Chapter 1

Introduction to Data Structure and algorithms

1. INTRODUCTION

This chapter introduces what data structures are and how data structures are important for algorithms to find the efficient solutions of problems. This chapter also discusses the different types of data structures, abstract data types, different sorts of algorithms and complexity analysis of algorithms.

Data structures are the way to organize data in some logical model. Algorithms use the data structures to store the data or to retrieve the data to accomplish some tasks. Based on the logical way how a data structure organizes the data, the same problem may have slightly or completely different algorithm to solve the same problem. In addition, the ways in which data are organized in data structures determine the number of computations and comparisons to solve the same problem. Therefore, it is very important to choose a right data structure for an algorithm to get an efficient solution for a problem.

A type is a collection of values. For example, the Boolean type consists of the values true and false. A data type is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type. An abstract data type (ADT) is the specification of a data type within some language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation. A data structure is the implementation for an ADT.

2. BASIC TERMINOLOGIES

- Data – They are values or set of values.
- Data Item or Item or Element – It refers to single unit of values.
- Group Items – Data items that are divided into sub items are called as Group Items.
- Elementary Items – Data items that cannot be divided are called as Elementary Items.
- Entity – It is any real world object.

- Attribute – It is a property of an entity which may have a value.
- Entity Set – Entities of similar attributes form an entity set.
- Information – organized form of data which gives some meanings about an entity.
- Field – It is a single elementary unit of information representing an attribute of an entity.
- Record – It is a collection of field values of a given entity.
- File – It is a collection of records of the entities in a given entity set.
- Primary key – it is a field of a record in file and can uniquely identify the record in the file.

3. Data Structures and Algorithms

Let us consider you want to develop a simple system like a google map to find a path from your home to your school. Suppose a small section of the map containing your home, your friend's home, a supermarket, a store and school as shown in Fig. 1. There are streets as well. These streets are named as A, B, C, D, E, F and G. A street line with arrow indicates one way street and street line without arrow indicates a two way street. For example, you can go from your home to you friend's home by street A but you cannot return from friend's home to your home by street A. Using street G, you can go from store to school and school to store as well.

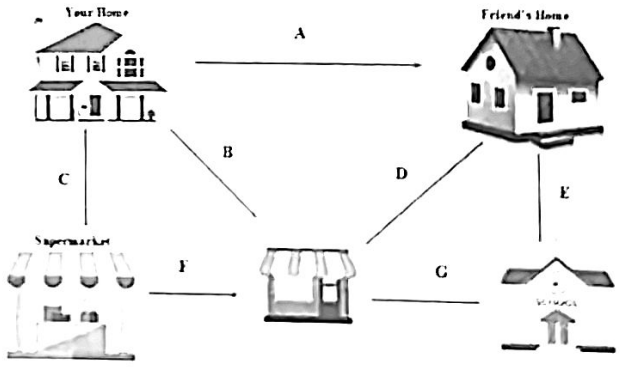


Fig. 1: A section of a map containing your home

Each place has its own coordinates containing latitude and longitude. Suppose the coordinates of your home, your friend's home, supermarket, store and school as shown in Table 1.

Table 1: Coordinates of places

Places	Coordinates
Home	(70.4, -135.8)
Supermarket	(70.6, -135.8)
Store	(70.6, -135.6)
School	(70.6, -135.4)
Friend's home	(70.4, -135.4)

From the information provided in Table 1, you can say where the places are exactly located. But you cannot say how these places are connected with streets and where the streets are exactly. Therefore, now you need to find out some way to store these information in computer. There are different way to store the information. One of the way you can use is to store the information about those places and streets in a list like format as in Fig 2 (a). This is one way to store the data set and it looks like an array or a list structure. You can also use another way to organize and store the same data sets as shown in Fig 2 (b).

(Your Home, Friend's Home)
(Your Home, Store)
(Store, Your Home)
(Your Home, Supermarket)
(Supermarket, Your Home)
(Friend's Home, School)
(School, Friend's Home)
(Friend's Home, Store)
(Store, Friend's Home)
(Supermarket, Store)
(Store, School)
(School, Store)

(a) Data Structure 1 which looks like a list or an array

Your Home	(Friend's Home, Store, Supermarket)
Friend's Home	(Store, School)
Supermarket	(Your Home, Store)
Store	(Your Home, Friend's Home, School)
School	(Friend's Home, Store)

(b) Data Structure 2 which looks like a hash table

Fig. 2: Two ways of organizing the same data set

In this case, the first column contains a place and the second column listed all the other places to which there is a street and you can go from the place. This way of storing the data set looks like a hash table. These two ways are basically two different ways of storing exactly the same set of data. You can see they have different structures. These two ways of storing data are the examples of how data structures look like.

Data can be organized and stored in computer in different ways. A logical or mathematical model of a particular organization of data is called a data structure. Basically, a good data structure should be able to organize or represent data and their relationships as they appear in real world. In addition, it should be able to organize the data in such a way that an algorithm can efficiently and effectively process the data. A data structure provides an efficient way to organize and store data in computer so that an algorithm can efficiently retrieve the data to process. The way of data organization of a data structure determines the performance of an algorithm.

