```
#include <stdio.h>
#define MAX 100
void heap_sort(int arr[],int size);
void buildHeap(int arr[],int size);
int del_root(int arr[],int *size);
void restoreDown(int arr[],int i,int size);
void display(int arr[],int n);
main()
{
      int i,n arr[MAX];
      printf("Enter number of elements : ");
      scanf("%d",&n);
      for(i=1; i<=n; i++)
      {
            printf("Enter element %d : ",i);
            scanf("%d",&arr[i]);

      }
      printf("Entered list is :\n");
      display(arr,n);
      heap_sort(arr,n);
      printf("Sorted list is :\n");
      display(arr,n);
}/*End of main()*/
void heap_sort(int arr[],int size)
{
      int max;
      buildHeap(arr,size);
      printf("Heap is : ");
      display(arr,size);
      while(size>1)
      {
            max = del_root(arr,&size);
            arr[size+1]=max;
      }
}/*End of heap_sort*/
void buildHeap(int arr[],int size)
{
      int i;
      for(i=size/2; i>=1; i--)
            restoreDown(arr,i,size);
}/*End of buildHeap()*/
int del_root(int arr[],int *size)
{
      int max = arr[1];
      arr[1] = arr[*size];
      (*size)--;
      restoreDown(arr,1,*size);
      return max;
}/*End of del_root()*/
void restoreDown(int arr[],int i,int size )
{
      int left=2*i, right=left+1;
      int num = arr[i];
      while(right<=size)
      {
            if(num>=arr[left] && num>=arr[right])
            {
                  arr[i] = num;
                  return;
            }
            else if(arr[left] > arr[right])
```

```
                {
                        arr[i] = arr[left];
                        i = left;
                }
                else
                {
                        arr[i] = arr[right];
                        i = right;
                }
                left = 2 * i;
                right = left + 1;
        }
        if(left == size && num < arr[left])  /*when right == size+1*/
        {
                arr[i] = arr[left];
                i = left;
        }
        arr[i] = num;
}/*End of restoreDown()*/
void display(int arr[],int n)
{
        int i;
        for(i=1;i<=n;i++)
                printf("%d  ",arr[i]);
        printf("\n");
}/*End of display()*/
```

8.15.1 Analysis of Heap Sort

The algorithm of heap sort proceeds in two phases. In the first phase we build a heap and its running time is O(n) if we use the bottom up approach. The delete operation in a heap takes O(log n) time, and it is called n-1 times, hence the complexity of second phase is O(nlog n). So the worst case complexity of heap sort is O(nlog n). The average case and best case complexity is also O(nlog n).

Heap sort is good for larger lists but it is not preferable for small list of data. It is not a stable sort. It has no need of extra space other than one temporary variable so it is an in place sort and space complexity is O(1).

8.16 Radix Sort

Radix sort is a very old sorting technique and was used to sort punch cards before the invention of computers. If we are given a list of some names and asked to sort them alphabetically, then we would intuitively proceed by first dividing the names into 26 piles with each pile containing names that start with the same alphabet. In the first pile we will put all the names which start with alphabet 'A', in the second pile we will put all the names which start with alphabet 'B' and so on. After this each pile is further divided into subpiles according to the second alphabet of the names. This process of creating subpiles will continue and the list will become totally sorted when the number of times subpiles are created is equal to the number of alphabets in the largest name.

In the above case, where we had to sort names the radix was 26(all alphabets). If we have a list of decimal numbers then the radix will be 10 (digits 0 to 9). We have sorted the names starting from the most significant position (from left to right). While sorting numbers also we can do the same thing. If all the numbers don't have same number of digits then we can add leading zeros. First all numbers are divided into 10 groups based on most significant digit, and then these groups are again divided into subgroups based on the next significant digit. The problem with the implementation of this method is that we have to keep track of lots of groups and subgroups. The implementation would be much simpler if the sorting starts from the least significant digit(from right to left). These two methods are called most significant digit (MSD) radix sort and least significant digit (LSD) radix sort.

In LSD radix sort, sorting is performed digit by digit starting from the least significant digit and ending at the most significant digit. In first pass, elements are sorted according to their units (least significant) digits, in second pass elements are sorted according to their tens digit, in third pass elements are sorted according to hundreds digit and so on till the last pass where elements are sorted according to their most significant digits.

For implementing this method we will take ten separate queues for each digit from 0 to 9. In the first pass, numbers are inserted into appropriate queues depending on their least significant digits(units digit), for example 283 will be inserted in queue 3 and 540 will be inserted in queue 0. After this all the queues are combined starting from the digit 0 queue to digit 9 queue and a single list is formed. Now in the second pass the numbers from this new list are inserted in queues based on tens digit, for example 283 will be inserted in queue 8 and 540 will be inserted in queue 4. These queues are combined to form a single list which is used in third pass. This process continues till numbers are inserted into queues based on the most significant digit. The single list that we will get after combining these queues will be the sorted list. We can see that the total number of passes will be equal to the number of digits in the largest number. Let us take some numbers in unsorted order and sort them by applying radix sort.

Original List : 62, 234, 456, 750, 789, 3, 21, 345, 983, 99, 153, 65, 23, 5, 98, 10, 6, 372

Pass 1 : Numbers classified according to units digit(least significant digit)

Queue for digit 0	750, 10
Queue for digit 1	21
Queue for digit 2	62, 372
Queue for digit 3	3, 983, 153, 23
Queue for digit 4	234
Queue for digit 5	345, 65, 5
Queue for digit 6	456, 6
Queue for digit 7	
Queue for digit 8	98
Queue for digit 9	789, 99

List after first pass : 750, 10, 21, 62, 372, 3, 983, 153, 23, 234, 345, 65, 5, 456, 6, 98, 789, 99

Pass 2 : Numbers classified according to tens digit

Queue for digit 0	3, 5, 6
Queue for digit 1	10
Queue for digit 2	21, 23
Queue for digit 3	234
Queue for digit 4	345
Queue for digit 5	750, 153, 456
Queue for digit 6	62, 65
Queue for digit 7	372
Queue for digit 8	983, 789
Queue for digit 9	98, 99

List after second pass : 3, 5, 6, 10, 21, 23, 234, 345, 750, 153, 456, 62, 65, 372, 983, 789, 98, 99

Pass 3 : Numbers classified according to hundreds digit(most significant digit)

Queue for digit 0	3, 5, 6, 10, 21, 23, 62, 65, 98, 99
Queue for digit 1	153
Queue for digit 2	234
Queue for digit 3	345, 372
Queue for digit 4	456
Queue for digit 5	
Queue for digit 6	
Queue for digit 7	750, 789
Queue for digit 8	
Queue for digit 9	983

List after third pass : 3, 5, 6, 10, 21, 23, 62, 65, 98, 99, 153, 234, 345, 372, 456, 750, 789, 983

Sorted List : 3, 5, 6, 10, 21, 23, 62, 65, 98, 99, 153, 234, 345, 372, 456, 750, 789, 983

Figure 8.20 Radix Sort

It is important that in each pass the sorting on digits should be stable i.e. numbers which have same i^{th} digit(from right) should remain sorted on the $(i-1)^{th}$ digit(from right). After any pass when we get a single list, we should enter the numbers in the queue in the same order as they are in list. This will ensure that the digit sorts are stable.

We take the initial input in linked list to simplify the process. If the input is in array then we can convert it to linked list. In the program we just traverse the linked list and add the number to the appropriate queue. After this we can combine the queues into one single list by joining the end of a queue to start of another queue.

We do not know in advance how many numbers will be inserted in a particular queue in any pass. It may be possible that in a particular pass, the digit is same for all the numbers, and all numbers may have to be inserted in the same queue. If we use arrays for implementing queues then each array should be of size n, and we will need space equal to 10*n. So it is better to take linked allocation of queues instead of sequential allocation.

```
/*P8.12 Sorting using radix sort*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
}*start = NULL;
void radix_sort();
int large_dig();
int digit(int number,int k);
main()
{
        struct node *tmp,*q;
        int i,n,item;
        printf("Enter the number of elements in the list : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&item);
                /*Inserting elements in the linked list*/
                tmp = malloc(sizeof(struct node));
                tmp->info = item;
                tmp->link = NULL;
                if(start==NULL) /*Inserting first element */
                        start = tmp;
                else
                {
                        q = start;
                        while(q->link!=NULL)
                                q = q->link;
                        q->link = tmp;
                }
        }/*End of for*/
        radix_sort();
        printf("Sorted list is :\n");
        q = start;
        while(q!=NULL)
        {
                printf("%d ",q->info);
                q = q->link;
        }
        printf("\n");
}/*End of main()*/
void radix_sort()
{
        int i,k,dig,least_sig,most_sig;
```

```
    struct node *p,*rear[10],*front[10];
    least_sig = 1;
    most_sig = large_dig(start);
    for(k=least_sig; k<=most_sig; k++)
    {
            /*Make all the queues empty at the beginning of each pass*/
            for(i=0; i<=9; i++)
            {
                    rear[i] = NULL;
                    front[i] = NULL ;
            }
            for(p=start; p!=NULL; p=p->link)
            {
                    dig = digit(p->info,k); /*Find kth digit in the number*/
                    /*Add the number to queue of dig*/
                    if(front[dig] == NULL)
                            front[dig] = p ;
                    else
                            rear[dig]->link = p;
                    rear[dig] = p;
            }
            /*Join all the queues to form the new linked list*/
            i = 0;
            while(front[i] == NULL)
                    i++;
            start = front[i];
            while(i<9)
            {
                    if(rear[i+1]!=NULL)
                            rear[i]->link = front[i+1];
                    else
                            rear[i+1] = rear[i];
                    i++;
            }
            rear[9]->link = NULL;
    }
}/*End of radix_sort*/
/*This function finds number of digits in the largest element of the list */
int large_dig()
{
    struct node *p=start;
    int large=0,ndig=0;
    /*Find largest element*/
    while(p!=NULL)
    {
            if(p->info > large)
                    large = p->info;
            p = p->link ;
    }
    /*Find number of digits in largest element*/
    while(large!=0)
    {
            ndig++;
            large = large/10 ;
    }
    return(ndig);
} /*End of large_dig()*/
/*This function returns kth digit of a number*/
int digit(int number,int k)
{
    int digit,i;
    for(i=1; i<=k; i++)
```

```
        {
                digit = number%10 ;
                number = number/10 ;
        }
        return(digit);
}/*End of digit()*/
```

8.16.1 Analysis of Radix Sort

The number of passes p is equal to the number of digits in largest element and in each pass we have n operations where n is the total number of elements. In the program we can see that the outer loop iterates p times and the inner loop iterates n times. So the run time complexity of radix sort is given by O(p*n).

If the number of digits in largest element is equal to n, then the run time becomes $O(n^2)$ and this is the worst performance of radix sort. It performs best if the number of digits in largest element is log n, in this case the run time becomes O(n log n). So the radix sort is most efficient when the number of digits in the elements is small. The disadvantage of radix sort is extra space requirement for maintaining queues and so it is not an in-place sort. It is a stable sort.

8.17 Address Calculation Sort

This technique makes use of hashing function(explained in chapter 9) for sorting the elements. Any hashing function f(x) that can be used for sorting should have this property-

If $x < y$, then $f(x) \leq f(y)$

These types of functions are called non decreasing functions or order preserving hashing functions. This function is applied to each element, and according to the value of the hashing function each element is placed in a particular set. When two elements are to be placed in the same set, then they are placed in sorted order. Now we will take some numbers and sort them using address calculation sort.

194, 289, 566, 432, 654, 98, 232, 415, 345, 276, 532, 254, 165, 965, 476

Let us take a function f(x) whose value is equal to the first digit of x, this will obviously be a non decreasing function because if first digit of any number A is less than the first digit of any number B, then A will definitely be less than B.

x	194	289	566	432	654	098	232	415	345	276	532	254	165	965	476
f(x)	1	2	5	4	6	0	2	4	3	2	5	2	1	9	4

Now all the elements will be placed in different sets according to the corresponding values of f(x). The value of function f(x) can be 0, 1, 2....., 9 so there will be 10 sets into which these elements are inserted.

0	098
1	165, 194
2	232, 254, 276, 289
3	345
4	415, 432, 476
5	532, 566
6	654
7	
8	
9	965

We can see that the value of f(x) for the elements 289, 232, 276, 254 is same, so they are inserted in the same set but in sorted order i.e. 232, 254, 276, 289. Similarly other elements are also inserted in their particular sets in sorted order. All the elements in a set are less than elements of the next set and elements in a set are in sorted order. So if we concatenate all the sets then we will get the sorted list.

98, 165, 194, 232, 254, 276, 289, 345, 415, 432, 476, 532, 566, 654, 965

Let us sort the same list by taking another non decreasing hash function-
h(x) = (x/k) * 5 (where k the largest of all numbers)

x	194	289	586	432	654	98	232	415	345	276	532	254	165	965	476
h(x)	1	1	2	2	3	0	1	2	1	1	2	1	0	5	2

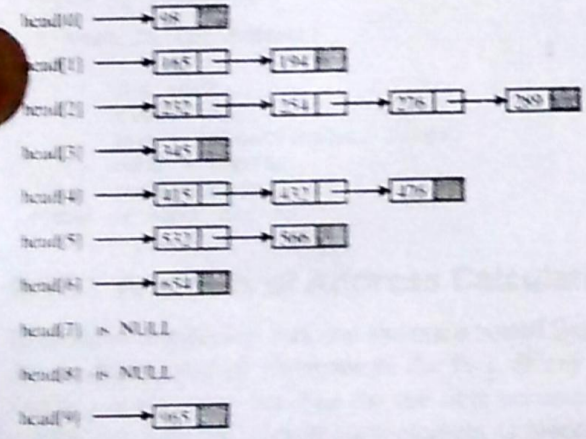
Here the value of function h(x) can be 0, 1, 2, 3, 4, 5 so there will be only 6 sets into which these elements are inserted.

0	98, 165
1	194, 232, 254, 276, 289, 345
2	415, 432, 476, 532, 566
3	654
4	
5	965

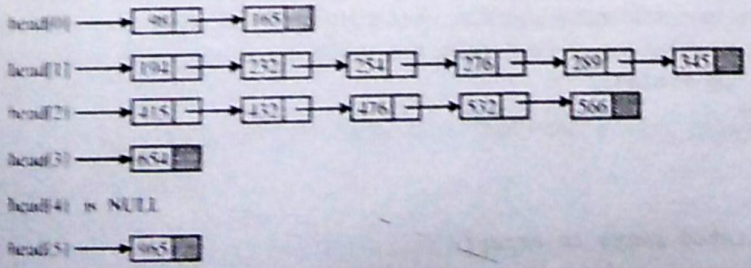
The sorted list that is obtained by concatenating these 6 sets is-
98, 165, 194, 232, 254, 276, 289, 345, 415, 432, 476, 532, 566, 654, 965

For implementing this sorting technique, we can represent each set by a linked list. We know that in each set the elements have to be inserted in sorted order so we will take sorted linked lists. Starting address of each linked list can be maintained in an array of pointers.

In the case of function f(x) which gives us the first digit of an element, there will be 10 linked lists each corresponding to one set. We can take an array of pointers `head[10]`, and each of its element will be a pointer pointing to the first element of these lists. For example `head[0]` will be a pointer that will point to the first element of that linked list in which all elements starting with digit 0 are inserted. Similarly `head[1]`, `head[2]`, ...`head[9]` will be pointers pointing to first elements of other lists.



These linked lists can be easily concatenated to get the final sorted list. In the case of function h(x) discussed before, there will be only 6 linked lists.




```
/*P8.13 Program of sorting using address calculation sort*/
#include<stdio.h>
#include<stdlib.h>
```

```
#define MAX 100
struct node
{
        int info;
        struct node *link;
};
struct node *head[5];
int n,arr[MAX];
int large;
void addr_sort();
void insert(int num,int addr);
int hash_fn(int number);
main()
{
        int i;
        printf("Enter the number of elements in the list : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }/*End of for*/
        for(i=0; i<n; i++)
        {
                if(arr[i] > large)
                        large = arr[i];
        }
        addr_sort();
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d  ",arr[i]);
        printf("\n");
}/*End of main()*/
void addr_sort()
{
        int i,k;
        struct node *p;
        int addr;
        for(i=0; i<=5; i++)
                head[i]=NULL;
        for(i=0; i<n; i++)
        {
                addr = hash_fn(arr[i]);
                insert(arr[i],addr);
        }
        printf("\n");
        for(i=0; i<=5; i++)
        {
                printf("head(%d) -> ",i);
                p = head[i];
                while(p!=NULL)
                {
                        printf("%d ",p->info);
                        p = p->link;
                }
                printf("\n");
        }
        printf("\n");
        /*Taking the elements of linked lists in array*/
        i=0;
        for(k=0; k<=5; k++)
        {
```

```
                p = head[k];
                while(p!=NULL)
                {
                        arr[i++] = p->info;
                        p = p->link;
                }
        }
}/*End of addr_sort()*/
/*Insert the number in sorted linked list*/
void insert(int num,int addr)
{
        struct node *q,*tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = num;
        /*list empty or item to be added in beginning*/
        if(head[addr] == NULL || num < head[addr]->info)
        {
                tmp->link = head[addr];
                head[addr] = tmp;
                return;
        }
        else
        {
                q = head[addr];
                while(q->link != NULL && q->link->info < num)
                        q = q->link;
                tmp->link = q->link;
                q->link = tmp;
        }
}/*End of insert()*/
int hash_fn(int number)
{
        int addr;
        float tmp;
        tmp = (float)number/large;
        addr = tmp*5;
        return(addr);
}/*End of hash_fn()*/
```

8.17.1 Analysis of Address Calculation Sort

In address calculation sort, we maintain sorted linked lists and so the time requirement is mainly dependent on the insertion time of elements in the lists. If any list becomes too long i.e. most of the elements have to be inserted in the same list then the run time becomes close to $O(n^2)$. If all the elements are uniformly distributed among the lists, i.e. almost each element is inserted in a separate list then only little work has to be done to insert the elements in their respective lists and hence the run time becomes linear i.e. O(n). So the run time does not depend on the original order of the data but it depends on how the hashing function distributes the elements among the lists. If the hashing function is such that many elements are mapped into the same list, then the sort is not efficient. This is not an in-place sort since space is needed for nodes of linked lists and header nodes. It is a stable sort.

```
/*P8.14 Program of sorting records using bubble sort algorithm*/
#include <stdio.h>
#define MAX 100
struct record
{
        char name[20];
        int age;
        int salary;
};
```

```
main()
{
        struct record arr[MAX],temp;
        int i,j,n, xchanges;
        printf("Enter the number of records : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter record  %d : \n",i+1);
                printf("Enter name : ");
                scanf("%s",arr[i].name);
                printf("Enter age : ");
                scanf("%d", &arr[i].age);
                printf("Enter salary : ");
                scanf("%d", &arr[i].salary);
                printf("\n");
        }
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(arr[j].age > arr[j+1].age)
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp ;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
        printf("List of Records Sorted on age \n");
        for(i=0; i<n; i++)
        {
                printf("%s\t\t",arr[i].name);
                printf("%d\t",arr[i].age);
                printf("%d\n",arr[i].salary);
        }
        printf("\n");
}/*End of main()*/

/*P8.15 Program to sort the records on different keys using bubble sort*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX 100
struct date
{
        int day;
        int month;
        int year;
};
struct employee
{
        char name[20];
        struct date dob;
        struct date doj;
        int salary;
};
void sort_name(struct employee emp[],int n);
void sort_dob(struct employee emp[],int n);
```

```
void sort_doj(struct employee emp[],int n);
void sort_salary(struct employee emp[],int n);
int datecmp(struct date date1,struct date date2);
main()
{
        struct employee emp[MAX];
        int i,n,choice;
        printf("Enter the number of employees : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter name : ");
                scanf("%s",emp[i].name);
                printf("Enter date of birth(dd/mm/yy): ");
                scanf("%d/%d",&emp[i].dob.day,&emp[i].dob.month);
                scanf("/%d",&emp[i].dob.year);
                printf("Enter date of joining(dd/mm/yy) : ");
                scanf("%d/%d",&emp[i].doj.day,&emp[i].doj.month);
                scanf("/%d",&emp[i].doj.year);
                printf("Enter salary : ");
                scanf("%d",&emp[i].salary);
                printf("\n");
        }
        while(1)
        {
                printf("1.Sort by name alphabetically\n");
                printf("2.Sort by date of birth, in descending order\n");
                printf("3.Sort by date of joining, in descending order\n");
                printf("4.Sort by salary in ascending order\n");
                printf("5.Exit\n");
                printf("Enter your choice :");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                sort_name(emp,n);
                                break;
                        case 2:
                                sort_dob(emp,n);
                                break;
                        case 3:
                                sort_doj(emp,n);
                                break;
                        case 4:
                                sort_salary(emp,n);
                                break;
                        case 5:
                                exit(1);
                        default:
                                printf("Wrong choice\n");
                }/*End of switch*/
                for(i=0; i<n; i++)
                {
                        printf("%s\t\t",emp[i].name);
                        printf("%d/%d",emp[i].dob.day,emp[i].dob.month);
                        printf("/%d\t\t",emp[i].dob.year);
                        printf("%d/%d",emp[i].doj.day,emp[i].doj.month);
                        printf("/%d\t\t",emp[i].doj.year);
                        printf("%d\n",emp[i].salary);
                }
                printf("\n");
        }/*End of while*/
}/*End of main()*/
```

```
void sort_name(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges=0;
                for(j=0; j<n-1-i; j++)
                {
                        if( strcmp(emp[j].name, emp[j+1].name) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
}/*End of sort_name()*/
void sort_dob(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(datecmp(emp[j].dob,emp[j+1].dob) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
}/*End of sort_dob()*/
void sort_doj(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(datecmp(emp[j].doj,emp[j+1].doj ) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
```



```
}/*End of sort_do
void sort_salary(
{       struct em
        int i,j,x
        for(i=0;
        {
                x
                f
                {
                }
}/*End of sort_
/*Returns 1 if
int datecmp(str
{
        if(date
        if(date
        if(date
        if(date
        if(date
        if(date
        return
}/*End of date
/*P8.16 Progra
#include <stdi
#define MAX 10
struct record
{
        char na
        int age
        int sa
};
main()
{
        struct
        int i,
        printf
        scanf(
        for(i=
        {
```

```
}/*End of sort_doj()*/
void sort_salary(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(emp[j].salary > emp[j+1].salary)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
}/*End of sort_salary()*/
/*Returns 1 if date1 < date2, returns -1 if date1 > date2, return 0 if equal*/
int datecmp(struct date date1, struct date date2 )
{
        if(date1.year < date2.year)
                return 1;
        if(date1.year > date2.year)
                return -1;
        if(date1.month < date2.month)
                return 1;
        if(date1.month > date2.month)
                return -1;
        if(date1.day < date2.day)
                return 1;
        if(date1.day > date2.day)
                return -1;
        return 0;
}/*End of datecmp()*/
/*P8.16 Program of sorting by address using bubble sort algorithm*/
#include <stdio.h>
#define MAX 100
struct record
{
        char name[20];
        int age;
        int salary;
};
main()
{
        struct record arr[MAX],*ptr[MAX],*temp;
        int i,j,n, xchanges;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter record %d : \n",i+1);
                printf("Enter name : ");
                scanf("%s",arr[i].name);
                printf("Enter age : ");
                scanf("%d",&arr[i].age);
```

```
                printf("Enter salary : ");
                scanf("%d",&arr[i].salary);
                printf("\n");
                ptr[i] = &arr[i];
        }
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(ptr[j]->age >  ptr[j+1]->age)
                        {
                                temp = ptr[j];
                                ptr[j] = ptr[j+1];
                                ptr[j+1] = temp ;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
        printf("List of Records Sorted on age \n");
        for(i=0; i<n; i++)
        {
                printf("%s\t\t", ptr[i]->name);
                printf("%d\t\t",ptr[i]->age);
                printf("%d\n",ptr[i]->salary);
        }
        printf("\n");
}/*End of main()*/
```

Exercise

1. Show with an example that selection sort is not data sensitive.
2. Write recursive program for sorting an array through selection sort.
3. Write a program to sort an array in descending order using selection sort.
4. Modify the selection sort program given in the chapter so that in each pass the larger element moves towards the end.
5. Show the elements of the following array after four passes of the bubble sort.
   34 23 12 9 45 67 21 89 32 10
6. Modify the bubble sort program so that the smallest element is bubbled up in each pass.
7. Modify the bubble sort given in section 8.9 so that in each pass the sorting is done in both directions. i.e. in each pass the largest element is bubbled up and smallest element is bubbled down. This sorting technique is called bidirectional bubble sort or the shaker sort.
8. Show with the help of an example that selection sort is not a stable sort.
9. Show with the help of an example that bubble sort is a stable sort i.e. it preserves the initial order of the elements.
10. The elements of array after three passes of insertion sort are-
    2 5 20 34 13 19 5 21 89 3
    Show the array after four more passes of the insertion sort are finished.
11. Write a function to sort a linked list using insertion sort.
12. Consider the following array of size 10.
    45 3 12 89 54 15 43 78 28 10
    Selection sort, bubble sort and insertion sort were applied to this array and the contents of the array after three passes in each sort are shown below.
    (i) 3 12 15 43 45 28 10 54 78 89

(ii) 3 10 12 89 54 15 43 78 28 45
(iii) 3 12 45 89 54 15 43 78 28 10
Identify the sorting technique used in each case.

13. The insertion sort algorithm in the chapter uses linear search to find the position for insertion. Modify this algorithm and use binary search instead of linear search.

14. Suppose we are sorting an array of size 10 using quicksort. The elements of array after finishing the first partitioning are -
3 2 1 5 8 12 16 11 9 20
Which element(or elements) could be the pivot.

15. Show the contents of the following array after placing the pivot 47 at proper place.
47 21 23 56 12 87 19 35 11 36 72 12

16. Show how this input is sorted using heap sort.
12 45 21 76 83 97 82 54

17. Write a program to sort a list of strings using bubble sort.

18. Show the elements of the following array after first pass of the shell sort with increment 5.
34 21 65 89 11 54 88 23 76 98 17 51 72 91 24 12