Data Structure

Introduction

In the modern world, Data and its information is an essential part, and various implementations are being made to store in different ways. Data are just a collection of facts and figures, or you can say data are values or a set of values that are in a particular format. A data item refers to a single set of values. Data items are then further categorized into sub-items which are the group of items which are not being called a plain elementary form of items.

What is Data Structure?

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a specific data model depends on the two factors: -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real-world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Data structure mainly specifies the following four things:

1. Organization of data.
2. Accessing methods.
3. Degree of associativity.
4. Processing alternatives for information.

Categories of Data Structure

The data structure can be subdivided into major types:

1. **Linear Data Structure:** A data structure is said to be linear if its elements combine to form any specific order. There are two techniques of representing such linear structure within memory.

The common examples of the linear data structure are:

- The first way is to provide the linear relationships among all the elements represented using linear memory location. These linear structures are termed as arrays.

- The second technique is to provide a linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

2. **Nonlinear Data Structure**:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non-Linear Data Structures are listed below:

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as a rooted tree graph or a tree.

Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such a data structure is termed as a Graph.

Difference between Linear and Non Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

Data structure study covers the following points:

1. Amount of memory require to store.
2. Amount of time require to process.
3. Representation of data in memory.
4. Operation performed on the data.

How to choose the suitable data structure:

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

1- Data size and the required memory.
2- The dynamic nature of the data.
3- The required time to obtain any data element from the data structure.

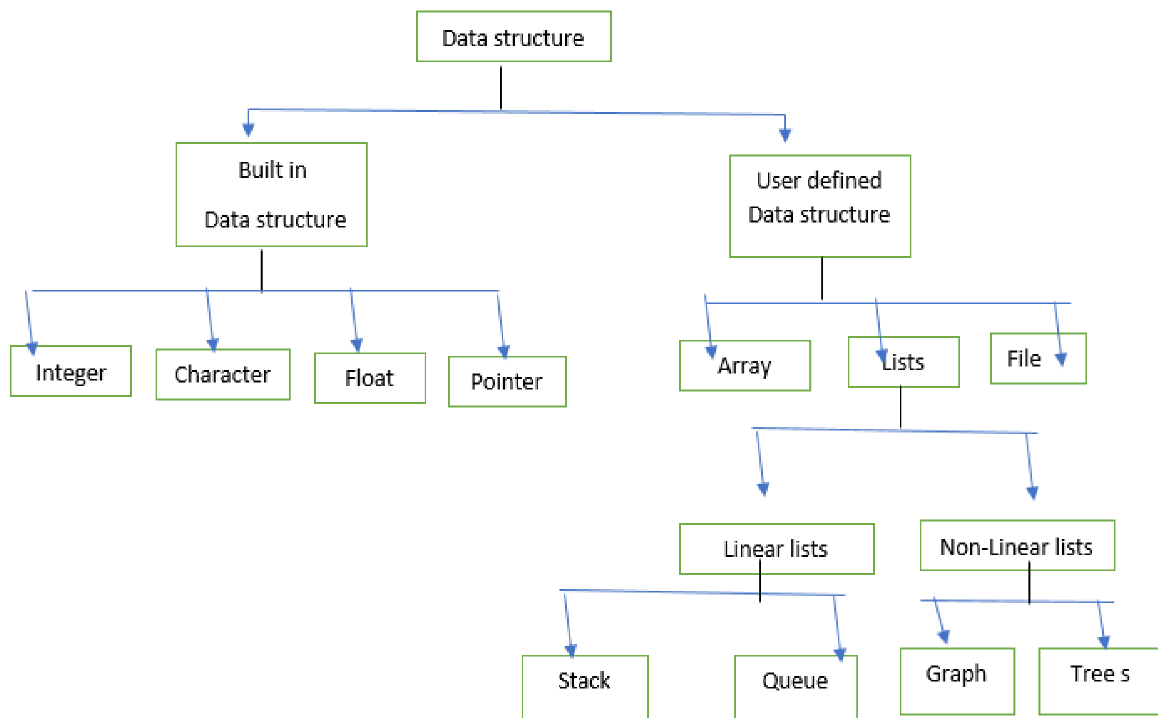
4- The programming approach and the algorithm that will be used to manipulate these.

Algorithm : an algorithm states explicitly how data will be manipulated.

Algorithm +data structure =program.

Difference between algorithm and program:

Algorithm	program
1. Semantic	1. Syntax
2. Human language	2. Machine language



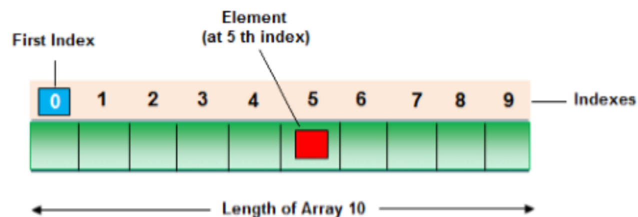
Arrays

1. One dimensional array (vector)

An array consists of a set of objects (called its elements), all of which are of the same type and are arranged contiguously in memory.

In general, only the array itself has a symbolic name, not its elements. Each element is identified by an index which denotes the position of the element in the array. The number of elements in an array is called its dimension. The dimension of an array is fixed and predetermined; it cannot be changed during program execution. Arrays are suitable for representing composite data which consist of many similar, individual items.

Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account. An array variable is defined by specifying its dimension and the type of its elements.



For example, an array representing 10 height measurements (each being an integer quantity) may be defined as:

```
int heights[10];
```

The individual elements of the array are accessed by indexing the array. The first array element always has the index 0. Therefore, heights[0] and heights[9] denote, respectively, the first and last element of heights. Each of heights elements can be treated as an integer variable. So, for example, to set the third element to 177, we may write:

```
heights[2] = 177;
```

Attempting to access a nonexistent array element (e.g., heights[-1] or heights[10]) leads to a serious runtime error (called 'index out of bounds' error).

Processing of an array usually involves a loop which goes through the array element by element.

The following example illustrates using a function which takes an array of integers and returns the average of its elements.

```
const int size = 3;
double Average (int nums[size])
{
  double average = 0;
for (i = 0; i < size; ++i)
average += nums[i];
return average/size;
 }
```

Array Initialization

An array may have an initializer. Braces are used to specify a list of comma separated initial values for array elements.

For example,

int nums[3] = {5, 10, 15}; initializes the three elements of nums to 5, 10, and 15, respectively.

When the number of values in the initializer is less than the number of elements, the remaining elements are initialized to zero:

int nums[3] = {5, 10};

When a complete initializer is used, the array dimension becomes redundant, because the number of elements is implicit in the initializer. The first definition of nums can therefore be equivalently written as:

int nums[] = {5, 10, 15}; // no dimension needed

example: write c++ program to find average:

```
1    #include <iostream.h>
2        main ( )
3        {
4        int stud[10]  ,total=0 , i ;
5        float Avrege;
6        cout <<  "Please Enter all  grades of stud:\n" ;
7        for (i=0 ; i<10 ; i++)
8        {
9        cout << "grade number" << i+1 << endl;
10     cin >> stud[i] ;
```

```
11      total=total+stud[ i ] ;
12       }
13       Avrege=total /10;
14       cout << "The Avrege of  all student is:  " << Avrege ;
15       return 0;
16       }
```

2. Two dimensional Arrays (matrix)

An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmer's perceived organization of the elements is different,

for Example

int seasonTemp[3][4];

The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each.

Multidimensional Arrays Initialization

The array may be initialized using a nested initializer:

int seasonTemp[3][4] = { {26, 34, 22, 17}, {24, 32, 19, 13}, {28, 38, 25, 10} };

Because this is mapped to a one-dimensional array of 12 elements in memory, it is equivalent to:

int seasonTemp[3][4] = {26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 10};

Multidimensional Arrays Processing

Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop.