Algorithms use sets of instructions and process some data to accomplish certain tasks. During the processing of data, algorithms retrieves data which are stored in some data structure. In addition, algorithms may need to store necessary data in some sort of data structures.

Let us consider the previous example. Suppose you want to find a shortest path from your home to the school. There are the following six possible paths to go from your home to school:

Path 1: (Your Home, Friend's Home, School)

Path 2: (Your Home, Friend's Home, Store, School)

Path 3: (Your Home, Store, School)

Path 4: (Your Home, Store, Friend's Home, School)

Path 5: (Your Home, Supermarket, Store, Friend's Home, School)

Path 6: (Your Home, Supermarket, Store, School)

For example, you can start from your home and follow the path to your friend's home and to school. Similarly another possible path is from your home to your friend's home then to store and to school. You can trace out the remaining four paths that you can follow to reach school from your home. After finding all possible paths from your home to school, determine the distance/cost of each path. Finally, choose the path with shortest distance. You can calculate the distance from the coordinates of those places- your home, your friend's home, supermarket, store and school. To perform these set of operations to find the shortest path from your home to the school, the algorithm that run in computer will contain the following steps:

ALGORITHM: SHORTEST-PATH

Step 1: Find all possible places you can go from your home.

Step 2: From each place found in step 1, find all possible places you can go from current place.

Step 3: Determine and keep track of distances you have traveled.

Step 4: Repeat the previous steps until you reach school.

Step 5: Compare the distance of all paths you traveled.

Step 6: Choose the path with lowest distance.

This systematic set of steps is called an algorithm. The problem was to find the shortest path from your home to the school. The algorithm is one of many possible solution. A problem can have many solutions. Note that based on a particular data structure that you are using to store the data, the solution can have slightly or completely different algorithm to solve the same problem. In addition, the efficiency of an algorithm depends on the way of data organization in data structure. Consider the previous example, there are two sorts of data structures as shown in Fig 2. If you tested the algorithm- Shortest-Path for data structure 1 and data structure 2, you may have different number of data comparisons/computations. The data structure for which the number of computations/comparisons for an algorithm is lowest is the most efficient data structure for the algorithm. Therefore, to find an efficient solution, data structure plays a very important and vital role for algorithm.

4. TYPES OF DATA STRUCTURES

Based on the way how data structures organize and store the data, they are classified into two main categories: - Primitive Data Structures and Non-primitive Data structures. The detail classification of Data Structures. Fig. 3 shows the detail classification of data structures. Primitive data structures are the basic data types. For example, the basic data types such as integer, real, character and Boolean are called primitive data structures. In C programming language, integer, float, double and character are primitive data structures. These primitive data structures contains atomic data. For example, a single integer variable can contain only one integer value.

Non-primitive data structures are those which can contain collection of data of fix size as in array or variable size as in linked list. Arrays, linked lists, stacks, queues, trees and graphs are non-primitive data structures. Non-primitive data structures can be further divided into linear and non-linear data structures. Linear data structures are those in which data are organized in a sequence or a linear list fashion. Arrays, linked lists, stacks, and queues are linear data structures since the data in these data structure are stored in linear fashion. In contrast, non-linear data structures organizes the data in some hierarchical fashion but not in a linear sequence. Trees and graphs are examples of non-linear data structure. -

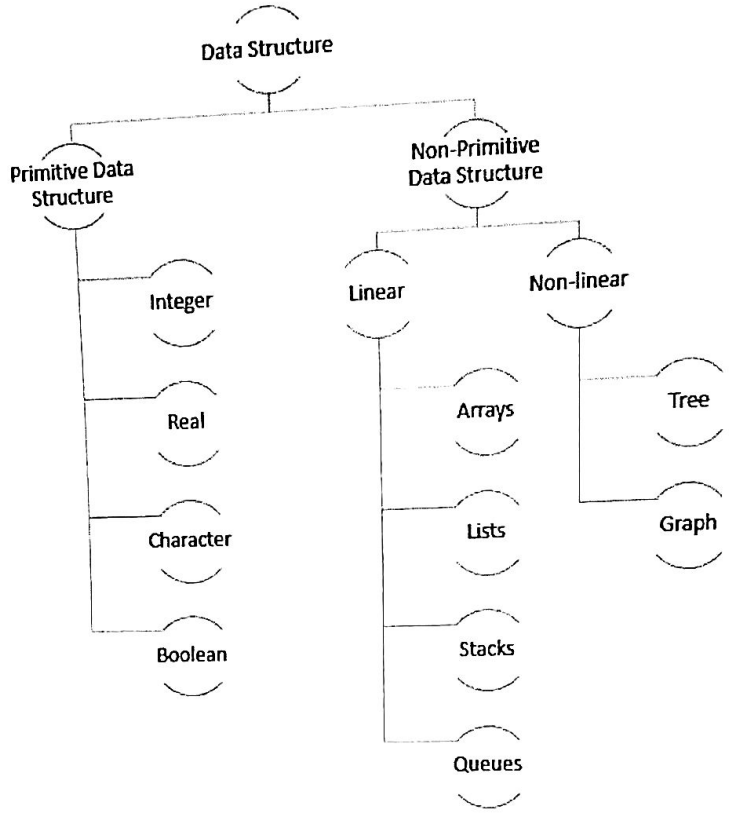
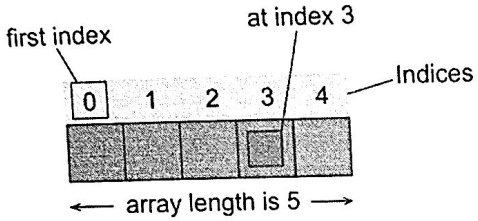


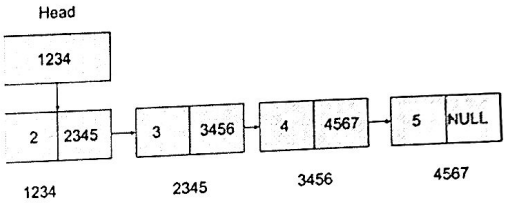
Fig. 3: Types of Data Structures

Based on whether the size of data structure is fixed or variable, data structures can be static or dynamic. A data structure is said to be static if the size of the data structure is declared at creation time and remains fix all time. For example, you create an array with size 100, then the array can contain only 100 elements. If your requirement is changed and want to store 5 more elements in the array, you cannot store the last 5 elements since its size is fixed for 100 elements. Static data structures can be efficient to store such type of data which are fixed in number. For example, if you want to store the name of month or name of days in array, then array will be best choice since they are fixed in number. In many cases, we cannot be sure about the size of an array to store the elements. In such cases, we assume a maximum size of the array it can have. Suppose you created an array of size 1000, and you have only 400 elements to store in some situation, then memory space of 600 elements will be empty and wasted. Similarly, if you want to store 1200 elements, you cannot store last 200 elements due to its size limitation to 1000. In such cases, the static data structures are not suitable and you need dynamic data structure.

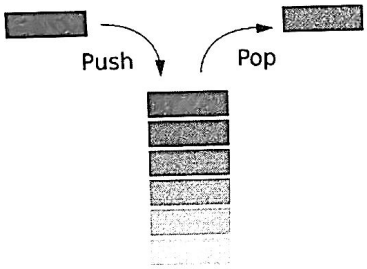
A data structure is said to be dynamic if the size of the data structures can be increased or decreased on demand. Therefore, you no need to worry about the size of the data structure at the time of creation. You can expand the size when you need more memory space and you can shrink the size when you don't need memory space anymore. Therefore, there will not be case of lack or wastage of memory space as in static data structures.



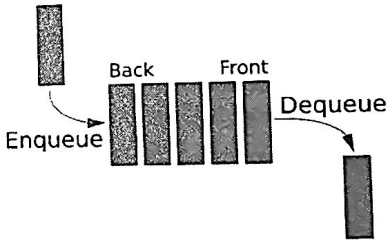
(a) Array



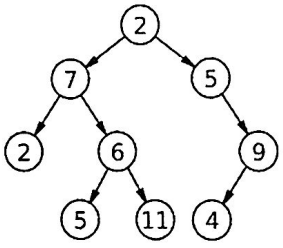
(b) Linked List



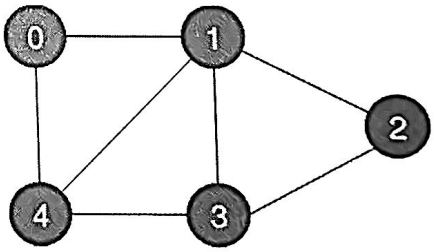
(c) Stack



(d) Queue



(e) Tree



(f) Graph

Fig. 4: Examples of different data structures

5. OPERATIONS IN DATA STRUCTURES

Data structures are used to organize and store data in some logical ways so that algorithms can efficiently use those data. The algorithms access data stored on data structure for different purposes using some operations. For example, sometimes algorithms need to traverse the data in data structure to search some particular data. The basic operations that can be performed in data structures are:

a) **Traversing:** - This operation is also called visiting. It is the operation of access each data/item stored in data structure exactly once so that certain data can be processed. For example, if you want to find an item in a tree data structure, then you will traverse each element of the tree using some traversing method to search the item.

b) **Searching:** - It is the operation that is commonly performed in data structure to find some data with a given key value. To search an item in data structure you have to traverse the data structure.

c) **Inserting:** - It is the operation which adds data/item into the data structure. Before you insert an item into the data structure, you have to find an appropriate location where you want to insert the item. You cannot insert an item into a data structure which is already full.

d) **Deleting:** - It is the operation of removing an item from data structure. You cannot delete an item from a data structure which is already empty.

e) **Sorting:** - It is the process of arranging the data in some logical order such as alphabetical or numeric order. For example, you should need to sort the elements in data structure before you perform binary search.

f) **Merging:** - It is the process of combining the elements of two different sorted data structures into a single data structure. This operation is mainly performed during the sorting operation to sort two different sorted data structures.

These are common operations that can be done with data stored in data structure. However, the way of doing these operations vary from one sort of data structure to another. For example, the insertion of an item into a stack is completely different when you insert the item into a queue.

