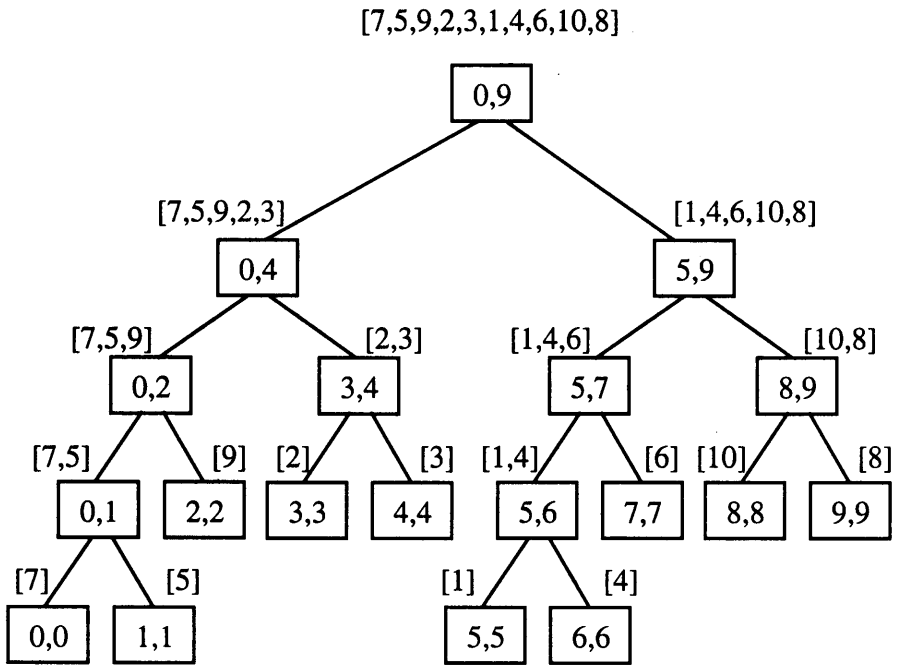


Fig. 8.6 Recursive tree for Mergesort()

Program 8.3
Merge Sort

```
void MergeSort(int a[], int low, int high)
{
     int mid;
     if (low < high)
     {
          mid = (low + high)/2;
          MergeSort(a, low, mid);
          MergeSort(a, mid+1, high);
          Merge(a, low, mid, high);
     }
}
```

```
void Merge(int a[], int low, int mid, int high)
{
     int i, h, j, k, b[MAX];

     h = i = low;
     j= mid + 1;

     while(h <= mid && j <= high)
          if(a[h] < a[j])
               b[i++]= a[h++];
          else
               b[i++] =a[j++];

          if (h > mid)
               for(k = j; k <= high; k++)
                    b[i++] = a[k];
          else
               for (k = h; k <= mid; k++)
                    b[i++] = a[k];

          for (k = low; k <= high; k++)
                    a[k] = b[k];
}
```

8.7 HEAP SORT

As we have already discussed in Section 7.10.4 about the heap structure, the `MaxDelete()` function can be used to sort a list of n-elements. Recall that `MaxDelete()` returns the maximum element out of n numbers. This largest element, say `e`, can be saved in the last position of the array. Call `MaxDelete()` again, but this time by sending $(n - 1)$ element so that now it would return the largest of the these $(n - 1)$ elements. Save this in the last but one position of the array and so on, until $n = 1$. The final array `a[1:n]` would contain the sorted elements in ascending order.

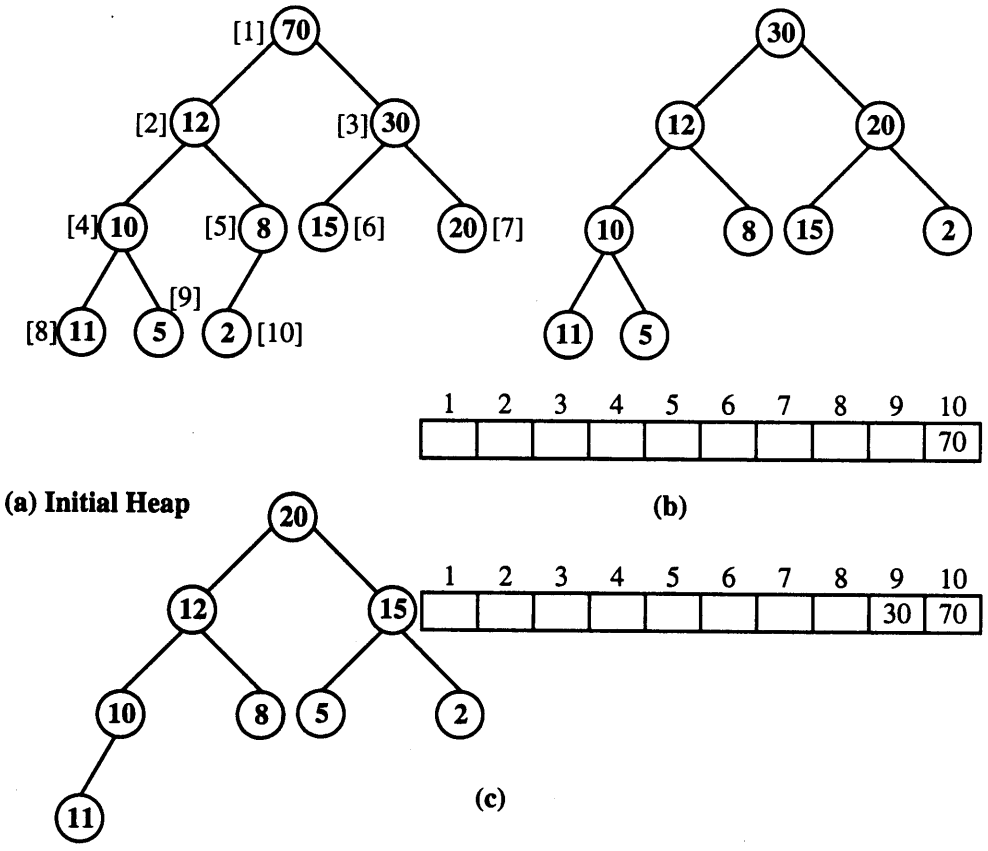
The Program 8.4 is the result of the above strategy called as **heap sort**. The program uses the `InitializeHeap()` function stored in `Init.c` file and similarly `MaxDelete()` is assumed to be stored in file `Mdel.c`.

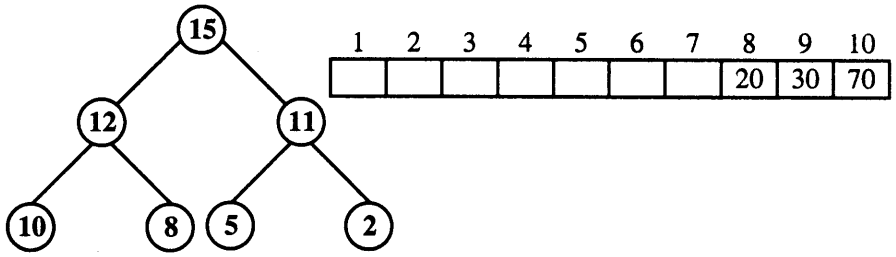
Let us show the sequence of steps in arriving at the final sorted list in array a (see Figure 8.1) consider the initial array a[1:10] = [70, 12, 30, 10, 8, 15, 20, 11, 5, 2] which is a heap. In general, the function `InitializeHeap()` should be called first after reading the array to make the array into a heap.

