MODULE 1

INTRODUCTION TO DATA STRUCTURES:

Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations

1. It must be rich enough in structure to mirror the actual relationships of the data in the real world.

2. The structure should be simple enough that one can effectively process the data whenever necessary.

Basic Terminology: Elementary Data Organization:

Data: Data are simply values or sets of values.

Data items: Data items refers to a single unit of values.

Data items that are divided into sub-items are called Group items. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.

Data items that are not able to divide into sub-items are called Elementary items. Ex: SSN

Entity: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

Ex: **Attributes**- Names, Age, Sex, SSN
Values- Rohland Gail, 34, F, 134-34-5533

Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute. The term "information" is sometimes used for data with given attributes, of, in other words meaningful or processed data.

Field is a single elementary unit of information representing an attribute of an entity.

Record is the collection of field values of a given entity.

File is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a primary key and the values k1, k2, ….. in such a field are called keys or key values.

Records may also be classified according to length.

A file can have fixed-length records or variable-length records.

- In fixed-length records, all the records contain the same data items with the same amount of space assigned to each data item.
- In variable-length records file records may contain different lengths.

Example: Student records have variable lengths, since different students take different numbers of courses. Variable-length records have a minimum and a maximum length. The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures.

The study of complex data structures includes the following three steps:

1. Logical or mathematical description of the structure

2. Implementation of the structure on a computer

3. Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into

- Primitive data Structures
- Non-primitive data Structures

1. Primitive data Structures: Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

2. Non-Primitive data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

1. Linear Data Structure: A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

1. One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. **Non-linear Data Structure**: A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.

DATA STRUCTURES OPERATIONS:

The data appearing in data structures are processed by means of certain operations. The following four operations play a major role in this text:

1. Traversing: accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)

2. Searching: Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

3. Inserting: Adding a new node/record to the structure.

4. Deleting: Removing a node/record from the structure.

The following two operations, which are used in special situations:

1. Sorting: Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

2. Merging: Combining the records in two different sorted files into a single sorted file

Review of Pointers and Dynamic Memory Allocation:

Pointers:

C provides extensive support for pointers. The actual value of a pointer type is an address of memory. The two most important operators used with the pointer type are:

& the address operator

* the dereferencing (or indirection) operator

If we have the declaration:

```
int i, *pi ;
```

then i is an integer variable and pi is a pointer to an integer. If we say:

```
pi = &i;
```

then &i returns the address of i and assigns it as the value ofpi. To assign a value to i we can say:

```
i= 10;
```

or

```
*pi 10;
```

In both cases the integer 10 is stored as the value of i. In the second case, the * in front of the pointer pi causes it to be dereferenced, by which we mean that instead of storing 10 into the pointer, 10 is stored into the location pointed at by the pointer pi.

There are other operations we can do on pointers. We may assign a pointer to a variable of type pointer. Since a pointer is just a nonnegative integer number, C allows us to perform arithmetic operations such as addition, subtraction, multiplication, and division, on pointers. We also can determine if one pointer is greater than, less than, or equal to another, and we can convert pointers explicitly to integers.

The size of a pointer can be different on different computers. In some cases the size of a pointer on a computer can vary. For example, the size of a pointer to a char can be longer than a pointer to a float. C has a special value that it treats as a null pointer. The null pointer points to no object or function. Typically the null pointer is represented by the integer 0. There is a macro called NULL which is defined to be this constant. The macro is defined either in stddefh for ANSI C or in stdio.h for K&R C. The null pointer can be used in relational expressions, where it is interpreted as false.

Therefore, to test for the null pointer in C we can say:

```
if (pi == NULL)
```

or more simply:

```
if (!pi)
```

Dynamic Memory Allocation:

In your program you may wish to acquire space in which you will store information. When you write your program you may not know how much space you will need, nor do you wish to allocate some very large area that may never be required. To solve this problem C provides a mechanism, called a heap, for allocating storage at run-time. Whenever you need a new area of memory, you may call a function, malloc, and request the amount you need. If the memory is available, a pointer to the start of an area of

memory of the required size is returned. At a later time when you no longer need an area of memory, you may free it by calling another function, free, and return the area of memory to the system. Once an area of memory is freed, it is improper to use it. The following program shows how we might allocate and deallocate storage to pointer variable

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Program 4.1: Allocation and deallocation of pointers

The call to malloc includes a parameter that determines the size of storage required to hold the int or the float. The result is a pointer to the first address of a storage area of the proper size. The type of the result can vary. On some systems the result of malloc is a char *, a pointer to a char. However, those who use ANSI C will find that the result is void *. The notation {int *) and (float *) are type cast expressions. They transform the resulting pointer into a pointer to the correct type. The pointer is then assigned to the proper pointer variable. The free function deallocates an area of memory previously allocated by malloc. In some versions of C, free expects an argument that is a char *, while ANSI C expects void *. However, the casting of the argument is generally omitted in the call to free

In Program 4.1 if we insert the line:

pf = (float *) malloc(sizeof(float));

immediately after the printf statement, then the pointer to the storage used to hold the value 3.14 has disappeared. Now there is no way to retrieve this storage. This is an example of a dangling reference. Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. As we examine programs that make use of pointers and dynamic storage, we will make it a point to always return storage after we no longer need it

Pointers can be dangerous:

By using pointers, we can attain a high degree of flexibility and efficiency. But pointers can be dangerous as well. When programming in C, it is a wise practice to set all pointers to NULL when they are not actually pointing to an object. This makes it less likely that you will attempt to access an area of memory that is either out of range of your program or that does not contain a pointer reference to a legitimate object. On some computers, it is possible to dereference the null pointer and the result is

NULL, permitting execution to continue. On other computers, the result is whatever the bits are in location zero, often producing a serious error.

Another wise programming tactic is to use explicit type casts when converting between pointer types. For example:

pi = malloc(sizeof(int)); /*assign to pi a pointer to int */

pf = (float *) pi; /*casts an int pointer to a float pointer */

Another area of concern is that in many systems, pointers have the same size as type int. Since int is the default type specifier, some programmers omit the return type when defining a function. The return type defaults to int which can later be interpreted

Arrays and Structures:

Abstract Data Type Array

---

structure *Array* is
objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, ··· , *n*−1} for one dimension, {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.
functions:
for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ integer

Array Create(*j*, *list*)	::=	**return** an array of *j* dimensions where *list* is a *j*-tuple whose *i*th element is the the size of the *i*th dimension. *Items* are undefined.
Item Retrieve(*A*, *i*)	::=	**if** ($i \in index$) **return** the item associated with index value *i* in array *A* **else return** error
Array Store(*A*,*i*,*x*)	::=	**if** (*i* in *index*) **return** an array that is identical to array *A* except the new pair <*i*, *x*> has been inserted **else return** error.

end *Array*

Structure 2.1: Abstract Data Type *Array*

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number n of similar data referenced respectively by a set of n consecutive numbers, usually 1, 2, 3 . . . . . n. if A is chosen the name for the array, then the elements of A are denoted by subscript notation a1, a2, a3….. an or

by the parenthesis notation A (1), A (2), A (3).....A (n)

or

by the bracket notation A [1], A [2], A [3].....A [n]

For example,

list [5], *plist[5];

declares two arrays each containing five elements. The first array defines five integers, while the second defines five pointers to integers. In C all arrays start at index 0, so list[0], list[l], list[2], list[3] are the names of the five array elements, each of which contains an integer value. Similarly, plist[0], plist[1], plist[2], plist[3], and plist[4] are the names of five array elements, each of which contains a pointer to an integer.

We now consider the implementation of one-dimensional arrays. When the compiler encounters an array declaration such as the one used above to create list, it allocates five consecutive memory locations. Each memory location is large enough to hold a single integer. The address of the first element /z5Z[0], is called the base address. If the size of an integer on your machine is denoted by sizeofiint), then we get the following memory addresses for the five elements of list[]

Variable	Memory Address
list[0]	base address = α
list[1]	α + *sizeof(int)*
list[2]	α + 2·*sizeof(int)*
list[3]	α + 3·*sizeof(int)*
list[4]	α + 4·*sizeof(int)*

In fact, when we write list[i] in a C program, C interprets it as a pointer to an integer whose address is the one in the table above. Observe that there is a difference between a declaration such as

int *listl;

and

int list2 [5];

The variables list1 and list2 are both pointers to an int, but in the second case five memory locations for holding integers have been reserved, list2 is a pointer to list2[0] and list2+i is a pointer to list2[i]. Notice that in C, we do not multiply the offset i with the size of the type to get to the appropriate element of the array. Thus, regardless of the type of the array list2, it is always the case that (list2 + i) equals &list2[i]. So, *(list2 + i) equals list2[i].

It is useful to consider the way C treats an array when it is a parameter to a function. All parameters of a C function must be declared within the function. However, the range of a one-dimensional array is defined only in the main program since new storage for an array is not allocated within a function. If the size of a one-dimensional array is needed, it must be either passed into the function as an argument or accessed as a global variable.

Consider Program 2.1. When sum is invoked, input = &input [0] is copied into a temporary location and associated with the formal parameter list. When /z5r[zj occurs on the right-hand side of the equals sign, a dereference takes place and the value pointed at by (list + z) is returned. If list[i} appears on the left-hand side of the equals sign, then the value produced on the right-hand side is stored in the location (list 4- z). Thus in C, array parameters have their values altered, despite the fact that the parameter passing is done using call-by-value.

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 2.1: Example array program

Example 2.1 [One-dimensional array addressing]:

Assume that we have the following declaration:

int one[] = { 0 , 1, 2 , 3 , 4 } ;

We would like to write a function that prints out both the address of the zth element of this array and the value found at this address. To do this, print1{Program 2.2) uses pointer arithmetic. The function is invoked as print1(&one[0],5). As you can see from the printf statement, the address of the ith

element is simply ptr + i. To obtain the value of the ith element, we use the dereferencing operator, *. Thus, *(ptr + i) indicates that we want the contents of the ptr+i position rather than the address.

```
void print1(int *ptr, int rows)
{
/* print out a one-dimensional array using a pointer */
   int i;
   printf("Address Contents\n");
   for (i = 0; i < rows; i++)
      printf("%8u%5d\n", ptr + i, *(ptr + i));
   printf("\n");
}
```

Program 2.2: One-dimensional array accessed by address

Structures and Unions:

STRUCTURES

In C, a way to group data that permits the data to vary in type. This mechanism is called the structure, for short struct.

