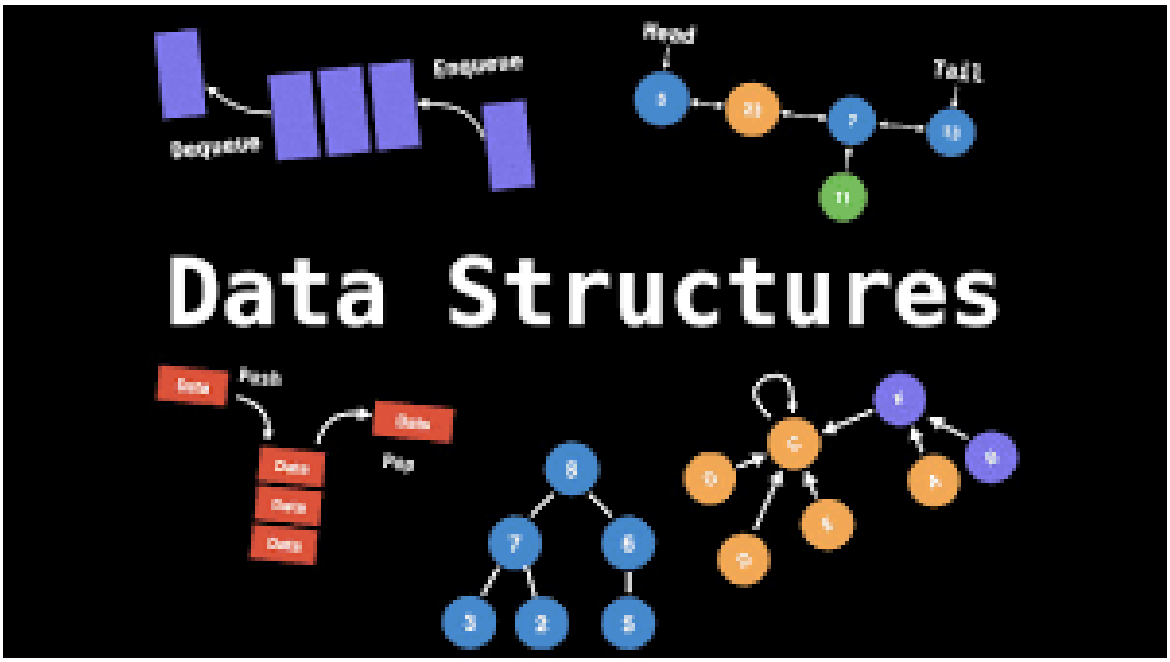
SOUTH VALLEY UNIVERSITY

FACULTY OF COMPUTERS AND INFORMATION

DATA STRUCTURE



1 Programming Methodologies

Programming methodologies deal with different methods of designing programs. This will teach you how to program efficiently. This book restricts itself to the basics of programming in C and C++, by assuming that you are familiar with the syntax of C and C++ and can write, debug and run programs in C and C++. Discussions in this chapter outline the importance of structuring the programs, not only the data pertaining to the solution of a problem but also the programs that operates on the data.

Data is the basic entity or fact that is used in calculation or manipulation process. There are two types of data such as numerical and alphanumerical data. Integer and floating-point numbers are of numerical data type and strings are of alphanumeric data type. Data may be single or a set of values, and it is to be organized in a particular fashion. This organization or structuring of data will have profound impact on the efficiency of the program.

1.1. AN INTRODUCTION TO DATA STRUCTURE

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data Structure = Program

Data structures are the building blocks of a program; here the selection of a particular data structure will help the programmer to design more efficient programs as the complexity and volume of the problems solved by the computer is steadily increasing day by day. The programmers have to strive hard to solve these problems. If the problem is analyzed and divided into sub problems, the task will be much easier *i.e.*, divide, conquer and combine.

A complex problem usually cannot be divided and programmed by set of modules unless its solution is structured or organized. This is because when we divide the big problems into sub problems, these sub problems will be programmed by different programmers or group of programmers. But all the programmers should follow a standard structural method so as to make easy and efficient integration of these modules. Such type of hierarchical structuring of program modules and sub modules should not only reduce the complexity and control the flow of program statements but also promote the proper structuring of information. By choosing a particular structure (or data structure) for the data items, certain data items become friends while others loses its relations.

The representation of a particular data structure in the memory of a computer is called a storage structure. That is, a data structure should be represented in such a way that it utilizes maximum efficiency. The data structure can be represented in both main and auxiliary memory of the computer. A storage structure representation in auxiliary memory is often called a file structure.

It is clear from the above discussion that the data structure and the operations on organized data items can integrally solve the problem using a computer

Data structure = Organized data + Operations

1.2. ALGORITHM

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called *stepwise refinement method.* There are two approaches for algorithm design; they are *top-down* and *bottom-up* algorithm design.

1.3. STEPWISE REFINEMENT TECHNIQUES

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.*,; informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual.

From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion). That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Fig. 1.1.

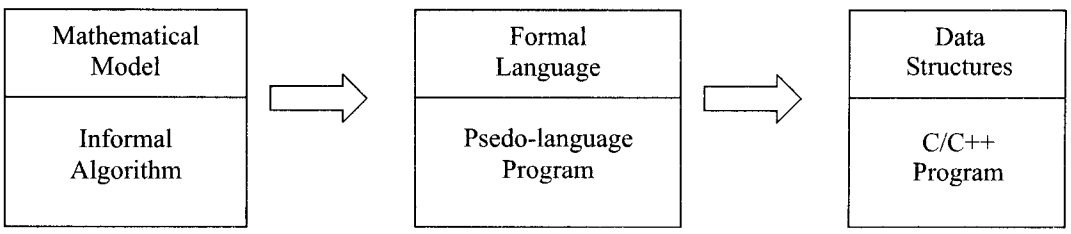


Fig. 1.1

1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Following sections will discuss different programming methodologies to design a program.

1.4. MODULAR PROGRAMMING

Modular Programming is heavily procedural. The focus is entirely on writing code (functions). Data is passive in Modular Programming. Any code may access the contents of any data structure passed to it. (There is no concept of encapsulation.) Modular Programming is the act of designing and writing programs as functions, that each one performs a single well-defined function, and which have minimal interaction between them. That is, the content of each function is cohesive, and there is low coupling between functions.

Modular Programming discourages the use of control variables and flags in parameters; their presence tends to indicate that the caller needs to know too much about how the function is implemented. It encourages splitting of functionality into two types: “Master” functions controls the program flow and primarily contain calls to “Slave” functions that handle low-level details, like moving data between structures.

Two methods may be used for modular programming. They are known as top-down and bottom-up, which we have discussed in the above section. Regardless of whether the top-down or bottom-up method is used, the end result is a modular program. This end result is important, because not all errors may be detected at the time of the initial testing. It is possible that there are still bugs in the program. If an error is discovered after the program supposedly has been fully tested, then the modules concerned can be isolated and retested by them.

Regardless of the design method used, if a program has been written in modular form, it is easier to detect the source of the error and to test it in isolation, than if the program were written as one function.

1.5. TOP-DOWN ALGORITHM DESIGN

The principles of top-down design dictates that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.

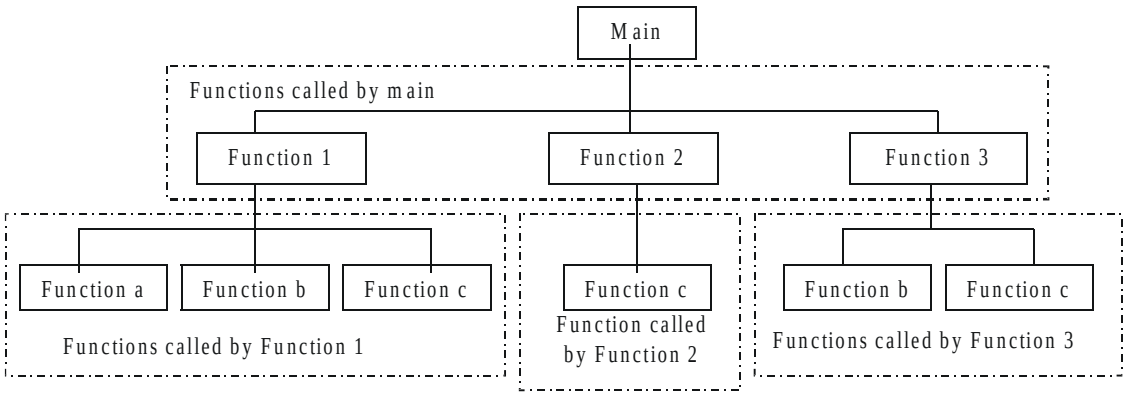


Fig. 1.2

Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

In C, the idea of top-down design is done using functions. A C program is made of one or more functions, one and only one of which must be named *main*. The execution of the program always starts and ends with *main*, but it can call other functions to do special tasks.

1.6. BOTTOM-UP ALGORITHM DESIGN

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and build them into more complex structures, ending at the top.

The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.

1.7. STRUCTURED PROGRAMMING

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

1. Sequence of sequentially executed statements
2. Conditional execution of statements (*i.e.*, "if" statements)
3. Looping or iteration (*i.e.*, "for, do...while, and while" statements)
4. Structured subroutine calls (*i.e.*, functions)

In particular, the following language usage is forbidden:

- "GoTo" statements
- "Break" or "continue" out of the middle of loops
- Multiple exit points to a function/procedure/subroutine (*i.e.*, multiple "return" statements)
- Multiple entry points to a function/procedure/subroutine/method

In this style of programming there is a great risk that implementation details of many data structures have to be shared between functions, and thus globally exposed. This in turn tempts other functions to use these implementation details; thereby creating unwanted dependencies between different parts of the program.

The main disadvantage is that all decisions made from the start of the project depends directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over a time. When that happens, there is a great risk that large parts of the application need to be rewritten.

1.8. ANALYSIS OF ALGORITHM

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements :

1. Space complexity
2. Time complexity

1.8.1. SPACE COMPLEXITY

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion.

Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.
2. We may be interested to know in advance that whether sufficient memory is available to run the program.
3. There may be several possible solutions with different space requirements.
4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

- *Instruction space* : Space needed to store the executable version of the program and it is fixed.
- *Data space* : Space needed to store all constants, variable values and has further two components :
  - (*a*) Space needed by constants and simple variables. This space is fixed.
  - (*b*) Space needed by fixed sized structural variables, such as arrays and structures.
  - (*c*) Dynamically allocated space. This space usually varies.
- *Environment stack space:* This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :
  - (*a*) Return address : *i.e.*, from where it has to resume after completion of the called function.
  - (*b*) Values of all lead variables and the values of formal parameters in the function being invoked .

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

1.8.2. TIME COMPLEXITY

The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used. Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Fig. 1.3.

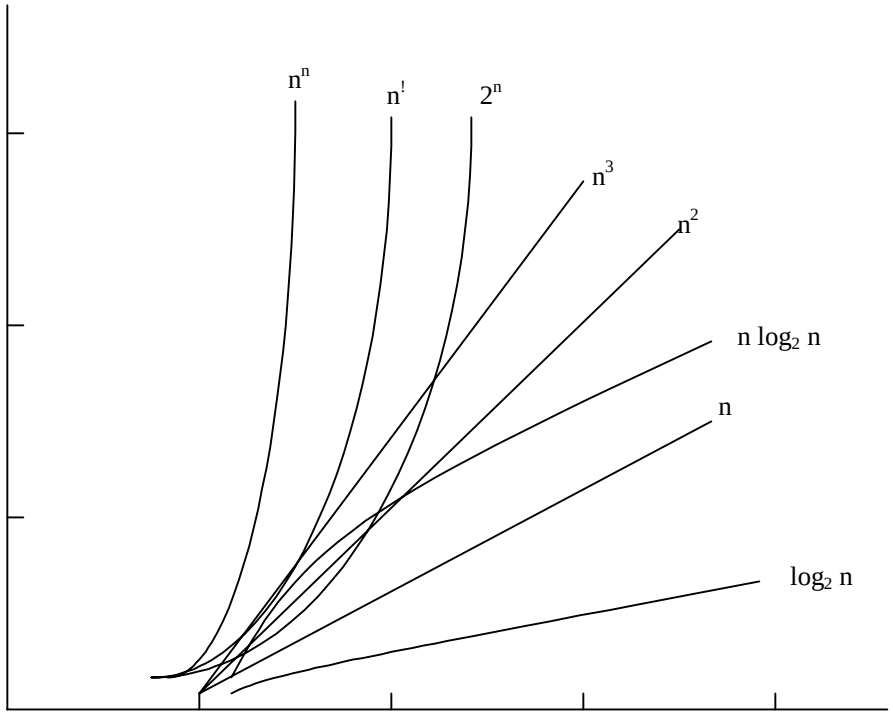


Fig. 1.3

The function that involves '*n*' as an exponent, *i.e.*, 2^n, n^n, n ! are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c,(where '*c*' is a constant) *i.e.*; n^3, n^2, $n \log_2 n$, n, $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

1.8.3. AMSTRONG COMPLEXITY

In many situations, data structures are subjected to a sequence of instructions rather than one set of instruction. In this sequence, one instruction may perform certain modifications that have an impact on other instructions in the sequence at the run time

itself. For example in a *for* loop there are 100 instructions in an *if* statement. If *if* condition is false then these 100 instructions will not be executed. If we apply the time complexity analysis in worst case, entire sequence is considered to compute the efficiency, which is an excessively large and unrealistic analysis of efficiency. But when we apply amortized complexity, the complexity is calculated when the instructions are executed (*i.e.*, when *if* condition is true)

Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

1.9. TIME-SPACE TRADE OFF

There may be more than one approach (or algorithm) to solve a problem. The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real time system, we have to choose a program that takes less time to complete its execution at the cost of more space.

1.10. BIG "OH" NOTATION

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function $f(n)$ of an algorithm increases as 'n' increases.

Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' *i.e.*, the worst case. That is the target value will be found at the nth position of the array.

$$f(n) = n$$

i.e., the worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array.

The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (*i.e.*, we need to compare the target value with only one element from the array)—we have found the element by executing only one iteration (or by least possible statements)

$$f(n) = 1$$

Average case falls between these two extremes (*i.e.*, best and worst). If the target value is found at the $n/2$nd position, on an average we need to compare the target value with only half of the elements in the array, so

$$f(n) = n/2$$

The complexity function f(n) of an algorithm increases as '*n*' increases. The function $f(n) = O(n)$ can be read as "*f* of *n* is big Oh of *n*" or as "$f(n)$ is of the order of *n*". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

The generalized form of the theorem is

$$f(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots\ldots + c_2 n^2 + c_1 n^1 + c_0 n^0$$

Where the constant $c_k > 0$

Then, $f(n) = O(n^k)$

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :

1. Constant time O(1)
2. Logarithmic time Olog(*n*)
3. Linear time O(*n*)
4. Polynomial time $O(n^c)$
5. Exponential time $O(c^n)$

} Where $c > 1$

1.11. LIMITATION OF BIG "OH" NOTATION

Big Oh Notation has following two basic limitations :

1. It contains no effort to improve the programming methodology. Big Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyze and calculate the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constants. For example, one algorithm is taking $1000n^2$ time to execute and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

We will analyze and design the problems in data structure. As we have discussed to develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

1.12. CLASSIFICATION OF DATA STRUCTURE

Data structures are broadly divided into two :

1. *Primitive data structures* : These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc. These primitive data structures are the basis for the discussion of more sophisticated (non-primitive) data structures in this book.

2. *Non-primitive data structures* : It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.

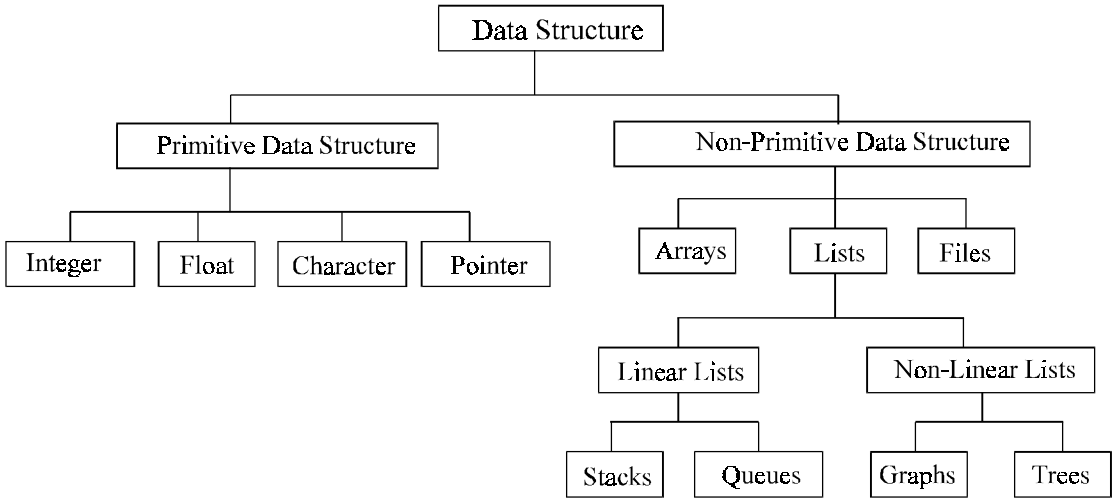


Fig. 1.4. Classifications of data structures

The Fig. 1.4 will briefly explain other classifications of data structures. Basic operations on data structure are to create a (non-primitive) data structure; which is considered to be the first step of writing a program. For example, in Pascal, C and C++, variables are created by using declaration statements.

```
int Int_Variable;
```

In C/C++, memory space is allocated for the variable "*Int_Variable*" when the above declaration statement executes. That is a data structure is created. Discussions on primitive data structures are beyond the scope of this book. Let us consider non-primitive data structures.

1.13. ARRAYS

Arrays are most frequently used in programming. Mathematical problems like matrix, algebra and etc can be easily handled by arrays. An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional array and so on. Analogously the arrays whose elements are referenced by two or more subscripts are called multi dimensional arrays.

1.13.1. ONE DIMENSIONAL ARRAY

One-dimensional array (or linear array) is a set of '*n*' finite numbers of homogenous data elements such as :

1. The elements of the array are referenced respectively by an index set consisting of '*n*' consecutive numbers.
2. The elements of the array are stored respectively in successive memory locations.

'*n*' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as

A[0], A[1], A[2], ... A[*n* –1].

The number '*n*' in A[*n*] is called a subscript or an index and A[n] is called a subscripted variable. If '*n*' is 10, then the array elements A[0], A[1].....A[9] are stored in sequential memory locations as follows :

A[0]	A[1]	A[2]	.....	A[9]

In C, array can always be read or written through loop. To read a one-dimensional array, it requires one loop for reading and writing the array, for example:

For reading an array of '*n*' elements

```
for (i = 0; i < n; i ++)
scanf ("%d",&a[i]);
```

For writing an array

```
for (i = 0; i < n; i ++)
printf ("%d", &a[i]);
```

1.13.2. MULTI DIMENSIONAL ARRAY

If we are reading or writing two-dimensional array, two loops are required. Similarly the array of '*n*' dimensions would require '*n*' loops. The structure of the two dimensional array is illustrated in the following figure :

```
int A[10][10];
```

A_{00}	A_{01}	A_{02}						A_{08}	A_{09}
A_{10}	A_{11}								A_{19}
A_{20}									
A_{30}									
									A_{69}
A_{70}								A_{78}	A_{79}
A_{80}	A_{81}						A_{87}	A_{88}	A_{89}
A_{90}	A_{91}	A_{92}				A_{96}	A_{97}	A_{98}	A_{99}

1.13.3. SPARSE ARRAYS

Sparse array is an important application of arrays. A sparse array is an array where nearly all of the elements have the same value (usually zero) and this value is a constant. One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called sparse matrix.

The main objective of using arrays is to minimize the memory space requirement and to improve the execution speed of a program. This can be achieved by allocating memory space for only non-zero elements.

For example a sparse array can be viewed as

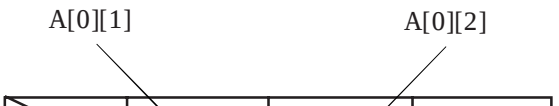
0	0	8	0	0	0	0
0	1	0	0	0	9	0
0	0	0	3	0	0	0
0	31	0	0	0	4	0
0	0	0	0	7	0	0

Fig. 1.5. Sparse array

We will store only non-zero elements in the above sparse matrix because storing all the elements of the sparse array will be consisting of memory sparse. The non-zero elements are stored in an array of the form.

A[0……*n*][1……3]

Where '*n*' is the number of non-zero elements in the array. In the above Fig. 1.4 '*n* = 7'. The space array given in Fig. 1.4 may be represented in the array A[0……7][1…..3].



	1	**2**	**3**
0	5	7	7
1	1	3	8
2	2	2	1
3	2	6	9
4	3	4	3
5	4	2	31
6	4	6	4
7	5	5	7

Fig. 1.6. Sparse array representation

The element A[0][1] and A[0][2] contain the number of rows and columns of the sparse array. A[0][3] contains the total number of nonzero elements in the sparse array.

A[1][1] contains the number of the row where the first nonzero element is present in the sparse array. A[1][2] contains the number of the column of the corresponding nonzero element. A[1][3] contains the value of the nonzero element. In the Fig. 1.4, the first nonzero element can be found at 1st row in 3rd column.

1.14. VECTORS

A vector is a one-dimensional ordered collection of numbers. Normally, a number of contiguous memory locations are sequentially allocated to the vector. A vector size is fixed and, therefore, requires a fixed number of memory locations. A vector can be a column vector which represents a '*n*' by 1 ordered collections, or a row vector which represents a 1 by '*n*' ordered collections.

The column vector appears symbolically as follows :

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ \vdots \\ A_n \end{pmatrix}$$

A row vector appears symbolically as follows :

$$A = (A_1, A_2, A_3, \ldots\ldots A_n)$$

Vectors can contain either real or complex numbers. When they contain real numbers, they are sometime called real vectors. When they contain complex numbers, they are called complex vectors.

1.15. LISTS

As we have discussed, an array is an ordered set, which consist of a fixed number of elements. No deletion or insertion operations are performed on arrays. Another main disadvantage is its fixed length; we cannot add elements to the array. Lists overcome all the above limitations. A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list. Any other list is said to be non linear. List can be implemented by using pointers. Each element is referred to as nodes; therefore a list can be defined as a collection of nodes as shown below :

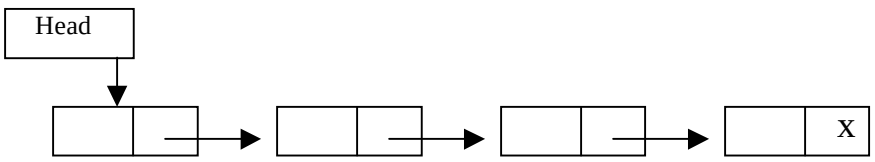


Fig. 1.7

1.16. FILES AND RECORDS

A file is typically a large list that is stored in the external memory (*e.g.*, a magnetic disk) of a computer.

A record is a collection of information (or data items) about a particular entity. More specifically, a record is a collection of related data items, each of which is called a filed or attribute and a file is a collection of similar records.

Although a record is a collection of data items, it differs from a linear array in the following ways:

(*a*) A record may be a collection of non-homogeneous data; *i.e.*, the data items in a record may have different data types.

(*b*) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

1.17. CHARACTERISTICS OF STRINGS

In computer terminology the term 'string' refers to a sequence of characters. A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string.

1.17.1. STRING REPRESENTATION

Strings are stored or represented in memory by using following three types of structures :

- Fixed length structures
- Variable length structures with fixed maximum
- Linear structures

FIXED LENGTH REPRESENTATION. In fixed length storage each line is viewed as a record, where all records have the same length. That is each record accommodates maximum of same number of characters.

The main advantage of representing the string in the above way is :

1. To access data from any given record easily.
2. It is easy to update the data in any given record.

The main disadvantages are :

1. Entire record will be read even if most of the storage consists of inessential blank space. Time is wasted in reading these blank spaces.
2. The length of certain records will be more than the fixed length. That is certain records may require more memory space than available.

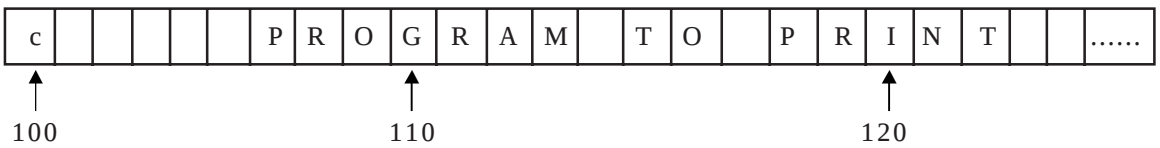


Fig. 1.8. Input data

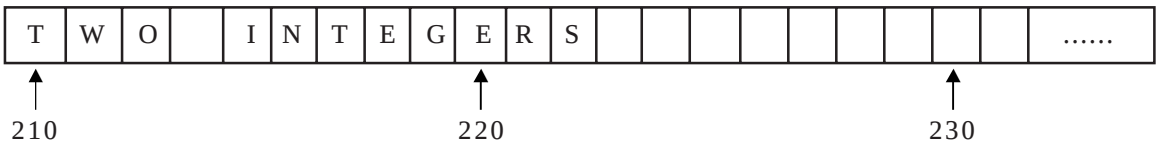


Fig. 1.9. Fixed length representation

Fig. 1.9 is a representation of input data (which is in Fig. 1.8) in a fixed length (records) storage media in a computer.

Variable Length Representation: In variable length representation, strings are stored in a fixed length storage medium. This is done in two ways.

1. One can use a marker, (any special characters) such as two-dollar sign ($$), to signal the end of the string.
2. Listing the length of the string at the first place is another way of representing strings in this method.

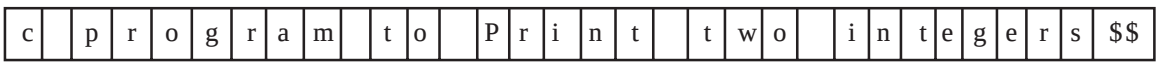


Fig. 1.10. String representation using marker

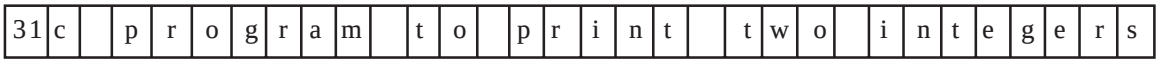


Fig. 1.11. String representation by listing the length

Linked List Representations: In linked list representations each characters in a string are sequentially arranged in memory cells, called nodes, where each node contain an item and link, which points to the next node in the list (*i.e.*, link contain the address of the next node).

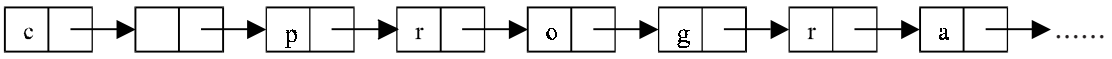


Fig. 1.12. One character per node

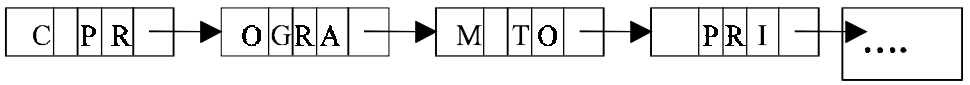


Fig. 1.13. Four character per node

We will discuss the implementation issues of linked list in chapter 5.

1.17.2. SUB STRING

Group of consecutive elements or characters in a string (or sentence) is called sub string. This group of consecutive elements may not have any special meaning. To access a sub string from the given string we need following information :

(*a*) Name of the string

(*b*) Position of the first character of the sub string in the given string

(*c*) The length of the sub string

Finding whether the sub string is available in a string by matching its characters is called pattern matching.

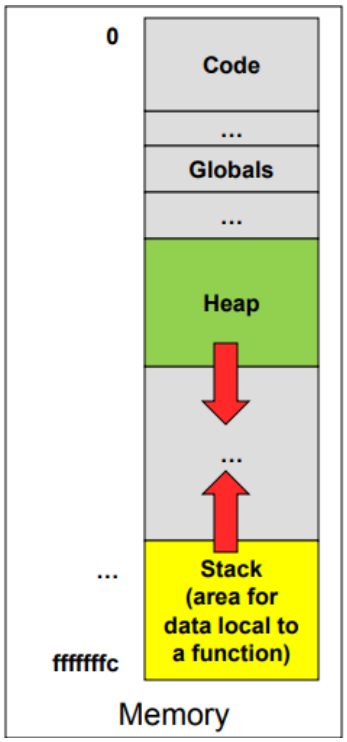
What is Memory Management?

Memory management can be defined as a process in which management of a computer's memory occurs, for example assigning memory to programs, variables etc., in such a way that it doesn't affect the overall performance. Sometimes, the computer's data can range up to terabytes, so efficient use of memory is necessary to minimize memory wastage and boost up performance.

Why Do We Need Memory Management, and How Does It Work?

Memory management is required to ensure that there is no wastage of memory and that allocation takes place efficiently. The memory that a C++ program uses is divided into different parts. Here, In C++, we can divide a program's memory into three parts:

1. **Static region**, wherein static variables are stored. Static variables are variables that remain in use throughout the execution of a program. The size of the static region does not change during the C++ program's execution.
2. **Stack**, wherein stack frames are stored. A new stack frame is created for every function call. A stack frame is a frame of data that contains the corresponding function's local variables and is destroyed (popped) when that function returns.
3. **Heap**, wherein dynamically allocated memory is stored. To optimize memory utilization, heap and stack typically grow towards each other, as illustrated in the following figure



During the declaration of an array, there are times when the correct memory is not determined until runtime. To avoid such scenarios, we usually declare an array of the maximum size. However, because of this full size, some memory remains unused. For example, let us suppose we have declared an array of size 30, and after declaring the array, it turns out that we only need space of 10 size, so the rest of the space is of no use, or we can say it will get wasted.

To avoid such cases, we use memory allocation. We can allocate the memory at runtime from the heap using an operator.

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.

Memory Management Operators

In C language, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax

1. pointer_variable = **new** data-type

The above syntax is used to create the object using the new operator. In the above syntax, '**pointer_variable**' is the name of the pointer variable, '**new**' is the operator, and '**data-type**' defines the type of the data.

Example 1:

```
1. int *p;
2. p = new int;
```

In the above example, 'p' is a pointer of type int.

Example 2:

1. **float** *q;
2. q = **new float**;

In the above example, 'q' is a pointer of type float.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

1. **int** *p = **new int**;
2. **float** *q = **new float**;

Assigning a value to the newly created object

Two ways of assigning values to the newly created object:

- We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

1. *p = 45;
2. *q = 9.8;

We assign 45 to the newly created int object and 9.8 to the newly created float object.

- We can also assign the values by using new operator which can be done as follows:

1. pointer_variable = **new** data-type(value);

Let's look at some examples.

1. **int** *p = **new int**(45);
2. **float** *p = **new float**(9.8);

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

1. pointer-variable = **new** data-type[size];

Examples:

1. **int** *a1 = **new int**[8];

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

1. **delete** pointer_variable;

In the above statement, **'delete'** is the operator used to delete the existing object, and **'pointer_variable'** is the name of the pointer variable.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

1. **delete** p;
2. **delete** q;

The dynamically allocated array can also be removed from the memory space by using the following syntax:

1. **delete** [size] pointer_variable;

In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:

1. **delete** [ ] pointer_variable;

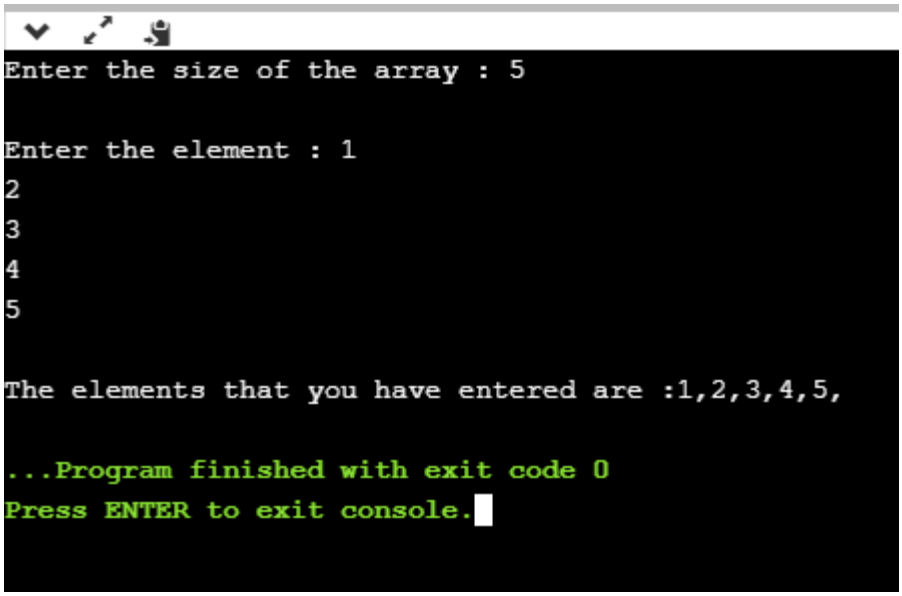
Let's understand through a simple example:

```
1. #include <iostream>
2. using namespace std
3. int main()
4. {
5. int size;  // variable declaration
6. int *arr = new int[size];   // creating an array
7. cout<<"Enter the size of the array : ";
8. std::cin >> size;    //
9. cout<<"\nEnter the element : ";
```

```
10. for(int i=0;i<size;i++)   // for loop
11. {
12. cin>>arr[i];
13. }
14. cout<<"\nThe elements that you have entered are :";
15. for(int i=0;i<size;i++)    // for loop
16. {
17. cout<<arr[i]<<",";
18. }
19. delete arr;  // deleting an existing array.
20. return 0;
21. }
```

In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement **delete arr**.

Output



Advantages of the new operator

The following are the advantages of the new operator over malloc() function:

- It does not use the sizeof() operator as it automatically computes the size of the data object.
- It automatically returns the correct data type pointer, so it does not need to use the typecasting.
- Like other operators, the new and delete operator can also be overloaded.
- It also allows you to initialize the data object while creating the memory space for the object.

malloc() vs new in C++

Both the **malloc()** and new in C++ are used for the same purpose. They are used for allocating memory at the runtime. But, malloc() and new have different syntax. The main difference between the malloc() and new is that the new is an operator while malloc() is a standard library function that is predefined in a **stdlib** header file.

What is new?

The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable. The functionality of the new operator in C++ is similar to the malloc() function, which was used in the C programming language. C++ is compatible with the malloc() function also, but the new operator is mostly used because of its advantages.

Syntax of new operator

1. type variable = **new** type(parameter_list);

In the above syntax

type: It defines the datatype of the variable for which the memory is allocated by the new operator.

variable: It is the name of the variable that points to the memory.

parameter_list: It is the list of values that are initialized to a variable.

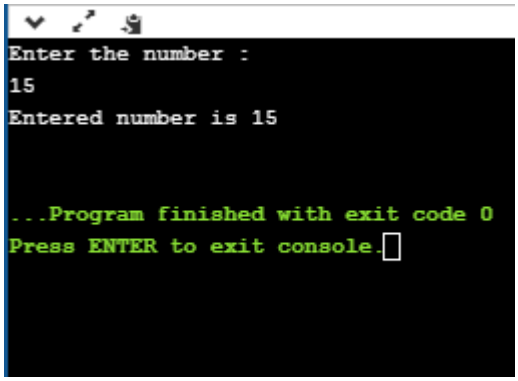
The new operator does not use the sizeof() operator to allocate the memory. It also does not use the resize as the new operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object.

As we know that the new operator allocates the memory in a heap; if the memory is not available in a heap and the new operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

Let's understand the new operator through an example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.  int *ptr;  // integer pointer variable declaration
6.  ptr=new int; // allocating memory to the pointer variable ptr.
7.  std::cout << "Enter the number : " << std::endl;
8.  std::cin >>*ptr;
9.  std::cout << "Entered number is " <<*ptr<< std::endl;
10. return 0;
11. }
```

Output:



What is malloc()?

A malloc() is a function that allocates memory at the runtime. This function returns the void pointer, which means that it can be assigned to any pointer type. This void pointer can be further typecast to get the pointer that points to the memory of a specified type.

The syntax of the malloc() function is given below:

```
1. type variable_name = (type *)malloc(sizeof(type));
```

where,

type: it is the datatype of the variable for which the memory has to be allocated.

variable_name: It defines the name of the variable that points to the memory.

(type*): It is used for typecasting so that we can get the pointer of a specified type that points to the memory.

sizeof(): The sizeof() operator is used in the malloc() function to obtain the memory size required for the allocation.

Note: The malloc() function returns the void pointer, so typecasting is required to assign a different type to the pointer. The sizeof() operator is required in the malloc() function as the malloc() function returns the raw memory, so the sizeof() operator will tell the malloc() function how much memory is required for the allocation.

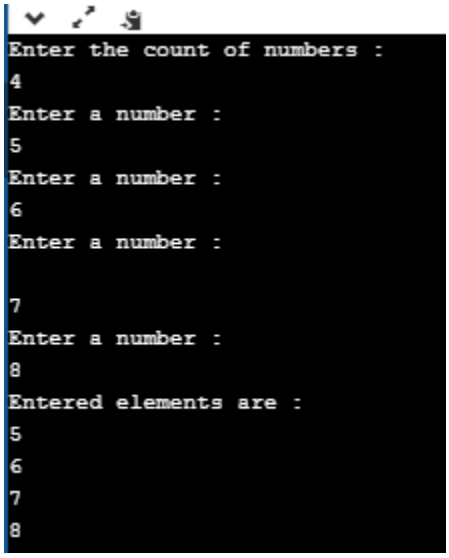
If the sufficient memory is not available, then the memory can be resized using realloc() function. As we know that all the dynamic memory requirements are fulfilled using heap memory, so malloc() function also allocates the memory in a heap and returns the pointer to it. The heap memory is very limited, so when our code starts execution, it marks the memory in use, and when our code completes its task, then it frees the memory by using the free() function. If the sufficient memory is not available, and our code tries to access the memory, then the malloc() function returns the NULL pointer. The memory which is allocated by the malloc() function can be deallocated by using the free() function.

Let's understand through an example.

```
1. #include <iostream>
2. #include<stdlib.h>
3. using namespace std;
4.
5. int main()
6. {
7.
8.    int len;   // variable declaration
9.    std::cout << "Enter the count of numbers :" << std::endl;
10.   std::cin >> len;
11.   int *ptr; // pointer variable declaration
12.   ptr=(int*) malloc(sizeof(int)*len);  // allocating memory to  the poiner variable
13.   for(int i=0;i<len;i++)
14.   {
15.      std::cout << "Enter a number : " << std::endl;
```

```
16.       std::cin >> *(ptr+i);
17.    }
18.    std::cout << "Entered elements are : " << std::endl;
19.     for(int i=0;i<len;i++)
20.    {
21.       std::cout << *(ptr+i) << std::endl;
22.    }
23. free(ptr);
24.      return 0;
25. }
```

Output:



If we do not use the **free()** function at the correct place, then it can lead to the cause of the dangling pointer. **Let's understand this scenario through an example.**

```
1.  #include <iostream>
2.  #include<stdlib.h>
3.  using namespace std;
4.  int *func()
5.  {
6.       int *p;
```

```
7.      p=(int*) malloc(sizeof(int));
8.      free(p);
9.      return p;
10. }
11. int main()
12. {
13.
14.  int *ptr;
15.  ptr=func();
16.  free(ptr);
17.      return 0;
18. }
```

In the above code, we are calling the func() function. The func() function returns the integer pointer. Inside the func() function, we have declared a *p pointer, and the memory is allocated to this pointer variable using malloc() function. In this case, we are returning the pointer whose memory is already released. The ptr is a dangling pointer as it is pointing to the released memory location. Or we can say ptr is referring to that memory which is not pointed by the pointer.

Till now, we get to know about the new operator and the malloc() function. Now, we will see the differences between the new operator and the malloc() function.

Differences between the malloc() and new

- The new operator constructs an object, i.e., it calls the constructor to initialize an object while **malloc()** function does not call the constructor. The new operator

invokes the constructor, and the delete operator invokes the destructor to destroy the object. This is the biggest difference between the malloc() and new.

- The new is an operator, while malloc() is a predefined function in the stdlib header file.
- The operator new can be overloaded while the malloc() function cannot be overloaded.
- If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.
- In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.
- In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the memory.

Syntax of new operator

```
type reference_variable = new type name;
```

where,

type: It defines the data type of the reference variable.

reference_variable: It is the name of the pointer variable.

new: It is an operator used for allocating the memory.

type name: It can be any basic data type.

For example,

```
int *p;
p = new int;
```

In the above statements, we are declaring an integer pointer variable. The statement **p = new int;** allocates the memory space for an integer variable.

Syntax of malloc() is given below:

```
int *ptr = (data_type*) malloc(sizeof(data_type));
```

ptr: It is a pointer variable.

data_type: It can be any basic data type.

For example,

```
int *p;
```

2. p = (**int** *) malloc(**sizeof**(**int**))

The above statement will allocate the memory for an integer variable in a heap, and then stores the address of the reserved memory in 'p' variable.

- On the other hand, the memory allocated using malloc() function can be deallocated using the free() function.
- Once the memory is allocated using the new operator, then it cannot be resized. On the other hand, the memory is allocated using malloc() function; then, it can be reallocated using realloc() function.
- The execution time of new is less than the malloc() function as new is a construct, and malloc is a function.
- The new operator does not return the separate pointer variable; it returns the address of the newly created object. On the other hand, the malloc() function returns the void pointer which can be further typecast in a specified type.

free vs delete in C++

In this topic, we are going to learn about the **free()** function and **delete** operator in C++.

free() function

The free() function is used in C++ to de-allocate the memory dynamically. It is basically a library function used in C++, and it is defined in **stdlib.h** header file. This library function is used when the pointers either pointing to the memory allocated using malloc() function or Null pointer.

Syntax of free() function

Suppose we have declared a pointer 'ptr', and now, we want to de-allocate its memory:

1. free(ptr);

The above syntax would de-allocate the memory of the pointer variable 'ptr'.

free() parameters

In the above syntax, ptr is a parameter inside the free() function. The ptr is a pointer pointing to the memory block allocated using malloc(), calloc() or realloc function. This pointer can also be null or a pointer allocated using malloc but not pointing to any other memory block.

- If the pointer is null, then the free() function will not do anything.
- If the pointer is allocated using malloc, calloc, or realloc, but not pointing to any memory block then this function will cause undefined behavior.

free() Return Value

The free() function does not return any value. Its main function is to free the memory.

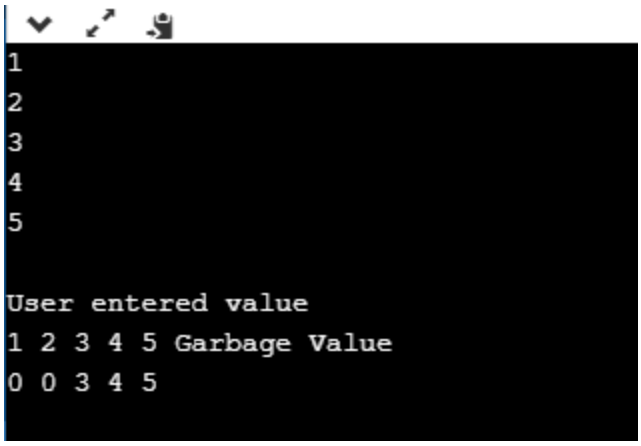
Let's understand through an example.

```
1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4.
5. int main()
6. {
7.     int *ptr;
8.     ptr = (int*) malloc(5*sizeof(int));
9.     cout << "Enter 5 integer" << endl;
10.
11.    for (int i=0; i<5; i++)
12.    {
13.    // *(ptr+i) can be replaced by ptr[i]
14.        cin >>ptr[i];
15.    }
16.    cout << endl << "User entered value"<< endl;
17.
18.    for (int i=0; i<5; i++)
19.    {
20.        cout <<*(ptr+i)  << " ";
21.    }
22.    free(ptr);
23.
24.    /* prints a garbage value after ptr is free */
25.    cout << "Garbage Value" << endl;
26.
```

```
27.    for (int i=0; i<5; i++)
28.    {
29.        cout << *(ptr+i)<< " ";
30.    }
31.    return 0;
32. }
```

The above code shows how free() function works with malloc(). First, we declare integer pointer *ptr, and then we allocate the memory to this pointer variable by using malloc() function. Now, ptr is pointing to the uninitialized memory block of 5 integers. After allocating the memory, we use the free() function to destroy this allocated memory. When we try to print the value, which is pointed by the ptr, we get a garbage value, which means that memory is de-allocated.

Output



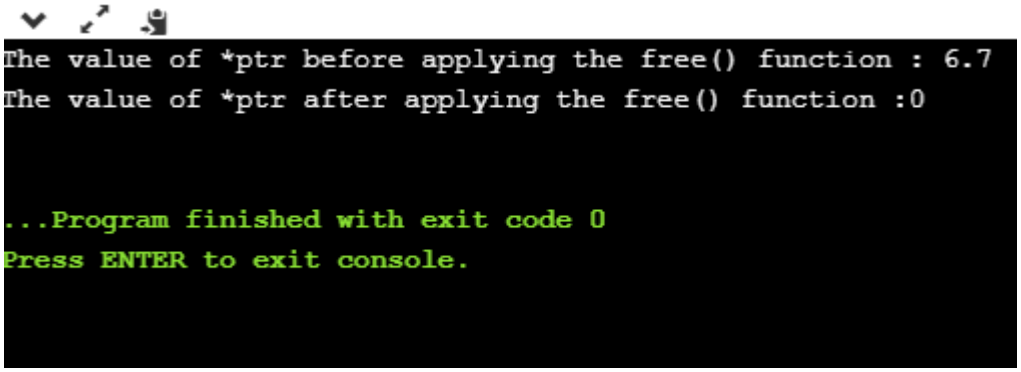
Let's see how free() function works with a calloc.

```
1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4. int main()
5. {
6.  float *ptr; // float pointer declaration
7.  ptr=(float*)calloc(1,sizeof(float));
8.  *ptr=6.7;
```

```
9.  std::cout << "The value of *ptr before applying the free() function : " <<*ptr<< st
    d::endl;
10. free(ptr);
11. std::cout << "The value of *ptr after applying the free() function :" <<*ptr<< std:
    :endl;
12.    return 0;
13. }
```

In the above example, we can observe that free() function works with a calloc(). We use the calloc() function to allocate the memory block to the float pointer ptr. We have assigned a memory block to the ptr that can have a single float type value.

Output:



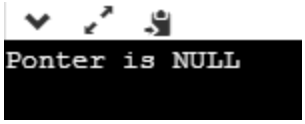
Let's look at another example.

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  using namespace std;
4.  int main()
5.  {
6.   int *ptr1=NULL;
7.   int *ptr2;
8.   int x=9;
9.  ptr2=&x;
10. if(ptr1)
11. {
```

```
12.     std::cout << "Pointer is not Null" << std::endl;
13.  }
14.  else
15.  {
16.     cout<<"Ponter is NULL";
17.  }
18.  free(ptr1);
19.  //free(ptr2); // If this statement is executed, then it gives a runtime error.
20.  return 0;
21. }
```

The above code shows how free() function works with a NULL pointer. We have declared two pointers, i.e., ptr1 and ptr2. We assign a NULL value to the pointer ptr1 and the address of x variable to pointer ptr2. When we apply the free(ptr1) function to the ptr1, then the memory block assigned to the ptr is successfully freed. The statement free(ptr2) shows a runtime error as the memory block assigned to the ptr2 is not allocated using malloc or calloc function.

Output



Delete operator

It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically. This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax

1. **delete** pointer_name

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:

1. **delete** p;

To delete the array, we use the statement as given below:

1. **delete** [] p;

Some important points related to delete operator are:

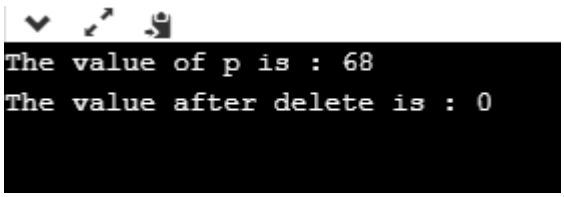
- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use delete[] and delete operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

Let's look at the simple example of a delete operator.

```
1. #include <iostream>
2. #include <cstdlib>
3. using namespace std;
4.
5. int main()
6. {
7.  int *ptr;
8.  ptr=new int;
9.  *ptr=68;
10. std::cout << "The value of p is : " <<*ptr<< std::endl;
11. delete ptr;
12. std::cout <<"The value after delete is :  " <<*ptr<< std::endl;
13. return 0;
14. }
```

In the above code, we use the new operator to allocate the memory, so we use the delete ptr operator to destroy the memory block, which is pointed by the pointer ptr.

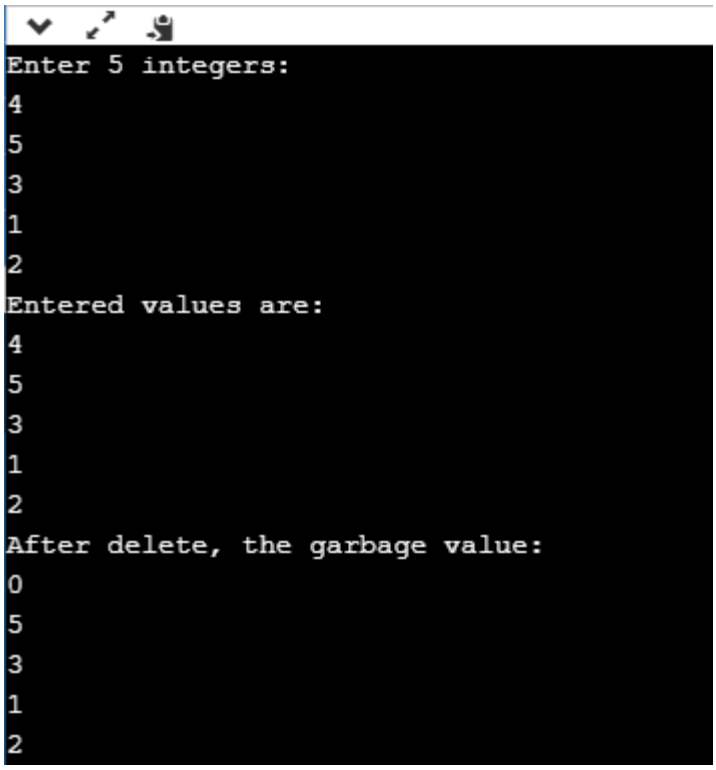
Output



Let's see how delete works with an array of objects.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int *ptr=new int[5];  // memory allocation using new operator.
6.     std::cout << "Enter 5 integers:" << std::endl;
7.     for(int i=1;i<=5;i++)
8.     {
9.         cin>>ptr[i];
10.    }
11.    std::cout << "Entered values are:" << std::endl;
12.        for(int i=1;i<=5;i++)
13.    {
14.        cout<<*(ptr+i)<<endl;
15.    }
16.    delete[] ptr; // deleting the memory block pointed by the ptr.
17.    std::cout << "After delete, the garbage value:" << std::endl;
18.        for(int i=1;i<=5;i++)
19.    {
20.        cout<<*(ptr+i)<<endl;
21.    }
22.  return 0;
23. }
```

Output



Differences between delete and free()

The following are the differences between delete and free() in C++ are:

- The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
- The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
- When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.
- The delete() operator is faster than the free() function.

3 The Stack

A **stack** is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the *top* of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called ***Last-in-First-out (LIFO)***. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack. The operation of the stack can be illustrated as in Fig. 3.1.

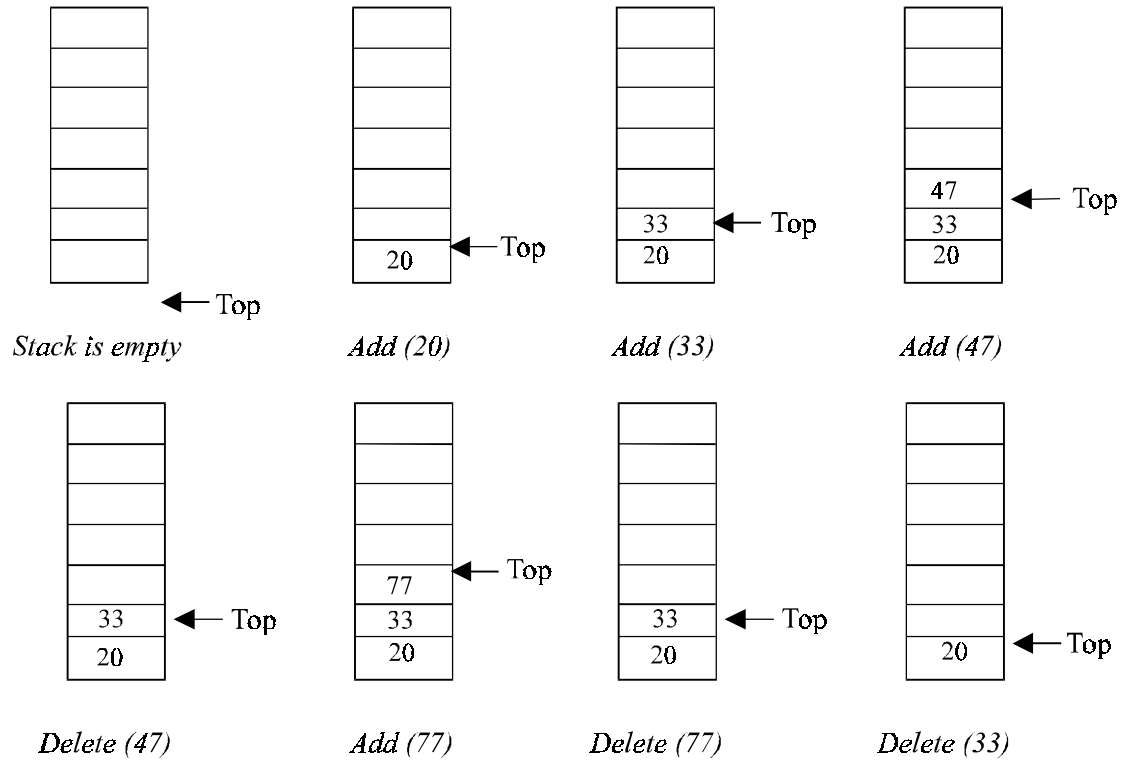


Fig. 3.1. Stack operation.

The insertion (or addition) operation is referred to as *push*, and the deletion (or remove) operation as *pop*. A stack is said to be *empty* or *underflow*, if the stack contains no

elements. At this point the top of the stack is present at the bottom of the stack. And it is *overflow* when the stack becomes full, *i.e.*, no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

3.1. OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

3.2. STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (*i.e.*, increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (*i.e.*, memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity.

The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

3.3. STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:
   (*a*) Display "The stack is in overflow condition"
   (*b*) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then
   (*a*) Display "The Stack is empty"
   (*b*) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit

PROGRAM 3.1

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS PERFORMED
//ON THE STACK  AND IT IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>

//Defining the maximum size of the stack
#define MAXSIZE 100

//Declaring the stack array and top variables in a structure
struct stack
{
      int stack[MAXSIZE];
      int Top;
};

//type definition allows the user to define an identifier that would
//represent an existing data type. The user-defined data type identifier
//can later be used to declare variables.
typedef struct stack NODE;
```

```
//This function will add/insert an element to Top of the stack
void push(NODE *pu)
{
int item;
//if the top pointer already reached the maximum allowed size then
//we can say that the stack is full or overflow
if (pu->Top == MAXSIZE–1)
{
        printf("\nThe Stack Is Full");
        getch();
}
//Otherwise an element can be added or inserted by
//incrementing the stack pointer Top as follows
else
{
                printf("\nEnter The Element To Be Inserted = ");
                scanf("%d",&item);
                pu->stack[++pu->Top]=item;
        }
}

//This function will delete an element from the Top of the stack
void pop(NODE *po)
{
                int item;
                //If the Top pointer points to NULL, then the stack is empty
                //That is NO element is there to delete or pop
                if (po->Top == -1)
                        printf("\nThe Stack Is Empty");
                //Otherwise the top most element in the stack is popped or
                //deleted by decrementing the Top pointer
                else
                {
                        item=po->stack[po->Top--];
                        printf("\nThe Deleted Element Is = %d",item);
                }
}
//This function to print all the existing elements in the stack
void traverse(NODE *pt)
{
                int i;
                //If the Top pointer points to NULL, then the stack is empty
```

```
        //That is NO element is there to delete or pop
        if (pt->Top == -1)
                printf("\nThe Stack is Empty");
        //Otherwise all the elements in the stack is printed
        else
        {
                printf("\n\nThe Element(s) In The Stack(s) is/are...");
                for(i=pt->Top; i>=0; i--)
                        printf ("\n %d",pt->stack[i]);
        }
}

void main( )
{
        int choice;
        char ch;
        //Declaring an pointer variable to the structure
        NODE *ps;
        //Initializing the Top pointer to NULL
        ps->Top=–1;
        do
        {
                clrscr();
                //A menu for the stack operations
                printf("\n1. PUSH");
                printf("\n2. POP");
                printf("\n3. TRAVERSE");
                printf("\nEnter Your Choice = ");
                scanf ("%d", &choice);
                switch(choice)
                {
                        case 1://Calling push() function by passing
                        //the structure pointer to the function
                        push(ps);
                        break;

                        case 2://calling pop() function
                        pop(ps);
                        break;
                        case 3://calling traverse() function
                        traverse(ps);
                        break;
```

```
                default:
                printf("\nYou Entered Wrong Choice") ;
            }
            printf("\n\nPress (Y/y) To Continue = ");
            //Removing all characters in the input buffer
            //for fresh input(s), especially <<Enter>> key
            fflush(stdin);
            scanf("%c",&ch);
        }while(ch == 'Y' || ch == 'y');
}
```

PROGRAM 3.2

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS
//PERFORMED ON STACK & IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C++

#include<iostream.h>
#include<conio.h>

//Defining the maximum size of the stack
#define MAXSIZE 100

//A class initialized with public and private variables and functions
class STACK_ARRAY
{
        int stack[MAXSIZE];
        int Top;

        public:
        //constructor is called and Top pointer is initialized to –1
        //when an object is created for the class
            STACK_ARRAY()
            {
                Top=–1;
            }
            void push();
            void pop();
            void traverse();
};
```

```
//This function will add/insert an element to Top of the stack
void STACK_ARRAY::push()
{
        int item;
        //if the top pointer already reached the maximum allowed size then
        //we can say that the stack is full or overflow
        if (Top == MAXSIZE–1)
        {
            cout<<"\nThe Stack Is Full";
            getch();
        }
        //Otherwise an element can be added or inserted by
        //incrementing the stack pointer Top as follows
        else
        {
            cout<<"\nEnter The Element To Be Inserted = ";
            cin>>item;
            stack[++Top]=item;
        }
}

//This function will delete an element from the Top of the stack
void STACK_ARRAY::pop()
{
        int item;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == –1)
            cout<<"\nThe Stack Is Empty";
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
        {
            item=stack[Top--];
            cout<<"\nThe Deleted Element Is = "<<item;
        }
}

//This function to print all the existing elements in the stack
void STACK_ARRAY::traverse()
{
```

```
        int i;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == -1)
                cout<<"\nThe Stack is Empty";
        //Otherwise all the elements in the stack is printed
        else
        {
                cout<<"\n\nThe Element(s) In The Stack(s) is/are...";
                for(i=Top; i>=0; i--)
                        cout<<"\n "<<stack[i];
        }
}


void main()
{
        int choice;
        char ch;
        //Declaring an object to the class
        STACK_ARRAY ps;
        do
        {
                clrscr();
                //A menu for the stack operations
                cout<<"\n1. PUSH";
                cout<<"\n2. POP";
                cout<<"\n3. TRAVERSE";
                cout<<"\nEnter Your Choice = ";
                cin>>choice;

                switch(choice)
                {
                        case 1://Calling push() function by class object
                        ps.push();
                        break;

                        case 2://calling pop() function
                        ps.pop();
                        break;
```

```
                    case 3://calling traverse() function
                    ps.traverse();
                    break;

                    default:
                    cout<<"\nYou Entered Wrong Choice" ;
                }
                cout<<"\n\nPress (Y/y) To Continue = ";
                cin>>ch;
            }while(ch == 'Y' || ch == 'y');
}
```

3.4. APPLICATIONS OF STACKS

There are a number of applications of stacks; three of them are discussed briefly in the preceding sections. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

3.4.1. RECURSION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.
2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.

A flowchart model for any recursive algorithm is given in Fig. 3.2.

It is difficult to understand a recursive function from its flowchart, and the best way is to have an intuitive understanding of the function. The key box in the flowchart contained in the body of the function is the one, which invokes a call to itself. Each time a function call to itself is executed, the prologue of the function saves necessary information required for its proper functioning.

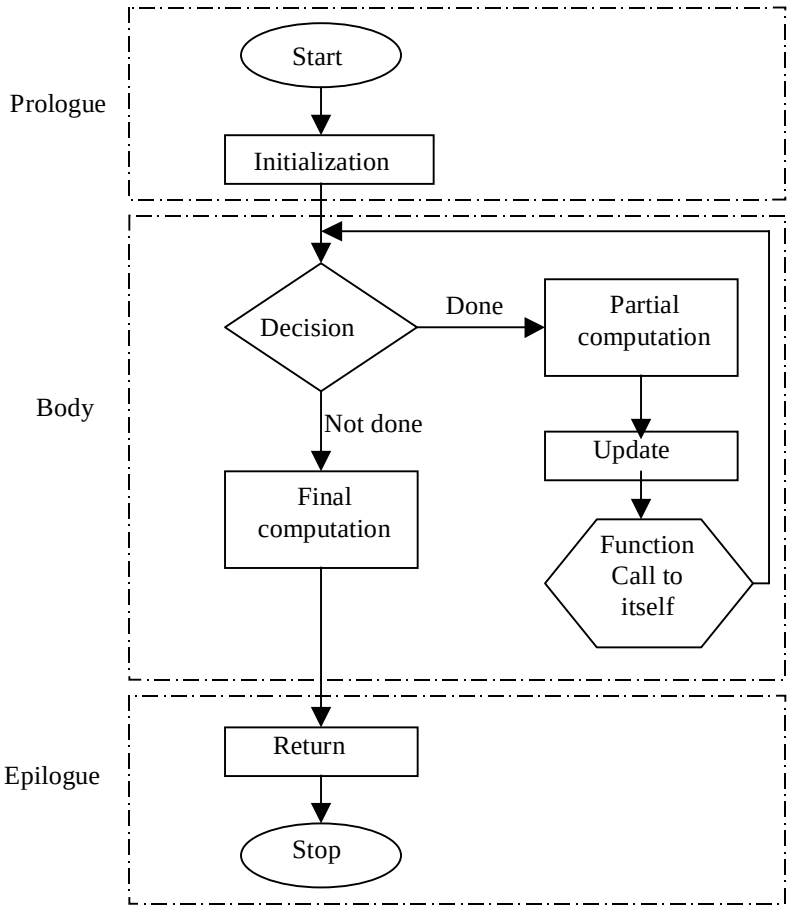


Fig. 3.2. Flowchart model for a recursive algorithm

The Last-in-First-Out characteristics of a recursive function points that the stack is the most obvious data structure to implement the recursive function. Programs compiled in modern high-level languages (even C) make use of a stack for the procedure or function invocation in memory. When any procedure or function is called, a number of words (such as variables, return address and other arguments and its data(s) for future use) are pushed onto the program stack. When the procedure or function returns, this frame of data is popped off the stack.

As a function calls a (may be or may not be another) function, its arguments, return address and local variables are pushed onto the stack. Since each function runs in its own environment or context, it becomes possible for a function to call itself — a technique known as *recursion*. This capability is extremely useful and extensively used — because many problems are elegantly specified or solved in a recursive way.

The stack is a region of main memory within which programs temporarily store data as they execute. For example, when a program sends parameters to a function, the parameters are placed on the stack. When the function completes its execution these parameters are popped off from the stack. When a function calls other function the current contents (ie., variables) of the caller function are pushed onto the stack with the address of the instruction just next to the call instruction, this is done so after the execution of called function, the compiler can backtrack (or remember) the path from where it is sent to the called function.

The recursive mechanism can be best described by an example. Consider the following program to calculate factorial of a number recursively, which explicitly describes the recursive framework.

PROGRAM 3.3

```
//PROGRAM TO FIND FACTORIAL OF A NUMBER, RECURSIVELY

#include<conio.h>
#include<iostream.h>

void fact(int no, int facto)
{

        if (no <= 1)
        {
            //Final computation and returning and restoring address
            cout<<"\nThe Factorial is = "<<facto;
            return;
        }
        else
        {
            //Partiial computation of the program
            facto=facto*no;
            //Function call to itself, that is recursion
            fact(--no,facto);
        }

}
void main()
{
        clrscr();
        int number,factorial;
```

```
        //Initialization of formal parameters, local variables and etc.
        factorial=1;
        cout<<"\nEnter the No = ";
        cin>>number;
        //Starting point of the function, which calls itself
        fact(number,factorial);
        getch();
}
```

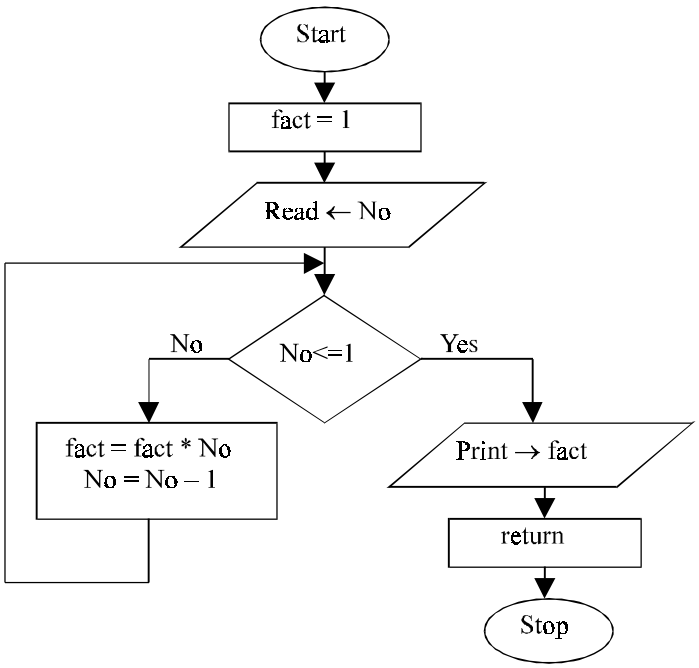


Fig. 3.3. Flowchart for finding factorial recursively

3.4.2. RECURSION vs ITERATION

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

1. It requires stack implementation.
2. It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
3. Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

Though inefficient way to solve general problems, it is too handy in several problems as discussed in the starting of this chapter. It provides a programmer with certain pitfalls,

and quite sharp concepts about programming. Moreover recursive functions are often easier to implement d maintain, particularly in case of data structures which are by nature recursive. Such data structures are queues, trees, and linked lists. Given below are some of the important points, which differentiate iteration from recursion.

No.	*Iteration*	*Recursion*
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.
2	Iteration involves four clear-cut Steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory 1,1tilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

3.4.3. DISADVANTAGES OF RECURSION

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

3.4.4. TOWER OF HANOI

So far we have discussed the comparative definition and disadvantages of recursion with examples. Now let us look at the Tower of Hanoi problem and see how we can use recursive technique to produce a logical and elegant solution.

The initial setup of the problem is shown below. Here three pegs (or towers) X, Y and Z exists. There will be four different sized disks, say A, B, C and D. Each disk has a hole in the center so that it can be stacked on any of the pegs. At the beginning, the disks are stacked on the X peg, that is the largest sized disk on the bottom and the smallest sized disk on top as shown in Fig. 3.4.

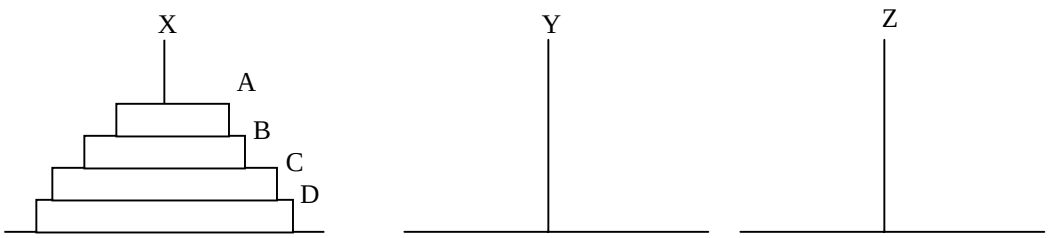


Fig. 3.4. Initial position of the Tower of Hanoi

Here we have to transfer all the disks from source peg X to the destination peg Z by using an intermediate peg Y. Following are the rules to be followed during transfer :

1. Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
2. Only one disk may be moved at a time.
3. Each disk must be stacked on any one of the pegs.

Now Tower of Hanoi problem can be solved as shown below :

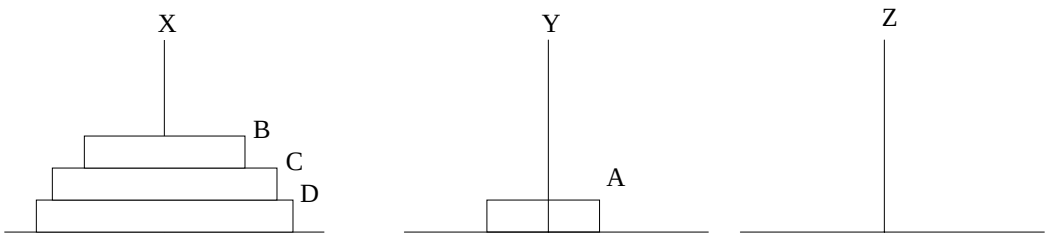


Fig. 3.5. Move disk A from the peg X to peg Y

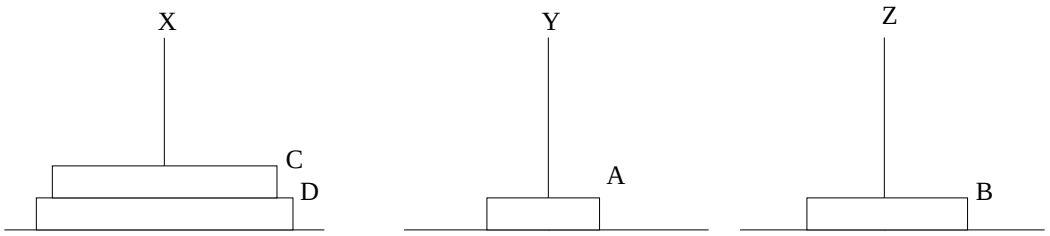


Fig. 3.6. Move disk B from the peg X to peg Z

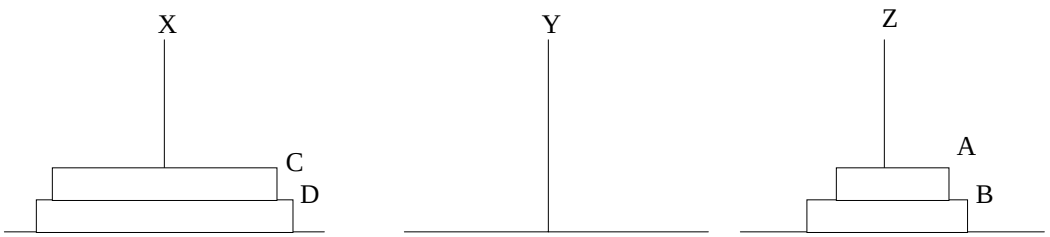


Fig. 3.7. Move disk A from the peg Y to peg Z

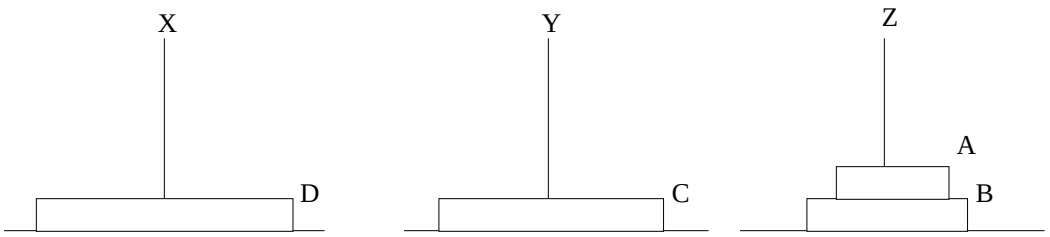


Fig. 3.8. Move disk C from the peg X to peg Y

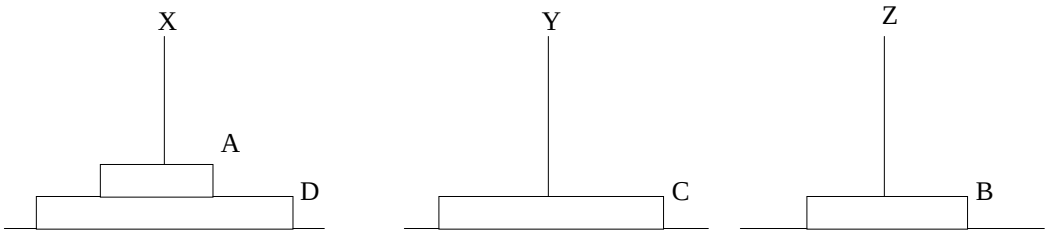


Fig. 3.9. Move disk A from the peg Z to peg X

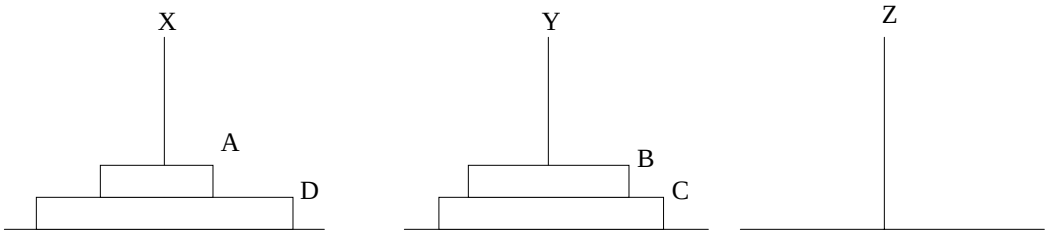


Fig. 3.10. Move disk B from the peg Z to peg Y

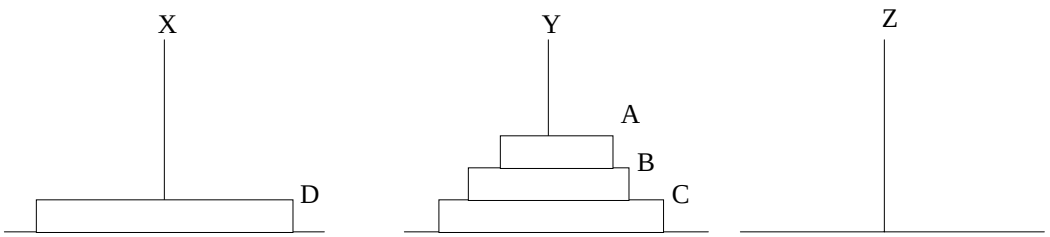


Fig. 3.11. Move disk A from the peg X to peg Y

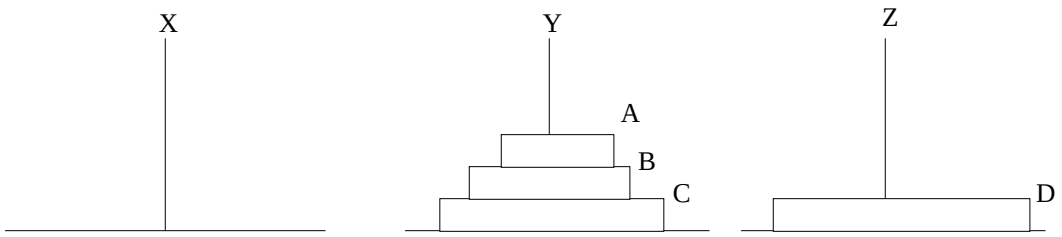


Fig. 3.12. Move disk D from the peg X to peg Z

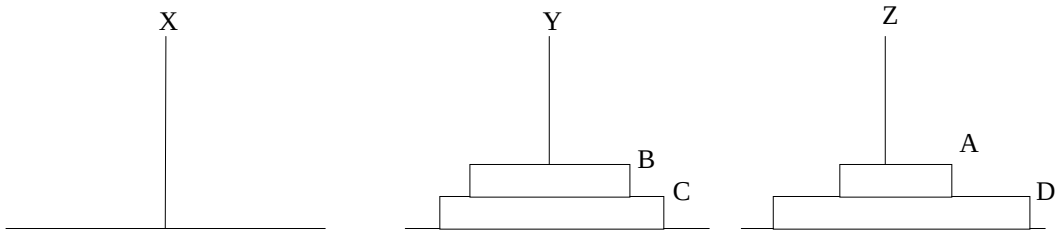


Fig. 3.13. Move disk A from the peg Y to peg Z

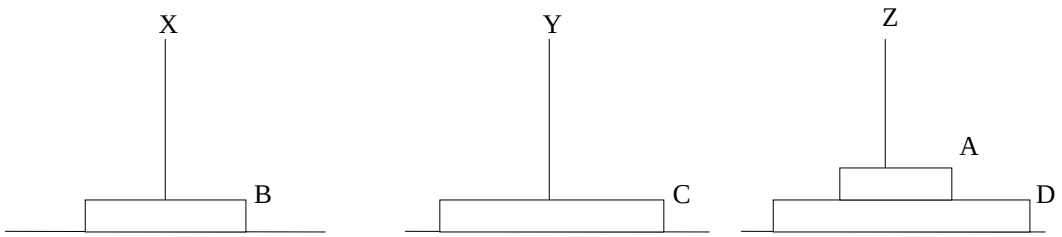


Fig. 3.14. Move disk B from the peg Y to peg X

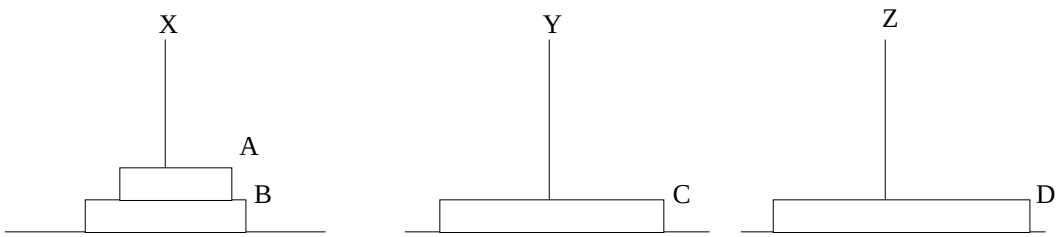


Fig. 3.15. Move disk A from the peg Z to peg X

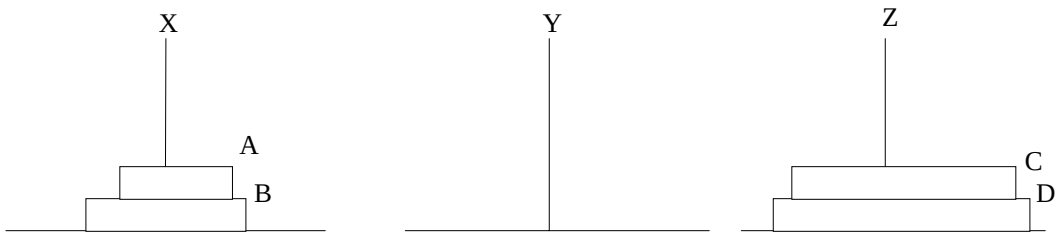


Fig. 3.16. Move disk C from the peg Y to peg Z

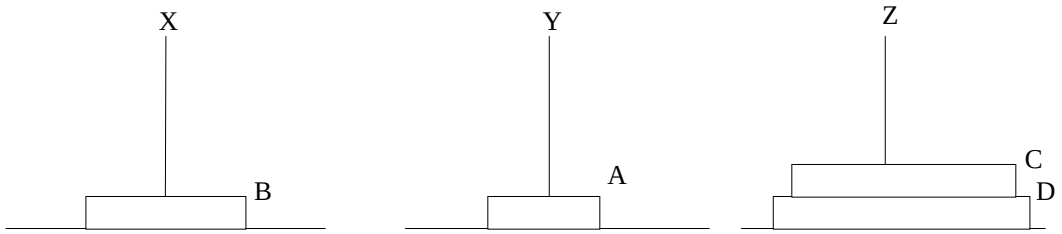


Fig. 3.17. Move disk A from the peg X to peg Y

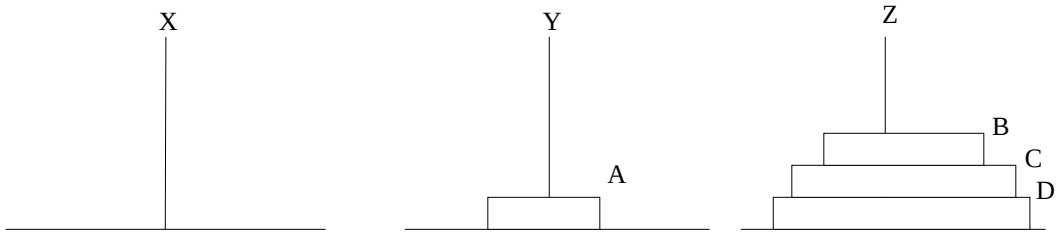


Fig. 3.18. Move disk B from the peg X to peg Z

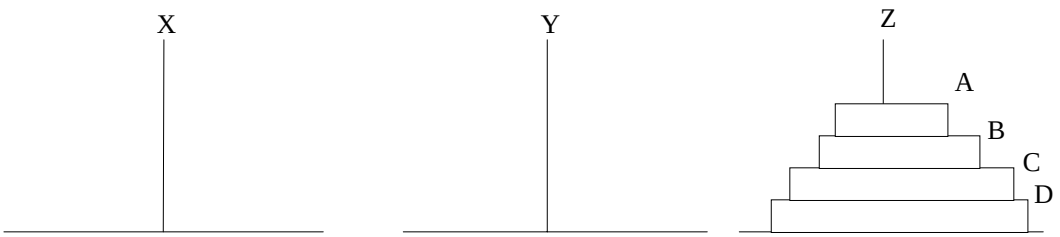


Fig. 3.19. Move disk A from the tower Y to tower Z

We can generalize the solution to the Tower of Hanoi problem recursively as follows :
To move *n* disks from peg X to peg Z, using Y as auxiliary peg:
1. If *n* = 1, move the single disk from X to Z and stop.
2. Move the top(*n* – 1) disks from the peg X to the peg Y, using Z as auxiliary.
3. Move *n*th disk to peg Z.
4. Now move *n* – 1 disk from Y to Z, using Z as auxiliary.

PROGRAM 3.4

```
//PROGRAM TO SIMULATE THE TOWER OF HANOI PROBLEM
//CODED AND COMPILED IN TURBO C++

#include<conio.h>
#include<iostream.h>

class tower
{
        //Private variables are declared
        int NoDisk;
        char FromTower,ToTower,AuxTower;

        public:
            void hanoi(int,char,char,char);
};

void tower::hanoi(int NoDisk,char FromTower,char ToTower, char AuxTower)
{
        //if only one disk, make the move and return
        if (NoDisk == 1)
        {
```

```
            cout<<"\nMove from disk 1 from tower "<<FromTower<<" to tower "<<ToTower;
            return;
          }
          //Move top n–1 disks from X to Y, using Z as auxiliary tower
          hanoi(NoDisk–1,FromTower,AuxTower,ToTower);
          //Move remaining disk from X to Z
          cout<<"\nMove from disk "<<NoDisk<<" from tower "<<FromTower<<" to tower
"<<ToTower;
          //Move n–1 disk from Y to Z using X as auxiliary tower
          hanoi(NoDisk–1,AuxTower,ToTower,FromTower);
          return;
}

void main()
{
          int No;
          tower Ob;
          clrscr();
          cout<<"\n\t\t\t--- Tower of Hanoi ---\n";
          //Imput the number of disk in the tower
          cout<<"\n\nEnter the number of disks = ";
          cin>>No;
          //We assume that the towers are X, Y and Z
          Ob.hanoi(No,'X','Z','Y');
          cout<<"\n\nPress any key to continue...";
          getch();
}
```

3.4.5. EXPRESSION

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation
2. Prefix notation
3. Postfix notation

The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example : The expression to add two numbers A and B is written in infix notation as:

A + B

Note that the operator '+' is written in between the operands A and B.

The *prefix notation* is a notation in which the operator(s) is written before the operands, it is also called polish notation in the honor of the polish mathematician Jan

Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:

+ A B

As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:

A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:

add(A, B)

Note that the operator *add* (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. In the preceding sections we will study the conversion of the expression from one notation to other.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in *infix* notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS, and associativity.

Using infix notation, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error

in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

3.5. CONVERTING INFIX TO POSTFIX EXPRESSION

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Problem 3.4.2.1. Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE + F *] / G}
A + { [(BC + (DE + F * +] / G} .
A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Problem 3.4.2.2. Give postfix form for (A + B) * C / D + E ^ A / B

Solution. Evaluation order is

[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]
[(AB +) C * D /] + [(EA ^) B /]

(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Push "(" onto stack, and add")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.
   (*b*) Add ⊗ to stack.
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.

Note. Special character ⊗ is used to symbolize any operator in P.

Consider the following arithmetic infix expression P

P = A + (B / C - (D * E ^ F) + G) * H

Fig. 3.20 shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character scanned	*Stack*	*Postfix Expression (Q)*
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
/	(+ (/	A B
C	(+ (/	A B C

–	(+ (-	A B C /
(	(+ (- (	A B C /
D	(+ (- (	A B C / D
*	(+ (- (*	A B C / D
E	(+ (- (*	A B C / D E
^	(+ (- (* ^	A B C / D E
F	(+ (- (* ^	A B C / D E F
)	(+ (-	A B C / D E F ^ *
+	(+ (+	A B C / D E F ^ * -
G	(+ (+	A B C / D E F ^ * - G
)	(+	A B C / D E F ^ * - G +
*	(+ *	A B C / D E F ^ * - G +
H	(+ *	A B C / D E F ^ * - G + H
)		A B C / D E F ^ * - G + H * +

Fig. 3.20

PROGRAM 3.5

```
//THIS PROGRAM IS TO COVERT THE INFIX TO POSTFIX EXPRESSION
//STACK IS USED AND IT IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<string.h>

//Defining the maximum size of the stack
#define MAXSIZE 100
//Declaring the stack array and top variables in a structure
struct stack
{
        char stack[MAXSIZE];
        int Top;
};

//type definition allows the user to define an identifier that would
//represent an existing data type. The user-defined data type identifier
//can later be used to declare variables.
typedef struct stack NODE;
```

```
//This function will add/insert an element to Top of the stack
void push(NODE *pu,char item)
{
        //if the top pointer already reached the maximum allowed size then
        //we can say that the stack is full or overflow
        if (pu->Top == MAXSIZE-1)
        {
                printf("\nThe Stack Is Full");
                getch();
        }
        //Otherwise an element can be added or inserted by
        //incrementing the stack pointer Top as follows
        else
                pu->stack[++pu->Top]=item;
}

//This function will delete an element from the Top of the stack
char pop(NODE *po)
{
        char item='#';
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if(po->Top == –1)
                printf("\nThe Stack Is Empty. Invalid Infix expression");
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
                item=po->stack[po->Top--];
        return(item);
}


//This function returns the precedence of the operator
int prec(char symbol)
{
        switch(symbol)
        {
                case '(':
                        return(1);
                case ')':
                        return(2);
                case '+':
```

```
                case '-':
                        return(3);
                case '*':
                case '/':
                case '%':
                        return(4);
                case '^':
                        return(5);
                default:
                        return(0);
        }
}

//This function will return the postfix expression of an infix
void Infix_Postfix(char infix[])
{

        int len,priority;
        char postfix[MAXSIZE],ch;
        //Declaring an pointer variable to the structure
        NODE *ps;
        //Initializing the Top pointer to NULL
        ps->Top=–1;
        //Finding length of the string
        len=strlen(infix);
        //At the end of the string inputting a parenthesis ')'
        infix[len++]=')';
        push(ps,'(');//Parenthesis is pushed to the stack
        for(int i=0,j=0;i<len;i++)
        {
                switch(prec(infix[i]))
                {
                        //Scanned char is '(' push to the stack
                        case 1:
                                push(ps,infix[i]);
                                break;
                        //Scanned char is ')' pop the operator(s) and add to //the postfix
                        expression
                        case 2:
                                ch=pop(ps);
                                while(ch != '(')
                                {
```

```
                postfix[j++]=ch;
                ch=pop(ps);
        }
        break;
//Scanned operator is +,– then pop the higher or same
//precedence operator to add postfix before pushing
//the scanned operator to the stack
case 3:
        ch=pop(ps);
        while(prec(ch) >= 3)
        {
                postfix[j++]=ch;
                ch=pop(ps);
        }
        push(ps,ch);
        push(ps,infix[i]);
        break;
//Scanned operator is *,/,% then pop the higher or
//same precedence operator to add postfix before
//pushing the scanned operator to the stack
case 4:
        ch=pop(ps);
        while(prec(ch) >= 4)
        {
                postfix[j++]=ch;
                ch=pop(ps);
        }
        push(ps,ch);
        push(ps,infix[i]);
        break;
//Scanned operator is ^ then pop the same
//precedence operator to add to postfix before pushing
//the scanned operator to the stack
case 5:
        ch=pop(ps);
        while(prec(ch) == 5)
        {
                postfix[j++]=ch;
                ch=pop(ps);
        }
        push(ps,ch);
        push(ps,infix[i]);
        break;
```

```
                        //Scanned char is a operand simply add to the postfix
                        //expression
                        default:
                                postfix[j++]=infix[i];
                                break;
                }
        }
        //Printing the postfix notation to the screen
        printf ("\nThe Postfix expression is = ");
        for(i=0;i<j;i++)
        printf ("%c",postfix[i]);
}

void main()
{
        char choice,infix[MAXSIZE];
        do
        {
                clrscr();
                printf("\n\nEnter the infix expression = ");
                fflush(stdin);
                gets(infix);//Inputting the infix notation
                Infix_Postfix(infix);//Calling the infix to postfix function
                printf("\n\nDo you want to continue (Y/y) =");
                fflush(stdin);
                scanf("%c",&choice);
        }while(choice == 'Y' || choice == 'y');
}
```

PROGRAM 3.6

```
//THIS PROGRAM IS TO COVERT THE INFIX TO POSTFIX EXPRESSION
//AND IT IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C++

#include<iostream.h>
#include<conio.h>
#include<string.h>
```

```
//Defining the maximum size of the stack
#define MAXSIZE 100

//A class initialised with public and private variables and functions
class STACK_ARRAY
{
        int stack[MAXSIZE];
        int Top;

        public:
        //constructor is called and Top pointer is initialised to –1
        //when an object is created for the class
            STACK_ARRAY()
            {
                Top=–1;
            }
            void push(char);
            char pop();
            int prec(char);
            void Infix_Postfix();

};

//This function will add/insert an element to Top of the stack
void STACK_ARRAY::push(char item)
{
        //if the top pointer already reached the maximum allows size then
        //we can say that the stack is full or overflow
        if (Top == MAXSIZE-1)
        {
            cout<<"\nThe Stack Is Full";
            getch();
        }
        //Otherwise an element can be added or inserted by
        //incrementing the stack pointer Top as follows
        else
            stack[++Top]=item;
}

//This function will delete an element from the Top of the stack
char STACK_ARRAY::pop()
{
```

```
        char item='#';
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == –1)
                cout<<"\nThe Stack Is Empty. Invalid Infix expression";
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
                item=stack[Top--];
        return(item);
}
//This function returns the precedence of the operator
int STACK_ARRAY::prec(char symbol)
{
        switch(symbol)
        {
                case '(':
                        return(1);
                case ')':
                        return(2);
                case '+':
                case '-':
                        return(3);
                case '*':
                case '/':
                case '%':
                        return(4);
                case '^':
                        return(5);
                default:
                        return(0);
        }
}

//This function will return the postfix expression of an infix
void STACK_ARRAY::Infix_Postfix()
{

        int len,priority;
        char infix[MAXSIZE],postfix[MAXSIZE],ch;
        cout<<"\n\nEnter the infix expression = ";
        cin>>infix;//Inputting the infix notation
```

```
//Finding length of the string
len=strlen(infix);
//At the end of the string inputting a parenthesis ')'
infix[len++]=')';
push('(');//Parenthesis is pushed to the stack
for(int i=0,j=0;i<len;i++)
{
        switch(prec(infix[i]))
        {
                //Scanned char is '(' push to the stack
                case 1:
                        push(infix[i]);
                        break;
        //Scanned char is ')' pop the operator(s) and add to
        //the postfix expression
                case 2:
                        ch=pop();
                        while(ch != '(')
                        {
                                postfix[j++]=ch;
                                ch=pop();
                        }
                        break;
        //Scanned operator is +,- then pop the higher or
        //same precedence operator to add postfix before
        //pushing the scanned operator to the stack
                case 3:
                        ch=pop();
                        while(prec(ch) >= 3)
                        {
                                postfix[j++]=ch;
                                ch=pop();
                        }
                        push(ch);
                        push(infix[i]);
                        break;
        //Scanned operator is *,/,% then pop the higher or
        //same precedence operator to add postfix before
        //pushing the scanned operator to the stack
                case 4:
                        ch=pop();
                        while(prec(ch) >= 4)
                        {
```

```
                                postfix[j++]=ch;
                                ch=pop();
                        }
                        push(ch);
                        push(infix[i]);
                        break;
                //Scanned operator is ^ then pop the same
                //precedence operator to add to postfix before
                //pushing the scanned operator to the stack
                case 5:
                        ch=pop();
                        while(prec(ch) == 5)
                        {
                                postfix[j++]=ch;
                                ch=pop();
                        }
                        push(ch);
                        push(infix[i]);
                        break;
                //Scanned char is a operand simply add to the
                //postfix expression
                default:
                        postfix[j++]=infix[i];
                        break;
        }
    }
    //Printing the postfix notation to the screen
    cout<<"\nThe Postfix expression is = ";
    for(i=0;i<j;i++)
    cout<<postfix[i];
}
void main()
{
    char choice;
    INFI_POST ip;
    do
    {
        clrscr();
        ip.Infix_Postfix();//Calling the infix to postfix function
        cout<<"\n\nDo you want to continue (Y/y) =";
        cin>>choice;
    }while(choice == 'Y' || choice == 'y');
}
```

3.6. EVALUATING POSTFIX EXPRESSION

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then:
   (*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (*b*) Evaluate B ⊗ A.
   (*c*) Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

PROGRAM 3.7

```
//THIS PROGRAM IS TO EVALUATE POSTFIX EXPRESSION. THE STACK
//IS USED AND IT IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<math.h>
#include<conio.h>
#include<string.h>

//Defining the maximum size of the stack
#define MAXSIZE 100

//Declaring the stack array and top variables in a structure
struct stack
{
        int stack[MAXSIZE];
        int Top;
};

//type definition allows the user to define an identifier that would
//represent an existing data type. The user-defined data type identifier
```

```
//can later be used to declare variables.
typedef struct stack NODE;

//This function will add/insert an element to Top of the stack
void push(NODE *pu,int item)
{
        //if the top pointer already reached the maximum allowed size then
        //we can say that the stack is full or overflow
        if (pu->Top == MAXSIZE-1)
        {
                printf("\nThe Stack Is Full");
                getch();
        }
        //Otherwise an element can be added or inserted by
        //incrementing the stack pointer Top as follows
        else
                pu->stack[++pu->Top]=item;
}

//This function will delete an element from the Top of the stack
int pop(NODE *po)
{
        int item;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (po->Top == –1)
                printf("\nThe Stack Is Empty. Invalid Infix expression");
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
                item=po->stack[po->Top--];
        return(item);
}

//This function will return the postfix expression of an infix
int Postfix_Eval(char postfix[])
{
        int a,b,temp,len;
        //Declaring an pointer variable to the structure
        NODE *ps;
        //Initializing the Top pointer to NULL
        ps->Top=–1;
```

```
        //Finding length of the string
        len=strlen(postfix);

        for(int i=0;i<len;i++)
        {
            if(postfix[i]<='9' && postfix[i]>='0')
                //Operand is pushed on the stack
    push(ps,(postfix[i]-48));
            else
            {
                //Pop the top most two operand for operation
                a=pop(ps);
                b=pop(ps);

                switch(postfix[i])
                {
                case '+':
                    temp=b+a; break;
                case '-':
                    temp=b–a;break;
                case '*':
                    temp=b*a;break;
                case '/':
                    temp=b/a;break;
                case '%':
                    temp=b%a;break;
                case '^':
                    temp=pow(b,a);
                }/*End of switch */
                push(ps,temp);
            }
        }
        return(pop(ps));
}

void main()
{
        char choice,postfix[MAXSIZE];
        do
        {
            clrscr();
            printf("\n\nEnter the Postfix expression = ");
```

```
                fflush(stdin);
                gets(postfix);//Inputting the postfix notation
        printf("\n\nThe postfix evaluation is = %d",Postfix_Eval(postfix));
                printf("\n\nDo you want to continue (Y/y) =");
                fflush(stdin);
                scanf("%c",&choice);
            }while(choice == 'Y' || choice == 'y');
}
```

PROGRAM 3.8

```
//THIS PROGRAM IS TO COVERT THE INFIX TO POSTFIX EXPRESSION
//AND IT IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C++

#include<iostream.h>
#include<math.h>
#include<conio.h>
#include<string.h>

//Defining the maximum size of the stack
#define MAXSIZE 100

//A class initialised with public and private variables and functions
class POST_EVAL
{
            int stack[MAXSIZE];
            int Top;

            public:
            //constructor is called and Top pointer is initialised to –1
            //when an object is created for the class
                POST_EVAL()
                {
                    Top=–1;
                }
                void push(int);
                int pop();
                int Postfix_Eval();
```

```
};

//This function will add/insert an element to Top of the stack
void POST_EVAL::push(int item)
{
        //if the top pointer already reached the maximum allows size
        //then we can say that the stack is full or overflow
        if (Top == MAXSIZE-1)
        {
                cout<<"\nThe Stack Is Full";
                getch();
        }
        //Otherwise an element can be added or inserted by
        //incrementing the stack pointer Top as follows
        else
                stack[++Top]=item;
}

//This function will delete an element from the Top of the stack
int POST_EVAL::pop()
{
        int item;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == -1)
                cout<<"\nThe Stack Is Empty. Invalid Infix expression";
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
                item=stack[Top--];
        return(item);
}

//This function will return the postfix expression of an infix
int POST_EVAL::Postfix_Eval()
{
        int a,b,temp,len;
        char postfix[MAXSIZE];
        cout<<"\n\nEnter the Postfix expression = ";
        cin>>postfix;//Inputting the postfix notation

        //Finding length of the string
```

```
        len=strlen(postfix);

        for(int i=0;i<len;i++)
        {
            if (postfix[i]<='9' && postfix[i]>='0')
                push(postfix[i]-48);
            else
            {
                a=pop();
                b=pop();

                switch(postfix[i])
                {
                case '+':
                    temp=b+a; break;
                case '-':
                    temp=b–a;break;
                case '*':
                    temp=b*a;break;
                case '/':
                    temp=b/a;break;
                case '%':
                    temp=b%a;break;
                case '^':
                    temp=pow(b,a);
                }/*End of switch */
                push(temp);
            }/*End of else*/
        }/*End of for */
        return(pop());
}

void main()
{
        char choice;
        int RESULT;
        POST_EVAL ps;
        do
        {
            clrscr();
            RESULT=ps.Postfix_Eval();
            cout<<"\n\nThe postfix evaluation is = "<<RESULT;
```

```
            cout<<"\n\nDo you want to continue (Y/y) =";
            cin>>choice;
        }while(choice == 'Y' || choice == 'y');
}
```

4 The Queues

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre.

It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

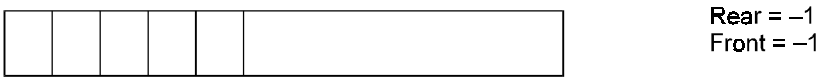


Fig. 4.1. Queue is empty.

Fig. 4.2. push(10)

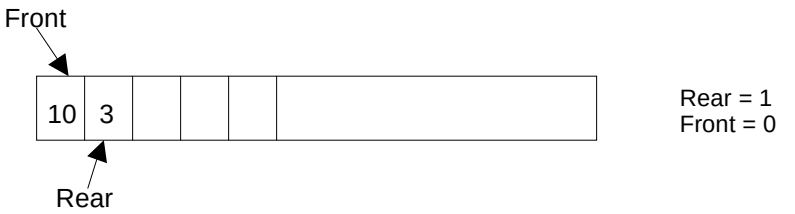


Fig. 4.3. push(3)

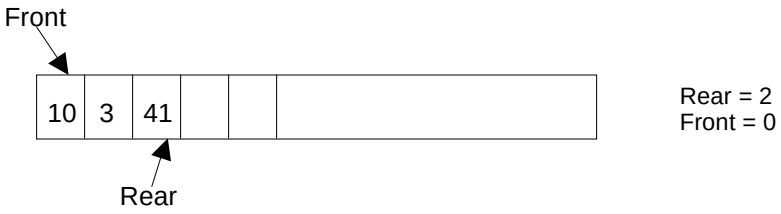


Fig. 4.4. push(41)

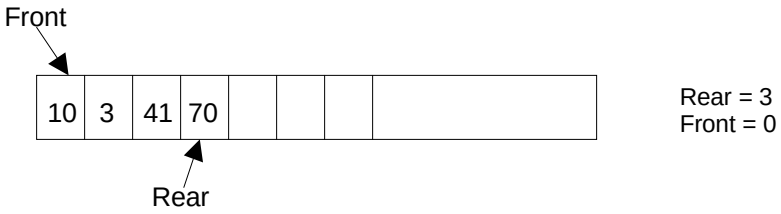


Fig. 4.5. push(70)

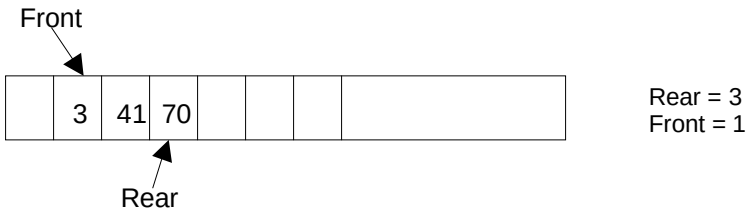


Fig. 4.6. x = pop() (*i.e.*; x = 10)

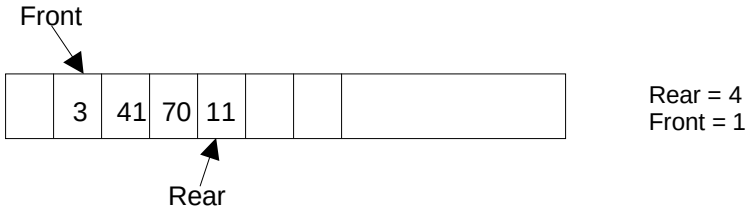


Fig. 4.7. push(11)

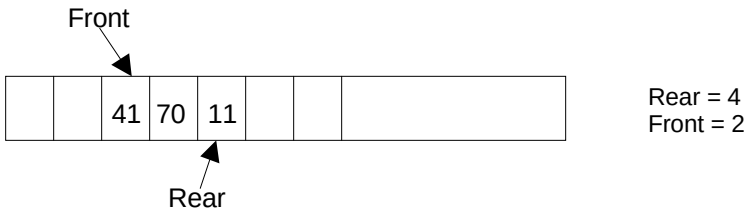


Fig. 4.8. x = pop() (*i.e.*; x = 3)

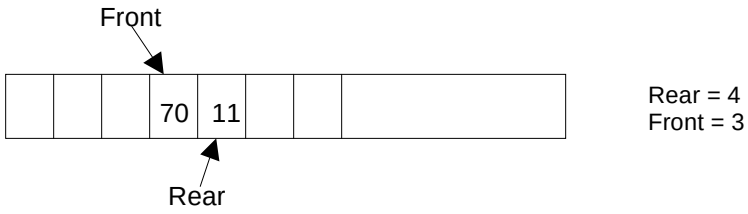


Fig. 4.9. x = pop() (*i.e.*, x = 41)

Queue can be implemented in two ways:

1. Using arrays (static)
2. Using pointers (dynamic)

Implementation of queue using pointers will be discussed in chapter 5. Let us discuss underflow and overflow conditions when a queue is implemented using arrays.

If we try to pop (or delete or remove) an element from queue when it is empty, underflow occurs. It is not possible to delete (or take out) any element when there is no element in the queue.

Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements

4.1. ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

4.1.1. INSERTING AN ELEMENT INTO THE QUEUE

1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)
   (*a*) Display “Queue overflow”
   (*b*) Exit
4. Else
   (*a*) Rear = rear +1
5. Q[rear] = data
6. Exit

4.1.2. DELETING AN ELEMENT FROM QUEUE

1. If (rear< front)
   (*a*) Front = 0, rear = –1
   (*b*) Display “The queue is empty”
   (*c*) Exit

2. Else
   (*a*) Data = Q[front]
3. Front = front +1
4. Exit

PROGRAM 4.1

```
//PROGRAM TO IMPLEMENT QUEUE USING ARRAYS
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>
#include<process.h>

#define MAX 50

int queue_arr[MAX];
int rear = –1;
int front = –1;

//This function will insert an element to the queue
void insert ()
{
        int added_item;
        if (rear==MAX-1)
        {
                printf("\nQueue Overflow\n");
                getch();
                return;
        }
        else
        {
                if (front==–1)  /*If queue is initially empty */
                        front=0;
                printf("\nInput the element for adding in queue: ");
                scanf("%d", &added_item);
                rear=rear+1;
                //Inserting the element
                queue_arr[rear] = added_item ;
    }
}/*End of insert()*/
```

```
//This function will delete (or pop) an element from the queue
void del()
{
        if (front == –1 || front > rear)
        {
                printf ("\nQueue Underflow\n");
                return;
        }
        else
        {
                //deleteing the element
    printf ("\nElement deleted from queue is : %d\n",
    queue_arr[front]);
                front=front+1;
        }
}/*End of del()*/

//Displaying all the elements of the queue
void display()
{
        int i;
        //Checking whether the queue is empty or not
        if (front == –1 || front > rear)
        {
                printf (“\nQueue is empty\n”);
                return;
        }
        else
        {
                printf(“\nQueue is :\n”);
                for(i=front;i<= rear;i++)
                        printf(“%d  ”,queue_arr[i]);
                printf(“\n”);
        }
}/*End of display() */

void main()
{
        int choice;
        while (1)
        {
                clrscr();
```

```
            //Menu options
            printf("\n1.Insert\n");
            printf("2.Delete\n");
            printf("3.Display\n");
            printf("4.Quit\n");
            printf("\nEnter your choice:");
            scanf("%d", & choice);
            switch(choice)
            {
            case 1 :
                  insert();
                  break;
            case 2:
                  del();
                  getch();
                  break;
            case 3:
                  display();
                  getch();
                  break;
            case 4:
                  exit(1);
            default:
                  printf ("\n Wrong choice\n");
                  getch();
            }/*End of switch*/
      }/*End of while*/
}/*End of main()*/
```

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

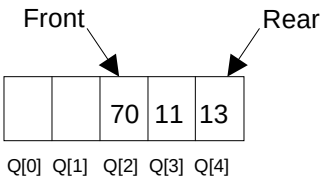


Fig. 4.10

Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.

4.2. OTHER QUEUES

There are three major variations in a simple queue. They are

1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue

Priority queue is generally implemented using linked list, which is discussed in the section 5.13. The other two queue variations are discussed in the following sections.

4.3. CIRCULAR QUEUE

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.

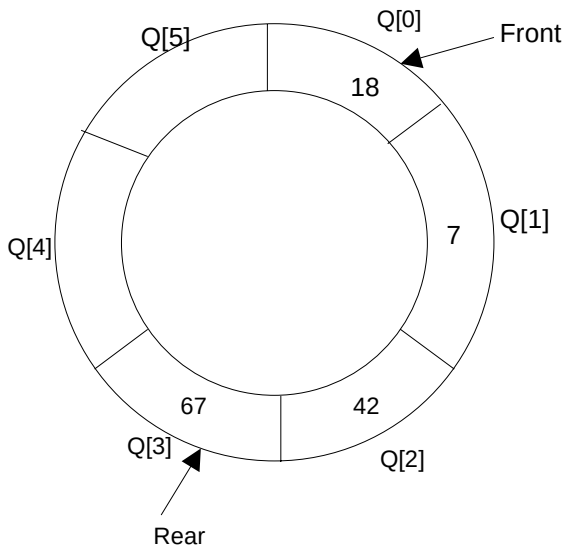


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

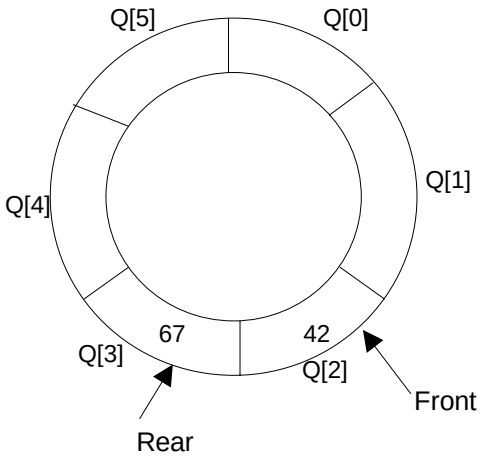


Fig. 4.12. A circular queue after popping 18, 7

After inserting an element at last location Q[5], the next element will be inserted at the very first location (*i.e.*, Q[0]) that is circular queue is one in which the first element comes just after the last element.

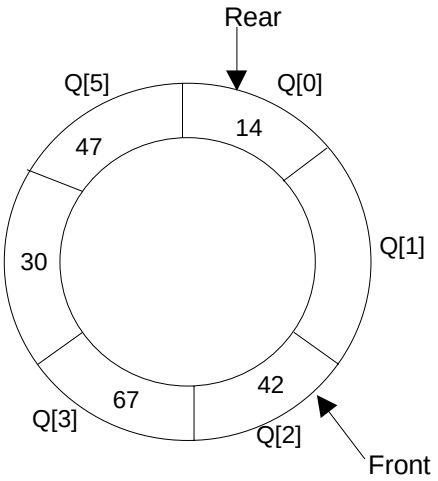


Fig. 4.13. A circular queue after pushing 30, 47, 14

At any time the position of the element to be inserted will be calculated by the relation Rear = (Rear + 1) % SIZE

After deleting an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE

After locating the position of the new element to be inserted, *rear*, compare it with *front*. If (rear = front), the queue is full and cannot be inserted anymore.

4.3.1. ALGORITHMS

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Inserting an element to circular Queue

1. Initialize FRONT = – 1; REAR = 1
2. REAR = (REAR + 1) % SIZE
3. If (FRONT is equal to REAR)
   (*a*) Display "Queue is full"
   (*b*) Exit
4. Else
   (*a*) Input the value to be inserted and assign to variable "DATA"
5. If (FRONT is equal to – 1)
   (*a*) FRONT = 0
   (*b*) REAR = 0
6. Q[REAR] = DATA
7. Repeat steps 2 to 5 if we want to insert more elements
8. Exit

Deleting an element from a circular queue

1. If (FRONT is equal to – 1)
   (*a*) Display "Queue is empty"
   (*b*) Exit
2. Else
   (*a*) DATA = Q[FRONT]
3. If (REAR is equal to FRONT)
   (*a*) FRONT = –1
   (*b*) REAR = –1
4. Else
   (*a*) FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit

PROGRAM 4.2

```
/// PROGRAM TO IMPLEMENT CIRCULAR QUEUE USING ARRAY
//CODED AND COMPILED USING TURBO C++
```

```
#include<conio.h>
#include<process.h>
#include<iostream.h>

#define MAX 50

//A class is created for the circular queue
class circular_queue
{
        int cqueue_arr[MAX];
        int front,rear;

        public:
                //a constructor is created to initialize the variables
                circular_queue()
                {
                        front = –1;
                        rear = –1;
                }
                //public function declarations
                void insert();
                void del();
                void display();
};

//Function to insert an element to the circular queue
void circular_queue::insert()
{
        int added_item;
        //Checking for overflow condition
        if ((front == 0 && rear == MAX-1) || (front == rear +1))
        {
                cout<<"\nQueue Overflow \n";
                getch();
                return;
        }
        if (front == –1)  /*If queue is empty */
        {
                front = 0;
                rear = 0;
        }
```

```
            else
                  if (rear == MAX-1)/*rear is at last position of queue */
                        rear = 0;
                  else
                        rear = rear + 1;
            cout<<"\nInput the element for insertion in queue:";
            cin>>added_item;
            cqueue_arr[rear] = added_item;
}/*End of insert()*/

//This function will delete an element from the queue
void circular_queue::del()
{
            //Checking for queue underflow
            if (front == –1)
            {
                  cout<<"\nQueue Underflow\n";
                  return;
            }
            cout<<"\nElement deleted from queue is:"<<cqueue_arr[front]<<"\n";
            if (front == rear) /* queue has only one element */
            {
                  front = –1;
                  rear = –1;
            }
            else
                  if(front == MAX-1)
                        front = 0;
                  else
                        front = front + 1;
}/*End of del()*/

//Function to display the elements in the queue
void circular_queue::display()
{
            int front_pos = front,rear_pos = rear;
            //Checking whether the circular queue is empty or not
            if (front == –1)
            {
                  cout<<"\nQueue is empty\n";
                  return;
            }
```

```
            //Displaying the queue elements
            cout<<"\nQueue elements:\n";
            if(front_pos <= rear_pos )
                  while(front_pos <= rear_pos)
                  {
                        cout<<cqueue_arr[front_pos]<<", ";
                        front_pos++;
                  }
            else
            {
                  while(front_pos <= MAX-1)
                  {
                        cout<<cqueue_arr[front_pos]<<", ";
                        front_pos++;
                  }
                  front_pos = 0;
                  while(front_pos <= rear_pos)
                  {
                        cout<<cqueue_arr[front_pos]<<", ";
                        front_pos++;
                  }
            }/*End of else*/
            cout<<"\n";
}/*End of display() */

void main()
{
            int choice;
            //Creating the objects for the class
            circular_queue co;
            while(1)
            {
                  clrscr();
                  //Menu options
                  cout <<"\n1.Insert\n";
                  cout <<"2.Delete\n";
                  cout <<"3.Display\n";
                  cout <<"4.Quit\n";
                  cout <<"\nEnter your choice: ";
                  cin>>choice;

                  switch(choice)
```

```
                {
                case 1:
                        co.insert();
                        break;
                case 2 :
                        co.del();
                        getch();
                        break;
                case 3:
                        co.display();
                        getch();
                        break;
                case 4:
                        exit(1);
                default:
                        cout<<"\nWrong choice\n";
                        getch();
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

4.4. DEQUES

A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

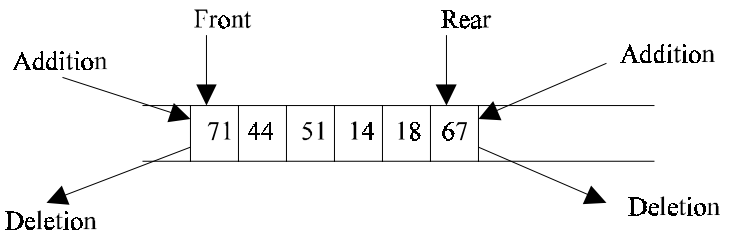


Fig. 4.14. A deque

There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque
2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.

An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation performed on deque is

1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Only 1^{st}, 3^{rd} and 4^{th} operations are performed by input-restricted deque and 1^{st}, 2^{nd} and 3^{rd} operations are performed by output-restricted deque.

4.4.1. ALGORITHMS FOR INSERTING AN ELEMENT

Let Q be the array of MAX elements. *front* (or left) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue *left* and *right* pointer will point to the – 1.

INSERT AN ELEMENT AT THE RIGHT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right + 1))
   (*a*) Display "Queue Overflow"
   (*b*) Exit
3. If (left == –1)
   (*a*) left = 0
   (*b*) right = 0
4. Else
   (*a*) if (right == MAX –1)
      (*i*) left = 0
   (*b*) else
      (*i*) right = right+1
5. Q[right] = DATA
6. Exit

INSERT AN ELEMENT AT THE LEFT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right+1))
   (*a*) Display "Queue Overflow"
   (*b*) Exit

3. If (left == – 1)
   (*a*) Left = 0
   (*b*) Right = 0
4. Else
   (*a*) if (left == 0)
      (*i*) left = MAX – 1
   (*b*) else
      (*i*) left = left – 1
5. Q[left] = DATA
6. Exit

4.4.2. ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

DELETE AN ELEMENT FROM THE RIGHT SIDE OF THE DE-QUEUE

1. If (left == – 1)
   (*a*) Display "Queue Underflow"
   (*b*) Exit
2. DATA = Q [right]
3. If (left == right)
   (*a*) left = – 1
   (*b*) right = – 1
4. Else
   (*a*) if(right == 0)
      (*i*) right = MAX-1
   (*b*) else
      (*i*) right = right-1
5. Exit

DELETE AN ELEMENT FROM THE LEFT SIDE OF THE DE-QUEUE

1. If (left == – 1)
   (*a*) Display "Queue Underflow"
   (*b*) Exit
2. DATA = Q [left]
3. If(left == right)
   (*a*) left = – 1
   (*b*) right = – 1

4. Else
   (*a*) if (left == MAX-1)
      (*i*) left = 0
   (*b*) Else
      (*i*) left = left +1
5. Exit

PROGRAM 4.3

```
//PROGRAM TO IMPLEMENT INPUT AND OUTPUT
//RESTRICTED DE-QUEUE USING ARRAYS
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>
#include<process.h>

#define MAX 50

int deque_arr[MAX];
int left = –1;
int right = –1;

//This function will insert an element at the
//right side of the de-queue
void insert_right()
{
        int added_item;
        if ((left == 0 && right == MAX-1) || (left == right+1))
        {
                printf ("\nQueue Overflow\n");
                getch();
                return;
        }
        if (left == –1)  /* if queue is initially empty */
        {
                left = 0;
                right = 0;
        }
        else
```

```
            if(right == MAX-1)  /*right is at last position of queue */
                  right = 0;
            else
                  right = right+1;
            printf("\n Input the element for adding in queue: ");
            scanf ("%d", &added_item);
            //Inputting the element at the right
            deque_arr[right] = added_item ;
}/*End of insert_right()*/

//Function to insert an element at the left position
//of the de-queue
void insert_left()
{
            int added_item;
            //Checking for queue overflow
            if ((left == 0 && right == MAX-1) || (left == right+1))
            {
                  printf ("\nQueue Overflow \n");
                  getch();
                  return;
            }
            if (left == –1)/*If queue is initially empty*/
            {
                  left = 0;
                  right = 0;
            }
            else
            if (left== 0)
                  left = MAX –1;
            else
                  left = left-1;
            printf("\nInput the element for adding in queue:");
            scanf ("%d", &added_item);
            //inputting at the left side of the queue
            deque_arr[left] = added_item ;
}/*End of insert_left()*/

//This function will delete an element from the queue
//from the left side
void delete_left()
{
```

```
        //Checking for queue underflow
        if (left == –1)
        {
                printf("\nQueue Underflow\n");
                return;
        }
        //deleting the element from the left side
        printf ("\nElement deleted from queue is: %d\n",deque_arr[left]);
        if(left == right) /*Queue has only one element */
        {
                left = –1;
                right=–1;
        }
        else
                if (left == MAX-1)
                        left = 0;
                else
                        left = left+1;
}/*End of delete_left()*/

//Function to delete an element from the right hand
//side of the de-queue
void delete_right()
{
        //Checking for underflow conditions
        if (left == –1)
        {
                printf("\nQueue Underflow\n");
                return;
        }
        printf("\nElement deleted from queue is : %d\n",deque_arr[right]);
        if(left == right) /*queue has only one element*/
        {
                left = –1;
                right=–1;
        }
        else
                if (right == 0)
                        right=MAX-1;
                else
                        right=right-1;
}/*End of delete_right() */
```

```
//Displaying all the contents of the queue
void display_queue()
{
        int front_pos = left, rear_pos = right;
        //Checking whether the queue is empty or not
        if (left == –1)
        {
                printf (“\nQueue is empty\n”);
                return;
        }
        //displaying the queue elements
        printf ("\nQueue elements :\n");
        if ( front_pos <= rear_pos )
        {
                while(front_pos <= rear_pos)
                {
                        printf (“%d ”,deque_arr[front_pos]);
                        front_pos++;
                }
        }
        else
        {
                while(front_pos <= MAX-1)
                {
                        printf(“%d ”,deque_arr[front_pos]);
                        front_pos++;
                }
                front_pos = 0;
                while(front_pos <= rear_pos)
                {
                        printf (“%d ”,deque_arr[front_pos]);
                        front_pos++;
                }
        }/*End of else */
        printf (“\n”);
}/*End of display_queue() */

//Function to implement all the operation of the
//input restricted queue
void input_que()
{
        int choice;
        while(1)
```

```
        {
                clrscr();
                //menu options to input restricted queue
                printf ("\n1.Insert at right\n");
                printf ("2.Delete from left\n");
                printf ("3.Delete from right\n");
                printf ("4.Display\n");
                printf ("5.Quit\n");
                printf ("\nEnter your choice : ");
                scanf ("%d",&choice);

                switch(choice)
                {
                 case 1:
                        insert_right();
                        break;
                 case 2:
                        delete_left();
                        getch();
                        break;
                 case 3:
                        delete_right();
                        getch();
                        break;
                 case 4:
                        display_queue();
                        getch();
                        break;
                 case 5:
                        exit(0);
                 default:
                        printf("\nWrong choice\n");
                        getch();
                }/*End of switch*/
        }/*End of while*/
}/*End of input_que() */

//This function will implement all the operation of the
//output restricted queue
void output_que()
{
        int choice;
```

```
        while(1)
        {
                clrscr();
                //menu options for output restricted queue
                printf ("\n1.Insert at right\n");
                printf ("2.Insert at left\n");
                printf ("3.Delete from left\n");
                printf ("4.Display\n");
                printf ("5.Quit\n");
                printf ("\nEnter your choice:");
                scanf ("%d",&choice);

                switch(choice)
                {
                 case 1:
                        insert_right();
                        break;
                 case 2:
                        insert_left();
                        break;
                 case 3:
                        delete_left();
                        getch();
                        break;
                 case 4:
                        display_queue();
                        getch();
                        break;
                 case 5:
                        exit(0);
                 default:
                        printf("\nWrong choice\n");
                        getch();
                }/*End of switch*/
        }/*End of while*/
}/*End of output_que() */

void main()
{
        int choice;
        clrscr();
        //Main menu options
        printf ("\n1.Input restricted dequeue\n");
```

```
        printf ("2.Output restricted dequeue\n");
        printf ("Enter your choice:");
        scanf ("%d",&choice);

        switch(choice)
        {
         case 1:
               input_que();
               break;
         case 2:
               output_que();
               break;
         default:
               printf("\nWrong choice\n");
        }/*End of switch*/
}/*End of main()*/
```

If we analyze the algorithms in this chapter the time needed to add or delete a data is constant, *i.e.* time complexity is of order O(1).

4.5. APPLICATIONS OF QUEUE

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.

5 Linked List

If the memory is allocated for the variable during the compilation (*i.e.; before execution)* of a program, then it is fixed and cannot be changed. For example, an array A[100] is declared with 100 elements, then the allocated memory is fixed and cannot decrease or increase the SIZE of the array if required. So we have to adopt an alternative strategy to allocate memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not require the use of arrays.

5.1. LINKED LISTS

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field. Following Fig 5:1 shows a typical example of node.

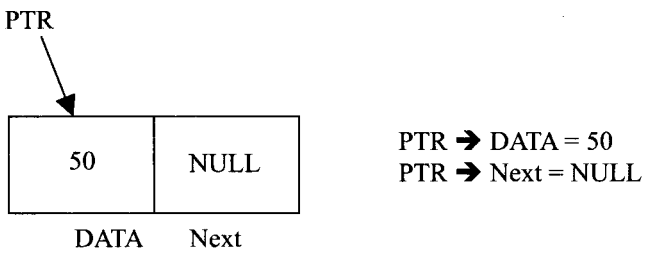


Fig. 5.1. Nodes.

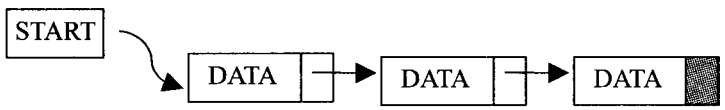


Fig. 5.2. Linked List.

Fig. 5.3. Linked List representation in memory.

Fig. 5.2 shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START = NULL if there is no list (*i.e.*; NULL list or empty list).

5.2. REPRESENTATION OF LINKED LIST

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

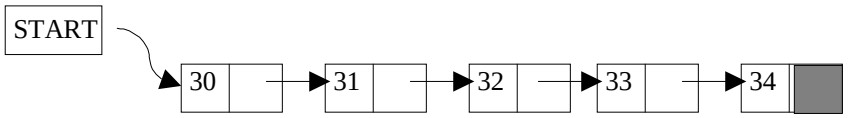


Fig. 5.4. Linked list representation of integers

The linear linked list can be represented in memory with the following declaration.

```
struct Node
{
        int DATA; //Instead of 'DATA' we also use 'Info'
        struct Node *Next; //Instead of 'Next' we also use 'Link'
};
typedef struct Node *NODE;
```

5.3. ADVANTAGES AND DISADVANTAGES

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.

3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

5.4. OPERATION ON LINKED LIST

The primitive operations performed on the linked list are as follows

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Concatenation

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(*a*) At the beginning of the linked list

(*b*) At the end of the linked list

(*c*) At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

(*a*) Beginning of a linked list

(*b*) End of a linked list

(*c*) Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Concatenation is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes

5.5. TYPES OF LINKED LIST

Basically we can divide the linked list into the following three types in the order in which they (or node) are arranged.

1. Singly linked list
2. Doubly linked list
3. Circular linked list

5.6. SINGLY LINKED LIST

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure. Following figure explains the different operations on a singly linked list.

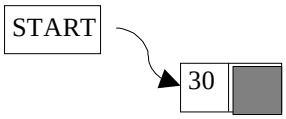


Fig. 5.5. Create a node with DATA(30)

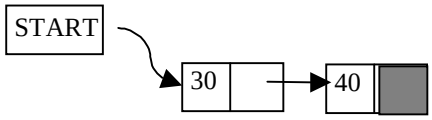


Fig. 5.6. Insert a node with DATA(40) at the end

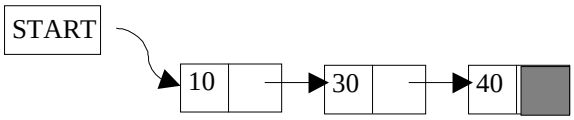


Fig. 5.7. Insert a node with DATA(10) at the beginning

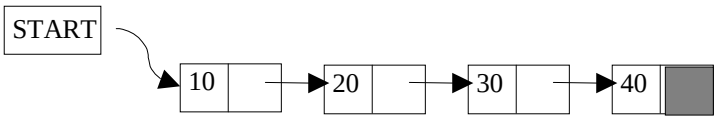


Fig. 5.8. Insert a node with DATA(20) at the 2nd position

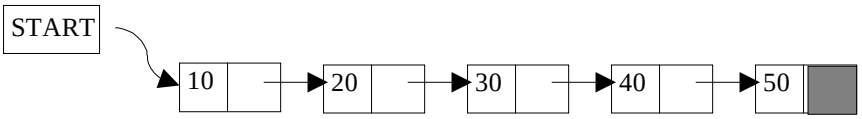


Fig. 5.9. Insert a node with DATA(50) at the end

Output → 10, 20, 30, 40, 50

Fig. 5.10. Traversing the nodes from left to right

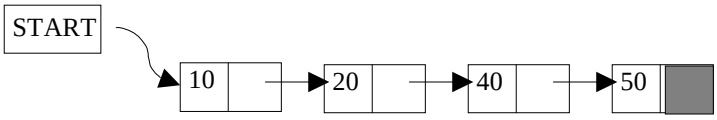


Fig. 5.11. Delete the 3rd node from the list

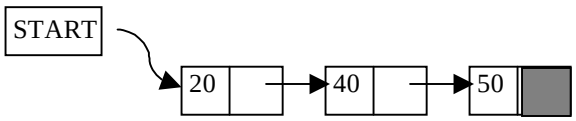


Fig. 5.12. Delete the 1st node

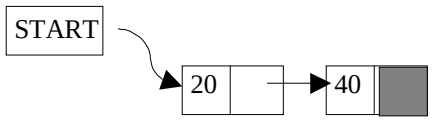


Fig. 5.13. Delete the last node

5.6.1. ALGORITHM FOR INSERTING A NODE

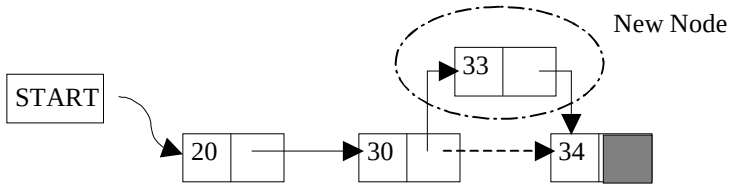


Fig. 5.14. Insertion of New Node

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

Insert a Node at the beginning

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. If (SATRT equal to NULL)
   (*a*) NewNode → Link = NULL
5. Else
   (*a*) NewNode → Link = START
6. START = NewNode
7. Exit

Insert a Node at the end

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
8. If (SATRT equal to NULL)
   (*a*) START = NewNode

9. Else
   (*a*) TEMP = START
   (*b*) While (TEMP → Next not equal to NULL)
      (*i*) TEMP = TEMP → Next
10. TEMP → Next = NewNode
11. Exit

Insert a Node at any specified position

1. Input DATA and POS to be inserted
2. intialise TEMP = START; and j = 0
3. Repeat the step 3 while(k is less than POS)
   (*a*) TEMP = TEMP è Next
   (*b*) If (TEMP is equal to NULL)
      (*i*) Display "Node in the list less than the position"
      (*ii*) Exit
   (*c*) *k* = *k* + 1
4. Create a New Node
5. NewNode → DATA = DATA
6. NewNode → Next = TEMP → Next
7. TEMP → Next = NewNode
8. Exit

5.6.2. ALGORITHM FOR DELETING A NODE

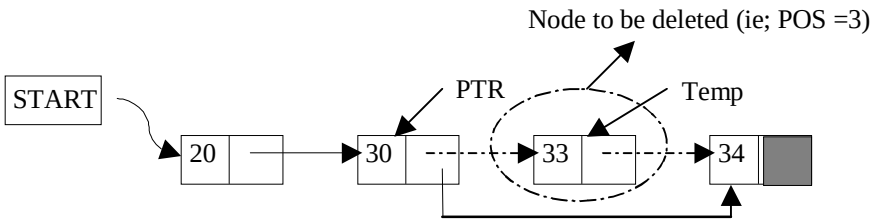


Fig. 5.15. Deletion of a Node.

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

1. Input the DATA to be deleted
2. if ((START → DATA) is equal to DATA)
   (*a*) TEMP = START
   (*b*) START = START → Next
   (*c*) Set free the node TEMP, which is deleted
   (*d*) Exit

3. HOLD = START
4. while ((HOLD → Next → Next) not equal to NULL))
   (*a*) if ((HOLD → NEXT → DATA) equal to DATA)
      (*i*) TEMP = HOLD → Next
      (*ii*) HOLD → Next = TEMP → Next
      (*iii*) Set free the node TEMP, which is deleted
      (*iv*) Exit
   (*b*) HOLD = HOLD → Next
5. if ((HOLD → next → DATA) == DATA)
   (*a*) TEMP = HOLD → Next
   (*b*) Set free the node TEMP, which is deleted
   (*c*) HOLD → Next = NULL
   (*d*) Exit
6. Disply "DATA not found"
7. Exit

5.6.3. ALGORITHM FOR SEARCHING A NODE

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
   (*a*) Display "The data is found at POS"
   (*b*) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
   (*a*) Display "The data is not found in the list"
8. Exit

5.6.4. ALGORITHM FOR DISPLAY ALL NODES

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

1. If (START is equal to NULL)
   (*a*) Display "The list is Empty"
   (*b*) Exit
2. Initialize TEMP = START

3. Repeat the step 4 and 5 until (TEMP == NULL)
4. Display "TEMP → DATA"
5. TEMP = TEMP → Next
6. Exit

PROGRAM 5.1

```
//THIS PROGRAM WILL IMPLEMENT ALL THE OPERATIONS
//OF THE SINGLY LINKED LIST
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>

//Structure declaration for the node
struct node
{
        int info;
        struct node *link;
}*start;

//This function will create a new linked list
void Create_List(int data)
{
        struct node *q,*tmp;
        //Dynamic memory is been allocated for a node
        tmp= (struct node*)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=NULL;

        if(start==NULL) /*If list is empty*/
                start=tmp;
        else
        {       /*Element inserted at the end*/
                q=start;
                while(q->link!=NULL)
                        q=q->link;
                q->link=tmp;
        }
```

```
}/*End of create_list()*/

//This function will add new element at the beginning of the linked list
void AddAtBeg(int data)
{
        struct node *tmp;
        tmp=(struct node*)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=start;
        start=tmp;
}/*End of addatbeg()*/

//Following function will add new element at any position
void AddAfter(int data,int pos)
{
        struct node *tmp,*q;
        int i;
        q=start;
        //Finding the position to add new element to the linked list
        for(i=0;i<pos-1;i++)
        {
                q=q->link;
                if(q==NULL)
                {
                        printf ("\n\n There are less than %d elements",pos);
                        getch();
                        return;
                }
        }/*End of for*/

        tmp=(struct node*)malloc(sizeof (struct node));
        tmp->link=q->link;
        tmp->info=data;
        q->link=tmp;
}/*End of addafter()*/

//Delete any element from the linked list
void Del(int data)
{
        struct node *tmp,*q;
        if (start->info == data)
        {
```

```
                tmp=start;
                start=start->link; /*First element deleted*/
                free(tmp);
                return;
        }
        q=start;
        while(q->link->link ! = NULL)
        {
                if(q->link->info == data) /*Element deleted in between*/
                {
                        tmp=q->link;
                        q->link=tmp->link;
                        free(tmp);
                        return;
                }
                q=q->link;
        }/*End of while */
        if(q->link->info==data) /*Last element deleted*/
        {
                tmp=q->link;
                free(tmp);
                q->link=NULL;
                return;
        }
        printf ("\n\nElement %d not found",data);
        getch();
}/*End of del()*/

//This function will display all the element(s) in the linked list
void Display()
{
        struct node *q;
        if(start == NULL)
        {
                printf ("\n\nList is empty");
                return;
        }
        q=start;
        printf("\n\nList is : ");
        while(q!=NULL)
        {
                printf ("%d ", q->info);
                q=q->link;
```

```
        }
        printf ("\n");
        getch();
}/*End of display() */

//Function to count the number of nodes in the linked list
void Count()
{
        struct node *q=start;
        int cnt=0;
        while(q!=NULL)
        {
                q=q->link;
                cnt++;
        }
        printf ("Number of elements are %d\n",cnt);
        getch();
}/*End of count()*/

//This function will reverse the linked list
void Rev()
{
        struct node *p1,*p2,*p3;
        if(start->link==NULL)    /*only one element*/
                return;
        p1=start;
        p2=p1->link;
        p3=p2->link;
        p1->link=NULL;
        p2->link=p1;
        while(p3!=NULL)
        {
                p1=p2;
                p2=p3;
                p3=p3->link;
                p2->link=p1;
        }
        start=p2;
}/*End of rev()*/

//Function to search an element from the linked list
void Search(int data)
{
```

```
        struct node *ptr = start;
        int pos = 1;
        //searching for an element in the linked list
        while(ptr!=NULL)
        {
            if (ptr->info==data)
            {
                printf ("\n\nItem %d found at position %d", data, pos);
                getch();
                return;
            }
            ptr = ptr->link;
            pos++;
        }
        if (ptr == NULL)
            printf ("\n\nItem %d not found in list",data);
        getch();
}


void main()
{
        int choice,n,m,position,i;
        start=NULL;
        while(1)
        {
            clrscr();
            printf ("1.Create List\n");
            printf ("2.Add at beginning\n");
            printf ("3.Add after \n");
            printf ("4.Delete\n");
            printf ("5.Display\n");
            printf ("6.Count\n");
            printf ("7.Reverse\n");
            printf ("8.Search\n");
            printf ("9.Quit\n");
            printf ("\nEnter your choice:");
            scanf ("%d",&choice);
            switch (choice)
            {
             case 1:
                printf ("\n\nHow many nodes you want:");
                scanf ("%d",&n);
```

```
        for(i = 0;i<n;i++)
        {
                printf ("\nEnter the element:");
                scanf ("%d",&m);
                Create_List(m);
        }
        break;
case 2:
        printf ("\n\nEnter the element : ");
        scanf ("%d",&m);
        AddAtBeg(m);
        break;
case 3:
        printf ("\n\nEnter the element:");
        scanf ("%d",&m);
        printf ("\nEnter the position after which this element is inserted:");
        scanf ("%d",&position);
        Add After(m,position);
        break;
case 4:
        if (start == NULL)
        {
                printf("\n\nList is empty");
                continue;
        }
        printf ("\n\nEnter the element for deletion:");
        scanf ("%d",&m);
        Del(m);
        break;
case 5:
        Display();
        break;
case 6:
        Count();
        break;
case 7:
        Rev();
        break;
case 8:
        printf("\n\nEnter the element to be searched:");
        scanf ("%d",&m);
```

```
                        Search(m);
                        break;
                    case 9:
                        exit(0);
                    default:
                        printf ("\n\nWrong choice");
                    }/*End of switch*/
                }/*End of while*/
}/*End of main()*/
```

PROGRAM 5.2

```
//THIS PROGRAM WILL IMPLEMENT ALL THE OPERATIONS
//OF THE SINGLY LINKED LIST
//CODED AND COMPILED IN TURBO C++


#include<iostream.h>
#include<conio.h>
#include<process.h>


class Linked_List
{
        //Structure declaration for the node
        struct node
        {
                        int info;
                        struct node *link;
        };

                    //private structure variable declared
                    struct node *start;
                public:
                    Linked_List()//Constructor defined
                    {
                        start = NULL;
                    }
                    //public fucntion declared
                    void Create_List(int);
```

```
                void AddAtBeg(int);
                void AddAfter(int,int);
                void Delete();
                void Count();
                void Search(int);
                void Display();
                void Reverse();
};

//This function will create a new linked list of elements
void Linked_List::Create_List(int data)
{
        struct node *q,*tmp;
        //New node is created with new operator
        tmp= (struct node *)new(struct node);
        tmp->info=data;
        tmp->link=NULL;

        if (start==NULL) /*If list is empty */
                start=tmp;
        else
        {       /*Element inserted at the end */
                q=start;
                while(q->link!=NULL)
                        q=q->link;
                q-> link=tmp;
        }
}/*End of create_list()*/

//following function will add new element at the beginning
void Linked_List::AddAtBeg(int data)
{
        struct node *tmp;
        tmp=(struct node*)new(struct node);
        tmp->info=data;
        tmp->link=start;
        start=tmp;
}/*End of addatbeg()*/

//This function will add new element at any specified position
void Linked_List::AddAfter(int data,int pos)
{
```

```
        struct node *tmp,*q;
        int i;
        q=start;
        //Finding the position in the linked list to insert
        for(i=0;i<pos-1;i++)
        {
                q=q->link;
                if(q==NULL)
                {
                        cout<<"\n\nThere are less than "<<pos<<" elements";
                        getch();
                        return;
                }
        }/*End of for*/

        tmp=(struct node*)new(struct node);
        tmp->link=q->link;
        tmp->info=data;
        q->link=tmp;
}/*End of addafter()*/

//Funtion to delete an element from the list
void Linked_List::Delete()
{
        struct node *tmp,*q;
        int data;
        if(start==NULL)
        {
                cout<<"\n\nList is empty";
                getch();
                return;
        }
        cout<<"\n\nEnter the element for deletion : ";
        cin>>data;

        if(start->info == data)
        {
                tmp=start;
                start=start->link; //First element deleted
                delete(tmp);
                return;
        }
```

```
        q=start;
        while(q->link->link != NULL)
        {
            if(q->link->info==data) //Element deleted in between
            {
                tmp=q->link;
                q->link=tmp->link;
                delete(tmp);
                return;
            }
            q=q->link;
        }/*End of while */
        if(q->link->info==data) //Last element deleted
        {
            tmp=q->link;
            delete(tmp);
            q->link=NULL;
            return;
        }
        cout<<"\n\nElement "<<data<<" not found";
        getch();
}/*End of del()*/

void Linked_List::Display()
{
        struct node *q;
        if(start == NULL)
        {
            cout<<"\n\nList is empty";
            return;
        }
        q=start;
        cout<<"\n\nList is : ";
        while(q!=NULL)
        {
            cout<<q->info;
            q=q->link;
        }
        cout<<"\n";
        getch();
}/*End of display() */
```

```
void Linked_List::Count()
{
        struct node *q=start;
        int cnt=0;
        while(q!=NULL)
        {
                q=q->link;
                cnt++;
        }
        cout<<"Number of elements are \n"<<cnt;
        getch();
}/*End of count() */

void Linked_List::Reverse()
{
        struct node *p1,*p2,*p3;
        if(start->link==NULL)    /*only one element*/
                return;
        p1=start;
        p2=p1->link;
        p3=p2->link;
        p1->link=NULL;
        p2->link=p1;
        while(p3!=NULL)
        {
                p1=p2;
                p2=p3;
                p3=p3->link;
                p2->link=p1;
        }
        start=p2;
}/*End of rev()*/

void Linked_List::Search(int data)
{
        struct node *ptr = start;
        int pos = 1;
        while(ptr!=NULL)
        {
                if(ptr->info==data)
                {
                        cout<<"\n\nItem "<<data<<" found at position "<<pos;
```

```
                        getch();
                        return;
                }
                ptr = ptr->link;
                pos++;
        }
        if(ptr == NULL)
                cout<<"\n\nItem "<<data<<" not found in list";
        getch();
}


void main()
{
        int choice,n,m,position,i;
        Linked_List po;
        while(1)
        {
                clrscr();
                cout<<"1.Create List\n";
                cout<<"2.Add at begining\n";
                cout<<"3.Add after \n";
                cout<<"4.Delete\n";
                cout<<"5.Display\n";
                cout<<"6.Count\n";
                cout<<"7.Reverse\n";
                cout<<"8.Search\n";
                cout<<"9.Quit\n";
                cout<<"\nEnter your choice:";
                cin>>choice;
                switch(choice)
                {
                 case 1:
                        cout<<"\n\nHow many nodes you want:";
                        cin>>n;
                        for(i=0;i<n;i++)
                        {
                                cout<<"\nEnter the element:";
                                cin>>m;
                                po.Create_List(m);
                        }
                        break;
```

```
                case 2:
                        cout<<"\n\nEnter the element:";
                        cin>>m;
                        po.AddAtBeg(m);
                        break;
                case 3:
                        cout<<"\n\nEnter the element:";
                        cin>>m;
                        cout<<"\nEnter the position after which this element is inserted:";
                        cin>>position;
                        po.AddAfter(m,position);
                        break;
                case 4:
                        po.Delete();
                        break;
                case 5:
                        po.Display();
                        break;
                case 6:
                        po.Count();
                        break;
                case 7:
                        po.Reverse();
                        break;
                case 8:
                        cout<<"\n\nEnter the element to be searched:";
                        cin>>m;
                        po.Search(m);
                        break;
                case 9:
                        exit(0);
                default:
                        cout<<"\n\nWrong choice";
                }/*End of switch */
        }/*End of while */
}/*End of main()*/
```

5.7. STACK USING LINKED LIST

In chapter 3, we have discussed what a stack means and its different operations. And we have also discussed the implementation of stack using array, *i.e.*, static memory

allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.

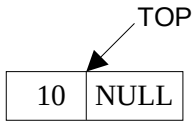


Fig. 5.11. push (10)

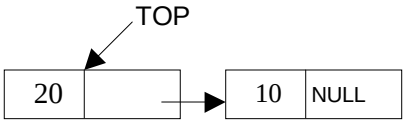


Fig. 5.12. push (20)

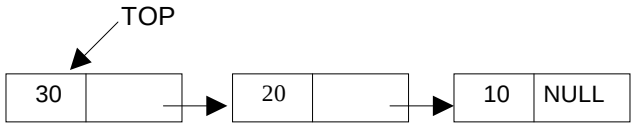


Fig. 5.13. push (30)

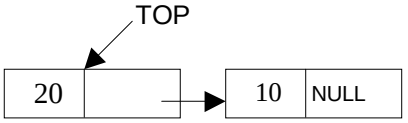


Fig. 5.14. X = pop() (ie; X = 30)

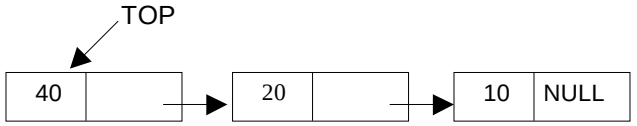


Fig. 5.15. push (40)

5.7.1. ALGORITHM FOR PUSH OPERATION

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit

5.7.2. ALGORITHM FOR POP OPERATION

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
   (*a*) Display "The stack is empty"
2. Else
   (*a*) TEMP = TOP
   (*b*) Display "The popped element TOP → DATA"
   (*c*) TOP = TOP → Next
   (*d*) TEMP → Next = NULL
   (*e*) Free the TEMP node
3. Exit

PROGRAM 5.3

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS
//PERFORMED ON THE STACK INPLEMENTED USING LINKED LIST
//CODED AND COMPILED IN TURBO C
#include<conio.h>
#include<stdio.h>
#include<malloc.h>
#include<process.h>

//Structure is created a node
struct node
{
        int info;
        struct node *link;//A link to the next node
};

//A variable named NODE is been defined for the structure
typedef struct node *NODE;

//This function is to perform the push operation
NODE push(NODE top)
{
        NODE NewNode;
        int pushed_item;
        //A new node is created dynamically
```

```
            NewNode = (NODE)malloc(sizeof(struct node));
            printf("\nInput the new value to be pushed on the stack:");
            scanf("%d",&pushed_item);
            NewNode->info=pushed_item;//Data is pushed to the stack
            NewNode->link=top;//Link pointer is set to the next node
            top=NewNode;//Top pointer is set
            return(top);
}/*End of push()*/

//Following function will implement the pop operation
NODE pop(NODE top)
{
            NODE tmp;
            if(top == NULL)//checking whether the stack is empty or not
                  printf ("\nStack is empty\n");
            else
            {
      tmp=top;//popping the element
                  printf("\nPopped item is %d\n",tmp->info);
                  top=top->link;//resetting the top pointer
                  tmp->link=NULL
                  free(tmp);//freeing the popped node
            }
            return(top);

}/*End of pop()*/

//This is to display the entire element in the stack
void display(NODE top)
{
            if(top==NULL)
                  printf("\nStack is empty\n");
            else
            {
                  printf("\nStack elements:\n");
                  while(top != NULL)
                  {
                        printf("%d\n",top->info);
                        top = top->link;
                  }/*End of while */
            }/*End of else*/
}/*End of display()*/
```

```
void main()
{
        char opt;
        int choice;
        NODE Top=NULL;
        do
        {
                clrscr();
                printf("\n1.PUSH\n");
                printf("2.POP\n");
                printf("3.DISPLAY\n");
                printf("4.EXIT\n");
                printf("\nEnter your choice:");
                scanf("%d", &choice);
                switch(choice)
                {
                case 1:
                        Top=push(Top);
                        break;
                case 2:
                        Top=pop(Top);
                        break;
                case 3:
                        display(Top);
                        break;
                case 4:
                        exit(1);
                default:
                        printf("\nWrong choice\n");
                }/*End of switch*/

                printf ("\n\nDo you want to continue (Y/y) = ");
                fflush(stdin);
                scanf("%c",&opt);
        }while((opt == 'Y') || (opt == 'y'));
}/*End of main() */
```

PROGRAM 5.4

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS
//PERFORMED ON THE STACK IMPLEMENTATION
```

```
//USING LINKED LIST
//CODED AND COMPILED IN TURBO C++

#include<conio.h>
#include<iostream.h>
#include<process.h>

//Class is created for the linked list
class Stack_Linked
{
        //Structure is created for the node
        struct node
        {
                int info;
                struct node *link;//A link to the next node
        };

        //A variable top is been declared for the structure
        struct node *top;
        //NODE is defined as the data type of the structure node
        typedef struct node *NODE;

        public:
                //Constructer is defined for the class
                Stack_Linked()
                {
                        //top pointer is initialized
                        top=NULL;
                }
                //function declarations
                void push();
                void pop();
                void display();
};

//This function is to perform the push operation
void Stack_Linked::push()
{
        NODE NewNode;
        int pushed_item;
        //A new node is created dynamically
        NewNode=(NODE)new(struct node);
        cout<<"\nInput the new value to be pushed on the stack:";
```

```
            cin>>pushed_item;
            NewNode->info=pushed_item;//Data is pushed to the stack
            NewNode->link=top;//Link pointer is set to the next node
            top=NewNode;//Top pointer is set
}/*End of push()*/

//Following function will implement the pop operation
void Stack_Linked::pop()
{
            NODE tmp;
            if(top == NULL)//checking whether the stack is empty or not
                  cout<<"\nStack is empty\n";
            else
            {     tmp=top;//popping the element
                  cout<<"\nPopped item is:"<<tmp->info;
                  top=top->link;//resetting the top pointer
                  tmp->link=NULL;
                  delete(tmp);//freeing the popped node
            }
}/*End of pop()*/

//This is to display all the element in the stack
void Stack_Linked::display()
{
            if(top==NULL)//Checking whether the stack is empty or not
                  cout<<"\nStack is empty\n";
            else
            {
                  NODE ptr=top;
                  cout<<"\nStack elements:\n";
                  while(ptr != NULL)
                  {
                        cout<<"\n"<<ptr->info;
                        ptr = ptr->link;
                  }/*End of while */
            }/*End of else*/
}/*End of display()*/

void main()
{
            char opt;
            int choice;
```

```
        Stack_Linked So;
        do
        {
                clrscr();
                //The menu options are listed below
                cout<<"\n1.PUSH\n";
                cout<<"2.POP\n";
                cout<<"3.DISPLAY\n";
                cout<<"4.EXIT\n";
                cout<<"\nEnter your choice : ";
                cin>>choice;

                switch(choice)
                {
                case 1:
                        So.push();//push function is called
                        break;
                case 2:
                        So.pop();//pop function is called
                        break;
                case 3:
                        So.display();//display function is called
                        break;
                case 4:
                        exit(1);
                default:
                        cout<<"\nWrong choice\n";
                }/*End of switch */

                cout<<"\n\nDo you want to continue (Y/y) = ";
                cin>>opt;
        }while((opt == 'Y') || (opt == 'y'));
}/*End of main() */
```

5.8. QUEUE USING LINKED LIST

Queue is a First In First Out [FIFO] data structure. In chapter 4, we have discussed about stacks and its different operations. And we have also discussed the implementation of stack using array, ie; static memory allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in the following figures.

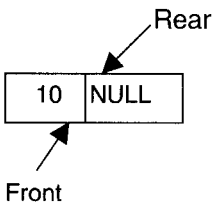


Fig. 5.16. push (10)

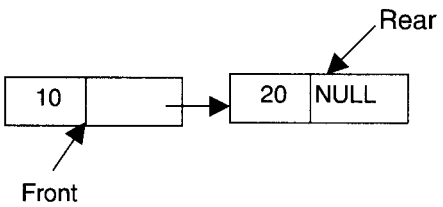


Fig. 5.17. push (20)

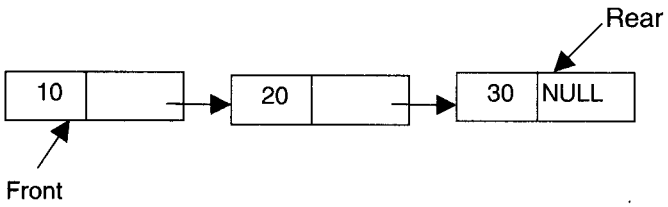


Fig. 5.18. push (30)

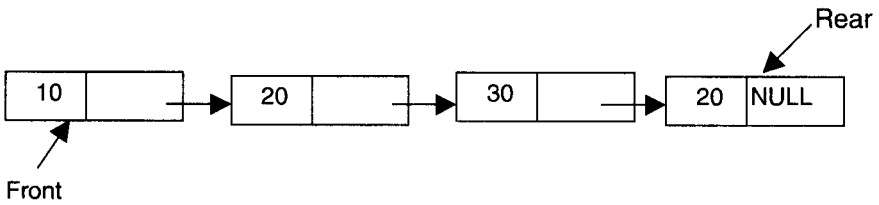


Fig. 5.19. push (40)

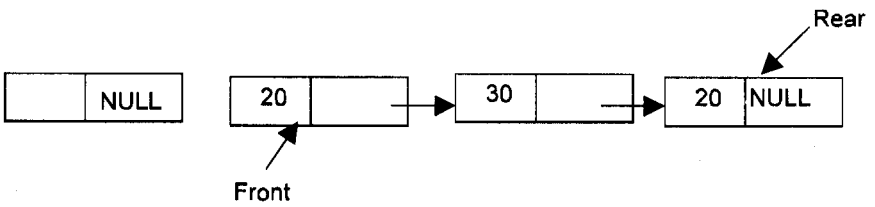


Fig. 5.20. X = pop() (*i.e.*; X = 10)

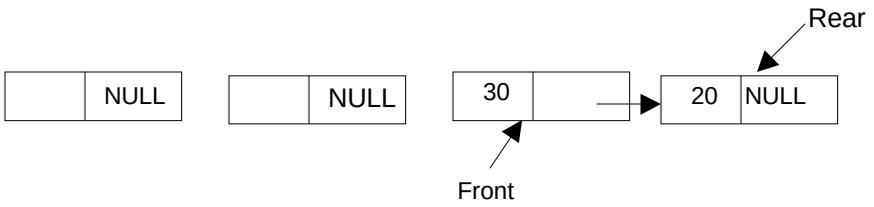


Fig. 5.21. X = pop() (*i.e.*; X = 20)

5.8.1. ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(REAR not equal to NULL)
   (*a*) REAR → next = NewNode;
6. REAR =NewNode;
7. Exit

5.8.2. ALGORITHM FOR POPPING AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT is equal to NULL)
   (*a*) Display "The Queue is empty"
2. Else
   (*a*) Display "The popped element is FRONT → DATA"
   (*b*) If(FRONT is not equal to REAR)
      (*i*) FRONT = FRONT → Next
   (*c*) Else
   (*d*) FRONT = NULL;
3. Exit

PROGRAM 5.5

```
//THIS PROGRAM WILL IMPLEMENT ALL THE OPERATIONS
//OF THE QUEUE, IMPLEMENTED USING LINKED LIST
```

```
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>

//A structure is created for the node in queue
struct queu
{
        int info;
        struct queu *next;//Next node address
};

typedef struct queu *NODE;

//This function will push an element into the queue
NODE push(NODE rear)
{
        NODE NewNode;
        //New node is created to push the data
        NewNode=(NODE)malloc(sizeof(struct queu));
        printf ("\nEnter the no to be pushed = ");
        scanf ("%d",&NewNode->info);
        NewNode->next=NULL;
        //setting the rear pointer
        if (rear != NULL)
                rear->next=NewNode;
        rear=NewNode;
        return(rear);
}

//This function will pop the element from the queue
NODE pop(NODE f,NODE r)
{
        //The Queue is empty when the front pointer is NULL
        if(f==NULL)
                printf ("\nThe Queue is empty");
        else
        {
                printf ("\nThe poped element is = %d",f->info);
                if(f ! = r)
                        f=f->next;
```

```
                else
                        f=NULL;
        }
        return(f);
}

//Function to display the element of the queue
void traverse(NODE fr,NODE re)
{
        //The queue is empty when the front pointer is NULL
        if (fr==NULL)
                printf ("\nThe Queue is empty");
        else
        {
                printf ("\nThe element(s) is/are = ");
                while(fr != re)
                {
                        printf("%d ",fr->info);
                        fr=fr->next;
                };
                printf ("%d",fr->info);
        }
}

void main()
{
        int choice;
        char option;
        //declaring the front and rear pointer
        NODE front, rear;
        //Initializing the front and rear pointer to NULL
        front = rear = NULL;
        dos
        {
                clrscr();
                printf ("1. Push\n");
                printf ("2. Pop\n");
                printf ("3. Traverse\n");
                printf ("\n\nEnter your choice = ");
                scanf ("%d",&choice);
                switch(choice)
                {
```

```
                        case 1:
                                //calling the push function
                                rear = push(rear);
                                if (front==NULL)
                                {
                                        front=rear;
                                }
                                break;
                        case 2:
                                //calling the pop function by passing
                                //front and rear pointers
                                front = pop(front,rear);
                                if (front == NULL)
                                        rear = NULL;
                                break;
                        case 3:
                                traverse(front,rear);
                                break;
                }
                printf ("\n\nPress (Y/y) to continue = ");
                fflush(stdin);
                scanf ("%c",&option);
        }while(option == 'Y' || option == 'y');
}
```

PROGRAM 5.6

```
//THIS PROGRAM WILL IMPLEMENT ALL THE OPERATIONS
//OF THE QUEUE, IMPLEMENTED USING LINKED LIST
//CODED AND COMPILED IN TURBO C++

#include<iostream.h>
#include<conio.h>
#include<malloc.h>

//class is created for the queue
class Queue_Linked
{
        //A structure is created for the node in queue
        struct queu
```

```
        {
                int info;
                struct queu *next;//Next node address
        };

        struct queu *front;
        struct queu *rear;
        typedef struct queu *NODE;

        public:
                //Constructor is created
                Queue_Linked()
                {
                        front = NULL;
                        rear = NULL;
                }
                void push();
                void pop();
                void traverse();
};
//This function will push an element into the queue
void Queue_Linked::push()
{
        NODE NewNode;
        //New node is created to push the data
        NewNode=(NODE)malloc(sizeof(struct queu));
        cout<<"\nEnter the no to be pushed = ";
        cin>>NewNode->info;
        NewNode->next=NULL;
        //setting the rear pointer
        if (rear != NULL)
                rear->next = NewNode;
        rear=NewNode;
        if (front == NULL)
                front = rear;
}

//This function will pop the element from the queue
void Queue_Linked::pop()
{
        //The Queue is empty when the front pointer is NULL
        if (front == NULL)
```

```
        {
                cout<<"\nThe Queue is empty";
                rear = NULL;
        }
        else
        {
                //Front element in the queue is popped
                cout<<"\nThe popped element is = "<<front->info;
                if (front != rear)
                        front=front->next;
                else
                        front = NULL;
        }
}

//Function to display the element of the queue
void Queue_Linked::traverse()
{
        //The queue is empty when the front pointer is NULL
        if (front==NULL)
                cout<<"\nThe Queue is empty";
        else
        {
                NODE Temp_Front=front;
                cout<<"\nThe element(s) is/are = ";
                while(Temp_Front ! = rear)
                {
                        cout<<Temp_Front->info;
                        Temp_Front=Temp_Front->next;
                };
                cout<<Temp_Front->info;
        }
}

void main()
{
        int choice;
        char option;
        Queue_Linked Qo;
        do
        {
                clrscr();
```

```
                cout<<"\n1. PUSH\n";
                cout<<"2. POP\n";
                cout<<"3. DISPLAY\n";
                cout<<"\n\nEnter your choice = ";
                cin>>choice;
                switch(choice)
                {
                        case 1:
                                //calling the push function
                                Qo.push();
                                break;
                        case 2:
                                //calling the pop function by passing
                                //front and rear pointers
                                Qo.pop();
                                break;
                        case 3:
                                Qo.traverse();
                                break;
                }
                cout<<"\n\nPress (Y/y) to continue = ";
                cin>>option;
        }while(option == 'Y' || option == 'y');
}
```

5.9. QUEUE USING TWO STACKS

A queue can be implemented using two stacks. Suppose STACK1 and STACK2 are the two stacks. When an element is pushed on to the queue, push the same on STACK1. When an element is popped from the queue, pop all elements of STACK1 and push the same on STACK2. Then pop the topmost element of STACK2; which is the first (front) element to be popped from the queue. Then pop all elements of STACK2 and push the same on STACK1 for next operation (*i.e., push or pop*). PROGRAM 5:5 gives the program to implement the queue using two stacks by singly linked list, coded in C language.

PROGRAM 5.7

```
//IMPLEMENTING THE QUEUE USING TWO STACKS
//BY SINGLY LINK LIST
//CODED AND COMPILED USING TURBO C
```

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

//Stack node is created with structure
struct stack
{
        int info;
        struct stack *next;
};

typedef struct stack *NODE;

//A new element is pushed to the stack
NODE push(NODE top)
{
        NODE NewNode;
        //New node is created
        NewNode=(NODE)malloc(sizeof(struct stack));
        NewNode->next=top;

        printf("\nEnter the no: to be pushed = ");
        scanf("%d",&NewNode->info);

        top=NewNode;
        return(top);
}

NODE pop(NODE top1)
{
        //checking for whether the queue is empty or not
        if(top1 == NULL)
        {
                printf("\nThe Queu is empty");
                return(top1);
        }

        //when top1->next == NULL the queue contains only one element
        if(top1->next == NULL)
        {
                //popping the only one element present in the queue
                printf("\nThe popped element is = %d",top1->info);
```

```
            free(top1);
            top1=NULL;
            return(top1);
        }

        NODE NewNode,top2,TEMP;

        //popping the elements from the first stack and pushing the
        //same element to the second stack
        top2=NULL;
        while(top1 != NULL)
        {

            TEMP=top1;
            //Creating the new node for the second stack
            NewNode=(NODE)malloc(sizeof(struct stack));
            NewNode->next=top2;
            NewNode->info=top1->info;
            top2=NewNode;
            top1=top1->next;
            free(TEMP);
        };

        //popping the top most element from the stack so as to
        //pop an element from the queue
        printf("\nThe popped element is = %d",top2->info);
        top2=top2->next;

        //popping rest of the element from the second stack and
        //pushing the same elements to the first stack
        top1=NULL;
        while(top2 != NULL)
        {
            TEMP=top2;
            //creating new nodes for the first stack
            NewNode=(NODE)malloc(sizeof(struct stack));
            NewNode->next=top1;
            NewNode->info=top2->info;
            top1=NewNode;
            top2=top2->next;
            free(TEMP);//freeing the nodes
};
return(top1);
```

```
}

//this function is to display all the elements in the queue
void traverse(NODE top)
{
        if(top == NULL)
        {
                printf("\nThe Queue is empty");
                return;
        }

        printf ("\nThe element(s) in the Queue is/are =");
        do
        {
                printf (" %d",top->info);
                top=top->next;
        }while(top != NULL);

        return;
}

void main()
{
        int choice;
        char ch;
        NODE top;
        top=NULL;
        do
        {
                clrscr();
                //A menu for the stack operations
                printf("\n1. PUSH");
                printf("\n2. POP");
                printf("\n3. TRAVERSE");
                printf("\nEnter Your Choice = ");
                scanf ("%d", &choice);

                switch(choice)
                {
                        case 1://Calling push() function by passing
                                //the structure pointer to the function
                                top=push(top);
```

```
                    break;

                case 2://calling pop() function
                    top=pop(top);
                    break;

                case 3://calling traverse() function
                    traverse(top);
                    break;

                default:
                    printf("\nYou Entered Wrong Choice");
                    break;
            }

            printf("\n\nPress (Y/y) To Continue = ");
            //Removing all characters in the input buffer
            //for fresh input(s), especially <<Enter>> key
            fflush(stdin);
            scanf("%c",&ch);
        }while(ch == 'Y' || ch == 'y');
}
```

5.10. POLYNOMIALS USING LINKED LIST

Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. In this section, we discuss about polynomial addition using linked list. Consider two polynomials $f(x)$ and $g(x)$; it can be represented using linked list as follows in Fig. 5.22.

$$f(x) = ax^3 + bx + c$$

$$g(x) = mx^4 + nx^3 + ox^2 + px + q$$

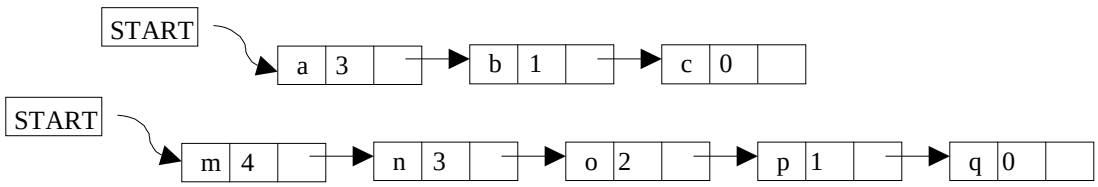


Fig. 5.22. Polynomial Representation

These two polynomials can be added by

$$h(x) = f(x) + g(x) = mx^4 + (a + n)\,x^3 + ox^2 + (b + p)x + (c + q)$$

i.e.; adding the constants of the corresponding polynomials of the same exponentials. $h(x)$ can be represented as in Fig. 5.23.

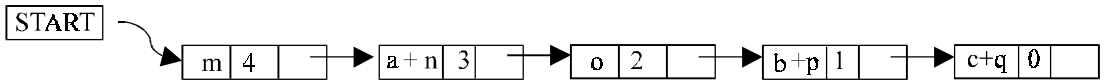


Fig. 5.23

PROGRAM 5.8

```
//Program of polynomial addition using linked list
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<malloc.h>

//structure is created for the node
struct node
{
        float coef;
        int expo;
        struct node *link;
};

typedef struct node *NODE;
//Function to add any node to the linked list
NODE insert(NODE start,float co,int ex)
{
        NODE ptr,tmp;
        //a new node is created
        tmp= (NODE)malloc(sizeof(struct node));
        tmp->coef=co;
        tmp->expo=ex;

        /*list empty or exp greater than first one */
        if(start==NULL || ex>start->expo)
        {
                tmp->link=start;//setting the start
                start=tmp;
        }
        else
```

```
        {
                ptr=start;
                while(ptr->link!=NULL && ptr->link->expo>ex)
                        ptr=ptr->link;
                tmp->link=ptr->link;
                ptr->link=tmp;
                if(ptr->link==NULL)  /*item to be added in the end */
                        tmp->link=NULL;
        }
        return start;
}/*End of insert()*/

//This function is to add two polynomials
NODE poly_add(NODE p1,NODE p2)
{
        NODE p3_start,p3,tmp;
        p3_start=NULL;
        if(p1==NULL && p2==NULL)
                return p3_start;

        while(p1!=NULL && p2!=NULL )
        {
                //New node is created
                tmp=(NODE)malloc(sizeof(struct node));
                if(p3_start==NULL)
                {
                        p3_start=tmp;
                        p3=p3_start;
                }
                else
                {
                        p3->link=tmp;
                        p3=p3->link;
                }
                if(p1->expo > p2->expo)
                {
                        tmp->coef=p1->coef;
                        tmp->expo=p1->expo;
                        p1=p1->link;
                }
                else
                        if(p2->expo > p1->expo)
```

```
                        {
                                tmp->coef=p2->coef;
                                tmp->expo=p2->expo;
                                p2=p2->link;
                        }
                        else
                                if(p1->expo == p2->expo)
                                {
                                        tmp->coef=p1->coef + p2->coef;
                                        tmp->expo=p1->expo;
                                        p1=p1->link;
                                        p2=p2->link;
                                }
}/*End of while*/
while(p1!=NULL)
{
                tmp=(NODE)malloc(sizeof(struct node));
                tmp->coef=p1->coef;
                tmp->expo=p1->expo;
                if (p3_start==NULL) /*poly 2 is empty*/
                {
                        p3_start=tmp;
                        p3=p3_start;
                }
                else
                {
                        p3->link=tmp;
                        p3=p3->link;
                }
                p1=p1->link;
        }/*End of while */
        while(p2!=NULL)
        {
                tmp=(NODE)malloc(sizeof(struct node));
                tmp->coef=p2->coef;
                tmp->expo=p2->expo;
                if (p3_start==NULL) /*poly 1 is empty*/
                {
                        p3_start=tmp;
                        p3=p3_start;
                }
                else
```

```
                {
                        p3->link=tmp;
                        p3=p3->link;
                }
                p2=p2->link;
        }/*End of while*/
        p3->link=NULL;
        return p3_start;
}/*End of poly_add() */

//Inputting the two polynomials
NODE enter(NODE start)
{
        int i,n,ex;
        float co;
        printf("\nHow many terms u want to enter:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("\nEnter coeficient for term %d:",i);
                scanf("%f",&co);
                printf("Enter exponent for term %d:",i);
                scanf("%d",&ex);
                start=insert(start,co,ex);
        }
        return start;
}/*End of enter()*/

//This function will display the two polynomials and its
//added polynomials
void display(NODE ptr)
{
        if (ptr==NULL)
        {
                printf ("\nEmpty\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf ("(%.1fx^%d) + ", ptr->coef,ptr->expo);
                ptr=ptr->link;
        }
```

```
            printf ("\b\b \n"); /* \b\b to erase the last + sign*/
}/*End of display()*/

void main()
{
            NODE p1_start,p2_start,p3_start;

            p1_start=NULL;
            p2_start=NULL;
            p3_start=NULL;

            printf("\nPolynomial 1 :\n");
            p1_start=enter(p1_start);

            printf("\nPolynomial 2 :\n");
            p2_start=enter(p2_start);
            //polynomial addition function is called
            p3_start=poly_add(p1_start,p2_start);

            clrscr();
            printf("\nPolynomial 1 is: ");
            display(p1_start);
            printf ("\nPolynomial 2 is: ");
            display(p2_start);
            printf ("\nAdded polynomial is: ");
            display(p3_start);
            getch();
}/*End of main()*/
```

5.11. DOUBLY LINKED LIST

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields: LeftPointer, RightPointer and DATA. Fig. 5.22 shows a typical doubly linked list.

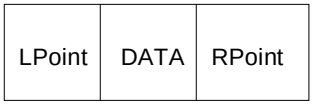


Fig. 5.24. A typical doubly linked list node

LPoint will point to the node in the left side (or previous node) that is LPoint will hold the address of the previous node. RPoint will point to the node in the right side (or next

node) that is RPoint will hold the address of the next node. DATA will store the information of the node.

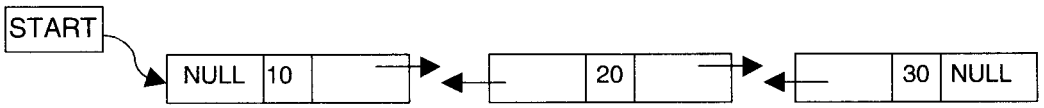


Fig. 5.25. Doubly Linked List

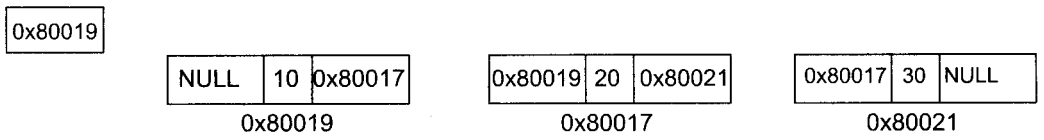


Fig. 5.26. Memory Representation of Doubly Linked List

5.11.1. REPRESENTATION OF DOUBLY LINKED LIST

A node in the doubly linked list can be represented in memory with the following declarations.

```
struct Node
{
    int DATA;
    struct Node *RChild;
    struct Node *LChild;
};
    typedef struct Node *NODE;
```

All the operations performed on singly linked list can also be performed on doubly linked list. Following figure will illustrate the insertion and deletion of nodes.

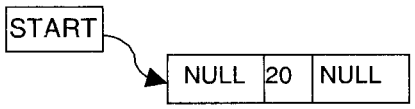


Fig. 5.27. Add(20)

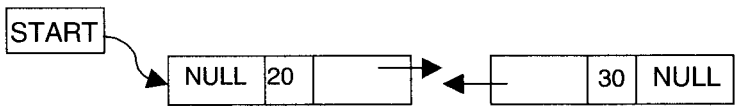


Fig 5.28. Insert (30) at the end

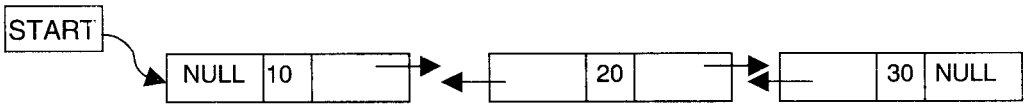


Fig 5.29. Insert (10) at the beginning

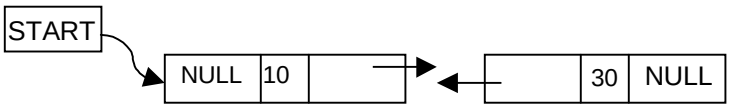


Fig 5.30. Delete a node at the 2nd position

5.11.2. ALGORITHM FOR INSERTING A NODE

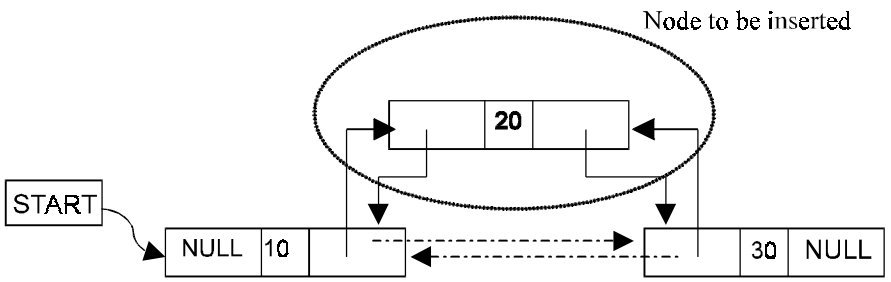


Fig. 5.31. Insert a node at the 2nd position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the NewNode is to be inserted. TEMP is a temporary pointer to hold the node address.

1. Input the DATA and POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → RPoint; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
   (*a*) Create a New Node
   (*b*) NewNode → DATA = DATA
   (*c*) NewNode → RPoint = TEMP → RPoint
   (*d*) NewNode → LPoint = TEMP
   (*e*) (TEMP → RPoint) → LPoint = NewNode
   (*f*) TEMP → RPoint = New Node
6. Else
   (*a*) Display “Position NOT found”
7. Exit

5.11.3. ALGORITHM FOR DELETING A NODE

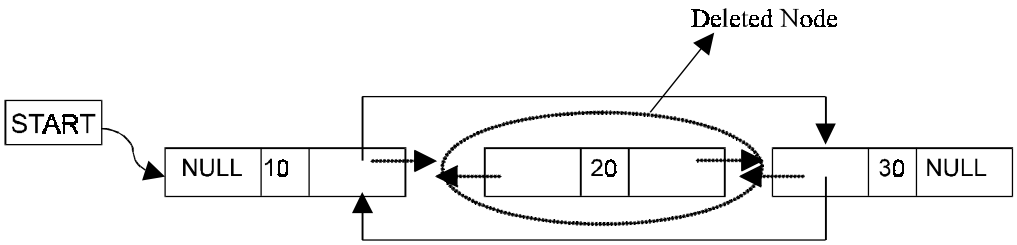


Fig. 5.32. Delete a node at the 2nd position

Suppose START is the address of the first node in the linked list. Let POS is the position of the node to be deleted. TEMP is the temporary pointer to hold the address of the node. After deletion, DATA will contain the information on the deleted node.

1. Input the POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → RPoint; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
   (*a*) Create a New Node
   (*b*) NewNode → DATA = DATA
   (*c*) NewNode → RPoint = TEMP → RPoint
   (*d*) NewNode → LPoint = TEMP
   (*e*) (TEMP → RPoint) → LPoint = NewNode
   (*f*) TEMP → RPoint = New Node
6. Else
   (*a*) Display "Position NOT found"
7. Exit

PROGRAM 5.9

```
// PROGRAM TO IMPLEMENT ALL THE OPERATIONS IN THE
//DOUBLY LINKED LIST
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
#include<malloc.h>
#include<process.h>
```

```
//Structure is created for the node
struct node
{
        struct node *prev;
        int info;
        struct node *next;
}*start;

typedef struct node *NODE;

//fucntion to create a doubly linked list
void create_list(int num)
{
        NODE q,tmp;
        //a new node is created
        tmp=(NODE)malloc(sizeof(struct node));
        tmp->info=num;//assigning the data to the new node
        tmp->next=NULL;
        if(start==NULL)
        {
                tmp->prev=NULL;
                start->prev=tmp;
                start=tmp;
        }
        else
        {
                q=start;
                while(q->next!=NULL)
                        q=q->next;
                q->next=tmp;
                tmp->prev=q;
        }
}/*End of create_list()*/

//Function to add new node at the beginning
void addatbeg(int num)
{
        NODE tmp;
        //a new node is created for inserting the data
        tmp=(NODE)malloc(sizeof(struct node));
        tmp->prev=NULL;
        tmp->info=num;
```

```
        tmp->next=start;
        start->prev=tmp;
        start=tmp;
}/*End of addatbeg()*/

//This fucntion will insert a node in any specific position
void addafter(int num,int pos)
{
        NODE tmp,q;
        int i;
        q=start;
        //Finding the position to be inserted
        for(i=0;i<pos-1;i++)
        {
                q=q->next;
                if(q==NULL)
                {
                        printf ("\nThere are less than %d elements\n",pos);
                        return;
                }
        }
        //a new node is created
        tmp=(NODE)malloc(sizeof(struct node) );
        tmp->info=num;
        q->next->prev=tmp;
        tmp->next=q->next;
        tmp->prev=q;
        q->next=tmp;
}/*End of addafter() */

//Function to delete a node
void del(int num)
{
        NODE tmp,q;
        if(start->info==num)
        {
                tmp=start;
                start=start->next;  /*first element deleted*/
                start->prev = NULL;
                free(tmp);//Freeing the deleted node
                return;
        }
```

```
        q=start;
        while(q->next->next!=NULL)
        {
            if(q->next->info==num)     /*Element deleted in between*/
            {
                tmp=q->next;
                q->next=tmp->next;
                tmp->next->prev=q;
                free(tmp);
                return;
            }
            q=q->next;
        }
        if (q->next->info==num)    /*last element deleted*/
        {   tmp=q->next;
            free(tmp);
            q->next=NULL;
            return;
        }
        printf("\nElement %d not found\n",num);
}/*End of del()*/

//Displaying all data(s) in the node
void display()
{
        NODE q;
        if(start==NULL)
        {
            printf("\nList is empty\n");
            return;
        }
        q=start;
        printf("\nList is :\n");
        while(q!=NULL)
        {
            printf("%d ", q->info);
            q=q->next;
        }
        printf("\n");
}/*End of display() */

//Function to count the number of nodes in the linked list
```

```
void count()
{
        NODE q=start;
        int cnt=0;
        while(q!=NULL)
        {
                q=q->next;
                cnt++;
        }
        printf("\nNumber of elements are %d\n",cnt);
}/*End of count()*/

//Reversing the linked list
void rev()
{
        NODE p1,p2;
        p1=start;
        p2=p1->next;
        p1->next=NULL;
        p1->prev=p2;
        while(p2!=NULL)
        {
                p2->prev=p2->next;
                p2->next=p1;
                p1=p2;
                p2=p2->prev; /*next of p2 changed to prev */
        }
        start=p1;
}/*End of rev()*/

void main()
{
        int choice,n,m,po,i;
        start=NULL;
        while(1)
        {
                //Menu options for the doubly linked list operation
                clrscr();
                printf("\n1.Create List\n");
                printf("2.Add at begining\n");
                printf("3.Add after\n");
                printf("4.Delete\n");
```

```
printf("5.Display\n");
printf("6.Count\n");
printf("7.Reverse\n");
printf("8.exit\n");
printf("\nEnter your choice:");
scanf("%d",&choice);
//switch instruction is called to execute
//correspoding function
switch(choice)
{
 case 1:
       printf("\nHow many nodes you want:");
       scanf("%d",&n);
       for(i=0;i<n;i++)
       {
             printf("\nEnter the element:");
             scanf("%d",&m);
             //create linked list function is called
             create_list(m);
       }
       break;

 case 2:
       printf("\nEnter the element:");
       scanf("%d",&m);
       addatbeg(m);
       break;
 case 3:
       printf("\nEnter the element:");
       scanf("%d",&m);
printf("\nEnter the position after which this element is inserted:");
       scanf("%d",&po);
       addafter(m,po);
       break;
 case 4:
       printf("\nEnter the element for deletion:");
       scanf("%d",&m);
       //Delete a node fucntion is called
       del(m);
       break;
 case 5:
       display();
       getch();
```

```
                    break;
                case 6:
                    count();
                    getch();
                    break;
                case 7:
                    rev();
                    break;
                case 8:
                    exit(0);
                default:
                    printf("\nWrong choice\n");
                    getch();
            }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

5.12. CIRCULAR LINKED LIST

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. A circular linked list is shown in Fig. 5.33. Implementation of circular linked list is in PROGRAM 5:8.

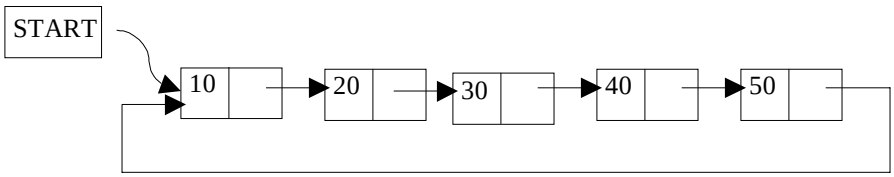


Fig. 5.33. Circular Linked list

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the Fig. 5.34. Implementation of circular doubly linked list is left to the readers.

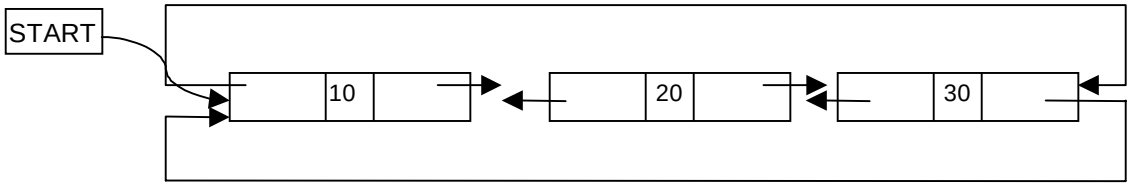


Fig. 5.34. Circular Doubly Linked list

PROGRAM 5.10

```
// PROGRAM TO IMPLEMENT CIRCULAR SINGLY LINKED LIST
//CODED AND COMPILED INTO TURBO C++
#include<iostream.h>
#include<process.h>

//class is created for the circular linked list
class Circular_Linked
{
        //structure node is created
        struct node
        {
            int info;
            struct node *link;
        };
        struct node *last;
        typedef struct node *NODE;

        public:
            //Constructor is defined
            Circular_Linked()
            {
                last=NULL;
            }

            void create_list(int);
            void addatbeg(int);
            void addafter(int,int);
            void del();
            void display();
};

//A circular list created in this function
void Circular_Linked::create_list(int num)
{
        NODE q,tmp;
        //New node is created
        tmp = (NODE)new(struct node);
        tmp->info = num;
```

```
        if (last == NULL)
        {
                last = tmp;
                tmp->link = last;
        }
        else
        {
                tmp->link = last->link; /*added at the end of list*/
                last->link = tmp;
                last = tmp;
        }
}/*End of create_list()*/

//This function will add new node at the beginning
void Circular_Linked::addatbeg(int num)
{
        NODE tmp;
        tmp = (NODE)new(struct node);
        tmp->info = num;
        tmp->link = last->link;
        last->link = tmp;
}/*End of addatbeg()*/

//Function to add new node at any position of the circular list
void Circular_Linked::addafter(int num,int pos)
{
        NODE tmp,q;
        int i;
        q = last->link;
        //finding the position to insert a new node
        for(i=0; i < pos-1; i++)
        {
                q = q->link;
                if (q == last->link)
                {
                        cout<<"There are less than "<<pos<<" elements\n";
                        return;
                }
        }/*End of for*/
        //creating the new node
        tmp = (NODE)new(struct node);
        tmp->link = q->link;
```

```
            tmp->info = num;
            q->link = tmp;
            if(q==last)    /*Element inserted at the end*/
                  last=tmp;
}/*End of addafter()*/

//Function to delete a node from the circular linked list
void Circular_Linked::del()
{
            int num;
            if(last == NULL)
            {
                  cout<<"\nList underflow\n";
                  return;
            }
            cout<<"\nEnter the number for deletion:";
            cin>>num;

            NODE tmp,q;
            if( last->link == last && last->info == num)  /*Only one element*/
            {
                  tmp = last;
                  last = NULL;
                  //deleting the node
                  delete(tmp);
                  return;
            }
            q = last->link;
            if(q->info == num)
            {
                  tmp = q;
                  last->link = q->link;
                  //deleting the node
                  delete(tmp);
                  return;
            }
            while(q->link != last)
            {
                  if(q->link->info == num)     /*Element deleted in between*/
                  {
                        tmp = q->link;
                        q->link = tmp->link;
```

```
                        delete(tmp);
                        cout<<"\n"<<num<<" deleted\n";
                        return;
                }
                q = q->link;
        }/*End of while*/
        if(q->link->info == num)    /*Last element deleted q->link=last*/
        {
                tmp = q->link;
                q->link = last->link;
                delete(tmp);
                last = q;
                return;
        }
        cout<<"\nElement "<<num<<" not found\n";
}/*End of del()*/

//Function to display all the nodes in the circular linked list
void Circular_Linked::display()
{
        NODE q;
        if(last == NULL)
        {
                cout<<"\nList is empty\n";
                return;
        }
        q = last->link;
        cout<<"\nList is:\n";
        while(q != last)
        {
                cout<< q->info;
                q = q->link;
        }
        cout<<"\n"<<last->info;
}/*End of display()*/

void main()
{
        int choice,n,m,po,i;
        Circular_Linked co;//Object is created for the class
        while(1)
        {
                //Menu options
```

```
cout<<"\n1.Create List\n";
cout<<"2.Add at begining\n";
cout<<"3.Add after \n";
cout<<"4.Delete\n";
cout<<"5.Display\n";
cout<<"6.Quit\n";
cout<<"\nEnter your choice:";
cin>>choice;

switch(choice)
{
 case 1:
       cout<<"\nHow many nodes you want:";
       cin>>n;
       for(i=0; i < n;i++)
       {
             cout<<"\nEnter the element:";
             cin>>m;
             co.create_list(m);
       }
       break;
 case 2:
       cout<<"\nEnter the element:";
       cin>>m;
       co.addatbeg(m);
       break;
 case 3:
       cout<<"\nEnter the element:";
       cin>>m;
       cout<<"\nEnter the position after which this element is inserted:";
       cin>>po;
       co.addafter(m,po);
       break;
 case 4:
       co.del();
       break;
 case 5:
       co.display();
       break;
 case 6:
       exit(0);
 default:
```

```
                    cout<<"\nWrong choice\n";
                }/*End of switch*/
            }/*End of while*/
}/*End of main()*/
```

5.13. PRIORITY QUEUES

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

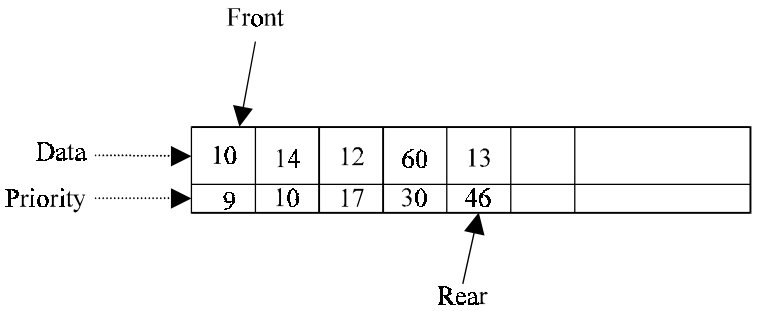


Fig. 5.35. Priority queue representation using arrays

Above Fig. 5.35 gives a pictorial representation of priority queue using arrays after adding 5 elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is O(n), which is much higher than the other queue (ie; other queues takes only O(1)) for inserting an element. So it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list - which is discussed in this section.

A node in the priority queue will contain DATA, PRIORITY and NEXT field. DATA field will store the actual information; PRIORITY field will store its corresponding priority of the DATA and NEXT will store the address of the next node. Fig. 5.36 shows the linked list representation of the node when a DATA (*i.e.*, 12) and PRIORITY (*i.e.*, 17) is inserted in a priority queue.

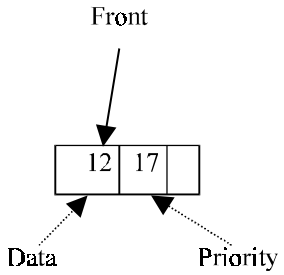


Fig. 5.36. Linked list representation of priority queue

When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end. Following figures will illustrate the push and pop operation of priority queue using linked list.

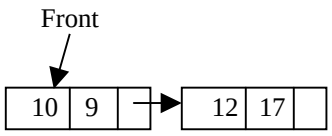


Fig. 5.37. push(DATA =10, PRIORITY = 9)

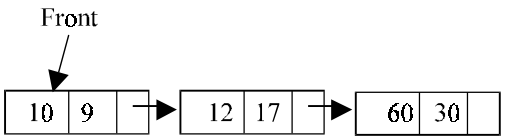


Fig. 5.38. push(DATA = 60, PRIORITY = 30)

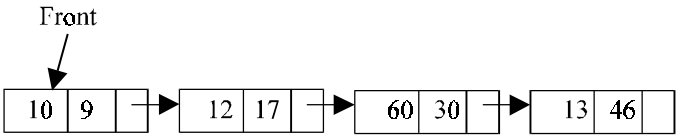


Fig. 5.39. push(DATA = 13, PRIORITY = 46)

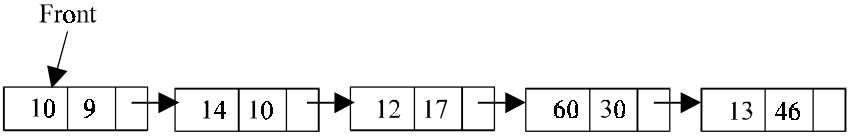


Fig. 5.40. push(DATA = 14, PRIORITY = 10)

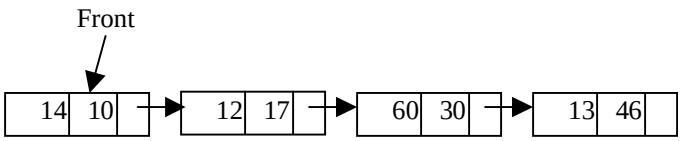


Fig. 5.41. *x* = pop() (*i.e.*, 10)

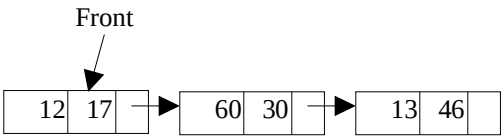


Fig. 5.42. *x* = pop() (*i.e.*, 14)

PROGRAM 5.11

```
//PROGRAM TO IMPLEMENT PRIORITY QUEUE USING LINKED LIST
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>
#include<malloc.h>
#include<process.h>

//A structure is created for a node
struct node
{
        int priority;
        int info;
        struct node *link;
};

typedef struct node *NODE;

//This function will insert a data and its priority
NODE insert(NODE front)
{
        NODE tmp,q;
        int added_item,item_priority;
        //New node is created
```

```
            tmp = (NODE)malloc(sizeof(struct node));
            printf("\nInput the item value to be added in the queue:");
            scanf("%d",&added_item);
            printf("\nEnter its priority:");
            scanf("%d",&item_priority);
            tmp->info = added_item;
            tmp->priority = item_priority;
            /*Queue is empty or item to be added has priority more than first item*/
            if(front == NULL || item_priority < front->priority)
            {
                  tmp->link = front;
                  front = tmp;
            }
            else
            {
                  q = front;
                  while(q->link != NULL && q->link->priority <= item_priority)
                        q=q->link;
                  tmp->link = q->link;
                  q->link = tmp;
            }/*End of else*/
            return(front);
}/*End of insert()*/

//Following function is to delete a node from the priority queue
NODE del(NODE front)
{
            NODE tmp;
            if(front == NULL)
                  printf("\nQueue Underflow\n");
            else
            {
                  tmp = front;
                  printf("\nDeleted item is %d\n",tmp->info);
                  front = front->link;
                  free(tmp);
            }
            return(front);
}/*End of del()*/

void display(NODE front)
{
```

```
        NODE ptr;
        ptr = front;
        if(front == NULL)
                printf("\nQueue is empty\n");
        else
        {       printf("\nQueue is:\n");
                printf("\nPriority Item\n");
                while(ptr != NULL)
                {
                        printf("%5d %5d\n",ptr->priority,ptr->info);
                        ptr = ptr->link;
                }
        }/*End of else */
}/*End of display() */


void main()
{
        int choice;
        NODE front=NULL;
        while(1)
        {
                clrscr();
                //Menu options
                printf("\n1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");
                printf("\nEnter your choic");
                scanf("%d", &choice);

                switch(choice)
                {
                 case 1:
                        front=insert(front);
                        break;
                 case 2:
                        front=del(front);
                        getch();
                        break;
                 case 3:
                        display(front);
```

```
                    getch();
                    break;
                case 4:
                    exit(1);
                default :
                    printf("\nWrong choice\n");
                }/*End of switch*/
            }/*End of while*/
}/*End of main()*/
```