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.

Program = Algorithm + Data Structure

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

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.

Representation of algorithm can written By:-

In natural language (English) / pseudo-code / diagrams (Flow chart) / etc.

Pseudo- code:-

A mixture of natural language and high – level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm**. Pseudo- code** is more structured than usual language but less formal than a programming language.

E.g.:- Algorithm arrayMax (A, n)

input: An array A sorting n integers s3a3zs4cvb ,m./

output: The Maximum element in A

currentMax ← A[0]

```
for i ← 1 to n-1 do
if currentMax < A[i] then currentMax ← A[i]
return currentMax
```

Ex: An algorithm to find sum n numbers between any given range numbers

1- Start
2- Read N
3- Sum =0
4- For I= 1 to N
    4.1 sum = sum + I
    4.1 next I
5- print Sum
6- End

What Makes a Good Algorithm?

Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?

- Faster
- Less space
- Easier to code
- Easier to maintain

Algorithm Analysis: The Big-O Notation

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- 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.

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.

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/2nd 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.

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

Example: Consider the following algorithm. (Assume that all variables are properly declared.)

```
cout << "Enter two numbers";                          //Line 1
cin >> num1 >> num2;                                  //Line 2
if (num1 >= num2)                                     //Line 3
        max = num1;                                   //Line 4
else                                                  //Line 5
        max = num2;                                   //Line 6
cout << "The maximum number is: " << max << endl;     //Line 7
```

Line 1 has one operation, <<; Line 2 has two operations; Line 3 has one operation, >=; Line 4 has one operation, =; Line 6 has one operation; and Line 7 has three operations. Either Line 4 or Line 6 executes. Therefore, the total number of operations executed in the preceding code is 1 + 2 + 1 + 1 + 3 = 8. In this algorithm, the number of operations executed is fixed.

Consider the following algorithm:

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1
count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3
cin >> num;                                                 //Line 4
while (num != -1)                                           //Line 5
{
```

```
        sum = sum + num;                                    //Line 6
        count++;                                            //Line 7
        cin >> num;                                         //Line 8
}
cout << "The sum of the numbers is: " << sum << endl;       //Line 9
if (count != 0)                                             //Line 10
        average = sum / count;                              //Line 11
else                                                        //Line 12
        average = 0;                                        //Line 13
cout << "The average is: " << average << endl;              //Line 14
```

This algorithm has five operations (Lines 1 through 4) before the **while** loop. Similarly, there are nine or eight operations after the **while** loop, depending on whether Line 11 or Line 13 executes.

Line 5 has one operation, and four operations within the **while** loop (Lines 6 through 8). Thus, Lines 5 through 8 have five operations. If the **while** loop executes 10 times, these five operations execute 10 times. One extra operation is also executed at Line 5 to terminate the loop. Therefore, the number of operations executed is 51 from Lines 5 through 8.

If the **while** loop executes 10 times, the total number of operations executed is:

10 * 5 + 1 + 5 + 9 or 10 * 5 + 1 + 5 + 8

that is,

10 * 5 + 15 or 10 * 5 + 14

We can generalize it to the case when the **while** loop executes n times. If the while loop executes n times, the number of operations executed is:

5n + 15 or 5n + 14

In these expressions, for very large values of n, the term 5n becomes the dominating term and the terms 15 and 14 become negligible.

Usually, in an algorithm, certain operations are dominant. For example, in the preceding algorithm, to add numbers, the dominant operation is in Line 6. Similarly,

in a search algorithm, because the search item is compared with the items in the list, the dominant operations would be comparison, that is, the relational operation. Therefore, in the case of a search algorithm, we count the number of comparisons. For another example, suppose that we write a program to multiply matrices. The multiplication of matrices involves addition and multiplication. Because multiplication takes more computer time to execute, to analyze a matrix multiplication algorithm, we count the number of multiplications. In addition to developing algorithms, we also provide a reasonable analysis of each algorithm. If there are various algorithms to accomplish a particular task, the algorithm analysis allows the programmer to choose between various options.

Consider the following code, where m and n are int variables and their values are nonnegative:

```
for (int i = 0; i < m; i++)              //Line 1
    for (int j = 0; j < n; j++)          //Line 2
        cout << i * j << endl;           //Line 3
```

This code contains nested for loops. The outer for loop, at Line 1, executes m times. For each iteration of the outer loop, the inner loop, at Line 2, executes n times. For each iteration of the inner loop, the output statement in Line 3 executes. It follows that the total number of iterations of the nested for loop is mn. So the number of times the statement in Line 3 executes is mn. Therefore, this algorithm is O(mn). Note that if m = n, then this algorithm is $O(n^2)$.

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.

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.

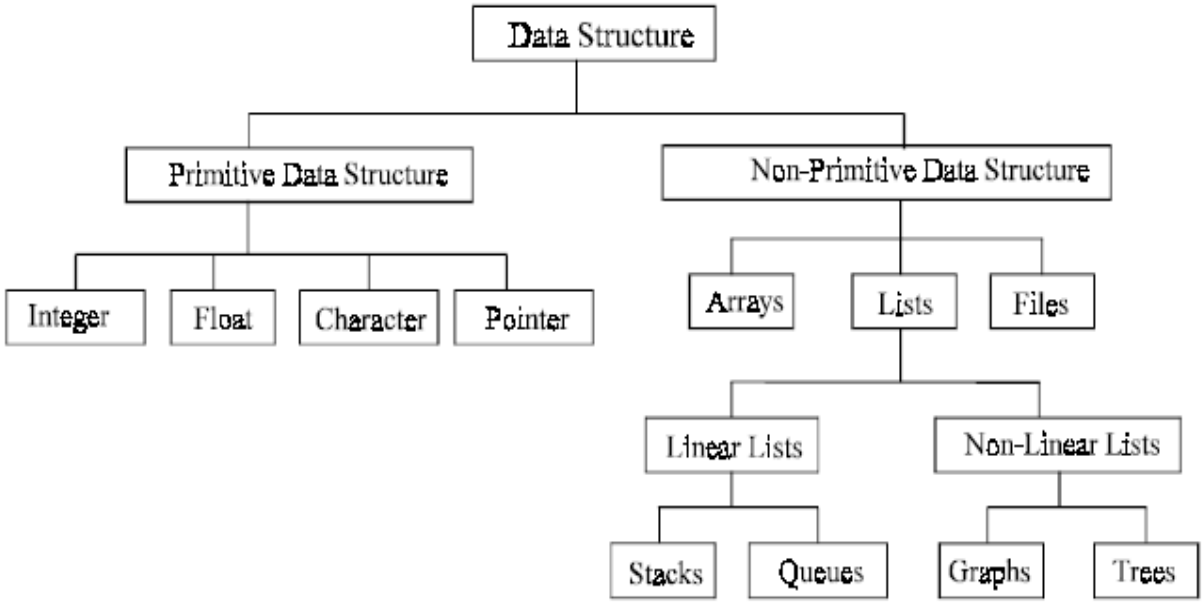


Fig.1 Classifications of data structures

The Fig.1 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.

How to choose the suitable data structure:-

For each set of data, there are different methods to organize these data in a particular data structure. To choose the suitable data structure, we must use the following criteria:-

1- Data size and the required memory.

2- The dynamic nature of the data.

3- The required time to obtain any data element from the data structure.

4- The programming approach and the algorithm that will be used to manipulate these data.

Assignment -1-

- **Write an algorithm for the following**

  a- sum of n numbers between given range

  b- sum of n numbers between given range

  c- count even& odd numbers in given range

  d- count even & odd numbers in given range

  e- count even & odd numbers in given range

****

- **Each of the following expressions represents the number of operations for** certain algorithms. What is the order of each of these expressions?

  a. n2 + 6n + 4

  b. 5n3 + 2n + 8

  c. (n2 + 1) (3n + 5)

  d. 5(6n + 4)

****

- **Consider the following function:**

```
int funcExercise(int list[], int size)
{
        int sum = 0;
        for (int index = 0; index < size; index++)
                sum = sum + list[index];
        return sum;
}
```

a. Find the number of operations executed by the function **funcExercise** if the value of size is 10.

b. Find the number of operations executed by the function **funcExercise** if the value of size is n.

c. What is the order (o(n)) of the function f**uncExercise**?

****

- Characterize the following algorithm in terms of Big-O notation. Also find the exact number of additions, subtractions, and multiplications executed by the loop. (Assume that all variables are properly declared.)

```
for (int i = 5; i <= 2 * n; i++)
        cout << 2 * n + i - 1 << endl;
```

****

- Characterize the following algorithm in terms of Big-O notation.

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        for (int k = 1; k <= n; k++)
            cout << i + j + k;
```

Array

It's a set of storage location in the memory, use and classify as:-

1- All locations are same data type, according to the definition (real, integer, char, …).

2- Can random access to any location without depending on any location in Array, the requirement time to access for any location is constant.

3- The element's location of Array are still steady, don't change when dealing with any elements of Array.

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.

One-Dimensional Array

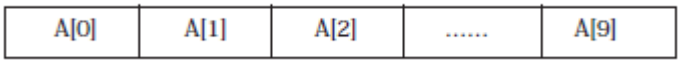
One-dimensional array (or linear array) is a set of 'n' finite numbers of homogenous data elements such as:

1. The elements of the array are referenced respectively by an index set consistin 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 / 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 :



In C/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 in C++

```
for (i = 0; i < n; i ++)
      cin >> A[i];
For writing an array
for (i = 0; i < n; i ++)
      cout << A[i];
```

Representation of One-Dimensional Array

In C and C++ language we can defined array as

Int ArrayName [n];

that means n's elements of integer type or we can choose any type of variables . That's means the structure contains a set of data elements, numbered n, for example ArrayName , its defined as type of element, the second type is the index type, is the type of values used to access individual element of the array, the value of index is

0 <= i < n

By this definition the compiler limits the storage region to storing set of element, and the first location is individual element of array, and this called the Base Address

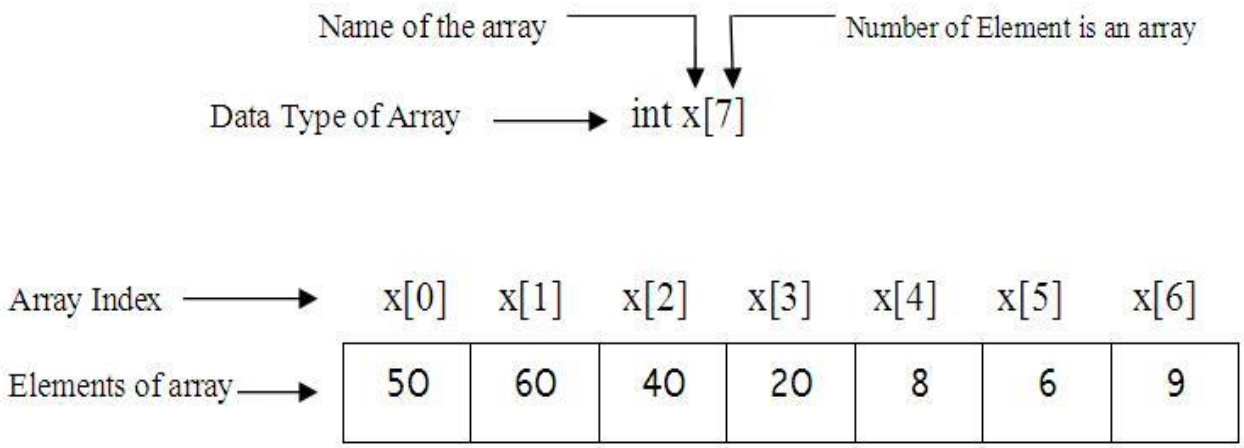


Fig. One dimensional array

, let's be as 500, the second address of array, is after Base Address (501) and like for the all elements and used the index i, by its value are range 0<= i < n according to Base Index (500), by using this relation: (for simplicity we assumed that array's name is X)

4	500	X[0]
6	501	X[1]
2	502	X[2]
8	503	X[3]
1	504	X[4]

To determine element address:

Location (X [i]) = Base Address + i * size of the element

When the requirement to bounding the fifth element (i=4):

Location (X[4]) = 500 + 4 * 1

= 500 + 4 = 504

So the address of fifth element is 504 because the first element in 500. When the program indicate or dealing with element of array in any instruction like (cin >> x), (cout << x), the compiler depend on going relation to bounding the requirement address .

Two 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}

For reading a two Dimensional array of 'm * n' elements in C++

```
for(int i = 0 ; i < m ; ++i)
        {
```

```
        for(int j = 0 ; j < n ; j ++)
        {
          cin >> A[i][j];
        }

    }
```

For writing a two Dimensional array of 'm * n' elements in C++

```
for(int i = 0 ; i < m ; ++i)
    {
        for(int j = 0 ; j < n ; j ++)
        {
          cout<< A[i][j];
        }

    }
```

Representation of Two Dimensional Array in memory

There are two method to represent two dimensional array as follows:

1- Row-Wise Method

2- Column-Wise Method

The definition of two dimensional array is: Int A[m][n];

That's mean structure name is A, consist of set of data elements as (m * n), and used two argument for access to the require element as:

0 <= i < m to bounding row element.

0 <= j < n to bounding column element.

For example A (3,5), where i=3, j=5, mean that the element located in third row and fifth column.

The compiler depend on one of two method to represent array:

1- Row-Wise Method (Row-Major Order)

In this method toke all element of first row (i=0) of array and store in memory from Base Address, lets 700:

700	BA	store in A[0][0]
701	BA+1	store in A[0][1]
702	BA+2	store in A[0][2]

Then token all elements of second row i=1 of array and store in memory, from the address that's after last address of first row. Then token all elements of third row i=2 of array and store in memory, from the address that's after last address of second row.

Ex: A=

$$\begin{bmatrix} 1 & 2 & 3 & 7 \\ 5 & 4 & 6 & 8 \\ 12 & 11 & 10 & 9 \end{bmatrix}$$

500	**1**	**A[0][0]**
501	**2**	**A[0][1]**
502	**3**	**A[0][2]**
503	**7**	**A[0][3]**
504	**5**	**A[1][0]**
505	**4**	**A[1][1]**
506	**6**	**A[1][2]**
507	**8**	**A[1][3]**
508	**12**	**A[2][0]**
509	**11**	**A[2][1]**
510	**10**	**A[2][2]**
511	**9**	**A[2][3]**

To determine element address A[i][j]:

Loc (A [i][j]) = Base Address + (i*n+j) * size of element

Ex: consider the matrix above

Loc(A[1][0]) = 500 + (1 * 4+ 0) * 1 = 504 = 5

Loc(A[1][1]) = 500 + (1 * 4 + 1) * 1 = 505 = 4

Loc(A[2][2]) = 500 + (2 * 4 + 2) * 1 = 510 = 10

Loc(A[1][3]) = 500 + (1 * 4 + 3) * 1 = 507 = 8

2- Column-Wise Method (Column-Major Order)

In this method toke all element of first column (j=0) of array and store in memory from Base Address, lets 200:

200 BA store in A[0][0]

201 BA+1 store in A[1][0]

202 BA+2 store in A[2][0]

Then token all elements of second column J=1 of array and store in memory, from the address that's after last address of first column. Then token all elements of third column J=2 of array and store in memory, from the address that's after last address of second column.

Ex: A=

$$\begin{bmatrix} 1 & 2 & 3 & 7 \\ 5 & 4 & 6 & 8 \\ 12 & 11 & 10 & 9 \end{bmatrix}$$

500	**1**	**A[0][0]**
501	**5**	**A[1][0]**
502	**12**	**A[2][0]**
503	**2**	**A[0][1]**
504	**4**	**A[1][1]**
505	**11**	**A[2][1]**
506	**3**	**A[0][2]**
507	**6**	**A[1][2]**
508	**10**	**A[2][2]**
509	**7**	**A[0][3]**
510	**8**	**A[1][3]**
511	**9**	**A[2][3]**

To determine element address A[i][j]:

Loc (A [i][j]) = Real Address + (j*m+i) * size of element

Ex: consider the matrix above

Loc(A[1][0]) = 500 + (0 * 3+ 1) * 1 = 501 = 5

Loc(A[1][1]) = 500 + (1 * 3 + 1) * 1 = 504 = 4

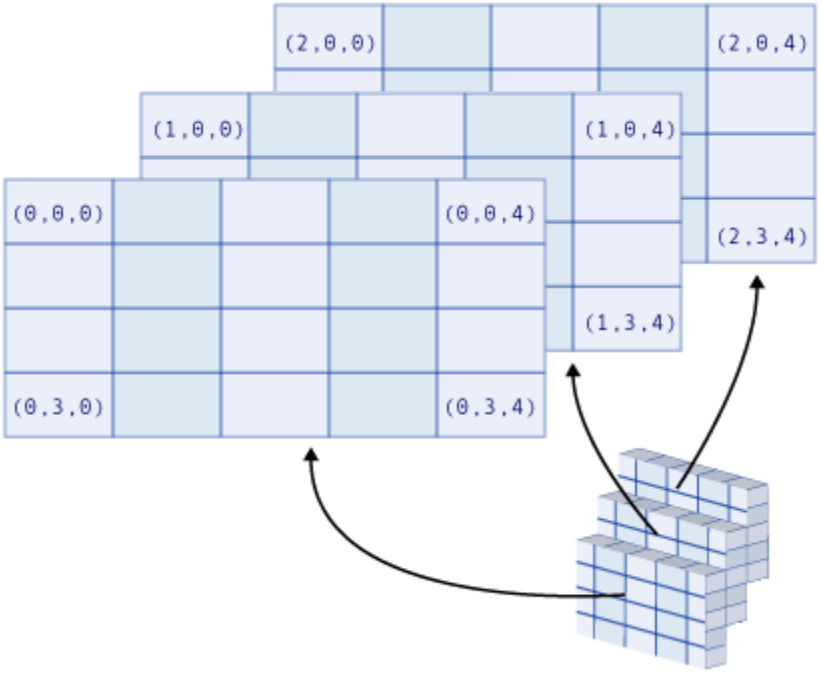
Loc(A[2][2]) = 500 + (2 * 3 + 2) * 1 = 508 = 10

Loc(A[1][3]) = 500 + (3 * 3 + 1) * 1 = 510 = 8

Three dimensional array

The declaration form of Three-dimensional array is

Data_type Array_name [level][m][n];



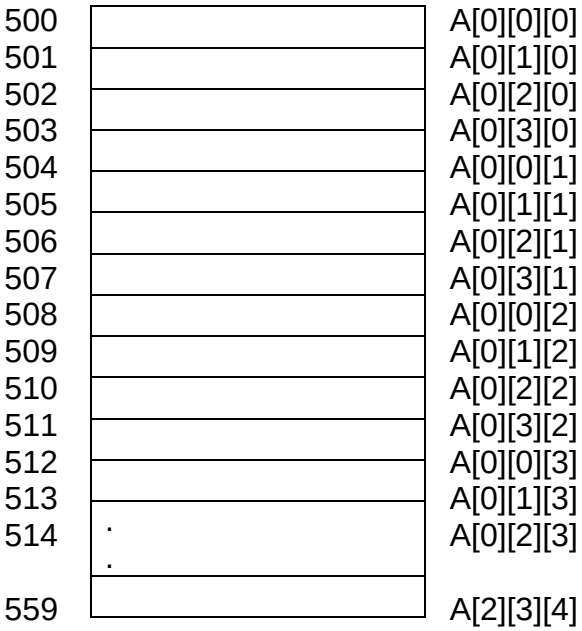
Ex:

int A[3][4][5];

Row-Wise Method

500		A[0][0][0]
501		A[0][0][1]
502		A[0][0][2]
503		A[0][0][3]
504		A[0][0][4]
505		A[0][1][0]
506		A[0][1][1]
507		A[0][1][2]
508		A[0][1][3]
509		A[0][1][4]
510		A[0][2][0]
511		A[0][2][1]
512		A[0][2][2]
513		A[0][2][3]
514	. .	A[0][2][4]
559		A[2][3][4]

Column wise method




```
//for reading in C++
        for(int i = 0 ; i < 2 ; ++i)                    // for levels
        {
                for(int j = 0 ; j < 3 ; j ++)            // for rows
                {
                  for(int k = 0; k < 4; k++)             // for columns
                  {
                     cin>>A[i][j][k];
                  }
                }

        }


//for writing in C++
        for(int i = 0 ; i < 2 ; ++i)
        {
                for(int j = 0 ; j < 3 ; j ++)
                {
                  for(int k = 0; k < 4; k++)
                  {
                     cout << A[i][j][k]<<endl;
                  }
                }

        }
```

Linear List

It is set of data elements (**item, nodes, elements**) are sequential, and connect each items contiguity relation, where each item are foregone another item (except the first item don't come any item before it, and the last item don't come any item after it). It differs from the stack and queue data structures in that additions and removals can be made at any position in the list. The order is important here (in list); this order is either due to sorting, the order may be due to the importance of the data items. If each item as a node, so the list is set of nodes

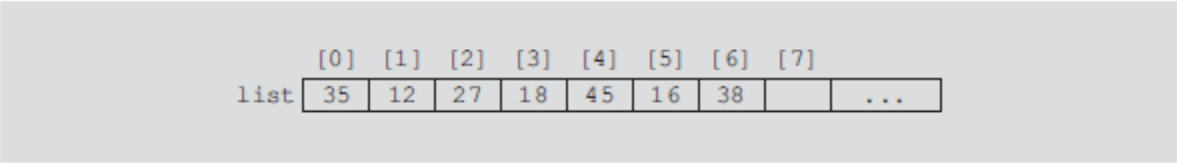
X[0]. X[1].X[2].....X[k-1]. X[k]. X[k+1] .....X[n-1]

Where first node is X[0], last node X[n-1].Each set of data and information can called **List**.

Types of Linear Lists

1-Array-Based Lists (Static List)

It's the lists don't used pointers, and be as data sequential, used array for representation, and use this type to processing data that's don't change dearly, because difficult the operations of delete and add, because of the locations of memory may be busy, so can't use easily for adding or deleting.



List Implementation

Now we will see what the implementation of the list is and how one can create a list. Suppose we want to create a list of integers. For this purpose, the methods of the list can be implemented with the use of an array inside. For example, the list of integers A= (2, 6, 8, 7, 1) can be represented in the following manner

A	2	6	8	7	1								
index	0	1	2	3	4								

The Operations on the Linear Lists

Can implement some of operations to any data structure, when execute its data, and the following types of operations that can be implementing some or all according to the applications:

1- Insert

To add new node (element) to data structure (list). The add method is used for adding an element to the list. Suppose we want to add a new element in the list i.e. add (9) .To add (9) to the list at the current position, at first, we have to make space for this element. For this purpose, we shift every element on the right of 8 (the current position) to one place on the right. Thus after creating the space for new element at position 4, the array can be represented as

A	2	6	8		7	1							
index	0	1	2	3	4	5							

Current = 4 (index = 3)

Now in the second step, we put the element 9 at the empty space i.e. position 4.

Thus the array will attain the following shape.

A	2	6	8	9	7	1							
index	0	1	2	3	4	5							

Current = 4 (index = 3)

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
int array[100], position, i, n, value;
cout <<"Enter number of elements in array\n";
cin>>n;
cout<<"Enter " << n <<" elements\n";
for(i = 0 ; i < n ; i ++)
    {
           cin >>array[i];
    }

cout <<"Enter the location where you wish to insert an element\n";
cin >> position;
cout <<"Enter the value to insert\n";
cin >> value;
for(i = n - 1; i >= position ; i --)
   {
         array[i + 1] = array[i];
```

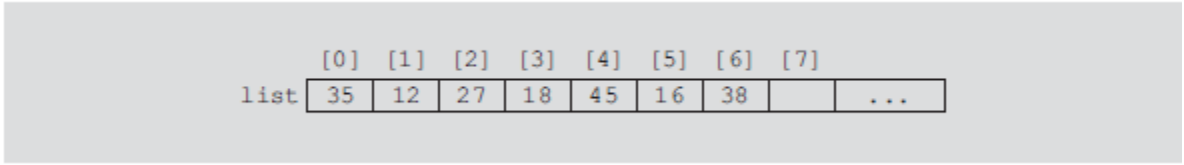
```
   }
array[position ] = value;
cout <<"Resultant array is\n";
for(i = 0 ; i <= n ; i ++)
  {
        cout <<array[i] <<"  ";
  }
getch();
return 0;
}
```

Worst Case: To add an element at the beginning of the list is the worst case of add method.

Best Case: We have to shift no element if we add at the end.

2- Search

Its operation of search inside data structure, its mean the access to element, according to the value of some fields, and called Key Field, that's means the searching is by the contents and not the address. Consider the list of seven elements shown in Figure below



Suppose that you want to determine whether 27 is in the list. The sequential search works as follows: First, you compare 27 with list[0]—that is, compare 27 with 35. Because list[0] ≠ 27, you then compare 27 with list[1] (that is, with 12, the second item in the list). Because list[1] ≠ 27, you compare 27 with the next element in the list—that is, compare 27 with list[2]. Because list[2] = 27, the search stops. This is a successful search.

Let us now search for 10. As before, the search starts with the first element in the list—that is, at list[0]. This time the search item, which is 10, is compared with every item in the list. Eventually, no more data is left in the list to compare with the search item. This is an unsuccessful search. It now follows that, as soon as you find an element in the list that is equal to the search item, you must stop the search and report ‘‘success.’’ (In this case, you usually also tell the location in the list where the search item was found.) Otherwise, after the search item is compared with every element in the list, you must stop the search and report ‘‘failure.’’

Worst Case: The worst case of the find method is that it has to search the entire list from beginning to end. Average Case: On average the find method searches at most half the list.

```
#include <iostream>
using namespace std;
const int N = 10;
```

```
int main ()
{
    int t[N], i=0, V;
    for (i = 0; i < N; i++)
    {
        cout << "Type an integer: ";
        cin >> t[i];
    }
    cout << "Type the value of V: ";
    cin >> V;
    for (i = 0; i < N; i++)
    {
        if (t[i] == V)
        {
          cout << "V is in the array" << endl;
          return 0;
        }
    }
    cout << "V is not in the array" << endl;
     system("pause");
    return 0;
}
```

3- Deletion

It's delete node from data structure (list).

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
int array[100], position, i, n, value;
cout <<"Enter number of elements in array\n";
cin>>n;
cout<<"Enter " << n <<" elements\n";
for(i = 0 ; i < n ; i ++)
    {
           cin >>array[i];
    }

cout <<"Enter the location where from you wish to deletet an element\n";
cin >> position;
//cout <<"Enter the value to dekete\n";
//cin >> value;
for(i = position; i < n ; i ++)
   {
         array[i] = array[i + 1] ;
   }
```

```
//array[position ] = value;
cout <<"Resultant array is\n";
for(i = 0 ; i < n-1 ; i ++)
  {
        cout <<array[i] <<"  ";
  }
getch();
return 0;
}
```

Worst Case: To remove an element at the beginning of the list is the worst case of remove method. Average Case: In average cases of the remove method we expect to shift half of the elements. Best Case: We have to shift no element if we remove at the end.

4- Counting

It's counting some of items or nodes in data structure.

5- Copying

Its copy data of data structure to another data structure.

6- Sorting

Its sort items or nodes in data structure according to the value of the field or set of fields.

Assignment -2-

1- Write a C++ program to count the elements of a list
2- Write a C++ program to get (count) number of elements in a list.
3- C++ Program to Sort Elements of Array in Ascending Order

2- Linked list (dynamic 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.

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 1 shows a typical example of node.

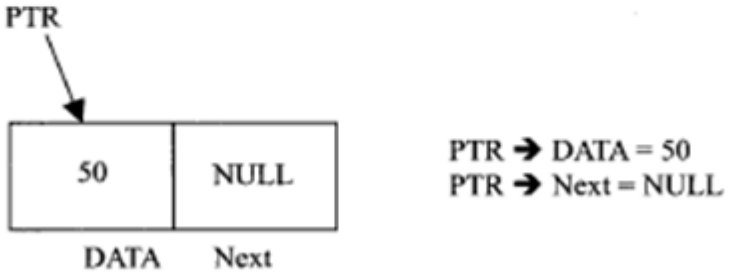


Fig 1. Node

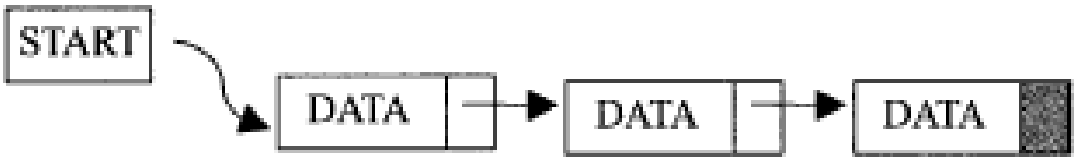


Fig 2. Linked list

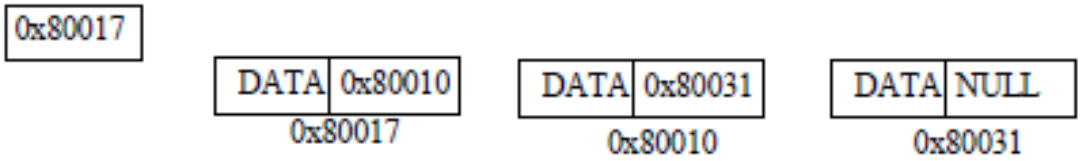


Fig 3. Linked list representation in memory

Fig. 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).

Representation of linked list

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

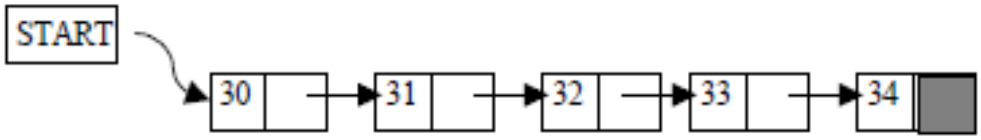


Fig. 4 Linked list representation of integers

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'
}Node *NODE;
```

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.

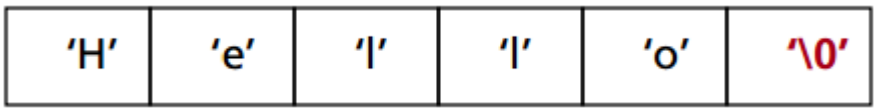
Operation on linked list

The primitive operations performed on the linked list are as follows

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

Strings

The most common use for one-dimensional arrays is to store strings of characters. In C++, a string is defined as a character array terminated by a null symbol ('\0').



To declare an array str that could hold a 10-character string, one would write:

char str[11];

Specifying the size as 11 makes room for the null at the end of the string.

Some examples of string constants in C++ are:

"hello there"

"I like C++."

"#$%§@@+*"

" "

The null string, "", only contains the null terminator and represents the empty string.

Reading a String from the Keyboard

How to read a string entered from the keyboard?

Make an array that will receive the string, the target of a cin stream. The following program reads (part of) a string entered by the user:

```
#include<stdio.h>
int main() {
   char str[80];
   cout << "Enter a string: ";
   cin >> str; // read string from keyboard
   cout << "Here is your string: ";
   cout << str;
   return(0);}
```

Problem: Entering the string "This is a test", the above program only returns "This", not the entire sentence.

Reason: The C++ input/output system stops reading a string when the first whitespace character is encountered.

Solution: Use another C++ library function, gets().

```
#include<iostream.h>
 #include<cstdio.h>
 int main() {
char str[80]; // long enough for user input?
cout << "Enter a string: ";
 gets(str); // read a string from the keyboard
cout << "Here is your string: ";
 cout << str << endl;
 return(0);}
```

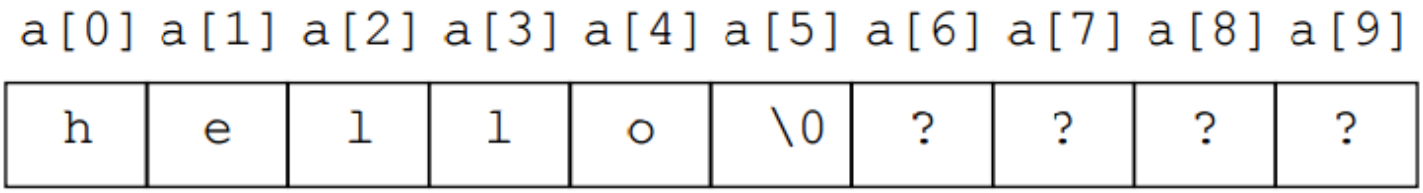
Some C++ Library Functions for Strings

The most common are:

- strcpy() : copy characters from one string to another
- strcat() : concatenation of strings
- strlen() : length of a string
- strcmp() : comparison of strings

strcpy(to_string, from_string) — String Copy:

```
#include<iostream.h>
 #include<cstring.h>
 int main() {
char a[10];
 strcpy(a, “hello”);
cout << a;      //or for(int i=0; a[i]!='\0';i++)  cout<<a[i];
return(0); }
```



strlen(string) — String Length

strlen(str) returns the length of the string pointed to by str, i.e., the number of characters excluding the null terminator.

```
#include<iostream>
 #include<cstdio.h>
 #include<cstring>
 int main() {
 char str[80];
 cout << “Enter a string: “;
 gets(str);
 cout << “Length is: “ << strlen(str);
 return(0);}
```

strcat(string_1, string_2) — Concatenation of Strings:

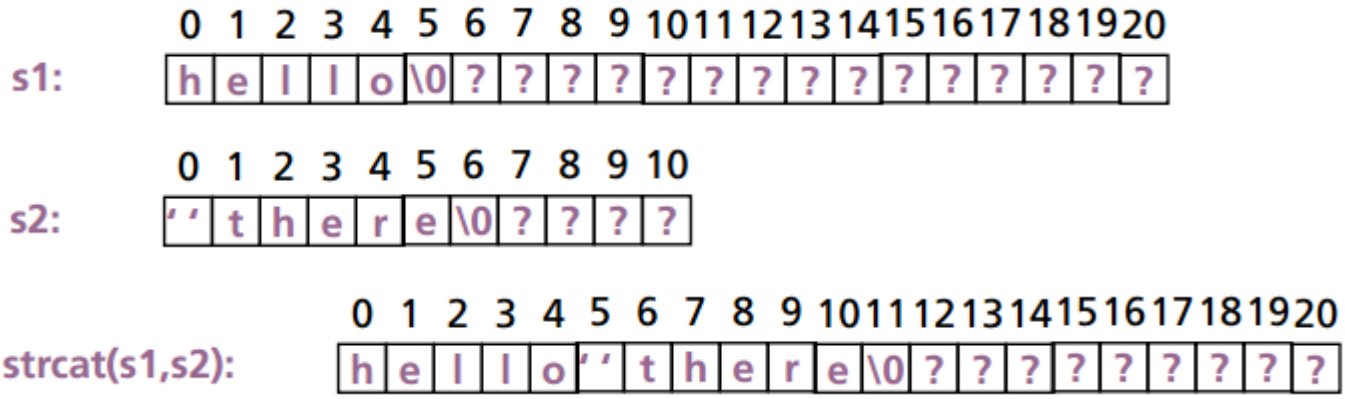
The strcat() function appends s2 to the end of s1. String s2 is unchanged.

```
// includes ...

 int main() {
char s1[21], s2[11];
```

```
strcpy(s1, “hello”);
strcpy(s2, “ there”);
strcat(s1, s2);
cout << s1 << endl;
cout << s2 << endl;
return(0); }
```

Note: The first string array has to be large enough to hold both strings:



• To be on the safe side:
strlen(s1concats2) >= strlen(s1) + strlen(s2)

strcmp(string_1, string_2) — Comparison of Strings

The strcmp(str_1, str_2) function compares two strings and returns the following result:
• str_1 == str_2 : 0
• str_1 > str_2 : positive number
• str_1 < str_2 : negative number

The strings are compared lexicographically (i.e., according to dictionary order):
a < aa < aaa < … < b < ba < bb < … < bz < baa < … < abca < abd < ...

```
// Comparing strings
 #include<iostream>
 #include<cstring>
#include
int main() {
 char str[80];
cout << “Enter password: “;
 gets(str);
 if(strcmp(str, “password”)) { // strings differ
cout << “Invalid password.\n”;}
 else cout << “Logged on.\n”;
 return(0); }
```

Using the Null Terminator:

Operations on strings can be simplified using the fact that all strings are null-terminated.

```
// Convert a string to uppercase
// ... includes ...
int main() {
 char str[80];
 int i;
 strcpy(str, “this is a test”);
 for(i=0; str[i]; i++)
str[i] = toupper(str[i]);
cout << str;
return(0); }
```

Character Array Initialization:

Character arrays that will hold strings allow a shorthand initialization that takes this form:

char array-name[size] = “string”;

For example, the following code fragment initializes str to the phrase “hello”:

char str[6] = “hello”;

This is the same as writing

char str[6] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

Remember that one has to make sure to make the array long enough to include the null terminator.

Unsized Array Initializations:

It is possible to let C++ automatically dimension the arrays through the use of unsized arrays.

```
char error_1[] = “Divide by 0\n”;
char error_2[] = “End-of-File\n”;
char error_3[] = “Access Denied\n”;
```

C++ will automatically create arrays large enough to hold all the initializers present.

For a multi-dimensional array, all but the leftmost dimension have to be specified. So we can write:

```
char errors[][20] = {
“Divide by 0\n”,
 “End-of-File\n”,
“Access Denied\n” };
```

Arrays of Strings:

An array of strings is a special form of a two-dimensional array.

- The size of the left index determines the number of strings.
- The size of the right index specifies the maximum length of each string.

For example, the following declares an array of 30 strings, each having a maximum length of 80 characters (with one extra character for the null terminator):

char string_array[30][81];

For accessing an individual string, one simply specifies only the left index:

firstString = string_array[0];

sixthString = string_array[5];
The following example calls the gets() function with the third string in the array:
gets(string_array[2]);

This program accepts lines of text entered at the keyboard and redisplays them after a blank line is entered.

```
 // includes go here
 int main() {
 int t, i;
 char text[100][80];
 for(t=0; t<100; t++ ){
   cout<<t<<": ";
  gets(text[t]);
 if(!text[t][0]) break; }  // quit on blank line
for(i=0; i<t;i++)
cout<< text[i] << "\n";
 return(0);}
```

An Example Using String Arrays:
Arrays of strings are commonly used for handling tables of information. One such application would be an employee database that stores

- the name
- telephone number
- hours worked per pay period, and
- hourly wage.

These data we could store in arrays:

```
char name[20][80];      // employee names
 int phone[20];           // phone numbers
 float hours[20];         // hours worked
 float wage[20];         // wage
```

Entering Information:

```
void enter() {
for(int i=0; i<20;i++){
cout<< "Last name: ";
cin >> name[i];
 cout << "Phone number: ";
 cin >> phone[i];
 cout << "Hours worked: ";
 cin >> hours[i];
 cout << "Wage: ";
 cin >> wage[i]; } }
```

Displaying Database Contents:

```
void report() {
for(int i=0; i < 20; i++) {
cout << "Name: " << name[i] << " / " << "phone: " << phone[i] << “\n”;
cout << “Pay for the week: “;
cout << wage[i] * hours[i];
 cout << “\n”; } }
```

Menu For User´s Selection:

```
int menu() {
int choice;
 cout << “0. Quit\n”;
 cout << “1. Enter information\n”;
cout << “2. Report information\n”;
 cout << “\n”;
cout << “Choose one: “;
 cin >> choice;
 return choice; }
```

Main Function:

```
int main() {
 int choice;
 do { choice = menu();       // get selection
 switch(choice) { case 0: break;
 case 1: enter(); break;
case 2: report(); break;
 default: cout << “Try again.\n\n”; }
 } while( choice != 0);
 return(0); }
```

Putting It All Together: (HW)

```
#include // array declarations
 int menu();
void enter();
 void report();
 int main() { ... }
int menu() { ... }
void enter() { ... }
 void report() { ... }
```

LINKED LIST DATA STRUCTURE

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.

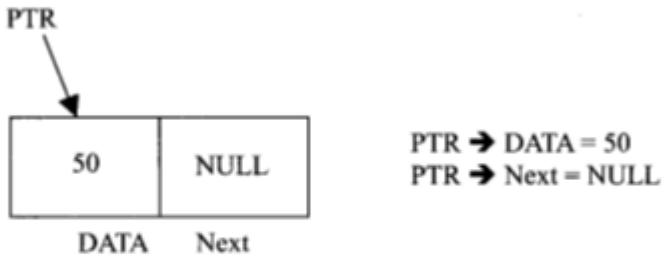


Fig. 5.1. Nodes.

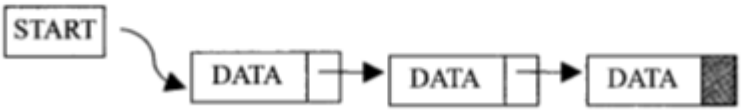


Fig. 5.2. Linked List.

Fig.5.2.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).

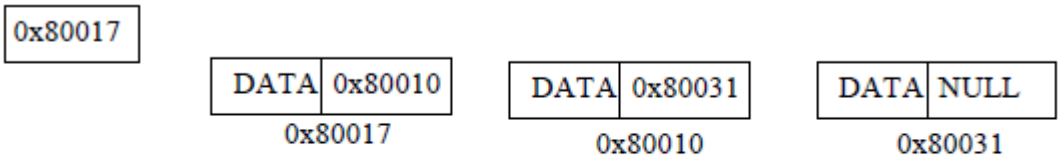


Fig. Linked List representation in memory.

Explanation:

Because each node of a linked list has two components, we need to declare each node as a class or struct. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The

data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodeType
    {
      int info;
      nodeType *link;
     };
```

The variable declaration is as follows:

nodeType *head;

Linked List: Some Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 5-4.

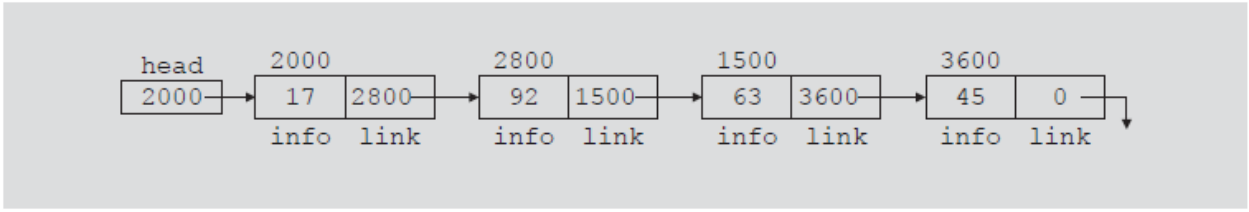


FIGURE 5-4 Linked list with four nodes

This linked list has four nodes. The address of the first node is stored in the pointer head. Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int.

Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table Table 5-1 shows the values of head and some other nodes in the list shown in Figure 5-4.

TABLE 5-1 Values of head and some of the nodes of the linked list in Figure 5-4

	Value	Explanation
head	2000	
head->info	17	Because head is 2000 and the info of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because head->link is 2800 and the info of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

current = head;

copies the value of head into current. Now consider the following statement:

current = current->link;

This statement copies the value of current->link, which is 2800, into current. Therefore, after this statement executes, current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 5-5.

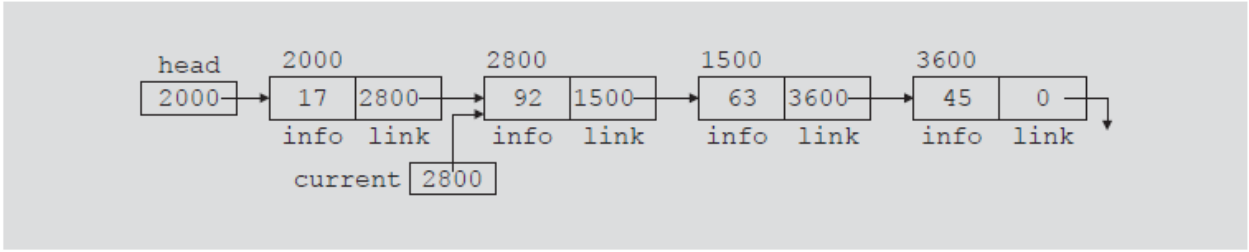


FIGURE 5-5 List after the statement current = current->link; executes

Table 5-2 shows the values of current, head, and some other nodes in Figure 5-5

TABLE 5-2 Values of current, head, and some of the nodes of the linked list in Figure 5-5

	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

TRAVERSING A LINKED LIST

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, display the elements of the list, and delete an item from the list.

These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer **head** to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to

maintain the list). Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

```
current = head;
while (current != NULL)
{
        //Process current
        current = current->link;
}
```

For example, suppose that **head** points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;
while (current != NULL)
{
         cout << current->info << " ";
         current = current->link;
}
```

LINKED LIST ALGORITHMS

This section discusses the algorithms of linked list data structures. Consider the following definition of a node. (For simplicity, we assume that the info type is **int**.)

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

We will use the following variable declaration:

```
nodeType *head, *p, *q, *newNode;
```

ALGORITHM FOR INSERTING A NODE

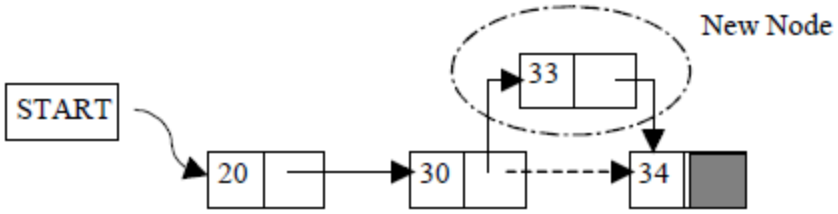


Fig. 5.14. Insertion of New Node

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
5. If (SATRT equal to NULL)
   (a) START = NewNode
6. Else
   (a) TEMP = START
   (b) While (TEMP -> Next not equal to NULL)

(i) TEMP = TEMP -> Next

7. TEMP -> Next = NewNode
8. Exit

Insert a Node at any specified position

1. Input DATA and POS to be inserted
2. initialize TEMP = START; and k = 0
3. Repeat the step 3 while(k is less than POS)
   (a) TEMP = TEMP -> 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

Consider the linked list shown in Figure 5-6.

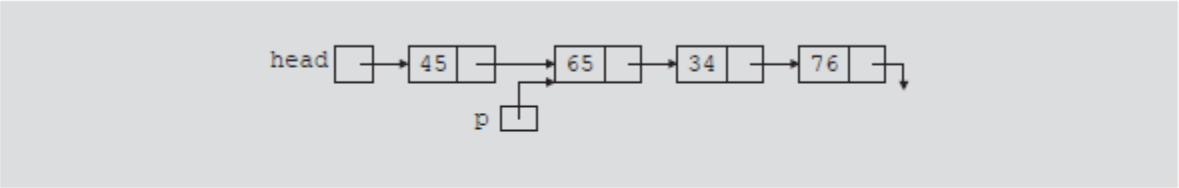


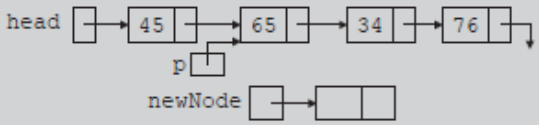
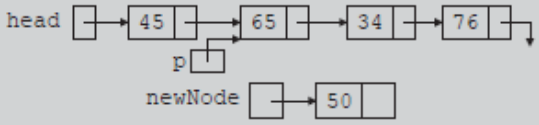
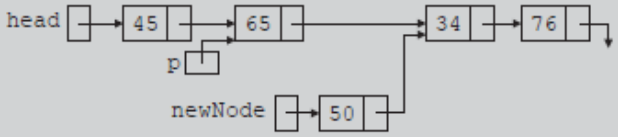
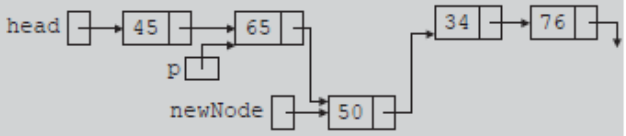
FIGURE 5-6 Linked list before item insertion

Suppose that **p** points to the node with **info 65**, and a new node with **info 50** is to be created and inserted after **p**. Consider the following statements:

```
newNode = new nodeType;          //create newNode
newNode->info = 50;              //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

Table 5-3 shows the effect of these statements.

TABLE 5-3 Inserting a node in a linked list

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Note that the sequence of statements to insert the node, that is,

```
newNode->link = p->link;
p->link = newNode;
```

is very important because to insert **newNode** in the list we use only one pointer, **p**, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link = newNode;
newNode->link = p->link;
```

Figure 5-7 shows the resulting list after these statements execute.

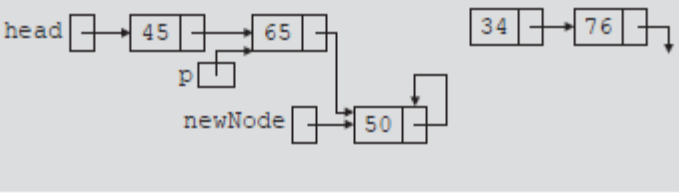


FIGURE 5-7 List after the execution of the statement `p->link = newNode;` followed by the execution of the statement `newNode->link = p->link;`

From Figure 5-7, it is clear that **newNode** points back to itself and the remainder of the list is lost.

Using two pointers, we can simplify the insertion code somewhat. Suppose **q** points to the node with **info 34**. (See Figure 5-8.)

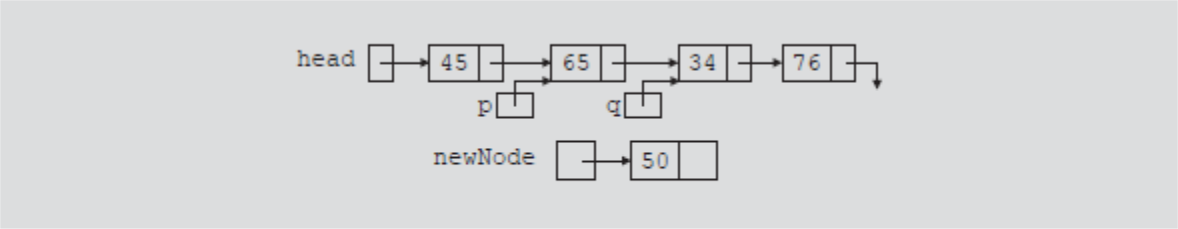


FIGURE 5-8 List with pointers p and q

The following statements insert **newNode** between **p** and **q**:

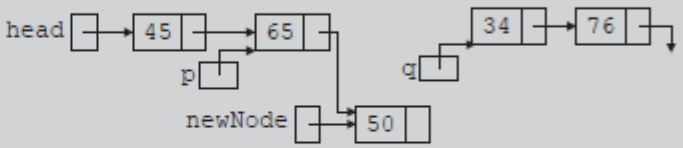
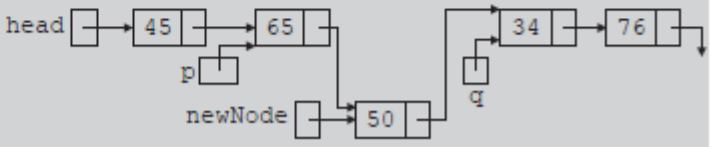
```
newNode->link = q;
p->link = newNode;
```

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

```
p->link = newNode;
newNode->link = q;
```

Table 5-4 shows the effect of these statements.

TABLE 5-4 Inserting a node in a linked list using two pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

ALGORITHM FOR DELETING A NODE

Consider the linked list shown in Figure 5-9.

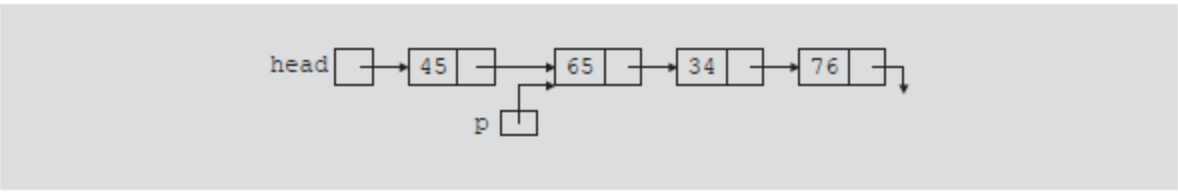


FIGURE 5-9 Node to be deleted is with info 34

Suppose that the node with **info 34** is to be deleted from the list. The following statement removes the node from the list:

```
p->link = p->link->link;
```

Figure 5-10 shows the resulting list after the preceding statement executes.

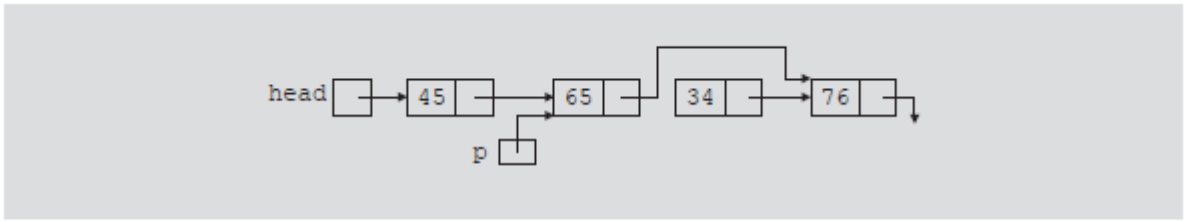


FIGURE 5-10 List after the statement `p->link = p->link->link;` executes

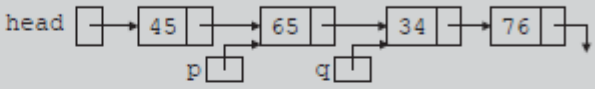
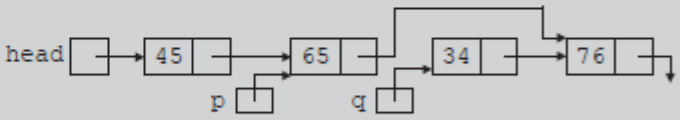
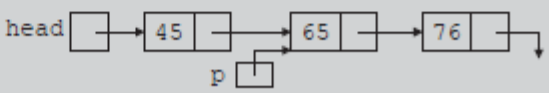
From Figure 5-10, it is clear that the node with **info 34** is removed from the list.

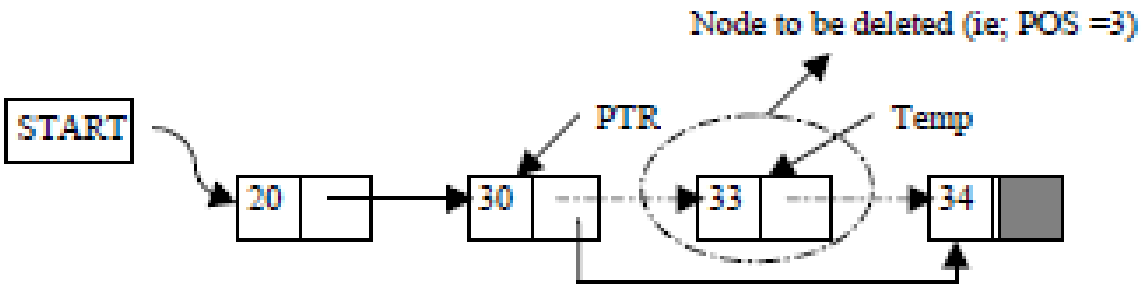
However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link;
p->link = q->link;
delete q;
```

Table 5-5 shows the effect of these statements.

TABLE 5-5 Deleting a node from a linked list

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	



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

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

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

BUILDING A LINKED LIST

Now that we know how to insert a node in a linked list, let us see how to build a linked list. First, we consider a linked list in general. If the data we read is unsorted, the linked list will be unsorted. Such a list can be built in two ways: forward and backward. In the forward manner, a new node is always inserted at the end of the linked list. In the backward manner, a new node is always inserted at the beginning of the list. We will consider both cases.

BUILDING A LINKED LIST FORWARD

Suppose that the nodes are in the usual **info-link** form and **info** is of type **int**. Let us assume that we process the following data: 2 15 8 24 34

We need three pointers to build the list: one to point to the first node in the list, which cannot be moved, one to point to the last node in the list, and one to create the newnode. Consider the following variable declaration:

```
nodeType *first, *last, *newNode;
int num;
```

Suppose that first points to the first node in the list. Initially, the list is empty, so both first and last are NULL. Thus, we must have the statements

```
first = NULL;
last = NULL;
```

to initialize first and last to NULL. Next, consider the following statements:

```
1  cin >> num;                          //read and store a number in num
2  newNode = new nodeType;              //allocate memory of type nodeType and store the address of the
                                        //allocated memory in newNode
3  newNode->info = num;                 //copy the value of num into the info field of newNode
4  newNode->link = NULL;                //initialize the link field of newNode to NULL
5  if (first == NULL)                   //if first is NULL, the list is empty;
                                        //make first and last point to newNode
       {
5a          first = newNode;
5b          last = newNode;
       }
6  else                                 //list is not empty
       {
6a         last->link = newNode;        //insert newNode at the end of the list
6b         last = newNode;              //set last so that it points to the
                                        //actual last node in the list
       }
```

Let us now execute these statements. Initially, both **first** and **last** are **NULL**. Therefore, we have the list as shown in Figure 5-11.

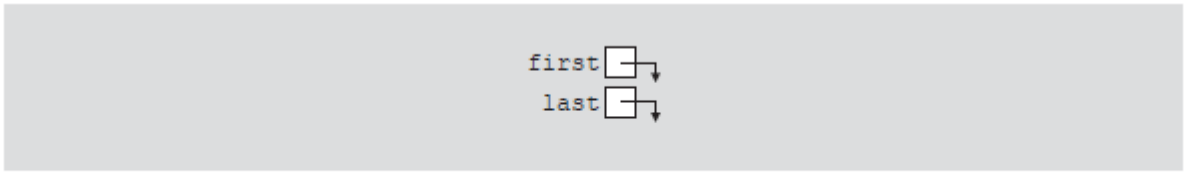


FIGURE 5-11 Empty list

After statement 1 executes, **num** is 2. Statement 2 creates a node and stores the address of that node in **newNode**. Statement 3 stores 2 in the info field of **newNode**, and statement 4 stores **NULL** in the link field of **newNode**. (See Figure 5-12.)

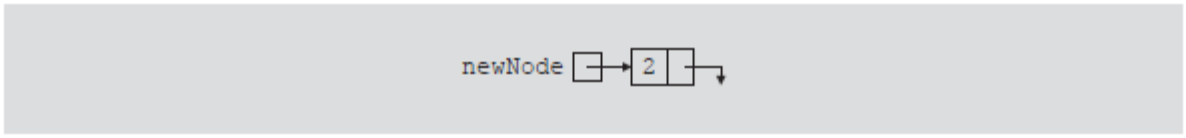


FIGURE 5-12 newNode with info 2

Because first is NULL, we execute statements 5a and 5b. Figure 5-13 shows the resulting list.

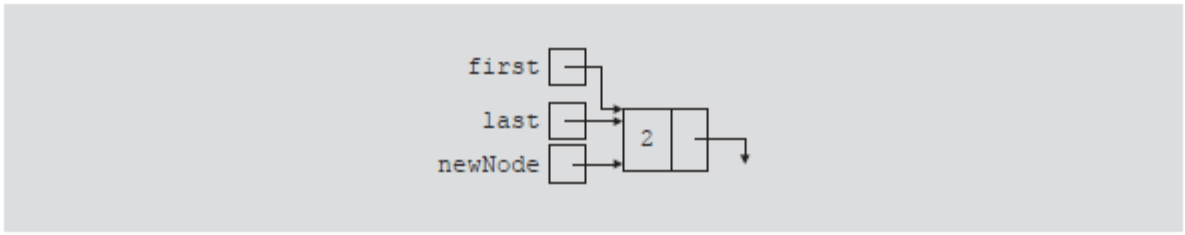


FIGURE 5-13 List after inserting newNode in it

We now repeat statements 1 through 6b. After statement 1 executes, **num** is 15. Statement 2 creates a node and stores the address of this node in **newNode**. Statement 3 stores 15 in the info field of **newNode**, and statement 4 stores **NULL** in the link field of **newNode**. (See Figure 5-14.)

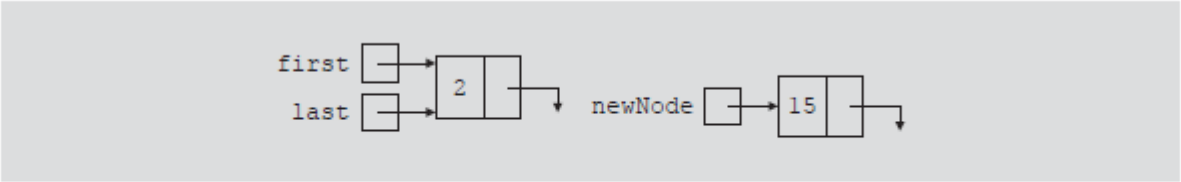


FIGURE 5-14 List and newNode with info 15

Because **first** is not **NULL**, we execute statements 6a and 6b. Figure 5-15 shows the resulting list.

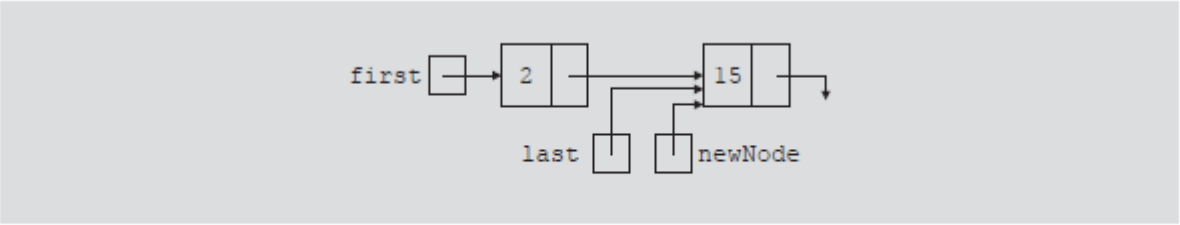


FIGURE 5-15 List after inserting newNode at the end

We now repeat statements 1 through 6b three more times. Figure 5-16 shows the resulting list.

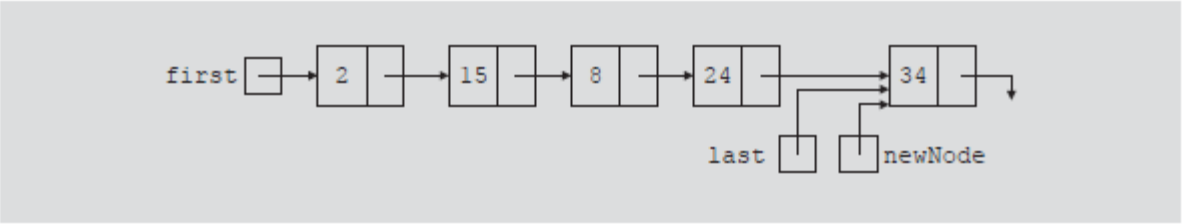


FIGURE 5-16 List after inserting 8, 24, and 34

We can put the previous statements in a loop, and execute the loop until certain conditions are met, to build the linked list. We can, in fact, write a C++ function to build a linked list.

Suppose that we read a list of integers ending with -999. The following function, buildListForward, builds a linked list (in a forward manner) and returns the pointer of the built list:

```
nodeType* buildListForward()
{
    nodeType *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999." << endl;
    cin >> num;
```

```
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
            last = newNode;
        }
        cin >> num;
    }   //end while

    return first;

}       //end buildListForward
```

BUILDING A LINKED LIST BACKWARD

Now we consider the case of building a linked list backward. For the previously given data—2, 15, 8, 24, and 34—the linked list is as shown in Figure 5-17.

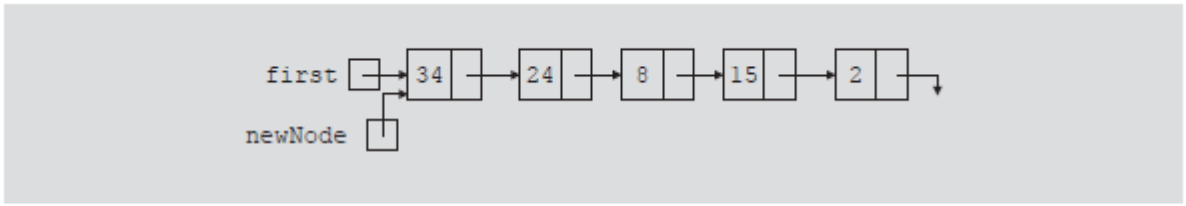


FIGURE 5-17 List after building it backward

Because the new node is always inserted at the beginning of the list, we do not need to know the end of the list, so the pointer **last** is not needed. Also, after inserting the new node at the beginning, the new node becomes the first node in the list. Thus, we need to update the value of the pointer **first** to correctly point to the first node in the list. We see, then, that we need only two pointers to build the linked list: one to point to the list and one to create the new node. Because initially the list is empty, the pointer first must be initialized to **NULL**.

The following C++ function builds the linked list backward and returns the pointer of the built list:

```
nodeType* buildListBackward()
{
        nodeType *first, *newNode;
        int num;

        cout << "Enter a list of integers ending with -999." << endl;
        cin >> num;
        first = NULL;

        while (num != -999)
        {
                newNode = new nodeType;      //create a node
                newNode->info = num;         //store the data in newNode
                newNode->link = first;      //put newNode at the beginning
                                             //of the list
                first = newNode;            //update the head pointer of
                                             //the list, that is, first
                cin >> num;                  //read the next number
        }

        return first;
}                                       //end buildListBackward
```

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.

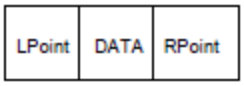


Fig. 5.24. A typical doubly linked list node

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 nex 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

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;
    Node *next;
    Node *prev;
};
```

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.

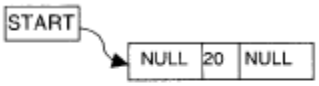


Fig. 5.27. Add(20)

Fig 5.28. Insert (30) at the end

Fig 5.29. Insert (10) at the beginning

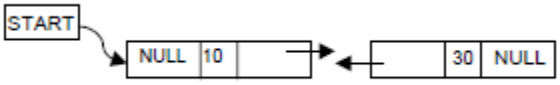


Fig 5.30. Delete a node at the 2nd position

Algorithm for creating a doubly linked list (inserting at the end)

1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->info=DATA
4. TEMP->next=NULL
5. if (START is equal to NULL)
    5.a. TEMP->prev=NULL
    5.b. START=TEMP
    5.c. exit
6. else
    6.a. HOLD =START
    6.b. while(HOLD->next not equal to NULL)
        6.b.1 HOLD=HOLD->next
    6.c. HOLD->next=TEMP
    6.d. TEMP->prev=HOLD
7. exit

Algorithm for inserting a node at the beginning

1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->prev=NULL
4. TEMP->info=DATA
5. TEMP->next=START
6. if (START is equal to NUUl)
    6.a. START=TEMP
    6.b. exit
7. START->prev=TEMP
8. START=TEMP
9. exit

Algorithm for inserting a node at a specific position

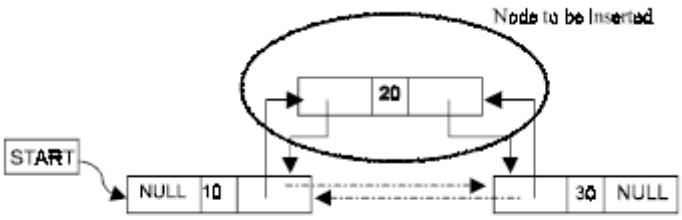


Fig. 5.31. Insert a node at the 2nd position

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 ->next; 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 -> next = TEMP -> next
    (d) NewNode -> prev = TEMP
    (e) (TEMP -> next) -> prev = NewNode
    (f) TEMP -> next = New Node
6. Else
    (a) Display "Position NOT found"
7. Exit

Algorithm for deleting a node

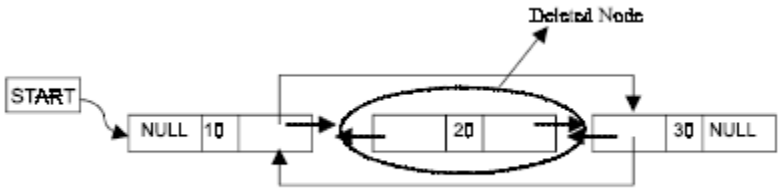


Fig. 5.32. Delete a node at the 2nd position

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

1. Input the DATA to be deleted
2. if ((START->DATA)is equal to DATA)
    (a) TEMP = START
    (b) START = STAR->next
    (c) START->prev=NULL
    (d) set free the node TEMP , which is deleted
    (e) Exit
3. HOLD = START
4. while((HOLD->next->next) not equal to NULL)
    (a)if (HOLD->next->DATA) equal to DATA)
        (a1) TEMP = HOLD->next
        (a2)HOLD->next=TEMP->next
        (a3) TEMP->next->prev=HOLD
        (a4) set free the node TEMP, which is deleted
        (a5) 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. Display "DATA not found"

7.Exit

Algorithm for displaying the doubly linked list

1. if (START is equal to NULL)
    1.a. display “The list is empty”
    1.b. exit
2. TEMP=START
3. while TEMP is not equal to NULL
    3.a. display TEMP->info
    3.b. TEMP=TEMP->next
4. exit

Assignment

1. write an algorithm and a function to display a doubly linked list in reverse order.
2. write an algorithm and a function to count the number of elements in the doubly linked list.
3. write an algorithm and function for searching a number in doubly linked list.

The Stack data structure

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.

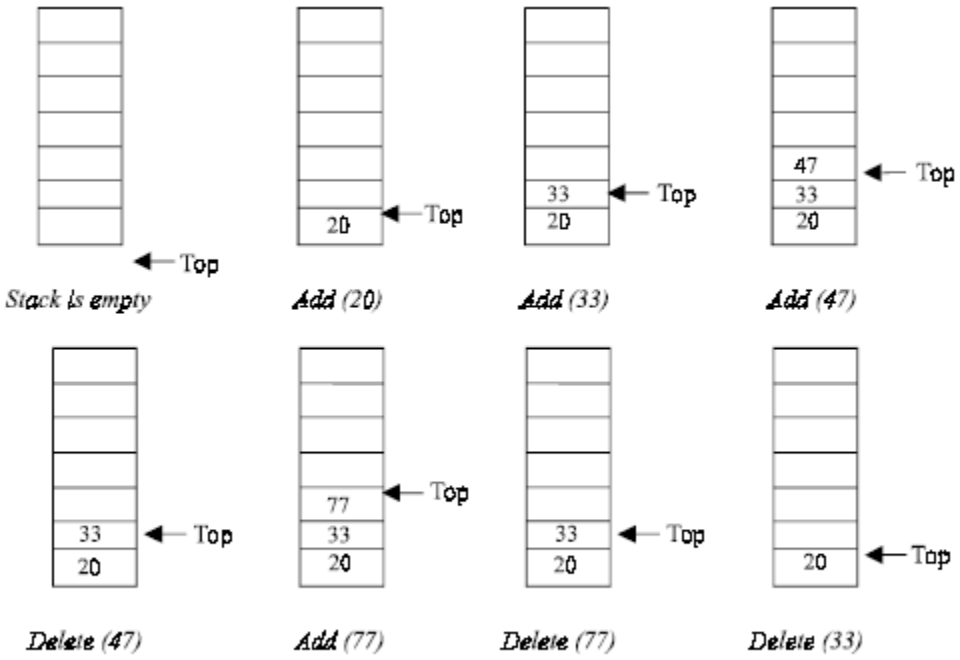


Fig. 3.1. Stack operation.

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.

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.

STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (linked list)

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.

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

```
void push(void)
{
        int x;
        if(top==max-1)     // Condition for checking If Stack is Full
        {
                cout<<"\nstack overflow\n";
                return;
        }
        cout<<"enter a no:  ";
        cin>>x;
        a[++top]=x;    //increment the top and inserting element
        cout<< "\nsucc. pushed:  " << x << endl << endl;
             return;
}
```

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 is equal to -1, then
   (a) Display "The Stack is empty"
   (b) Exit
2. DATA = STACK[TOP]
3. TOP = TOP – 1
4. Exit

```
void pop(void)
{
        int y;
        if(top==-1)            // Condition for checking If Stack is Empty
        {
                cout <<"stack underflow\n";
                return;
        }
        y=a[top];
        a[top--]=0;     //insert 0 at place of removing element and decrement the top
        cout <<"\n succ.poped\n\n";
        return;
}
```

Algorithm for display

1. If TOP is equal to -1, then
   (a) Display "the stack is empty"
   (b) exit
2. i ←TOP to 0
   (a) display Array[i]
3. Exit

```
void display(void)
{
        if(top==-1)
        {
               cout <<"stack is empty\n";
               return;
        }
        cout<<"\nelements of Stack are : ";
        for(int i=0;i<=top;i++)
        {
                cout << a[i] << "  ";
         }
             cout << endl << endl;
         return;
}
```

STACK USING LINKED LIST

we have 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.

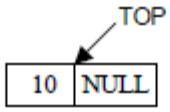


Fig. 5.11. push (10)

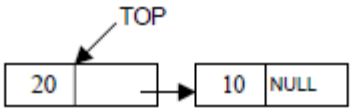


Fig. 5.12. push (20)

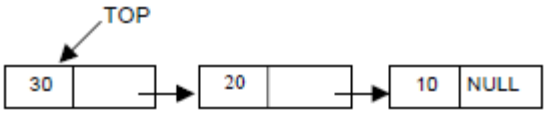


Fig. 5.13. push (30)

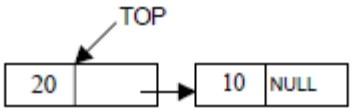


Fig. 5.14. X = pop() (ie; X = 30)

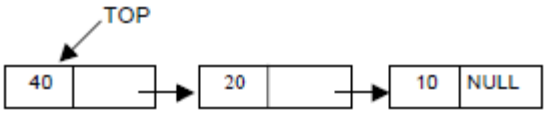


Fig. 5.15. push (40)

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

```
void push()
{
     int item;
     Node *NewNode;
     NewNode = new Node;
     cout<<"\ninput the new value to be pushed on the stack: ";
     cin>>item;
     NewNode->info=item;
     NewNode->link=top;
     top=NewNode;
}
```

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

```
void pop(){
if (top==NULL)
    cout<<"the stack is empty\n";
else
    {
       Node *temp=top;
       cout<<"the poped element is: "<<top->info;
       top=top->link;
       temp->link=NULL;
       delete temp;
    }
}
```

Algorithm for display operation

1. if (TOP is equal to NULL)
   (a) display "the stack is empty"
   (b) exit
2. else
   (a) temp = top
   (b) while temp is not equal to null
      (b.1) display temp->info
      (b.2) temp = temp->link
3. exit

```
void display(){
if(top==NULL)
    cout<<"the stack is empty"<<endl;
else
{
    cout<<"\nthe stack elements are: "<<endl;
    Node *temp=top;
    while(temp!=NULL)
    {
       cout<<temp->info;
       temp=temp->link;
    }
}
}
```

Stack applications

1- Check for balanced parentheses in an expression

Stack is used to check whether a given arithmetic expression containing nested parentheses is properly parenthesized.

Given an expression string exp, write a program to examine whether the pairs and the orders of "{","}","(",")","[","]" are correct in exp. For example, the program should print true for exp = "[()]{}{[()()]()}" and false for exp = "[(])"

Ex: (((a + b)*c + d – e) / (f + g) - (h + j)*(k – 1)/(m – n))

3 2 1	**2 1**	**6 1**	**8 1**	**10 1**	**12 1**	**1**	**empty**

Do the same for (a – b) * (c + d / (e – f)) / (g + h)

Algorithm:

1) Declare a character stack S.
2) Now traverse the expression string exp.
   a) If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
   b) If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3) After complete traversal, if there is some starting bracket left in stack then "not balanced"(should end with an empty stack)

AreParanthesesBalanced (exp)

{

n ← length (exp)

creat a stack S

for i ← 0 to n – 1

{

if exp[i] is opening symbol

Push (exp[i])

else if exp[i] is closing symbol

{

if (S is empty) ||(top does not pair with exp[i])

{

return false

}

else

Pop()

}

}

return S is empty ? true : false

}

The program

```
#include<iostream>
#include<stack>
#include<string>
#include<conio.h>
using namespace std;

// Function to check whether two characters are opening
// and closing of same type.
bool ArePair(char opening,char closing)
{
     if(opening == '(' && closing == ')') return true;
     else if(opening == '{' && closing == '}') return true;
     else if(opening == '[' && closing == ']') return true;
     return false;
}

bool AreParanthesesBalanced(string exp)
{
     stack<char> S;
     for(int i =0;i<exp.length();i++)
```

```
	{
		if(exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
			S.push(exp[i]);
		else if(exp[i] == ')' || exp[i] == '}' || exp[i] == ']')
		{
			if(S.empty() || !ArePair(S.top(),exp[i]))
				return false;
			else
				S.pop();
		}
	}
	return S.empty() ? true:false;
}

int main()
{
	/*Code to test the function AreParanthesesBalanced*/
	string expression;
	cout<<"Enter an expression:  "; // input expression from STDIN/Console
	cin>>expression;
	if(AreParanthesesBalanced(expression))
		cout<<"Balanced\n";
	else
		cout<<"Not Balanced\n";

	getch();
	return 0;
}
```

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(Order of Operations), 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

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

H.W. Write a C++ program to implement the algorithm above (converting infix to postfix algorithm).

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. Scan P from left to right and repeat Steps 3 and 4 for each element of P.
2. If an operand is encountered, put it on STACK.
3. If an operator $\otimes$ 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 $\otimes$ A.
   (c) Place the result on to the STACK.
4. Result equal to the top element on STACK.
5. Exit.

H.W. Write a C++ program to implement the algorithm above (Evaluating postfix expression).