Program 8.4
Heap Sort

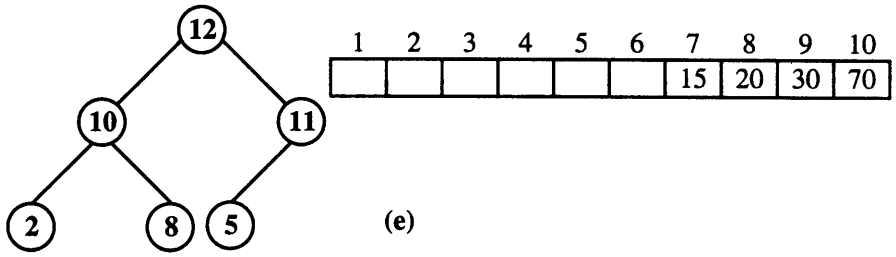
```
#include <stdio.h>
#define MAX  100
#include "d:\ds-book\c-prog~1\Init.c"
#include "d:\ds-book\c-prog~1\Mdel.c"
void HeapSort(int [], int);

void HeapSort (int a [], int n)
{
     int i;
     InitializeHeap(a, n); /* heap a */
     for (i = n; i >= 1; i--)
            a[i] = MaxDelete(a, &n);
}
```

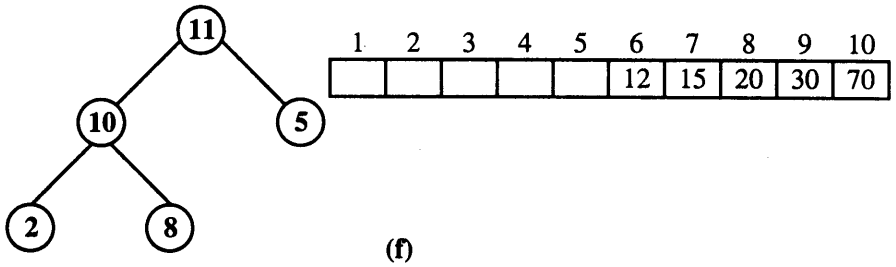



15
12
11
10
8
5
2
1 2 3 4 5 6 7 8 9 10
20 30 70

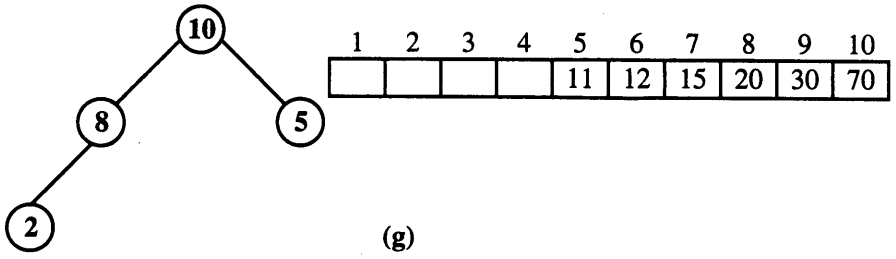
(d)


12
10
11
2
8
5
1 2 3 4 5 6 7 8 9 10
15 20 30 70

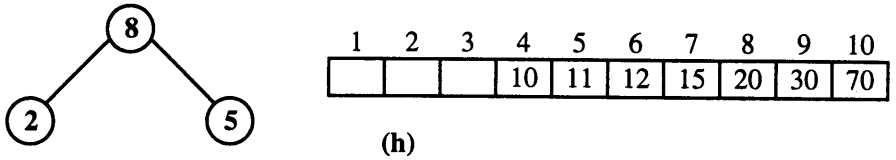
(e)


11
10
5
2
8
1 2 3 4 5 6 7 8 9 10
12 15 20 30 70

(f)


10
8
5
2
1 2 3 4 5 6 7 8 9 10
11 12 15 20 30 70

(g)


8
2
5
1 2 3 4 5 6 7 8 9 10
10 11 12 15 20 30 70

(h)

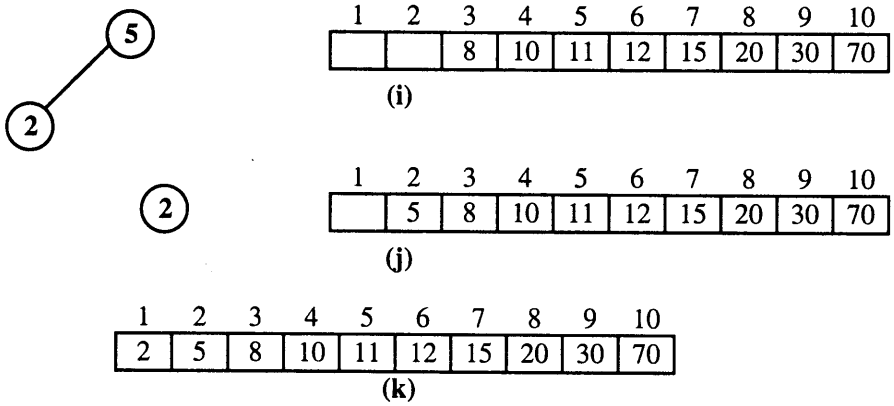


Fig. 8.1 Snapshot of Heap Sort

The initialization of the given array into a heap takes $\Theta(n)$ time and `MaxDelete()` take O(log n) and hence the over all complexity of heap sort is O(n log n)

8.8 SELECTION SORT

Selection sort uses a different method compared to those of the methods discussed already. In a given array $a[0{:}n\text{-}1]$, determine the largest element and push it to $a[n - 1]$. Next, out of the remaining elements find the largest and move it to $a[n - 2]$ and so on. We need an algorithm to find the position of the largest element in each pass and in every pass the number of elements would get reduced. Compare this method with bubble sort, where bubbling concept moved the largest in each pass to the (n - 1), (n - 2), etc., positions. Instead of bubbling we use maximum element finding in selection sort (see Figure 8.7).

0	1	2	3	4	5	6	7	8	9	*j*	(*pass*-1)
[70	12	30	10	8	15	20	11	5	2]	0	9
[2	12	30	10	8	15	20	11	5]	70	2	8
[2	12	5	10	8	15	20	11]	30	70	6	7
[2	12	5	10	8	15	11]	20	30	70	5	6
[2	12	5	10	8	11]	15	20	30	70	1	5
[2	11	5	10	8]	12	15	20	30	70	1	4
[2	8	5	10]	11	12	15	20	30	70	3	3
[2	8	5]	10	11	12	15	20	30	70	1	2
[2	5]	8	10	11	12	15	20	30	70	1	1
[2]	5	8	10	11	12	15	20	30	70	0	0
2	5	8	10	11	12	15	20	30	70		

Figure 8.7 Snapshot of Selection Sort

The elements in the angle brackets indicate the current size of the elements to be considered to pick the largest. Variable *j* is the position of the maximum element in the array. And (*pass*-1) is the position to which *a*[*j*] is to be swapped. Program 8.5 shows the resulting C to sort array of n elements using selection sort.

Program 8.5
Selection Sort

```
void SelectionSort (int a[], int n)
{
     /* sort array a[0:n-1] in ascending order */
     int i, pass, t;
     int j;
     /* iterate through the array for n-1 passes */
     for (pass = n; pass > 1; pass--)
     {
          j = Max(a, pass); /* position of the largest */
          /* swap this with a[n-1], a[n-2], ...a[0] */
          t = a[j];
          a[j] = a[pass-1];
          a[pass-1] = t;
     }
}
```

The complexity of selection sort program is same as bubble sort i.e., $O(n^2)$.

8.9 INSERTION SORT

We start with an array in which the first element alone is considered. With just one element with out any ambiguity, it is a sorted sequence. By inserting the second element into this one element array, we get a sorted sequence of size 2. Similarly, each subsequent element being added retains the array in a partially sorted array. When we finally insert the last element, we get the fully sorted array.

The above strategy is shown in Figure 8.8, considering the array as *a*[0:9] = [70, 12, 30, 10, 8, 15, 20, 11, 5, 2].

The first element 70 in the input array is the sorted sequence of one element (shown in brackets[ ]). We start picking up the second element 12 and check for its appropriate position for insertion. A looping structure is initiated from its previous element and up to 0th position. In this process, we move the elements to the right side by one position so that once we find the place for the element we can insert it.

0	1	2	3	4	5	6	7	8	9
[70]	12	30	10	8	15	20	11	5	2
[12	70]	30	10	8	15	20	11	5	2
[12	30	70]	10	8	15	20	11	5	2
[10	12	30	70]	8	15	20	11	5	2
[8	10	12	30	70]	15	20	11	5	2
[8	10	12	15	30	70]	20	11	5	2
[8	10	12	15	20	30	70]	11	5	2
[8	10	11	12	15	20	30	70]	5	2
[5	8	10	11	12	15	20	30	70]	2
[2	5	8	10	11	12	15	20	30	70]

Fig. 8.8 Snapshot of Insertion Sort

For example, consider the second line where the partially sorted array is [12, 70] -just two elements. Element 30 is to be inserted in this sorted sequence. Since 30 is less than 70 we move to the third position (assume that 30 is saved in a temporary variable in the beginning itself) and then check with 12. Now it is the position for 30, hence insert it. The current partially sorted sequence is [12, 30, 70]. Program 8.6 is the code for insertion sort.

Program 8.6
Insertion Sort

```
void  InsertionSort(int a[], int n)
{
     int j,p,x,k;
     for (j = 1; j < n; j++)
     {
          k = a[j];
          for (p = j-1; (p >= 0  &&  k < a[p]); p--)
               a[p+1] = a[p];
          a[p+1] = k;
     }
}
```

The outer `for` loop starts from the 2nd element on wards as the first element is already in the sorted order (one element). The inner `for` loop index `p` scans from the current element to be inserted (`k`) until it finds its place. The statement `a[p + 1] = k;` finally inserts the key element. Shifting of the elements from the current position `p` to its right is done using

```
a[p + 1] = a[p];
```

In the best case the number of comparisons is $(n - 1)$ and in the worst case it is $(n - 1)\, n / 2$.

8.10 SHELL SORT

Selection sort moves the elements very efficiently but does many redundant comparisons. Insertion sort does minimum number of comparisons (best case), but it is inefficient in moving elements only one place at a time. Because, it compares only adjacent elements it is poor in terms of moving the elements.

Our aim is to devise an algorithm removing the drawbacks of both selection sort and insertion sort. For implementing faster data movement and retaining efficient comparisons of insertion sort a new sorting algorithm, called as **Shell Sort** was proposed by D.L.Shell in 1959. It is also sometimes called as **diminishing increment sort.**

We compare the elements at a particular distance, d and reduce this distance until it becomes 1. For example, if we could start with $d = 5$ and sort all elements which are 5 elements apart.

Then considering $d = 3$ sort all elements that are 3 elements apart. Finally, when $d = 1$, all the elements that are apart by 1 are sorted. The shell sort would become same as insertion sort when $d = 1$.

Let us try this idea with a sample array $a[0:9]$ and is shown in Figure 8.9.

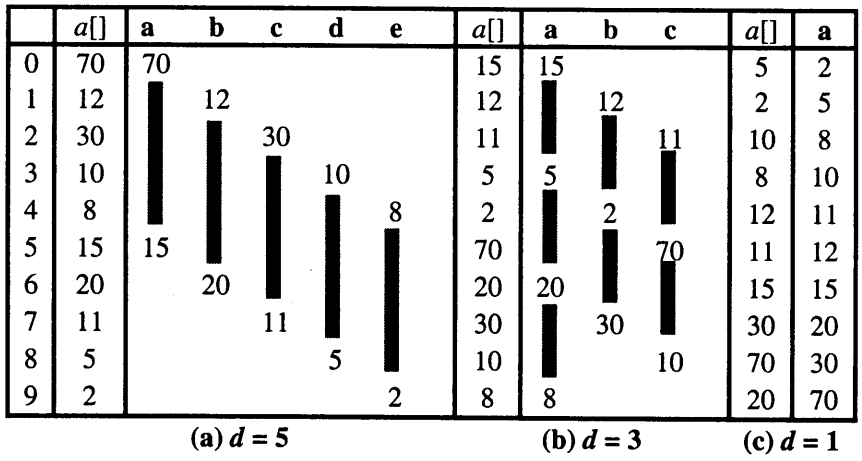


Fig. 8.9 Snapshot of Shell Sort

Figure 8.4(a) shows the first pass with $d = 5$ and the input array is shown in a vertical fashion, $a[]$. Adjacent columns (**a**), (**b**), (**c**), (**d**) and (**e**) are the sublists for various values of d.

When $d = 5$ we get 5 unsorted sublists and they are,

[70, 15], [12, 20], [30, 11], [10, 5], [8, 2].

Each sublist is sorted using an insertion sort and they are put back into the array. Figure 8.4(b) shows the next pass, with $d = 3$ and the input for this is nothing but the output of Figure 8.9(a). Again, all elements that are apart by distance 3 form sublists as given below:

[15, 5, 20, 8], [12, 2, 30], and [11, 70, 10]

These are again sorted with their distance maintained. For example, the first sublist [15, 5, 20, 8] that are at positions [0, 3, 6, 9] are sorted as [5, 8, 15, 20] and occupy again the same positions [0, 3, 6, 9] respectively.

Figure 8.4(c) shows the final pass i.e., $d = 1$. All elements will be sorted in this pass as the adjacent elements are compared.

Implementation

As suggested by Knuth, we can use the modified insertion sort for implementing Shell sort. The Program 8.7 is the result of that, where the modified `InsertionSort()` accepts four parameters instead of just two (see Section 8.9).

Program 8.7
Shell Sort

```
void ShellSort(int a[], int n)
{
     int i, inc;
     inc = 12;
     do
     {
          inc = inc / 3 + 1; /* consider d = 5, 3, 1 */
          for (i = 0; i < inc; i++)
               InsertionSort(i, inc, a, n);
     } while (inc > 1);
}

void  InsertionSort(int start, int step, int a[], int n)
{
     int j, p, x, k;
     for (j = start; j < n; j+=step)
     {
          k = a[j+step];
          for (p = j; (p>=start  &&  k<a[p]); p-=step)
               a[p+step] = a[p];
          a[p+step] = k;
     }
}
```

This is because that we can not compare adjacent elements, instead compare elements that are apart by a step (we have used letter *d* in the Figure). Another important change required is in the starting of each sublist. In the ordinary insertion sort the beginning location is always zero. However, in shell sort the beginning location depends upon the sublist. Each sublist starts at some specific location and ends at some other location.

Function `ShellSort()` sends these information and also diminishes the value of *d* in each iteration using,

```
inc = inc / 3 + 1;
```

The +1 ensures that the last past will always have a value 1 for `inc`. The program uses starting value for `inc` as 12 so that we get 5, 3, 1 as subsequent values for `inc`. After a large empirical study, shell sort showed the number of moves for a large value of *n*, is in the range $n^{1.25}$ to $n^{1.6}$.

8.11 ADDRESS CALCULATION SORT

In this method, the most significant digit from each element is extracted and this decides the address of the linked list for insertion. Let us make things clear - totally ten linked lists are maintained one for each decimal digit 0 to 9. When the-most significant digit of an element is say 5, it is inserted in the 6th linked list and when its value is 9 it is inserted in to the last linked list irrespective of the other digits.

When there is a **collision** (numbers having the same most significant digit) the number is inserted in the same list, but in an ordered fashion. After all the elements are placed in their respective lists, the ten lists are merged that would give the ordered list (see Figure 8.5).

We shall assume,

(1) The numbers are two digits only.

(2) To implement the linked list, use the implicit array representation (though you may use the linked representation also).

(3) `h[0:9]` is an array that stores the addresses of the 10 linked lists.

Assume that we wish to store 72 in the list, first extract its most significant digit by using,

```
msd = x / 10 = 72 / 10 = 7.
```

While inserting 72, ensure that it is put in the ascending order. For instance, when element 71 (4th element) is considered for insertion, its `msd` = 71 / 10 = 7 is again falls in `h[7]` linked list where 72 is already present. While inserting 71, ensure that an ordered insertion is done.

a[0:15] = [72, 11, 50, 71, 10, 86, 68, 59, 81, 45, 82, 27, 15, 22, 77, 12]

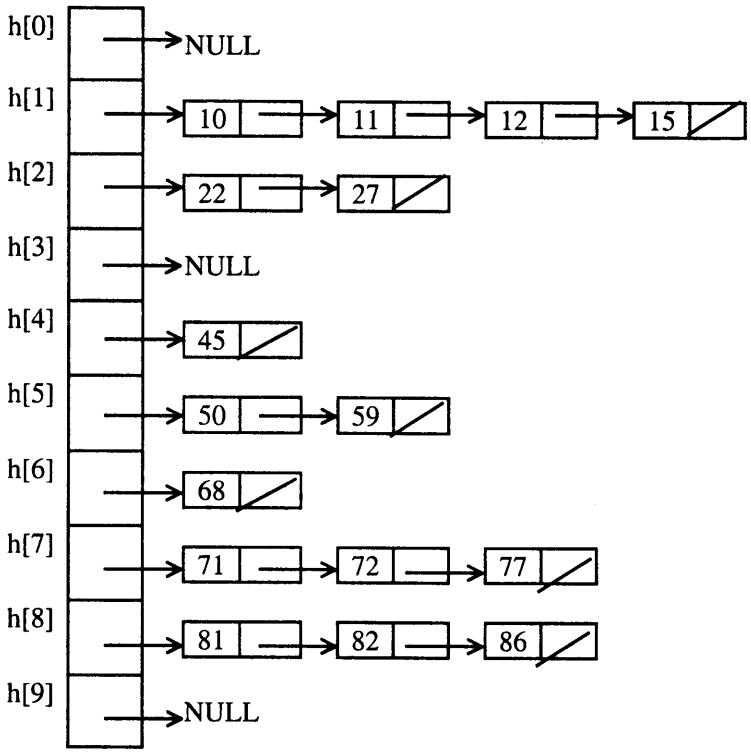


Fig. 8.5 Linked list arrangement of address calculation sort

After inserting all the elements in the lists, the ten linked lists are merged together to form the final sorted array. Now, the steps for the address calculation sort can be summarized as,

Step 1: Initialize the table `h[0:9]` with -1(or NULL)
Initialize each linked list with –1 (or NULL)

Step 2: Repeat thru **Step 3** for `i` = 0 to `n`

Step 3: `x = a[i];`
`msd = x / 10;`
insert `x` is `h[msd]` in an ordered fashion.

Step 4: Repeat thru **Step 5** for `j` = 0 to 9

Step 5: `p = t[j];` // get the address of the `j`th linked list.
while `p` <> - 1 do
put all the elements from the list into the array;

Step 6: Return the sorted array `a`.

In this pseudo code, we have assumed that `t` is a linked list represented as an implicit array for address calculation sort.

The C source code for address calculation sort based upon the pseudo code just explained appears as Program 8.8.

Program 8.8
Address Calculation Sort

```
#define MAX 80
struct node
{
     int info;
     int link;
};

/* linked lists availability list */
struct node list[MAX];
int avail = 0;  /* header for the availability list */
int sf[10];     /* for address of each linked list  */

void AddrCalSort(int a[], int n)
{
     int i,j,p,x;
     int digit;
     avail = 0;
     /* create the availability list as a linked list */
     /* like a heap in dynamic allocation             */
     for (i = 0; i < n-1; i++)
          list[i].link = i + 1;
     list[n-1].link = -1;

     /*  Initialize the sub files with NULL i.e. -1 */
     for (i = 0; i < 10; i++)
          sf[i] = -1;

     /* insert the elements in the appropriate list */
     for (i = 0; i < n; i++)
     {
          x = a[i];
          digit = x / 10;   /* get the first digit */
     /* search and place the element in the linked list */
          Insert(digit, x);
     }

     /* reconstruct the array */
     i = 0;
     for (j = 0; j < 10; j++)
     {
          p = sf[j];
```

```
            while (p != -1)
            {
                  a[i++] = list[p].info;
                  p = list[p].link;
            }
      }
}

void Insert( int digit, int x)
{
      int q;
      int pred; int temp;
      q = getnode(); /* allocate node */
      list[q].info = x;
      /* first node insertion for a particular subfile */
      if (sf[digit] == -1)
      {
            sf[digit] = q;
            list[q].link = -1;
            return;
      }
      pred = -1;
      /* find the address for insertion */
      for (temp = sf[digit]; temp != -1 &&
                              x > list[temp].info;
                              temp = list[temp].link)
            pred = temp;
      if (pred == -1)  /* insert as the first element */
      {
            list[q].link = temp;
            sf[digit] = q;
      }
      else    /* insert after */
      {
            list[q].link = list[pred].link;
            list[pred].link = q;
      }
}

/* function to return one location */
int getnode ()
{
      int temp;
      if (avail == -1)
      {
            printf("Overflow\n");
```

```
            exit(1);
        }
        temp = avail;
        avail = list[avail].link;
        return temp;
}
```

8.12 RADIX SORT

Suppose that a list maintains the student data with the attributes [*regno, name, marks*]. Assume that the marks are in the range 0 to 100 - which is the case in most of the Universities. We are to sort the student data based on their marks scored. An ordinary method might take $O(n^2)$, but we will use **radix sort** to increase the efficiency to $\Theta(n)$

In radix sort, we do not need to take care about the range as we can sort n integers in the range 0 to $n^c - 1$ where c is a constant. Each number in the input is decomposed using some radix r.

For example, using $r = 10$ (decimal) the number 478 can be decomposed to

$$478 = 4 * 10^2 + 7 * 10^1 + 8 * 10^0$$

For sorting in this method, we first decompose the number into its digits and sort based on left significant to most significant digits. This is shown in Figure 8.6.

$$a[0 : 9] = [216, 521, 425, 116, 91, 515, 124, 34, 96, 24]$$

Step 1: Put the input numbers into one of 0 to 9 bin or bucket based on the least significant digit.

Bin[0]	-
Bin[1]	521 91
Bin[2]	-
Bin[3]	-
Bin[4]	124 34 24
Bin[5]	425 515
Bin[6]	216 116 96
Bin[7]	-
Bin[8]	-
Bin[9]	-

(a) Sort on least significant digit

Step 2: Collect all these bins to form a single list.

= [521, 91, 124, 34, 24, 425, 515, 216, 116, 96]

Step 3: Now sort on the 2nd least significant digit.

Bin[0]	-
Bin[1]	515 216 116
Bin[2]	521 124 24 425
Bin[3]	34
Bin[4]	-
Bin[5]	-
Bin[6]	-
Bin[7]	-
Bin[8]	-
Bin[9]	91 96

Sort on 2nd least significant digit

Step 4: Merge all the bins again

= [515, 216, 116, 521, 124, 24, 425, 34, 91, 96]

Step 5: Put the numbers into the bins based on most significant digit.

Bin[0]	24 34 91 96
Bin[1]	116 124
Bin[2]	216
Bin[3]	-
Bin[4]	425
Bin[5]	515 521
Bin[6]	-
Bin[7]	-
Bin[8]	-
Bin[9]	-

(c) Sort on most significant digit

Step 6: Merge all bins.

= [24, 34, 91, 96, 116, 124, 216, 425, 515, 521]

This is the final sorted sequence. Now, the question is how to decompose the digits of a number? We use a simple expression to obtain the least significant digit to the most significant digit as follows,

x %10; (x % 100) / 10; (x % 1000) / 100; . . .

For example, let x = 425, then

The least significant digit is = 425 % 10 = 5

The second least significant digit is = (425 % 100) / 10 = 25 / 10 = 2

The most significant digit is = (425 % 100) / 100 = 425 / 100 = 4

- **Method-1 [Array implementation]**

The first method is based on the sequential storage in which each bin is represented as a queue. Each queue has a *front* and *rear* pointer. When the elements are put in the bins, the corresponding queue pointers are updated. This is exactly similar to the insertion of an element in an ordinary queue.

The next step is to collect all these queues so that sorting can be done on the second least significant digit. This collection process involves physical movement of data. To avoid such movements, we can adjust the links as if that they are stored in an implicit array based linked list. The resulting code is shown in Program 8.9.

Program 8.9
Radix Sort – Array Implementation

```
void RadixSort (int a[], int n)
{
     int f[10], r[10];
     struct
     {
          int info;
          int link;
     } node[MAX];
     int ex, first, i, j, k, p, q, y;
     for (i = 0; i < n-1; i++)
     {
          node[i].info = a[i];
          node[i].link = i + 1;
     }
     node[n-1].info = a[n-1];
     node[n-1].link = -1;
     first = 0;
     for (k = 1; i < 5; i++)
     {
          for (i = 0; i < 10; i++)
          {
               r[i] = -1; f[i] = -1;
          }
          for (; first != -1; first = node[first].link)
          {
               p = first;
               y = node[p].info;
```

```
                ex = pow(10, k-1);
                j = (y/ex) % 10;
                q = r[j];
                if (q == -1) f[j] = p;
                else node[q].link = p;
                r[j] = p;
            }
            /* combine the queues */
            for (j = 0; j < 10 && f[j] == -1; j++);
            first = f[j];
            while (j < 9)
            {
                for (i = j+1; i && f[i] == -1; i++);
                if (i <= 9)
                {
                    p = i;
                    node[r[j]].link = f[i];
                }
                j = i;
            }
            node[r[p]].link = -1;
        }
        /* copy the elements from list to a */
        for (i = 0; i < n; i++)
        {
            a[i] = node[first].info;
            first = node[first].link;
        }
}
```

- **Method-2 [Linked Implementation]**

The elements are first assumed to be stored in a singly linked list. Next, each bin has two pointers `bottom` and `top`. Since, there are ten bins; instead of having scalar pointers we must have vector pointers. This means that {`bottom[0]`, `top[0]`}, {`bottom[1]`, `top[1]`}, . . ., {`bottom[9]`, `top[9]`} are the pointer combinations for each bin starting from 0 to 9. The pointer `bottom[b]` points to the last element and `top[b]` points to the first element of bin `b`. For example, in Figure 8.6(a) bin 4 consists of three elements 124, 34, and 24. Now, `bottom[4]` will points to 24 and `top[4]` will point to 124.

The implementation involves distributing the elements to various bins and updating the bottom and top pointers for each digit position. At the end of each significant digit, collection phase links the `top` and `bottom` pointers such that it becomes a linked list. Once the `k`th digit (where `k` is the maximum number of digits allowed in a number) is

processed, the linked list would contain the sorted list. Program 8.10 shows the source code for `RadixSort()` function with linked lists.

Program 8.10
Radix Sort – Linked list implementation

```
void RadixSort (int a[], int n)

{
     NODE t = NULL;
     int i;
     /* create the linked list */
     for (i = 0; i < n; i++)
          t = Create(t, a[i]); /* creates a linked list */
                               /* out of the array      */
     t = RSort(t, 10); /* radix is 10 */

     /* copy sorted list to a */
     for (i = 0; i < n; i++)
     {
          a[i] = t->info;
          t = t->link;
     }
}

NODE RSort (NODE first, int range)
{
     int b, k, d, div, x;
     NODE bottom[MAX], top[MAX];
     NODE y;
     int digits = 4; /* max. no. of digits allowed */
     d = 10;
     div = 1;
     for (k = 1; k < digits; k++) {
          y = NULL;

     for (b = 0; b <= range; b++)
           { bottom[b] = NULL; top[b] = NULL; }

     for (; first; first = first->link)
     {
          x = first->info;
          b = (x % d) / div; /* decompose into digits */
          /* put x into appropriate bin */
          if (bottom[b]) /* bin has elements */
```

```
            {
                  top[b]->link = first;
                  top[b] = first;
            }
            else  /* add to empty bin */
                  bottom[b] = top[b] = first; }
      /* merge all the bins */
      for (b = 0; b <= range; b++)
            if (bottom[b])
            {
                  if (y)
                        y->link = bottom[b];
                  else
                        first = bottom[b];
                  y = top[b];
            }
      if (y) y->link = NULL;
      d = d * 10; /* for next digit position */
      div = div * 10; }
      return first;
}
```

This function `RadixSort()`, first creates the linked list from the given array `a` and sends the list to `RSort()`.

In the function `RSort()` the outer most `for` loop is used to sorting on each significant digit (*k*th digit). Also we assume a maximum of 3 digits are allowed in the input sequence. The next `for` loop picks each number and decomposes into individual digits (first iteration, the least significant digit, and so on), Then the numbers are distributed to the appropriate bins. Once the distribution is over, collection phase starts and is done with another `for` loop ranging from bin 0 to 10.

The pointer `first` points to the first element of first bin (`if-else` clause) and the subsequent bins are linked, provided there are elements in the bin. Finally, the element are copied back to the array `a` in the function `RadixSort()`.

8.13 SUMMARY

- The sorting problem has an input sequence of n numbers typically in an array $a[1{:}n]$ and the output is a sorted sequence either in ascending order or descending order. For instance,

  input: $[a_1, a_2, a_3, \ldots, a_n]$
  output: $a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_n$

- Time and space complexity are two important concepts that are required in algorithmic design.
- In we do not worry about the **time** taken by a program, certain programs may take even years of computer time (examples: traveling sales person problem, colouring a map, etc.) we cannot wait for years to get the results.
- The **space complexity** of a program is the amount of memory needed to run a program. The time complexity is the amount of time a program takes for the execution.
- The word **asymptotic** means that it is the study of functions of a parameter n, as n becomes larger and larger without bound. In other words, we are concerned with how the running time of an algorithm increases with the size of the input.
- Three mathematical notations were introduced: O(Big Oh), Ω(Omega), and Θ(Theta).
- The big Oh notation is defined as, $f(n) = O(g(n))$ (read as "f of n is equal to the big Oh of g of n" such that there exists two positive constants c and n_0 with the constraint that $|f(n) \leq c\,|g(n)|$ for all $n \geq n_0$.
- The big Omega notation is defined as, $f(n) = \Omega(g(n))$ (read as "f of n is equal to omega of g of n") iff there exists positive constants c and n_0 such that for all $n > n_o$, $|f(n)| \geq c\,|g(n)|$
- The big Theta notation is defined as, $f(n) = \Theta(g(n))$ iff there exist positive constants c_1 and c_2 such that for all $n > n_0$, $c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|$
- Bubble sort works on the principle of **bubbling** strategy. This means that, At the end of each *pass* the largest element is moved to the right most position of the array, assuming that we wish to sort in an ascending order.
- Merge Sort works on the principle of simple merge. A simple merge is to combine two ordered lists into a single ordered list.
- Heap sort works on the basis of Max Heap. The root node of the heap (the max element) will occupy the last position and the second largest will occupy the last but one, and so on.
- Several other sorting algorithms were discussed: shell sort, address calculation sort, selection, insertion, and radix sorts.

8.14 EXERCISES

8.1 Define the following:
(1) Big Oh notation
(2) Omega Notation
(3) Theta Notation

8.2 What is the significance of finding the time complexity of a program? Discuss.

8.3 Considering the following data set, trace the Bubble sort program.

[45, 23, 1, 78, 9, 3, 47, 22, 40, 99]

Write an efficient bubble sort program. Derive an expression for the number of passes required for n numbers.

8.4 Write the Quick Sort algorithm and trace the same with the following data set:

[3, 6, 56, 8, 23, 44, 20, 5, 10, 78]

8.5 Explain the simple merge process and write a C function that sorts an array of n numbers using Merge Sort.

8.6 Using the data set of Problem 8.5 write the recursive tree of Merge sort.

8.7 Show the complete sequence of steps in sorting a list using Heap Sort. You may consider the same data set of Problem 8.4.

8.8 Write algorithms for the selection sort and insertion sort. What are the differences between these two sorting algorithms?

8.9 In Shell sort of Section 8.10, we started with d as 5. How to select the initial value for d? Take a large array and sort it using the shell sort. Vary the initial value of d and write report about the behavior of the sorting process.

8.10 Compare the addressing calculation sort and radix sort. Give C programs for both.

Chapte 9

Searching

9.1 INTRODUCTION

The **searching** problem can be formally stated as:

Given a set of n numbers a[1 : n] = [a_1, a_2, a_3, ..., a_n] and a key k, the task of a searching algorithm is to return the index i if k is in the list or return a special value NULL or –1 if k does not appear in a.

Searching for a *key* as stated above is required almost in every application. Consider for example a student information system that contain the student details like *register number*, *name*, *marks*, *attendance*, *address*, etc, in the form of a table. We may wish to search for a particular student record by specifying his/her register number as this would search and return the record or its index uniquely. The same is true in an employee information system or in web based applications. We are all familiar with the famous web site http://www.google.com that helps the users to search for a particular item given in terms of a phrase.

The algorithms that we develop in this chapter assume that the list is a static array *a* and its size n. The return value will be an integer value of the index of the key or –1 depending upon whether it exists in the list or not. Following are the searching algorithms that we study in this chapter:

- Linear Search
- Binary Search
- Interpolation Search
- Hashing

Each method will be explained with the basic concept behind the design, the C code and implementation issues.

9.2 LINEAR SEARCH

The most straight forward method to search for the key is to begin at one end (1st location) of the list and scan down it until the key is found or the other end (*n*th location) is reached. This method is called as the **sequential search** or **linear search**.

To implement the above strategy, all that we require is a simple looping structure to scan the list from 0 to *n*-1 in the array *a*. In every iteration, we check the array element *a*[*i*] with the key and if it is equal we have found the key and its index *i* can be returned. If the loop continues until the end of the list, then last statement will be returning –1 indicating that the key is not found in the array. Program 9.1 uses this logic.

Program 9.1
Linear Search

```
int LinearSearch (int a[], int n, int key)
{
  /* return */
     int i;
     for (i = 0; i < n; i++)
          if (a[i] == key )
               return i;
     return -1;
 }
```

When the key is found in the list, the *loop* gets terminated in the middle because of the `return` statement. If the control reaches `return -1;` this means that the key does not match with any of the array element.

In the case of the linear search, if the key is found in the 1st position itself then the `for` loop need not be executed for *n* times as explained already. Then the number of comparisons is just 1. However, if it's situated at the last location, then *n* comparisons would be made.

This amounts to the probability of the key being in the 1st, or second or in the last position. If the key is found in the first position itself, then it is called as the **best case**, where as if it is found at the last, then it is called as the **worst case**. We are also interested in the **average case.**

Time Complexity

For the sake of curiosity, we can show the average number of comparisons of linear search.

Average comparisons = number of comparisons upto 0th position
+ number of comparisons upto 1st position
+ .....
+ number of comparisons upto ith position
+ number of comparisons upto nth position

$$= \frac{1+2+---+n}{n}$$

$$= \frac{n(n+1)}{n.2}$$

$$= \frac{1}{2}(n+1)$$

The best case time complexity for the linear search is $O(1)$ and the worst case is $O(n)$.

9.2.1 Recursive Linear Search

The Program 9.1 is an iterative version for searching an element in an array. The same task can be accomplished using the recursive technique. We need to design the recursive version by comparing the key with the array elements starting from (n-1)th position down to 0th position. If there is no match, then we recursively call the search function by reducing n by 1, i.e. n-2, n-3, ..., 0. When n goes down to a negative value, it is obvious that the key is not found in the array.

