MODULE -2 ALGORITHM ANALYSIS

STRUCTURE

1. Learning Objectives
2. Introduction
3. Space Complexity
4. Time Complexity
5. Asymptotic Notation and Algorithmic complexity
6. Abstract Data Type
7. Summary
8. Key Words/Abbreviations
9. Learning Activity
10. Unit End Questions (MCQ and Descriptive)
11. Suggested readings

LEARNING OBJECTIVES

After studying this unit, you will be able to:

- Understand the basic concepts of Algorithm Analysis

INTRODUCTION

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms –

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.

- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

SPACE COMPLEXITY

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM (A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

TIME COMPLEXITY

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

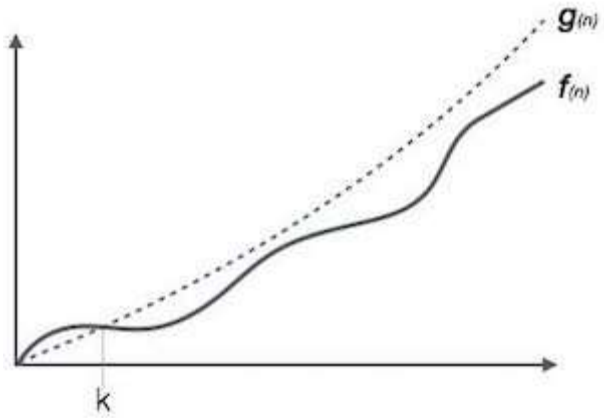
ASYMPTOTIC NOTATION AND ALGORITHMIC COMPLEXITY

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

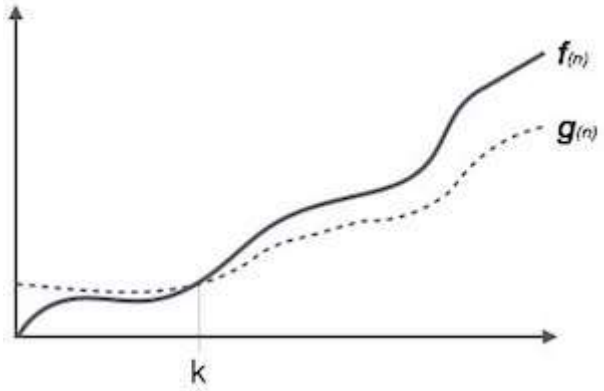


For example, for a function f(n)

O(f(n)) = {g(n): there exists c > 0 and n_0 such that f(n) ≤ c.g(n) for all n > n_0.}

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

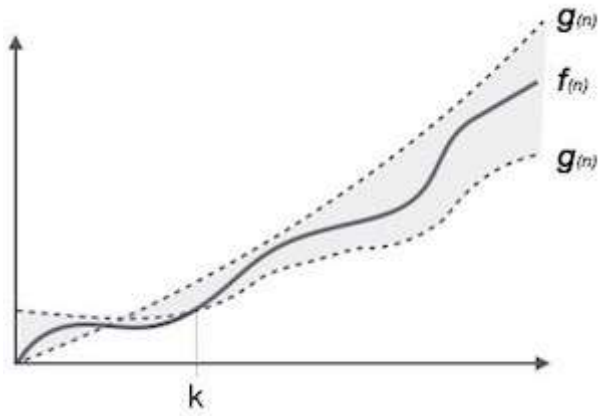


For example, for a function **f(n)**

Ω(f(n)) ≥ {g(n): there exists c > 0 and n_0 such that g(n) ≤ c.f(n) for all n > n_0.}

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



$\theta(f(n))$ = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n_0. }

Common Asymptotic Notations

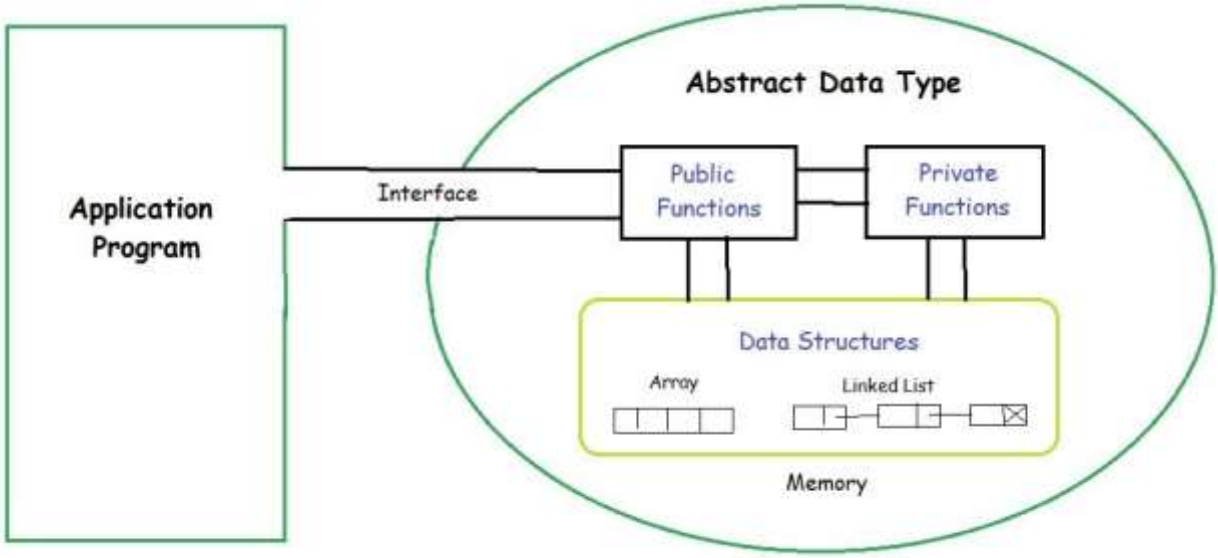
Following is a list of some common asymptotic notations −

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

ABSTRACT DATA TYPE

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

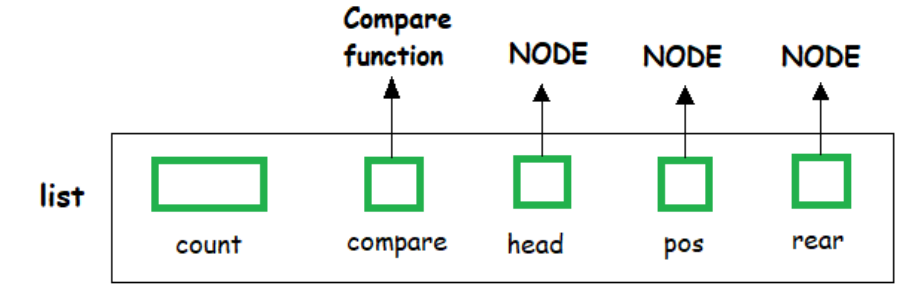
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

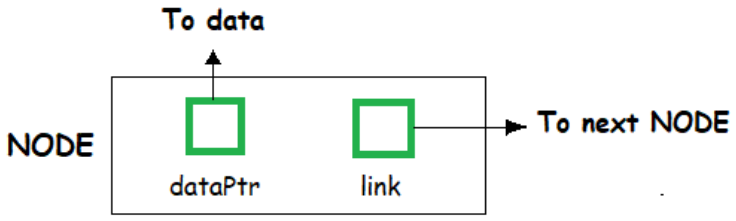


The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

List ADT

The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

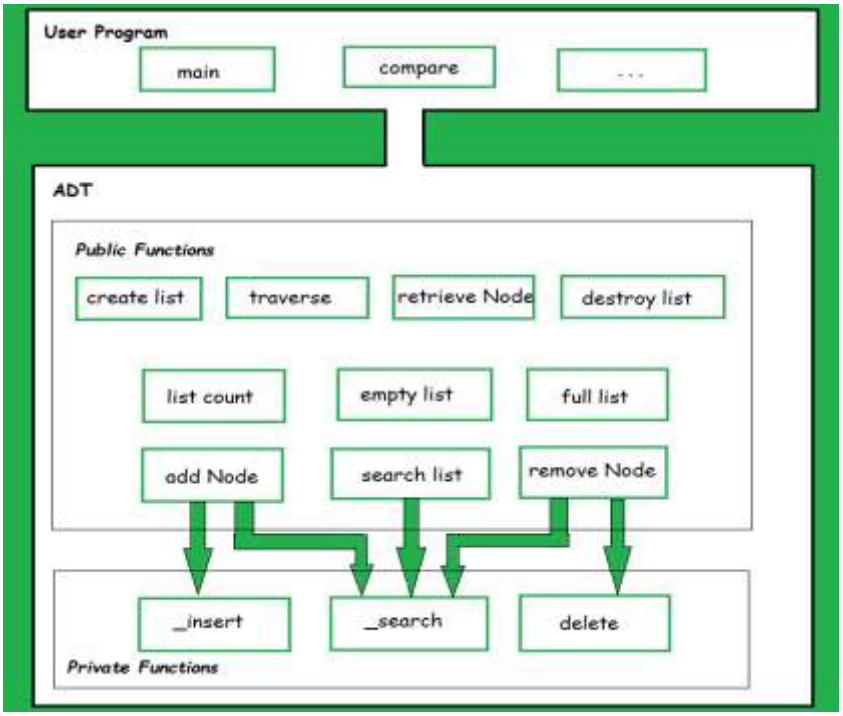




The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

```
//List ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} Node;
typedef struct
{
 int count;
 Node *pos;
 Node *head;
 Node *rear;
 int (*compare) (void *argument1, void *argument2)
} LIST;
```

The List ADT Functions is given below:



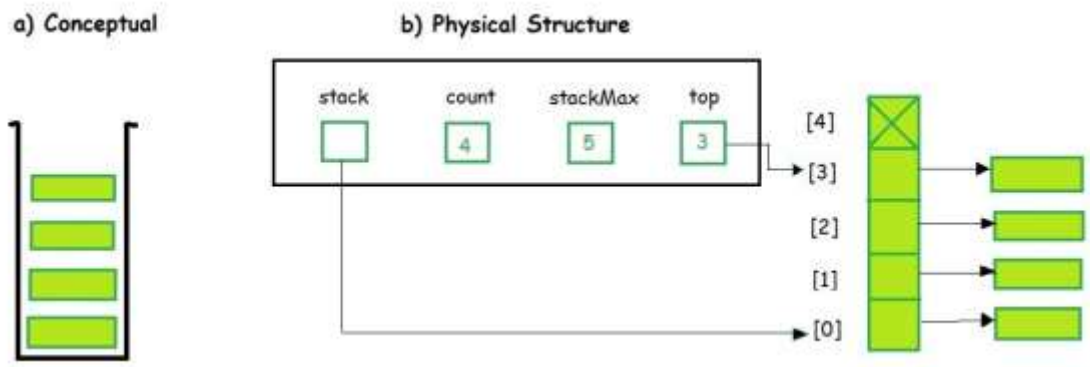
A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get () – Return an element from the list at any given position.
- insert () – Insert an element at any position of the list.
- remove () – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace () – Replace an element at any position by another element.
- size () – Return the number of elements in the list.
- isEmpty () – Return true if the list is empty, otherwise return false.
- isFull () – Return true if the list is full, otherwise return false.

Stack ADT

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.

The program allocates memory for the data and address is passed to the stack ADT.



- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and count of number of entries currently in stack.

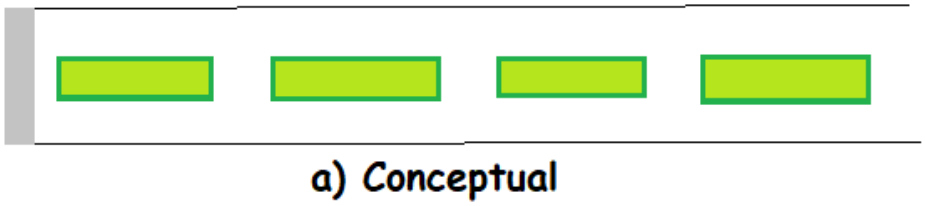
```
//Stack ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} StackNode;
typedef struct
{
 int count;
 StackNode *top;
} STACK;
```

A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

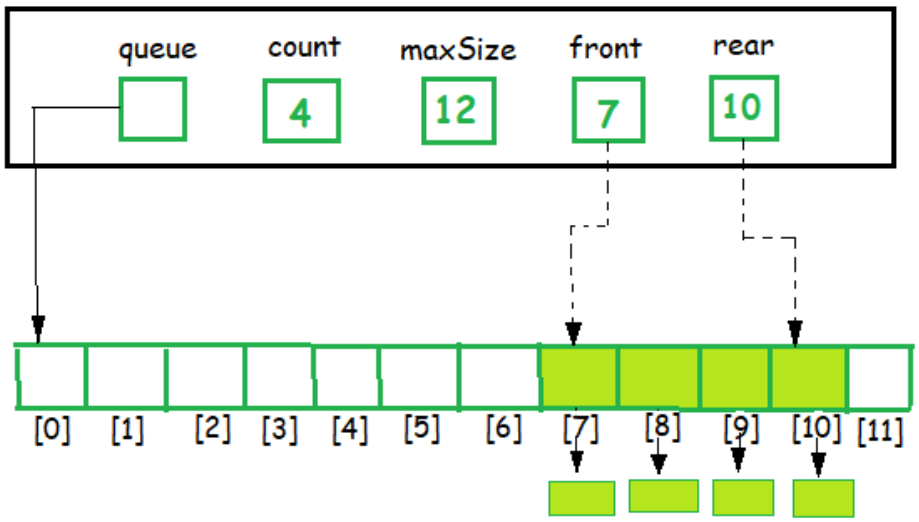
- push () – Insert an element at one end of the stack called top.
- pop () – Remove and return the element at the top of the stack, if it is not empty.
- peek () – Return the element at the top of the stack without removing it, if the stack is not empty.
- size () – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull () – Return true if the stack is full, otherwise return false.

Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



a) Conceptual



b) Physical Structures

Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

```
//Queue ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *next;
} QueueNode;
typedef struct
{
 QueueNode *front;
 QueueNode *rear;
 int count;
}
QUEUE;
```

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

- enqueue () – Insert an element at the end of the queue.
- dequeue () – Remove and return the first element of the queue, if the queue is not empty.
- peek () – Return the element of the queue without removing it, if the queue is not empty.
- size () – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull () – Return true if the queue is full, otherwise return false.
- From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

SUMMARY

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. 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.

KEY WORDS/ABBREVIATIONS

- **Abstract data type:** Abbreviated ADT. 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.

- **Address**: A location in memory.
- **ADT:** Abbreviation for abstract data type.
- **Adversary:** A fictional construct introduced for use in an adversary argument.
- **big-Oh notation:** In algorithm analysis, a shorthand notation for describing the upper bound for an algorithm or problem.

LEARNING ACTIVITY

1. Write program to allocate memory for storing the data.

……..……
……..

2. Write the program using stack ADT

……
……

UNIT END QUESTIONS (MCQ AND DESCRIPTIVE)

A. Descriptive Types Questions

1. Define ADT
2. Explain Space Complexity
3. Explain Time Complexity
4. Explain Abstract Data Type
5. Explain big Oh notation

B. Multiple Choice Questions

1. What is the time complexity of following code:

```
int a = 0, i = N;

while (i > 0)

{

   a += i;

   i /= 2;

}
```

(a)O(N)

(b) O(Sqrt(N))

(c)O (N

(d) O(log N)

2. The complexity of Binary search algorithm is

(a). O(n)

(b)O(log)

(c) O(n2)

(d). O(n log n

3. A program P reads in 500 integers in the range [0.100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies?

(a)An array of 50 numbers

(b) An array of 100 numbers

(c) array of 500 numbers

(d) A dynamically allocated array of 550 numbers

4. Which of these best describes an array?

(a)data structure that shows a hierarchical behavior

(b)Container of objects of similar types

(c) Container of objects of mixed types

(d) All of the mentioned

5.What is the output of the following piece of code?

public class array

```
{
  public static void main (String args[])
  {
    int []arr = {1,2,3,4,5};
    System.out.println(arr[2]);
```

```
        System.out.println(arr[4]);
    }
}
```

(a) 3 and 5

(b)5 and 3

(c) 2 and 4

(d) 4 and 2

Answers

1. (b), 2. (b), 3. (a), 4. (b), 5. (a)

SUGGESTED READINGS

- Weiss : Data Structures & Algorithm analysis in C++ (Pearson)
- Aho, Hopcroft & Ullman: Data Structures and Algorithms (Pearson)
- Sahni; Data Structures, Algorithms & Applications in C++ (TMH)
- An Introduction to Data Structures With Applications (Mcgraw Hill Computer Science Series) by Jean-Paul Tremblay, Paul G. Sorenson, P. G. Sorenson Doc