Data Structure

In the modern world, data and its information have significance, and there are different implementations taking place to store it in different ways. Data is simply a collection of facts and figures, or you can say that data is a set of values or values in a particular format that refers to a single set of item values. The data items are then classified into sub-items, which is the group of items that are not called the simple primary form of the item.

Let's take an example where a student's name can be broken down into three sub-items: first, middle, and last. But an ID assigned to a student will usually be considered a single item.

The example mentioned above, such as ID, Age, Gender, First, Middle, Last, Street, Area, etc., are elementary data items, whereas the Name and the Address are group data items.

What is Data Structure?

In the context of computers, the data structure is a specific way of storing and organizing data in the computer's memory so that these data can be easily retrieved and efficiently used when needed later. The data can be managed in many different ways, such as a

logical or mathematical model for a particular organization of data is called a data structure.

The variety of a specific data model depends on the two factors:

- First, it must be loaded enough into the structure to reflect the actual relationship of the data with a real-world object.
- Second, the formation should be so simple that one can efficiently process the data whenever necessary.

Categories of Data Structure

Data structures can be subdivided into two major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure

A data structure is said to be linear if its elements combine to form any specific order. There are two techniques for representing such linear structure within memory.

- The first way is to provide a linear relationship between all the elements represented using a linear memory location. These linear structures are called arrays.
- The second technique provides a linear relationship between all the elements represented using the

concept of pointers or links. These linear structures are called linked lists.

The typical examples of the linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non-linear Data Structure

This structure mainly represents data with a hierarchical relationship between different elements.

Examples of Non-Linear Data Structures are listed below:

- Graphs
- Family of trees and
- Table of contents

Tree: In this case, the data often has a hierarchical relationship between the different elements. The data structure that represents this relationship is called a rooted tree graph or tree.

Graph: In this case, the data sometimes has relationships between pairs of elements, which do not necessarily follow a hierarchical structure. Such a data structure is called a graph.

You will learn more about Linear and Non-linear Data Structures in subsequent lessons.

To do data structure and implement its various concepts in these upcoming tutorials, you must have a compiler to execute all the concepts as a program. Here, all programs of the data structure will be shown using C++. So for that, you have to install a local compiler on your PC or laptop.

C++ Compiler Setup

If you want to install the C++ compiler on your PC to perform the data structure concepts, then you have many choices. The first choice you can use a text editor such as vi / vim / gedit, or EMACS for Linux Users. For Windows, the text editors will be Notepad or Notepad++. The name and versions of text editors vary based on the operating systems.

The files you create with your text editor will be the source file and will contain the program's source code. Here you will be using C++, so the source file will have the extension as ".cpp".

Another option you can install a compiler for C++. There are various C++ compilers available online, and some of them come with GUI, such as:

For Windows

- Turbo C++

- Borland C++
- Dev C++
- Intel C++
- Visual C++

For Linux

- Open64
- GNU Compiler Collection
- Intel C++ Compiler PE

For Mac

- Apple C++
- Sun Studio
- Cygwin (GNU C++)
- Digital Mars C++

The source code that will be written into the compiler and saved as the source file is in human-readable form, which will be your data structure program. That code then needs to be "compiled" to be converted into machine language so that the CPU can actually execute the program as the code is written.

Out of all, any one of these above C++ language compilers will be required for compiling your source code into the final executable program and creating the ".exe" file. Basic knowledge about a programming language is required before approaching to grab the concepts of the data structure.

The most commonly used and free available compiler is the GNU C/C++ compiler, or you can use different compilers from Intel, Oracle, or Solaris.

Steps to Install Turbo C++ Compiler

Step 1: Turbo C++ Compiler is freely available to use; you need to search on the Internet to download it.

Step 2: Unzip the compiler that you have downloaded, i.e., the "Turbo C++ 3.2.zip" file.

Step 3: Run the "setup.exe" file.

Step 4: Select the location path to place the compiler on your PC.

Steps to Install Compiler on UNIX/Linux

Step1: Check if it is already installed or not using the command below

Step2: In case you find the GCC is not installed on your Linux system, then you need to install it by yourself using the given instructions that are available at

The present generation of digital computers is made and employed as a device that facilitates and speeds up complex and time-consuming computations. In the majority of applications or programs, it can store and access a huge amount of information take part as the dominant one and is measured to be its primary feature

and its ability to compute, that is to calculate or to carry out arithmetic, has in many cases become almost unrelated.

In most cases, the vast amount of information that is to be developed in some sense signifies a concept of a part of reality. The information that is accessible to the computer consists of a specific set of data about the real problem that is set and is considered applicable to the problem at hand. The data signifies an abstraction of reality because certain properties and distinctiveness of the real objects get ignored as they are peripheral and inappropriate to the particular problem. A concept of abstraction is thereby also an overview of facts.

In this chapter, you will learn about the fundamental elements of the data structure.

Characteristics of Data Types in Data Structure

- The data type chooses the set of values to which a constant will belong and which may be assumed by a variable or an expression within a program, or which may be produced by an operator or a function.
- The type of a value indicated by a constant or a variable or expression may result from its form or its declaration without the need of executing the computational process.

- Each operator and function expects some arguments of a fixed type which is represented by assigning a data type to those specific sets of arguments and yields a result of a fixed type. If an operator declares arguments of several types, such as the '+' will be used to add both integers and real numbers, then the type of the answer can be determined from specific language rules.

Types of Data Structures

In this case, you will be studying the concepts of the data structure using C++. The Datatypes are mainly categorized into three major types. These are:

1. Built-in data type: These types of data types are predefined and has a fixed set of rules for declaration. In other words, these data types, when belonging to a particular programming language, has built-in support, and hence they are also called built-in data types. Examples of such data types are:
    - Integer type
    - Boolean type
    - Character type
    - Floating type
2. Derived Data type: These data types can be implemented independently within a language. These data types are built by combining both primary and built-in data types and then associating

operations on them. Examples of such data types are:

- Array
- Stack
- Queue
- List

You might be familiar with these basic data types if you have read either C or C++. For dealing with the various concepts of data structures, you can use any programming language. But it is recommended to use either C or C++ for better implementation purposes.

Basic Operations of Data Structures

Some specific operations process all data in the data structures. The specific data structure that has been chosen mostly depends on the number of times the occurrence of the operation which needs to be carried out on the data structure. Names of such operations are listed below:

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

You have seen so far that data structure uses some algorithms and need storage for storing values. For

storing these values, programmers must need to have the fundamental data type's names such as char, int, float & double. As you know, these particular data types are beneficial for declaring variables, constants or a return type for a function; they are in control by the fact that, these types can store only a specific form of value at a time. For many applications, there may arise some circumstances where programmers need to have a single name to store multiple values. For processing such a large amount of data, programmers need powerful data types that would facilitate efficient storage, accessing and dealing with such data items. Using C++, you can implement the concept of arrays.

What are arrays?

The array is a fixed-size sequenced collection of variables belonging to the same data types. The array has adjacent memory locations to store values. Since the array provides a convenient structure for representing data, it falls under the category of the data structures in C. The syntax for declaring array are:

```
data_type array_name [array_size];
```

Following are the essential terminologies used for understanding the concepts of Arrays:

Element: Every item stored in an array is termed as an element

Index: each memory location of an element in an array is denoted by a numerical index which is used for identifying the element

Why Do You Need Arrays for Building a Specific Data Structure?

When a program works with many variables which hold comparable forms of data, then organizational and managerial difficulty quickly arise. If you are not using arrays, then the number of variables used will increase. Using the array, the number of variables reduces, i.e., you can use a single name for multiple values, you need to deal with its index values (starting from 0 to n).

So if the total run of each player is getting stored in separate variables, using arrays you can bring them all into one array having single name like: plrscore[11];

Collecting Input Data in Arrays

Arrays are particularly helpful for making a collection of input data which arrive in random order. An excellent example will be vote counting: You can write a program which tallies the votes of a four-candidate in an election. (For your ease, you will say use the candidates' names as Cand 0, Cand 1, Cand 2, and Cand 3.) Votes arrive once

at a time, where a vote for Candidate i is denoted by the number, i. So according to this example, two votes for Cand 3 followed by one vote for Cand 0 would appear:

```
3

3

0
and so on....
```

i.e., the structure will be:

```
int votes[4];
```

Basic Operations

There is some specific operation that can be performed or those that are supported by the array. These are:

- Traversing: It prints all the array elements one after another.
- Inserting: It adds an element at given index.
- Deleting: It is used to delete an element at given index.
- Searching: It searches for an element(s) using given index or by value.
- Updating: It is used to update an element at given index.

Polynomials and Sparse Matrix are two important applications of arrays and linked lists. A polynomial is

composed of different terms where each of them holds a coefficient and an exponent. This tutorial chapter includes the representation of polynomials using linked lists and arrays.

What is Polynomial?

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one

- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

Representation of Polynomial

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

Representation of Polynomials Using Arrays

There may arise some situation where you need to evaluate many polynomial expressions and perform basic arithmetic operations like addition and subtraction with those numbers. For this, you will have to get a way to represent those polynomials. The simple way is to represent a polynomial with degree 'n' and store the coefficient of n+1 terms of the polynomial in the array. So every array element will consist of two values:

- Coefficient and

- Exponent

A polynomial can be represented using the C++ code:

```
#include <iostream>
#include <iomanip.h>
using namespace std;

struct poly {
   int coeff;
   int pow_val;
   poly* next;
};

class add {
   poly *poly1, *poly2, *poly3;

public:
   add() { poly1 = poly2 = poly3 = NULL; }
   void addpoly();
   void display();
};

void add::addpoly()
{
   int i, p;
   poly *newl = NULL, *end = NULL;
   cout << "Enter highest power for x\n"; cin >> p;
   //Read first poly
   cout << "\nFirst Polynomial\n"; for (i = p; i >= 0; i--) {
      newl = new poly;
      newl->pow_val = p;
```

```
      cout << "Enter Co-efficient for degree" << i << ":: "; cin
>> newl->coeff;
      newl->next = NULL;
      if (poly1 == NULL)
         poly1 = newl;
      else
         end->next = newl;
      end = newl;
   }

   //Read Second poly
   cout << "\n\nSecond Polynomial\n"; end = NULL; for (i
= p; i >= 0; i--) {
      newl = new poly;
      newl->pow_val = p;
      cout << "Enter Co-efficient for degree" << i << ":: "; cin
>> newl->coeff;
      newl->next = NULL;
      if (poly2 == NULL)
         poly2 = newl;
      else
         end->next = newl;
      end = newl;
   }

   //Addition Logic
   poly *p1 = poly1, *p2 = poly2;
   end = NULL;
   while (p1 != NULL && p2 != NULL) {
      if (p1->pow_val == p2->pow_val) {
         newl = new poly;
         newl->pow_val = p--;
         newl->coeff = p1->coeff + p2->coeff;
```

```
            newl->next = NULL;
            if (poly3 == NULL)
                poly3 = newl;
            else
                end->next = newl;
            end = newl;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
}

void add::display()
{
    poly* t = poly3;
    cout << "\n\nAnswer after addition is : ";
    while (t != NULL) {
        cout.setf(ios::showpos);
        cout << t->coeff;
        cout.unsetf(ios::showpos);
        cout << "X" << t->pow_val;
        t = t->next;
    }
}
int main()
{
    add obj;
    obj.addpoly();
    obj.display();
}
```

Output:

Representation of Polynomial Using Linked Lists

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

- The exponent part
- The coefficient part

Program for the addition of Polynomial

```
#include <iostream>
using namespace std;

class polyll {
private:
   struct polynode {
      float coeff;
      int exp;
      polynode* link;
   } * p;
```

```
public:
    polyll();
    void poly_append(float c, int e);
    void display_poly();
    void poly_add(polyll& l1, polyll& l2);
    ~polyll();
};
polyll::polyll()
{
    p = NULL;
}
void polyll::poly_append(float c, int e)
{
    polynode* temp = p;
    if (temp == NULL) {
        temp = new polynode;
        p = temp;
    }
    else {
        while (temp->link != NULL)
            temp = temp->link;
        temp->link = new polynode;
        temp = temp->link;
    }
    temp->coeff = c;
    temp->exp = e;
    temp->link = NULL;
}
void polyll::display_poly()
{
    polynode* temp = p;
    int f = 0;
```

```
   cout << endl; while (temp != NULL) { if (f != 0) { if (temp-
>coeff > 0)
            cout << " + ";
         else
            cout << " "; } if (temp->exp != 0)
         cout << temp->coeff << "x^" << temp->exp;
      else
         cout << temp->coeff;
      temp = temp->link;
      f = 1;
   }
}
void polyll::poly_add(polyll& l1, polyll& l2)
{
   polynode* z;
   if (l1.p == NULL && l2.p == NULL)
      return;
   polynode *temp1, *temp2;
   temp1 = l1.p;
   temp2 = l2.p;
   while (temp1 != NULL && temp2 != NULL) {
      if (p == NULL) {
         p = new polynode;
         z = p;
      }
      else {
         z->link = new polynode;
         z = z->link;
      }
      if (temp1->exp < temp2->exp) {
         z->coeff = temp2->coeff;
         z->exp = temp2->exp;
         temp2 = temp2->link;
```

```
    }
    else {
      if (temp1->exp > temp2->exp) {
        z->coeff = temp1->coeff;
        z->exp = temp1->exp;
        temp1 = temp1->link;
      }
      else {
        if (temp1->exp == temp2->exp) {
          z->coeff = temp1->coeff + temp2->coeff;
          z->exp = temp1->exp;
          temp1 = temp1->link;
          temp2 = temp2->link;
        }
      }
    }
  }
  while (temp1 != NULL) {
    if (p == NULL) {
      p = new polynode;
      z = p;
    }
    else {
      z->link = new polynode;
      z = z->link;
    }
    z->coeff = temp1->coeff;
    z->exp = temp1->exp;
    temp1 = temp1->link;
  }
  while (temp2 != NULL) {
    if (p == NULL) {
      p = new polynode;
```

```
            z = p;
        }
        else {
            z->link = new polynode;
            z = z->link;
        }
        z->coeff = temp2->coeff;
        z->exp = temp2->exp;
        temp2 = temp2->link;
    }
    z->link = NULL;
}
polyll::~polyll()
{
    polynode* q;
    while (p != NULL) {
        q = p->link;
        delete p;
        p = q;
    }
}
int main()
{
    polyll p1;
    p1.poly_append(1.4, 5);
    p1.poly_append(1.5, 4);
    p1.poly_append(1.7, 2);
    p1.poly_append(1.8, 1);
    p1.poly_append(1.9, 0);
    cout << "\nFirst polynomial:";
    p1.display_poly();
    polyll p2;
    p2.poly_append(1.5, 6);
```

```
    p2.poly_append(2.5, 5);
    p2.poly_append(-3.5, 4);
    p2.poly_append(4.5, 3);
    p2.poly_append(6.5, 1);
    cout << "\nSecond polynomial:";
    p2.display_poly();
    polyll p3;
    p3.poly_add(p1, p2);
    cout << "\nResultant polynomial: ";
    p3.display_poly();
    getch();
}
```

In this chapter, you will explore one of the most important data structures which are used in many fields of programming and data handling, i.e., the Stack. It falls under the category of an abstract data type which serves as a concrete and valuable tool for problem-solving. In this chapter, you will study the various operations and working technique of stack data structure.

What is stack?

A stack is a linear data structure in which all the insertion and deletion of data or you can say its values are done at one end only, rather than in the middle. Stacks can be implemented by using arrays of type linear.

The stack is mostly used in converting and evaluating expressions in Polish notations, i.e.:

- Infix
- Prefix
- Postfix

In case of arrays and linked lists, these two allows programmers to insert and delete elements from any place within the list, i.e., from the beginning or the end or even from the middle also. But in computer programming and development, there may arise some situations where insertion and deletion require only at one end wither at the beginning or end of the list. The stack is a linear data structure, and all the insertion and deletion of its values are done in the same end which is called the top of the stack. Let us suppose take the real-life example of a stack of plates or a pile of books etc. As the item in this form of data structure can be removed or added from the top only which means the last item to be added to the stack is the first item to be removed. So you can say that the stack follows the Last In First Out (LIFO) structure.

Stack as Abstract Data Type

He stacks of elements of any particular type is a finite sequence of elements of that type together with the following operations:

- Initialize the stack to be empty

- Determine whether the stack is empty or not
- Check whether the stack is full or not
- If the stack is not full, add or insert a new node at the top of the stack. This operation is termed as Push Operation
- If the stack is not empty, then retrieve the node at its top
- If the stack is not empty, the delete the node at its top. This operation is called as Pop operation

Representation of Stack using Arrays

The stack can be represented in memory with the use of arrays. To do this job, you need to maintain a linear array STACK, a pointer variable top which contains the top element.

Example:

```
#include <iostream>
#include<stdlib.h>
using namespace std;
```

```
class stack {
   int stk[5];
   int top;

public:
   stack()
   {
      top = -1;
   }
   void push(int x)
   {
      if (top >= 4) {
         cout << "stack overflow";
         return;
      }
      stk[++top] = x;
      cout << "inserted " << x;
   }
   void pop()
   {
      if (top < 0) {
         cout << "stack underflow";
         return;
      }
      cout << "deleted " << stk[top--];
   }
   void display()
   {
      if (top < 0) {
         cout << " stack empty"; return; } for (int i = top; i >=
0; i--)
         cout << stk[i] << " ";
```

```
    }
};


int main()
{
    int ch;
    stack st;
    while (1) {
        cout << "\n1.push 2.pop 3.display 4.exit\nEnter ur
choice: "; cin >> ch;
        switch (ch) {
        case 1:
            cout << "enter the element: "; cin >> ch;
            st.push(ch);
            break;
        case 2:
            st.pop();
            break;
        case 3:
            st.display();
            break;
        case 4:
            exit(0);
        }
    }
}
```

Output:

The queue is a linear data structure used to represent a linear list. It allows insertion of an element to be done at

one end and deletion of an element to be performed at the other end.

In this chapter, you will be given an introduction to the basic concepts of queues along with the various types of queues which will be discussed simulating the real world situation.

What is a Queue?

A queue is a linear list of elements in which deletion of an element can take place only at one end called the front and insertion can take place on the other end which is termed as the rear. The term front and rear are frequently used while describing queues in a linked list. In this chapter, you will deal with the queue as arrays.

In the concept of a queue, the first element to be inserted in the queue will be the first element to be deleted or removed from the list. So Queue is said to follow the FIFO (First In First Out) structure. A real-life scenario in the form of example for queue will be the queue of people waiting to accomplish a particular task where the first person in the queue is the first person to be served first.

Other examples can also be noted within a computer system where the queue of tasks arranged in the list to perform for the line printer, for accessing the disk storage, or even in the time-sharing system for the use of CPU. So basically queue is used within a single program where there are multiple programs kept in the queue or one task may create other tasks which must have to be executed in turn by keeping them in the queue.

Queue as an ADT (Abstract Data Type)

The meaning of an abstract data type clearly says that for a data structure to be abstract, it should have the below-mentioned characteristics:

- First, there should be a particular way in which components are related to each other
- Second, a statement for the operation that can be performed on elements of abstract data type must have to be specified

Thus for defining a Queue as an abstract data type, these are the following criteria:

- Initialize a queue to be empty
- Check whether a queue is empty or not
- Check whether a queue is full or not
- Insert a new element after the last element in a queue, if the queue is not full

- Retrieve the first element of the queue, if it is not empty
- Delete the first element in a queue, if it is not empty

Representation of Queue as an Array

Queue is a linear data structure can be represented by using arrays. Here is a program showing the implementation of a queue using an array.

Example:

```
#include <iostream>
#include<stdlib.h>
using namespace std;

class queuearr {
   int queue1[5];
   int rear, front;

public:
   queuearr()
   {
      rear = -1;
      front = -1;
   }
   void insert(int x)
   {
      if (rear > 4) {
         cout << "queue over flow";
         front = rear = -1;
         return;
      }
```

```
        queue1[++rear] = x;
        cout << "inserted " << x;
    }

    void delet()
    {
        if (front == rear) {
            cout << "queue under flow";
            return;
        }
        cout << "deleted " << queue1[++front];
    }

    void display()
    {
        if (rear == front) {
            cout << " queue empty";
            return;
        }
        for (int i = front + 1; i <= rear; i++)
            cout << queue1[i] << " ";
    }
};

int main()
{
    int ch;
    queuearr qu;
    while (1) {
        cout << "\n1.insert 2.delet 3.display 4.exit\nEnter ur
choice: "; cin >> ch;
        switch (ch) {
        case 1:
```

```
        cout << "enter the element: "; cin >> ch;
        qu.insert(ch);
        break;
      case 2:
        qu.delet();
        break;
      case 3:
        qu.display();
        break;
      case 4:
        exit(0);
      }
   }
}
```

Output:

This chapter starts with the basic information regarding the fundamental knowledge required to solve various problems. Algorithm design is one of the primary steps in solving problems. Algorithms are set of steps or instructions required and designed to solve a specific problem.

What is Software Development Life Cycle?

Developing good software is a tedious process which keeps on going i.e. under development for a long time before the software or the program takes the final shape. This process is often termed as Software Development Life Cycle (SDLC). Here, the output of one stage becomes the input of next stage.

The various steps involved in the Software Development Life Cycle are as follows:

- Analyze the problem with precision

- Create a prototype and experiment with it until all requirements are finalized
- Design an algorithm for the task using the tools of the data structure
- Verify the algorithmic steps
- Analyze the algorithm for checking its requirements
- Code the algorithm to any suitable programming language
- Test and evaluate the code
- Refine and repeat the preceding steps until the software is complete
- Optimize the code to improve performance
- Maintain the application that you have designed so that it meets the upcoming client's and users need

Program Design

Program design is an important stage of software development. This phase takes the help of algorithms and different concepts of data structures to solve the problem(s) that is proposed.

Different Approaches to Designing Algorithms

Complex code can be subdivided into smaller units called modules. The advantage of modularity is that it allows the principle of separation of concerns to be applied into two phases are -

- While dealing with details of each module in isolation
- While dealing with overall characteristics of all modules and their relationships

Modularity enhances design clarity, which in turn eases implementation and readability. Debugging, testing,

documenting and maintenance of product also increase due to modularity.

What Do You Mean by Complexity of Algorithm?

Complexity is an essential concept in Data structure. When you talk about complexity is related to computer, you call it as computational complexity. It can be termed as the characterization of time and space requirements for solving a problem using some specific algorithm. These requirements are expressed regarding one single parameter which is used to represent the size of the problem.

Example:

Let 'n' denotes the size of the problem. Then the time required for a specific type of algorithm for solving a problem can be expressed as:

f : R -> R, where f is the function and f(n) is the most significant amount of time needed. Thus you can conclude that analysis of any program requires two vital concepts:

- Time Complexity
- Space Complexity

Time Complexity of a program can be defined as the amount of time the computer takes to run a program to its completion. On the other hand, the space complexity of an algorithm can be defined as the memory that it needs to run that algorithm to its completion.

In this chapter, you will learn about the different algorithmic approaches that are usually followed while programming or designing an algorithm. Then you will

get the basic idea of what Big-O notation is and how it is used. Finally, there will be brief lists of the different types of algorithmic analysis that is being performed using the types of complexity.

Various Approaches to Algorithmic Design

Any system can have components which have components of their own. Certainly, a system is a hierarchy of components. The highest level of components corresponds to the total system. There are usually two approaches to design such hierarchy:

1. Top-down approach
2. Bottom-up approach

Now let's discuss both of them:

Top-down approach

The top-down approach starts by identifying the major components of the system or program decomposing them into their lower level components and iterating until the desired level of modular complexity is achieved. The top-down method takes the form of stepwise working and refinement of instructions. Thus the top-down approach starts from an abstract design, and each step is refined into more concrete level until the final refined stage is not reached.

Bottom-up approach

The bottom-up design starts with designing the most basic or primitive components and proceeds to the higher level component. It works with layers of abstraction. Starting from below, the operation that provides a layer of abstraction is implemented. These operations are further used to implement more powerful operations and still higher layers of abstraction until the final stage is reached.

What is Big - O notation?

When resolving a computer-related problem, there will frequently be more than just one solution. These individual solutions will often be in the shape of different algorithms or instructions having different logic, and you will normally want to compare the algorithms to see which one is more proficient. So, this is where Big O analysis helps program developers to give programmers some basis for computing and measuring the efficiency of a specific algorithm.

If f(n) represents the computing time of some algorithm and g(n) represents a known standard function like n, n^2, n log n, then to write:

f(n) is O g(n)

which means that f(n) of n equals to the biggest order of the function, i.e., the g(n).

So what Big - O does? It helps to determine the time as well as space complexity of the algorithm. Using Big - O notation, the time taken by the algorithm and the space required to run the algorithm can be ascertained. Some of the lists of common computing times of algorithms in order of performance are as follows:

- O (1)
- O (log n)
- O (n)
- O (nlog n)
- O (n^2)
- O (n^3)
- O (2^n)

Thus algorithm with their computational complexity can be rated as per the mentioned order of performance.

Algorithm Analysis

In the last chapter, you have studied about the time and space complexity. This complexity is used to analyze the algorithm in the data structure. There are various ways of solving a problem and there exists different algorithms which can be designed to solve the problem.

Consequently, analysis of algorithms focuses on the computation of space and time complexity. Here are various types of time complexities which can be analyzed for the algorithm:

- Best case time complexity: The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' The running time of many algorithms varies not only for the inputs of different sizes but also for the different inputs of the same size.
- Worst case time Complexity: The worst case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' Therefore, if various algorithms for sorting are taken into account and say 'n,' input data items are supplied in reverse order for a sorting algorithm, then the algorithm will require n^2 operations to perform the sort which will correspond to the worst case time complexity of the algorithm.
- Average Time complexity Algorithm: This is the time that the algorithm will require to execute a typical input data of size 'n' is known as the average case time complexity.

This chapter explores various searching techniques. The process of identifying or finding a particular record is called Searching. You often spend time in searching for any desired item. If the data is kept properly in sorted order, then searching becomes very easy and efficient. In

this chapter, you will get to know the basic concepts of searching that are used in the data structure and case of programming also.

What is searching?

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

What is Linear Search?

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0^{th} element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0^{th} element and continues until the element or the end of the list is reached.

Example:

The list given below is the list of elements in an unsorted array. The array contains ten elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0^{th} element, and the searching process ends where 46 is found, or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparison is 0(n).

Algorithm for Linear Search

It is a simple algorithm that searches for a specific item inside a list. It operates looping on each element O(n) unless and until a match occurs or the end of the array is reached.

- algorithm Seqnl_Search(list, item)
- Pre: list != ;
- Post: return the index of the item if found, otherwise: 1

- index <- fi
- while index < list.Cnt and list[index] != item //cnt: counter variable
- index <- index + 1
- end while
- if index < list.Cnt and list[index] = item
- return index
- end if
- return: 1
- end Seqnl_Search

What is Binary Search?

Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the center of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0^{th} element to the middle element which is the center element (first half) another from the center element to the last element (which is the 2^{nd} half) where all values are greater than the center element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.

Algorithm for Binary Search

- algorithm Binary_Search(list, item)
- Set L to 0 and R to n: 1
- if L > R, then Binary_Search terminates as unsuccessful
- else
- Set m (the position in the mid element) to the floor of (L + R) / 2
- if Am < T, set L to m + 1 and go to step 3
- if Am > T, set R to m: 1 and go to step 3
- Now, Am = T,
- the search is done; return (m)