1.1 What is an algorithm?

An algorithm is a finite set of step by step instructions to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows:

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Properties

Every algorithm must satisfy the following properties:

1. **Definiteness** - Every step in an algorithm must be clear and unambiguous
2. **Finiteness –** Every algorithm must produce result within a finite number of steps.
3. **Effectiveness -** Every instruction must be executed in a finite amount of time.
4. **Input & Output** - Every algorithm must take zero or more number of inputs and must produce at least one output as result.

1.2 Performance Analysis

In computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyses them and pick the one which is best suitable for our requirements. Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.

Performance analysis of an algorithm is performed by using the following measures:

1. Space Complexity
2. Time Complexity

1.2.1 What is Space complexity?

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

NOTE: When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack. That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

Consider the following piece of code...

```
int square(int a)
{
        return a*a;
}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value. That means, totally it requires 4 bytes of memory to complete its execution.

1.2.2 What is Time complexity?

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. Generally, running time of an algorithm depends upon the following:

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine
3. Read and Write speed of the machine.
4. The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,

NOTE: When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

Consider the following piece of code...

```
Algorithm Search (A, n, x)
{       // where A is an array, n is the size of an array and x is the item to be searched.
   for i := 1 to n  do
   {
      if(x=A[i]) then
        {
          write (item found at location i)
           return;
        }
    }
    write (item not found)
}
```

For the above code, time complexity can be calculated as follows:
Cost is the amount of computer time required for a single operation in each line. Repetition is the amount of computer time required by each operation for all its repetitions, so above code requires '*n*' units of computer time to complete the task.

1.2.3 Asymptotic Notation

Asymptotic notation of an algorithm is a mathematical representation of its complexity. Majorly, we use THREE types of Asymptotic Notations and those are:

1. Big - Oh (O)
2. Omega (Ω)

3. Theta (Θ)

Big - Oh Notation (O)

- ✓ Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- ✓ Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- ✓ Big - Oh notation describes the worst case of an algorithm time complexity.
- ✓ It is represented as O(T)

Omega Notation (Ω)

- ✓ Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- ✓ Omega notation always indicates the minimum time required by an algorithm for all input values.
- ✓ Omega notation describes the best case of an algorithm time complexity.
- ✓ It is represented as **Ω** (T)

Theta Notation (Θ)

- ✓ Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.
- ✓ Theta notation always indicates the average time required by an algorithm for all input values.
- ✓ Theta notation describes the average case of an algorithm time complexity.
- ✓ It is represented as **Θ** (T)

Example

Consider the following piece of code...

```
Algorithm Search (A, n, x)
{       // where A is an array, n is the size of an array and x is the item to be searched.
   for i := 1 to n  do
   {
      if(x=A[i]) then
        {
          write (item found at location i)
           return;
        }
   }
   write (item not found)
}
```

The time complexity for the above algorithm

1. Best case is **Ω** (1)
2. Average case is **Θ** (n/2)
3. Worst case is **O** (n)

1.3 What is Data Structure?

Data may be organized in many different ways: The logical or mathematical model of a particular organization of data is called *data structure*. Data structures are generally classified into primitive and non- primitive data structures.

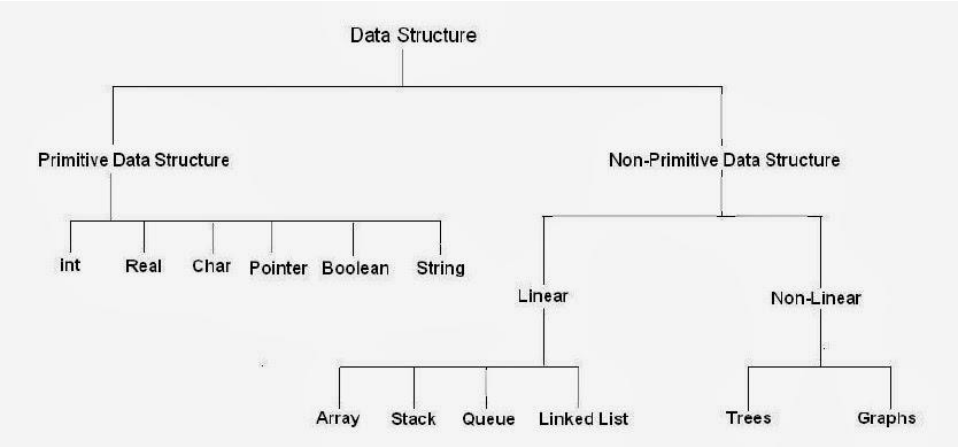


Fig. Classification of Data Structures

Based on the organizing method of a data structure, data structures are divided into two types.

1. Linear Data Structures
2. Non - Linear Data Structures

Linear Data Structures

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure. For example:

1. Arrays
2. Lists (Linked List)
3. Stacks
4. Queues

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure. For example:

1. Trees
2. Graphs
3. Dictionaries
4. Heaps , etc.,

Operations on Data Structures

The basic operations that are performed on data structures are as follows:

1. **Traversal:** Traversal of a data structure means processing all the data elements present in it exactly once.
2. **Insertion:** Insertion means addition of a new data element in a data structure.
3. **Deletion:** Deletion means removal of a data element from a data structure if it is found.
4. **Searching:** Searching involves searching for the specified data element in a data structure.
5. **Sorting:** Arranging data elements of a data structure in a specified order is called sorting.
6. **Merging:** Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

1.4 Abstract Data Type (ADT)

An Abstract Data type refers to set of data values and associated operations that are specified accurately, independent of any particular implementation.

(Or)

ADT is a user defined data type which encapsulates a range of data values and their functions.

(Or)

An **Abstract Data Type** is a mathematical model of a **data structure**. It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.

Advantages:

- ✓ Code is easier to understand.
- ✓ Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.
- ✓ ADTs can be reused in future programs.

ADT Model

The ADT model is shown in below figure. It consists of two different parts:

1. Public part
2. Private part

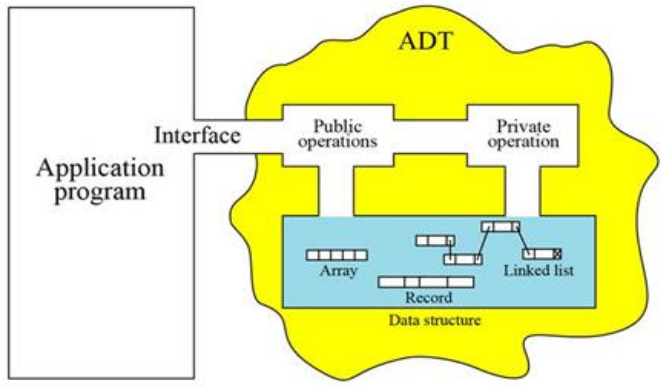


Figure : The model for an ADT

The **public** or **external** part, which consists of:

- ✓ The conceptual picture (the user's view of how the object looks like, how the structure is organized)
- ✓ The conceptual operations (what the user can do to the ADT)

The **private** or **internal** part, which consists of:

- ✓ The representation (how the structure is actually stored).
- ✓ The implementation of the operations (the actual code)

1.5 Array as an ADT

The **array** is a basic abstract data type that holds an ordered collection of items accessible by an integer index. Since it's an ADT, it doesn't specify an implementation, but is almost always implemented by an array data structure or dynamic array.

1.5.1 Linear Array

An array is collection of homogeneous elements that are represented under a single variable name.

(Or)

A linear array is a list of a finite number of *n* homogeneous data elements (that is data elements of the same type) such that

- ✓ The elements are referenced respectively by an index set consisting of *n* consecutive numbers
- ✓ The elements are stored respectively in successive memory locations
- ✓ The number *n* of elements is called the *length* or *size* of the array.
- ✓ The index set consists of the integer 0,1, 2, … n-1.
- ✓ Length or the number of data elements of the array can be obtained from the index set by

  Length = UB – LB + 1

  where UB is the largest index called the upper bound and LB is the smallest index called the lower bound of the arrays

- ✓ Element of an array A may be denoted by
  - Subscript notation $A_1, A_2, , \ldots , A_n$
  - Parenthesis notation A(1), A(2), …. , A(n)
  - Bracket notation A[1], A[2], ….. , A[n]
  - The number K in A[K] is called subscript or an index and A[K] is called a subscripted variable

1.5.2 Representation of linear array in memory

- ✓ Let LA be a linear array in the memory of the computer.
- ✓ LOC(LA[K]) = address of the element LA[K] of the array LA
- ✓ The elements of LA are stored in the successive memory locations.

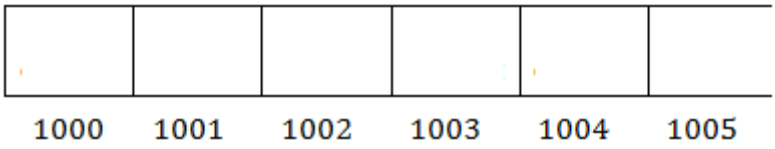


Fig. memory representation of an array of elements

Computer does not need to keep track of the address of every element of LA, but need to track only the address of the first element of the array denoted by

Base(LA)

and called the *base address* of LA. Using this address, the computer calculates the address of any element of LA by the following formula:

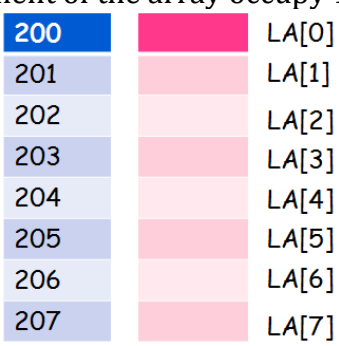
LOC(LA[K]) = Base(LA) + w(K – LB)

where **w** is the number of words per memory cell of the array LA [**w** is the size of the **data type**] .

Example: Find the address for LA [6]. Each element of the array occupy 1 byte

LOC(LA[K]) = Base(LA) + w(K – lower bound)

LOC(LA[6]) = 200 + 1(6 – 0) = 206



1.6 Operations on Arrays

The operation performed on any linear structure, where it can be an array or linked list, include the following

1. **Traversal:** processing each element in the list exactly once.
2. **Insertion:** adding new element to the list.
3. **Deletion:** removing an element from the list.
4. **Searching:** finding location of the element with a given value or key.
5. **Sorting:** Arranging elements in some type of order.
6. **Merging:** Combining two lists into a single list.

1.6.1 Traversing

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or count the number of elements of A with a given property. This can be accomplished by *traversing* A, that is, by accessing and processing (frequently called *visiting*) each element of A exactly once.

The following algorithm traverses a linear array:

Algorithm: (Traversing a linear array.)

Here, A is a linear array with lower bound LB and upper bound UB. This algorithm traverses A applying an operation PROCESS to each element of A.

Step 1: [Initialize counter] Set I:= LB.
Step 2: Repeat steps 3 and 4 while I<= UB:
Step 3: [Visit element.] Apply PROCESS to A.
Step 4: [Increase counter.] Set I:= I+1.
[End of step 2 loop.]
Step 5: Exit.

Implementation of array traversing using C.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int i, a[5]={12,45,-10,3,28};
        clrscr();
        printf("\n The array elements are ");
        for(i=0;i<5;i++)
        printf("\t %d", arr[i]);
        getch();
}
```

Output

The array elements are 12 45 -10 3 28

1.6.2 Insertion

Let A be a collection of data elements in the memory of the computer. “Inserting” refers to the operation of adding another element to the collection A,

Inserting an element at the ‘end’ of a linear array can be easily done. On the other hand, suppose we need to insert an element in the middle of the array. Then on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep an order of the order of the other elements.

Algorithm: (Inserting into a linear array.)
INSERT (A, N, K, ITEM).
Here, A is a linear array with N elements and K is a positive integer such that K <= N. This algorithm inserts an element ITEM into the K^{th} position in A.

Step 1: [Initialize counter.] Set I:= N.
Step 2: Repeat steps 3 and 4 while I >= K:
Step 3: [Move element downward.] Set A[I+1] := A[I].
Step 4: [Decrease counter.] I:= I-1.
[End of step 2 loop.]
Step 5: [Insert element.] Set A[K] := ITEM.
Step 6: [Reset N.] Set N:= N+1.
Step 7: Exit.

Implementing array insertion algorithm using C.

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

void main()
 {
  int a[100], n,element,i, pos;
  clrscr();
  printf("\nEnter size of an array:");
  scanf("%d", &n);
  printf("\nEnter elements :");
  for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
  printf("\nEnter the element to be inserted :");
  scanf("%d", &element);

  printf("\nEnter the position :");
  scanf("%d", &pos);

  //Create space at the specified location
  for (i = n; i >= pos; i--)
  {
    a[i] = a[i - 1];
  }

  n++;
  a[pos - 1] = element;

  //Print out the result of insertion
printf("\nResultant Array: ");
 for (i = 0; i < n; i++)
    printf("  %d", a[i]);
  getch();
}
```

Output
Enter size of an array : 5
Enter elements: 1 2 3 4 5
Enter the element to be inserted : 6
Enter the location : 2
Resultant Array: 1 6 2 3 4 5

1.6.3 Deletion

Let A be a collection of data elements in the memory of the computer. "Deleting" refers to the operation of removing one of the elements from A.

Deleting an element at the 'end' of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element is moved one location upward in order to 'fill up' the array.

Algorithm: (Deletion from a linear array.)

DELETE (A, K, N, ITEM).

Here, A is a linear array with N elements and K is a positive integer such that K<= N. This algorithm deletes the K^{th} element from A.

Step 1: [Initialize counter.] Set ITEM:= A[K] and I:= K.
Step 2: Repeat steps 3 and 4 while I>N:
Step 3: [Move element upward.] A[I] := A[I+1].
Step 4: [Increase counter.] Set I:= I+1.
[End of step 2 loop.]
Step 5: [Reset N.] Set N:= N-1.
Step 6: Exit.

Implementing array deletion algorithm using C

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a[100], n, i, pos;
  clrscr();
  printf("\nEnter size of an array:");
  scanf("%d", &n);
  printf("\nEnter elements :");          //Read elements in an array
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
  printf("\n Enter position of the element to be deleted :");          //Read the position
  scanf("%d", &pos);
  while(pos<n)                    /* loop for the deletion  */
  {
        a[pos] = a[pos+1];
        pos++;
  }
  n--;   // No of elements reduced by 1
  printf("\nResultant Array: ");
  for (i = 0; i < n; i++)              //Print Array
    printf("  %d", a[i]);
  getch();
}
```

Output

```
Enter size of an array : 5
Enter elements: 4 8 16 12 5
Enter position of element to be
deleted : 2
Resultant Array:  4 8 12 5
```

1.6.4 Sorting

Sorting means arranging the elements of an array in specific order may be either ascending or descending. There are different types of sorting techniques are available:

1. Bubble sort
2. Selection sort
3. Insert sort
4. Merge sort
5. Quick sort etc.

Bubble sort, sometimes referred to as *sinking sort*, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Let A be a list of 'n' numbers. Sorting A refers to the operation of re-arranging the elements of A, so they are in increasing order, i.e. so that

A[1]< A[2]< A[3]< …….. <A[N]

For example, Suppose A originally is the list 8, 4, 19, 2, 7, 13, 5, 16. After sorting, A is the list 2, 4, 5, 7, 8, 13, 16, 19.

Algorithm: (Bubble Sort.)
BUBBLE (A, N).
Here, A is an array with N elements. This algorithm sorts the elements in A.

Step 1: Repeat steps 2 to 4 for I=1 to N-1.
Step 2: Set J:= 1. [Initializes pass pointer J.]
Step 3: Repeat step 4 while J <= N-I: [Execute pass.]
Step 4: If A[J] > A[J+1], then:
[Swap the elements.]
TEMP:= A[J]
A[J]:= A[J+1]
A[J+1] := TEMP
[End of If structure.]
Step 5: Set PTR:= PTR+1. [Increase pointer.]
[End of step 3 (inner) loop.]
[End of step 1 (outer) loop.]
Step 6: Exit.

```
/ * C Program to implement Bubble Sort Technique  */
#include <stdio.h>
#include <conio.h>
void main()
{
   int a[100],n,i,j,temp;
   printf("Enter number of elements:");
   scanf("%d", &n);
   printf("Enter elements\n");
   /* Read array elements */
   for(i=0;i<n;i++)
   {
        printf("Enter a[%d]=",i);
        scanf("%d", &a[i]);
```

Output

```
Enter number of elements: 5
Enter elements
Enter a[0] = 14
Enter a[1] = 5
Enter a[2] = 23
Enter a[3] = 9
Enter a[4] = 15
Sorted elements....
5       9       14      15      23
```

```
  }
  /* bubble sort logic starts from here */
  for(i=0;i<n-1;i++)
  {
   for(j=i+1;j<n;j++)
         if(a[i]>a[j])          //> for ascending order, < for descending order
         {
          temp=a[i];
          a[i] = a[j];
          a[j] = temp;
         }
  }
  printf("Sorted elements ....\n");
  for(i=0;i<n;i++)
    printf("%3d",a[i]);
  printf("\n");
  getch();
}
```

1.6.5 Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be *successful* and the searching process gives the location of that value in the array. If the value is not present in the array, the searching is said to be *unsuccessful*. There are two popular methods for searching the array elements:

1. Linear search
2. Binary search.

1.6.5.1 Linear Search

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. It is mostly used to search an unordered list of elements. For example, if an array *a[]* is declared and initialized as,

int a[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

and the value to be searched is *VAL = 7*, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm

LINEAR SEARCH (A, N, ITEM, LOC).

Here, A is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in A, or sets LOC := -1 if the search is unsuccessful.

Step 1: [Initialize Location.] Set LOC := -1.
Step 2: [Initialize Counter.] Set I:= 0.
Step 3: Repeat steps 3 and 4 while I<N [Search for ITEM.]
IF A[I] = ITEM
Set := LOC := I.
Write(Search successful or ITEM found)

[End of IF.]
Step 4: Set I:= I+1.
[End of Step 2 Loop]
Step 5: Write (Search unsuccessful or ITEM not found)
Step 6: Exit.

```
/* C Program to search an element in an array using the linear search  */
#include <stdio.h>
#include <conio.h>
void main()
{
   int a[100],n,i,key,flag=0;
   printf("Enter number of elements:");
   scanf("%d", &n);
   printf("Enter elements\n");
   /* Read array elements */
   for(i=0;i<n;i++)
   {
         printf("Enter a[%d]=",i);
         scanf("%d", &a[i]);
   }
   printf("Enter an element to be searched:");
   scanf("%d", &key);
   /* linear search starts here */
   for(i=0;i<n;i++)
   {
      if(key==a[i])
      {
         printf("%d is found at position %d\n", key, i);
             flag=1;
         break;
      }
   }
   if(flag==0)
         printf("%d is not found\n",key);
   getch();
}
```

Output

```
Enter number of elements: 5
Enter elements
Enter a[0] = 14
Enter a[1] = 5
Enter a[2] = 23
Enter a[3] = 9
Enter a[4] = 15
Enter an element to be searched: 9
9 is found at position 3
```

1.6.5.2 Binary Search

Suppose A is an array which is sorted in increasing numerical order or, equivalently alphabetically. Then, there is an extremely efficient searching algorithm, called 'binary search', which can be used to find the location LOC of a given ITEM of information in A. It works efficiently with a sorted list.

Algorithm: BINARY SEARCH (A, LB, UB, ITEM, LOC).

Here, A is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variable BEG, END, and MID denote respectively, the beginning, end and middle locations of a segment of elements of A. This algorithm finds the location LOC of ITEM in A or sets LOC = -1.

Step 1: [Initialize Location.] SET LOC = -1
Step 2: [Initialize Bounds.]
Set BEG := LB, END := UB and MID := (BEG+END)/2.
Step 3: Repeat steps 4 and 5 while BEG <= END and A[MID] != ITEM
Step 4: Set MID := (BEG+END)/2).
Step 5: If A[MID] = ITEM then
Set LOC := MID
Else If A[MID] > ITEM then
Set END := MID-1.
Else
Set BEG := MID+1.
[End of If structure.]
[End of step 2 loop.]
Step 6: If LOC := -1 then
Write (Search Unsuccessful or element not found).
Else
Write (Search Successful or element found at LOC).
Step 7: Exit.

```
/* C Program to search an element in an array using binary search. */
#include <stdio.h>
#include <conio.h>
void main()
{
        int a[25],i, n, key, high, low, mid, flag=0;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements\n");
        for(i=0;i<n;i++)
        {
                printf("Enter a[%d]=",i);
                scanf("%d", &a[i]);
        }
        printf("Enter the element to be searched: ");
        scanf("%d", &key);
        low = 0, high = n-1;
        while(high>=low)
        {
                mid = (low + high)/2;
                if (a[mid] == key)
                {
                        printf("\n %d is found at position %d", key, mid);
                        flag = 1;
                        break;
                }
                else if (a[mid]>key)
                        high = mid-1;
                else
                        low = mid+1;
        }
```

Output

```
Enter the number of elements in the array: 5
Enter the elements
Enter a[0] = 11
Enter a[1] = 26
Enter a[2] = 32
Enter a[3] = 49
Enter a[4] = 68
Enter the element to be searched: 49
49 is found at position 3
```

```
        if (flag == 0)
                printf("\n %d does not found in the array", key);
        getch();
}
```

1.7 Representation of polynomials using arrays

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

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

A polynomial thus may be represented using arrays. A single dimensional array is used for representing a single variable polynomial. The index of such array can be considered as an exponent and coefficient can be stored at that particular index. The array representation for the above polynomial expression is given below:

arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

Advantages:

1. Easy to handle

Drawbacks

1. It is time consuming
2. Wastage of memory space
3. We can change the size of an array

Algorithm for Addition of two polynomials

Assume that there are two polynomials P and Q. The polynomial result is stored in third array SUM.

Step 1: While P and Q are not null, repeat step 2.

Step 2: If powers of the two terms are equal then

    If the terms do not cancel then
        insert the sum of the terms into the SUM Polynomial
        Advance P
        Advance Q
    Else if the power of the first polynomial> power of second Then
        insert the term from first polynomial into SUM polynomial
        Advance P
    Else insert the term from second polynomial into SUM polynomial
        Advance Q

Step 3: Copy the remaining terms from the non empty polynomial into the SUM polynomial.

Step 4: Print Sum

Step 5: Exit

/* Implementation of polynomial addition algorithm using C */

```
include <stdio.h>
#include <conio.h>
void main()
{
   int a[6],b[6],c[6],i,flag=0;
```

```
   clrscr();
   for (i=0;i<6;i++)
          a[i]=b[i]=0;

  a[1]= 1; a[2]=4; a[5]=-7;

  b[0]= -14; b[1]=10;
  b[2]=6; b[3]= 5; b[4]=3;

printf ("Polynomial one is : ");          //Printing first polynomial
for (i=5;i>=0;i--)
{
    if(a[i]!=0 && i>0)
    {
          printf ("%d(x)%d ",a[i],i);
          flag=1;
    }
    if(i>0 && a[i-1]>0 && flag==1)
         printf ("+ ");
    if(a[i]!=0 && i==0)
          printf ("%d",a[i]);
}
flag=0;
printf ("\nPolynomial two is : ");     //Printing second polynomial
for (i=5;i>=0;i--)
{
     if(b[i]!=0 && i>0)
     {
          printf ("%d(x)%d ",b[i],i);
          flag=1;
     }
     if(i>0 && b[i-1]>0 && flag==1)
          printf ("+ ");
     if(b[i]!=0 && i==0)
          printf ("%d",b[i]);
}
for (i=0;i<6;i++)        //polynomial addition
          c[i] = a[i]+b[i];

printf ("\nResultant Polynomial is : ");       //Printing Resultant polynomial
for (i=5;i>=0;i--)
{
    if(c[i]!=0 && i>0)
    {
          printf ("%d(x)%d ",c[i],i);
          lag=1;
     }
     if(i>0 && c[i-1]>0 && flag==1)
          printf ("+ ");
     if(c[i]!=0 && i==0)
```

```
        printf ("%d",c[i]);
}
getch();
}
```

Output

Polynomial one is : -7(x)5 + 4(x)2 +1(x)1
Polynomial two is : -3(x)4 + 5(x)3 + 6(x)2 + 10(x)1 – 14
Resultant polynomial is : -7(x)5 + 3(x)4 + 5(x)3 + 10(x)2 + 11(x)1 – 14

1.8 Multi-Dimensional Arrays

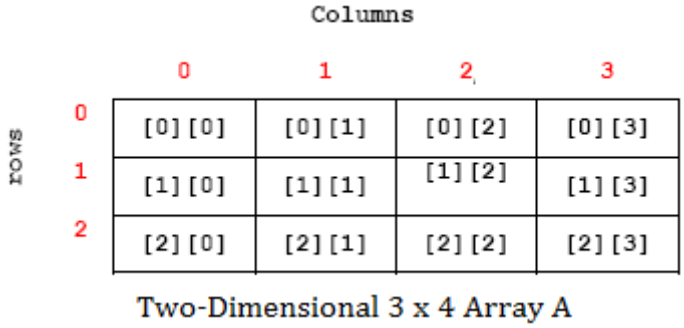
1.8.1 Two-Dimensional Array

A Two-Dimensional **m x n** array **A** is a collection of **m . n** data elements such that each element is specified by a pair of integers (such as J, K) called subscripts with property that

$$\mathbf{1 \le J \le m} \text{ and } \mathbf{1 \le K \le n}$$

The element of **A** with first subscript **J** and second subscript **K** will be denoted by $\mathbf{A_{J,K}}$ or **A[J][K]**

The two-dimensional arrays are also called matrices in mathematics and tables business applications. Hence 2d arrays are sometimes called matrix arrays.
The standard way of representing 2-d arrays:



Two-Dimensional 3 x 4 Array A

Storage Representations

Let *A* be a two-dimensional m x n array. The array A will be represented in the memory by a block of m x n sequential memory locations. Programming language will store array A either

1. Column-Major Order
2. Row-Major Order

If a two-dimensional array can be represented as a single row with many columns and mapped sequentially is known as column-major representation.

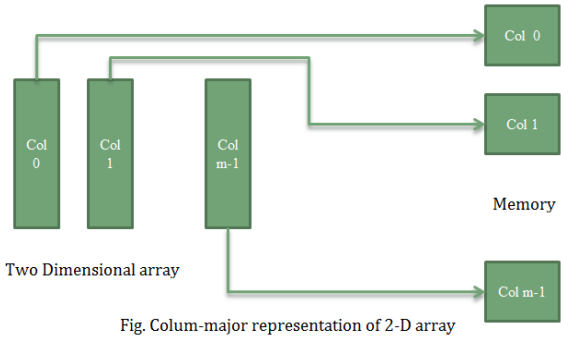


Fig. Colum-major representation of 2-D array

If a two-dimensional array can be represented as a single column with many rows and mapped sequentially is known as row-major representation.

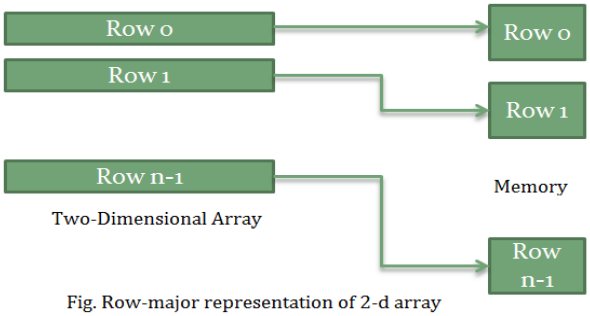
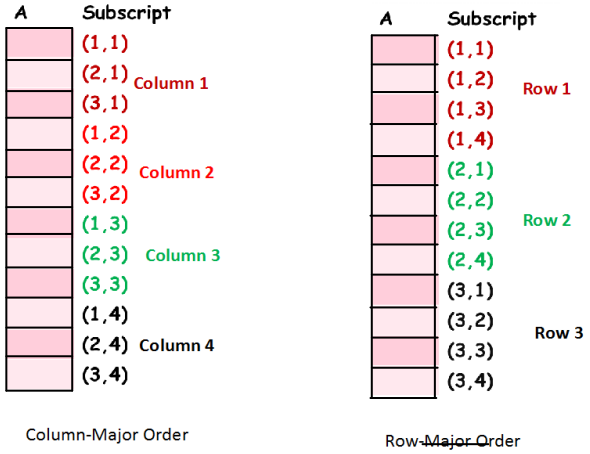


Fig. Row-major representation of 2-d array

Representation of 2-D Arrays in memory

Let A be a two-dimensional m x n array. The array A will be represented in the memory by a block of m x n sequential memory location



1.8.2 Multi dimensional arrays

An **n**-dimensional m_1 x m_2 x …. x m_n array **B** is a collection of $m_1.m_2...m_n$ data elements in which each element is specified by a list of **n** integers indices – such as K_1, K_2, …., K_n – called subscript with the property that

$1 \le K_1 \le m_1$, $1 \le K_2 \le m_2$, …. $1 \le K_n \le m_n$

The Element **B** with subscript K_1, K_2, …,K_n will be denoted by

$B_{K1,K2,...,Kn}$ or $B[K_1,K_2,....,K_n]$

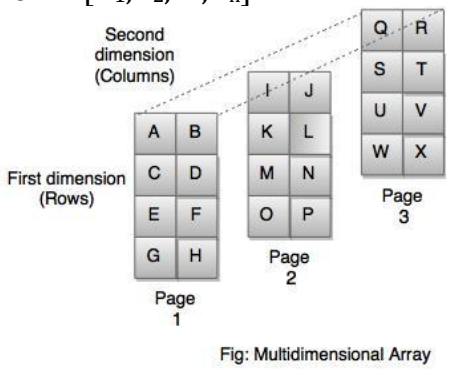


Fig: Multidimensional Array

1.8.3 Matrix multiplication

The following algorithm finds the product AB of matrices A and B. which are stored in two – dimensional arrays.

Algorithm: MATMUL(AB,C,M,P,N)

Let A be an MxP matrix array , and Let B be an PxN matrix array. This algorithm stores the product of A and B in MxN matrix array C.

```
Step 1:  Repeat steps 2 to 4 for I = 1 to M:
Step 2:          Repeat steps 3 to 4 for J = 1 to N:
Step 3:                  Set C[I,J]:= 0    [Initialize C[I,J]
Step 4:                  Repeat for K = 1 to P
                                 C[I,J]:= C[I,J] + A[I,K] * B[K,J]
                         [End of inner loop]
                 [End of step 2 middle loop]
         [End of step 1 outer loop]
Step 5: Exit.
```

Example: Write a C program to perform multiplication of two matrices

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int a[10][10],b[10][10],c[10][10];
   int m,n,p,q,i,j,k;
   clrscr();
   printf("Enter number of rows and columns of first matrix(between 1 and 10):");
   scanf("%d%d",&m,&n);
   printf("Enter number of rows and columns of second matrix(between 1 and 10):");
   scanf("%d%d",&p,&q);
   if(n==p)
   {
         /* Read the elements of first matrix */
         printf("Enter elements of first matrix\n");
         for(i=0;i<m;i++)
         {
              for(j=0;j<n;j++)
                   scanf("%d",&a[i][j]);
          }
         /* Read the elements of second matrix */
         Print f("Enter elements of second matrix\n");
         for(i=0;i<p;i++)
         {
              for(j=0;j<q;j++)
                   scanf("%d",&b[i][j]);
          }
     /* Display A and B matrices */
     printf("The Matrix A\n");
     for(i=0;i<m;i++)
     {
           for(j=0;j<n;j++)
              printf("%3d",a[i][j]);
           printf("\n");
     }
     printf("The Matrix B\n");
     for(i=0;i<p;i++)
     {
```

```
            for(j=0;j<q;j++)
              printf("%3d",b[i][j]);
            printf("\n");
      }
      /*multiply two matrices and print resultant matrix */
      printf("The resultant matrix is\n");
      for(i=0;i<m;i++)
      {
            for(j=0;j<q;j++)
            {
              c[i][j]=0;              /* Initializing resultant matrix elements to zero */
              for(k=0;k<p;k++)
                    c[i][j] += a[i][k]*b[k][j];
              printf("%3d",c[i][j]);
            }

            printf("\n");
      }
    }
    else
     printf("Multiplication is not possible\n");
    getch();
}
```

Output

```
Enter number of rows and columns of first matrix (between 1 and 10):2 3
Enter number of rows and columns of second matrix (between 1 and 10):3 2
Enter elements of first matrix: 1 2 3  4 5 6
Enter elements of second matrix: 1 2 3  4  5 6
The Matrix A
1   2   3
4   5   6
The Matrix B
1   2
3   4
5   6
The resultant matrix is
22   28
49   64
```

1.9 Sparse Matrix

Matrices with a relatively high proportion of zero entries are called sparse matrices. Two general types of n-square sparse matrices occur in various applications. They are:

1. Triangular matrix
2. Tridiagonal matrix

In *triangular matrix*, all entries above the main diagonal are zero or equivalently, where nonzero entries can only occur on or below the main diagonal.

In *tridiagonal matrix*, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal.

$$\begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 4 & & \\ 3 & 4 & 1 & \\ & 2 & 3 & 4 \\ & & 1 & 3 \end{pmatrix}.$$

(a) Triangular matrix (b) Tridiagonal matrix

Sparse Matrix Representations

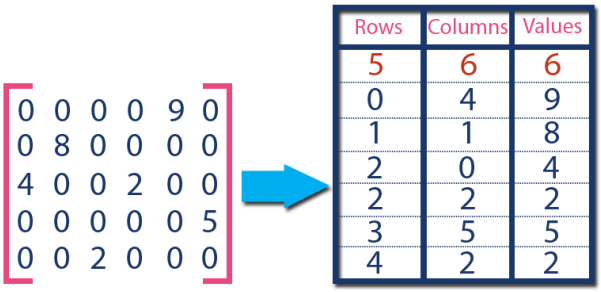
A sparse matrix can be represented by using two representations, those are:

1. Triplet Representation
2. Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the table:



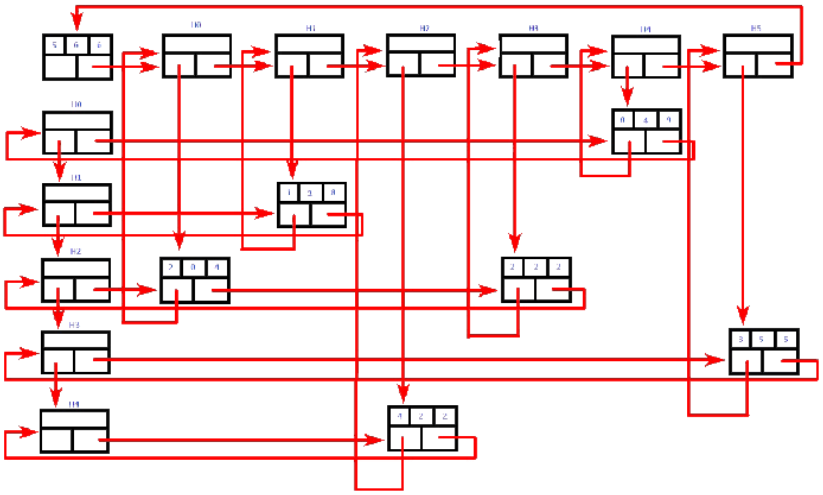
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above table. Here the first row in the right side table is filled with values 5, 6 & 6 which indicate that it is a sparse matrix with 5 rows, 6 columns and 6 non-zero values. Second row is filled with 0, 4, and 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the fig.

Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below figure.



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

Example : C Program for transposing a sparse matrix

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int s[9][3], t[9][3];
  int r, c, nzvs, i;
  clrscr();
  printf("\nEnter no. of rows, cols and non-zero elements:");
  scanf("%d%d%d",&r,&c,&nzvs);
  s[0][0]=r;
  s[0][1]=c;
  s[0][2]=nzvs;

  for(i=1;i<=nzvs;i++)                    //reading sparse matrix
  {
        printf("Enter the next triplet(row,column,value):");
        scanf("%d%d%d",&s[i][0],&s[i][1],&s[i][2]);
  }

 printf("\n******Sparse matrix*********\n");      //printing sparse matrix
 printf("\nrow\t\tcolumn\t\tvalue\n");
 for(i=0;i<=nzvs;i++)
        printf("%d\t\t%d\t\t%d\n",s[i][0],s[i][1],s[i][2]);
```

```
  for(i=0;i<=nzvs;i++)                    //Transposing sparse matrix
  {
        t[i][0]=s[i][1];
        t[i][1]=s[i][0];
        t[i][2]=s[i][2];
  }
        //Printing transposed sparse matrix
  printf("\n*********After Transpose***********\n");
  printf("\nrow\t\tcolumn\t\tvalue\n");
  for(i=0; i<=nzvs; i++)
        printf("%d\t\t%d\t\t%d\n",t[i][0],t[i][1],t[i][2]);
  getch();
}
```

Output

```
Enter no. of rows, cols and non-zero elements:6 6 8
Enter the next triplet(row, column, value): 0 0 15
Enter the next triplet(row, column, value): 0 3 22
Enter the next triplet(row, column, value): 0 5 -15
Enter the next triplet(row, column, value): 1 1 11
Enter the next triplet(row, column, value): 1 2 3
Enter the next triplet(row, column, value): 2 3 -6
Enter the next triplet(row, column, value): 4 0 91
Enter the next triplet(row, column, value): 5 2 28

******Sparse matrix*********
row     column          value
 6          6               8
 0          0              15
 0          3              22
 0          5              -15
 1          1              11
 1          2               3
 2          3              -6
 4          0              91
 5          2              28

******After Transpose*********
row     column          value
 6          6               8
 0          0              15
 3          0              22
 5          0              -15
 1          1              11
 2          1               3
 3          2              -6
 0          4              91
 2          5              28
```