The Program 9.2 shows the `RecLinearSearch()` function employing recursive technique.

Program 9.2
Recursive Linear Search

```
int RecLinearSearch (int a[], int n, int key)
{
     if (n < 0) return -1;
     if (a[n - 1] == key)
          return n;
     else
          return RecLinearSearch(a, n-1, key);
}
```

```
            mid = (low + high) / 2; /* divide the list */
            if (key == a[mid])
                  return mid + 1; /* found */

            if (key < a[mid])
                  high = mid-1; /* update low or high */
            else low = mid+1;
      }
      return -1; /* not found */
}
```

The binary search algorithm works based on the concept of binary search tree where searching is made fast by traversing the binary tree towards left subtree or towards the right subtree. The time complexity of binary search is,

	Best case	Worst case	Average case
1. Successful Search (key found):	$O(1)$	$O(\log n)$	$O(\log n)$
2. Unsuccessful case (key not found):	$O(\log n)$	$O(\log n)$	$O(\log n)$

9.3.1 Recursive Binary Search

The binary search algorithm can also be designed to work with recursive technique. The terminating condition is almost same as the iterative version, i.e. we invoke `RecBinarySearch()` function as long as `low < high`. During each invocation, we update the `low` or `high` pointer in the calling sequence. The C code is shown in Program 9.4.

Program 9.4
Recursive Binary Search

```
int RecBinarySearch(int x[], int n, int key,
                            int low, int high)
{
     int mid;
     if (low > high) return -1;
     mid = (low + high) / 2; /* divide the list */
     if (key == x[mid])
          return mid;  /* key found */

     if ( key < x[mid] )  /* key not found */
          return RecBinarySearch(x, n, key, low, mid-1);
     else
```

```
        return RecBinarySearch(x, n, key, mid+1, high);
}
```

Note that there are two extra parameters in the function `low` and `high`. Therefore, the function should be called with the following syntax initially sending `low` as 0 and high as `n-1`,

```
RecBinarySearch(a, n, key, 0, n-1);
```

The subsequent calling of this function modifies the value of `low` or `high` as you see in the program. The function returns -1 for *not found* case and for *found case* the key's index.

9.4 INTERPOLATION SEARCH

Another variant of binary search is not to use the middle element of a subset to compare to the key item, but to guess more precisely where the key being sought falls within the current interval of interest (think of searching in a telephone directory again). This improved version is called **interpolation search**. The new element is calculated as follows:

```
mid = round(low + (key - a[low]) /
                    (a[high] - a[low]) * (high - low));
```

Using this new technique, we shall modify the binary search program of 9.3 as Program 9.5.

Program 9.5
Interpolation Search

```
int InterpolationSearch (int a[], int n, int key)
{
     int low, high, mid;
     low = 0; high = n-1;
     while (low <= high)
     {
          mid = low + (key-a[low])/(a[high]-a[low])
                                   *(high-low);
          if (key < a[mid])
               high = ((low + mid - 1)/2)-1;
          else if (key > a[mid])
                    low = ((mid+1 + high)/2)+1;
               else
                    return mid;
```

```
    }
      return -1; /* not found */
}
```

In the interpolation search also we initialize `low` to 0 and `high` to `n-1`. However, the `low` and `high` are updated in a different way. The rest of the statements are similar to Program 9.3.

For all the searching functions Linear, Binary and Interpolation, Program 9.6 can be used as a `main()` or driver program.

Program 9.6
Driver program

```
#include <stdio.h>
#define MAX 100
int BinarySearch (int [], int, int);

void main()
{
     int a[MAX];
     int n, i, key, y;
     printf("Enter the Max. no. of Elements: ");
     scanf("%d", &n);
     printf("Enter Elements in ascending order: ");
     for (i = 0; i < n; i++)
          scanf("%d", &a[i]);
     printf("Enter the Key to be searched: ");
          scanf("%d", &key);

     /* call appropriate search function */
     y = BinarySearch(a, n, key);

     if (y == -1)
          printf("Element not found\n");
     else  printf("Element found at = %d\n", y);
}
```

Most of the search functions accept the array, size of the array and the key to be searched. In the Program 9.6, the statement `y = BinarySearch(a, n, key);` should be replaced with the name of the search function.

9.5 HASHING

Many applications require a dynamic set that supports only the dictionary operations Insert, Search, and Delete. For example, a compiler maintains a symbol table, in which the keys are arbitrary character strings that correspond to identifiers in the language.

A **hash table** is an effective data structure for implementing **dictionaries**. In the worst case, hashing may take $\Theta(n)$, but with suitable assumptions we can achieve an expected time of $O(1)$ to search for an element in a hash table. A hash table is stores the keys that are mapped using a **hash function.**

Assuming, k is the key to be stored in the hash table, f is the hash function and M is the hash table size, then k is stored in position $f(k)$. To search for the key k, first it is mapped using the hash function. Then we see if there is an element at $f(k)$. If so, we have found the element, otherwise the element is not found. This is of course an ideal case that we have presented. We shall see soon the anomalies in this method and how to correct it.

Example

For this example we shall consider the hash function as,

$$f(k) = k \% \text{ M} \qquad \ldots.(9.1)$$

The hash table is a simple array, h[0 : M] in which the hashed key k will be stored. Figure 9.2 shows a hash table h[0:9] of size 10 and the elements to be stored are :

[23, 56, 111, 9, 200]

The hash values are calculated using Equation 9.1 for all the elements and are shown below:

$f(23) = 23 \% 10 = 3$
$f(56) = 56 \% 10 = 6$
$f(111) = 111 \% 10 = 1$
$f(9) = 9 \% 10 = 9$
$f(200) = 200 \% 10 = 0$

We wish to insert element 23, for which we calculate the hash value as 3 and this element will be stored in h[3]. Similarly all the elements are stored in the hash table h as shown in Figure 9.2.

h	0	1	2	3	4	5	6	7	8	9
	200	111		23			56			9

Fig. 9.2 Hash table

Now what ever be the key element to be stored in the hash table, it would fall in the address range 0 to 9, because of the *mod* (%) operator and the table size M. When we want to search for a key whose value is 56, it is first hashed using Equation 9.1 and the

element at $h[k]$ will be checked whether it has an element or not (*empty*). In this example, since the location 6 is not *empty*, we get a successful search. On the other hand if we search for 45, the hashed value is 45 % 10 = 5, and $h[5]$ is *empty*, Hence, it is an unsuccessful search. On an average the time complexity of this ideal hash method is $O(1)$.

However, this method has drawbacks. Suppose we wish to insert the element 83. As explained already, it is hashed and the address we get is 3. But $h[3]$ is already occupied with element 23. Note that we can insert an element only when the hash table location is **empty**. This situation is called as **collision**. This type of situation may occur for many elements that produce the same hash address – 13, 3, 203, 63, 73, 113, etc., all these give hash address as 3 producing collisions.

We shall study several techniques to resolve collisions which will be our major topic in this section. One possible solution may to use a good hash function. The following section discusses that.

9.5.1 Hash functions

In this section, we discuss some issues regarding the design of good hash functions namely: **truncation** and **folding**. The method that we have already used in Equation 9.1 is called as **hashing by division**. There are two important considerations that should be considered in the design of good hash function *(1) the hash function should not consume more time in calculating the hash address. (2) the function should not yield more collisions.*

Truncation

This method picks only few digits of the number and appends these digits to form the hash address. For example, we may wish to store the telephone numbers in the hash table. Each telephone number is a 7 digit number. We can extract the first, third and fifth digit from the beginning of the number and concatenate these digits to form the address.
For instance,

$$h[6605789] = 607,\ h[6661890] = 668 \text{ and } h[3351834] = 358$$

We need the table size M as 1000 to store the elements and you can notice that this method produces collision rarely. The only requirement is that all the elements should be of fixed size. All the telephone numbers in Bangalore City are of 7 digits. Selection of M also plays an important role in reducing collisions. If you select M = 100, then you must select only two digits, say 2 and fourth digits. Then there will be more collisions that you can expect.

Folding

Another method is to pick group of digits, first two digits, second two digits and last three digits. Then, add these groups to get the final address. If the digits are numbered $d_1, d_2,\ldots,d_7$, then the hash address is

$$= d_1\, d_2 + d_3\, d_4 + d_5\, d_6\, d_7$$

Considering again the same example of Bangalore telephone numbers,

$h[6605789] = 66 + 05 + 789 = 860$
$h[6661890] = 66 + 61 + 890 = 017$
$h[3351834] = 33 + 51 + 834 = 918$

In the second case, we get a four digit number after adding 66 + 61 + 890 = 1017 in which we discard the most significant digit.

9.5.2 Collision resolution techniques

One of the primary requirements in hashing is to have a smaller hash table size for a larger range of the key. This section concentrates two methods when collision occurs namely

- Hashing with Linear Open Addressing
- Hashing with Chains

Though an introduction has already been given in Figure 9.2, we shall discuss the implementation issues in length.

Hashing with Linear Open Addressing

Referring to Figure 9.2, the collision occurred when we try to insert element 83, but where to insert this element? The easiest thing to do is search the table for the next available location and place 83. This method is called as **linear open addressing**. In the present example, the next *empty* location is $h[4]$ and hence it is inserted in that place. (see Figure 9.3).

h	0	1	2	3	4	5	6	7	8	9
	200	111		23	83		56			9

Fig. 9.3 Hash table with 83

h	0	1	2	3	4	5	6	7	8	9
	200	111		23	83	104	56			9

Fig. 9.3a Hash table with 104

Next, if we get an element 104 for insertion its hash position is already occupied by 84 and so it is placed in $h[5]$ (see Figure 9.3a).

How to get the next *empty* hash table location when a collision occurs? We can use a technique called **rehashing**. If the hash address is *empty* we can insert the key, else rehash the key using the following formula,

$f(k) = k \% M$
$f'(k) = (f(k) + 1) \% M$

$f(k)$ is the hash function that we have already seen and $f'(k)$ is the rehash function. For example, let us assume $k = 83$.

$$f(83) = 83 \% 10 = 3$$
$$f'(k) = (3 + 1) \% 10 = 4$$

Therefore, we can insert element 83 in the location $h[4]$.

Implementation

We shall develop two functions `Insert()` and `Search()` to insert a key in the hash table and search the hash table for a given key respectively. The rehash method will be used for getting an `empty` location. Additional functions `Initialize()` and `Hsearch()` will be used for insertion and search operations.

Initialize()

The hash table is a simple C array `h[0 : Size]`, where `Size` is the table size. We also assume that the table will be initialized with –1 (`empty`) before we attempt any of the hash table operations. Following piece of code (Program 9.7) does the required initialization.

```
#define Size   10
#define empty  -1
#define true   1
#define false  0
```

Program 9.7
Initialization of hash table

```
void Initialize (int h [])
{
     int i;
     for (i = 0; i < Size; i++)
          h[i] = empty; /* empty */
}
```

Insert()

This function is designed to accept the key to be inserted and obtains the hash address using a hash function. If the home address is already occupied by an element, then rehashing is tried in a wrap around fashion, until wether an `empty` position is reached or table is full. In case of table full, it returns false, otherwise the key is inserted. To obtain the hash address, we invoke a function `HSearch()`. The main functionality of this function is to return the hash address, if any. See Program 9.8 for the C code.

The `HSearch()` function tries to see that the hashed value is `empty`. If so, its index, `j` is returned so that insertion can be done. Also this function tries for rehashing

to locate an empty position in a wrap around fashion. Suppose if there is no vacancy it returns false and no insertion can be done.

Program 9.8
Insert a key in the hash table

```
int Insert (int h [], int k)
{
     int j = HSearch(h, k);
     if (h[j] == empty)
     {
          h[j] = k;     /* insert */
          return true;
     }
     if (h[j] == k || j == empty)
          return false; /* duplicate or table full */
}

int HSearch (int h [], int k)
{
     int i = k % Size; /* hash function */
     int j = i;

     do
     {
          if (h[j] == empty || h[j] == k) return j;
          j = (j + 1) % Size; /* rehash function */
     } while (j != i); /* search wrap around */

     return false; /* table full */
}
```

Search()

We can use the function `HSearch()` to search for a given key in the hash table. We shall look for the key in the table and if it is found it is returned via a reference parameter, `e`. Firstly, the key is hashed and rehashed - may be because of a collision during insertion – to obtain the address, `i`.

This is done using `HSearch()`. If the hashed value contains an `empty` then the key is not found in the table or the location may contain some other element, because of the collision. In either case we can return false. Otherwise, simply extract the element to the parameter `e` and return *true*. The function `Search()` appears in Program 9.9.

Program 9.9
Search for a key

```
int Search (int h [], int k, int *e)
{
   /* search k in hash table */
   /* return true if k is found and e holds the element */
   /* otherwise return false */
     int j = HSearch(h, k);
     if (j == empty || h[j] != k)
          return false; /* no match */
     *e = h[j];
     return true;
}
```

Deletion from the hash table

This section discusses the issues with regard to the deletion of an element in the hash table. This becomes important as it not just setting the hash address of the key to be deleted to `empty`! The problem occurs when a collision occurred during an insertion.

h	0	1	2	3	4	5	6	7	8	9
	200	111		**23**	**83**	85	56	**33**	**13**	9

Fig. 9.4 Hash table deletion

h	0	1	2	3	4	5	6	7	8	9
	200	111		**23**	-1	85	56	**33**	**13**	9

Fig. 9.4a Element 83 is deleted

Let us show the deletion problem by considering the hash table as it appears in Figure 9.4. Assume that the keys are inserted in the order 23, 83, 33 and 13 (shown in bold letters). The element 23 goes to `h[3]` directly, next element 83 goes to `h[4]` after rehashing, then element 33 goes to `h[7]`, since `h[5]` and `h[6]` are already occupied and finally 13 goes to `h[8]` because of collision at several places.

Now, we wish to delete key 83. The hash address we get is 3 and `h[3]` is not the key and so rehashing is done which takes you to `h[4]`. This equals the key and hence `h[4] = empty` (shown in Figure 9.4a).

Next we would like search for the key 33 (or 13). We invoke `Search()` function, in which it returns false. Because, the hash address for 33 is 3 and it is already occupied with 23 and next rehashing is done which yields 4 and since `h[4] = -1`, *false* is returned