Laboratory Manual

Department of Computer Science and Engineering

National Institute of Technology, Silchar

ASSAM, INDIA

July, 2013

A Laboratory Manual For

Data Structures

Department of
Computer Science & Engineering

Aim

The main aim and objective of conducting laboratory experiment of the Data Structures subject is to helps campus placement of student, to support higher study of student, to make familiar with bugs and to form a background of future research work of the student. Therefore, the student must literate with data organization with effective algorithm. The student can feel that how data are organized for store and retrieve purpose with efficient time and/or space complexity algorithm. Without a laboratory experiment, a student cannot feel the possible errors, test cases, indentation, requirement of comments and real life application of the Data Structures.

Contents

Chapter 1

Introduction

The Data Structures is a core subject in Computer Science & Engineering and it is extensively used in the applied field and research field. This laboratory is conducted due to importance of this subject and to form a strong background of the student. A student can feel the Data organization with its time and space complexity. Furthermore, a student can see the running time of their program or functions using built in time functions.

1.1 Guide to the Student

1.1.1 Common Errors in GCC or CC

There are lots of common errors which are faced by the student during compile time of the program.

1. Segmentation fault: Segmentation fault is occurred due to request of non-permitted memory block or cannot grant the requested memory block due to lacking of memory or violating the rule of memory. Examples are given below:
    - Memory is full, long array[99999999].
    - Stack overflow, infinite recursion.
    - Infinite loop.

- Pointer is assigned to garbage value.
- Request memory block used by other process.
- Request memory block from non-permitted memory location.
- Array out of bound problem.

2. Function conflicting type: It is returned, if a function is being used before definition.

3. Curly braces: Please maintain **Indentation**, while coding, to avoid curly braces error.

4. (.text+0xb4): undefined reference to 'function'
   collect2: error: ld returned 1 exit status
   These statements means the function(...) has not been defined.

5. Infinite loop: Infinite loop does not terminate. Please check the termination condition.

1.1.2 How to write a program

Guidance is given below:

1. Maintain proper Indentation.

2. The name of the file should be meaningful. For example, we are going to implement quick sort using randomize algorithm, then filename can be Quick_Sort_Random_Pivot_Selection.c, so that the people can easily understand by looking the file name.

3. Function and Variable name must have meaning in English dictionary to reflect its properties. e.g, write index1 and index2 instead of i and j respectively and write quick_sort() instead of sort().

4. Write one-line comment after every line of C code and write multiple comment before a block, function, main and header-file.

5. Take huge input to feel the real life applications.

1.1.3 How to approach a problem to solve in better way

Please follow the following steps:

Step I: Understand the problem thoroughly.

Step II: Take an example, try to solve it with pen and paper.

Step III: Generate possible test cases for the problem to solve.

Step IV: Measure the time/space complexity.

Step V: Start coding.

Step VI: Compile and run it.

Step VII: Input the test cases. Generate worst, best and average case(if any) for input to the program.

Step VIII:Conclude its advantages and/or disadvantages.

Step IX: Find out the real life applications.

Step X: Note down the code on your notebook.

1.1.4 DoNotS

1. Do not try to copy the code from other. Try to code and run yourself to feel the problem.
2. Do not use unnecessary temporary variable.
3. Do not start coding without understanding the problem.
4. Do not omit boundary condition of your program.
5. Be careful about divide by zero.
6. Try to avoid unnecessary repeated statements.
7. Be sure about termination of any loop or recursion.

1.2 Guide to the evaluator

When a an evaluator evaluates a program which is done by a student, he/she must follow the following steps:

- Examine Variable names, Functions name, file name so that these are must be in meaningful.
- Examine the comments, indentations.
- Input boundary conditions.
- Input some selective condition.
- Try to crash the program, there must a weakness of their program and if it is not, then program is correct.

Chapter 2

Array

The array is a linear data structure to store and retrieve similar kind of data in contiguous memory location. The advantage of the array is its random accessibility. The disadvantages are wastage of memory, cannot grow or shrink the memory like linked list.

2.1 Aim

Aim of the laboratory of the Array is to introduce the use of arrays in Data Structures in terms of time and space complexity.

2.2 Experiments (Exp)

2.2.1 Exp: Rearranging array elements

Given an array containing +ve and -ve elements. Rearrange the positive and negative elements in O(n) time complexity and O(1) extra space.
Objective: To partition negative and positive element in optimized time and space complexity.
Algorithm:

```
1  void rearrange_numbers(int a[],int n)
2  {
3    int i=0;
4    int k=n-1;
5    while(i<k)
6    {
7      while(a[i]<0)
8        i++;
9      while(a[k]>=0)
10       k--;
11     if(a[i]>=0&&a[k]<0)
12     {
13       swap(a[i],a[k]);    //Exchange a[i] and a[k].
14       i++;
15       k--;
16     }
17   }
18 }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *Input: -1,6,-9,0,2,5,-8,0,-8*

  Output: -1,-8,-9,-8,0,2,5,0,6

- *Input: 1*

  Output: 1

2.2.2 Exp: Repeated array element

Given an array with n elements and most of the elements are repeated. Find the repeated array elements with O(n) and O(n) extra space.

Objective: To understand how hash works.

Algorithm:

```
1  void printRepeating(int arr[], int size)
2  {
3    int temp = 0,i;
4    int b[MAX];    //O(MAX) extra spaces.
5    for(i=0;i<MAX;i++)
6      b[i]=0;
7    for (i = 0; i < size; i++)
8    {
9      b[a[i]]+=1;
10   }
11   printf("Repeating elements are:\n");
12   for(i=0;i<MAX;i++)
13     if(b[i]>1)
14       printf("%d->%d\t",b[i],i);
15 }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *Input: 0,-1*

  Output: Error due to negative input.

- *Input: 1,2,3,6,5,4,2,1,4,3,3,1,2,5,2,3,2*

  Output:

  $1 \rightarrow 3$

$2 \rightarrow 5$

$3 \rightarrow 4$

$4 \rightarrow 2$

$5 \rightarrow 2$

2.2.3 Exp: Repeated array element with space optimization

Given an array with n elements and most of the elements are repeated. Find the repeating array elements with O(n) and O(1) extra space.

Objective: To optimize the space complexity.

Algorithm:

```
1  void printRepeating(int arr[], int size)
2  {
3    int temp = 0,i;
4    int t =0;
5    for (i = 0; i < size; i++)
6    {
7      if(temp & (1 << arr[i]))
8      {
9        if(t &(1<<arr[i]))
10       {
11         printf("%d \t", arr[i]);
12         t = t^(1<<arr[i]);
13       }
14     }
15     else
16     {
17       temp = (temp|(1<<arr[i]));
18       t = (t | (1<<arr[i]));
```

```
19          }
20      }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *input: 0,-1*

  Output: None

- *Input: 5,1,-1,-1,2,10,3,0,0,1,9,7,8, 2,0, 3, 1, 3, 6, 6*

  Output: -1 0 1 2 3 6

2.2.4 Exp: Frequency count

Given an array of size n which contains repeating elements. Write a program to count the frequency of the elements with O(n) time complexity.

Objective: To count the frequency of the elements with O(n) extra space complexity.

Algorithm:

```
1   void Frequency_count(int arr[], int n, int k)
2   {
3     int i,b[k];   //O(k) extra space
4     for(i=0;i<k;k++)
5       b[i]=0;
6     for(i=0;i<n;i++)
7       b[a[i]]+=1;
8     for(i=0;i<k;i++)
9     {
10      if(b[i]>0)
11        printf("%d -> %d\n",i,b[i]);
```

```
12      }
13  }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *input: 0,-1*

  Output: Error due to negative number

- *Input: 1,2,3,6,5,4,2,1,4,3,3,1,2,5,2,3,2*

  Output:

  $1 \rightarrow 3$

  $2 \rightarrow 5$

  $3 \rightarrow 4$

  $4 \rightarrow 2$

  $5 \rightarrow 2$

  $6 \rightarrow 1$

2.2.5 Exp: Binary Search

Given an array containing ordered elements. Write a program to perform binary search to find an elements.

Objective: To discuss various situation about binary search.

Algorithm:

```
1  int binarySearch(int a[],int low, int high, int k)
2  {
3    mid=(low+high)/2;
4    while(low<=high)
5    {
6      if(a[mid]==k)
```

```
7           return 1;
8         else
9         {
10          if(a[high]==a[low])
11            return 0;
12
13          if(a[mid]<k)
14            low=mid+1;
15          else
16            high=mid-1;
17        }
18        mid=(low+high)/2;
19      }
20      return 0;
21    }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *Input: 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,... and k=3*

  Output: Not found in O(1) time complexity.

- *Input: 1,2,3,6,5,4,2,1,4,3,3,1,2,5,2,3,2*

  Output: Not sorted array.

- *Input: 1,2,3,4,5,6 and key=3*

  Output: Found in index 2.

2.2.6 Exp: Maximum and Minimum

Given an array containing n elements. Write a function to find minimum and maximum array element simultaneously.

Objective: To minimize the number of comparison

Algorithm:

```
1  void getMinMax(int arr[], int n)
2  {
3    int min,max;
4    int i;
5    if(n\%2 == 0)
6    {
7      if(arr[0] > arr[1])
8      {
9        max = arr[0];
10       min = arr[1];
11     }
12     else
13     {
14       min = arr[0];
15       max = arr[1];
16     }
17     i = 2;
18   }
19   else
20   {
21     min = arr[0];
22     max = arr[0];
23     i = 1;
24   }
```

```
25      while(i < n-1)      //Pairwise comparisions
26      {
27        if(arr[i] > arr[i+1])
28        {
29          if(arr[i] > max)
30            max = arr[i];
31          if(arr[i+1] < min)
32            min = arr[i+1];
33        }
34        else
35        {
36          if(arr[i+1] > max)
37            max = arr[i+1];
38          if(arr[i] < min)
39            min = arr[i];
40        }
41        i += 2;
42      }
43      print max and min.
44    }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *Input: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2*

  Output: Max=2, Min=2

- *input: -1 4 5 2 7 0 9 6 10 -9*

  Output: Max=10, Min=-9

2.2.7 Exp: Maximum repeated array element

Given an array containing repeating and non-repeating element. Find the maximum repeating number in O(n) time and O(1) extra space.

Objective: To find maximum repeated element in optimized way using hashing.

Algorithm:

```
1  void maxRepeating(int arr[], int n, int k)
2  {
3    for (int i = 0; i< n; i++)
4      arr[arr[i]%k] += k;
5    int max = arr[0], result = 0;
6    for (int i = 1; i < n; i++)
7    {
8      if (arr[i] > max)
9      {
10       max = arr[i];
11       result = i;
12     }
13   }
14   print result;
15 }
```

Test Cases:

- *Input: Empty array.*

  Output: Array is empty.

- *Input: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2*

  Output: Max=2

- *Input: 2 2 3 3 4 5 1 2 4 3 6 9 8 5 8 9 7 8*

  Output: Max=2, 3, 8

- *input: 2 2 3 3 4 5 1 2 4 3 6 9 8 5 8 9 7 8 2*

  Output: Max=2

2.3 Matrix

The Matrix is two dimensional representation of array.

2.3.1 Aim

To demonstrate two dimensional array and its applications.

2.3.2 Exp: Display Matrix spirally

Given $n \times n$ matrix. Write a program to display $n \times n$ matrix spirally.

Objective: How can a matrix be displayed.

Algorithm:

```
1  void Spiral_Matrix(int arr[][], int n)
2  {
3    int i,j,k,m;
4    for(i=n-1, j=0; i > 0; i--, j++)
5    {
6      for(k=j; k < i; k++)
7        printf("%d ", a[j][k]);
8      for(k=j; k < i; k++)
9        printf("%d ", a[k][i]);
10     for(k=i; k > j; k--)
11       printf("%d ", a[i][k]);
12     for(k=i; k > j; k--)
13       printf("%d ", a[k][j]);
14   }
```

```
15    }
16    /*To print middle element of the matrix
17      in case of odd number of matrix size only.*/
18    m = (n-1)/2;
19    if (n%2 == 1)
20      printf("%d", a[m][m]);
21    }
```

Test Cases:

- *Input: Empty Matrix.*

  Output: Matrix is empty.

- *Input:*

  1 2 3

  4 5 6

  7 8 9

  and n=3

  Output: 1 2 3 6 9 8 7 4 5

- *Input: 1*

  Output: 1

- *Input:*

  1 2

  3 4

  Output: 1 2 4 3

2.3.3 Exp: Matrix Rotation

Given $n \times n$ matrix. Write a program to rotate $n \times n$ matrix into 90° angle.

Objective: To rotate an image into into 90° angle.

Algorithm:

```
1  void rotate(int a[n][n],int n)
2  {
3    int i, j,tmp;
4
5        // Transpose the matrix
6       for (i = 0; i < n; i++)
7        {
8            for (j = i + 1; j < n; j++)
9            {
10                tmp = a[i][j];
11                a[i][j] = a[j][i];
12                a[j][i] = tmp;
13           }
14       }
15
16       // mirror the matrix horizontally.
17       for (i = 0; i < n / 2; i++)
18       {
19           for (j = 0; j < n; j++)
20           {
21                tmp = a[j][i];
22                a[j][i] = a[j][n - i - 1];
23               a[j][n - i - 1] = tmp;
24           }
25       }
26 }
```

Test Cases:

- *Input: Empty Matrix.*

Output: Matrix is empty.

- *Input:*

  1 2 3

  4 5 6

  7 8 9

  and n=3

  Output:

  7 4 1

  8 5 2

  9 6 3

- *Input: 1*

  Output: 1

- *Input:*

  1 2

  3 4

  Output:

  3 1

  4 2

2.3.4 Exp: Sparse Matrix Representation

Given $n \times n$ matrix which contains null elements. Write a program to represent the matrix in terms of efficiency of time and space complexity.

Objective: To reduce the wastage of spaces and time.

Algorithm:

```
1  void sparse_rep(int a[][],int n, int sp[][],int *r)
2  {
3     for(i=0;i<n;i++)
```

```
4       {
5           for(j=0;j<n;j++)
6           {
7               if(a[i][j]!=0)
8               {
9                   sp[r][0]=i;
10                  sp[r][1]=j;
11                  sp[r][2]=a[i][j];
12                  (*r)++;         //Call by reference.
13              }
14          }
15      }
16  }
```

Test Cases:

- *Input: Empty Matrix.*

  Output: Matrix is empty.

- *Input:*

  0 2 0 0 0

  0 0 0 3 0

  0 0 1 0 0

  0 0 0 0 4

  0 0 0 0 0

  and n=4

  Output:

  0 1 2

  1 3 3

  2 2 1

  3 4 4

2.3.5 Exp: Sparse Matrix Addition

Given two triplet of sparse matrix representation of $n \times n$ sparse matrix.

Write a program to add two triplet.

Objective: To deal with sparse matrix representation.

Algorithm:

```
int add_sp_mat(int sp1[10][3],int sp2[10][3],int sp3[10][3])
{
  int r,c,i,j,k1,k2,k3,tot1,tot2;

  if( sp1[0][0] != sp2[0][0] || sp1[0][1] != sp2[0][1] )
  {
    printf("Invalid matrix size ");
    exit(0);
  }
  tot1 = sp1[0][2];
  tot2 = sp2[0][2];
  k1 = k2 = k3 = 1;
  while ( k1 <= tot1 && k2 <= tot2)
  {
    if ( sp1[k1][0] < sp2[k2][0] )
    {
      sp3[k3][0] = sp1[k1][0];
      sp3[k3][1] = sp1[k1][1];
      sp3[k3][2] = sp1[k1][2];
      k3++;k1++;
    }
    else
    if ( sp1[k1][0] > sp2[k2][0] )
    {
```

```
25        sp3[k3][0] = sp2[k2][0];
26        sp3[k3][1] = sp2[k2][1];
27        sp3[k3][2] = sp2[k2][2];
28        k3++;k2++;
29      }
30      else if ( sp1[k1][0] == sp2[k2][0] )
31      {
32      if ( sp1[k1][1] < sp2[k2][1] )
33      {
34        sp3[k3][0] = sp1[k1][0];
35        sp3[k3][1] = sp1[k1][1];
36        sp3[k3][2] = sp1[k1][2];
37        k3++;k1++;
38      }
39      else
40      if ( sp1[k1][1] > sp2[k2][1] )
41      {
42        sp3[k3][0] = sp2[k2][0];
43        sp3[k3][1] = sp2[k2][1];
44        sp3[k3][2] = sp2[k2][2];
45        k3++;k2++;
46      }
47      else
48      {
49        sp3[k3][0] = sp2[k2][0];
50        sp3[k3][1] = sp2[k2][1];
51        sp3[k3][2] = sp1[k1][2] + sp2[k2][2];
52        k3++;k2++;k1++;
53      }
54      }
```

```
55    }
56    while ( k1 <=tot1 )
57    {
58      sp3[k3][0] = sp1[k1][0];
59      sp3[k3][1] = sp1[k1][1];
60      sp3[k3][2] = sp1[k1][2];
61      k3++;k1++;
62    }
63
64    while ( k2 <= tot2 )
65    {
66      sp3[k3][0] = sp2[k2][0];
67      sp3[k3][1] = sp2[k2][1];
68      sp3[k3][2] = sp2[k2][2];
69      k3++;k2++;
70    }
71    sp3[0][0] = sp1[0][0];
72    sp3[0][1] = sp1[0][1];
73    sp3[0][2] = k3-1;
74    return;
75  }
```

Test Cases:

- *Input: One Empty Sparse Matrix.*

  Output: Other sparse matrix

- *Input:*

  0 1 2

  1 3 3

  2 2 1

  2 3 6

3 4 4

&

0 1 2

0 3 2

1 3 3

2 2 1

3 4 4

Ouput:

0 1 4

0 3 2

1 3 3

2 2 2

2 3 6

3 4 8

Chapter 3

Linked List

The Linked list is linear data structure to store and retrieve similar kind of data in non-contiguous memory location. The linked list save lots of memory spaces over array, but linked list cannot be accessed randomly. Again, the accessing the element of linked list is slower because of the memory blocks are allocated in non-contiguous memory area.

3.1 Aim

To elaborate use of linked list in the data structure with respect to time and space complexity.

3.2 Experiments Name(Exp)

3.2.1 Exp: Insertion

Write a program to insert an element at front, at last and at any.
Objective: To make familiar with linked list in terms of non-contiguous memory allocation.

Program:

```
#include<stdio.h>
#include<malloc.h>
struct linked_list
{
  int info;
  struct linked_list *next;
};
typedef struct linked_list node;

//Insert an element at front in a linked list
node *insert_at_front(node *head, int n)
{
  node *temp;
  if(head==NULL)  // if the list is empty
  {
    head=(node *)malloc(sizeof(node));
    if(head==NULL)    // if the memory is unavailable
    {
      printf("Memory is not available");
      return;
    }
    head->info=n;     // insertion at empty list
    head->next=NULL;
    return head;
  }
  temp=(node *)malloc(sizeof(node));
  if(temp==NULL)  // if the memory is unavailable
  {
    printf("Memory is not available");
```

```
30        return;
31      }
32      temp->info=n;     // insertion at non-empty list. Insert
33      temp->next=head;
34      head=temp;
35      return head;
36    }
37
38
39
40    //Insert an element in a linked list
41    node *insert_at_last(node *head,int n)
42    {
43      node *start=head,*temp;
44      if(head==NULL)
45      {
46        head=(node *)malloc(sizeof(node));
47        if(head==NULL)          // if the list is empty
48        {
49          printf("Memory is not available");
50          return;
51        }
52        head->info=n;
53        head->next=NULL;
54      }
55      else
56      {
57        while(start->next!=NULL)
58          start=start->next;
59        temp=(node *)malloc(sizeof(node));
```

```
60        if(temp==NULL)          // if the memory is unavailable
61        {
62          printf("Memory is not available");
63          return;
64        }
65        temp->info=n;          //insert at last....
66        temp->next=NULL;
67        start->next=temp;
68      }
69      return head;
70    }
71
72    //insert at given position.
73
74    node * insert_at_any(node *head, int key, int pos)
75    {
76      int count=0;
77      node *p=head,*q=NULL;
78      node *temp=(node *)malloc(sizeof(node));
79      temp->info=key;
80      if(head==NULL)
81      {
82        temp->next=NULL;
83        return temp;
84      }
85      if(pos==0)
86      {
87        temp->next=head;
88        return temp;
89      }
```

```
90      while(p->next!=NULL&&count<=pos)
91      {
92        p=p->next;
93        count++;
94      }
95      if(p->next==NULL)
96      {
97        p->next=temp;
98        temp->next=NULL;
99        return head;
100     }
101     q=p->next;
102     p->next=temp;
103     temp->next=q;
104     return head;
105   }
106
107   void display(node *head)
108   {
109     if(head)
110     {
111       printf("%d->",head->info);
112       display(head->next);
113     }
114   }
115
116   int main()
117   {
118     node *head=NULL;
119     int n,p;
```

```
120     while(1)
121     {
122       printf("Enter the elements of first list
123       (type -9999 to end):");
124       scanf("%d",&n);
125       if(n!=-9999)
126       {
127         head=insert_at_first(head,n);
128         display(head);
129       }
130       else
131         break;
132     }
133     display(head);
134     return 0;
135   }
```

3.2.2 Exp: Deletion

Write a program to delete an element from Linked list.

Objective: To demonstrate freeing a memory block.

Algorithm:

```
1   node *delete_key(node *head, int key)
2   {
3     node *p=head,*q,*r;
4     if(head==NULL)
5     {
6       printf("Empty list");
7       return NULL;
8     }
```

```
9       if(p->info==key)
10      {
11        q=head->next;
12        free(head);
13        return q;
14      }
15      while(p->info!=key&&p->next)
16      {
17        q=p;
18        p=p->next;
19      }
20      if(p->next==NULL&&p->info!=key)
21      {
22        printf("Key not found!");
23        return head;
24      }
25      if(p->next==NULL)&&p->info==key)
26      {
27        free(p);
28        q->next=NULL;
29        return head;
30      }
31      r=p->next;
32      q->next=r;
33      free(p);
34      return head;
35    }
```

3.2.3 Exp: Reverse a Linked List

Write a program to reverse a Linked list.

Objective: To reverse a linked list with O(n) time complexity.

Algorithm:

```
1  node * reverse_linkedlist(node *head)
2  {
3    node *prev=NULL,*curr=head,*nxt=NULL;
4    while(curr)
5    {
6      nxt=curr->next;
7      curr->next=prev;
8      prev=curr;
9      curr=nxt;
10   }
11   return prev;
12 }
```

3.2.4 Exp: Insertion sort

Write a program to insert an element into Linked list so that the output should remain sorted.

Objective: To demonstrate insertion sort.

Algorithm:

```
1 node *insertion_sort(node *head, int n)
2 {
3   node *temp,*start=head,*p;
4   temp=(node *)malloc(sizeof(node));
5   temp->info=n;
6   if(head==NULL)
```

```
7   {
8     head=temp;
9     head->next=NULL;
10    return head;
11  }
12  if(head->next==NULL)
13  {
14    if(head->info>=temp->info)
15    {
16      temp->next=head;
17      head=temp;
18    }
19    else
20    {
21      head->next=temp;
22      temp->next=NULL;
23    }
24    return head;
25  }
26  p=head;
27  while(p->info<n&&p->next)
28  {
29    start=p;
30    p=p->next;
31
32  }
33
34  if(p->info<n&&p->next==NULL)
35  {
36    p->next=temp;
```

```
37      temp->next=NULL;
38      return head;
39    }
40    if(p==head)
41    {
42      temp->next=head;
43      head=temp;
44      return head;
45    }
46    start->next=temp;
47    temp->next=p;
48    return head;
49  }
```

3.2.5 Exp: Merge

Write a program to merge two sorted Linked list to produce sorted output.

Objective: To demonstrate merge using linked list without using extra spaces.

Algorithm:

```
1  //Merging two sorted output. Produce sorted output.
2  // Method: It takes O(1) xtra space.
3
4  node *merge(node *head1,node *head2)
5  {
6    node *p1=head1,*p2=head2,*s,*prev;
7
8    if(head1==NULL && head2==NULL)
9      return NULL;
```

```
10    else
11    if(head1==NULL)
12      return head2;
13    else
14    if(head2==NULL)
15      return head1;
16    else
17    {
18      if(p1->info>p2->info)
19      {
20        s=p2;
21        prev=p2;
22        p2=p2->next;
23      }
24      else
25      {
26        s=p1;
27        prev=p1;
28        p1=p1->next;
29      }
30      while(p1&&p2)
31      {
32
33        if(p1->info>p2->info)
34        {
35          prev->next=p2;
36          prev=prev->next;
37          p2=p2->next;
38
39        }
```

```
40        else
41        {
42          prev->next=p1;
43          prev=prev->next;
44          p1=p1->next;
45        }
46
47      }
48      if(p1)
49        prev->next=p1;
50      else
51        prev->next=p2;
52    }
53
54    return s;
55  }
```

3.2.6 Exp: Detect "Y" shape

Write a program to detect "Y" shape of two Linked list and print the intersection points.

Objective: To detect intersection point.

Algorithm:

```
1  // creating Y shape of two linked list
2  // n is a input from user
3  //where the intersection is made in head1.
4  //
5  node *create_Y_shape(node *head1, node *head2,int n)
6  {
7    int i=0;
```

```
8    node *p1=head1,*p2=head2;
9    while(p2->next!=NULL)
10     p2=p2->next;
11   while(++i!=n)
12     p1=p1->next;
13   p2->next=p1;
14   return head2;
15 }
16
17
18
19
20 //Detecting Y shape
21 //Find the length of both list
22 //i=-1 because the both list can be same length
23
24 void detect_Y_shape(node *head1, node *head2)
25 {
26   int m=0,n=0,i=-1;
27   node *p1=head1,*p2=head2;
28   while(p1!=NULL)
29   {
30     p1=p1->next;
31     m++;
32   }
33   while(p2!=NULL)
34   {
35     p2=p2->next;
36     n++;
37   }
```

```
38    p1=head1;
39    p2=head2;
40    if(m>n)
41    {
42       while(++i!=(m-n))
43          p1=p1->next;
44
45    }
46    else
47    {
48       while(++i!=(n-m))
49          p2=p2->next;
50    }
51    while((p1!=p2)&&p1&&p2)
52    {
53       p1=p1->next;
54       p2=p2->next;
55    }
56    if(p1==p2&&p1)
57       printf("%d is intersecting point\n",p1->info);
58    else
59       printf("No intersecting point\n");
60 }
```

3.2.7 Exp: Detect loop

Write a program to detect loop of a Linked list and print the starting point of the loop.

Objective: To detect defective link.

Algorithm:

```
1  node *make_loop(node *head, int pos)
2  {
3    node *p=head,*p1;
4    int i=0;
5    printf("Here");
6    while(p->next&&++i!=pos)
7      p=p->next;
8    if(p->next==NULL)
9    {
10     p->next=head;
11     return head;
12   }
13   p1=p;
14   while(p1->next)
15     p1=p1->next;
16   p1->next=p;
17   printf("Here");
18   return head;
19 }
20 void break_starting_of_loop(node *head, node **head2,
21  node **head3)
22 {
23   node *p=head,*p1=head,*p2=head->next;
24   int flag=0;
25   while(1)
26   {
27     if(p1==p2)
28     {
29       flag=1;
```

```
30          break;
31        }
32        if(p1==NULL||p2==NULL)
33          break;
34        p1=p1->next;
35        p2=p2->next;
36      }
37      if(flag)
38      {
39
40        while(p!=p1)
41        {
42          p2=p1;
43          p=p->next;
44          p1=p1->next;
45        }
46        p2->next=NULL;
47        printf("Starting of the loop is:%d",p->info);
48        p1=head;
49        p2=head;
50        while(1)
51        {
52          if(p1->next==NULL||p1->next->next==NULL)
53            break;
54          p1=p1->next;
55          p2=p->next->next;
56        }
57        p2=p1->next;
58        *head3=p2;
59        *head2=head;
```

```
60    }
61 }
```

3.2.8 Exp: Finding Middle element

Write a program to find middle element of a list.

Objective: To find middle element in single pass.

Algorithm:

```
1  void find_middle(node *head)
2  {
3    node *ptr=head,*p=head;
4    int flag=0;
5    if(head==NULL)    ///List may empty
6    {
7      printf("Emtpy List\n");
8      return;
9    }
10   while(1)
11   {
12     if(p->next==NULL)          //List may contains odd numbe
13       break;
14     if(p->next->next==NULL)  //List may contains even numbe
15     {
16       flag=1;           //flag is set to zero, if the list co
17       break;
18     }
19     ptr=ptr->next;
20     p=p->next->next;
21   }
22   if(flag)       //if the list contains even numbers of eleme
```

```
23      printf("\nThe middle elements is: %d      %d\n",
24              ptr->info,ptr->next->info);
25    else
26      printf("\nThe middle elements is: %d\n",ptr->info);
27  }
```

3.2.9 Exp: Add two numbers representing by Linked List

Write a program to add two numbers and the digits are represented by linked list and store the result in another linked list.

Objective: To add two numbers dealing with carry.

Algorithm:

```
1  //add two numbers represented by linked list
2  node *add_two_numbers(node *head1,node *head2)
3  {
4    node *result=NULL;
5    int carry=0,n,r;
6    if(head1==NULL)
7      return head2;
8    if(head2==NULL)
9      return head1;
10   while(head1&&head2)
11   {
12     //result=(node *)malloc(sizof(node));
13     n=(int)(head1->info+head2->info);
14     n+=carry;
15     r=n%10;
16     carry=n/10;
17     result=insert_at_first(result,r);
```

```
18      head1=head1->next;
19      head2=head2->next;
20    }
21    if(head2==NULL)
22    {
23      while(head1)
24      {
25        //result=(node *)malloc(sizof(node));
26        n=(int)head1->info;
27        n+=carry;
28        r=n%10;
29        carry=n/10;
30        result=insert_at_first(result,r);
31        head1=head1->next;
32      }
33    }
34    if(head1==NULL)
35    {
36      while(head2)
37      {
38        //result=(node *)malloc(sizof(node));
39        n=(int)head2->info;
40        n+=carry;
41        r=n%10;
42        carry=n/10;
43        result=insert_at_first(result,r);
44        head2=head2->next;
45      }
46    }
47    if(carry)
```

```
48    {
49      result=insert_at_first(result,carry);
50    }
51    return result;
52  }
```

3.2.10 Exp: Rotate Singly Linked List

Write a program to rotate a singly linked list k times.

Objective: To rotate the singly linked list.

Algorithm:

```
1  //rotate a linked list k times
2
3  node *rotate_ll(node *head,int k)
4  {
5    int c=0,i=0;
6    node *p=head,*p1,*p2;
7    if(k<=0)
8      return head;
9    while(p)
10   {
11     c++;
12     p=p->next;
13   }
14   k=k%c;
15   if(k==0)
16     return head;
17   p1=head;
18   while(i!=k)
19   {
```

```
20      p1=p1->next;
21      i++;
22    }
23    p=head;
24    while(p1->next!=NULL)
25    {
26      p=p->next;
27      p1=p1->next;
28    }
29    p2=p->next;
30    p1->next=head;
31    p->next=NULL;
32    return p2;
33  }
```

3.2.11 Exp: Swap node pairwise of a Linked List

Write a program to swap nodes pairwise of a Linked List.

Objective: To swap every pair.

Algorithm:

```
1  node *swap_pairwise(node *head)
2  {
3    node *p1=head,*p2=p1->next;
4    node *p3=p2->next,*p4=p3->next;
5    if(p1==head)
6    {
7      p1->next=p3;
8      p2->next=p1;
9      head=p2;
10     //p1=p1->next;
```

```
11    }
12    while(1)
13    {
14       if(p4==NULL||p4->next==NULL)
15          break;
16       p2=p1->next;
17       p3=p2->next;
18       p4=p3->next;
19       p1->next=p3;
20       p2->next=p4;
21       p3->next=p2;
22       p1=p1->next->next;
23
24    }
25
26    return head;
27 }
```

3.2.12 Exp: Circular Linked List

Write a program to circular linked list.

Objective: To dealing with circular linked list.

Algorithm:

```
1   node *insert_CLL(node *head, int n)
2   {
3      node *temp,*p=head;
4      temp=(node *)malloc(sizeof(node));
5      temp->info=n;
6      if(head==NULL)
7      {
```

```
8         head=temp;
9         head->next=head;
10        return head;
11      }
12      if(head->next==head)
13      {
14        head->next=temp;
15        temp->next=head;
16        return head;
17      }
18      p=head->next;
19      head->next=temp;
20      temp->next=p;
21      return head;
22    }
23
24  void display_CLL(node *head)
25  {
26    node *p=head;
27    while(p->next!=head)
28    {
29      printf("%d->",p->info);
30      p=p->next;
31    }
32    printf("%d->",p->info);
33  }
```

3.2.13 Exp: next pointer points to next node and prev pointer points to sorted order of Double Linked List

Write a program to point next pointer to next node and prev pointer to sorted order of Double Linked List.

Objective: To make familiar with prev and next pointer linked list.

Algorithm:

```
dll *insert_last_sort(dll *head,int n)
{
  dll *temp=(dll *)malloc(sizeof(dll));
  dll *p=head,*min=head,*p1;
  temp->info=n;
  temp->next=NULL;
  if(head==NULL) // At the very begining of the insertion.
  {
    head=temp;
    head->prev=NULL;
    return head;
  }
  if(head->next==NULL)     //If there is only one elements.
  {
    head->next=temp;
    if(head->info>temp->info)
      temp->prev=head;
    else
    {
      head->prev=temp;
      temp->prev=NULL;
    }
```

```
23      return head;
24    }
25    while(p->next)     // Traverse to the last to insert the el
26          // and Min number is searched while traversing the a
27    {
28      p=p->next;
29      if(p->info<min->info)
30        min=p;
31    }
32    p->next=temp;    //insert at last
33    p1=min;      //p1 is the head for prev pointer in ascending
34    while(p1->info<temp->info&&p1->prev)     // Findout where t
35    {            // The correct position for n to be inserted.
36      p=p1;          // p is saved because p1 move ahead.
37      p1=p1->prev;         // n is to be inserted between p and
38    }
39    if(p1->info<n&&p1->prev==NULL) // if n is the largest and
40    {
41      p1->prev=temp;
42      temp->prev=NULL;
43      return head;
44    }
45    if(p1==min)        // if n is the smallest. Insert at first
46    {
47      temp->prev=min;
48      min=temp;
49      return head;
50    }
51    p->prev=temp;        // n is inserted between p and p1;
52    temp->prev=p1;
```

```
53    return head;
54  }
55
56  void display(dll *head)
57  {
58    dll *min=head;
59    while(head->next)
60    {
61      printf("%d\t",head->info);
62      head=head->next;
63      if(min->info>head->info)
64        min=head;
65    }
66    printf("%d\n",head->info); //printing the last elements.
67    while(min->prev)
68    {
69      printf("%d\t",min->info);
70      min=min->prev;
71    }
72    printf("%d\t",min->info); ////printing the last elements.
73    printf("\n");
74  }
```

Chapter 4

Stack

A stack is a basic data structure, where insertion and deletion of items takes place at one end called top of the stack.

4.1 Aim

To demonstrate the importamce of in data structure and in other field of Computer Engineering.

4.2 Experiments (Exp)

4.2.1 Exp: Implementation of Push, Pop and Peek

Implement stack using linked list.

Objective: To demonstrate Push, Pop and Peek function of a Stack.

Algorithm

```
1  struct Stack
2  {
3    node *top;
4  };
```

```
5   typdef struct Stack stack;
6
7   //insert element on top of the stack.
8   void Push(stack **s,int data)
9   {
10    node *tmp=(node *)malloc(sizeof(node));
11    tmp->info=data;
12    if(*s==NULL)
13    {
14      (*s)->top=tmp;
15      (*s)->top->next=NULL;
16    }
17    temp->next=(*s)->top;
18    (*s)->top=temp;
19  }
20
21  //delete element fom the top of the stack.
22  void Pop(stack **s)
23  {
24    node *tmp;
25    if(*s==NULL)
26    {
27      printf("Stack is empty!");
28      return;
29    }
30    tmp=(*s)->top;
31    (*s)->top=(*s)->top->next;
32    free(tmp);
33  }
34
```

```
35   //return top element.
36   int Peek(stack *s)
37   {
38     if(s)
39       return s->top->info;
40     else
41     {
42       printf("Stack is empty!");
43       return;
44     }
45   }
```

4.2.2 Exp: Convert Infix to Postfix

Write a program to convert an expression from infix notation to postfix notation.

Objective: To demonstrate the use of stack.

Algorithm

```
1
2 #include<stdio.h>
3 #include<string.h>
4 #include<math.h>
5
6 #define oper(x) (x=='+' || x=='-' || x=='*' || x=='/')
7
8 char in[30], post[30], stack[30];
9 int top=-1;
10
11 void push(char x)
12 {
```

```
13        stack[++top]=x;
14  }
15
16  char pop()
17  {
18        return stack[top--];
19  }
20
21  int precedence(char c)
22  {
23        if (c=='+' || c=='-')
24            return 1;
25        if (c=='*' || c=='/')
26            return 2;
27        if (c=='(')
28            return 3;
29  }
30
31  int main()
32  {
33        char c;
34        int l,i,j=0,st1[20],k,h,f,eval,s,N;
35        printf("Enter the infix expression : ");
36        scanf("%s",&in);
37       l=strlen(in);
38        for(i=0;i<=l;i++)
39        {
40            if(oper(in[i]))
41            {
42                  post[j++]=' ';
```

```
43                  while(precedence(in[i])<precedence(stack[top
44                  {
45                      post[j++]=stack[top];
46                      pop();
47                      post[j++]=' ';
48
49                  }
50                  push(in[i]);
51              }
52              else if(in[i]=='\0')
53              {
54                  while(top!=-1)
55                  {
56                      post[j++]=' ';
57                      post[j++]=stack[top];
58                      pop();
59                  }
60              }
61              else
62                  post[j++]=in[i];
63          }
64          post[j]='\0';
65          printf("Postfix Expression : %s\n",post);
66          i=0;
67      top=-1;
68      f=0;
69      k=0;
70          while(i<j)
71          {
72              if(oper(post[i]))
```

```
73              {
74                      f=1;
75                      c=post[i];
76                      eval=0;
77                  switch(c)
78                      {
79                          case '+':
80                                  eval=st1[top-1]+st1[top];
81                                  break;
82                          case '-':
83                                  eval=st1[top-1]-st1[top];
84                              break;
85                          case '*':
86                                  eval=st1[top-1]*st1[top];
87                                  break;
88                          case '/':
89                                  eval=st1[top-1]/st1[top];
90                                  break;
91                      }
92                      top--;
93                      st1[top]=eval;
94              }
95              else if(post[i]==' ')
96              {
97                      if(f==0)
98                      {
99                          h=i-k;
100                         s=0;
101                         while(post[h]!=' ')
102                         {
```

```
103                                 N=(int)post[h];
104                                 N=N-48;
105                                 s=s+N*(pow(10,(k-1)));
106                                 k--;
107                                 h++;
108                             }
109                             st1[++top]=s;
110                         }
111                         k=0;
112                 }
113                 else
114                 {
115                         k++;
116                         f=0;
117                 }
118                 i++;
119         }
120         printf("Value : %d\n",st1[top]);
121   return 0;
122 }
```

4.3 Recursion

Although it has certain disadvantage, recursion is being used excessively due to compactness of the codes.

4.3.1 Exp: n^{th} Fibonacci

Write a function of n^{th} fibonacci number using recursion and iteration.

Objective: Comparision of time complexity.

Algorithm:

```
//iteration
int fibo(int n)
{
  for(i=0;i<n;i++)
  {
        c=a+b;
        a=b;
        b=c;
      }
  return c;
}

int fibo(int n)
{
  if(n<=1)
    return n;
  return fibo(n-1)+fibo(n-2);
}
```

4.3.2 Exp: Tower of Hanoi

Write a program to implement hanoi tower.

Objective: To demonstrate stack behaviour in main memory.

Algorithm:

```
void towers(int n,char source,char dest,char inter)
{
  if(n!=0)
  {
```

```
5       towers(n-1,source,inter,dest);
6       printf("\nMove disk %d from  %c to  %c",n,source,dest);
7       towers(n-1,inter,dest,source);
8     }
9   }
```

4.3.3 Exp: Balanced Parenthesis

Write a program to check whether an input is balanced parenthesis or not.

Objective: To demonstrate use of stack.

Algorithm

```
1  void check()
2  {
3    const char leftparen = '(';
4    const char rightparen = ')';
5    bool error = false;
6    char ch;
7    printf("Please enter an algebraic expression: ");
8    scanf("%c",&ch);
9    while (( ch != '\n') && (!error) )
10   {
11     if (ch == leftparen)
12       push(s,ch);
13     if (ch == rightparen)
14     if (isEmpty(s))
15       error = true;
16     else
17       pop(s,ch);
18     scanf("%c",&ch);
```

```
19    }
20    if ((!error) && isEmpty(s))
21      printf("\nResult: This expression is valid. \n");
22    else
23      printf("\nResult: This expression is invalid. \n");
24  }
```

4.3.4 Exp: Catalan Parenthesis

Write a program to print catalan brackets.

Objective: To demonstrate stack behaviour in main memory.

Algorithm

```
1  void printParenthesis(int pos, int n, int open, int close);
2
3  void print_Parenthesis(int n)
4  {
5      if(n > 0)
6          printParenthesis(0, n, 0, 0);
7    return;
8  }
9
10 void printParenthesis(int pos, int n, int open, int close)
11 {
12     static char str[100];
13   if(close == n)
14     {
15         printf("%s\n", str);
16         return;
17     }
18     else
```

```
19      {
20          if(open > close)
21      {
22              str[pos] = ')';
23        printParenthesis(pos+1, n, open, close+1);
24          }
25          if(open < n)
26      {
27              str[pos] = '(';
28              printParenthesis(pos+1, n, open+1, close);
29          }
30      }
31  }
```

4.3.5 Exp: Powerset

Write a program to generate powerset of an input.

Objective: To demonstrate stack behaviour in main memory.

Algorithm:

```
1  void printPowerSet(char *set, int set_size)
2  {
3        unsigned int pow_set_size = pow(2, set_size);
4        int counter, j;
5
6          for(counter = 0; counter < pow_set_size; counter++)
7        {
8            for(j = 0; j < set_size; j++)
9            {
10                       if(counter & (1<<j))
11                 printf("%c", set[j]);
```

```
12              }
13              printf("\n");
14          }
15  }
```

Chapter 5

Queues

Queue is a linear data structure which follows fist-in-first-out fashion. Queue has lots of applications, such like, process queue, flight landing queue etc.

5.1 Aim

To make familiar of Queue and its applications.

5.2 Experiment(Exp)

5.2.1 Exp: Queue using Linked list

Write a program to implement Queue using linked list.

Algorithm:

```
1 struct Queue
2 {
3   node *front;
4   node *rear;
5 };
6 typedef struct Queue Q;
```

```
7  Q *initQ(Q *q)
8  {
9    q=(Q *)malloc(sizeof(Q));
10   q->rear=NULL;
11   q->front=NULL;
12   return q;
13 }
14 Q *insert(Q *q,int key)
15 {
16   if(q->rear==NULL)
17   {
18     q->front=(node *)malloc(sizeof(node));
19     q->front->info=key;
20     q->rear=q->front;
21     return q;
22   }
23   q->rear->next=(node *)malloc(sizeof(node));
24   q->rear->next->info=key;
25   q->rear=q->rear->next;
26   return q;
27 }
28 Q *delete(Q *q)
29 {
30   node *temp=NULL;
31   if(q->front==q->rear)
32   {
33     free(q);
34     q=NULL;
35     printf("Q is empty!\n");
36     return q;
```

```
37    }
38    temp=q->front;
39    q->front=q->front->next;
40    free(temp);
41    return q;
42 }
```

5.2.2 Exp: Queue using array

Write a program implement using queue using array.

Algorithm:

```
1 #include<conio.h>
2 #include<stdio.h>
3 #define N 1000
4
5 int queue[N]={0};
6 int rear=0,front=0;
7
8 void insert(void);
9 void del(void);
10 void disp(void);
11 void cre(void);
12
13 void main()
14 {
15        int user=0;
16        while(user!=4)
17        {
18                  printf("\n\n\n\t\t\t THE SIZE OF QUEUE IS %
19            printf("\n\t  1.INSERT");
```

```
20              printf("\n\t  2.DELETE");
21              printf("\n\t  3.EXIT");
22              scanf("%d",&user);
23              switch(user)
24              {
25                      case  1:
26                           enqueue();
27                           break;
28                      case  2:
29                           dequeue();
30                           break;
31                      case  3:
32                           printf("\n\t THANK U");
33                           break;
34                  }
35          }
36 }
37
38 void  enqueue(void)
39 {
40        int  t;
41      if(rear<N)
42        {
43              printf("\n\t ENTER A VALUE IN QUEUE");
44              scanf("%d",&t);
45              queue[rear]=t;
46              rear++;
47        }
48        else
49        {
```

```
50              printf("\n\t Q OVERFLOW!!!!!!!!!!!!!!!!");
51          }
52  }
53  void dequeue(void)
54  {
55          int i;
56          printf("\n\t %d gets deleted.........",queue[front]);
57          queue[front]=0;
58          front++;
59  }
```

5.2.3 Exp: Circular Queue

Write a program to implement Circular queue.

Objective: To manage circularness of a queue using array.

Algorithm:

```
1   void enqueue(void)
2   {
3           int x;
4           if((front==0&&rear==max-1)||(front==rear+1))
5           {
6                   printf("Queue is overflow\n");return;
7           }
8           if(front==-1)
9           {
10                  front=rear=0;
11          }
12          else
13          {
14                  if(rear==max-1)
```

```
15              {
16                      rear=0;
17              }
18              else
19              {
20                      rear++;
21              }
22      }
23      printf("enter the no:\n");
24      scanf("%d",&x);
25      q[rear]=x;
26  }
27  void dequeue(void)
28  {
29      int y;
30      if(front==-1)
31      {
32              printf("q is underflow\n");return;
33      }
34      y=q[front];
35      if(front==rear)
36      {
37              front=rear=-1;
38      }
39      else
40      {
41              if(front==max-1)
42              {
43                      front=0;
44              }
```

```
45                  else
46                  {
47                          front++;
48                  }
49          }
50  }
```

Chapter 6

Sorting

A sorting algorithm is an algorithm that puts elements of a list in a certain order

6.1 Aim

6.2 Experiments: (Exp)

6.2.1 Exp: Selection Sort

Write a program to implement selection sort.

Objective: To sort with minimum swapping.

Algorithm:

```
1 void selction_sort(int arrr[], int n)
2 {
3   int i,temp,j,min;
4   for(i=0; i<n; i++)
5   {
6     min = i;
7     for(j=i+1; j<n; j++)
8     {
```

```
9         if(array[i]>array[j])
10        {
11          min = j;
12        }
13      }
14      temp = array[i];
15      array[i] = array[min];
16      array[min] = temp;
17    }
18  }
```

6.2.2 Exp: Bubble Sort

Write a program to implement bubble sort.

Objective: To deal with nearly sorted input.

Algorithm:

```
1  void bubble_sort(int arr[], int n)
2  {
3    int i,j,flag=0;
4    for(i=0; i<n; i++)
5
6    {
7
8      flag=0;
9
10     for(j=0; j<n-i; j++)
11
12     {
13
14       if(array[i]>array[j])
```

```
15
16          {
17
18            swap(array[j+1],array[j+1]);
19            flag=1;
20
21          }
22
23        }
24        if(flag==0)
25          break;
26      }
27  }
```

6.2.3 Exp: Insertion Sort

Write a program to implement insertion sort.

Objective: To deal with already sorted input.

Algorithm:

```
1  void insertion_sort(int arr[], int n)
2  {
3    for(i=0;i<n;i++)
4          {
5            value=arr[i];
6              for(j=i-1;j>=0&& value<arr[j];j--)
7                arr[j+1]=arr[j];
8            arr[j+1]=value;
9          }
10 }
```

6.2.4 Exp: Quick Sort

Write a program to implement quick sort.

Objective: To deal with median and randomness.

Algorithm:

```
1  void quicksort( int a[], int low, int high )
2  {
3    int pivot;
4    if ( high > low )
5    {
6      pivot = partition( a, low, high );
7        quicksort( a, low, pivot-1 );
8        quicksort( a, pivot+1, high );
9    }
10 }
11
12
13 int partition( int a[], int low, int high )
14 {
15   int left, right;
16   int pivot_item;
17   int pivot = left = low;
18   int x=low+rand()%high;
19   swap(&a[low],&a[x]);
20   pivot_item = a[low];
21   right = high;
22   while ( left < right )
23   {
24       while( a[left] <= pivot_item )
25         left++;
```

```
26        while( a[right] > pivot_item )
27          right--;
28        if ( left < right )
29          swap(a,left,right);
30    }
31      a[low] = a[right];
32    a[right] = pivot_item;
33    return right;
34  }
```

6.2.5 Exp: Merge Sort

Write a program to implement merge sort.

Objective: To demonstrate divide and conquer.

Algorithm:

```
1  void partition(int arr[],int low,int high){
2
3     int mid;
4
5     if(low<high){
6           mid=(low+high)/2;
7           partition(arr,low,mid);
8           partition(arr,mid+1,high);
9           mergeSort(arr,low,mid,high);
10    }
11 }
12
13 void mergeSort(int arr[],int low,int mid,int high){
14
15    int i,m,k,l,temp[MAX];
```

```
16
17      l=low;
18      i=low;
19      m=mid+1;
20
21      while((l<=mid)&&(m<=high)){
22
23              if(arr[l]<=arr[m]){
24                      temp[i]=arr[l];
25                      l++;
26              }
27              else{
28                      temp[i]=arr[m];
29                      m++;
30              }
31              i++;
32      }
33
34      if(l>mid){
35              for(k=m;k<=high;k++){
36                      temp[i]=arr[k];
37                      i++;
38              }
39      }
40      else{
41              for(k=l;k<=mid;k++){
42                      temp[i]=arr[k];
43                      i++;
44              }
45      }
```

```
46
47      for(k=low;k<=high;k++){
48            arr[k]=temp[k];
49      }
50 }
```

6.2.6 Exp: Counting Sort

Write a program to implement counting sort.

Objective: To sort in O(n) time & space complexity and to demonstrate stable sorting.

Algorithm:

```
1  int Counting_sort(int A[], int k, int n)
2  {
3        int i, j;
4        int B[15], C[100];
5        for(i = 0; i <= k; i++)
6              C[i] = 0;
7        for(j =1; j <= n; j++)
8              C[A[j]] = C[A[j]] + 1;
9        for(i = 1; i <= k; i++)
10             C[i] = C[i] + C[i-1];
11       for(j = n; j >= 1; j--)
12       {
13             B[C[A[j]]] = A[j];
14             C[A[j]] = C[A[j]] - 1;
15       }
16 }
```

6.2.7 Exp: Radix Sort

Write a program to implement Radix sort.

Objective: To reduce space complexity in counting sort.

Algorithm:

```
1  int countSort(int arr[], int n, int exp)
2  {
3        int output[n];
4        int i, count[10] = {0};
5          for (i = 0; i < n; i++)
6           count[ (arr[i]/exp)%10 ]++;
7
8          for (i = 1; i < 10; i++)
9            count[i] += count[i - 1];
10
11       for (i = n - 1; i >= 0; i--)
12       {
13             output[count[(arr[i]/exp)%10 ]-1] = arr[i];
14             count[(arr[i]/exp)%10]--;
15       }
16
17       for (i = 0; i < n; i++)
18           arr[i] = output[i];
19 }
20
21 void radixsort(int arr[], int n)
22 {
23       int m = getMax(arr, n);
24   for (int exp = 1; m/exp > 0; exp *= 10)
25           countSort(arr, n, exp);
```

```
26 }
```

6.2.8 Exp: Bucket Sort

Write a program to implement Bucket sort.

Objective: To sort in O(nk) time complexity.

Algorithm:

```
1  void Bucket_Sort(int array[], int n)
2  {
3    int i, j;
4    int count[n];
5    for(i=0; i < n; i++)
6    {
7        count[i] = 0;
8    }
9    for(i=0; i < n; i++)
10   {
11       (count[array[i]])++;
12   }
13   for(i=0,j=0; i < n; i++)
14   {
15       for(; count[i]>0;(count[i])--)
16       {
17         array[j++] = i;
18       }
19   }
20 }
```

Chapter 7

Tree

7.1 Aim

To demonstrate the importance of Tree in data structure and in other field of Computer Engineering.

7.2 Experiments (Exp)

7.2.1 Exp: Tree traversal of binary tree

Write a program to implement inorder, preorder and postorder traversal of binary tree.

Objective: To demonstrate traversal using recursion.

Algorithm

```
1 struct tree *insert(struct tree *root, int x)
2 {
3    if(!root)
4     {
5          root=(struct tree*)malloc(sizeof(struct tree));
6          root->info = x;
7          root->left = NULL;
```

```
8           root->right = NULL;
9           return(root);
10      }
11      if(root->info > x)
12          root->left = insert(root->left,x);
13      else
14      {
15          if(root->info < x)
16        root->right = insert(root->right,x);
17      }
18       return(root);
19  }
20
21  void inorder(struct tree *root)
22  {
23        if(root != NULL)
24      {
25          inorder(root->left);
26          printf(" %d",root->info);
27          inorder(root->right);
28      }
29   }
30
31   void postorder(struct tree *root)
32   {
33        if(root != NULL)
34      {
35          postorder(root->left);
36          postorder(root->right);
37          printf(" %d",root->info);
```

```
38      }
39  }
40
41  void preorder(struct tree *root)
42  {
43        if(root != NULL)
44      {
45          printf(" %d",root->info);
46          preorder(root->left);
47          preorder(root->right);
48      }
49  }
50
51 struct tree *delet(struct tree *ptr,int x)
52 {
53      struct tree *p1,*p2;
54      if(!ptr)
55      {
56          printf("\n Node not found ");
57          return(ptr);
58      }
59      else
60      {
61          if(ptr->info < x)
62          {
63              ptr->right = delet(ptr->right,x);
64              }
65          else if (ptr->info >x)
66            {
67              ptr->left=delet(ptr->left,x);
```

```
68              return ptr;
69            }
70            else
71            {
72        if(ptr->info == x)
73        {
74          if(ptr->left == ptr->right)
75            {
76              free(ptr);
77              return(NULL);
78            }
79          else if(ptr->left==NULL)
80          {
81              p1=ptr->right;
82            free(ptr);
83            return p1;
84          }
85          else if(ptr->right==NULL)
86          {
87            p1=ptr->left;
88            free(ptr);
89            return p1;
90          }
91          else
92          {
93            p1=ptr->right;
94            p2=ptr->right;
95            while(p1->left != NULL)
96                p1=p1->left;
97            p1->left=ptr->left;
```

```
98              free(ptr);
99              return p2;
100          }
101              }
102           }
103      }
104      return(ptr);
105  }
```

7.2.2 Exp: Inorder traversal using stack

Write a program to implement inorder traversal using stack.

Objective: To use stack push and pop function.

Algorithm

```
1
2  void inOrder(struct tree *root)
3  {
4      struct tree *current = root;
5      struct stack *s = NULL;
6      int done = 0;
7      while (!done)
8      {
9        if(current !=  NULL)
10         {
11               push(&s, current);
12             current = current->left;
13         }
14               else
15         {
16             if (!isEmpty(s))
```

```
17              {
18                current = pop(&s);
19                printf("%d ", current->data);
20                current = current->right;
21              }
22              else
23                done = 1;
24          }
25      }
26  }
```

7.2.3 Exp: Morris traversal

Write a program to implement inorder traversal without using stack and recursion.

Objective: To demonstrate inorder successor.

Algorithm

```
1
2  void MorrisTraversal( struct tree *root)
3  {
4    struct tree *curr,*pre;
5    if(root == NULL)
6          return;
7    curr = root;
8      while(current != NULL)
9      {
10         if(curr->left == NULL)
11         {
12             printf(" %d ", current->data);
13             current = current->right;
```

```
14          }
15          else
16          {
17                  pre = current->left;
18              while(pre->right != NULL && pre->right != curren
19                pre = pre->right;
20              if(pre->right == NULL)
21              {
22                pre->right = current;
23                current = current->left;
24              }
25        else
26              {
27                pre->right = NULL;
28                printf(" %d ",current->data);
29                current = current->right;
30              }
31          }
32      }
33  }
```

7.2.4 Exp: Level order tree traversal using stack

Write a program to print tree elements using levelorder traversal.

Objective: To use queue in tree.

Algorithm

```
1  void LevelOrder( struct tree *root)
2  {
3    struct tree *tmp;
4    struct queue *q;
```

```
5    if(!root)
6      return;
7    q=createQ();
8    enqueue(q,root);
9    while(!IsEmpty(Q))
10   {
11     tmp=dequeue(q);
12     printf("%d",tmp->data);
13     if(tmp->left)
14       enqueue(q,tmp->left);
15     if(tmp->right)
16       enqueue(q,tmp->right);
17   }
18 }
```

7.2.5 Exp: Equal

Write a program to check whether two trees are equal or not.

Algorithm

```
1  int IsEqual(struct tree *root1,struct tree root2)
2  {
3    if(!root1&&!root2)
4      return 1;
5    else
6    if(!root||!root2)
7      return 0;
8    else
9    if(root1->data!=root2->data)
10     return 0;
11   else
```

```
12    return IsEqual(root1->left,root2->left)
13    &&mirror(root1->right,root2->right);
14  }
```

7.2.6 Exp: Mirror

Write a program to check whether two trees are mirror of each other not.

Algorithm

```
1  int mirror(struct tree *root1,struct tree root2)
2  {
3    if(!root1&&!root2)
4      return 1;
5    else
6    if(!root||!root2)
7      return 0;
8    else
9    if(root1->data!=root2->data)
10     return 0;
11   else
12   return mirror(root1->left,root2->right)
13   &&mirror(root1->right,root2->left);
14 }
```

7.2.7 Exp: Height

Write a program to find the height of a binary tree.

Algorithm

```
1
2  int height(struct tree *root)
```

```
3  {
4    int l,r;
5    if(!root)
6      return 0;
7    else
8    {
9      l=heght(root->left);
10     r=height(root->right);
11   }
12   if(l>r)
13     return (l+1);
14   else
15     return (r+1);
16 }
```

7.2.8 Exp: Copy

Write a program to copy a binary tree into other binary tree.

Algorithm

```
1
2  struct tree *copy(struct tree *root)
3  {
4    struct tree *l,*r,*T;
5    if(root)
6    {
7      l=copy(root->left)
8      r=copy(root->right)
9      T=getnode();
10     T->info=root->info;
11     T->left=l;
```

```
12      T->right=r;
13    }
14    return T;
15  }
```

7.2.9 Exp: LCA

Write a program to find out longest common ancestor between two nodes.

Algorithm

```
1
2  void LCA(struct tree *root, struct tree *a,
3  struct tree *b)
4  {
5    struct tree *l,*r;
6    if(!root||root==a||root==b)
7      return root;
8    l=LCA(root->left,a,b);
9    r=LCA(root->right,a,b);
10   if(l&r)
11     return root;
12   else
13     return (l?l:r);
14 }
```

7.2.10 Exp: Heapsort

Write a program to implement Heap sort.

Objective: To demonstrate maxHeapify, Build max heap and heapsort.

Algorithm

```
1  void maxHeapify(int a[], int i,n)
2  {
3    int l,r;
4    l=2*i+1;
5    r=2*i+2;
6    if(l<=n&&a[l]>a[i])
7      largest=l;
8    if(r<=n&&a[r]>a[largest])
9      largest=r;
10   if(largest!=i)
11   {
12     swap(a[i],a[largest]);
13     maxHepify(a,largest,n);
14   }
15 }
16
17 void BuildMaxHeap(int a[],n)
18 {
19   for(i=n/2;i>0;i--)
20     maxHeapify(a,i,n);
21 }
22
23 void HeapSort(int a[],int n)
24 {
25   BuildMaxHeap(a,n)
26   for(i=n;i>1;i--)
27   {
28     swap(a[1],a[i]);
29     n--;
```

```
30      maxHeapify(a,i,n);
31    }
32 }
```

7.2.11 Exp: Insertion and deletion of Binary Search Tree

Write a program to insert and delete a node into/from binary search tree.

Objective: To give implementation idea of binary search tree.

Algorithm

```
1  BST *Insertion(BST *T,int data)
2  {
3    if(T==NULL)
4    {
5      T=getnode();
6      T->data=data;
7      T->left=T->right=NULL;
8    }
9    else
10   {
11     if(data<T->data)
12       T->left=Insertion(T->left,data);
13     else
14     if(data>T->data)
15       T->right=Insert(T->right,data);
16   }
17   return T;
18 }
19
```

```
20 BST *deletion(BST *T,int data)
21 {
22   BST *tmp;
23   if(T==NULL)
24     printf("Element not found!");
25   else
26   if(data<T->left)
27     T->left=deletion(T->left,data);
28   else
29   if(data>T->right)
30     T->right=deletion(T->right,data);
31   else
32   {
33     if(T->left&&T->right)
34     {
35       temp=Max(T->left,T->right);
36       T->data=tmp->data;
37       T->left=deletion(T->left,T->data);
38     }
39     else
40     {
41       tmp=T;
42       if(T->left==NULL)
43         T=T->right;
44       if(T->right==NULL)
45         T=T->left;
46       free(tmp);
47     }
48   }
49   return T;
```

```
50 }
```