DIPLOMA WALLAH

(Your **One Stop Hub** For Diploma Resources)

DATA STRUCTURES

WITH PYTHON

Complete Notes Based on Full Syllabus

- Diploma Engineering

4th Semester

www.diplomawallah.in

Notes prepared by **Sangam**

Diploma Wallah
©

Unit-01

Data Structure:—

A data structure is a way to store organize, and manage data in a computer so that it can be accessed and modified efficiently.
An efficient data structure also use minimum memory space and execution time to process the structure.

Why data structure important?
Imagine a library with thousand of books. if they are scattered, we will never find what we need, but if they are arranged by category, author, and title, we can find any book easily.

- Make searching, insertion, deletion, and updating of data faster.
- Help in writing optimized algorithms
- Are crucial in system design, compilers, database, OS, AI etc.
- Save memory space.
- Data representation is easy.
- It require less time.

Sangam

common operation in Data Structure

1. Insertion — Add an element
2. Deletion — Remove an element
3. Traversal — visit every element eg (printing)
4. Searching — find a specific element
5. Sorting — Arrange elements in a specific order

* Characteristics are:—

1. Linear :- Elements arranged in a sequence (one after another).
   Ex- Array, Stack, Queue, Linked list.

2. Non-Linear :- Elements are connected in hierarchial or graph based format.
   Ex- Tree, graph

3. Static :- Memory allocated at compile time.
   Example :- array

4. Dynamic Data Structure :—
   Size can change during runtime, memory allocated dynamically.
   Ex- Linked list, Stack (with Linked list), Tree.

5. Efficiency :- A good data structure uses memory efficiently and minizes wastage of space.
Ex - Linked list doesn't waste memory like arrays do (no need for fixed size).

6. Time Complexity :- Operation (insertion, deletion, search) should be fast and efficient.
Ex- Hash Table provides average O search time.

7. Data Access method :- Defined how data is accessed.

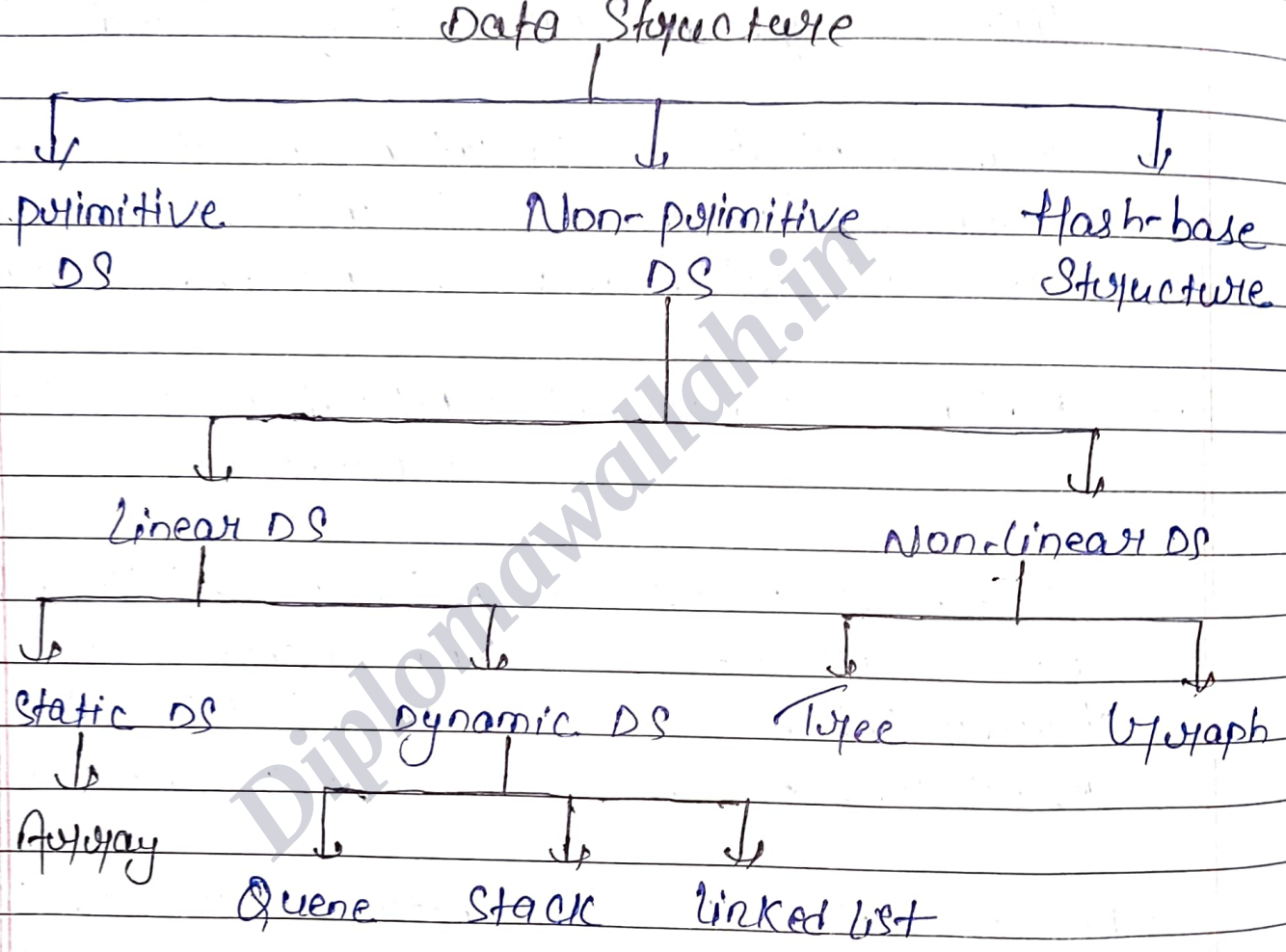
Sequential Access — one by one → Linked list
Direct access → Any element → Array
Hierarchial Acess — Root to child → Tree
Associative Access — Key - Value. - Hash map.

8. Reusability :- A good data structure can be reused in different program/application.
ex - Stack used in expression evaluation undo operations etc.

9. Modularity :- Can be broken into smaller modules or functions, making code easier to maintain and test.
Ex - Queue implement using Equene() and Dequene() functions.

Diploma Wallah

Classification of Data Structure.



Sangam

Characteristics: Advantages:- simple, fixed size, efficient, Directly supported.

* primitive DS:- They are the basic building blocks of data representation in programming, datatypes are (int, float, char, boolean, byte, short long, double) Directly store value in memory, defined by the programming language itself.
* Non-primitive Data Structure:-
  A non-primitive DS is a type of Data Structure that is built using primitive data types and can store multiple values (unlike primitive which single values). It can be linear and non-linear in arrangement.

Eg.

* Built in non-primitive DS:- List, tuple, set, dictonary.
* user-defined:- Stack, Queue, linked list, Tree, Graph, Hashtable.

Types

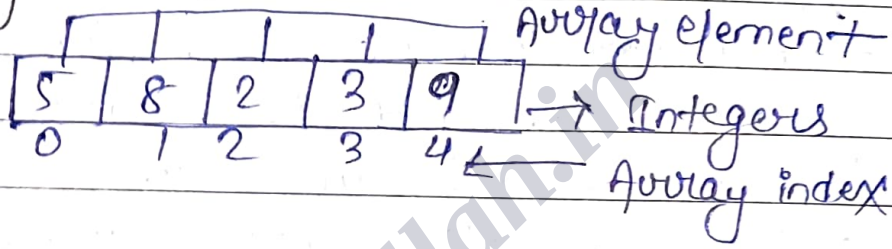
1. Linear Data Structure:-
   Data structure where data element are arranged sequentially or linearly where each and every elements is attached to its previous and next adjacent is called linear data structure.

Ex- [1,2,3], (4,5,6) {"name": "Sagar"}

Dataelement >> Sequence >> Store

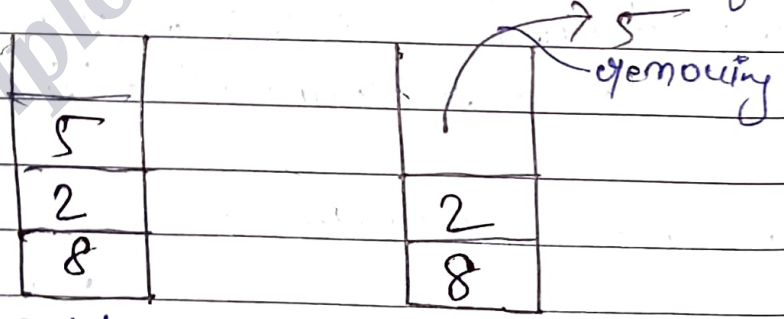
Example of linear data structure.

(1) Array



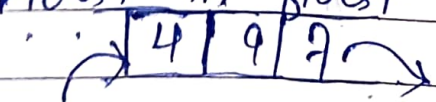
(11) Stack (LIFO)

Last in first out means the element which added in last will remove first.



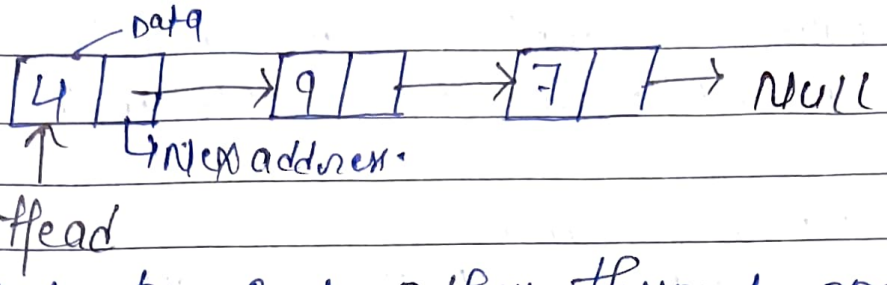
(111) Queue (FIFO) add

First in first out



(Ticket counter) first go first get.

(IV) Linked list



Connected to each other through node and each node contain data items and address of next node.

Types:-

(1) List → dynamic Array

A list is a built-in linear data structure in python that stores ordered and changeable (mutable) elements. You can store different data types in one list.

Operation:-

- Access: using index (eg. list[0])
- Insert: append(), insert(), extend().
- Delete: remove(), pop(), del

Examples:-

```
fruits = ['apple', "banana", "cherry"]
print(fruits[0])   op// apple
```

// Adding element

```
fruits.append("orange")
print(fruits)  # ['apple', 'banana', 'cherry',
                                    'orange']
```

// Removing element

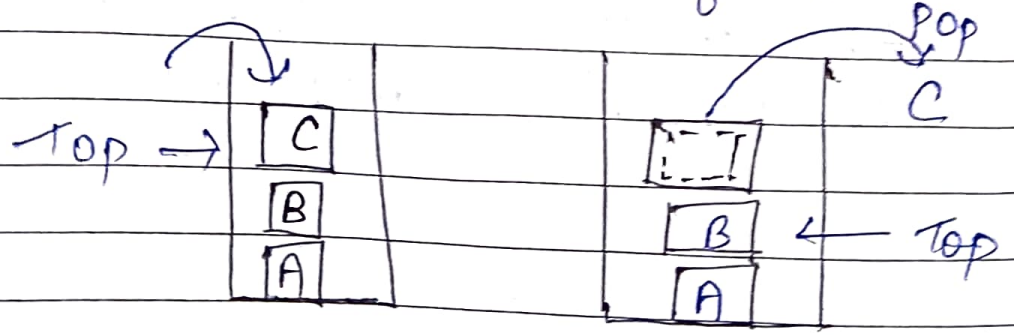
```
fruits.remove("banana")
print(fruits)  # ['apple', 'cherry', 'orang
```

2. Stack

A Stack is a linear data structure that follows a particular order in which the operations are performed.

The order is LIFO (Last In First Out)

* LIFO implies that the element that is inserted last, comes out first.



Types of Stack

① Fixed Size Stack:

It has fixed size and cannot grow or shrink dynamically.
If the stack is full and an attempt is made to add an element to it, an overflow error occurs.

If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

ex -

```
Class Stack
    def __init__(self, size):
        self.stack = []
        self.size = size
    def push(self, val):
        if len(self.stack) < self.size:
            self.stack.append(val)
        else:
            print("overflow")
    def pop(self):
        if self.stack:
            return self.stack.pop()
        else:
            print("underflow")
```

```
S = stack(3)
S.push(10)
S.push(20)
S.push(30)
S.push(40)  # overflow
print(S.pop())  # 30
print(S.pop())  # 20
print(S.pop())  # 10
print(S.pop())  # underflow
```

* Dynamic Size Stack:-

A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accomodate the new element, and the stack is empty, it decrease its size.

This types of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Example:-

```
Class DynamicStack:
    def __init__(self):
        self.stack = []
    def push(self, val):          → No size limit
        self.stack.append(val)
    def pop(self):
        if self.stack:
            return self.stack.pop()
        else:
            print("underflow")
    def display(self):
        print(self.stack)

s = DynamicStack()
s.push(10)
s.push(20)
s.push(30)
s.display()  # [10,20,30]
print(s.pop())  # 30
print(s.pop())  # 20
print(s.pop())  # 10
print(s.pop())  # underflow
```

Queue :-

A queue is a linear structure where the first element added is the first to be removed like people standing in a line.

Main operations :

- Enqueue (add to end) : append()
- Dequeue (remove from start) : popleft() (from collections.deque)

Example :-

```
from collections import deque
queue = deque()

# Enqueue
queue.append("A")
queue.append("B")
queue.append("C")
print(queue)  # deque(['A', 'B', 'C'])

# Dequeue
print(queue.popleft())  # A
print(queue)  # deque(['B', 'C'])
```

Deque - Double Ended Queue.

A deque allows insertion and deletion from both ends - front and rear.

It more flexible than a queue or stack.

Main operation:-

- append(), appendleft()
- pop(), popleft()

Example

```
from collections import deque

dq = deque([10,20,30])
dq.append(40)  # add to right
dq.appendleft(5)  # add to left

print(dq)  # deque([5,10,20,30,40])

dq.pop()  # remove from right
dq.popleft()  # remove from left

print(dq)  #deque([10,20,30])
```

* Array

A array is similar to a list, but stores only same type of elements and is slightly more memory-efficient.
you use it from the array module.

main operation:—
append(), pop(), index access like list

Example: —

```
import array
# create an array of integers.

arr = array.array('i', [1,2,3])
arr.append(4)
print(arr[0]) #1
print(arr) # array('i', [1,2,3,4])
```

Operation on Stack

① push :- Method to push element in the stack.

Algorithm →

- Check if stack is full using isFull() method.
- If Stack is full then print an error message ("Stack overflow") and return.
- Increament the 'top' to move to next empty position in the stack.
- Insert the new element value into Stack Array.
- print the message indicating the element has been pushed on to the stack.

Time Complexity : O(1)
Space Complexity : O(1)

Examples

Sangam

Example:-

```
class mystack
{
  int maxsize = 100;
  int [] stack = new int [maxsize];
  int top = -1;
// push opeyation

  void push (int value)
  {
    if (top == maxsize -1)
    {
      Sopln ("Stack overflow");
    } else {
      stack [++top] = value;
      Sopln (value + "pushed to
                  Stack");
// display Stack
  void display ()
  {
    for (int i = top ; i >=0; i--)
    {
      Sopln (stack (i) + " ");
    }
    Sopln (" ");
  }
```

```
public static main$
    psvm
    mystack = new mystack ();
    S.push (10);
    S.push (20);
    S.push (30);
    Sopln ("Stack");
    S.display ();
  }
}
```

o/p → 10 pushed to stack
20 pushed to stack
30 pushed to Stack
Stack = 30, 20, 10

(11) · pop operation :— method to pop element from the stack.

Algorithm :—

- Check whether if the Stack is empty using is empty () method.
- If stack is empty, print the error message ("Stack underflow") and return -1.
- Retrieve the top element from the Stack Array
- Decrement the top to remove the top element from stack.
- Return the popped element in stack.

Time Complexity : O(1) Space complexity : O(1)

(top)
3. peek operation:-
Return the top element from the Stack.

Algorithm:-

- check if Stack is empty using is empty() method.
- If Stack is empty, print the error message ("Stack is empty") and return -1
- Return the top element from Stack Array without removing it.

Time and space complexity :- O(1)

4. IsEmpty:- It is used to check whether the Stack is empty or not.

Algorithm:-
- It return true if Stack is empty that indicates top variables is -1 that means no elements in stack, otherwise false.

Time and Space Complexity :- O(1).

5. IsFull :- It check whether the stack is full or not.

Algorithm:-

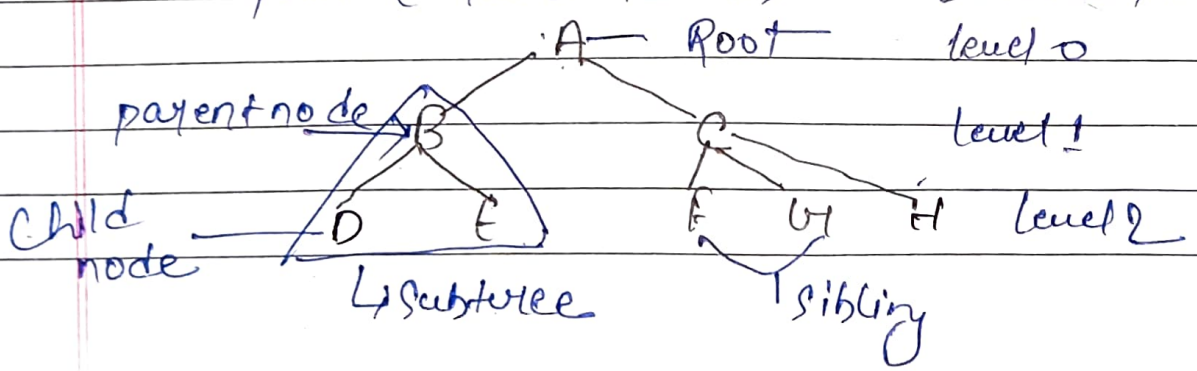
- It can return true if stack is full i.e if top variable has reach the maximum capacity of stack (maxsize-1) other-wise it return false.

Time complexity and Space complexity O(1).

2. * Non-Linear Data Structure:- Data is arranged in hierarchical or non-sequential form.
   - Tree:- Hierarchical Structure (eg- Binary Tree, BST).
   - Graph - Nodes connected by edges (Can be directed or undirected).

Ex- # Simple tree representation using dictonary

```
tree = {
    "root": ["child1", "child2"]
}
print(tree["root"])  # ['child1', 'child2']
```



Abstractions

Abstraction in data types is the process of representing essential features of a data type while hiding the internal implementation details.

It allows the user to interact with data through a defined interface without needing to understand the underlying structure or storage mechanism.

In simple words Abstraction means using data types by knowing what they do, not how they work internally.

* ~~Abstraction~~
Abstract Data Types

An abstract data Type (ADT) is a conceptual model that defines a set of operations and behaviours for a data structure, without specifying how these operations are implemented or how data is organized in memory.

The defination of ADT only mention what operations are to be performed but not how these operations will be implemented.

It does not specifies how data will organized in memory and what algorithms will be used for implementing the operations.

We use ADTs because:—
- Encapsulation
- Reusability
- Modularity
- Security

Examples:—

Student ADT

* Attributes
  - name
  - roll
  - marks
* Behaviours / operation
  - display() — Show Student details.
  - get_grade() - Return grade based on marks.

Example code:-

```
class Student:
    def __init__(self, name, roll, marks
        self.name = name  # private attribute
        self.__roll = roll  "
        self.__marks = marks  "
    def display(self):
        print(f"Name: {self.__name},
              Roll: {self.__roll}, Marks:
              {self.__marks}")
    def get_grade(self):
        if self.__marks >= 90:
            return 'A'
        else:
            return 'C'
```

```
s1 = Student("ABC", 101, 92)
s1.display()
Print("Grade:", s1.get_grade())
```

O/P → Name: ABC, Roll: 101, Marks: 92
Grade: A

Date

The internal details (how grade is calculated, how data stored) are hidden.
- The user interact with a clear interfaces not the implementation.

Date ADTs

Attributes
day: day of the month
month: Month of the year
year: year component

Behaviour:-
- display(): print date in DD-MM-YYYY
- is_leap_year(): check if the year is a leap year

Code

```
class Date:
    def __init__(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year
    def display(self):
        print(f"{self.__day:02d}-{self.__month:02d}-{self.__year}")
```

d2 = Date(15, 8, 2024)
d1.display()
print("Leap year:", d1.is_leap_year())

o/p:-
15.08-2024
Leap year: True

```
def is_leap_year(self):
    y = self.year
    return (y % 4 == 0 and y % 100 != 0
    or (y % 400 == 0)
```

* The user can use display() and is_leap_year() without knowing the underlying logic.
* Abstraction and data hiding make it a perfect ADT.
* It hides how the date is stored & calculated

Employee ADT

Attributes:
emp_id
name
Salary

Behaviours
- display()
- annual_Salary().

Code

```
Class Employee
    def __init__(self, emp_id, name, salary):
        self.__emp_id = emp_id
        self.__name = name
        self.__salary = salary
    def display(self):
        print(f"ID: {self.__emp_id}, Name: {self.__name}, Salary:
            {self.__salary}")
    def annual_salary(self):
        return self.__salary * 12
```

* The internal data (ID, name, salary) and the logic of salary calculation are hidden.

```
emp1 = Employee(101, 'Alice', 30000)
emp1.display()
print("Annual Salary:",
        emp1.annual_salary());
```

O/P → ID: 101, Name: Alice, Salary: 30000
Annual Salary: 360000

using ADT

using an ADT means working with a data structure by accessing only its defined operations, without needing to understand or interact with its internal implementation.

This make the programming more modular, secure and easier to manage.

Steps to ~~create~~ use ADT

1. Create an instance of the ADT (eg. Object in python)
2. Call defined operations/methods (eg - push(), pop(), display()).
3. Never access internal data directly.

purpose of using ADT

* Abstraction
* modularity
* Code Reusability
* Secure and data protection

Example:—

```
S1 = Student ("Sagar", 31, 101)  # object
S1.display()  # using Display()

grade = S1.get_grade()
print ("Grade:", grade)
```

- We are Calling public methods (display(), get_grade()).
- We are not dealing with how name or marks stored.

Implementing the ADT

Implementing an ADT means defining the internal structure of the data and writing the code for the operations that the ADTs.

It involves:

1. Creating the data structure
2. Writing the functions/operations
3. Hiding the internal details. (abstraction)
4. Exposing only the necessary interface.

Steps:-

1. Define the data (attributes) of the ADT
2. Write the function (operation)
3. Encapsulate using classes / structs.
4. Hide data using private access modifiers.
5. Expose public method of interaction.

eg. Student ADT

```
Class Student:
    def __init__(self, name, roll_no, course):
        self.name = name
        self.roll_no = rollno
        self.course = course
        self.enrolled = True

    def get_details(self):
        return (f"Name: {self.name}, Rollno:
                {self.rollno}, course.{self.course}")

    def update_course(self, new_course):
        self.course = new_course

    def is_enrolled(self):
        return self.enrolled
```

unit-01 completed

Sanjana