ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT I : Introduction to Data Structures (part – 1)

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

What is data ?

Def :-

- the quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

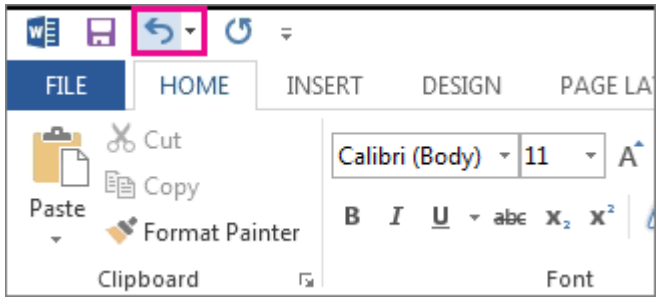
What is information ?

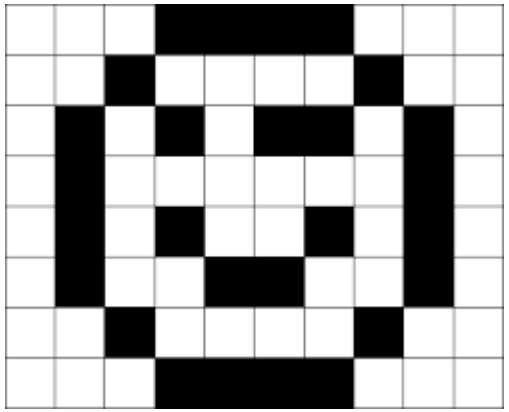
- Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.

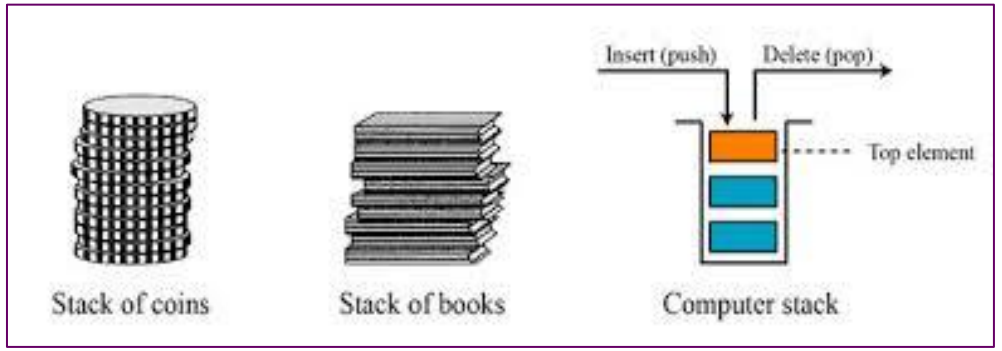
What is Data Structure ?

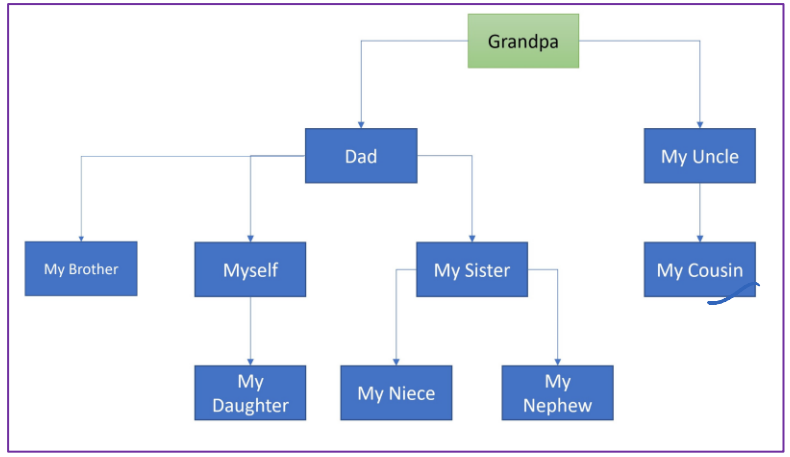
- Data structure is a crucial part of data management and in this book it will be our prime concern. A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Examples of Data Structure

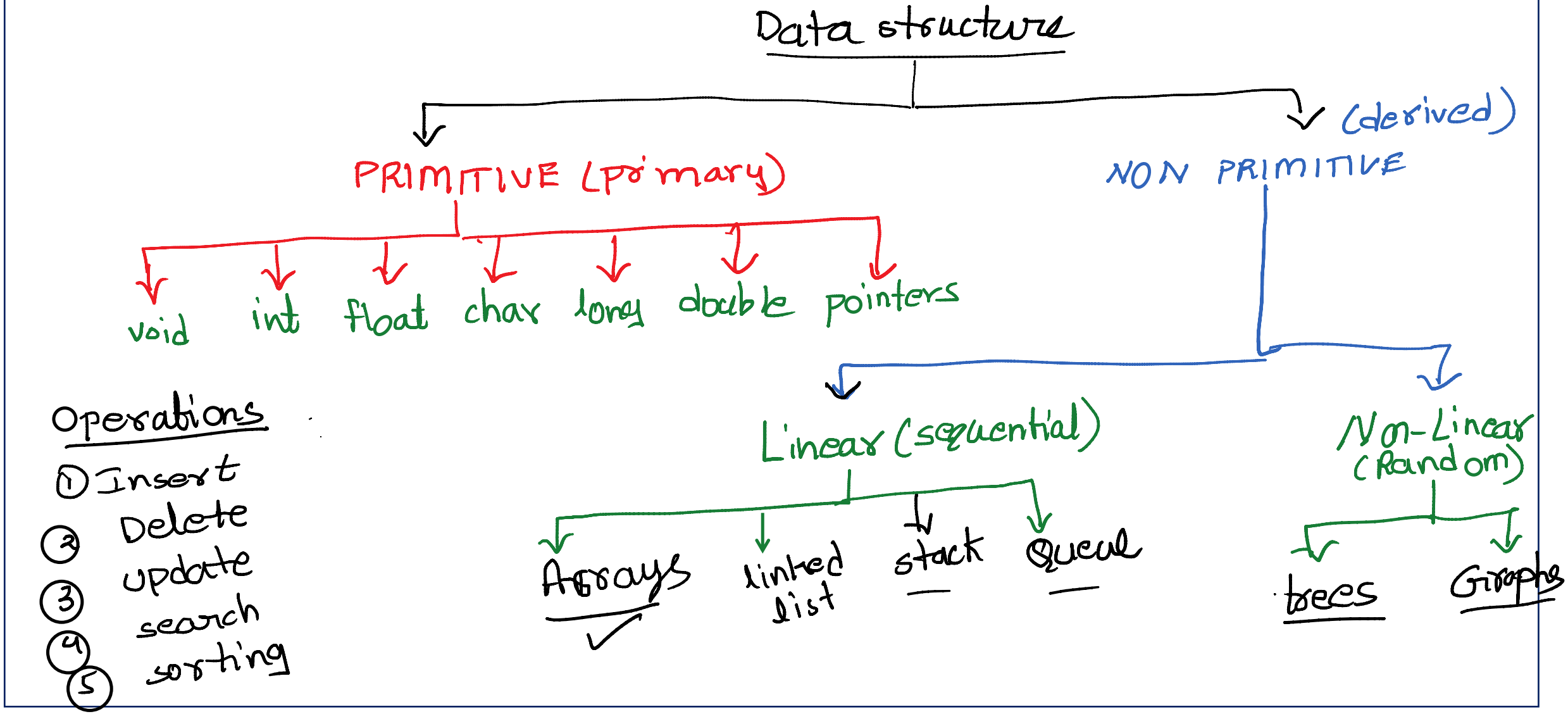








Classification of Data Structure



Syllabus

UNIT I

Data Structures - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity.

Searching - Linear search, Binary search, Fibonacci search.

Sorting- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

UNIT II

Linked List: Introduction, Single linked list, Representation of Linked list in memory, Operations on Single Linked list- Insertion, Deletion, Search and Traversal ,Reversing Single Linked list,

Applications on Single Linked list- Polynomial Expression Representation ,Addition and Multiplication, Sparse Matrix Representation using Linked List, Advantages and Disadvantages of Single Linked list,

Double Linked list-Insertion, Deletion, Circular Linked list-Insertion, Deletion.

Syllabus

UNIT III

Queues: Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues Circular Queues, Deques, Priority Queues, Multiple Queues.

Stacks: Introduction to Stacks, Array Representation of Stacks, Operations on Stacks, Linked list Representation of Stacks, Operations on Linked Stack, Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

UNIT IV

Trees: Basic Terminology in Trees, Binary Trees-Properties, Representation of Binary Trees using Arrays and Linked lists. Binary Search Trees- Basic Concepts, BST Operations: Insertion, Deletion, Tree Traversals, Applications-Expression Trees, Heap Sort, Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations.

UNIT V

Graphs: Basic Concepts, Representations of Graphs-Adjacency Matrix and using Linked list, Graph Traversals (BFT & DFT), Applications- Minimum Spanning Tree Using Prims & Kruskals Algorithm, Dijkstra's shortest path, Transitive closure, Warshall's Algorithm.

DATA STRUCTURES LAB

Exercise -1 (Searching)

a)Write C program that use both recursive and non recursive functions to perform Linear search for a Key value in a given list.

b) Write C program that use both recursive and non recursive functions to perform Binary search for a Key value in a given list.

Exercise -2 (Sorting-I)

a) Write C program that implement Bubble sort, to sort a given list of integers in ascending order

b) Write C program that implement Quick sort, to sort a given list of integers in ascending order

c) Write C program that implement Insertion sort, to sort a given list of integers in ascending order

Exercise -3(Sorting-II)

a) Write C program that implement radix sort, to sort a given list of integers in ascending order

b) Write C program that implement merge sort, to sort a given list of integers in ascending order

Exercise -4(Singly Linked List)

a) Write a C program that uses functions to create a singly linked list

b) Write a C program that uses functions to perform insertion operation on a singly linked list

c) Write a C program that uses functions to perform deletion operation on a singly linked list

d) Write a C program to reverse elements of a single linked list.

DATA STRUCTURES LAB

Exercise -5(Queue)

a) Write C program that implement Queue (its operations) using arrays.

b) Write C program that implement Queue (its operations) using linked lists

Exercise -6(Stack)

a) Write C program that implement stack (its operations) using arrays

b) Write C program that implement stack (its operations) using Linked list

c) Write a C program that uses Stack operations to evaluate postfix expression

Exercise -7(Binary Tree)

a) Write a recursive C program for traversing a binary tree in preorder, inorder and postorder.

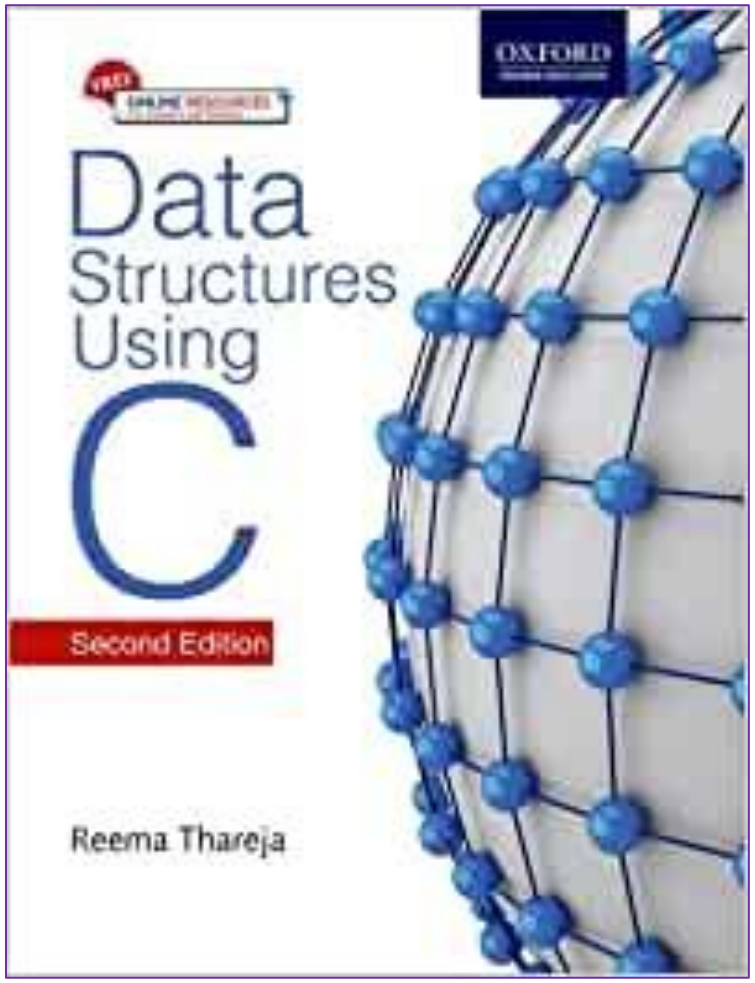
Exercise -8(Binary Search Tree)

a) Write a C program to Create a BST

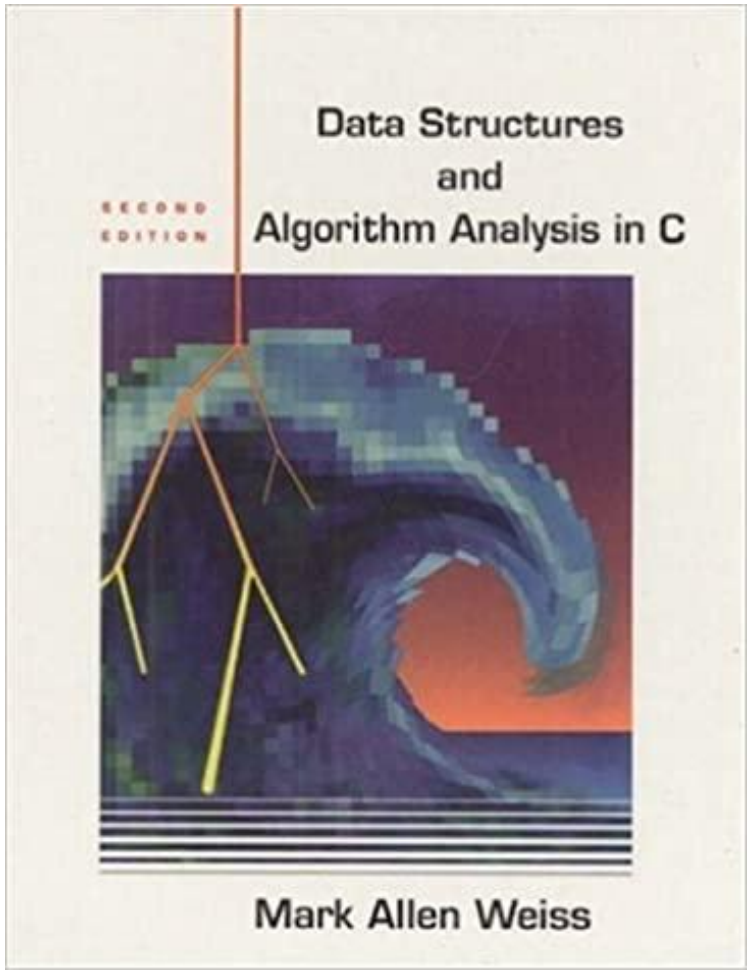
b) Write a C program to insert a node into a BST.

c) Write a C program to delete a node from a BST.

Text Books



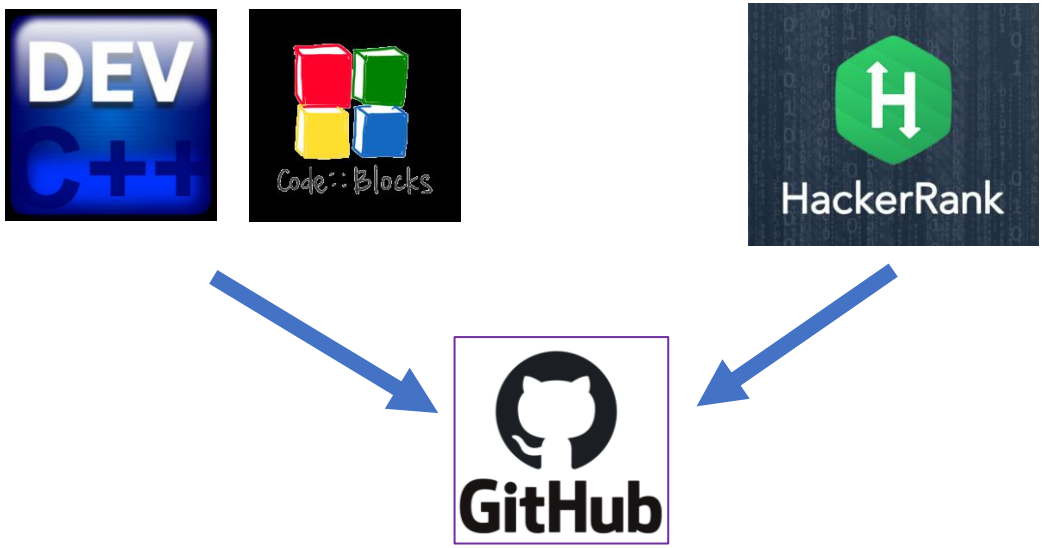
Data Structures Using C. 2nd Edition. Reema Thareja, Oxford.



Data Structures and algorithm analysis in C, 2nded, Mark Allen Weiss.

Process study

- **Understand** concepts (by using ppts or pdf).
- **Visualizing** the concepts.
- **Implementing** concepts by writing code (dev c++ or code blocks).
- Practising Data structure problems in **Hacker rank**.
- Adding to your **Git hub** repository.



Theory Exams
→ Def, Description
→ Algorithms
→ Examples
→ Programs

How to use Git Hub

What is GitHub?

GitHub is an immense platform for code hosting. It supports version controlling and collaboration and allows developers to work together on projects. It offers both distributed version control and source code management (SCM) functionality of Git. It also facilitates collaboration features such as bug tracking, feature requests, task management for every project.

Essential components of the GitHub are:

- Repositories, Branches, Commits, Pull Requests, Git (the version control tool GitHub is built on)

GitHub Education

- GitHub education offers free access to various developer tools with GitHub partners. It provides real-world experience.
- We can create a project in our college days on GitHub and show creativity to the world. We can collaborate with public repositories of other companies and impress them.

For further information you can refer to the following site.

https://www.javatpoint.com/github

Contents

Data Structures

- Definition
- Classification of Data Structures
- Operations on Data Structures
- Abstract Data Type (ADT)
- Preliminaries of algorithms.
- Time and Space complexity.

Searching

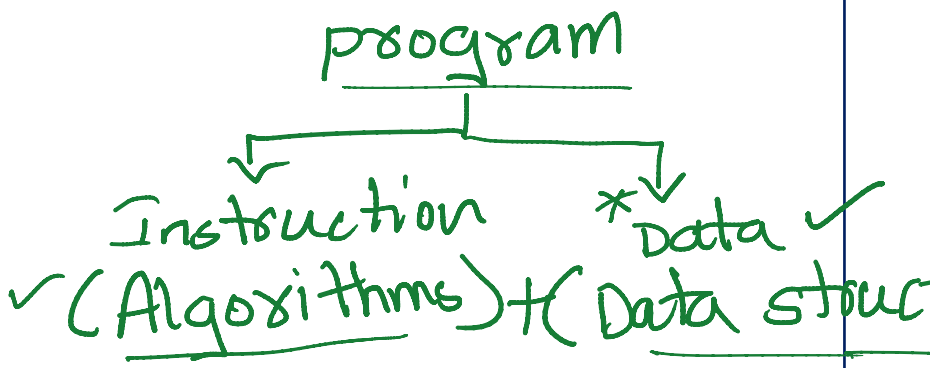
- Linear search
- Binary search
- Fibonacci search

Sorting

- Insertion sort,
- Selection sort,
- Exchange (Bubble sort, quick sort),
- distribution (radix sort),
- merging (Merge sort) algorithms.

Definition of Data Structure

- A **data structure** is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently
- Different kinds of data structures are meant for different kinds of applications, and some are highly specialized to specific tasks. Based on the organizing method of a data structure.
- Data structures are important for the following reasons:
  - Data structures are used in almost every program or software system.
  - Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large integrated collection of databases.
  - Some programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.
- Data structures are widely applied in the following areas:
  - Compiler design
  - Operating system
  - Statistical analysis package
  - DBMS
  - Numerical analysis
  - Simulation
  - Artificial intelligence
  - Graphics



Introduction to Data Structure

- Today computer programmers do not write programs just to solve a problem but to write an **efficient program**.
- For this, they first analyse the problem to determine the performance goals that must be achieved and then think of the most **appropriate data structure** for that job.
- When selecting a data structure to solve a problem, the following steps must be performed.

  1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
  2. Quantify the resource constraints for each operation.
  3. Select the data structure that best meets these requirements.
- This three-step approach to select an appropriate data structure for the problem at hand supports a data- centered view of the design process.

Classification of Data Structures

primary derived

- Data structures are generally categorized into two classes: **primitive** and **non-primitive** data structures.

Primitive and Non-primitive Data Structures

- **Primitive data structures** are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean.

- **Non-primitive data structures** are those data structures which are created using primitive data structures.
- **Examples** of such data structures include linked lists, stacks, trees, and graphs.

- Non-primitive data structures can further be classified into two categories: **linear** and **non-linear** data structures.

Linear and Non-linear Structures

Linear and Non-linear Structures

- If the elements of a data structure are stored in a **linear** or **sequential** order, then it is a **linear data structure**.

  Examples include arrays, linked lists, stacks, and queues.

- **Linear data structures** can be represented in memory in **two different ways**. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

- if the elements of a data structure are not stored in a sequential order, then it is a **non-linear data structure**. The relationship of adjacency is not maintained between elements of a non-linear data structure.

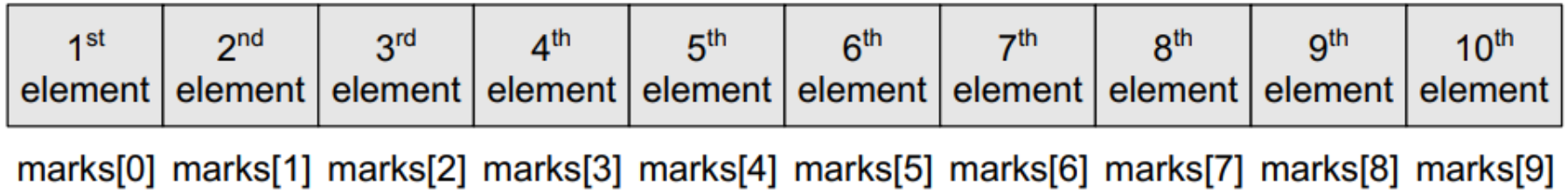
  Examples include trees and graphs.

1. Arrays

- An array is a collection of **similar data elements**. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).

For **example**,

int marks[10];



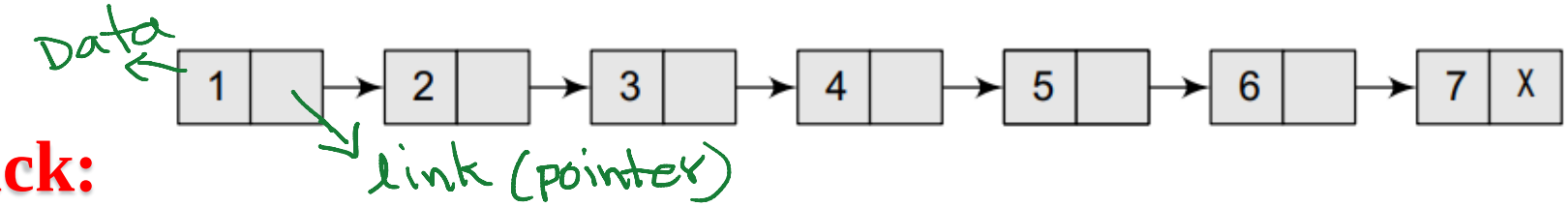
- Arrays are generally used when we want to store large amount of similar type of data.
- But they have the following **limitations**:

1. Arrays are of **fixed size**.
2. Data elements are stored in **contiguous memory locations** which may not be always available.
3. Insertion and deletion of elements can be problematic because of **shifting of elements** from their positions.

- However, these limitations can be solved by using linked lists

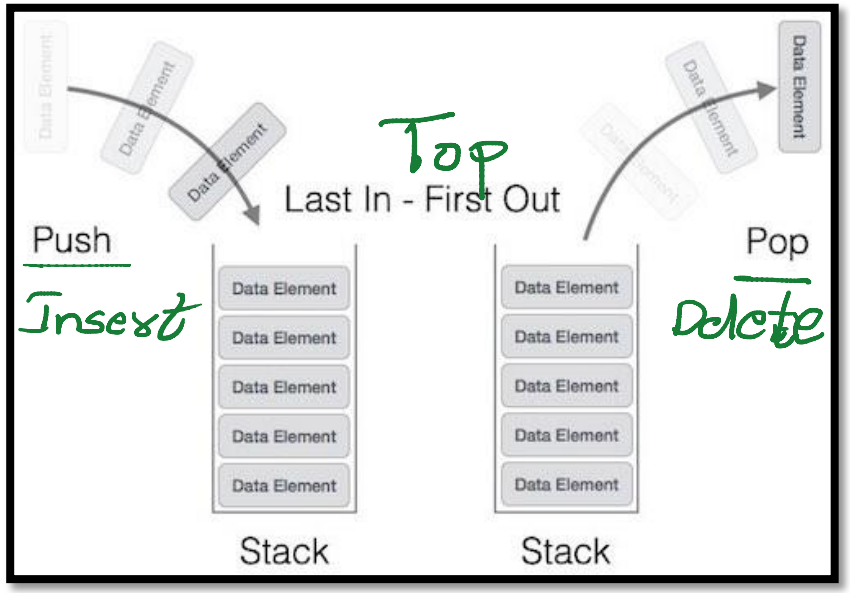
2. Linked list:

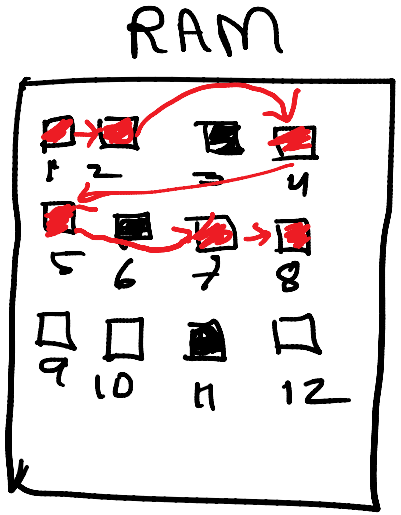
- A linked list is a **linear data structure**, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



3. Stack:

- A stack is a linear data structure in which insertion and deletion of elements are done at **only one end**, which is known as the **top of the stack**.
- Stack is called a **last-in, first-out (LIFO)**

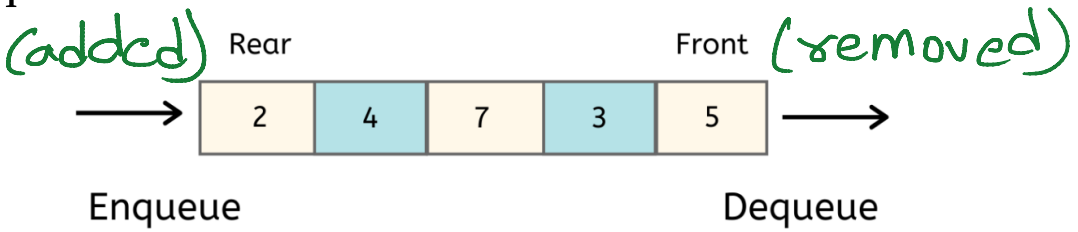




linked list stores as non-contiguous memory

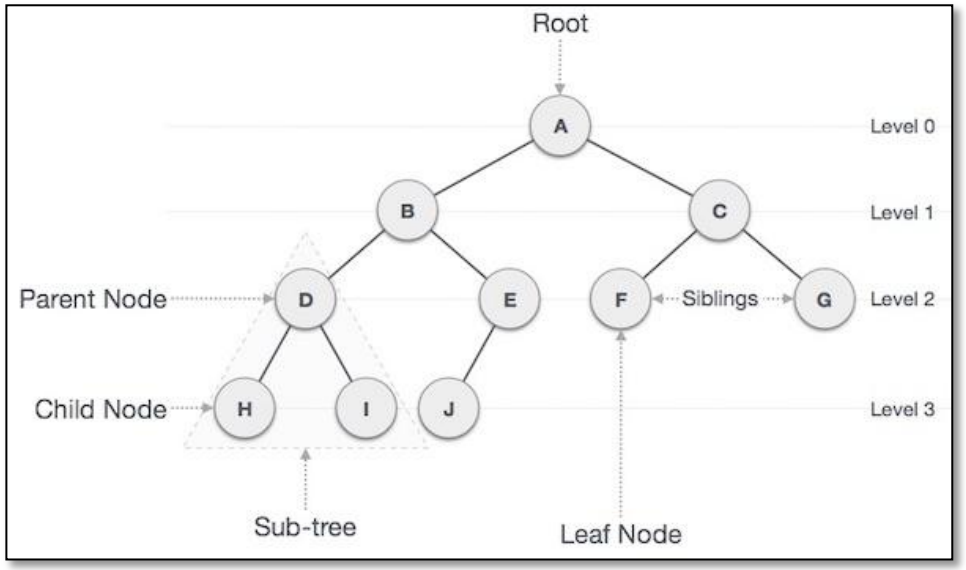
4. Queues

- A queue is a **first-in, first-out (FIFO)** data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the **rear** and removed from the other end called the **front**.



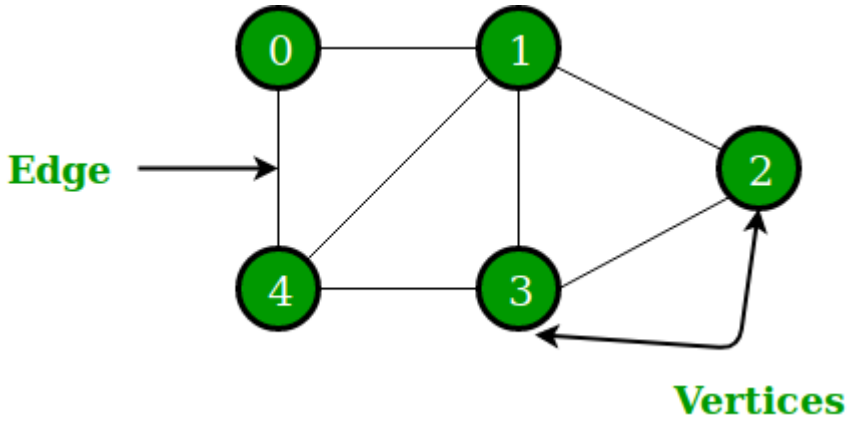
5. Trees

- A tree is a **non-linear data structure** which consists of a collection of nodes arranged in a **hierarchical order**. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.



6. Graphs

- A graph is a **non-linear data structure** which is a collection of **vertices** (also called nodes) and **edges** that connect these vertices.
- For **example**, A node in the graph may represent a city and the edges connecting the nodes can represent roads.
- A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.



OPERATIONS ON DATA STRUCTURES

This section discusses the **different operations** that can be performed on the various data structures previously mentioned.

- **Traversing** It means to access each data item exactly once so that it can be processed. For **example**, to print the names of all the students in a class.
- **Searching** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For **example**, to find the names of all the students who secured 100 marks in mathematics.
- **Inserting** It is used to add new data items to the given list of data items. For **example**, to add the details of a new student who has recently joined the course.
- **Deleting** It means to remove (delete) a particular data item from the given collection of data items. For **example**, to delete the name of a student who has left the course.
- **Sorting** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For **example**, arranging the names of students in a class in an alphabetical order.
- **Merging Lists** of two sorted data items can be combined to form a single list of sorted data items.

ABSTRACT DATA TYPE (ADT)

- An **abstract data type (ADT)** is the way we look at a data structure, focusing on **what it does** and ignoring **how it does** its job.
- For **example**, stacks and queues are perfect examples of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.
- 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**.

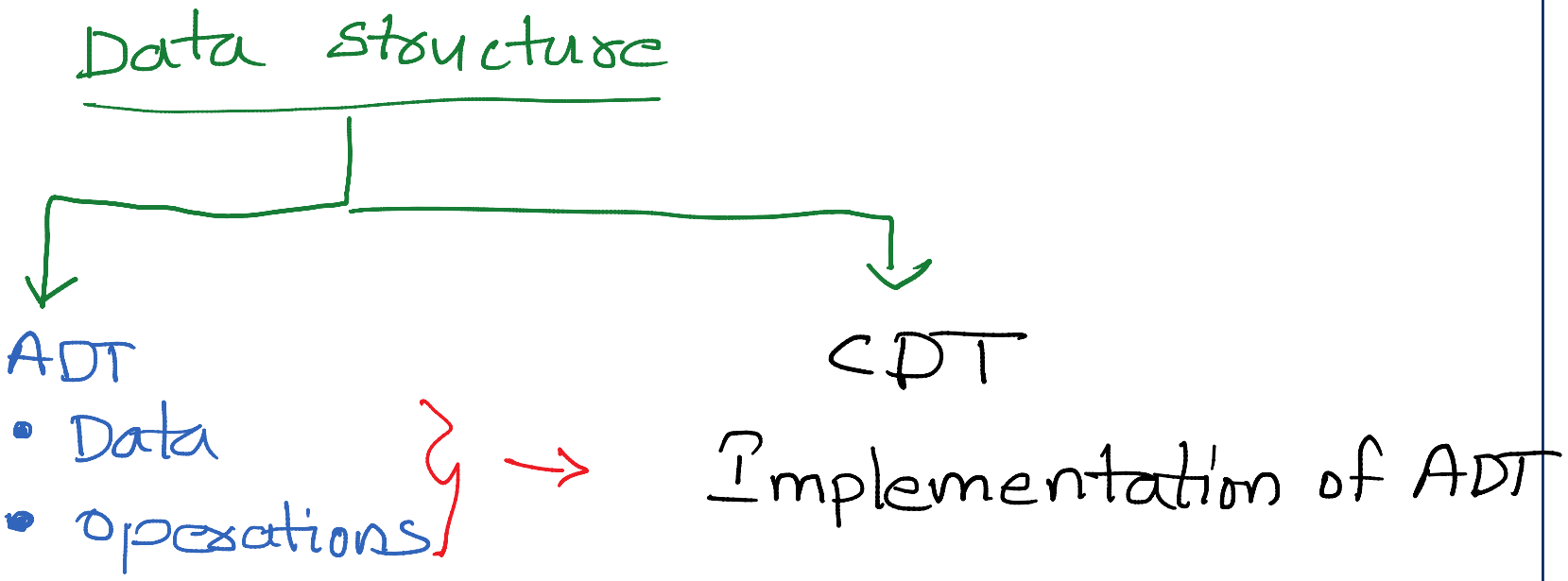
ADT Characteristics:

- In general terms, an ADT interface is a specification of the **values** and **operations** that has two specific properties:

1. It specifies everything you need to know about the data type
2. It makes no reference to the manner or the computer language in which the data type will be implemented.

Concrete Data Types vs Abstract Data Types

- **Abstract data type:** Defines the fundamental operations and properties of data but does not specify an implementation.
- **Concrete data type:** Defines the implementation of Abstract data structures.
- **Example:** Stack is ADT, implementation of stack is done by using CDT, we can use either Array or Linked list.



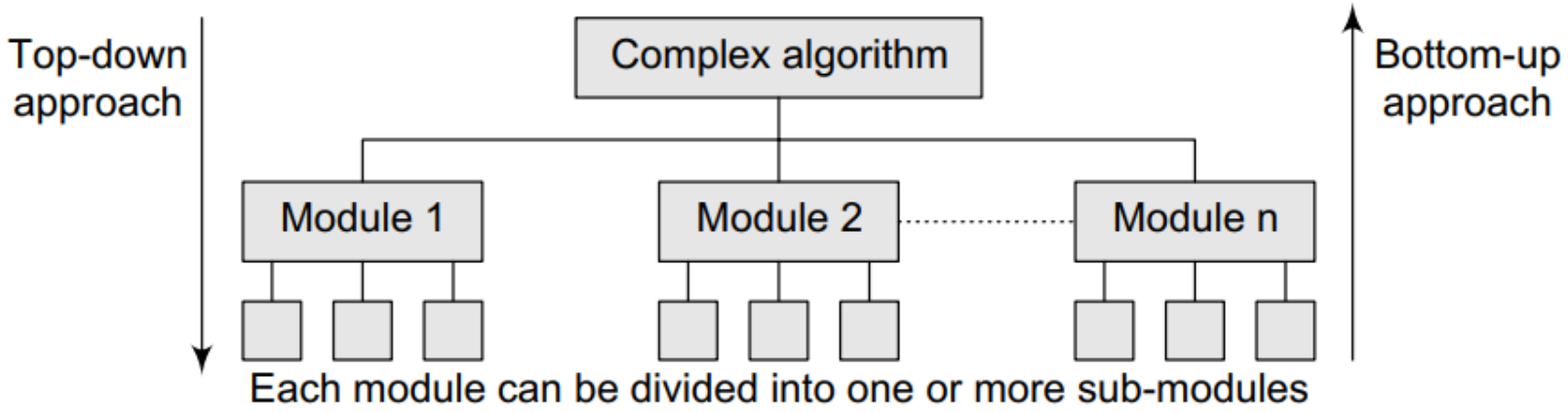
Algorithms

- The typical definition of algorithm is ‘a formally **defined procedure** for performing some calculation’.
- an algorithm provides a **blueprint** to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in **finite number of steps**.
- An algorithm is basically a **set of instructions** that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the **time and space complexity** of the algorithm.

Different Approaches To Designing An Algorithm

- Algorithms are used to **manipulate the data** contained in data structures.
- When working with data structures, algorithms are used to **perform operations** on the stored data.
- A complex algorithm is often divided into smaller units called **modules**. This process of dividing an algorithm into modules is called modularization.
- There are two main approaches to design an algorithm— **top-down approach** and **bottom-up approach.**

Top-down vs bottom-up approach



- **Top-down approach** A top-down design approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved.
- **Bottom-up approach** A bottom-up approach is just the reverse of top-down approach. In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules. The higher level modules are implemented by using the operations performed by lower level modules.
- Whether the top-down strategy should be followed or a bottom-up is a question that can be answered depending on the application at hand.

Difference between Top-down & bottom-up approach

TOP DOWN APPROACH	BOTTOM UP APPROACH
➢ In this approach We focus on breaking up the problem into smaller parts.	➢ In bottom up approach, we solve smaller problems and integrate it as whole and complete the solution.
➢ Mainly used by structured programming language such as COBOL, Fortran, C, etc.	➢ Mainly used by object oriented programming language such as C++, C#, Python.
➢ Each part is programmed separately therefore contain redundancy.	➢ Redundancy is minimized by using data encapsulation and data hiding.
➢ In this the communications is less among modules.	➢ In this module must have communication.
➢ It is used in debugging, module documentation, etc.	➢ It is basically used in testing.
➢ In top down approach, decomposition takes place.	➢ In bottom up approach composition takes place.
➢ In this top function of system might be hard to identify.	➢ In this sometimes we can not build a program from the piece we have started.

Control Structures Used In Algorithms

- An algorithm has **a finite number of steps**. Some steps may involve decision-making and repetition.
- Broadly speaking, an algorithm may employ one of the following control structures: (a) **sequence**, (b) **decision**, and (c) **repetition**.

Sequence

- By sequence, we mean that each step of an algorithm is executed in **a specified order**. Let us write an algorithm to add two numbers.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Decision

- Decision statements are used when the execution of a process depends on the outcome of some condition.
- A decision statement can also be stated in the following manner:

```
IF condition
        Then process1
ELSE process2
```

- an algorithm to check if two numbers are equal.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
              PRINT "EQUAL"
        ELSE
              PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

Repetition

- Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as **`while`**, **`do-while`**, and **`for`** loops. These loops execute one or more steps until some condition is true.
- an algorithm that prints the first 10 natural numbers.

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

- Write an algorithm to find the greatest of three numbers.
- Write an algorithm to check the given number is prime number or not.

Time and Space Complexity

- The efficiency or complexity of an algorithm is stated in terms of **time** and **space** complexity.
- **The time complexity** of an algorithm is basically the running time of a program as a function of the input size. In other words, the number of machine instructions which a program executes is called its **time complexity.**
- **The space complexity** of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

- Generally, the space needed by a program depends on the following two parts:
  - **Fixed part:** It varies from problem to problem. It includes the space needed for storing **instructions, constants, variables, and structured variables** (like arrays and structures).
  - **Variable part:** It varies from program to program. It includes the space needed for **recursion stack**, and for structured variables that are allocated space **dynamically** during the runtime of a program

Worst-case, Average-case, Best-case, and Amortized Time Complexity

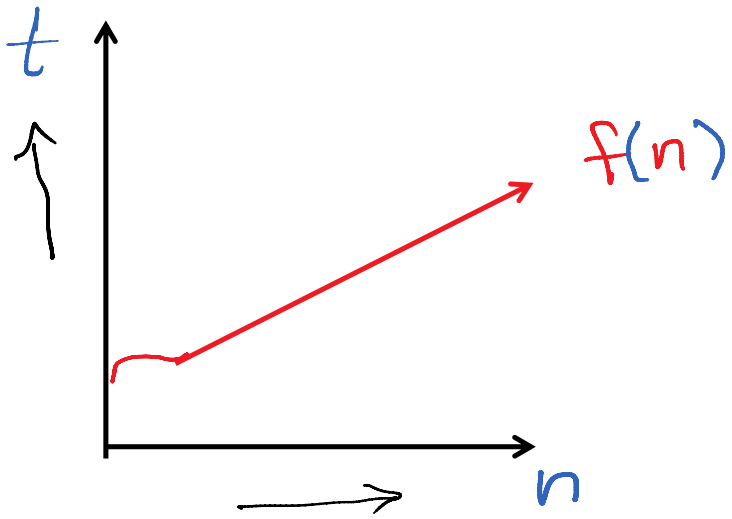
- **Worst-case running time** This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.
- **Average-case running time** The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.
- **Best-case running time** The term 'best-case performance' is used to analyse an algorithm under optimal conditions. while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance.
- **Amortized running time** Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

Time–Space Trade-off

- The **best algorithm** to solve a particular problem at hand is no doubt the one that requires **less memory space** and takes **less time to complete its execution**.
- But practically, There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute.
- Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a **time–space trade-off** among algorithms.
- So, if **space is a big constraint**, then one might choose a program that takes **less space** at the cost of more CPU time.
- On the contrary, if **time is a major constraint**, then one might choose a program that takes **minimum time** to execute at the cost of more space.

Expressing Time and Space Complexity

- The time and space complexity can be expressed using a function **f(n)** where n is the input size for a given instance of the problem being solved.
- Expressing the complexity is required when
  1. We want to predict the rate of growth of complexity as the input size of the problem increases.
  2. There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.
- The most widely used notation to express this function f(n) is the **Big O notation**. It provides the upper bound for the complexity



Algorithm Efficiency

- If a function is **linear** (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- However, if an algorithm contains **loops**, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.
- Let us consider **different cases** in which **loops** determine the efficiency of an algorithm.
- **Linear Loops**
  - To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed For example, consider the loop given below:

```
for(i=0;i<100;i++)
        statement block;
```

  - Here, 100 is the loop factor. **Efficiency is directly proportional to the number of iterations**. Hence, the general formula in the case of linear loops may be given as **f(n) = n**
  - Consider the loop given below:

```
for(i=0;i<100;i+=2)
        statement block;
```

  - Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as **f(n) = n/2**.

Logarithmic Loops

- We have seen that in linear loops, the loop updation statement either **adds** or **subtracts** the loop-controlling variable.
- However, in logarithmic loops, the loop-controlling variable is either **multiplied** or **divided** during each iteration of the loop.
- For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)
        statement block;
```

1 2 4 8 16 32 64 ---- n-1

2^0 2^1 2^2 2^3 2^4 2^5 2^6 ---- 2^k

$2^k = n \Rightarrow k = \log_2 n$

- The loop-controlling variable **i** is multiplied by **2**. The loop will be executed only **10 times** and not **1000 times** because in each iteration the value of i doubles.

```
for(i=1000;i>=1;i/=2)
        statement block;
```

- The loop-controlling variable i is divided by 2. In this case also, the loop will be executed 10 times.
- Thus, the number of iterations is a function of the number by which the loop-controlling variable is **divided** or **multiplied**.
- when **n = 1000**, the number of iterations can be given by **log 1000** which is approximately equal to **10**.
- Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as **f(n) = log n**

$f(n) = \log 2^n$

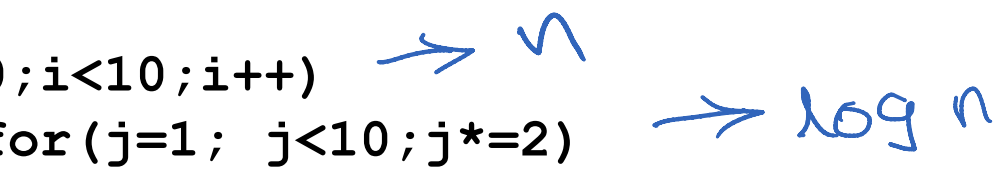
Nested Loops

- Loops that contain loops are known as **nested loops**.
- In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the **product** of the number of iterations in the inner loop and the number of iterations in the outer loop.
- In this case, we analyse the efficiency of the algorithm based on whether it is a **linear logarithmic**, **quadratic**, or **dependent quadratic nested loop**.

Linear logarithmic loop

- Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is **log 10**.
- This inner loop is controlled by an outer loop which iterates **10 times**.
- Therefore, according to the formula, the number of iterations for this code can be given as **10 log 10.**

```
for(i=0;i<10;i++)
        for(j=1; j<10;j*=2)
                statement block;
```



- In more general terms, the efficiency of such loops can be given as **f(n) = n log n.**

Quadratic loop

- In a quadratic loop, the number of iterations in the inner loop is **equal to** the number of iterations in the outer loop.
- Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
    for(j=0; j<10;j++)
        statement block;
```

- The generalized formula for quadratic loop can be given as **f(n)** $= n^2$.

Dependent quadratic loop

- In a dependent quadratic loop, the number of iterations in the inner loop is **dependent** on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
        for(j=0; j<=i;j++)
                statement block;
```

- In this code, the inner loop will execute just **once** in the first iteration, **twice** in the second iteration, **thrice** in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

$$1 + 2 + 3 + \ldots + 9 + 10 = 55$$

- If we calculate the **average** of this loop (55/10 = 5.5),
- we will observe that it is equal to the number of iterations in the outer loop **(10) plus 1 divided by 2**.
- In general terms, the inner loop iterates **(n + 1)/2** times.
- Therefore, the efficiency of such a code can be given as **$f(n) = n\,(n + 1)/2$**

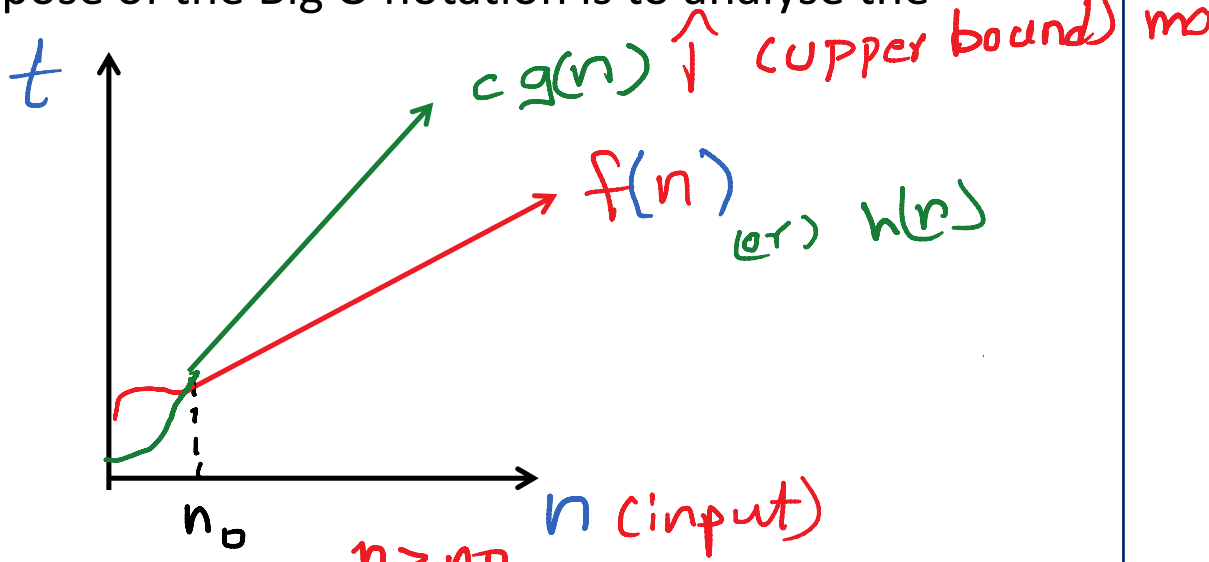
BIG O NOTATION

- The **Big O** notation, where O stands for **'order of'**, is concerned with what happens for very large values of n.
- For example, if a sorting algorithm performs n^2 operations to sort just **n** elements, then that algorithm would be described as an **O(n^2)** algorithm.
- When expressing complexity using the Big O notation, constant multipliers are ignored. So, an **O(4n)** algorithm is equivalent to **O(n)**, which is how it should be written.
- If f(n) and g(n) are the functions defined on a positive integer number n, then **f(n) = O(g(n))**
- Here, n is the problem size and **O(g(n)) = {h(n): ∃ positive constants c, n0 such that 0 ≤ h (n) ≤ cg(n), ∀ n ≥ n0 }.** Hence, we can say that **O(g(n))** comprises a set of all the functions **h(n)** that are less than or equal to **cg(n)** for all values of **n ≥ n0** .

If $f(n) \leq cg(n)$, $c > 0$, $\forall\ n \geq n_0$, then $f(n) = O(g(n))$ and $g(n)$ is an asymptotically tight upper bound for $f(n)$.

- let us look at some examples of g(n) and f(n). Table shows the relationship between g(n) and f(n). Note that the constant values will be ignored because the main purpose of the Big O notation is to analyse the algorithm in a general fashion.

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$



Example. show that $3n+2 = O(n)$

If $f(n) \leq cg(n)$, $c > 0$, $\forall\, n \geq n_0$, then $f(n) = O(g(n))$ and $g(n)$ is an asymptotically tight upper bound for $f(n)$.

$f(n) \leq c\, g(n)$

$3n+2 \leq cn$

substitute $c-4$

$3n+2 \leq 4n$

$\Rightarrow n \geq 2$

$\therefore 3n+2 = O(n)$.

Example 2.2 Show that $400n^3 + 20n^2 = O(n^3)$.

Solution By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting $400n^3 + 20n^2$ as $h(n)$ and n^3 as $g(n)$, we get

$$0 \le 400n^3 + 20n^2 \le cn^3$$

Dividing by n^3

$$0/n^3 \le 400n^3/n^3 + 20n^2/n^3 \le cn^3/n^3$$
$$0 \le 400 + 20/n \le c$$

Note that $20/n \rightarrow 0$ as $n \rightarrow \infty$, and $20/n$ is maximum when $n = 1$. Therefore,

$$0 \le 400 + 20/1 \le c$$

This means, $c = 420$

To determine the value of n_0,

$$0 \le 400 + 20/n_0 \le 420$$
$$-400 \le 400 + 20/n_0 - 400 \le 420 - 400$$
$$-400 \le 20/n_0 \le 20$$
$$-20 \le 1/n_0 \le 1$$

$-20\ n_0 \le 1 \le n_0$. This implies $n_0 = 1$.

Hence, $0 \le 400n^3 + 20n^2 \le 420n^3\ \forall\ n \ge n_0 = 1$.

$400n^3 + 20n^2 \Rightarrow O(n^3)$

O(g(n)) = {h(n): ∃ positive constants c, n0 such that 0 ≤ h (n) ≤ cg(n), ∀ n ≥ n0 }.

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as $O(1)$
- Linear time algorithm: running time complexity given as $O(n)$
- Logarithmic time algorithm: running time complexity given as $O(\log n)$
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Table 2.2 shows the number of operations that would be performd for various values of n.

Table 2.2 Number of operations for different functions of n

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

Limitations of Big O Notation

Big Oh Notation has following two basic limitations :

1. It contains no effort to improve the programming methodology. Big Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyze and calculate the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constants. For example, one algorithm is taking $1000n^2$ time to execute and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

OMEGA NOTATION (Ω)

- The Omega notation provides a tight **lower bound for f(n).**
- Ω notation is simply written as, **f(n) ∈ Ω(g(n)),** where n is the problem size and **Ω(g(n)) = {h(n): ∃ positive constants c > 0, n0 such that 0 ≤ cg(n) ≤ h(n), ∀ n ≥ n0 }.**
- Hence, we can say that **Ω(g(n))** comprises a set of all the functions **h(n)** that are greater than or equal to **cg(n)** for all values of **n ≥ n0** .

> If $cg(n) \le f(n)$, $c > 0$, $\forall n \ge n_0$, then $f(n) \in \Omega(g(n))$ and $g(n)$ is an asymptotically tight lower bound for $f(n)$.

Example: show that $3n+2 = \Omega(n)$

$f(n) \ge c\,g(n)$

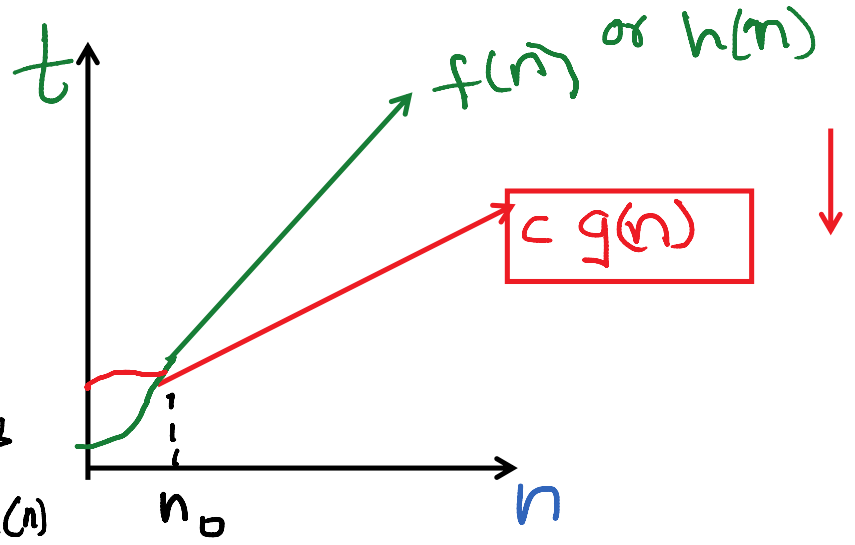
$3n+2 \ge cn$

substitute $c = 1$ then condition satisfies

$3n+2 \ge n$

for all values of n condition satisfies

$n \ge 1$ $\therefore 3n+2 = \Omega(n)$



THETA NOTATION (θ)

- Theta notation provides an asymptotically tight bound for **f(n)**.
- Θ notation is simply written as, **f(n) ∈ Θ(g(n)),** where n is the problem size and **Θ(g(n))** = {**h(n): ∃** positive constants **c1 , c2** , and **n0** such that **0 ≤ c1 g(n) ≤ h(n) ≤ c2 g(n), ∀ n ≥ n0** }.
- Hence, we can say that **Θ(g(n))** comprises a set of all the functions h(n) that are between **c1 g(n)** and **c2 g(n)** for all values of **n ≥ n0** .

If $f(n)$ is between $c_1 g(n)$ and $c_2 g(n)$, $\forall n \geq n_0$, then $f(n) \in \Theta(g(n))$ and $g(n)$ is an asymptotically tight bound for $f(n)$ and $f(n)$ is amongst $h(n)$ in the set.

$\Rightarrow f(n) = \Theta(g(n)),\ c_1 g(n) \leq f(n) \leq c_2 g(n),$

$c_1 \cdot c_2 > 0,\ n \geq n_0$

Example $f(n) = 3n+2$ $g(n) = n$

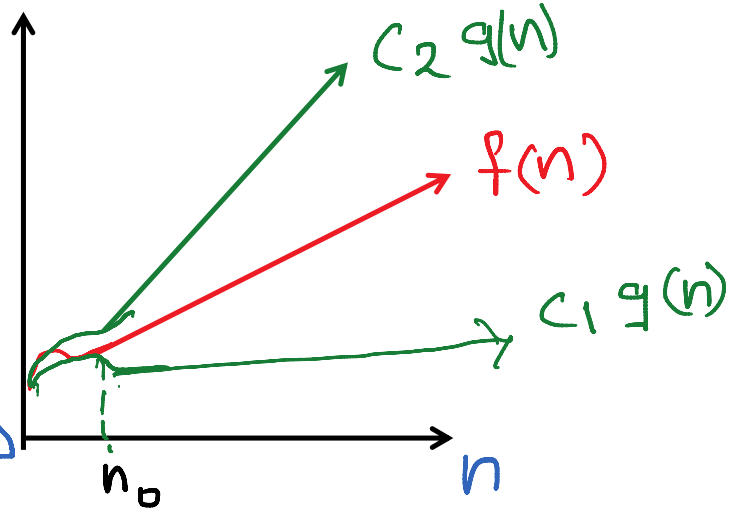
O → $f(n) \leq c_2 g(n)$

$3n+2 \leq c_2 n \Rightarrow 3n+2 \leq 4n \Rightarrow n \geq 2$

Ω → $f(n) \geq c_1 g(n)$

$3n+2 \geq c\,n \Rightarrow 3n+2 \geq n \Rightarrow n \geq 1$ ($c=1$)

$\therefore 3n+2 = \Theta(n)$



SEARCHING

- **Searching** means to find whether a particular value is present in an array or not.
- If the value is **present** in the array, then searching is said to be **successful** and the searching process gives the location of that value in the array.
- However, if the value is **not present** in the array, the searching process displays an appropriate message and in this case searching is said to be **unsuccessful**.
- Three popular searching methods are used.
    1. Linear search
    2. Binary search
    3. Fibonacci search

Linear search

- Linear search, also called as **sequential search**, is a very simple method used for searching an array for a particular value.
- It works by **comparing** the value to be searched with every element of the array one by one in a sequence until a match is found.
- Linear search is mostly used to search an **unordered list of elements** (array in which data elements are not sorted).
- **For example**, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

- the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm for linear search

```
LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:              IF A[I] = VAL
                             SET POS = I
                             PRINT POS
                             Go to Step 6
                     [END OF IF]
                      SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Linear Search Algorithm.

worst case → $O(n)$

best case → $\Omega(1)$

average case → $\Theta(n/2) \Rightarrow \Theta$

Exercise -1 (Searching)

a)Write C program that use both recursive and non recursive functions to perform Linear search for a Key value in a given list.