Example 1

```
const int rows = 3;

const int columns = 4;

int seasonTemp[rows][columns] = { {26, 34, 22, 17}, {24, 32, 19, 13}, {28,
38, 25, 10} };

 int HighestTemp (int temp[rows][columns])

{

 int highest = 0;

for (i = 0; i < rows; ++i)

 for (j = 0; j < columns; ++j)

if (temp[i][j] > highest) highest = temp[i][j];

return highest;

 }
```

Pointers and References

A pointer is simply the address of a variable in memory. Generally, variables can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to a variable via a pointer to it, is called dereferencing the pointer. Pointer variables are defined to point to variables of a specific type so that when the pointer is dereferencing, a typed variable is obtained.

• Pointers are useful for creating dynamic variables during program execution.

• Unlike normal (global and local) variables which are allocated storage on the runtime stack, a dynamic variable is allocated memory from a different storage area called the heap.

• Dynamic variables do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

• A pointer variable is defined to 'point to' data of a specific type. For example:

```
int  *ptr1; // pointer to an int

char  *ptr2; // pointer to a char
```

• The value of a pointer variable is the address to which it points. For example, given the definitions

```
    int   num; • we can write:

 ptr1 = &num;
```

• The symbol & is the address operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of num is assigned to ptr1. Therefore, we say that ptr1 points to num.

Pointer Values • Given that ptr1 points to num, the expression

*ptr1

• Dereferences ptr1 to get to what it points to, and is therefore equivalent to num. The symbol * is the dereference operator; it takes a pointer as argument and returns the contents of the location to which it points.

• In general, the type of a pointer must match the type of the data it is set to point to.

• Regardless of its type, a pointer may be assigned the value 0 (called the null pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

Pointer Arithmetic

In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is not the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an int is represented by 4 bytes. Now, given

```
char   *str = "HELLO";

int   nums[] = {10, 20, 30, 40};

int   *ptr = &nums[0]; // pointer to first element
```

str++ advances str by one char (i.e., one byte) so that it points to the second character of "HELLO", whereas ptr++ advances ptr by one int (i.e., four bytes) of that it points to the second element of nums.

CODE

```
1        #include <iostream.h>
2        main( )
3        {
4        int employee[3][5] , size[4] , i , j , sum=0 ;
5        size [3]=0;
6       cout << "Please Enter all employees salary" << endl;
7
8       for (j=0 ; j < 3 ; j ++ )
9       {      cout << " Enter the department " << j+1 << endl;
10          for (i=0 ; i < 5 ; i++ )
11                  {
12                    cout << " Employee number " << i+1 << endl;
13                   cin >> employee[ i ] [ j ] ;
14                   sum= employee[ i ] [ j ] + sum ;
15                  }
16             size[i] = sum/5 ;
17               cout << " The avreg is" << size[i] << endl;
18              cout <<"________________________________";
18             size[3] = size[3] + size [i];
19             sum=0
20             }
21        cout << " The avrege of all salary of employee is: " <<
size[3] << endl;
22     return 0;
23     }
```

CODE

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.     int array[5]={50,32,93,2,74};
7.     int sure=0;
8.     int x=0;
9.     cout << "Here is the Array befor sorted\n";
10.         for (int j=0;j<5;j++)
11.             cout << array[j] << endl;
12.
13.         for (int i=0;i<5-1;i++) {
14.             sure=0;
15.             for (int j=i; j<5;j++) {
16.                 if (array[j] <array[i]) {
17.                     x=array[j];
18.                     array[j]=array[i];
19.                     array[i]=x;
20.                     sure=1;
21.                 }
22.             }
23.             if (sure ==0) break;
24.         }
25.
26.         cout << "Here is the Array after sorted\n";
27.         for (i=0;i<5;i++)
28.             cout << array[i] << endl;
29.
30.         return 0;
31.     }
```