A structure (a record) is a collection of data items, where each item is identified as to its type and name.

Syntax: struct {

data_type member 1;

data_type member 2;

…… ……

data_type member n; } variable_name;

Ex: struct {

char name[10];

int age;

float salary; } Person;

The above example creates a structure and variable name is Person and that has three fields:

name = a name that is a character array

age = an integer value representing the age of the person

salary = a float value representing the salary of the individual

Assign values to fields

To assign values to the fields, use . (dot) as the structure member operator. This operator is used to select a particular member of the structure

Ex: strcpy(Person.name,“james”);
Person.age = 10;
Person.salary = 35000;

Type-Defined Structure

The structure definition associated with keyword typedef is called Type-Defined Structure.

Syntax 1: typedef struct
{
data_type member 1;
data_type member 2;
……
……
data_type member n;
}Type_name;

where,

- typedef is the keyword used at the beginning of the definition and by using typedef user defined data type can be obtained.
- struct is the keyword which tells structure is defined to the complier
- The members are declare with their data_type
- Type_name is not a variable, it is user defined data_type.

Syntax 2: struct struct_name {
data_type member 1;
data_type member 2;
……
……
data_type member n;
}; typedefstruct struct_name Type_name;

Ex: typedef struct{
char name[10];
int age;
float salary;
}humanBeing;

In above example, humanBeing is the name of the type and it is a user defined data type.

Declarations of structure variables:

humanBeing person1, person2;

This statement declares the variable person1 and person2 are of type humanBeing.

Structure Operation

The various operations can be performed on structures and structure members.

1. Structure Equality Check:

   Here, the equality or inequality check of two structure variable of same type or dissimilar type is not allowed

```
typedef struct{
        char name[10];
        int age;
        float salary;
}humanBeing;
humanBeing person1, person2;
if (person1 = = person2) is invalid.
```

The valid function is shown below

```
#define FALSE 0
#define TRUE 1
if (humansEqual(person1,person2))
        printf("The two human beings are the same\n");
else
        printf("The two human beings are not the same\n");


int humansEqual(humanBeing person1, humanBeing person2)
{ /* return TRUE if person1 and person2 are the same human being otherwise return FALSE
*/
        if (strcmp(person1.name, person2.name))
                return FALSE;
        if (person1.age!= person2.age)
                return FALSE;
        if (person1.salary != person2.salary)
                return FALSE;
```

return TRUE;

}

Program: Function to check equality of structures

2. Assignment operation on Structure variables:

person1 = person2

The above statement means that the value of every field of the structure of person 2 is assigned as the value of the corresponding field of person 1, but this is invalid statement.

Valid Statements is given below:

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

Structure within a structure:

There is possibility to embed a structure within a structure. There are 2 ways to embed structure.

1. The structures are defined separately and a variable of structure type is declared inside the definition of another structure. The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other member of that structure

Example:

The following example shows two structures, where both the structure are defined separately.

```
typedef struct {
        int month;
        int day;
        int year;
        }date;
typedef struct {
        char name[10];
        int age;
        float salary;
        date dob;
        } humanBeing;
humanBeing person1;
```

A person born on February 11, 1944, would have the values for the date struct set as:

```
person1.dob.month = 2;
```

```
person1.dob.day = 11;
person1.dob.year = 1944;
```

2. The complete definition of a structure is placed inside the definition of another structure.

Example:

```
typedef struct {
        char name[10];
        int age;
        float salary;
        struct {
                int month;
                int day;
                int year;
        } date;
} humanBeing;
```

SELF-REFERENTIAL STRUCTURES

A self-referential structure is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

Consider as an example:

```
typedef struct {
        char data;
        struct list *link ;
       }list;
```

Each instance of the structure list will have two components data and link.

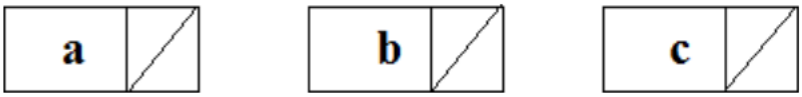
- Data: is a single character,
- Link: link is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.

Consider these statements, which create three structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
```

item3.data = 'c';

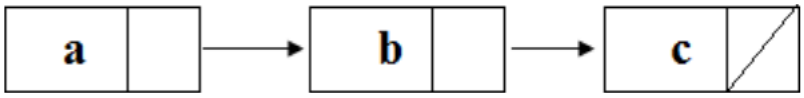
item1.link = item2.1ink = item3.link = NULL;



Structures item1, item2 and item3 each contain the data item a, b, and c respectively, and the null pointer. These structures can be attached together by replacing the null link field in item2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item 2.

item1.link = &item2;

item2.1ink = &item3;



Unions:

Unions: A union is similar to a structure, it is collection of data similar data type or dissimilar.

Syntax:

```
union{
        data_type member 1;
        data_type member 2;
        ……
        ……
        data_type member n;
        }variable_name;
```

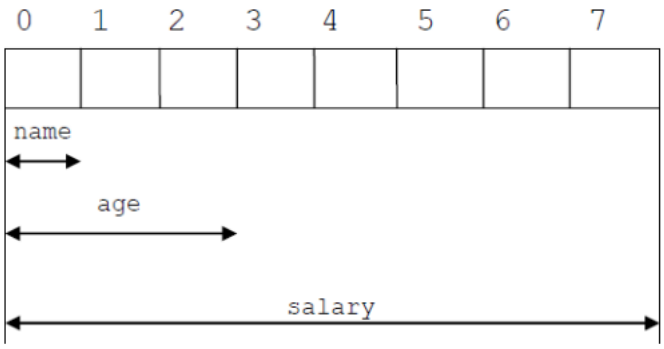
Example:

```
union{
        int children;
        int beard;
        } u;
```

Union Declaration:

A union declaration is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time.

```
union{
        char name;
        int age;
        float salary;
        }u;
```



The major difference between a union and a structure is that unlike structure members whichare stored in separate memory locations, all the members of union must share the same memory space. This means that only one field of the union is "active" at any given time.

POLYNOMIALS in Data Structures

What is a polynomial?

- A *polynomial* object is a homogeneous ordered list of pairs <*exponent*,*coefficient*>, where each coefficient is unique.
- Operations include returning the degree, extracting the coefficient for a given exponent, addition, multiplication, evaluation for a given input.

- **Polynomial Representation in C:**
- We have a polynomial,

$$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$$

-

- It is a collection of terms with a single variable '**x**'. Now let us see how to evaluate a polynomial and perform the addition of polynomials. Once we know addition, we can also learn how to perform subtraction and multiplication of polynomials.

```
struct Term{
        int coefficient;
        int exponent;
}
```

```
struct Poly{
        int n;
        struct Term *t;
}
```

This is having two-member, non-zero terms as '**n**' and pointer of type 'Term' to point on the array of 'Terms'. So, this is a structure for polynomial. It is using the 'Term' structure. Now let us learn how to write the program for creating the representation of any polynomial. Here we are writing the pseudocode,

Polynomial Representation Pseudo Code:

```
struct Poly p;

printf("No. of non-zero Terms: ");
scanf("%d", &p.n);

p.t = (Term*) malloc(p.n * sizeof(Term));
printf("Enter polynomial terms: ");
for(int i = 0; i < p.n; i++){
        printf("Term no. %d", i+1);
        scanf("%d%d", p.t[i].coefficient, &p.t[i].exponent);
}
```

Now, we will see how to evaluate a polynomial. We have a polynomial,

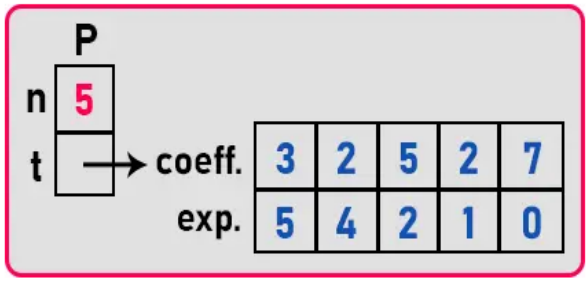
$$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$$

In C language, we have implemented the above data as,

```
struct Term{
        int coefficient;
        int exponent;
}
.struct Poly{
        int n;
        struct Term *t;
}
```

1st structure for storing the term data such as coefficient and exponent of term and 2nd structure for creating the polynomial from the array of terms.

Representation of Polynomial in C Language:



Now how to evaluate the polynomial? If we know the value of 'x' in the polynomial, then we can get the single value of the answer of the polynomial. First, we have to evaluate all the terms and add them. To evaluate a term, we have to first calculate the exponent part and multiplied with the coefficient. After this, add the result of the first term to the rest of the part of the polynomial.

Every time, we raise 'x' to that power and multiplied with the coefficient. All these terms can be computed and added one by one using the '**for**' loop. So, we have to process them all depending on the number of terms.

Polynomial Evaluation Pseudo Code:

```
struct Poly p;
int x = 5, sum = 0;

for(int i = 0; i < p.n; i++){
        sum += p.t[i].coefficient * pow(x, p.t[i].exponent);
}
return sum;
```

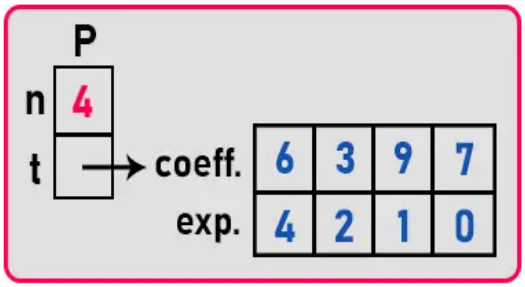
Here we have written pseudo code. In this pseudo code, we have declared a variable or object of type 'Poly'. Next, we have declared the value of 'x' as '5' and 'sum' as '0'. Then we run the 'for' loop to add all the terms. In the 'for' loop, we multiply the coefficient with the exponent of 'x' and store the result in the 'sum' variable, and modify the sum in every iteration. In this way, we can evaluate a polynomial.

Polynomial Addition in C:

Now, let us look at how we can add two polynomials and generate a third polynomial. We have our 1st polynomial is

$$p1(x) = 6x^4 + 3x^2 + 9x + 7$$

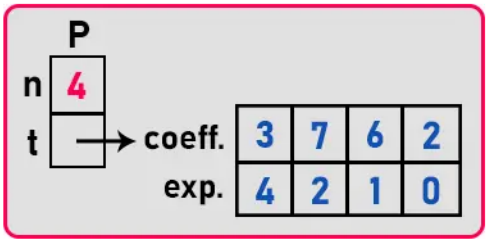
We will represent this polynomial data as,



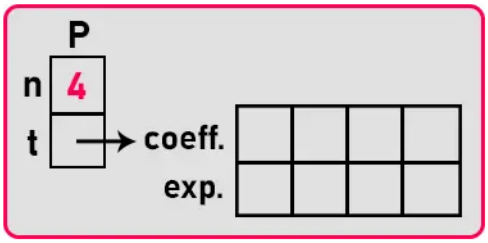
The 2nd polynomial is

$$p2(x) = 3x^4 + 7x^2 + 6x + 2$$

We can represent this as,



Now we want to add these polynomials. For adding these 2 polynomials, we have to create another structure of type 'Poly'. It will store the result of the addition of polynomials. Representation of 3rd polynomial is



Abstract Definition of Polynomial

structure *Polynomial* is
objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0
functions:
for all *poly*, *poly*1, *poly*2 ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent Lead_Exp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 · *poly*2

end *Polynomial*

Structure 2.2: Abstract data type *Polynomial*

What is a sparse matrix?

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Now, the question arises: we can also use the simple matrix to store the elements, then why is the sparse matrix required?

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Abstract Data Type Definition of Sparse Matrix

structure *Sparse_Matrix* is
objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.
functions:
for all $a, b \in$ *Sparse_Matrix*, $x \in$ *item*, i, j, *max_col*, *max_row* $\in$ *index*

Sparse_Matrix Create(*max_row*, *max_col*) ::=
return a *Sparse_Matrix* that can hold up to *max_items* = *max_row* × *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

Sparse_Matrix Transpose(*a*) ::=
return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=
if the dimensions of *a* and *b* are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else return error

Sparse_Matrix Multiply(*a*, *b*) ::=
if number of columns in *a* equals number of rows in *b*
return the matrix *d* produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
else return error.

Structure 2.3: Abstract data type *Sparse_Matrix*

Representation of sparse matrix

Now, let's see the representation of the sparse matrix. The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

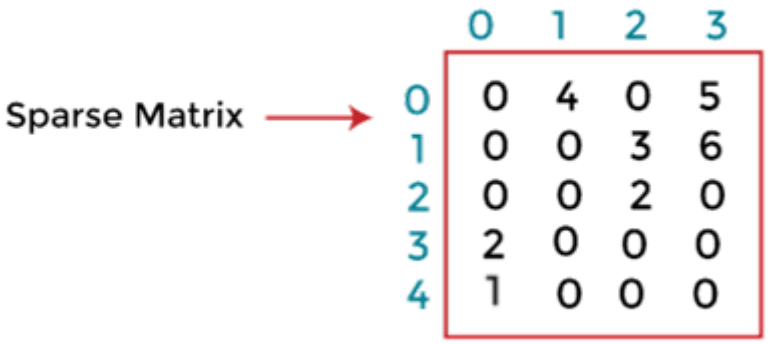
- Array representation
- Linked list representation

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

- **Row -** It is the index of a row where a non-zero element is located in the matrix.
- **Column -** It is the index of the column where a non-zero element is located in the matrix.
- **Value -** It is the value of the non-zero element that is located at the index (row, column).
- **Example -**
- Let's understand the array representation of sparse matrix with the help of the example given below -
- Consider the sparse matrix -



- In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements. The above matrix occupies 5x4 = 20 memory space. Increasing the size of matrix will increase the wastage space.
- The tabular representation of the above matrix is given below -

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

C Program for Sparse matrix implementation using arrays

- #include <stdio.h>

```
int main()
{
    // Sparse matrix having size 4*5
    int sparse_matrix[4][5] =
    {
        {0 , 0 , 6 , 0 , 9 },
        {0 , 0 , 4 , 6 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 1 , 2 , 0 , 0 }
    };
   // size of matrix
    int size = 0;
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<5; j++)
        {
            if(sparse_matrix[i][j]!=0)
            {
                size++;
            }
        }
    }
```

```
    // Defining final matrix
     int matrix[3][size];
      int k=0;
    // Computing final matrix
     for(int i=0; i<4; i++)
     {
        for(int j=0; j<5; j++)
        {
           if(sparse_matrix[i][j]!=0)
           {
              matrix[0][k] = i;
              matrix[1][k] = j;
              matrix[2][k] = sparse_matrix[i][j];
              k++;
           }
       }
     }
    // Displaying the final matrix
     for(int i=0 ;i<3; i++)
     {
        for(int j=0; j<size; j++)
        {
```

```
            printf("%d ", matrix[i][j]);
            printf("\t");
        }
        printf("\n");
    }
    return 0;
}
```

Linked List representation of the sparse matrix

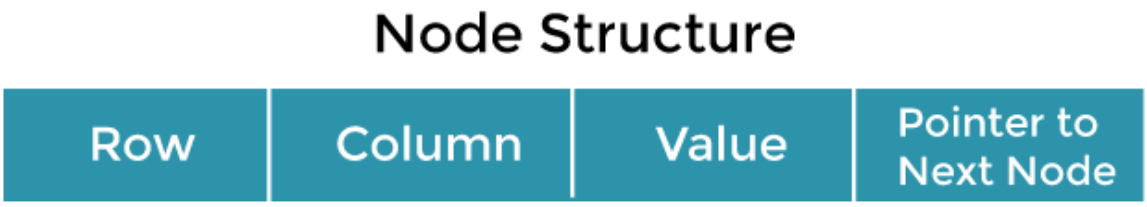
In a linked list representation, the linked list data structure is used to represent the sparse matrix. The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.

Unlike the array representation, a node in the linked list representation consists of four fields. The four fields of the linked list are given as follows -

- **Row -** It represents the index of the row where the non-zero element is located.

- **Column -** It represents the index of the column where the non-zero element is located.
- **Value -** It is the value of the non-zero element that is located at the index (row, column).
- **Next node -** It stores the address of the next node.

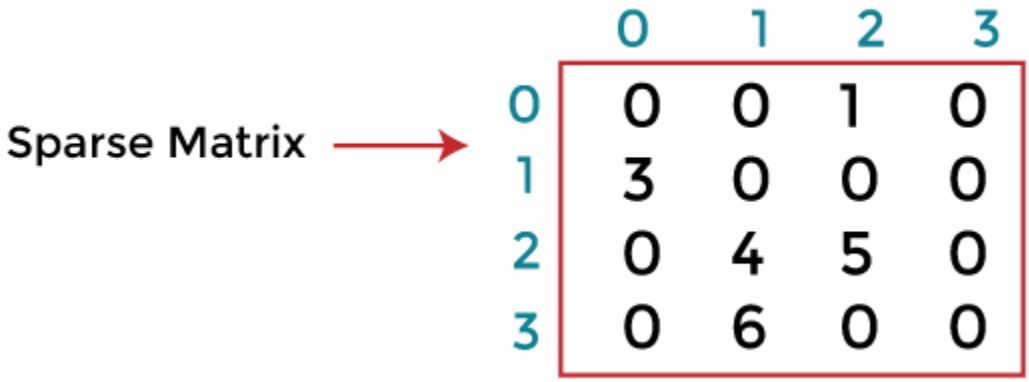
The node structure of the linked list representation of the sparse matrix is shown in the below image -



Example -

Let's understand the linked list representation of sparse matrix with the help of the example given below -

Consider the sparse matrix -



In the above figure, we can observe a 4x4 sparse matrix containing 5 non-zero elements and 11 zero elements. Above matrix occupies 4x4 = 16 memory space. Increasing the size of matrix will increase the wastage space.

The linked list representation of the above matrix is given below -

Implementation of linked list representation of sparse matrix

```
/*
 * Sparse Matrix Representation in C Using a Linked List
 */

#include<stdio.h>
#include<stdlib.h>
struct list
{
   int row, column, value;
```

```
    struct list *next;
};
struct list *HEAD=NULL;
void insert(int, int , int );
void print ();
int main()
{
    int Sparse_Matrix[4][4] ={ {9 , 0 , 0 , 0 },{0 , 0 , 0 , 0 },{0 ,
5 , 0 , 8 },{3 , 0 , 0 , 0 } };
    for(int i=0;i<4;i++)
    {
        for(int j=0;j<4;j++)
        {
            if(Sparse_Matrix[i][j] != 0)
            {
                insert(i, j, Sparse_Matrix[i][j]);
            }
        }
    }
    // print the linked list.
    print();
}
void insert( int r, int c, int v)
{
    struct list *ptr,*temp;
    int item;
    ptr = (struct list *)malloc(sizeof(struct list));
    if(ptr == NULL)
    {
        printf("\n OVERFLOW");
    }
    else
    {
```

```
        ptr->row = r;
        ptr->column = c;
        ptr->value = v;
        if(HEAD == NULL)
        {
            ptr->next = NULL;
            HEAD = ptr;
        }
        else
        {
            temp = HEAD;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
        }
    }
}
void print()
{
    struct list *tmp = HEAD;
    printf("ROW NO      COLUMN NO.        VALUE \n");
    while (tmp != NULL)
    {
        printf("%d \t\t %d \t\t %d \n", tmp->row, tmp->column,
tmp->value);
        tmp = tmp->next;
    }
}
```

Multidimensional Arrays in C

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays is generally stored in row-major order in the memory.

The ***general form of declaring N-dimensional arrays*** is

```
data_type array_name[size1][size2]....[sizeN];
```

- **data_type**: Type of data to be stored in the array.
- **array_name**: Name of the array.
- **size1, size2,…, sizeN**: Size of each dimension.
-

```
#include <stdio.h>


int main()
{    int a[2][3] = { { 1, 3, 2 }, { 6, 7, 8 } };
     int i, j;
     for (i = 0; i < 2; i++) {
          for (j = 0; j < 3; j++) {
```

```
            printf("\n a[%d][%d]=%d", i, j, a[i][j]);
        }   }
    return 0;
}
```

Output

```
a[0][0]=1
a[0][1]=3
a[0][2]=2
a[1][0]=6
a[1][1]=7
a[1][2]=8
```

String Handling in Data Structures

String in C

In C programming, String is a sequence of characters and it is terminated by null (‘\0’).

In other words, “String is a one-dimensional array of characters that is terminated by null.”

The last character of String should always be null (‘\0’). This shows where the string is ending.

Strings are always kept inside double quotes. While the character is kept inside the single quote.

Each character in the array takes up one byte of memory.

For example – char c [] = “yash”;

Declare string in C:

A string is a simple array whose data type is char. The general syntax to declare it is as follows:-

char str_name[size];

In the above syntax, str_name is the name given to the string variable and size is used to define the length of the string. The number of characters that can be stored in a string will be the size of it.

Initialize String in C:

There are many ways to initialize a string. An example has been given below with the help of which you can understand it easily. In this example there is a string named str and it is initialized with “Rajveer”.

The char str[] = “Rajveer”;

The char str[50] = “Rajveer”;

char str [] = {‘R’,’a’,’j’,’v’,’e’,’e’,’r’,’\ 0’};

char str [50] = {‘R’,’a’,’j’,’v’,’e’,’e’,’r’,’\ 0’};

Below you are given the memory representation of the string “Rajveer”.

Abstract Data Type Definition of String

---

structure *String* is
  objects: a finite set of zero or more characters.
  functions:
    for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

String Null(m)	::=	**return** a string whose maximum length is m characters, but is initially set to *NULL* We write *NULL* as "".
Integer Compare(s, t)	::=	**if** s equals t **return** 0 **else if** s precedes t **return** −1 **else return** +1
Boolean IsNull(s)	::=	**if** (Compare(s, *NULL*)) **return** *FALSE* **else return** *TRUE*
Integer Length(s)	::=	**if** (Compare(s, *NULL*)) **return** the number of characters in s **else return** 0.
String Concat(s, t)	::=	**if** (Compare(t, *NULL*)) **return** a string whose elements are those of s followed by those of t **else return** s.
String Substr(s, i, j)	::=	**if** (($j > 0$) && ($i + j - 1$) < Length(s)) **return** the string containing the characters of s at positions $i, i + 1, \cdots, i + j - 1$. **else return** *NULL*.

Structure 2.4: Abstract data type *String*

C String Functions:

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2*
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2*
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

Figure 2.7: C string functions

C String Program:-

```
#include<stdio.h>

#include<conio.h>

void main()

{

// declare and initialize string

char str[50] = “Hello World”;
```

```
// print string

printf(“%s”,str);

getch();

}
```

Output: Hello World

C String Functions –

String.h file header in C language supports all the string functions. All string functions are given below:-

Function	Description
strcat(s1, s2)	This concatenates string s2 at the end of string s1. (Concatenate means to add.)
strcpy(s1, s2)	This copies s2 into s1.
strlen(s1)	It returns the length of s1.
strcmp(s1, s2)	It returns 0 if s1 and s2 are equal. And returns 1 when s1<s2 or s1>s2.
strdup()	Creates a duplicate of the string.
strlwr()	It converts the string to lowercase.
strupr()	Converts the string to uppercase.
strrev()	It reverses the string.
strupr()	Converts the string to uppercase.
strrev()	It reverses the string.

C String Program:

```
// C String Program – String Functions
#include<stdio.h>
```

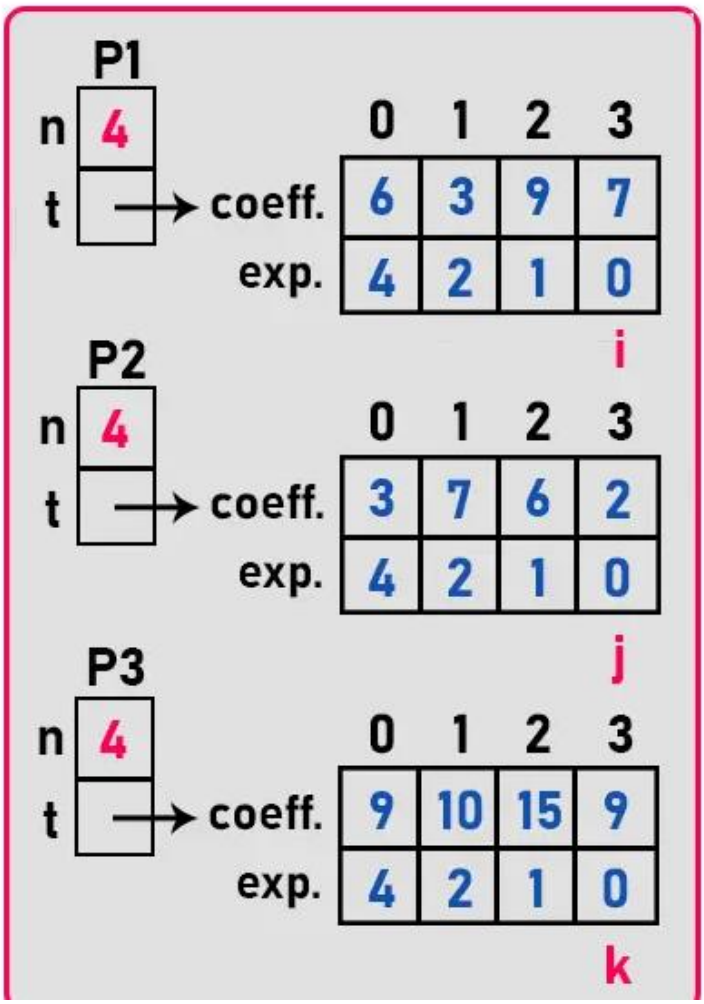
```
#include<conio.h>
#include<string.h>
void main()
{
char str1[15], str2[15], str3[20];
clrscr();
printf(“\n Enter the first string: “);
scanf(“%s”,str1);
printf(“\n Enter the second string: “);
scanf(“%s”,str2);
printf(“\n Show First string length: %d”,strlen(str1));
printf(“\n Show Frist string Lower case: %s”,strlwr(str1));
printf(“\n Show First string Upper case: %s”,strupr(str1));
printf(“\n Concatenate two String: %s”,strcat(str1,str2));
printf(“\n Show string Reverse order: %s”,strrev(str1));
printf(“\n Copy string str1 to str3: %s”,strcpy(str3,str1));
getch();
}
```

Output :

```
Enter the first string: Ram
Enter the second string: Sham
Show First string length: 3
Show Frist string Lower case: ram
Show First string Upper case: RAM
Concatenate two String: RAMSham
Show string Reverse order: mahSMAR
Copy string str1 to str3: mahSMAR
```

Another example Practical program 2 given.

In this way, we add two polynomials programmatically. The final value of 'P3' will be



'P3' will be represented as:

$$p(x) = 9x^4 + 10x^2 + 15x + 9$$

Polynomial Addition Pseudo Code:

```
int i = 0, j = 0, k = 0;
while(i < p1.n && j < p2.n){
        if(p1.t[i].exp > p2.t[j].exp)
                p3.t[k++] = p1.t[i++];
        else if(p2.t[j].exp > p1.t[i].exp)
                P3.t[k++] = p2.t[j++];
        else{
                p3.t[k].exp = p1.t[i].exp;
                p3.t[k++].coeff = p1.t[i++].coeff + p2.t[j++].coeff
        }
}
```

This is pseudo-code for addition.

String Insertion:

Example 2.2 [*String insertion*]: Assume that we have two strings, say *string* 1 and *string* 2, and that we want to insert *string* 2 into *string* 1 starting at the *i*th position of *string* 1. We begin with the declarations:

```
#include <string.h>
#define MAX_SIZE 100 /*size of largest string*/
char string1[MAX_SIZE], *s = string1;
char string2[MAX_SIZE],  *t = string2;
```

In addition to creating the two strings, we also have created a pointer for each string.

Now suppose that the first string contains "amobile" and the second contains "uto" (Figure 2.9). We want to insert "uto" starting at position 1 of the first string, thereby producing the word "automobile." We can accomplish this using only three function calls, as Figure 2.9 illustrates. Thus, in Figure 2.9(a), we assume that we have an empty string that is pointed to by *temp*. We use *strncpy* to copy the first *i* characters from *s* into *temp*.

Since $i = 1$, this produces the string "a." In Figure 2.9(b), we concatenate *temp* and *t* to produce the string "auto." Finally, we append the remainder of *s* to *temp*. Since *strncat* copied the first *i* characters, the remainder of the string is at address $(s + i)$. The final result is shown in Figure 2.9(c).

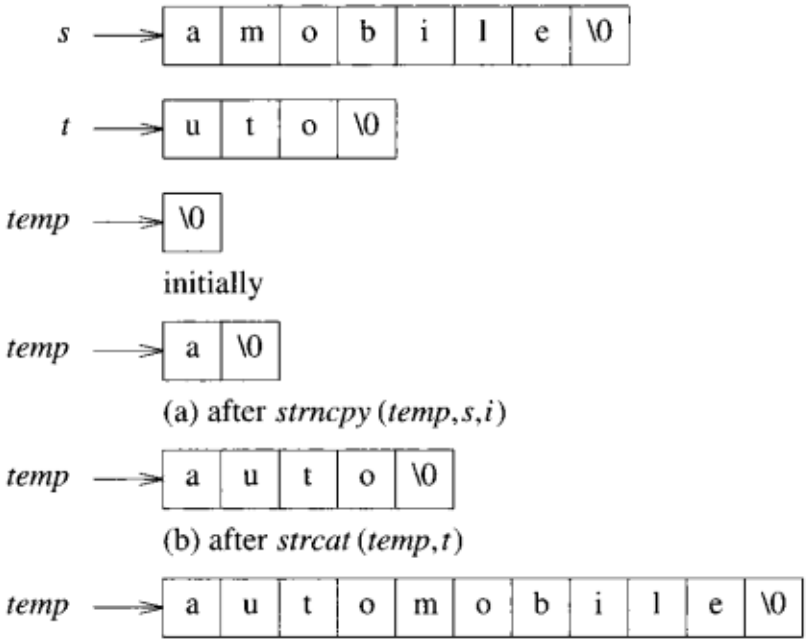


Figure 2.9: String insertion example

Program 2.11 inserts one string into another. This particular function is not normally found in <*string.h*>. Since either of the strings could be empty, we also include statements that check for these conditions. It is worth pointing out that the call *strnins* (*s*, *t*, 0) is equivalent to *strcat* (*t*, *s*). Program 2.11 is presented as an example of manipulating strings. It should never be used in practice as it is wasteful in its use of time and space. Try to revise it so the string *temp* is not required. □

2.6.2 Pattern Matching

Now let us develop an algorithm for a more sophisticated application of strings. Assume that we have two strings, *string* and *pat*, where *pat* is a pattern to be searched for in *string*. The easiest way to determine if *pat* is in *string* is to use the built-in function *strstr*. If we have the following declarations:

```
void strnins(char *s, char *t, int i)
{
/* insert string t into string s at position i */
   char string[MAX_SIZE], *temp = string;

   if (i < 0 && i > strlen(s)) {
      fprintf(stderr,"Position is out of bounds \n");
      exit(1);
   }
   if (!strlen(s))
      strcpy(s,t);
   else if (strlen(t)) {
      strncpy(temp, s,i);
      strcat(temp,t);
      strcat(temp, (s+i));
      strcpy(s, temp);
   }
}
```

Program 2.11: String insertion function

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

then we use the following statements to determine if *pat* is in *string*:

```
if (t = strstr(string,pat))
   printf("The string from strstr is: %s\n",t);
else
   printf("The pattern was not found with strstr\n");
```

The call (*t* = *strstr*(*string*,*pat*)) returns a null pointer if *pat* is not in *string*. If *pat* is in *string*, *t* holds a pointer to the start of *pat* in *string*. The entire string beginning at position *t* is printed out.

Although *strstr* seems ideally suited to pattern matching, there are two reasons why we may want to develop our own pattern matching function:

(1) The function *strstr* is new to ANSI C. Therefore, it may not be available with the compiler we are using.

(2) There are several different methods for implementing a pattern matching function. The easiest but least efficient method sequentially examines each character of the string until it finds the pattern or it reaches the end of the string. (We explore this approach in the Exercises.) If *pat* is not in *string*, this method has a computing

time of $O(n \cdot m)$ where n is the length of *pat* and m is the length of *string*. We can do much better than this, if we create our own pattern matching function.

We can improve on an exhaustive pattern matching technique by quitting when *strlen* (*pat*) is greater than the number of remaining characters in the string. Checking the first and last characters of *pat* and *string* before we check the remaining characters is a second improvement. These changes are incorporated in *nfind* (Program 2.12).

```
int nfind(char *string, char *pat)
{
/* match the last character of pattern first, and
then match from the beginning */
   int i,j,start = 0;
   int lasts = strlen(string)-1;
   int lastp = strlen(pat)-1;
   int endmatch = lastp;

   for (i = 0; endmatch <= lasts; endmatch++, start++) {
      if (string[endmatch] == pat[lastp])
         for (j = 0, i = start; j < lastp &&
                     string[i] == pat[j]; i++,j++)
            ;
      if (j == lastp)
         return start; /* successful */
      }
      return -1;
}
```

Program 2.12: Pattern matching by checking end indices first

Example 2.3 [*Simulation of nfind*]: Suppose *pat* = "aab" and *string* = "ababbaabaa." Figure 2.10 shows how *nfind* compares the characters from *pat* with those of *string*. The end of the *string* and *pat* arrays are held by *lasts* and *lastp*, respectively. First *nfind* compares *string* [*endmatch*] and *pat* [*lastp*]. If they match, *nfind* uses i and j to move through the two strings until a mismatch occurs or until all of *pat* has been matched. The variable *start* is used to reset i if a mismatch occurs. □

Analysis of *nfind*: If we apply *nfind* to *string* = "aa $\cdots$ a" and *pat* = "a $\cdots$ ab", then the computing time for these strings is linear in the length of the string $O(m)$, which is certainly far better than the sequential method. Although the improvements we made over the sequential method speed up processing on the average, the worst case computing time is still $O(n \cdot m)$. □

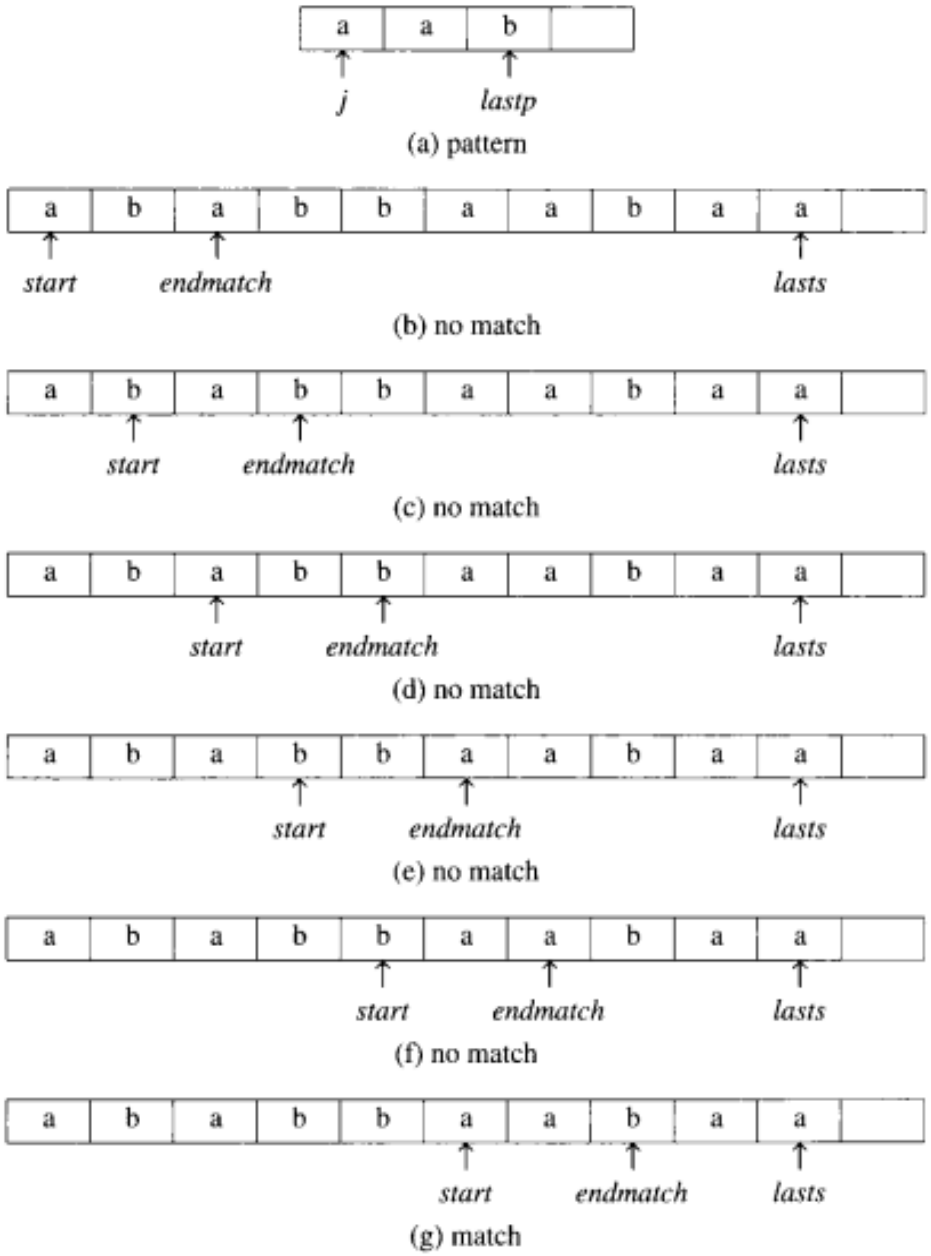


Figure 2.10: Simulation of *nfind*

Ideally, we would like an algorithm that works in O(*strlen* (*string*) + *strlen* (*pat*)) time. This is optimal for this problem as in the worst case it is necessary to look at all characters in the pattern and string at least once. We want to search the string for the pattern without moving backwards in the string. That is, if a mismatch occurs we want to use our knowledge of the characters in the pattern and the position in the pattern where the mismatch occurred to determine where we should continue the search. Knuth, Morris, and Pratt have developed a pattern matching algorithm that works in this way and has linear complexity. Using their example, suppose

$$pat = \text{`}a\ b\ c\ a\ b\ c\ a\ c\ a\ b\text{'}$$

Let $s = s_0\ s_2\ \cdots\ s_{m-1}$ be the string and assume that we are currently determining whether or not there is a match beginning at s_i. If $s_i \neq a$ then, clearly, we may proceed by comparing s_{i+1} and a. Similarly if $s_i = a$ and $s_{i+1} \neq b$ then we may proceed by comparing s_{i+1} and a. If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation:

$s =$	‘-	a	b	?	?	?	.	.	.	.	?’
$pat =$		‘a	b	c	a	b	c	a	c	a	b’

The ? implies that we do not know what the character in s is. The first ? in s represents s_{i+2} and $s_{i+2} \neq c$. At this point we know that we may continue the search for a match by comparing the first character in *pat* with s_{i+2}. There is no need to compare this character of *pat* with s_{i+1} as we already know that s_{i+1} is the same as the second character of *pat*, b, and so $s_{i+1} \neq a$. Let us try this again assuming a match of the first four characters in *pat* followed by a nonmatch, i.e., $s_{i+4} \neq b$. We now have the situation:

$s =$	‘-	a	b	c	a	?	?	.	.	.	?’
$pat =$		‘a	b	c	a	b	c	a	c	a	b’

We observe that the search for a match can proceed by comparing s_{i+4} and the second character in *pat*, b. This is the first place a partial match can occur by sliding the pattern *pat* towards the right. Thus, by knowing the characters in the patterm and the position in the pattern where a mismatch occurs with a character in s we can determine where in the pattern to continue the search for a match without moving backwards in s. To formalize this, we define a failure function for a pattern.

Definition: If $p = p_0 p_1\ \cdots\ p_{n-1}$ is a pattern, then its *failure function*, f, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1\ \cdots\ p_i = p_{j-i} p_{j-i+2}\ \cdots\ p_j \text{ if such an } i \geq 0 \text{ exists} \\ -1 \qquad\qquad \text{otherwise} \end{cases}$$ □

For the example pattern, *pat* = *abcabcacab*, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
f	−1	−1	−1	0	1	2	3	−1	0	1

From the definition of the failure function, we arrive at the following rule for pattern matching: *If a partial match is found such that* $s_{i-j} \cdots s_{i-1} = p_0 p_1 \cdots p_{j-1}$ *and* $s_i \neq p_j$ *then matching may be resumed by comparing* s_i *and* $p_{f(j-1)+1}$ *if* $j \neq 0$. *If* $j = 0$, *then we may continue by comparing* s_{i+1} *and* p_0. This pattern matching rule translates into function *pmatch* (Program 2.13). The following declarations are assumed:

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];
```

```
int pmatch(char *string, char *pat)
{
/* Knuth, Morris, Pratt string matching algorithm */
  int i = 0, j = 0;
  int lens = strlen(string);
  int lenp = strlen(pat);
  while ( i < lens && j < lenp ) {
    if (string[i] == pat[j]) {
      i++; j++; }
    else if (j == 0) i++;
        else j = failure[j-1]+1;
  }
  return ( (j == lenp) ? (i-lenp) : -1);
}
```

Program 2.13: Knuth, Morris, Pratt pattern matching algorithm

Note that we do not keep a pointer to the start of the pattern in the string. Instead we use the statement:

```
return ( (j == lenp) ? (i - lenp) : -1);
```

This statement checks to see whether or not we found the pattern. If we didn't find the pattern, the pattern index index j is not equal to the length of the pattern and we return -1. If we found the pattern, then the starting position is i – the length of the pattern.

Analysis of *pmatch*: The **while** loop is iterated until the end of either the string or the pattern is reached. Since i is never decreased, the lines that increase i cannot be executed more than $m = strlen(string)$ times. The resetting of j to $failure[j-1]+1$ decreases the value of j. So, this cannot be done more times than j is incremented by the statement j++ as otherwise, j falls off the pattern. Each time the statement j++ is executed, i is also incremented. So, j cannot be incremented more than m times. Consequently, no statement of Program 2.13 is executed more than m times. Hence the complexity of function *pmatch* is $O(m) = O(strlen(string))$. □

From the analysis of *pmatch*, it follows that if we can compute the failure function in $O(strlen(pat))$ time, then the entire pattern matching process will have a computing time proportional to the sum of the lengths of the string and pattern. Fortunately, there is a fast way to compute the failure function. This is based upon the following restatement of the failure function:

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

(note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$).

This definition yields the function in Program 2.14 for computing the failure function of a pattern.

Analysis of *fail*: In each iteration of the **while** loop the value of i decreases (by the definition of f). The variable i is reset at the beginning of each iteration of the **for** loop. However, it is either reset to -1 (initially or when the previous iteration of the **for** loop goes through the last **else** clause) or it is reset to a value 1 greater than its terminal value on the previous iteration (i.e., when the statement $failure[j] = i+1$ is executed). Since the **for** loop is iterated only $n-1$ (n is the length of the pattern) times, the value of i has a total increment of at most $n-1$. Hence it cannot be decremented more than $n-1$ times. Consequently the **while** loop is iterated at most $n-1$ times over the whole algorithm and the computing time of *fail* is $O(n) = O(strlen(pat))$. □

```
void fail(char *pat)
{
/* compute the pattern's failure function */
  int n = strlen(pat);
  failure[0] = -1;
  for (j=1; j < n; j++) {
    i = failure[j-1];
    while ((pat[j] != pat[i+1]) && (i >= 0))
      i = failure[i];
    if (pat[j] == pat[i+1])
      failure[j] = i+1;
    else failure[j] = -1;
  }
}
```

Program 2.14: Computing the failure function

Note that when the failure function is not known in advance, the time to first compute this function and then perform a pettern match is O(*strlen* (*pat*) + *strlen* (*string*)).

Abstract Data Type Definition of Stack

structure *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer
    Stack CreateS(*max_stack_size*) ::=
                create an empty stack whose maximum size is *max_stack_size*
    Boolean IsFull(*stack*, *max_stack_size*) ::=
                if (number of elements in *stack* == *max_stack_size*)
                return *TRUE*
                else return *FALSE*
    Stack Add(*stack*, *item*) ::=
                if (IsFull(*stack*)) *stack_full*
                else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
                if (*stack* == CreateS(*max_stack_size*))
                return *TRUE*
                else return *FALSE*
    Element Delete(*stack*) ::=
                if (IsEmpty(*stack*)) **return**
                else remove and return the *item* on the top of the stack.

Structure 3.1: Abstract data type *Stack*

```
void add(int *top, element item)
{
/* add an item to the global stack */
   if (*top >= MAX_STACK_SIZE-1) {
      stack_full();
      return;
   }
   stack[++*top] = item;
}
```

Program 3.1: Add to a stack

```
element delete(int *top)
{
/* return the top element from the stack */
   if (*top == -1)
      return stack_empty(); /* returns an error key */
   return stack[(*top)--];
}
```

Program 3.2: Delete from a stack

STACKS AND QUEUES

STACKS

"A **stack** is an **ordered list** in which insertions (pushes) and deletions (pops) are made at oneend called the **top**."

Given a stack **S= (a_0, ... ,a_{n-1})**, where **a_0** is the bottom element, **a_{n-1}** is the top element, and **a_i** ison top of element **a_{i-1}**, **0 < i < n**.

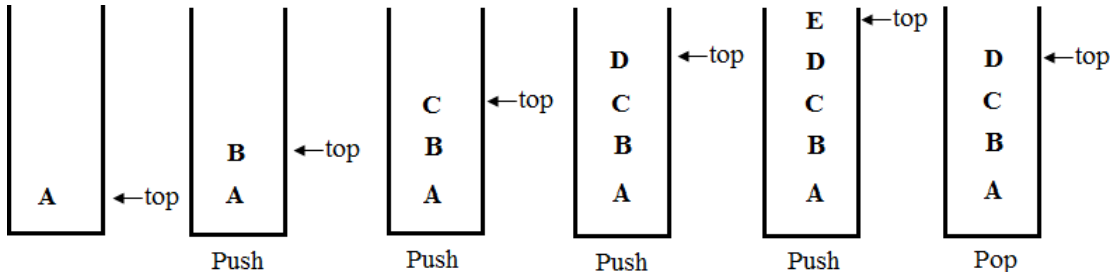


Figure: Inserting and deleting elements in a stack

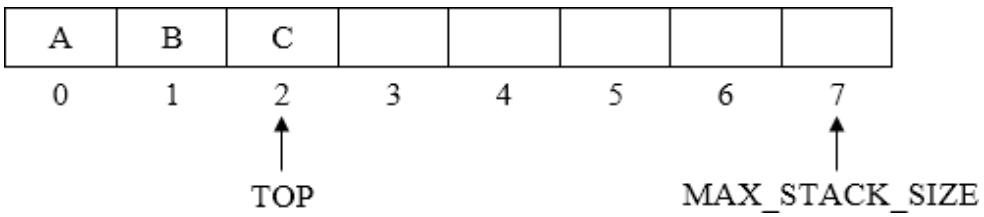
As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then **E** is the first element that is deleted from the stack and the last element is deleted from stack is **A**. Figure illustrates this sequence of operations.

Since the last element inserted into a stack is the first element removed, a stack is also known as a **Last-In-First-Out (LIFO)** list.

ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list(Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP= -1, then itindicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored instack.

Stack can represented using linear array as shown below



STACK OPERATIONS

Implementation of the stack operations as follows.

1. Stack Create

```
Stack CreateS(maxStackSize )::=
    #define MAX_STACK_SIZE 100 /* maximum stack size*/
    typedef struct
        {
            int key;
                        /* other fields */
        } element;


    element stack[MAX_STACK_SIZE];
    int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The I**sEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

4. Push( )

Function **push** checks whether stack is full. If it is, it calls stackFull( ), which prints an error message and terminates execution. When the stack is not full, increment top and assign item to stack [top].

```
    void push(element item)
        { /* add an item to the global stack */
            if (top >= MAX_STACK_SIZE-1)
                stackFull();
            stack[++top] = item;
        }
```

5. Pop( )

Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop ( )
        { /*delete and return the top element from the stack */
                if (top == -1)
                        return stackEmpty(); /*returns an error key */
                return stack[top--];
        }
```

6. stackFull( )

The **stackFull** which prints an error message and terminates execution.

```
void stackFull()
        {
                fprintf(stderr, "Stack is full, cannot add element");
                exit(EXIT_FAILURE);
        }
```

STACKS USING DYNAMIC ARRAYS

The array is used to implement stack, but the bound (MAX_STACK_SIZE) should be known during compile time. The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

Stack Operations using dynamic array

1. Stack CreateS( )::=

```
typedef struct
        {
                int key;        /* other fields */
        } element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity= 1;
int top= -1;
```

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= capacity-1;

4. push()

Here the MAX_STACK_SIZE is replaced with **capacity**

```
void push(element item)
        {    /* add an item to the global stack */
             if (top >= capacity-1)
                  stackFull();
             stack[++top] = item;
        }
```

5. pop( )

In this function, no changes are made.

```
element pop ( )
        {    /* delete and return the top element from the stack */
             if (top == -1)
                  return stackEmpty(); /* returns an error key */
             return stack[top--];
        }
```

6. stackFull( )

The new code shown below, attempts to increase the **capacity** of the array **stack** so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be.

In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
        REALLOC (stack, 2*capacity*sizeof(*stack));
        capacity *= 2;
}
```

Stack full with array

doubling<u>Analysis</u>

In the **worst case**, the realloc function needs to allocate **2*capacity*sizeof (*stack)** bytes of memory and copy **capacity *sizeof (*stack))** bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity). Initially, capacity is 1.

Suppose that, if all elements are pushed in stack and the capacity is 2k for some k, k>O, then the total time spent over all array doublings is **O ($\sum_{i=1}^{k} 2i$) = O(2^{k+1}) = O(2^k)**.

Since the total number of pushes is more than **2k-1**, the total time spend in array doubling is **O(n)**, where **n** is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of push over all **n** pushes is **O(n).**

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

X = a / b – c + d * e – a * c

In above expression contains operators (+, –, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

Example: A + B

Here, **A** & **B** are operands and + is operand

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: + A B

Here, **A** & **B** are operands and + is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: A B +

Here, **A** & **B** are operands and + is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a =4, b =c =2, d =e =3 in below expression

X = a / b – c + d * e – a * c

((4/2)-2) + (3*3)-(4*2)
=0+9-8
=1

OR

(4/ (2-2 +3)) *(3-4)*2
= (4/3) * (-1) * 2
= -2.66666

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

Token	Operator	Precedence	Associativity
( ) [ ] →	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
= = !=	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression **a * b / c % d / e** is equivalent to **((((a * b) / c) % d) / e)**
- Parentheses are used to override precedence, and expressions are always evaluated from theinnermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: a/b -c +d*e -a*c
Fully parenthesized : ((((a/b)-c) + (d*e))-a*c))

: a b / e – d e * + a c *

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.

The expression **a*(b +c)*d** which results **abc +*d*** in postfix. Figure shows the translation process.

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.
- The left parenthesis complicates matters because it behaves like a low-precedence operatorwhen it is on the stack and a high-precedence one when it is not. It is placed inthe stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence**

(icp).

The declarations that establish the precedence's are:

```
/* isp and icp arrays-index is value of precedence lparen rparen, plus, minus, times, divide,
mod, eos */
int isp[] = {0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

```
void postfix(void)
{
        char symbol;
        precedence token;
        int n = 0,top = 0; /* place eos on stack */
        stack[0] = eos;
        for (token = getToken(&symbol, &n); token != eos; token =
              getToken(&symbol,& n ))
                {
                        if (token == operand)
                                printf("%c", symbol);
                        else if (token == rparen)
                                {
                                        while (stack[top] != lparen)
                                                printToken(pop( ));
                                        pop( );
                                }
                        else{
                                while(isp[stack[top]] >= icp[token])
                                        printToken(pop());
                                push(token);
                             }
                }
        while((token = pop ())!= eos)
                printToken(token);
        printf("\n");
}
```

Program: Function to convert from infix to postfix

Analysis of postfix: Let **n** be the number of tokens in the expression. Θ (n) time is spent extracting tokens and outputting them. Time is spent in the **two while loops,** is Θ (n) as the number of tokens that get stacked and unstacked is linear in **n**. So, the complexity of function postfix is **Θ (n).**

EVALUATION OF POSTFIX EXPRESSION

- The evaluation process of postfix expression is simpler than the evaluation of infix expressions because there are no parentheses to consider.
- To evaluate an expression, make a single **left-to-right** scan of it. Place the operands on a stack until an operator is found. Then remove from the stack, the correct number of operands for the operator, perform the operation, and place the result back on the stack and continue this fashion until the end of the expression. We then remove the answer from the top of the stack.

```
int eval(void)
    {
        precedence token;
        char symbol;
        int opl,op2, n=0;
        int top= -1;
        token = getToken(&symbol, &n);
        while(token! = eos)
            {
                if (token == operand)
                    push(symbol-'0'); /* stack insert */
                else {
                    op2 = pop(); /* stack delete */
                    opl = pop();
                switch(token) {
                                    case plus:      push(opl+op2);
                                                    break;
                                    case minus:     push(opl-op2);
                                                    break;
                                    case times:     push(opl*op2);
                                                    break;
                                    case divide:    push(opl/op2);
                                                    break;
                                    case mod:       push(opl%op2);
                                }


                                }
                    token = getToken(&symbol, &n);
            }
        return pop(); /* return result */
    }
```

Program: Function to evaluate a postfix expression

```
precedence getToken(char *symbol, int *n)
    {
        *symbol = expr[(*n)++];
        switch (*symbol)
            {
                case '(' : return lparen;
                case ')' : return rparen;
                case '+' : return plus;
                case '-' : return minus;
                case '/' : return divide;
                case '*' : return times;
                case '%' : return mod;
                case ' ' : return eos;
                default: return operand;
            }
    }
```

Program: Function to get a token from the input string

- The function **eval ( )** contains the code to evaluate a postfix expression. Since an operand (symbol) is initially a character, convert it into a single digit integer.
- To convert use the statement, **symbol-'0'**. The statement takes the ASCII value of **symbol** and subtracts the ASCII value of '0', which is 48, from it. For example, suppose **symbol = '1.** The character '1' has an ASCII value of 49. Therefore, the statement **symbol-'0'** producesas result the number 1.
- The function **getToken( )**, obtain tokens from the expression string. If the token is an operand, convert it to a number and add it to the stack. Otherwise remove two operands from the stack, perform the specified operation, and place the result back on the stack. Whenthe end of expression is reached, remove the result from the stack.

RECURSION

A recursive procedure

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a recursive procedure. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

1. There must be certain criteria, called base criteria, for which the procedure does not callitself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to thebase criteria.

Recursive procedure with these two properties is said to be well-defined.

A recursive function

A function is said to be recursively defined if the function definition refers to itself. A recursive function must have the following two properties:

1. There must be certain arguments, called ***base values***, for which the function does notrefer to itself.
2. Each time the function does refer to itself, the argument of the function must be closerto a ***base value***

A recursive function with these two properties is also said to be well-defined.

Factorial Function

"The product of the positive integers from 1 to n, is called "n factorial" and is denoted by n!"

$$n! = 1*2 * 3 \ldots (n - 2)*(n - 1)*n$$

It is also convenient to define 0! = 1, so that the function is defined for all nonnegative integers.

Definition: (Factorial Function)

a) If n = 0, then n! = 1.
b) If n > 0, then n! = n*(n - 1)!

Observe that this definition of n! is recursive, since it refers to itself when it uses (n - 1)!

(a) The value of n! is explicitly given when n = 0 (thus 0 is the base value)

(b) The value of n! for arbitrary *n* is defined in terms of a smaller value of *n* which is closer to the base value 0.

The following are two procedures that each calculate n factorial .

1. Using for loop: This procedure evaluates N! using an iterative loop process

Procedure: FACTORIAL (FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT: = 1, and Return.
2. Set FACT: = 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.
   Set FACT: = K*FACT.
   [End of loop.]
4. Return.

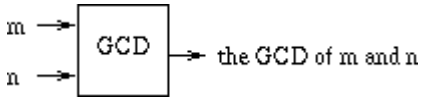
2. Using recursive function: This is a recursive procedure, since it contains a call to itself

Procedure: FACTORIAL (FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT: = 1, and Return.
2. Call FACTORIAL (FACT, N - 1).
3. Set FACT: = N*FACT.
4. Return.

GCD

The **greatest common divisor** (GCD) of two integers *m* and *n* is the greatest integer that divides both *m* and *n* with no remainder.



$$\text{For } m \geq n \geq 0,\ \underline{\gcd}(m, n) = \begin{cases} n & \text{If n divides m with no minimum} \\ \gcd(n, remainder\ of\ \frac{m}{n}) & \text{Otherwise} \end{cases}$$

Procedure: GCD (M, N)

1. If (M % N) = 0, then set GCD=N and RETURN
2. Call GCD (N, M % N)
3. Return

Fibonacci Sequence

The Fibonacci sequence (usually denoted by F0, F1, F2.) is as follows:

0, 1, 1, 2,3,5,8, 13, 21, 34, 55

That is, F0 = 0 and F1 = 1 and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci Sequence)

a) If n = 0 or n = 1, then $F_n = n$

b) If n > 1, then $F_n = F_{n-2} + F_{n-1}$

Here

(a) The base values are 0 and 1

(b) The value of F_n is defined in terms of smaller values of ***n*** which are closer to the base values.

A procedure for finding the n^{th} term F_n of the Fibonacci sequence follows.

Procedure: FIBONACCI (FIB, N)

This procedure calculates F_N and returns the value in the first parameter FIB.

1. If N = 0 or N = 1, then: Set FIB: = N, and Return.
2. Call FIBONACCI (FIBA, N - 2).
3. Call FIBONACCI (FIBB, N - I).
4. Set FIB: = FIBA + FIBB.
5. Return.

Tower of Hanoi

Problem description

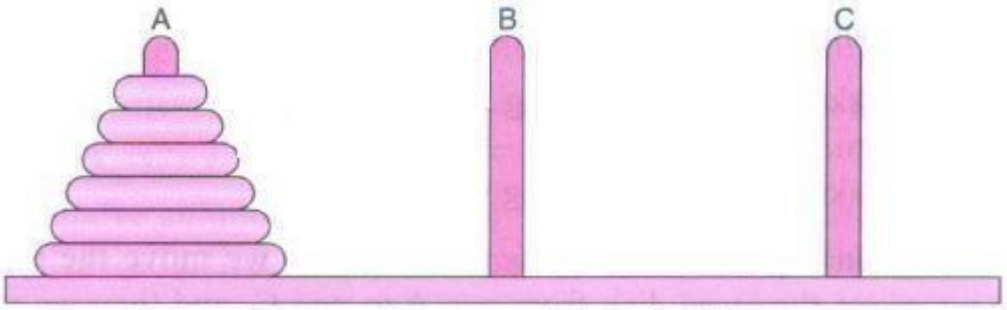
Suppose three pegs, labeled A, Band C, are given, and suppose on peg A a finite number ***n*** of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are as follows:

1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to any other peg.

2. At no time can a larger disk be placed on a smaller disk.

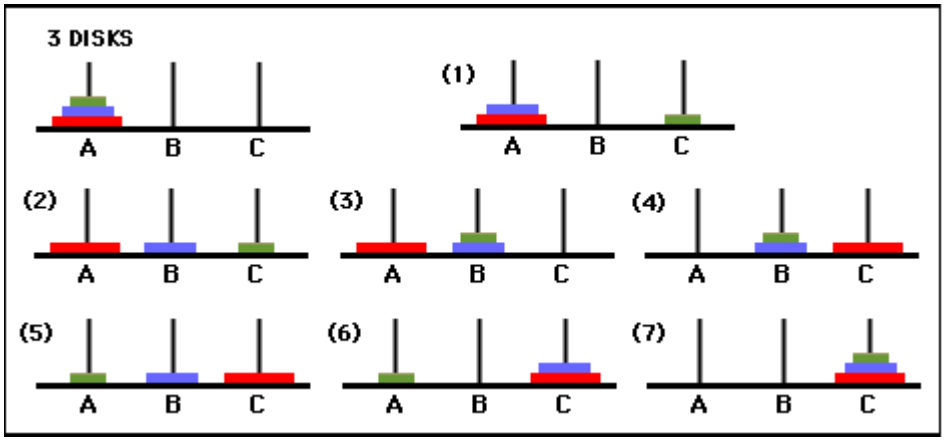


Initial Setup of Towers of Hanoi with n = 6

We write A→B to denote the instruction "Move top disk from peg A to peg B"

Example: Towers of Hanoi problem for n = 3.

Solution: Observe that it consists of the following seven moves



1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,
n=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C

For completeness, the solution to the Towers of Hanoi problem for n = 1 and n = 2
n=l: A→C
n=2: A→B, A→C, B→C

The Towers of Hanoi problem for n > 1 disks may be reduced to the following sub-problems:

(1) Move the top n - 1 disks from peg A to peg B

(2) Move the top disk from peg A to peg C: A→C.

(3) Move the top n - 1 disks from peg B to peg C.

The general notation

- TOWER (N, BEG, AUX, END) to denote a procedure which moves the top n disks fromthe initial peg BEG to the final peg END using the peg AUX as an auxiliary.

- When n = 1, the solution:
  TOWER (1, BEG, AUX, END) consists of the single instruction BEG→END

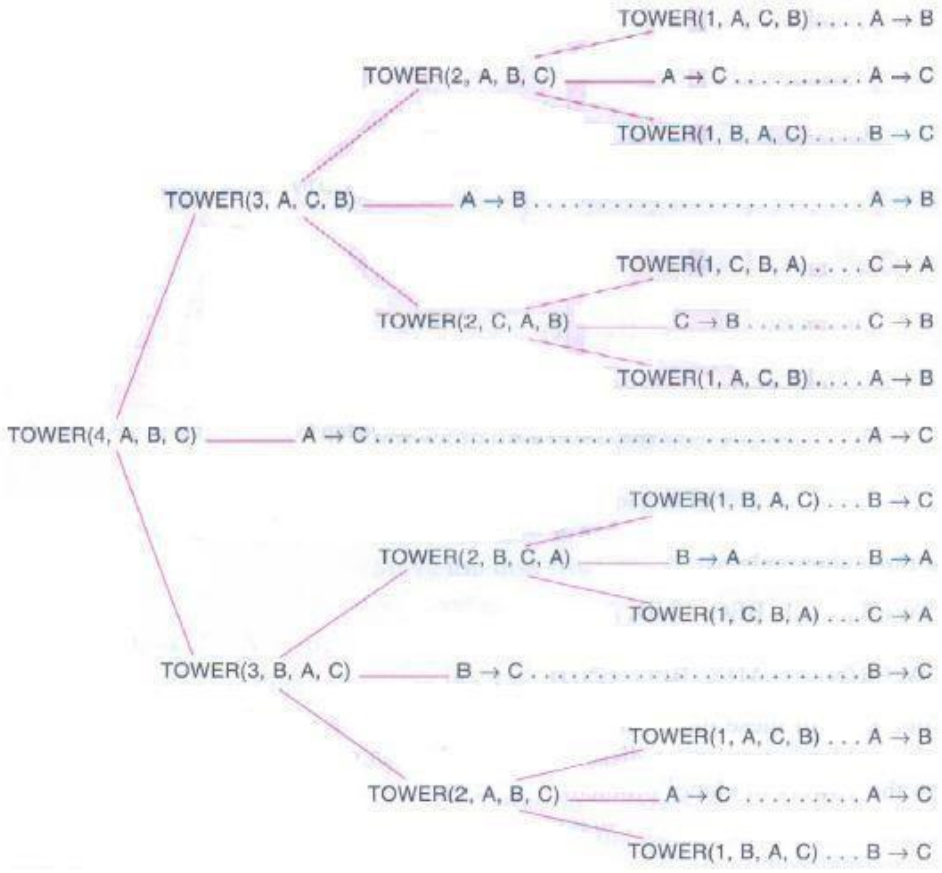
- When n > 1, the solution may be reduced to the solution of the following three sub-problems:
  (a) TOWER (N - I, BEG, END, AUX)
  (b) TOWER (I, BEG, AUX, END) or BEG → END
  (c) TOWER (N - I, AUX, BEG, END)

Procedure: TOWER (N, BEG, AUX, END)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N=I, then:
   (a) Write: BEG → END.
   (b) Return.

   [End of If structure.]
2. [Move N - 1 disks from peg BEG to peg AUX.]
   Call TOWER (N - 1, BEG, END, AUX).
3. Write: BEG → END.
4. [Move N - 1 disks from peg AUX to peg END.]Call
   TOWER (N - 1, AUX, BEG, END).
5. Return.

Example: Towers of Hanoi problem for n = 4



Ackermann function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2, ...

Definition: (Ackermann Function)

(a) If m = 0, then A (m, n) = n + 1.

(b) If m ≠ 0 but n = 0, then A(m, n) = A(m - 1, 1)

(c) If m ≠ 0 and n ≠ 0, then A(m, n) = A(m - 1, A(m, n - 1))

DYNAMIC MEMORY ALLOCATION FUNCTIONS

1. malloc( ):

The function *malloc* allocates a user- specified amount of memory and a pointer to the start of the allocated memory is returned.
If there is insufficient memory to make the allocation, the returned value is NULL.
Syntax:

```
data_type *x;
x= (data_type *) malloc(size);
```

Where,

x is a pointer variable of data_type
size is the number of bytes

Ex:

```
int *ptr;
ptr = (int *) malloc(100*sizeof(int));
```

2. calloc( ):

The function *calloc* allocates a user- specified amount of memory and initializes the allocated memory to **0** and a pointer to the start of the allocated memory is returned.
If there is insufficient memory to make the allocation, the returned value is NULL.

Syntax:

```
data_type *x;
x= (data_type *) calloc(n, size);
```

Where,

x is a pointer variable of type int
n is the number of block to be allocated
size is the number of bytes in each block

Ex: int *x

x= calloc (10, sizeof(int));

The above example is used to define a one-dimensional array of integers. The capacity of thisarray is n=10 and x [0: n-1] (x [0, 9]) are initially 0

Macro CALLOC

```
#define CALLOC (p, n, s)\
if ( ! ((p) = calloc (n, s)))\
        {\
                fprintf(stderr, "Insuffiient
                memory");\exit(EXIT_FAILURE);\
        }\
```

3. realloc( ):

- Before using the realloc( ) function, the memory should have been allocated using malloc( ) or calloc( ) functions.
- The function relloc( ) resizes memory previously allocated by either ***mallor*** or ***calloc***, which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.
- If the existing allocated memory cannot be extended, the function allocates a new block andcopies the contents of existing memory block into new memory block and then deletes theold memory block.
- When realloc is able to do the resizing, it returns a pointer to the start of the new block andwhen it is unable to do the resizing, the old block is unchanged and the function returns thevalue NULL

Syntax:

```
data_type *x;
x= (data_type *) realloc(p, s );
```

The size of the memory block pointed at by p changes to S. When **s > p** the additional **s-p** memory block have been extended and when **s < p**, then **p-s** bytes of the old block are freed.

Macro REALLOC #define
REALLOC(p,S)\if (!((p) =
realloc(p,s))) \

```
{ \
        fprintf(stderr, "Insufficient
        memory");\exit(EXIT_FAILURE);\
}\
```

4. free( )

Dynamically allocated memory with either malloc( ) or calloc ( ) does not return on its own. The programmer must use free( ) explicitly to release space.

Syntax:

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated