The Art of Algorithm Design

Sachi Nandan Mohanty
Pabitra Kumar Tripathy
Suneeta Satpathy

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

The Art of Algorithm Design

The Art of Algorithm Design

Sachi Nandan Mohanty
Pabitra Kumar Tripathy
Suneeta Satpathy

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

First edition published 2022
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-0-367-55511-5 (hbk)
ISBN: 978-0-367-55526-9 (pbk)
ISBN: 978-1-003-09388-6 (ebk)

DOI: 10.1201/9781003093886

Typeset in Minion
by codeMantra

This book is dedicated to my Late Father-in-law,
Sukanta Kumar Mohanty - Sachi Nandan Mohanty

This book is dedicated to my parents - Pabitra Kumar Tripathy

This book is dedicated to my parents - Suneeta Satpathy

Contents

Preface

After gaining a vast experience in the field of teaching, we decided to share our knowledge and the simplest way of teaching methodologies with the learners residing worldwide. This book is designed to provide fundamental knowledge about the theme of the algorithms and their implementations. No doubt in market a number of books related to this field are available, but the specialty of the current book lies in the emphasis on practical implementations of the respective algorithms through discussion of a number of problems along with its theoretical concepts, so as to enhance the skill of the learners.

This book is designed to provide an all-inclusive introduction to the modern study of computer algorithms along with its proper presentation and substantial intensity keeping in mind that *it* would be handy to all levels of readers. The area of expertise behind this book will familiarize simple concepts like recursion, array, stack, queue, link list, and tree to create an easy understanding of the abstraction of data structure. So, the depth of coverage or mathematical rigor is maintained along with the elementary explanations. Furthermore, each chapter of this book is designed with keen detailed presentation of an algorithm, its specific design technique, application areas and related topics to enforce simple understanding among the learners. Few algorithms that are primarily having theoretical interest are also provided with practical alternatives. Moreover, algorithms are designed using "pseudo-codes" to make it more interesting and comfortable for the learners. Each pseudo-code designed for respective algorithm leads to its clarity and briefness, which can further be translated in any programming language as a straightforward task. As we all know that visualization plays a major role in understanding complex concepts, this book also incorporates detailed working procedures of algorithms by using simple graphical images. All the algorithms in this book

are presented with a careful analysis of running times so as to maintain efficiency as a major design criterion.

The text material of this book is primarily intended to be used in undergraduate and graduate courses in algorithm design and data structure. However, the inclusion of engineering issues in algorithms design along with the mathematical apprehensions would make it more comfortable for self-study and well suitable for technical professionals too. Therefore, we look forward to provide the learners with an enjoyable introduction to the field of algorithms with the text material, its step-by-step description, and careful mathematical explanations to eradicate unfamiliarity or difficulty in the subject matter. Learners who have some familiarity with the topic are expected to find the organized chapters with soar preliminary sections proceeding towards more advanced materials.

This book comprises ten separate chapters. Chapter 1 puts forth the preliminary ideas about the concepts of data structures. Chapter 2 narrates the concepts of designing the algorithms and computing the complexities of algorithm as well as the idea about the recurrence relations. Chapter 3 describes the problems related to divide and conquer techniques with practical examples. Chapter 4 discusses the concepts and algorithms related to dynamic programming with appropriate examples. Chapter 5 covers the concepts and algorithms related to greedy choice with apposite examples. Chapter 6 centers on the concepts of different graph algorithms. Chapter 7 emphasizes the approximation algorithm with suitable examples. Chapter 8 deliberates the concepts of matrix operations, linear programming problems and polynomial classes. Chapter 9 highlights different theoretic algorithms. To conclude with, Chapter 10 demonstrates the programming implementations of all the cited algorithms using C-Language.

Authors

Dr. Sachi Nandan Mohanty received his postdoctoral from IIT Kanpur in 2019 and Ph.D. from IIT Kharagpur in 2015, with an MHRD scholarship from the Govt. of India. He has recently joined as an Associate Professor in the Department of Computer Science & Engineering at ICFAI Foundation for Higher Education Hyderabad. Prof. Mohanty's research areas include Data Mining, Big Data Analysis, Cognitive Science, Fuzzy Decision Making, Brain-Computer Interface and Computational Intelligence. Prof. S. N. Mohanty has received three Best Paper Awards during his Ph.D. at IIT Kharagpur from an International Conference on Computer Science & Information Technology at Beijing, China, and another at the International Conference on Soft Computing Applications organized by IIT Roorkee in 2013. He has published 20 research articles in SCI Journals. As a Fellow on Indian Society Technical Education (ISTE), The Institute of Engineering and Technology (IET), Computer Society of India (CSI), Member of Institute of Engineers and IEEE Computer Society, he is actively involved in the activities of the Professional Bodies/Societies.

He has been bestowed with several awards that include "Best Researcher Award from Biju Patnaik University of Technology in 2019," "Best Thesis Award (first Prize) from Computer Society of India in 2015," and "Outstanding Faculty in Engineering Award" from Dept. of Higher Education, Govt. of Odisha in 2020. He has received International Travel funding from SERB, Dept. of Science and Technology, Govt. of India for

chairing a session in international conferences held in the United States in 2020. Currently, he is the reviewer of various journals namely *Journal of Robotics and Autonomous Systems* (Elsevier), *Computational and Structural Biotechnology* (Elsevier), *Artificial Intelligence Review* (Springer) and *Spatial Information Research* (Springer). He has edited books published by Wiley, CRC Press, and Springer Nature.

Mr. Pabitra Kumar Tripathy completed his M.Tech. in Computer Science at Berhampur University, Odisha in 2009. He also completed an M.Sc. in Mathematics at Khallikote Autonomous College Berhampur, Odisha in 2003. He is currently pursuing his Ph.D. in Computer Science and Engineering at Biju Patnaik University of Technology, Rourkela, Odisha. He is working as the Head of Department in the Department of Computer Science and Engineering at Kalam Institute of Technology, Berhampur. He has 15 years of teaching and academic experience. His areas of interest are Computer Graphics, Programming Languages, Algorithms, Theory of Computation, Compiler Design, and Artificial Intelligence. He also has published five international journals and has two patents. He has published five books for graduate students.

Dr. Suneeta Satpathy received her Ph.D. from Utkal University, Bhubaneswar, Odisha in 2015, with a Directorate of Forensic Sciences, MHA scholarship from Govt. of India. She is currently working as an Associate Professor in the Department of Computer Science at Sri Sri University, Cuttack, Odisha, India. Her research interests include Computer Forensics, Cyber Security, Data Fusion, Data Mining, Big Data Analysis, Decision Mining and Machine Learning. In addition to research, she has guided many under- and post-graduate students. She has published papers in many international journals

and conferences in repute. She has two Indian patents in her credit. Her professional activities also include roles as editorial board member and/or reviewer of *Journal of Engineering Science*, *Advancement of Computer Technology and Applications*, *Robotics and Autonomous Systems* (Elsevier) and *Computational and Structural Biotechnology Journal* (Elsevier). She is also editor of several books on different topics such as Digital Forensics, Internet of Things, Machine Learning and Data Analytics published by leading publishers. She is a member of CSI, ISTE, OITS, ACM, IE and IEEE.

CHAPTER 1

Fundamental Concepts of Data Structure

1.1 ARRAY

Whenever we want to store some values, we have to take the help of a variable, and for this, we have to declare it before its use. For example, if we want to store the details of a student, we have to declare the variables as

```
char name [20], add [30];
int roll, age, regdno;
float total, avg;
        etc…
          for an individual student.
```

If we want to store the details of more than one student, then we have to declare a huge number of variables that would increase the length of the program and create difficulty in its access. Therefore, it is better to declare the variables in a group, i.e., the name variable will be used for more than one student, roll variable will be used for more than one student, etc.

Therefore, to declare the variables of the same kind in a group is known as an Array. The concept of array can be used for storing details of more than one student or other objects.

- **Definition**: The array is a collection of more than one element of the same kind with a single variable name.

DOI: 10.1201/9781003093886-1

- **Types of Arrays:**

The arrays can be further classified into two broad categories:

- One-dimensional (the array with one boundary specification)
- Multidimensional (the array with more than one boundary specification)

- One-Dimensional Array

Declaration:

Syntax:

Storage_class Data type variable_name[bound];

The data type may be one of the data types that we have studied. The variable name is also the same as the normal variable_name, but the bound is the number that will further specify the number of variables that you want to combine into a single unit.

Storage_class is optional. By default, it is auto.

Ex: int roll[15];

In the above example, roll is an array of 15 variables that can store the roll_number of 15 students.

In addition, the individual variables are

roll[0], roll[1], roll[2], roll[3], …, roll[14]

1.1.1 Array Element in Memory

The array elements are stored in consecutive memory locations, i.e. the array elements get allocated memory sequentially.

For Ex: int x[7];

If the x[0] is at the memory address 568, then the entire array can be represented in the memory as

x[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]
568	570	572	574	576	578	580

1.1.2 Initialization

The array is initialized just like other normal variables except that we have to pass a group of elements within a chain bracket separated by commas.

Ex: int x[5]= {24, 23, 5, 67, 897};

In the above statement, x[0] = 24, x[1] = 23, x[2] = 5, x[3] = 67, x[4] = 897.

1.1.3 Retrieving and Storing Some Values from/into the Array

Since the array is a collection of more than one element of the same kind, while performing any task with the array, we have to do that work repeatedly. Therefore, while retrieving or storing the elements from/into an array, we have to use the concept of looping.

```
   Ex: Write a Program to Input 10 elements into an
   array and display them.
   #include<stdio.h>
   #include<conio.h>
    int main()
         {
         int x[10],i;
         clrscr();
         printf("\nEnter 10 elements into the array");
                for(i=0 ; i<10; i++)
                   scanf(" %d ",&x[i]);
         printf("\n THE ENTERED ARRAY ELEMENTS ARE :");
                for(i=0 ; i<10; i++)
                   printf(" %4d ",x[i]);
         }
OUTPUT
   Enter 10 elements into the array
   12
   36
   89
   54
   6
   125
   35
   87
   49
   6
   THE ENTERED ARRAY ELEMENTS ARE  :  12   36  89
   54   6 125  35  87  49    6
```

```
/* PROGRAM TO INSERT, DELETE AND SEARCH OF AN
ELEMENT IN AN ARRAY*/
#include<stdio.h>
int insert(int *,int,int,int);
int delete(int *,int,int);
```

```
void input(int *,int);
void display(int *,int);
int search(int *,int,int);
//method to search an element from an array
int search(int *array,int number,int find)
  {
  int i;
    for(i=1;i<=number;i++)
      {  //compare the number to search with array
elements
        if(find == *(array+i))
          {
           return(i);  //return the index
          }
        else
         if(i==number)  //condition for not found
            return(0); //return 0 for failure
       }
    }
    //method to delete an element from an array
int delete(int *array,int number,int position)
 {
   int temp = position;
   while(temp<=number-1)
     {
         *(array+temp)= *(array+(temp+1));
         temp++;
     }
     number= number-1;
     return(number);
 }
 //method to insert an element into an array at
desired location
int insert(int *array,int number,int position,int
element)
  {
   int temp = number;
     while(temp>=position)
       {
          *(array+(temp+1)) = *(array+temp);
          temp -- ;
```

```
        }
      *(array+position)= element;
      number = number+1;
      return(number);
    }
/* INPUT */
  void input(int *array,int number)
    {
      int i;
           for(i=1;i<=number;i++)
            {
             printf("\nEnter the number");
             scanf("%d",array+i);
            }
     }
/*OUT PUT */
   void display(int *array,int number)
     {
       int i;
            for(i=1;i<=number;i++)
             {
              printf("\nValue at the position %d :
          %d",i,*(array+i));
             }
      }
  /* main */
    int main()
     {
      int number,n;
      int *array;
      int position;
      int element;
      int find;
      char ch='y';
      printf("\nInput the number of element into
list");
      scanf("%d",&number);
      array = (int *)malloc(number * 2);
          input(array,number);
           printf("\nEntered list");
      while(ch=='y' || ch=='Y')
           {
```

```
display(array,number);
 printf("\nENTER 1 for INSERT  2 for DELETE 3 FOR
 SEARCH");
 scanf("%d",&ch);
   if(ch==1)
    {
     printf("\nEnter the position to add the data");
     scanf("%d",&position);
     printf("\nEnter the value");
     scanf("%d",&element);
     number = insert(array,number,position,element);
     display(array,number);
    }
 else
      if(ch==2)
      {
        printf("\nEnter the position to delete the
data");
        scanf("%d",&position);
        number =delete(array,number,position);
        display(array,number);
      }
     else
      {
       printf("\nEnter the number to search");
       scanf("%d",&find);
       n=search(array,number,find);
       if(n!=0)
       printf("THE ENTERED NUMBER IS AT %d
POSITION",n);
       else
       printf("\nTHE NUMBER NOT FOUND");
      }
    printf("\nDO YOU WANT TO CONTINUE[y/n]");
    fflush(stdin);
    scanf("%c",&ch);
    }
  return 0;
}
```

OUTPUT

```
Input the number of element into list5

Enter the number12

Enter the number34

Enter the number54

Enter the number66

Enter the number76

Entered list
Value at the position 1 : 12
Value at the position 2 : 34
Value at the position 3 : 54
Value at the position 4 : 66
Value at the position 5 : 76
ENTER 1 for INSERT 2 for DELETE 3 FOR SEARCH3

Enter the number to search54
THE ENTERED NUMBER IS AT 3 POSITION
DO YOU WANT TO CONTINUE[y/n]y

Value at the position 1 : 12
Value at the position 2 : 34
Value at the position 3 : 54
Value at the position 4 : 66
Value at the position 5 : 76
ENTER 1 for INSERT 2 for DELETE 3 FOR SEARCH2

Enter the position to delete the data4

Value at the position 1 : 12
Value at the position 2 : 34
Value at the position 3 : 54
Value at the position 4 : 76
DO YOU WANT TO CONTINUE[y/n]y

Value at the position 1 : 12
Value at the position 2 : 34
Value at the position 3 : 54
Value at the position 4 : 76
ENTER 1 for INSERT 2 for DELETE 3 FOR SEARCH1

Enter the position to add the data4

Enter the value55

Value at the position 1 : 12
Value at the position 2 : 34
Value at the position 3 : 54
Value at the position 4 : 55
Value at the position 5 : 76
DO YOU WANT TO CONTINUE[y/n]n

--------------------------------
Process exited after 39.51 seconds with return value 110
Press any key to continue . . .
```

```
/* TEST SINGULARITY OF A GIVEN MATRIX */
  /* USING CRAMER'S RULES TO FIND DETERMINANT */
# include<stdio.h>
int i, j;
int mat[10][10];
int mat1[10][10];
void display( int, int);
void input( int, int);
int singular(int, int);
/* Output function */
void display( int row, int col)
{
  for(i = 0; i < row; i++)
          {
             for(j = 0; j < col; j++)
             {
                 printf("%5d", mat[i][j]);
             }
             printf("\n");
          }
}
/* Input function */
void input( int row, int col)
{
             for(i = 0 ; i< row; i++)
             {
                 for(j = 0 ;  j<col; j++)
                 {
                     printf("Enter a Number");
                     scanf("%d", &mat[i][j]);
                 }
          }
}
/* Find Determinant using Cramer's rule */
int singular( int row1, int col1)
{
   int i, j, k, l;
   int sum=0, psum=0, partial=0, nsum=0;
   if(row1 == col1)
   {
        printf("\n Number rows  = Number of cols");
        printf("\n Singular Test is possible\n");
```

```
        if(row1 < 3)
        {
             sum = mat[0][0]*mat[1][1] - mat[0]
             [1]*mat[1][0];
             return(sum);
        }
        else
        {
            for(k = 0; k <row1; k++)
                  for(j = 0; j < row1; j++)
                       mat1[k][j] = mat[k][j];
            for(k = 0; k <row1; k++)
                  for(j = row1; j < (2*row1-1); j++)
                       mat1[k][j] = mat1[k][j-row1];
            for(l = 0; l <row1; l++)
            {
                  partial = 1;
                  for(i = 0; i <row1; i++)
                  {
                       partial *= mat1[i][i+l];
                  }
                  psum += partial;
        }
            for(l = row1-1; l < ( 2*row1 -1); l++)
            {
                  partial = 1;
                  for(i =0; i < row1; i++)
                  {
                       partial *=mat1[i][l-i];
                  }
                  nsum += partial;
            }
        sum = psum - nsum ;
             return(sum);
        }
   }
else
        printf("\n Check about singularity is not
        possible");
   return 0;
}
```

```
int main()
{
    int Det;
    int r,c;
    printf("\n Input the number of rows:");
    scanf("%d", &r);
    printf(" Input the number of cols:");
    scanf("%d", &c);
    input(r, c);
    printf("\n Entered  array is as follows:\n");
    display(r, c);
    Det = singular(r, c);
    printf("\n Determinant is : %d", Det);

    if(Det == 0)
         printf("\n The Above matrix is Singular");

    else
         printf("\n Above matrix is not singular");
}
```

OUTPUT

```
 Input the number of rows:3
 Input the number of cols:3
Enter a Number1
Enter a Number3
Enter a Number5
Enter a Number4
Enter a Number2
Enter a Number1
Enter a Number3
Enter a Number4
Enter a Number3

 Entered  array is as follows:
    1    3    5
    4    2    1
    3    4    3

 Number rows  = Number of cols
 Singular Test is possible

 Determinant is : 25
 Above matrix is not singular
--------------------------------
Process exited after 7.099 seconds with return value 30
Press any key to continue . . .
```

1.2 STACK

Stack is a linear data structure that follows the principle of Last in First Out (LIFO). In other words, we can say that if the LIFO principle is implemented with the array, then that will be called the STACK.

The most commonly implemented operations with the stack are PUSH and POP.

Besides these two, more operations can also be implemented with the STACK such as PEEP and UPDATE.

The PUSH operation is known as the INSERT operation, and the POP operation is known as the DELETE operation. During the PUSH operation, we have to check the condition for OVERFLOW, and during the POP operation, we have to check the condition for UNDERFLOW.

- OVERFLOW

If one tries to insert an element in a filled stack, then that situation is called the OVERFLOW condition.

In general, if one tries to insert an element with a filled data structure, then that is called OVERFLOW.

Condition for OVERFLOW

Top = size –1 (for the STACK starts with 0)

Top = size (for the STACK starts with 1)

- UNDERFLOW

If one tries to delete an element from an empty stack, then that situation is called the UNDERFLOW.

In general, if one tries to DELETE an element from an empty data structure, then that is called OVERFLOW.

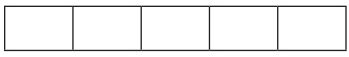
Condition for UNDERFLOW

Top = –1 (for the STACK that starts with 0)

Top = 0 (for the STACK that starts with 1)

Examples

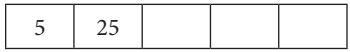
STACK[5]



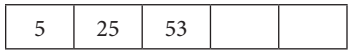
0 1 2 3 4 top = –1 (CONDITION FOR EMPTY STACK)

PUSH(5)

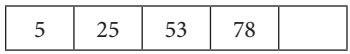
PUSH(25)



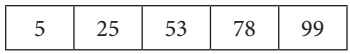
PUSH(53)



PUSH(78)



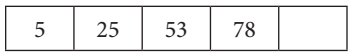
PUSH(99)



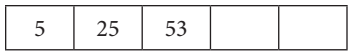
PUSH(145)

"OVERFLOW" (top=size –1 Condition for OVERFLOW)

POP



POP



POP

0 1 2 3 4 top=1

POP

0 1 2 3 4 top=0

POP

0 1 2 3 4 top=–1

POP

"UNDERFLOW" (top= –1 Condition for UNDERFLOW)

1.2.1 Algorithm for Push Operation

```
PUSH(STACK[SIZE], NO, TOP)      [STACK[SIZE] is the Stack]
                                [NO is the Number to Insert]
                                [Top is the position of the stack]
STEP 1: IF (TOP=SIZE - 1) THEN :
                WRITE : "OVERFLOW"
                RETURN
        [END OF IF]
STEP 2: TOP :=TOP +1
            STACK[TOP] := NO
STEP 3: RETURN
```

1.2.2 Algorithm for Pop Operation

```
POP(STACK[SIZE], TOP)           [STACK[SIZE] is the Stack]
                                    [Top is the position of the stack]
STEP 1: IF (TOP=-1) THEN :
                WRITE : "UNDERFLOW"
                RETURN
        [END OF IF]
```

```
STEP 2: WRITE : STACK[TOP]
                TOP := TOP -1
STEP 3: RETURN
```

1.2.3 Algorithm for Traverse Operation

```
TRAVERSE(STACK[SIZE], TOP)      [STACK[SIZE] is the Stack]
                                [Top is the position of the stack]
STEP 1: IF (TOP=-1) THEN :
            WRITE : "STACK IS EMPTY "
            RETURN
            [END OF IF]
STEP 2: SET I :=0
STEP 3: REPEAT FOR I=TOP TO 0 BY -1
            WRITE : STACK[I]
            [END OF LOOP]
STEP 4: RETURN
```

1.2.4 Algorithm for Peep Operation

```
PEEP(STACK[SIZE], NO, TOP)      [STACK[SIZE] is the Stack]
                                [NO is the Number to Search]
                                [Top is the position of the stack]
STEP 1: IF (TOP=- 1) THEN :
        WRITE : "STACK IS EMPTY"
        RETURN
        [END OF IF]
STEP 2: SET I: =0
STEP 3: REPEAT FOR I=TOP TO 0 BY -1
            IF (NO=STACK[I]) THEN:
                WRITE : "NUMBER IS FOUND AT"
                WRITE : TOP-I+1
                WRITE : "POSITION"
              RETURN
            [END OF IF]
            IF I=0 THEN:
                WRITE  : "NUMBER IS NOT FOUND"
            [END OF IF]
          [END OF LOOP]
STEP 4: RETURN
```

OR

```
PEEP(STACK[SIZE], IN, TOP)   [STACK[SIZE] is the Stack]
                             [IN is the Index Number to Search]
                             [Top is the position of the stack]
STEP 1: IF (TOP –IN +1 < 0) THEN :
                WRITE : "OUT OF BOUND"
                RETURN
        [END OF IF]
STEP 2: WRITE : STACK[TOP-IN+1]
STEP 3: RETURN
```

1.2.5 Algorithm for Update Operation

```
UPDATE(STACK[SIZE], NO, TOP)   [STACK[SIZE] is the Stack]
                               [NO is the Number to Update]
                               [Top is the position of the stack]
STEP 1: IF (TOP = - 1) THEN :
                WRITE : "STACK IS EMPTY"
                RETURN
        [END OF IF]

STEP 2: SET I: =0
STEP 3: REPEAT FOR I = TOP TO 0 BY -1
            IF (NO = STACK[I]) THEN:
                    STACK[I] = NO
                RETURN
            [END OF IF]
            IF I=0 THEN:
               WRITE : "UPDATE SUCCESSFULLY NOT COMPLETED"
            [END OF IF]
          [END OF LOOP]
STEP 4: RETURN
```

Program – 1
Wap to perform the PUSH, POP and TRAVERSE operation with the STACK.

```
#include<stdio.h>
#include<alloc.h>
```

```
static int *s, size, top=-1;

void push(int no)
    {
if(top == size-1)
    printf("\n STACK OVERFLOW");
else
 {
   top = top+1;
   *(s+top) = no;
 }
   }
void pop()
  {
if(top == -1)
    printf("\n STACK UNDERFLOW");
  else
    {
  printf("%d IS DELETED", *(s+top));
   --top;
    }
   }
void traverse()
   {
int i;
if(top == -1)
    printf("\n STACK IS EMPTY");
else
   for(i = top; i>=0;i--)
      printf("%5d",*(s+i));
  }
int main()
 { int opt;
printf("\n Enter the size of the stack");
scanf("%d",&size);
s= (int *)malloc(size * sizeof(int));
while(1)
   {
printf("\n Enter the choice");
printf("\n 1.PUSH  2. POP  3. DISPLAY 0. EXIT");
scanf("%d",&opt);
```

```
    if(opt==1)
         {
  printf("\n Enter the number to insert");
  scanf("%d",&opt);
      push(opt);
        }
else
    if(opt==2)
       pop();
   else
     if(opt==3)
       traverse();
      else
        if(opt==0)
           exit(0);
          else
            printf("\n INVALID CHOICE");
    }
}
```

PROGRAM – 2
/* peep operation of the stack using arrays */

```
# include<stdio.h>
# include<ctype.h>
int top = -1,n;
int *s;
/* Definition of the push function */
void push(int d)
{
        if(top ==(n-1))
               printf("\n OVERFLOW");
        else
        {
               ++top;
               *(s+top) = d;
        }
}
/* Definition of the peep function */
void peep()
{
```

```
int i;
      int p;
      printf("\nENTER THE INDEX TO PEEP");
      scanf("%d",&i);

      if((top-i+1) <0)
      {
             Printf("\n OUT OF BOUND");
      }
      else
      {
             Printf("THE PEEPED ELEMENT IS %d",
             *(s+(top-i+1));
      }
}

/* Definition of the display function */

void display()
{
      int i;
      if(top == -1)
      {
             printf("\n Stack is empty");
      }
      else
      {
             for(i = top; i >= 0; --i)
                    printf("\n %d", *(s+i) );
      }
}
int main()           /* Function main */
{        int  no;
         clrscr();
printf("\nEnter the boundary of the stack");
scanf("%d",&n);
      stack = (int *)malloc(n * 2);
     while(1)
      {
      printf("WHICH OPERATION DO YOU WANT TO
      PERFORM:\n");
             printf(" \n 1. Push  2. PEEP 0. EXIT");
```

```
            scanf("%d",&no);
            if(no==1)
            {
                  printf("\n Input the element to
                  push:");
                  scanf("%d", &no);
                  push(no);
                    printf("\n After inserting ");
                             display();
            }
              else
               if(no==2)
                        {
             peep();
                         display();
        }
      Else
        if(no == 0)
                       exit(0);
           else
                      printf("\n INVALID OPTION");
    }
```

PROGRAM – 3
/* update operation of the stack using arrays */

```
# include<stdio.h>
# include<ctype.h>
int top = -1,n;
int flag = 0;
int *stack;
void push(int *, int);
int update(int *);
void display(int *);
/* Definition of the push function */

void push(int *s, int d)
{
       if(top ==n-1)
              flag = 0;
       else
```

```
        {
                flag = 1;
                ++top;
                *(s+top) = d;
        }
}
/* Definition of the update function */
int update(int *s)
{
int i;
      int u;
      printf("\nEnter the index");
      scanf("%d",&i);
      if((top-i+1) <0)
      {
                u = 0;
                flag = 0;
      }
      else
      {
                flag = 1;
                u=*(s+(top-i+1));
                printf("\nENTER THE NUMBER TO UPDATE");
                scanf("%d",s+(top-i+1));
      }
      return (u);
      }
/* Definition of the display function */
void display(int *s)
{
      int i;
      if(top == -1)
      {
                printf("\n Stack is empty");
      }
      else
      {
                for(i = top; i >= 0; --i)
                      printf("\n %d", *(s+i) );
      }
}
```

```
int main()
{
      int  no, q=0;
      char ch;
      int top= -1;
printf("\nEnter the boundary of the stack");
scanf("%d",&n);
      stack = (int *) malloc(n * 2);
    up:
      printf("WHICH OPERATION DO YOU WANT TO
      PERFORM:\n");
            printf(" \n Push->i\n update->p");
            printf("\nInput the choice : ");
            fflush(stdin);
            scanf("%c",&ch);
            printf("Your choice is: %c",ch);
            if(tolower(ch)=='i')
            {
                  printf("\n Input the element to
                  push:");
                  scanf("%d", &no);
                  push(stack, no);
                  if(flag)
                  {
                        printf("\n After inserting ");
                        display(stack);
                        if(top == (n-1))
                              printf("\n Stack is
                              full");
                  }
                  else
                        printf("\n Stack overflow
                        after pushing");
            }
             else
             if(tolower(ch)=='p')
             {
                  no = update(stack);
                  if(flag)
                  {
```

```
                        printf("\n The No %d is
                        updated", no);
                    printf("\n Rest data in stack is as
                    follows:\n");
                        display(stack);
                    }
                    else
                        printf("\n Stack underflow" );
                        }
            opt:
    printf("\nDO YOU WANT TO OPERATE MORE");
    fflush(stdin);
    scanf("%c",&ch);
     if(toupper(ch)=='Y')
      goto up;
        else
          if(tolower(ch)=='n')
            exit();
              else
                {
                  printf("\nINVALID CHARACTER...Try
                  Again");
                   goto opt;
                  }
}
```

1.3 QUEUE

Queue is a linear data structure that follows the principle of FIFO. In other words, we can say that if the FIFO principle is implemented with the array, then that is called the QUEUE.

The most commonly implemented operations with the Queue are INSERT and DELETE.

Besides these two, more operations can also be implemented with the QUEUE such as PEEP and UPDATE.

During the INSERT operation, we have to check the condition for OVERFLOW, and during the DELETE operation, we have to check the condition for UNDERFLOW.

The end at which the insertion operation is performed will be called the REAR end, and the end at which the delete operation is performed is known as the FRONT end.

- **Types of Queue**
  - Linear Queue
  - Circular Queue
  - D – Queue (Double-ended queue)
  - Priority Queue.
- **Linear Queue**
  - OVERFLOW
    If one can try to insert an element with a filled QUEUE, then that situation is called an OVERFLOW condition.
- **Condition for OVERFLOW**

Rear=size –1 (for the QUEUE that starts with 0)
Rear=size (for the QUEUE that starts with 1)

  - UNDERFLOW
    If one can try to delete an element from an empty QUEUE, then that situation is called as UNDERFLOW condition.
- **Condition for UNDERFLOW**

Front=–1 (for the QUEUE that starts with 0)
Front=0 (for the QUEUE that starts with 1)

- **Condition for EMPTY QUEUE**

Front=–1 and Rear=–1 [for the QUEUE that starts with 0]
Front=0 and Rear=0 [for the QUEUE that starts with 1]

Examples

QUEUE[5]

0 1 2 3 4 front=–1 , rear=–1

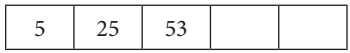
INSERT(5)

0 1 2 3 4 front=0, rear=0

INSERT(25)

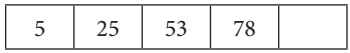
0 1 2 3 4 front=0, rear=1

INSERT(53)



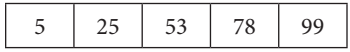
0 1 2 3 4 front=0, rear=2

INSERT(78)



0 1 2 3 4 front=0, rear=3

INSERT(99)

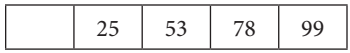


0 1 2 3 4 front=0, rear=4

INSERT(145)

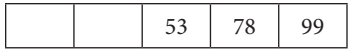
"OVERFLOW" (rear=size -1 Condition for OVERFLOW)

DELETE



0 1 2 3 4 front=1, rear=4

DELETE



0 1 2 3 4 front=2, rear=4

DELETE

0 1 2 3 4 front=3, rear=4

DELETE

DELETE

DELETE

"UNDERFLOW" (front=–1 Condition for UNDERFLOW

1.3.1 Algorithm for Insert Operation

```
INSERT(QUEUE[SIZE], FRONT, REAR, NO)
STEP 1: IF (REAR=SIZE - 1) THEN :
            WRITE : "OVERFLOW"
            RETURN
        [END OF IF]
STEP 2: IF (REAR=-1) THEN :
            FRONT := 0
            REAR :=0
        ELSE :
            REAR :=REAR+1
        [END OF IF]
STEP 3: QUEUE[REAR] :=NO
STEP 4: RETURN
```

1.3.2 Algorithm for Delete Operation

```
DELETE(QUEUE[SIZE], FRONT, REAR)
STEP 1: IF (FRONT=-1) THEN :
            WRITE : "UNDERFLOW"
            RETURN
        [END OF IF]
STEP 2: WRITE: QUEUE[FRONT]
STEP 3: IF (FRONT ==REAR) THEN :
            FRONT := -1
            REAR :=-1
```

```
        ELSE :
            FRONT := FRONT +1
        [END OF IF]
STEP 4: RETURN
```

1.3.3 Algorithm for Traverse Operation

```
TRAVERSE(QUEUE[SIZE], FRONT, REAR)
STEP 1 : IF (FRONT=-1) THEN :
            WRITE: "QUEUE IS EMPTY "
            RETURN
         [END OF IF]
STEP 2 : SET I:=0
STEP 3 : REPEAT FOR I= FRONT TO REAR
            WRITE: QUEUE[I]
         [END OF LOOP]
STEP 4:  RETURN
```

1.3.4 Algorithm for Peep Operation

```
PEEP(QUEUE[SIZE], NO, FRONT, REAR)
                                        [QUEUE[SIZE] is the Stack]
                                        [NO is the Number to Search]
                                        [Front & Rear are the positions of
                                        the stack]
STEP 1: IF (REAR=- 1) THEN :
                WRITE : "STACK IS EMPTY"
                RETURN
          [END OF IF]
STEP 2: SET I: =0
STEP 3: REPEAT FOR I=FRONT TO REAR
              IF (NO=QUEUE[I]) THEN:
                    WRITE : "NUMBER IS FOUND AT"
                    WRITE : I+1
                    WRITE : "POSITION"
                 RETURN
          [END OF IF]
          IF I= REAR THEN:
                WRITE  : "NUMBER IS NOT FOUND"
```

[END OF IF]
[END OF LOOP]
STEP 4: RETURN

1.3.5 Algorithm for Update Operation

UPDATE(QUEUE[SIZE], NO, FRONT, REAR)
[QUEUE[SIZE] is the QUEUE]
[NO is the Number to Update]
[FRONT & REAR is the position of the stack]
STEP 1: IF (REAR = - 1) THEN :
WRITE : "STACK IS EMPTY"
RETURN
[END OF IF]
STEP 2: SET I: =0
STEP 3: REPEAT FOR I = FRONT TO REAR
IF (NO = QUEUE[I]) THEN:
QUEUE[I] = NO
RETURN
[END OF IF]
IF I=REAR THEN:
WRITE : "UPDATE SUCCESSFULLY NOT COMPLETED"
[END OF IF]
[END OF LOOP]
STEP 4: RETURN

PROGRAM – 4
/*INSERTION AND DELETION IN A QUEUE ARRAY IMPLEMENTATION */

```
# include<stdio.h>
int *q,size,front=-1,rear=-1;
void  insert(int n)
{
        if(rear ==size-1)
            printf("\n QUEUE OVERFLOW");
     else
```

```
        {
         rear ++;
                *(q+rear) = n ;
                if(front == -1)
                          front = 0;
         }
}
/* Function to delete an element from queue */
void Delete()
{
        if (front == -1)
        {
                printf("\n Underflow");
                return ;
        }
        printf("\n Element deleted : %d", *(q+front));
        if(front == rear)
        {
                front = -1;
                rear = -1;
        }
        else
                front = front + 1;
}
void display()
{
        int i;
        if (front == -1)
                printf("\n EMPTY QUEUE");
        else
          {
printf("\nTHE QUEUE ELEMENTS ARE");
        for(i = front ; i <= rear; i++)
                printf("%4d", *(q+i));
        }
}
int main()
{
        int opt;
printf("\n Enter the size of the QUEUE");
scanf("%d",&size);
q= (int *) malloc(size * sizeof(int));
```

```
while(1)
   {
printf("\n Enter the choice");
printf("\n 1.INSERT  2. DELETE  3. DISPLAY 0. EXIT");
scanf("%d",&opt);
    if(opt==1)
       {
  printf("\n Enter the number to insert");
  scanf("%d",&opt);
     insert(opt);
       }
else
    if(opt==2)
       Delete();
   else
     if(opt==3)
       display();
      else
        if(opt==0)
           exit(0);
          else
            printf("\n INVALID CHOICE");
    }
}
```

1.4 LINKED LIST

The linked list is the way of representing the data structure that may be linear or nonlinear. The elements in the linked list are allocated memory randomly with a relation in between them. The elements in the linked list are known as NODES.

The link list is quite better than the array due to the proper usage of the memory.

- **Advantage of Link List over the Array**

  The array always requires the memory that is in sequential order, but the linked list requires a single memory allocation that is sufficient to store the data. In the case of an array, the memory may not be allotted even if the available memory space is greater than the required space because that may not be in sequential order.