6. ABSTRACT DATA TYPES

An abstract data type (ADT) is a set of data values and associated operations. However, it does not provide the implementation detail. For example, when you declare a variable in any programming language, the type of the variable specifies a set of values and a set of operations. Suppose, if you declare a variable say X of type int, using this declaration you are specifying a set of integer values and a set of operations (such as

addition, subtraction, multiplication and division) on X as well. But you don't need to know how ints are represented nor how the operations- addition, subtraction, multiplication and division are implemented. Therefore, the type int is an abstract data type. The abstract data type encapsulates the data and the operations and hide it from users. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

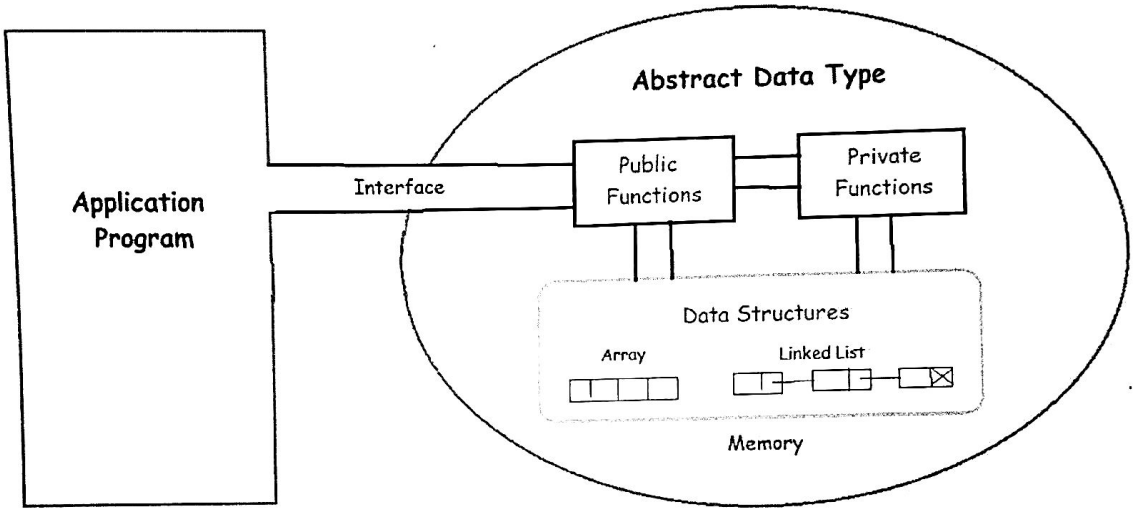


Fig. 5: ADT Model

Abstract data types provide an interface hiding the implementation detail for users. Users don't need to know about the implementation detail. This abstraction mechanism also provides the flexibility to change in the implementation without affecting the user's interface. It also provide a layer of security in the sense that user does not know the underlying representation and implementation and therefore they cannot change the implementation by mistakenly or intentionally. Fig. 5 shows abstract data type model. In the figure, you can see that the application programs access the data in the abstract data type through the use of interface using only the public functions. The interface only provides the list of supported operations, type of parameters they can accept and return type of these operations to the application programs. Implementation provides the internal representation of a data structure and the definition of the algorithms used in the operations of the data structure. Any ADT can be implemented using an array or a linked-list. The array implementation result in the static ADT and linked-list implementation results in dynamic ADT.

There are several programming languages such as C++, C#, Java, Python etc. support abstract data type through class mechanism. Classes are defined in terms of data and operations. The implementation of the operations of class is done separately.

A data structure is the implementation for an ADT. In an object-oriented language, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program. The term data structure often refers to data stored in a computer's main memory. The related term file structure often refers to the organization of data on peripheral storage, such as a disk drive or CD.

The concept of an ADT can help us to focus on key issues even in non-computing applications. For example, when operating a car, the primary activities are steering, accelerating, and braking. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer", "accelerate", and "brake". Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden.

These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation. For example, two popular implementations for large disk-based database applications are hashing and the B-tree. Both support efficient insertion and deletion of records, and both support exact-match queries. However, hashing is more efficient than the B-tree for exact-match queries. On the other hand, the B-tree can perform range queries efficiently, while hashing is hopelessly inefficient for range queries. Thus, if the database application limits searches to exact-match queries, hashing is preferred. On the other hand, if the application requires support for range queries, the B-tree is preferred. Despite these performance issues, both implementations solve versions of the same problem: updating and searching a large collection of records.

Data types have both a logical form and a physical form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form. The figure below illustrates this relationship between logical and physical forms for data types. When you implement an ADT, you are dealing with the

physical form of the associated data type. When you use an ADT elsewhere in your program, you are concerned with the associated data type's logical form.

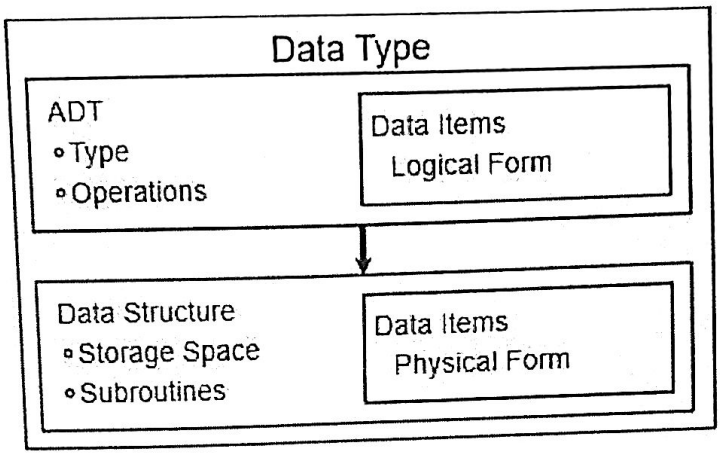


Figure 1.2.1: The relationship between data items, abstract data types, and data structures.

The ADT defines the logical form of the data type. The data structure implements the physical form of the data type. Users of an ADT are typically programmers working in the same language as the implementer of the ADT. Typically, these programmers want to use the ADT as a component in another application. The interface to an ADT is also commonly referred to as the Application Programmer Interface, or API, for the ADT. The interface becomes a form of communication between the two programmers. For Example, a particular programming environment might provide a library that includes a list class. The logical form of the list is defined by the public functions, their inputs, and their outputs that define the class. This might be all that you know about the list class implementation, and this should be all you need to know. Within the class, a variety of physical implementations for lists is possible.

7. Significance of Data Structures

Why we do have different data structures. Why we don't use the same data structure for all our needs. Each data structure has its own strengths and weaknesses. Some data structures are fast at sorting and others are not. Some data structures are fast at searching and others are not. Therefore, different data structures are used for different reasons and purposes. No single data structure works well for all purposes and therefore it is important to know the strengths and limitations of data structures. Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Data are growing massively. This creates big problems to store, manage and retrieve such huge amount of data. For example, consider the database of Facebook, YouTube and Gmail. You can imagine how big amount of data they are storing and managing. To search a data in such huge amount of data is another big problem. Processing speed is another issue when it is related to the huge amount of data. In internet billions of users can access data simultaneously.

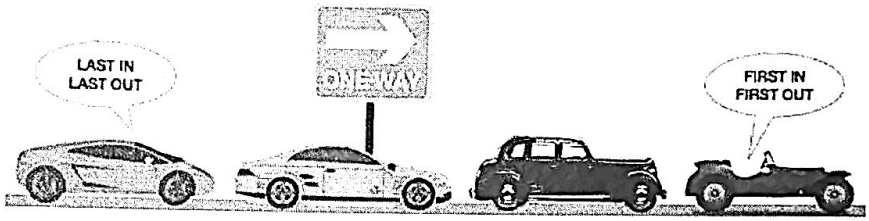
Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

8. Applications of Data Structures

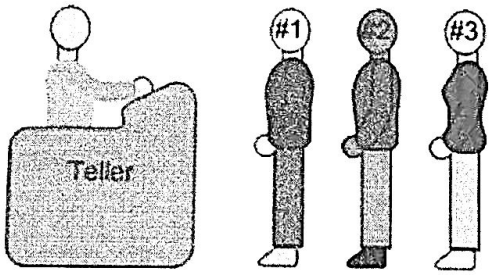
No single data structure works well for all purposes. Each data structure has its strengths and weaknesses. Therefore, based on a particular task, you need to choose a data structure which best fulfils your purpose. These are several data structures such as array, linked list stacks, queues, trees and graphs. Different kinds of data structures are used by algorithms to achieve various tasks or to solve various real world problems. The applications of data structures are listed by particular data structure-wise:

- **Linked List**
  - Linked list are usually a basic dynamic data structure which implements queues and stacks
  - In web-browsers, where it creates a linked list of web-pages visited, so that when you check history (traversal of a list) or press back button, the previous node's data is fetched.
- **Stack**
  - To reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
  - An "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
    - Undo/Redo stacks in Excel or Word.
  - Language processing

        - compiler's syntax check for matching braces is implemented by using stack.
    - A stack of plates/books in a cupboard.
    - A garage that is only one car wide. To remove the first car in we have to take out all the other cars in after it.
    - Wearing/Removing Bangles.
    - Back/Forward stacks on browsers.
    - Support for recursion
        - Activation records of method calls.
- **Queue**
    - Real World Example – line of a vehicle in one way road

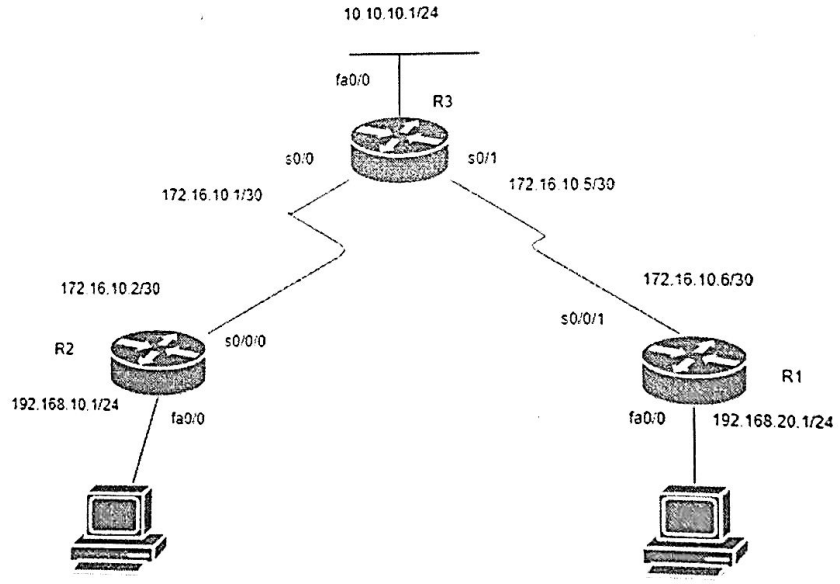


    - Real World Example – a line in a bank

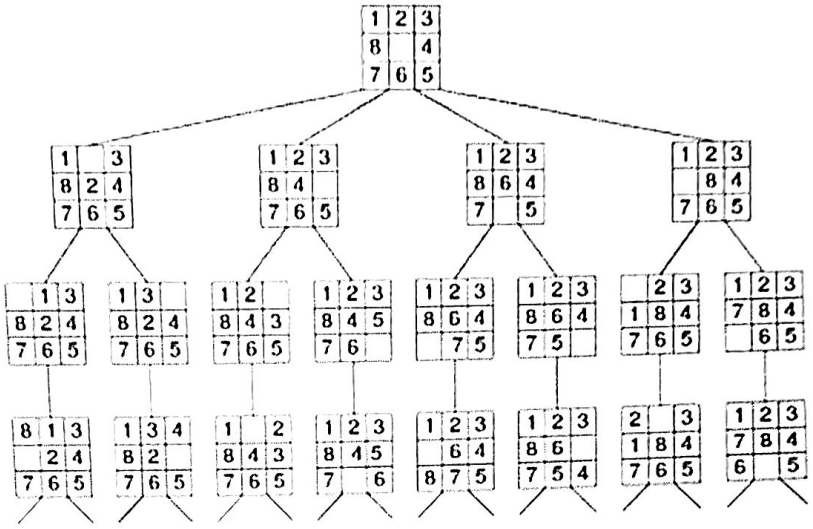


    - Programming Applications
        - When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
        - When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- **Dequeue**
    - Palindrome check
    - Steal Job Scheduling Algorithm
        - The steal algorithm implements task scheduling for several process(Multi-process scheduling)
        - Each Processor has a deque linked to it

    - If the process completes the execution of its own queue, it pulls the thread from the rear end of other processor and executes it.
  - Undo-Redo Operations in software applications
- **Trees**
  - Folders in Operating System
    - In windows go to command line and type tree. You can see the folder tree structure.
  - Linux File System is also a Tree
  - HTML Document
    - All html text, attributes are stored in tree called Document Object Model (DOM).
  - Network Routing also uses tree structure.



  - Syntax Tree in Compiler
    - In compiler, every expression is converted into syntax tree format.
  - Auto Corrector and Spell Checker also uses tree format
  - Next Move in Games
    - In Artificial intelligence game (opponent is CPU), next moves are stored using tree data structure.



9. REVIEW OF ARRAY, STRUCTURE, UNION, CLASS, POINTER

Arrays

Before understanding about an array, let us recall what a variable is. A variable is a name given to a memory location which can hold a single value. The value of the variable may vary during the life period of a program. That's why it is named as variable. In C/C++, an integer variable (say student_id) can be declared as

```
int student_id;
```

This variable declaration mainly indicates that variable name is student_id and the type of the data it can hold is integer type. Suppose, you want to store the ID of a single student, then you will declare the variable as above. However, suppose you want to store the ID of 5 students, then one way to store the ID of 5 students may be to declare five different variables as

```
int student_id1;
int student_id2;
int student_id3;
int student_id4;
int student_id5;
```

Now, suppose you want to store the ID of 1000 students, then what you will do. Do you declare 1000 different variables? If yes, what will happen if you want to store records of 1 million students? An ordinary variable can store a single value at a time. If you want to

store the records of large number of objects, it is almost impossible to store the records using ordinary variables to develop a program. The solution to this problem is an array.

An array is a collection of variables of same data types that are referred to by a common name. For example, an array can be a collection of integer variables only or float variables only or character variables only. An array of character variables in C is called string. Each element in the array are uniquely identified and accessed by an index. For example, an array to store the ID of 5 students can be declared as

int student_id[5];

Here, the identifier "student_id" is the name of the array. The type int indicates the type of the data the array holds. The value 5 enclosed within square bracket indicates the size of the array. The size of the array specifies the number of data the array can hold. In this example, an array named " student_id " is created. It can hold the student id of 5 employees. This array has 5 elements which are uniquely identified using index. Index in C starts from zero. Therefore, the index of last element will be size of the array minus one. The elements of the array are `student_id[0]`, `student_id[1]`, `student_id[2]`, `student_id[3]` and `student_id[4]`. The elements of an array are stored in continuous locations in memory. Using an array a large volume of data of same data type can be easily hold under a single name. An array named 'arr' that holds 1000 of integer data can be easily declared in a single line as

```
int arr[1000];
```

An array is a static data structure. Once the size of an array is defined during the declaration, then the size cannot be increased or decreased. In C/C++, arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Based on the dimensions, arrays are of the following types:

1. One-dimensional arrays
2. Multi-dimensional arrays

a) Some array operations

Insertion a new item into a specified location of array

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
```

```
{
    int arr[8] = {10, 20, 30, 40, 50, 60, NULL, NULL};
    int i;
    int newItem;
    printf("The array before insertion is:  ");
    for(i=0; i<8; i++)
    {
          printf("%d\t", arr[i]);
    }
    //insertion of newItem in index 2
    printf("Enter newItem to insert: ");
    scanf("%d", &newItem);
    for(i=7; i>=2; i--)
    {
          arr[i] = arr[i-1];
          if(i==2)
          {
                arr[i] = newItem;
          }
    }
    printf("The array After insertion is:  ");
    for(i=0; i<8; i++)
    {
          printf("%d\t", arr[i]);
    }
    getch();
    return 0;
}
```

Deletion a new item into a specified location of array

Lab work

Merging two arrays

Lab work

Traversing an array

Lab work

Pointer and Array

A pointer is a variable that points to another variable. A pointer variable contains the memory address of another variable to which it points to. Therefore a pointer references to a location in memory. The process of obtaining the value stored at the location is called dereferencing the pointer. A pointer can be declared in C as:

```
int *ptr;
```

This statement indicates that a pointer named ptr is declared and this pointer references to an int type object. The data type int is not the type of pointer but it is the type of the object to which it points.

```
int x = 5;
int *ptr;
ptr = &x;
int value;
value = *ptr;
int *p;
p = ptr;
```

The statement `int x = 5;` declares a variable `x` and a value `5` is assigned to `x`. The statement `int *ptr;` declares a pointer variable `ptr` which reference to an int type variable. The statement `ptr = &x;` assigns the memory address of variable x using the address operator `&`. The statement `value = *ptr;` shows how the value is obtained using a pointer variable. The statement `p = ptr;` shows how a pointer variable can be assigned to another pointer variable. Now the both pointers `p and ptr` point to the variable `value`.

Array and pointer are so related that the name of the array is actually the pointer to the first element of the array. The following program illustrate how a pointer can be used to access the elements of an array.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i;
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr;
    ptr = arr;
    printf("The Elements of Arr with Address and Value:\n\n");
    for(i = 0; i<5; i++)
    {
        printf("Address = %u\t", ptr);
        printf("Value = %d\n", *ptr);
        ptr++;
    }
    getch();
    return 0;
}
```

Output:

```
The Elements of Arr with Address and Value:

Address = 6487536       Value = 10
Address = 6487540       Value = 20
Address = 6487544       Value = 30
Address = 6487548       Value = 40
Address = 6487552       Value = 50
```

Passing an entire array to a function

A pointer can be used to pass the entire array to a function and can be used to access the elements of the array at that calling function. This is illustrated in the following program. In the program, the pointer to the first element of an array `arr` is passed to another pointer `ptr`. Then the pointer `ptr` is incremented in each iteration to point to the next element of the array `arr`.

```
#include<stdio.h>
#include<conio.h>
void display(int *, int);
int main()
{
    int arr[5] = {10, 20, 30, 40, 50};
    display(arr, 5);
    getch();
    return 0;
}
void display(int *ptr, int size)
{
    int i;
    printf("The Elements of Arr:\n");
    for(i = 0; i<5; i++)
    {
        printf("%d\t", *ptr);
        ptr++;
    }
}
```

```
#include<stdio.h>
#include<conio.h>
void display(int *, int);
int main()
{
    int arr[5] = {10, 20, 30, 40, 50};
    display(arr, 5);
    getch();
    return 0;
}
void display(int *ptr, int size)
{
    int i;
    printf("The Elements of Arr:\n");
    for(i = 0; i<5; i++)
    {
        printf("%d\t", *ptr+ i);
    }
}
```

Output:

```
The Elements of Arr:
10      20      30      40      50
```

Pointer to an array

Let us consider a statement

```
int (*ptr)[2];
```

This statement `int (*ptr)[2];` creates a pointer to an array of 2 integers. This is completely different to the statement

```
int *ptr[2];
```

This statement `int *ptr[2];` creates an array of 2 pointers ptr[0] and ptr[1]. Therefore, `int (*ptr)[2];` is a pointer to an array of 2 integers and `int *ptr[2];` is an array of 2 pointers.

The following program illustrate how a pointer to a two dimensional array is declared and used to access the elements of the two dimensional array.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i, j, *p;
    int arr[3][2] =     {
                        {10, 20},
                        {30, 40},
                        {50, 60}
                        };
    int (*ptr)[2];
    printf("The Elements of Arr:\n");
    for(i = 0; i<3; i++)
    {
        printf("\n");
        ptr = &arr[i];
        p = ptr;
        for(j = 0; j<2; j++)
        {
            printf("%d\t", *(p+j));
        }
    }
    getch();
    return 0;
}
```

Output:

```
The Elements of Arr:

10      20
30      40
50      60
```

Passing 2D array to a function

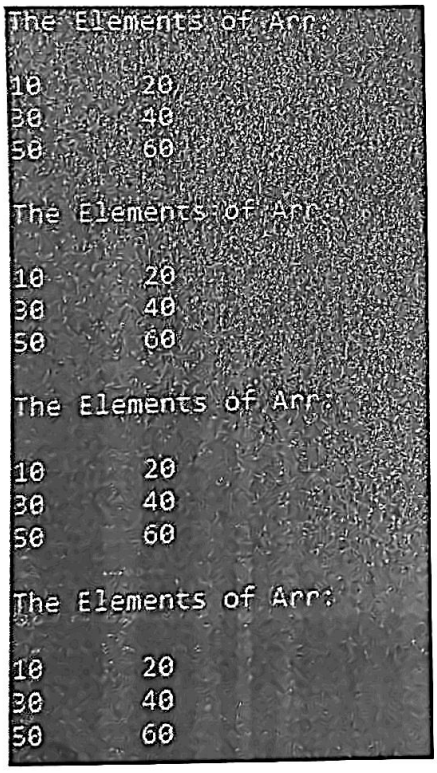
The following program illustrate how an array can be passed to a function and how the elements of the array can be accessed by using pointer to array or normal index of an array.

```
#include<stdio.h>
#include<conio.h>
void display1(int *ptr, int, int);
void display2(int *ptr, int, int);
int main()
{
   int arr[3][2] =     {
                       {10, 20},
                       {30, 40},
                       {50, 60}
                       };
```

```
    display1(arr, 3, 2);
    printf("\n\n");
    display2(arr, 3, 2);
    printf("\n\n");
    display3(arr, 3, 2);
    printf("\n\n");
    display4(arr, 3, 2);
    getch();
    return 0;
}
void display1(int *ptr, int row, int col)
{
    int i, j;
    printf("The Elements of Arr:\n");
    for(i = 0; i<row; i++)
    {
        printf("\n");
        for(j = 0; j<col; j++)
        {
            printf("%d\t", *ptr);
            ptr++;
        }
    }
}
void display2(int *ptr, int row, int col)
{
    int i, j;
    printf("The Elements of Arr:\n");
    for(i = 0; i<row; i++)
    {
        printf("\n");
        for(j = 0; j<col; j++)
        {
            printf("%d\t", *(ptr + i*col + j));
        }
    }
}
void display3(int (*p)[2], int row, int col)
{
    int i, j;
    int *ptr;
    printf("The Elements of Arr:\n");
    for(i = 0; i<row; i++)
    {
        printf("\n");
        ptr = p + i;
        for(j = 0; j<col; j++)
        {
```

```
            printf("%d\t", *(ptr + j));
        }
    }
}
void display4(int a[][2], int row, int col)
{
    int i, j;
    printf("The Elements of Arr:\n");
    for(i = 0; i<row; i++)
    {
        printf("\n");
        for(j = 0; j<col; j++)
        {
            printf("%d\t", a[i][j]);
        }
    }
}
```

Output:



Array of Pointers

The following program illustrate the use of array of pointers.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i, j, *p;
    int a = 10, b = 20, c = 30, d = 40, e = 50;
    int *ptr[5];
    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;
    ptr[3] = &d;
    ptr[4] = &e;
    for(i = 0; i<5; i++)
    {
        printf("%d\t", *ptr[i]);
    }
    getch();
    return 0;
}
```

Output:

```
10      20      30      40      50
```

Structure and Class

Structure and Class are very closely related. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent. The follow

```
#include <iostream>
#include <cstring>
#include<conio.h>
using namespace std;
struct  student
{
   void get_data(char n, int r);
   void display();
   private:
   char name;
   int roll;
};
void student::get_data(char n, int r)
{
   name = n;
   roll = r;
}
void student::display()
```

```
{
   cout<<"Name = "<<name<<endl;
   cout<<"Roll = "<<roll<<endl;
}
struct grade : public student
{
    int grade;
    void get_grade(int g)
    {
         grade = g;
    }
    void display2()
    {
          cout<<"Grade = "<<grade;
    }
};

int main()
{
   grade g;
   g.get_data('R', 101);
   g.get_grade(9);
   g.display();
   g.display2();
   getch();
   return 0;
}
```

Output:

```
Name = R
Roll = 101
Grade = 9
```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Although you can use a **struct** where you use a **class**, most programmers don't. Usually it is best to use a **class** when you want a class, and a **struct** when you want a C-like structure.

Pointer and structure/class

The following program illustrates how pointer and structure or class can be used to define a simple list data structure.

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

class node          //struct node
{
    int info;
    node *link;
};

int main()
{
    system("cls");

    node *first;

    first = new node;
    first->info=10;

    node *second;
    second = new node;
    second->info = 20;
    first->link=second;

    node *third;
    third = new node;
    third->info=30;
    second->link=third;
    third->link=NULL;

    node *temp;
    temp=first;
    while(temp!=NULL)
    {
        printf("%d", temp->info);
        temp=temp->link;
    }
```

```
    delete temp, first, second, third;
    getch();
    return 0;
}
```

REFERENCES

[1] https://www.geeksforgeeks.org/abstract-data-types/

[2] https://www.javatpoint.com/data-structure-algorithm

[3] https://www.tutorialspoint.com/data_structures_algorithms/data_structure_environment.htm

[4] https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/index.html

[5] https://livebook.manning.com/book/grokking-algorithms/chapter-1/9

[6] https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/