MODULE - I

# Introduction to Data Structures

MODULE 1

# Introduction to Data Structures

## Module Description

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. We have data player's name "Shane" and age 26. Here "Shane" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store players' records in a file or database as a data structure. ***For example,*** "Mitchelle" 30, "Steve" 31, "David" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

# Chapter Table of Contents

## Chapter 1.1

## Basics of Data Structure

## Aim

To equip the students with the basic skills of using Data Structures in programs

## Instructional Objectives

After completing this chapter, you should be able to:

- Describe Data Structures and its types
- Explain items included elementary data organisation
- Summarize the role of algorithms in programming
- Explain the procedure to calculate time and space complexities
- Explain the string processing with its functions
- Demonstrate memory allocation and address variable

## Learning Outcomes

At the end of this chapter, you are expected to:

- Outline the different types of data structures
- Elaborate asymptotic notations with example
- Calculate the time and space complexities of any sorting algorithm
- List down the string processing functions
- Summarize the contents in elementary data organisation
- Differentiate static and dynamic memory allocation

### 1.1.1 Introduction to Data Structures

Computers can store, retrieve and process vast amounts of data within its storage media. In order to work with a large amount of data, it is very important to organize the data properly. If the data is not organized properly, it becomes difficult to access the data. Thus, if the data is organized efficiently, any operation can be performed on data very easily and quickly. Hence it provides a faster response to the user.

This organization of data can be done with the help of data structures. Data structures enable a programmer to properly structure large amounts of data into conceptually manageable relationships. If we use a data structure to store the data, it becomes very easy to retrieve and process them.

A **data structure** can be defined as a particular method of organizing a large amount of data so that it can be used efficiently. A data structure can be used to store data in the form of a stack, queue etc.

**Any Data structure will follow these 4 rules:**

1. It should be an agreement on how to store data in memory,

***For example,*** data can be stored in an array, queue, linked list etc.

2. It should specify the operations we can perform on that data,

***For example,*** we can specify add, delete, search operations on any data structure

3. It should specify the algorithms for those operations,

***For example,*** efficient algorithms for searching element in array.

4. The algorithms used must be time and space efficient.

We have many primitive data types like integer, character, string etc. which stores specific kind of data. Data structures allow us to perform operations on groups of data, such as adding an item to a list, searching a particular element from the list etc. When a data structure provides operations, we can call the data structure an abstract data type (abbreviated as ADT).

A data structure is a form of abstract data type having its own set of data elements along with functions to perform operations on that data. Data structures allow us to manage large amounts of data efficiently so that it can be stored in large databases. These data structures can be used to design efficient sorting algorithms.

Every data structure has advantages and disadvantages. Every data structure suits to specific problem domain depending upon the type of operations and the data arrangement.

***For example,*** an array data structure is suitable for read operations. We can use an array as a data structure to store n number of elements in contiguous memory locations and read/ add elements as and when required. Other data structures include liked lists, queue, stack etc.

## Self-assessment Questions

1) ____________________is NOT the component of data structure.

   a) Operations    b) Storage Structures

   c) Algorithms    d)None of above

2) Which of the following are true about the characteristics of abstract data types?

   a) It exports a type    b) It exports a set of operations

   c) It exports a set of elements    d) It exports a set of arrays

3) Each array declaration need not give, implicitly or explicitly, the information about,

   a) The name of array    b) The data type of array

   c) The first data from the set to be stored    d) the index set of the array

## 1.1.2 Classification of data structures

A data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion.

There are certain linear data structures (e.g., stacks and queues) that permit the insertion and deletion operations only at the beginning or at end of the list, not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

### Types of data structures:

**A data structure can be broadly classified into:**

1. Primitive data structure
2. Non-primitive data structure

## (i) Primitive data structure

The data structures, typically those data structure that are directly operated upon by machine level instructions, *i.e.,* the fundamental data types such as int, float, double in case of ‘c’ are known as primitive data structures.

**Primitive data types are used to represent single values:**

- **Integer:** This is used to represent a number without decimal point.
  ***For example,*** 22, 80
- **Float and Double:** This is used to represent a number with decimal point.
  ***For example,*** 54.1, 57.8
- **Character:** This is used to represent single character.
  ***For example,*** ‘L’, ‘g’
- **String:** This is used to represent group of characters.
  ***For example,*** "Hospital Management”
- **Boolean:** This is used represent logical values either true or false.

## (ii) Non-primitive data structure

A non-primitive data type is something else such as an array structure or class is known as the non-primitive data type.

The data types that are derived from primary data types are known as non-Primitive data types. These data types are used to store group of values.

**The non-primitive data types are:**

- Arrays
- Structure
- Union
- linked list
- Stacks
- Queue etc.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data.

The non-primitive data types are used to store the group of values.

**There are two types of non-primitive data structures.**

1. Linear Data Structures
2. Non-linear data structures

### 1. Linear Data Structure:

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

A list is an ordered list, which consists of different data items connected by means of a link or pointer. This type of list is also called a linked list. A linked list may be a single list or double linked list.

- **Single linked list:** A single linked list is used to traverse among the nodes in one direction.

- **Double linked list:** A double linked list is used to traverse among the nodes in both the directions.

A linked list is normally used to represent any data used in word-processing applications, also applied in different DBMS packages.

**A list has two subsets. They are:**

- **Stack:** It is also called as last-in-first-out (LIFO) system. It is a linear list in which insertion and deletion take place only at one end. It is used to evaluate different expressions.

- **Queue:** It is also called as first-in-first-out (FIFO) system. It is a linear list in which insertion takes place at once end and deletion takes place at other end. It is generally used to schedule a job in operating systems and networks.

## 2. Non-Linear data structure:

A list, which does not show the relationship of adjacency between elements, is said to be non-linear data structure.

**The frequently used non-linear data structures are,**

- **Trees:** It maintains hierarchical relationship between various elements.

- **Graphs:** It maintains random relationship or point-to-point relationship between various elements.

The Figure 1.1.1 shows the classification of data structures.

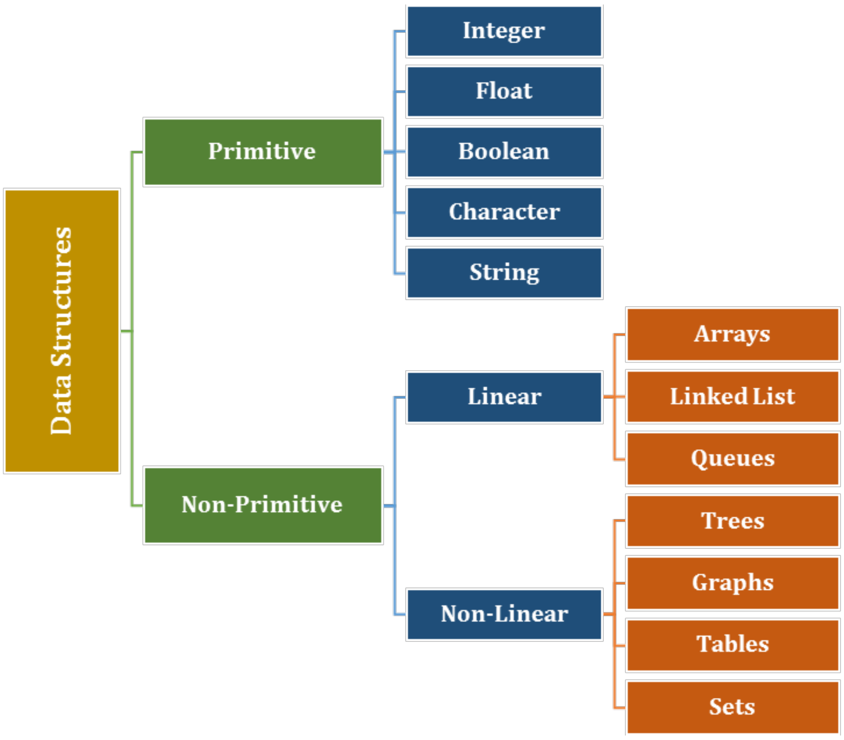


*Figure 1.1.1: Types of data structures*

## Self-assessment Questions

4) Which of the following data structure is linear type?

| | |
|---|---|
| a) Graph | b) Trees |
| c) Binary tree | d) Stack |

5) Which of the following data structure is non-linear type?

| | |
|---|---|
| a) Strings | b) Lists |
| c) Stacks | d) Graph |

6) Which of the following data structure can't store the non-homogeneous data elements?

| | |
|---|---|
| a) Arrays | b) Records |
| c) Pointers | d) Stacks |

### 1.1.3 Elementary Data Organization

**There are some basic terminologies related to Data Structures. They are detailed below:**

- **Data**

The term data means a value or set of values. These values may represent an observation, like roll number of student, marks of student, name of the employee, address of person, phone number, etc. In programming languages we generally express data in the form of variables with variable name as per the type of data like integer, floating point, character, etc.

***For example,*** figures obtained during exit polls, roll number of student, marks of student, name of the employee, address of person, phone number etc.

- **Data item:**

A data item is a set of characters which are used to represent a specific data element. It refers to a single unit of values. It is also called as Field.

***For example,*** name of student in class is represented by data item say std_name.

**The data item can be classified into two data types depending on usage:**

1. **Elementary data type:** These data items can't be further subdivided.
   ***For example,*** roll number

2. **Group data type:** These data items can be further sub divided into elementary data items. ***For example,*** Date can be divided into days, months, years

- **Entity:**

An entity is something that has a distinct, separate existence, though it need not be a material existence. An entity has certain attributes, or properties, which may be assigned values. Values assigned may be either numeric or non-numeric.

***For example,*** Student is an entity. The possible attributes for student can be roll number, name and date of birth, gender, and class. Possible values for these attributes can be 32, Alex, 24/09/2000, M, 11. In C language we usually use structures to represent an entity.

- **Entity Set:**

An entity set is a group of or set of similar entities.

***For example,*** Consider a situation where we have multiple entities with same attributes, say, students in B.Tech second year.

Here each student will have their respective roll numbers, names, marks obtained etc. All these students can represent an entity set like an array of students.

- **Information:**

When the data is processed by applying certain rules, the newly processed data is called information. The data is not useful for decision marking whereas information is useful for decision making.

***For example,*** a student who has scored maximum marks in a subject becomes information as it is processed and conveys a meaning

- **Record:**

Record is a collection of field values of a given entity. Set of multiple records with same fields form a file. Now a row in this file can be termed as a record. Every record has one or more than one fields associated with it. Generally each record has at least one unique identifier field.

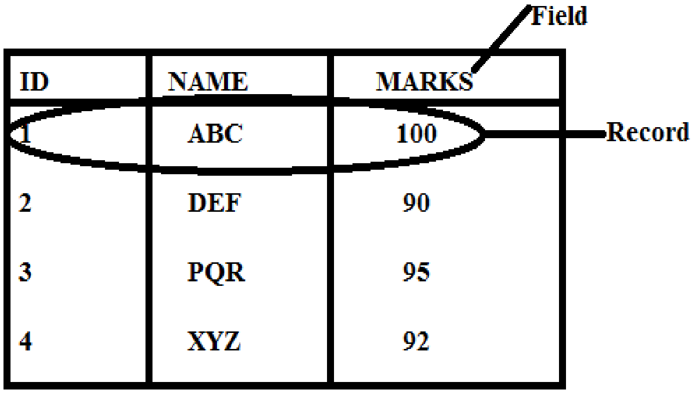
***For example,*** roll number, name, address etc. of a particular student.

- **File:**

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

***For example,*** file containing records of students of a particular class.

Figure 1.1.2 shows the student file structure.



| ID | NAME | MARKS |
|---|---|---|
| 1 | ABC | 100 |
| 2 | DEF | 90 |
| 3 | PQR | 95 |
| 4 | XYZ | 92 |

Student File

*Figure 1.1.2: Structure of a File*

- **Key:**

A key is one or more field(s) in a record that take(s) unique values and can be used to distinguish one record from the others.

***For example,*** in the above snapshot, ID is the key to identify particular record.

## 1.1.4 Time and Space Complexity

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

**It depends on:**

i. The speed of the computer

ii. The programming language

iii. The compiler that translates the programming language into code which runs directly on the computer

The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function.

**There are two main complexity measures of the efficiency of an algorithm:**

- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean:

    i. The number of memory accesses performed

    ii. The number of comparisons between integers

    iii. The number of times some inner loop is executed

    iv. Some other natural unit related to the amount of real time the algorithm will take

- This idea of time is always kept separate from "wall clock" time, since many factors unrelated to the algorithm itself can affect the real time like:

    i. The language used

    ii. Type of computing hardware

    iii. Proficiency of the programmer

    iv. Optimization in the compiler

It turns out that, if the units are chosen wisely, all the other things do not matter and thus an independent measure of the efficiency of the algorithm can be done.

- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. The requirement of the "extra "memory is often determined by not counting the actual memory needed to store the input itself.

    We use natural, but fixed-length units, to measure space complexity. We can use bytes, but it's easier to use units like: number of integers, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

    ***For example,*** "This algorithm takes n2 time". Here, n is the number of items in the input or "This algorithm takes constant extra space" because the amount of extra memory needed does not vary with the number of items processed.

An array of n floating point numbers is to be put into ascending numerical order. This task is called sorting. One simple algorithm for sorting is selection sort. Let an index i go from 0 to n-1, exchanging the ith element of the array with the minimum element from i up to n. Here are the iterations of selection sort carried out on the sequence {4 3 9 6 1 7 0}:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | comments |
|---|---|---|---|---|---|---|---|---|
| \| | 4 | 3 | 9 | 6 | 1 | 7 | 0 | initial |
| i=0 \| | 0 | 3 | 9 | 6 | 1 | 7 | 4 | swap 0, 4 |
| i=1 \| | 0 | 1 | 9 | 6 | 3 | 7 | 4 | swap 1, 3 |
| i=2 \| | 0 | 1 | 3 | 6 | 9 | 7 | 4 | swap 3, 9 |
| i=3 \| | 0 | 1 | 3 | 4 | 9 | 7 | 6 | swap 6, 4 |
| i=4 \| | 0 | 1 | 3 | 4 | 6 | 7 | 9 | swap 9, 6 |
| i=5 \| | 0 | 1 | 3 | 4 | 6 | 7 | 9 | (done) |

Here is a simple implementation in C:

```
int find_min_index (float [], int, int);
void swap (float [], int, int);
/* selection sort on array v of n floats */

Void selection_sort (float v[], int n) {
      int   i;

      /* for i from 0 to n-1, swap v[i] with the minimum
       * of the i'th to the n'th array elements
       */
      for (i=0; i<n-1; i++)
            swap (v, i, find_min_index (v, i, n));
}
/* find the index of the minimum element of float array v from
 * indices start to end
 */
int find_min_index (float v[], int start, int end) {
      int   i, mini;

      mini = start;
      for (i=start+1; i<end; i++)
            if (v[i] < v[mini]) mini = i;
      return mini;
}
/* swap i'th with j'th elements of float array v */
void swap (float v[], int i, int j) {
      float t;
      t = v[i];
      v[i] = v[j];
      v[j] = t;
}
```

The performance of the algorithm is quantified, the amount of time and space taken in terms of n. It is interesting to note how the time and space requirements change as n grows large; sorting 10 items is trivial for almost any reasonable algorithm one can think of, but what about 1,000, 10,000, 1,000,000 or more items?

It is clear from this *example*, the amount of space needed is clearly dominated by the memory consumed by the array. If the array can be stored, it can sort it. That is, it takes constant extra space.

The main interesting point is in the amount of time the algorithm takes. One approach is to count the number of array accesses made during the execution of the algorithm; since each array access takes a certain (small) amount of time related to the hardware, this count is proportional to the time the algorithm takes.

Thus end up with a function in terms of n that gives us the number of array accesses for the algorithm. This function is called *T* (n), for Time.

*T* (n) is the total number of accesses made from the beginning of selection sort until the end. Selection_sort itself simply calls swap and find_min_index as i go from 0 to n-1, so

$$\sum_{i=0}^{n-2}$$

*T* (n) = [time for swap + time for find_min_index (v, i, n)].

(n-2) because for loop goes from 0 up to **but not including** n-1). (**Note:** for those not familiar with Sigma notation, the looking formula above just means "the sum, as we let i go from 0 to n-2, of the time for swap plus the time for find_min_index (v, i, n).) The swap function makes four accesses to the array, so the function is now,

$$\sum_{i=0}^{n-2}$$

*T* (n) = [4 + time for find_min_index (v, i, n)].

With respect to find_min_index, it is seen that it does two array accesses for each iteration through the for loop, and it does the for loop n - i - 1 time:

$$\sum_{i=0}^{n-2}$$

*T* (n) = [4 + 2 (n - i - 1)].

With some mathematical manipulation, this can be broken up into:

$$\sum_{i=0}^{n-2}$$

*T* (n) = 4(n-1) + 2n (n-1) - 2(n-1) - 2.

(Everything times n-1 because it goes from 0 to n-2, *i.e.*, n-1 times). Remembering that the sum of i as i goes from 0 to n is (n (n+1))/2, then substituting in n-2 and cancelling out the 2's:

T (n) = 4(n-1) + 2n (n-1) - 2(n-1) - ((n-2) (n-1)).

And to make a long story short,

*T* (n) = n2 + 3n - 4.

So this function gives us the number of array accesses selection sort makes for a given array size, and thus an idea of the amount of time it takes. There are other factors affecting the performance, For instance the loop overhead, other processes running on the system, and the fact that access time to memory is not really a constant. But this kind of analysis gives a good idea of the amount of time one will spend waiting, and allows comparing these algorithms to other algorithms that have been analysed in a similar way.

## (i) Asymptotic Notation

The function, *T* (n) = n2 + 3n – 4 (refer earlier section), describes precisely the number of array accesses made in the algorithm. In a sense, it is a little too precise; all we really need to say is n2; the lower order terms contribute almost nothing to the sum when n is large. One likes a way to justify ignoring those lower order terms and to make comparisons between algorithms easy. So the asymptotic notation is used.

The **worst-case** complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.

The **best-case** complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.

Finally, the **average-case** complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n.

- **Lower Bound:** A non-empty set A and its subset B is given with relation ≤. An element a is called lower bound of B if a ≤ x x B (read as if a is less than equal to x for all x belongs to set B). ***For example,*** a non-empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The lower bound of B= 1, 2 as 1, 2 in the set A is less than or equal to all element of B.

- **Upper Bound:** An element A is called upper bound of B if x ≤ a x B. ***For example,*** a non-empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The upper bound of B= 3,4,5,6 as 3,4,5,6 in the set A is greater than or equal to all element of B.

- **Tight Bound:** A bound (upper bound or lower bound) is said to be tight bound if the inequality is less than or equal to (≤).

### Theta (Θ) Notation

It provides both upper and lower bounds for a given function.

**Θ (Theta) Notation:** means 'order exactly'. Order exactly implies a function is bounded above and bounded below both. This notation provides both minimum and maximum value for a function. It further gives that an algorithm will take this much of minimum and maximum time that a function can attain for any input.

Let g(n) be given function. f(n) be the set of function defined as,

Θ (g(n)) = {f(n): if there exist positive constant c1,c2 and n0 such that 0≤c1g(n)≤f(n) ≤c2g(n) for all n, n n0}

It can be written as f(n)= Θ(g(n)) or f(n) Θ(g(n)), here f(n) is bounded both above and below by some positive constant multiples of g(n) for all large values of n. It is described in the following figure.

Figure 1.1.3 shows the Graphical representation of Theta (Θ) Notation.

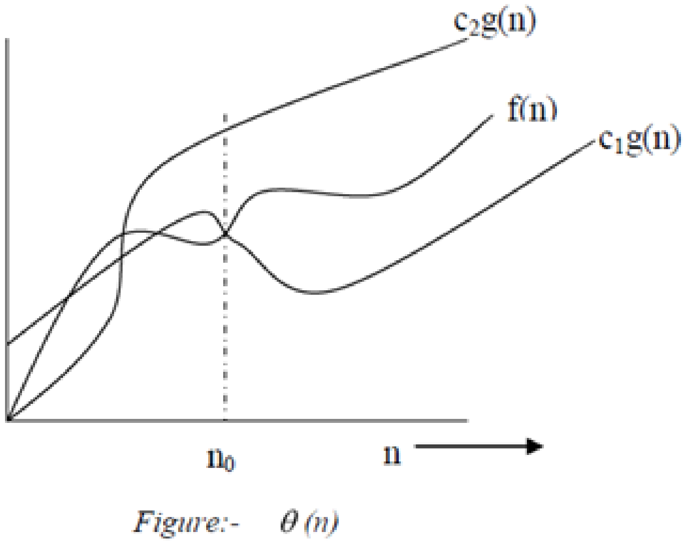


*Figure 1.1.3 Theta (Θ) Notation Graph*

In the above figure, function f(n) is bounded below by constant c1 times g(n) and above by constants c2 times g(n). We can explain this by following example:

***For example,***

To show that 3n+3 = Θ (n) or 3n+3 Θ (n) we will verify that f(n) g(n) or not with the help of the definition

*i.e.,* Θ (g(n)) = {f(n): if there exist positive constant $c_1$,$c_2$ and n0 such that 0≤$c_1$g(n)≤f(n) ≤$c_2$g(n) for all n, n n0}

In the given problem f(n)= 3n+3 and g(n)=n to prove f(n) g(n) we have to find $c_1$,$c_2$ and n0 such that 0≤$c_1$g(n)≤f(n) ≤$c_2$g(n) for all n, n n0

=> to verify f(n) ≤$c_2$g(n)

We can write f(n)=3n+3 as f(n)=3n+3 ≤ 3n+3n (write f(n) in terms of g(n) such that mathematically inequality should be true)

≤6n for all n > 0

$c_2$=6 for all n > 0 *i.e.,* n0=1

To verify 0≤$c_1$g(n)≤f(n)

We can write f(n)=3n+3 3n (again write f(n) in terms of g(n) such that mathematically inequality should be true)

$c_1$=3 for all n, n0=1

=> 3n≤3n+3≤6n for all n n0, n0=1

*i.e.,* we are able to find, $c_1$=3, $c_2$=6 n0=1 such that 0≤$c_1$g(n)≤f(n) ≤$c_2$g(n) for all n, n n0 So, f(n)= Θ (g(n)) for all n >=1

### Big O Notation

This notation provides upper bound for a given function. O(Big Oh) Notation: mean `order at most' *i.e.,* bounded above or it will give maximum time required to run the algorithm. For a function having only asymptotic upper bound, Big Oh „O‟ notation is used. Let a given function g(n), O(g(n))) is the set of functions f(n) defined as,

O(g(n)) = {f(n): if there exist positive constant c and n0 such that 0≤f(n) ≤cg(n) for all n, n n0}

f(n) = O(g(n)) or f(n) O(g(n)), f(n) is bounded above by some positive constant multiple of g(n) for all large values of n.

The definition is illustrated with the help of figure 1.1.4.

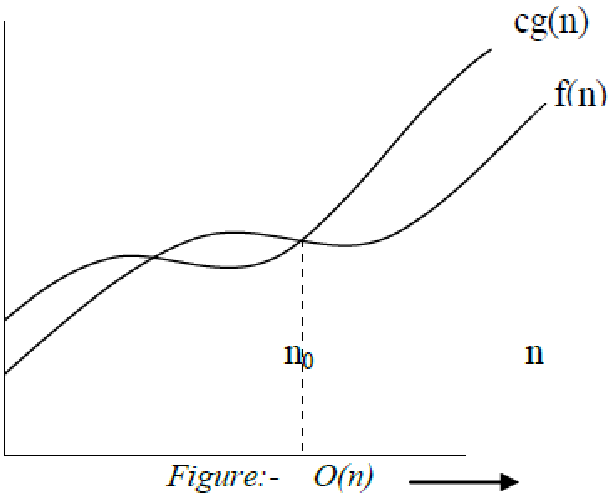


*Figure 1.1.4: Big O Notation Graph*

In this figure, function f(n) is bounded above by constant c times g(n). We can explain this by following examples:

***For example,***

To show $3n^2$+4n+6=O (n2) we will verify that f(n) g(n) or not with the help of the definition *i.e.,* O(g(n))={f(n): if there exist positive constant c and n0 such that 0≤f(n) ≤cg(n) for all n, n n0}

### In the given problem:

f(n)= $3n^2$+4n+6

g(n)= $n^2$

To show 0≤f(n) ≤cg(n) for all n, n n0

f(n)= $3n^2+4n+6 \leq 3n^2+n^2$ for n6

$\leq 4\ n^2$

c=4 for all nn0, n0=6

*i.e.,* we can identify , c=4, n0=6

So, f(n)=O(n2)

### Properties of Big O

The definition of big O is difficult to have to work with all the time, kind of like the "limit" definition of a derivative in Calculus.

**Here are some helpful theorems that can be used to simplify big O calculations:**

- Any kth degree polynomial is O (nk).
- a nk = O(nk) for any a > 0.
- Big O is transitive. That is, if f(n) = O(g(n)) and g(n) is O(h(n)), then f(n) = O(h(n)).

### Big-Ω (Big-Omega) notation

Sometimes, it is said that an algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation; that's the Greek letter "omega."

If a running time is Ω (f (n)), then for large enough n, the running time is at least k·f (n)  for some constant k. Here's how to think of a running time that is Ω(f (n)): Figure 1.1.5 shows the graph for Big-Ω (Big-Omega) notation

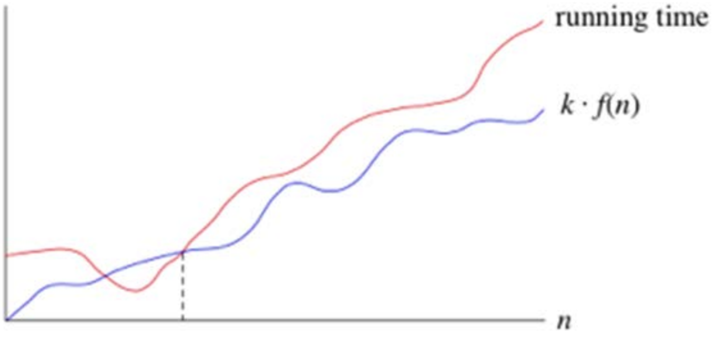


*Figure 1.1.5: Big-Ω (Big-Omega) notation*

### Graph

It is said that the running time is "big-Ω of f (n)." We use big-Ω notation for **asymptotic lower bounds**, since it binds the growth of the running time from below for large enough input sizes.

Just as Θ (f (n)) automatically implies O (f (n)), it also automatically implies Ω (f (n)). So it can be said that the worst-case running time of binary search is Ω (lg n). One can also make correct, but imprecise, statements using big-Ω notation. ***For example,*** just as if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's at least 10 dollars," you can also say that the worst-case running time of binary search is Ω(1), because it takes at least constant time.

## Self-assessment Questions

7) When determining the efficiency of algorithm, the space factor is measured by?

   a) Counting the maximum memory needed by the algorithm
   b) Counting the minimum memory needed by the algorithm
   c) Counting the average memory needed by the algorithm
   d) Counting the maximum disk space needed by the algorithm

8) The complexity of Bubble sort algorithm is,

   a) O (n)
   b) O (log n)
   c) O (n2)
   d) O (n log n)

9) The Average case occur in linear search algorithm:

   a) When Item is somewhere in the middle of the array
   b) When Item is not in the array at all
   c) When Item is the last element in the array
   d) When Item is the last element in the array or is not there at all

### 1.1.5 String Processing

In C, textual data is represented using arrays of characters called a *string*. The end of the string is marked with a special character, the null character, which is simply the character with the value 0. The null or string-terminating character is represented by another character escape sequence, \0.

In fact, C's only truly built-in string-handling is that it allows us to use string constants (also called string literals) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the \0 character. ***For example,*** we can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating \0). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that we decide how big the arrays and other data structures we use to hold strings are.

We must call functions to perform operations on strings like copying and comparing them, breaking strings up into parts, joining them etc. We will look at some of the basic string functions here.

1. **strlen:** Returns the length of the string; (*i.e.,* the number of characters in it), not including the \0 character

   ```
   char string7[] = "abc";
   int len = strlen(string7);
   printf("%d\n", len);
   ```

   Output is 3

2. **strcpy:** This function copies one string to another

   ```
   char string1[] = "Hello, world!";
   char string2[20];
   strcpy(string2, string1);
   ```

Here value of string1 *i.e.,* “Hello world!” will be copied into string2.

3. **strcat:** This function concatenates two strings. It appends one string onto the end of another.

```
char string5[20] = "Hello, ";
char string6[] = "world!";
printf("%s\n", string5);
          strcat(string5, string6);
          printf("%s\n", string5);
```

The first call to printf prints “Hello, ", and the second one prints “Hello, world!”, indicating that the contents of string6 have been tacked on to the end of string5.

4. **strcmp:** The standard library's strcmp function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically “less than” the second string, or a positive number if the first string is “greater.”

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
          printf("strings are equal\n");
else      printf("strings are different\n");
```

This code fragment will print “strings are different”. Notice that strcmp does not return a Boolean, true/false, zero/nonzero answer.

The table 1.1.1 below lists some more commonly used string functions.

*Table 1.1.1: Commonly used String Functions*

| Function | Description |
|---|---|
| Strcmpi() | Compares two strings with case insensitivity |
| Strrev() | Reverses a string |
| Strlwr () | Converts uppercase string letters to lowercase |

| | |
|---|---|
| Strupr() | Converts lowercase string letters to uppercase |
| Strchr() | Finds the first occurrence of a given character in a string |
| Strrchr() | Finds the last occurrence of a given character in a string |
| Strset() | Sets all characters in a string to a given character |
| Strnset() | Sets the specified number of characters ina  string to a given character |
| Strdup() | Used for duplicating a string |

## Self-assessment Questions

10) Which of the following function compares 2 strings with case-insensitively?

a) Strcmp(s, t) b) strcmpcase(s, t)

c) Strcasecmp(s, t) d) strchr(s, t)

11) How will you print \n on the screen?

a) printf("\n") b) printf('\n');";

c) echo "\\n d) printf("\\n");

12) Strcat function adds null character.

a) Only if there is space b) Always

c) Depends on the standard d) Depends on the compiler

### 1.1.6 Memory Allocation

Memory allocation is primarily a computer hardware operation but is managed through operating system and software applications. Memory allocation process is quite similar in physical and virtual memory management. Programs and services are assigned with a specific memory as per their requirements when they are executed. Once the program has finished its operation or is idle, the memory is released and allocated to another program or merged within the primary memory.

**Memory allocation has two core types;**

- **Static Memory Allocation:** The program is allocated memory at compile time.
- **Dynamic Memory Allocation:** Memory is allocated as required at run-time.

#### (i) Static memory allocation

The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variables have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation.

***For example,*** a variable in a function, is only there until the function finishes.

```
void func() {
int i; /* `i` only exists during `func` */
}
```

#### (ii) Dynamic memory allocation

Dynamic memory allocation is when an executing program requests that the operating system give it a block of main memory. Dynamic allocation is a unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

The program then uses this memory for some purpose. Usually the purpose is to add a node to a data structure. In object oriented languages, dynamic memory allocation is used to get the memory for a new object. The memory comes from above the static part of the data segment. Programs may request memory and may also return previously dynamically allocated memory. Memory may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated.

A new dynamic request for memory might return a range of addresses out of one of the holes. But it might not use up all the holes, so further dynamic requests might be satisfied out of the original hole. If too many small holes develop, memory is wasted because the total memory used by the holes may be large, but the holes cannot be used to satisfy dynamic requests. This situation is called memory fragmentation. Keeping track of allocated and deallocated memory is complicated. A modern operating system does all the tracking of memory.

```
int* func() {
int* mem = malloc(1024);
return mem;
}

int* mem = func(); /* still accessible */
```

In the above example, the allocated memory is still valid and accessible, even though the function terminated. When you are done with the memory, you have to free it:

```
free(mem);
```

## Self-assessment Questions

13) In static memory allocation, complier allocates required memory using dereference operator.

a) True b) False

14) Memory is dynamically allocated once the program is complied.

a) True b) False

15) Memory Fragmentation occurs when small holes of memory are formed which cannot be used to fulfil the dynamic requests.

a) True b) False

### 1.1.7 Accessing the address of a variable: Address of (&) operator

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator.

***For example,*** foo = &myvar;

This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the address-of operator (&), we are no longer assigning the content of the variable itself to foo, but its address. The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.

**In this case, consider the following code fragment:**

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:

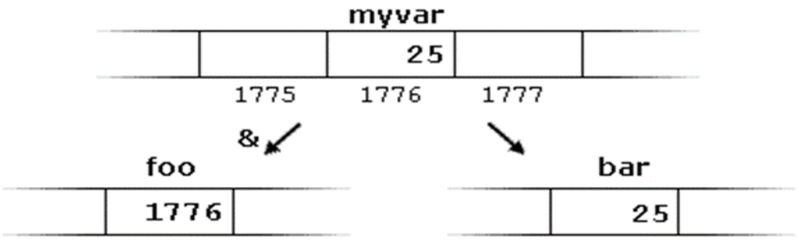


*Figure 1.1.6:*

First, we have assigned the value 25 to myvar (a variable whose address in memory we assumed to be 1776). The second statement assigns foo variable, the address of myvar, which we have assumed to be 1776. Finally, the third statement, assigns the value contained in myvar to bar. This is a standard assignment operation. The main difference between the second and third statements is the appearance of the address-of operator (&). The variable that stores the address of another variable (like foo in the earlier example) is what in C is called a pointer. Pointers are a very powerful feature of the language that has many uses in lower level programming.

## Did you Know?

In order to use these string functions you must include string.h file in your C program using #include <string.h>.

## Self-assessment Questions

16) Choose the right answer.

Prior to using a pointer variable.

a) It should be declared

b) It should be initialized

c) It should be declared and initialized

d) It should be neither declared nor initialized

17) The address operator &, cannot act on ________________ and ____________.

18) The operator > and < are meaningful when used with pointers, if,

a) The pointers point to data of similar type

b) The pointers point to structure of similar data type.

c) The pointers point to elements of the same array.

d) The pointers point to elements of the another array.

# Summary

- Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

- The address of a variable can be obtained by preceding the name of a variable with the address-of operator (&).

- Data structures are categorized into two types: linear and nonlinear. Linear data structures are the ones in which elements are arranged in a sequence, nonlinear data structures are the ones in which elements are not arranged sequentially.

- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

- The basic terminologies in the concept of data structures are Data, Data item, Entity, Entity set, Information, Record, file, key etc.

- String functions like strcmp, strcat, strlen are used for string processing in C.

- Static and Dynamic memory allocation are the core types of memory allocation.

## Terminal Questions

1. How is a primitive data structure different from that of a non-primitive data structure?
2. With an example explain upper bound, lower bound and tight bound.
3. Explain the concept of string processing in C with some basic string functions.
4. Draw a comparison between static memory allocation and dynamic memory allocation.

# Answer Keys

| Self-assessment Questions | |
|---|---|
| Question No. | Answer |
| 1 | d |
| 2 | a and c |
| 3 | c |
| 4 | d |
| 5 | d |
| 6 | a |
| 7 | a |
| 8 | b |
| 9 | a |
| 10 | c |
| 11 | d |
| 12 | b |
| 13 | b |
| 14 | b |
| 15 | a |
| 16 | c |
| 17 | r-values and arithmetic expressions |
| 18 | c |

# Activity

**1. Activity Type: Online** **Duration: 15 Minutes**

**Description:**

a. Divide the students into two groups.

b. Give selection sort algorithm and insertion sort algorithm to each of the groups.

c. Students should calculate the time complexities for both these algorithms.

## Case Study: Exponentiation

Let us look at the implementation of exponentiation using recursion and iteration. Illustrated is a very basic algorithm for raising a base b to a non-negative integer exponent e.

```
def exp(b, e):
       result = 1
       for i in range(e):
            result = result * b
       return result
def exp(b, e):
       if e == 0:
            return 1
else:
            return b * exp(b, e-1)
```

They both have a time complexity of Θ(e). It does not really matter if it is implemented using recursion or iteration; in either case, the b is multiplied together e times.

To construct a more efficient algorithm, we can apply the principle of divide and conquer: a problem of size n can be divided into two problems of size n/2, and then those solutions can be combined to solve the problem of interest. If the time to solve a problem of size n/2 is less than half of the time to solve a problem of n, then this is a better way to go.

**Here is an exponentiation procedure that wins by doing divide-and-conquer:**

```
def fastExp(b, e):
      if e == 0:
            return 1
      elif odd(e):
            return b * fastExp(b, e-1)
else:
      return square(fastExp(b, e/2))
```

It has to handle two cases slightly differently: if e is odd, then it does a single multiply of b with the result of a recursive call with exponent e-1. Then it can compute the result for exponent e/2, and square the result. Thus time taken to compute is less.

What is the time complexity of this algorithm? Let's start by considering the case when e is a power of 2. Then we'll always hit the last case of the function, until we get down to e = 1. To compute the result of recursive calls, just log2 e. Note that notation e is our variable, not the base of the natural log.) Further, if we start with a number with a binary representation like 1111111, then we'll always hit the odd case, then the even case, then the odd case, then the even case, and it will take 2 log2 e recursive calls. Each recursive call costs a constant amount. In the end, the algorithm has time complexity of Θ(log e).

# Bibliography

## e-References

- cs.utexas.edu, (2016). *Complexity Analysis.* Retrieved on 19 April 2016, from https://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture2.html

- compsci.hunter.cuny.edu, (2016). *C Strings and Pointers.* Retrieved on 19 April 2016, from http://www.compsci.hunter.cuny.edu/~sweiss/resources/cstrings.pdf

## External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C' (2nd ed.).* Pearson Education.

- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth (2nd ed.).* BPB Publications.

- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C (2nd ed.).* Pearson Education

## Video Links

| Topic | Link |
|---|---|
| Data Structures Introduction | https://www.youtube.com/watch?v=92S4zgXN17o |
| Types of Data Structures | https://www.youtube.com/watch?v=VeEneWqC5a4 |
| Asymptotic Notation | https://www.youtube.com/watch?v=6Ol2JbwoJp0 |
| Memory Allocation | https://www.youtube.com/watch?v=Dml54J3Kwm4 |
| Variables and Addresses | https://www.youtube.com/watch?v=2RsAt8RQ194 |

# Notes: