Data Structure Using Python

INTRODUCTION

What is Data Structure?

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms.

DATA STRUCTURE OPERATIONS

Data are processed by means of certain operations which appearing in the data structure. Following are the most common types of Data structure operations.

1. TRAVERSING
2. SEARCHING
3. INSERTING
4. DELETING
5. SORTING
6. MERGING

DATA STRUCTURE OPERATIONS

1. TRAVERSING

Accessing each records exactly once so that certain items in the record may be processed.

2. SEARCHING

Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.

DATA STRUCTURE OPERATIONS

3. INSERTING

Adding a new record to the structure.

4. DELETING

Removing the record from the structure.

DATA STRUCTURE OPERATIONS

5. SORTING

Managing the data or record in some logical order(Ascending or descending order).

6. MERGING

Combining the record in two different sorted files into a single sorted file.

DATA TYPE VS DATA STRUCTURES

DATA TYPE VS DATA STRUCTURES

DATA TYPE refers to the type i.e integer or float or character or string for the data which is represented in the memory location of the computer system.

DATA STRUCTURE refers to the physical implementation or arrangement of data in the memory. In other words it defines a way of storing, accessing, manipulating data stored in the data structure.

DIFFERENT DATA STRUCTURES

Data structures are broadly classified in to two types.

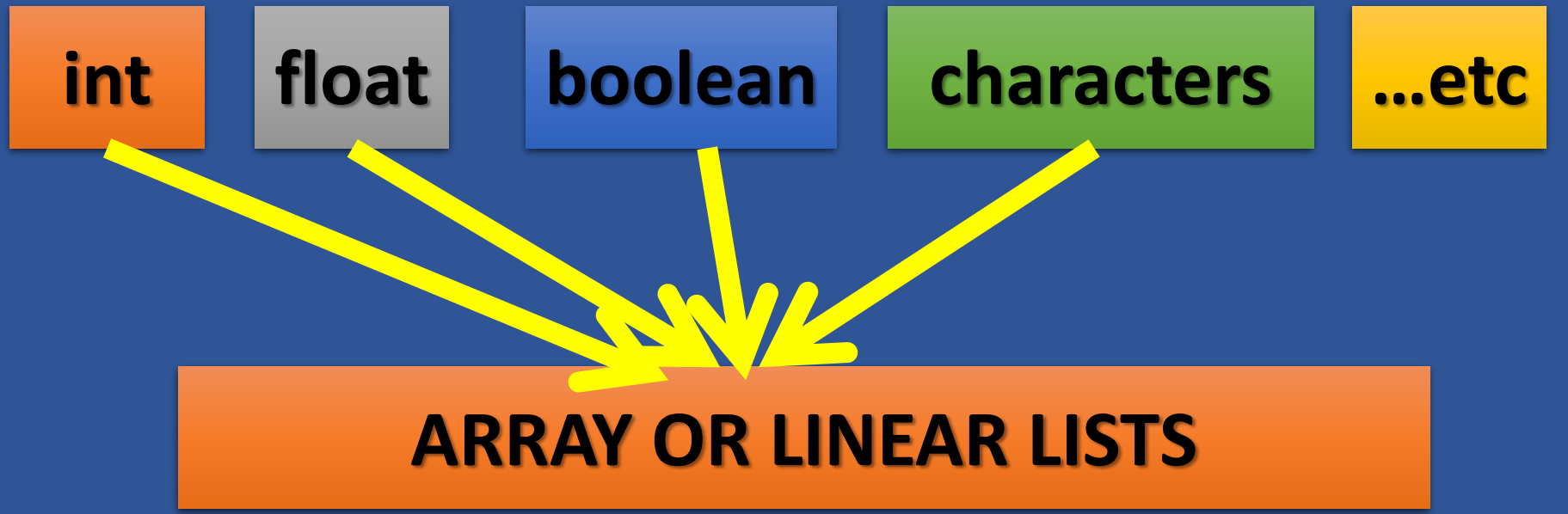
1. SIMPLE DATA STRUCTURE

2. COMPOUND DATA STRUCTURE

1. SIMPLE DATA STRUCTURE

1. SIMPLE DATA STRUCTURE

Simple Data structures are also called as primitive data structures and are built from basic data types viz int, float, boolean, characters are called simple data structures.



ARRAY or LINEAR LISTS

A linear array is a list of finite numbers of elements stored in the memory. In a linear array, we can store only homogeneous data elements. Elements of the array form a sequence or linear list that can have the same type of data. Each element of the array is referred by an index set.

Note: Arrays can be implemented through LIST or through NumPy arrays.

2. COMPOUND DATA STRUCTURE

2. COMPOUND DATA STRUCTURE

Compound Data Structure are complex in nature and are classified in to two types

1. LINEAR DATA STRUCTURES

2. NON LINEAR DATA STRUCTURES

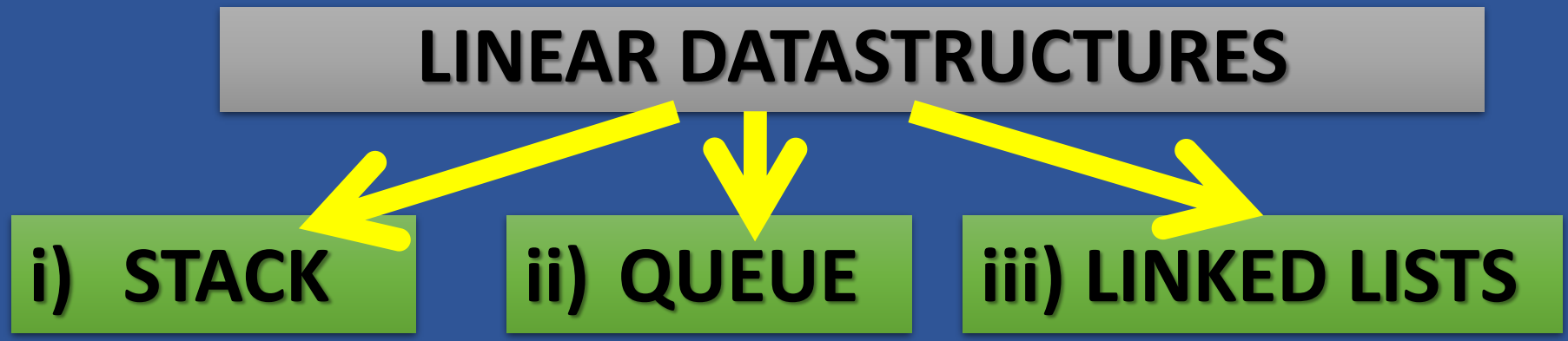
1. LINEAR DATASTRUCTURES

1. LINEAR DATASTRUCTURES

What is Linear Data Structure?

A data structure is said to be linear data structure if its elements form a sequence.

meaning sequential arrangement of data is the linear data structure. For Example

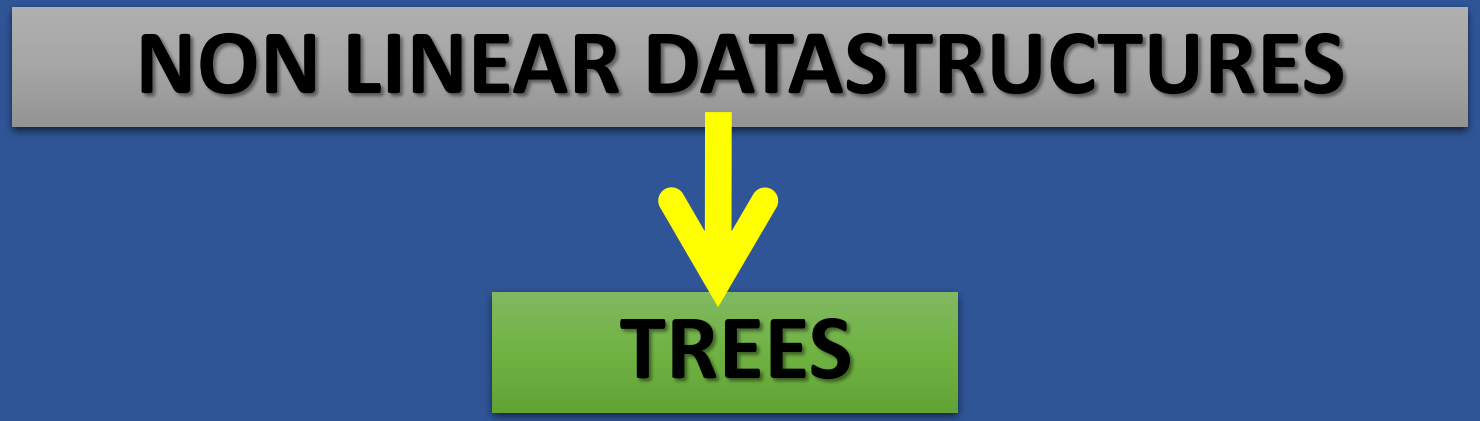


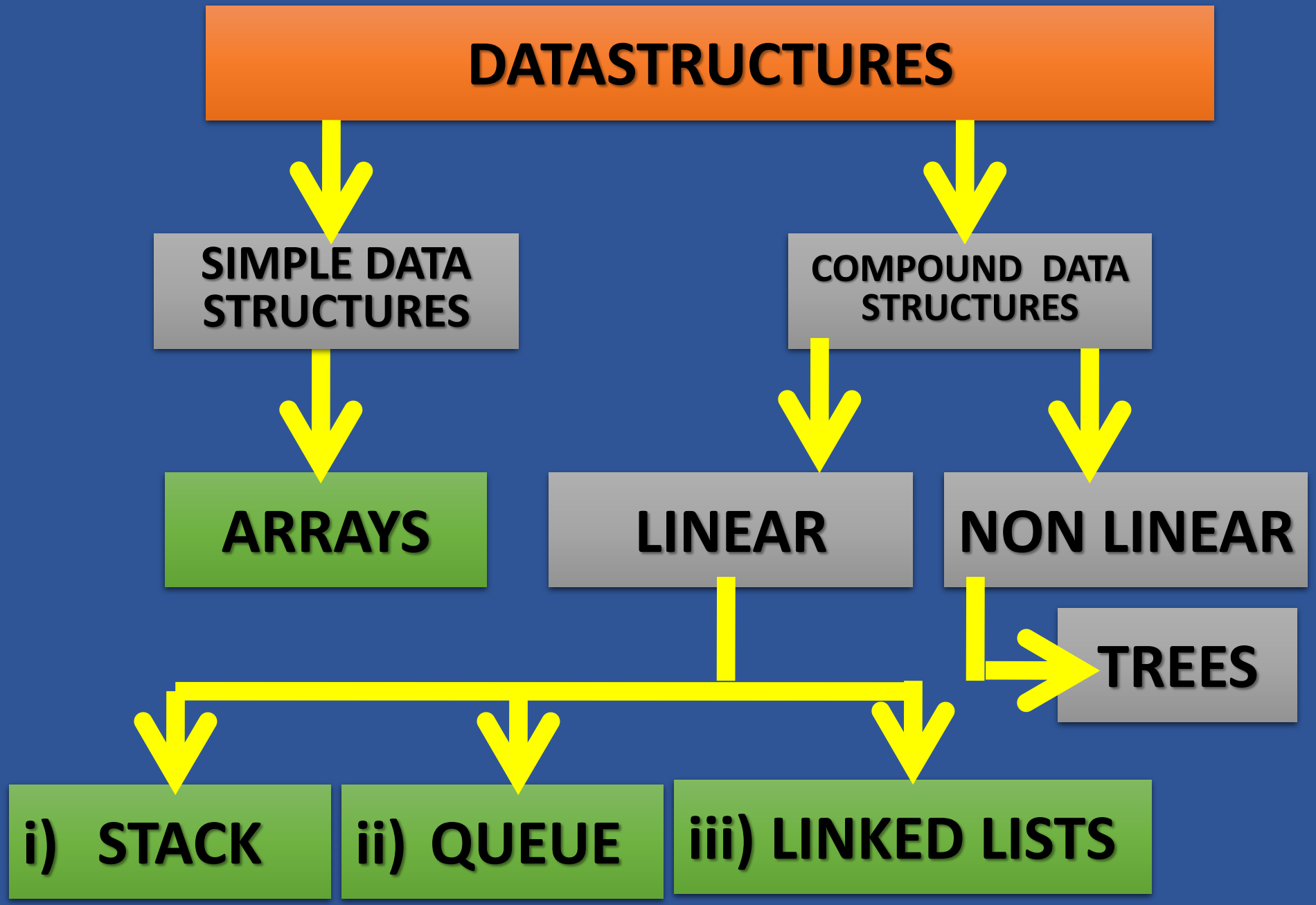
NON LINEAR DATASTRUCTURES

2. NON LINEAR DATASTRUCTURES

What is Non Linear Data Structure?

A non-linear data structure is a data structure in which a data item is connected to several other data items. So that a given data item has the possibility to reach one-or-more data items.




DATASTRUCTURES
SIMPLE DATA STRUCTURES
COMPOUND DATA STRUCTURES
ARRAYS
LINEAR
NON LINEAR
TREES
i) STACK
ii) QUEUE
iii) LINKED LISTS

STACKS

STACKS

What is STACK?

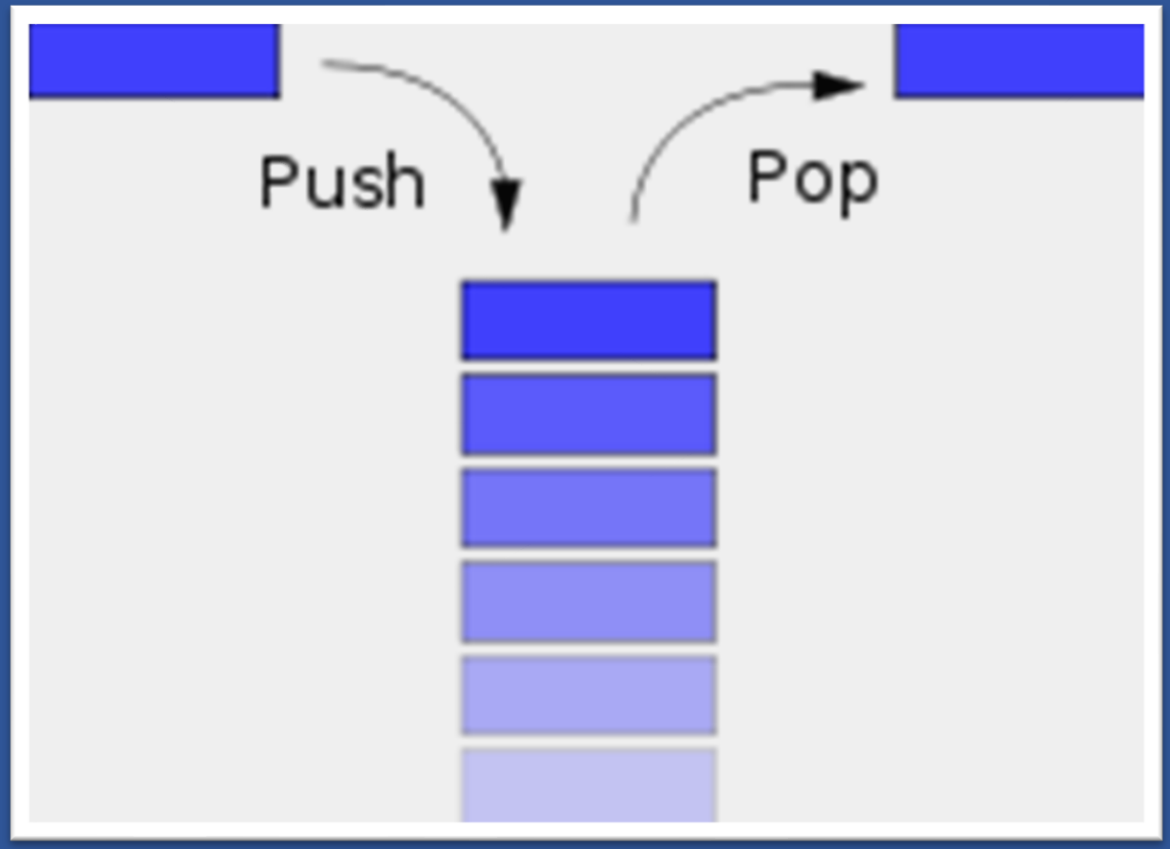
In computer science, a stack is a last in, first out (LIFO) data structure. A stack is characterized by only two fundamental operations and they are:

1. PUSH
2. POP

The PUSH operation adds an item to the top of the stack.

The POP operation removes an item from the top of the stack.

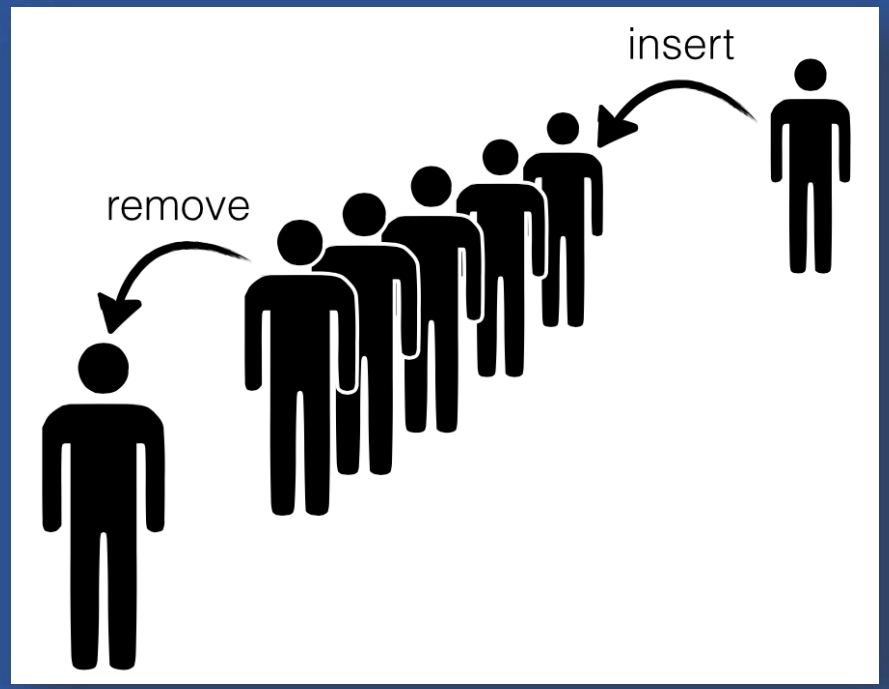
STACKS



QUEUE

QUEUE

What is Queue?



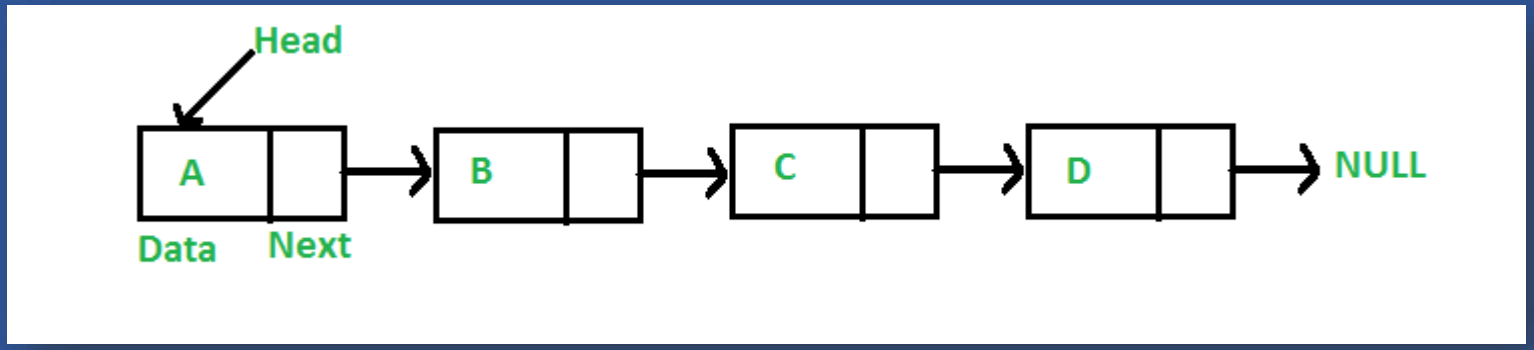
Queue is a linear data structure which follows First In First Out (FIFO) rule in which a new item is added at the rear end and deletion of item is from the front end of the queue. In a FIFO data structure, the first element added to the queue will be the first one to be removed.

LINKED LISTS

LINKED LISTS

What is 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:



SEARCHING

SEARCHING

What is Searching?

Searching is an operation or a technique that helps finds the place of a given element or value in the list.

There are various algorithms , most common searching algorithms are:-

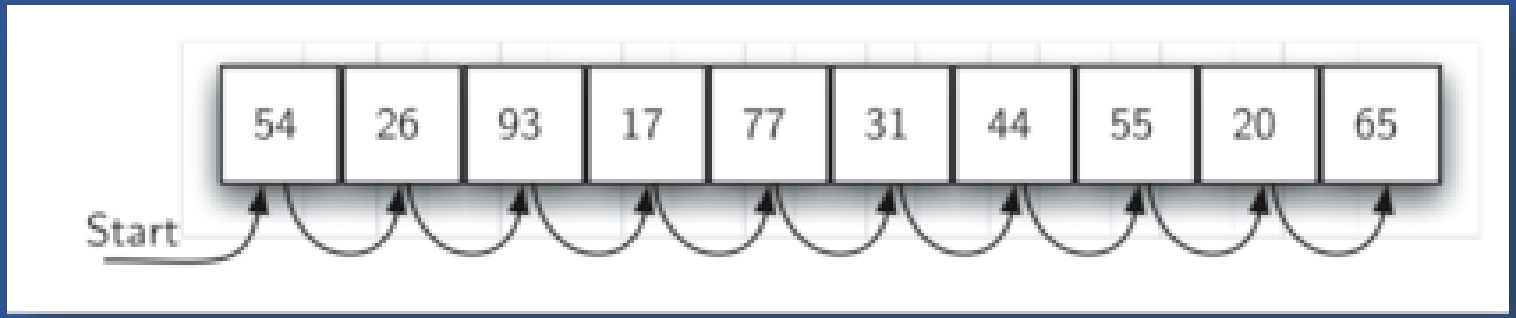
1. LINEAR SEARCH OR SEQUENTIAL SEARCH
2. BINARY SEARCH

1. LINEAR SEARCH OR SEQUENTIAL SEARCH

1. LINEAR SEARCH OR SEQUENTIAL SEARCH

What is Linear Search or Sequential Search?

In computer science, a linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.



1. LINEAR SEARCH OR SEQUENTIAL SEARCH

WAPP to search an element in a given list using Linear Search Method using NumPy

```
import numpy as np
def Linear_Search():
    L1=[9,32,67,72,899,390,879]
    a=np.array(L1)
    Search_Val=int(input("Enter the Searching Element"))
    found=0
    pos=1
```

1. LINEAR SEARCH OR SEQUENTIAL SEARCH

WAPP to search an element in a given list using Linear Search Method using NumPy

```
for i in a:
        if i==Search_Val:
            found=1
            break
        pos= pos+1
    if found:
        print("Element is found at location ",pos)
    else:
        print("Element is not found!")
Linear_Search()
```

1. LINEAR SEARCH OR SEQUENTIAL SEARCH

Linear Search or Sequential Search Analysis

ITEM PRESENT:

Best Case	:	1
Average Case	:	½
Worst Case	:	n

ITME NOT PRESENT:

Best Case	:	n
Average Case	:	n
Worst Case	:	n

2. BINARY SEARCH

2. BINARY SEARCH

What is Binary Search?

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array.

Binary Search. Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

2. BINARY SEARCH

```
def binary_search(alist, key):
    start = 0
    end = len(alist)
    while start < end:
        mid = (start + end)//2
        if alist[mid] > key:
            end = mid
        elif alist[mid] < key:
            start = mid + 1
else:
            return mid
    return -1
```

2. BINARY SEARCH

```
alist = input('Enter the sorted list of numbers: ')
alist = alist.split()
alist = [int(x) for x in alist]
key = int(input('The number to search for: '))

index = binary_search(alist, key)
if index < 0:
    print('{} was not found.'.format(key))
else:
    print('{} was found at index {}.'.format(key,
index))
```

INSERTION IN LINEAR LIST

INSERTION IN LINEAR LIST

Insertion of new element in array can be done in two ways:-

1) If the array is unordered it can be inserted at the end of array.
2) If the array is sorted then new element is added at appropriate position without altering the order of the list or array.

INSERTION IN LINEAR LIST

```
# Function to insert element
def insert(list, n):

    # Searching for the position
    for i in range(len(list)):
        if list[i] > n:
            index = i
            break
```

INSERTION IN LINEAR LIST

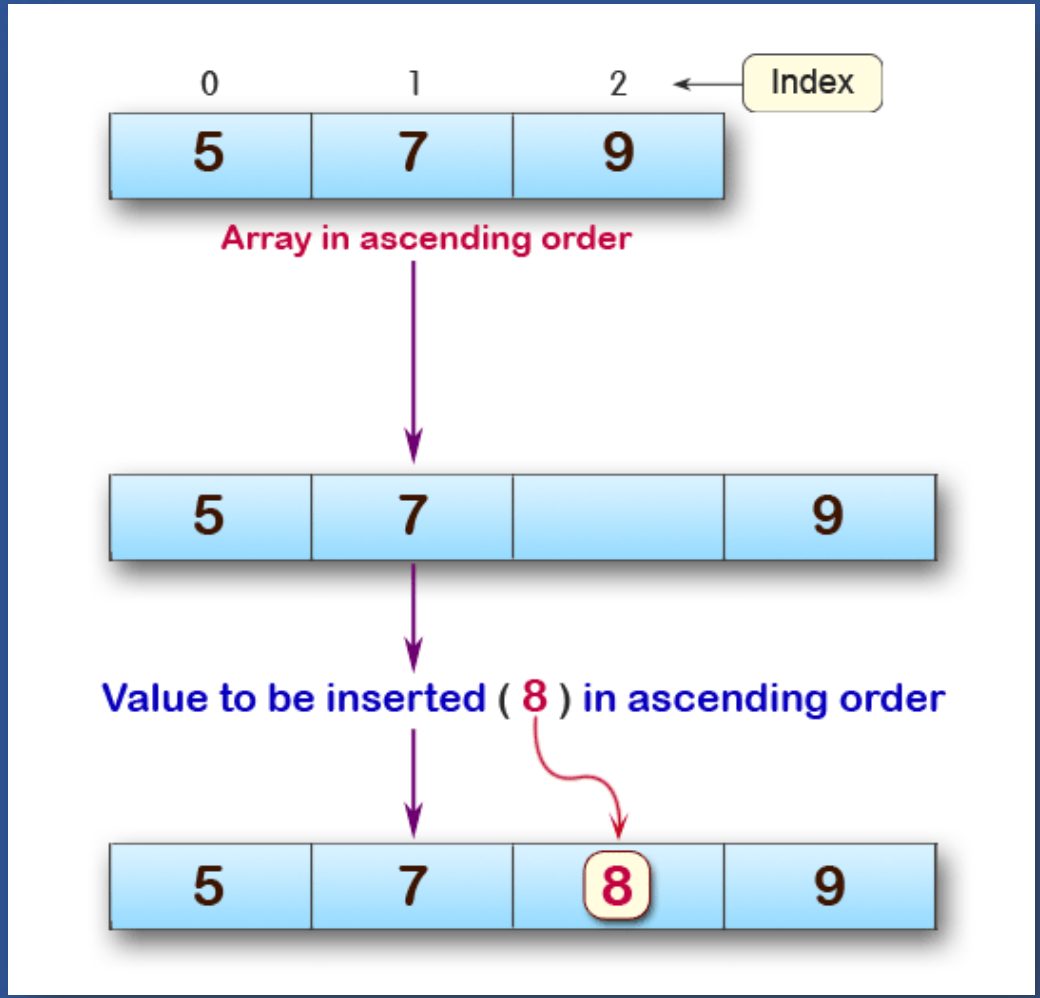
```
# Inserting n in the list
    list = list[:i] + [n] + list[i:]
    return list

# Driver function
list = [1, 2, 4]
n = 3

print(insert(list, n))
```

INSERTING AN ELEMENT IN SORTED ARRAY

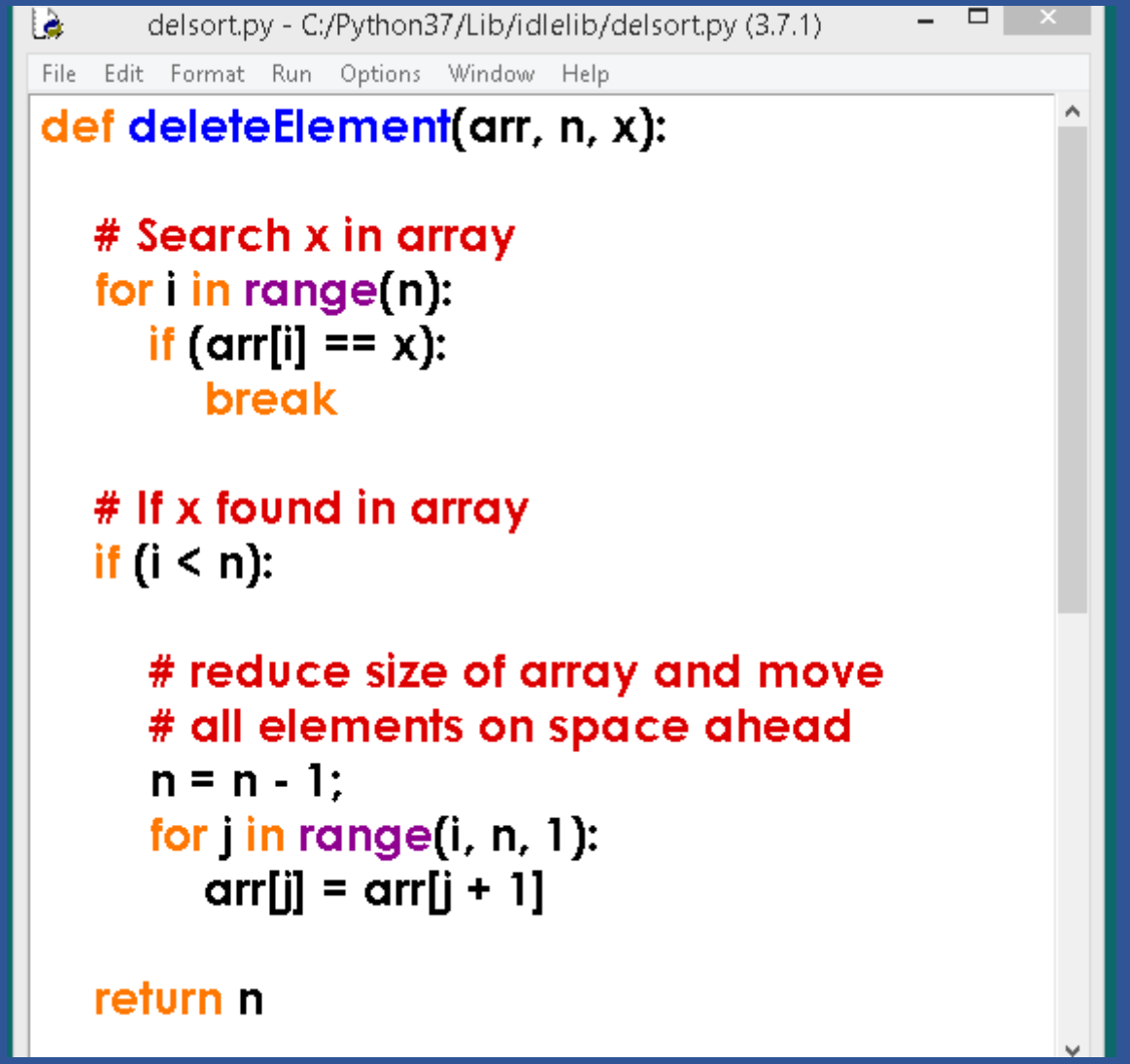
INSERTING AN ELEMENT IN SORTED ARRAY



Inserting new element in sorted list requires the searching the right position and shifting the elements. As shown in fig.

Delete an element from an array

Delete an element from an array



LIST COMPREHENSIONS (LC)

LIST COMPREHENSIONS (LC)

What is List Comprehensions (LC)?

List comprehension is an elegant way to define and create list in python.

We can create lists just like mathematical statements and in one line only. The syntax of list comprehension is easier to grasp.

LIST COMPREHENSIONS

The list comprehension starts with a '[' and ']', to help you remember that the result is going to be a list..

Syntax:

List1=[expression for item in list if conditional]

For example:

```
list2=[i for i in range(0,5)]
print (list2)
```

LIST COMPREHENSIONS - Examples

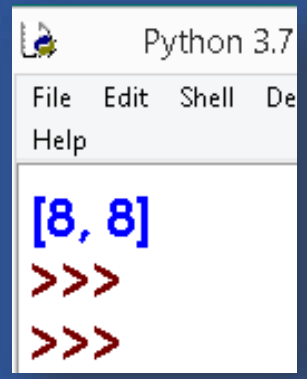
listcompri.py - C:/Python37/Lib/idlelib/listcompri.py (3.7.1)

File Edit Format Run Options Window Help

```
n=4
list2=[n*2 if n<5 else n*4  for i in range(0,2)]
print (list2)
```

Ln: 4 Col: 0

OUTPUT



Understanding the Code:

If n is less than 5 generate 2 values i.e n x 2 as per the for loop else generate 2 values i.e n x 4, the out put will be:- 8 8

LIST COMPREHENSIONS - Examples

Create a list called points

i) Using for loop

ii) Using List comprehension

Ans: i)

```
Points=[]
for i in range (0,5):
    Points.append(i)
```

Ans ii)

```
Points=[i for i in range (0,5)]
```

LIST COMPREHENSIONS - Examples

Create a list called vals which stores square of numbers

Ans:

```
vals=[i*2 for i in range (0,5)]
```

LIST COMPREHENSIONS - Examples

Given an input list scores, produce a list namely scores2, using list comprehension having numbers from scores that are multiples of 4.

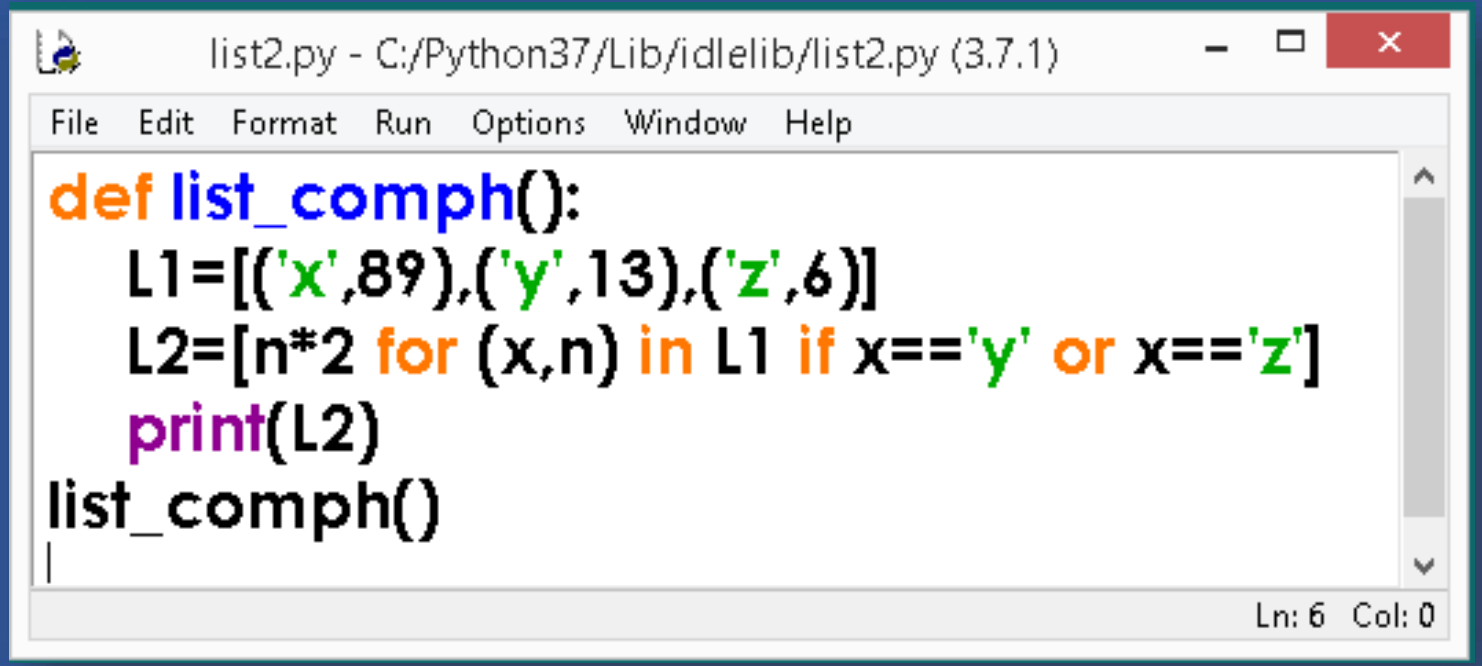
Scores=[23,4,12,16,18,9]

Ans:

```
Scores2=[num for num in scores if
    num%4==0]
```

LIST COMPREHENSIONS - Examples

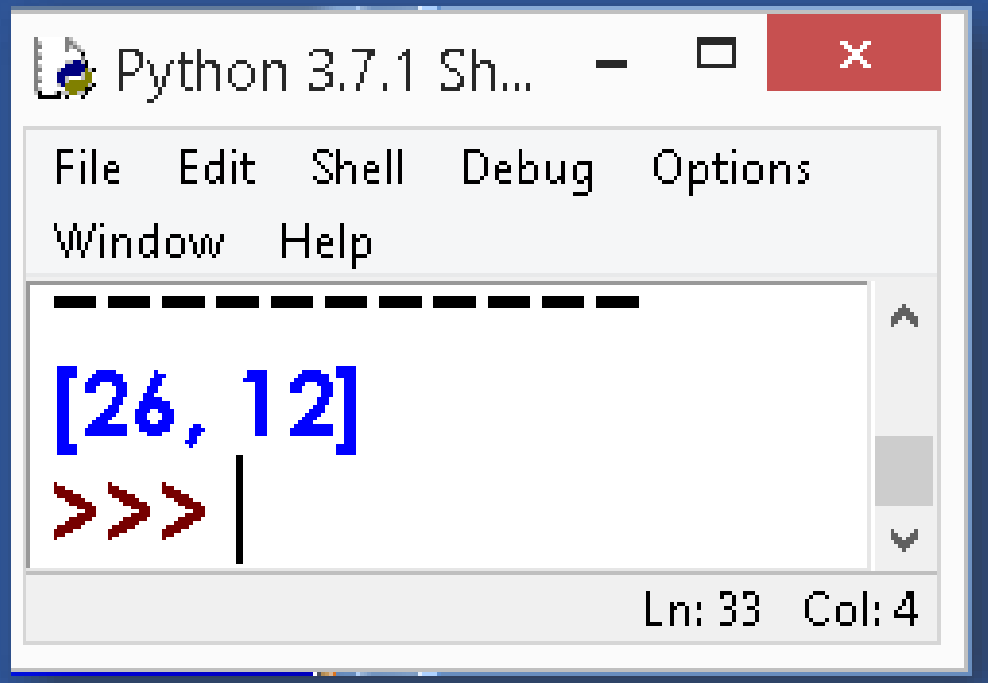
Consider the following code and write the output.



LIST COMPREHENSIONS - Examples

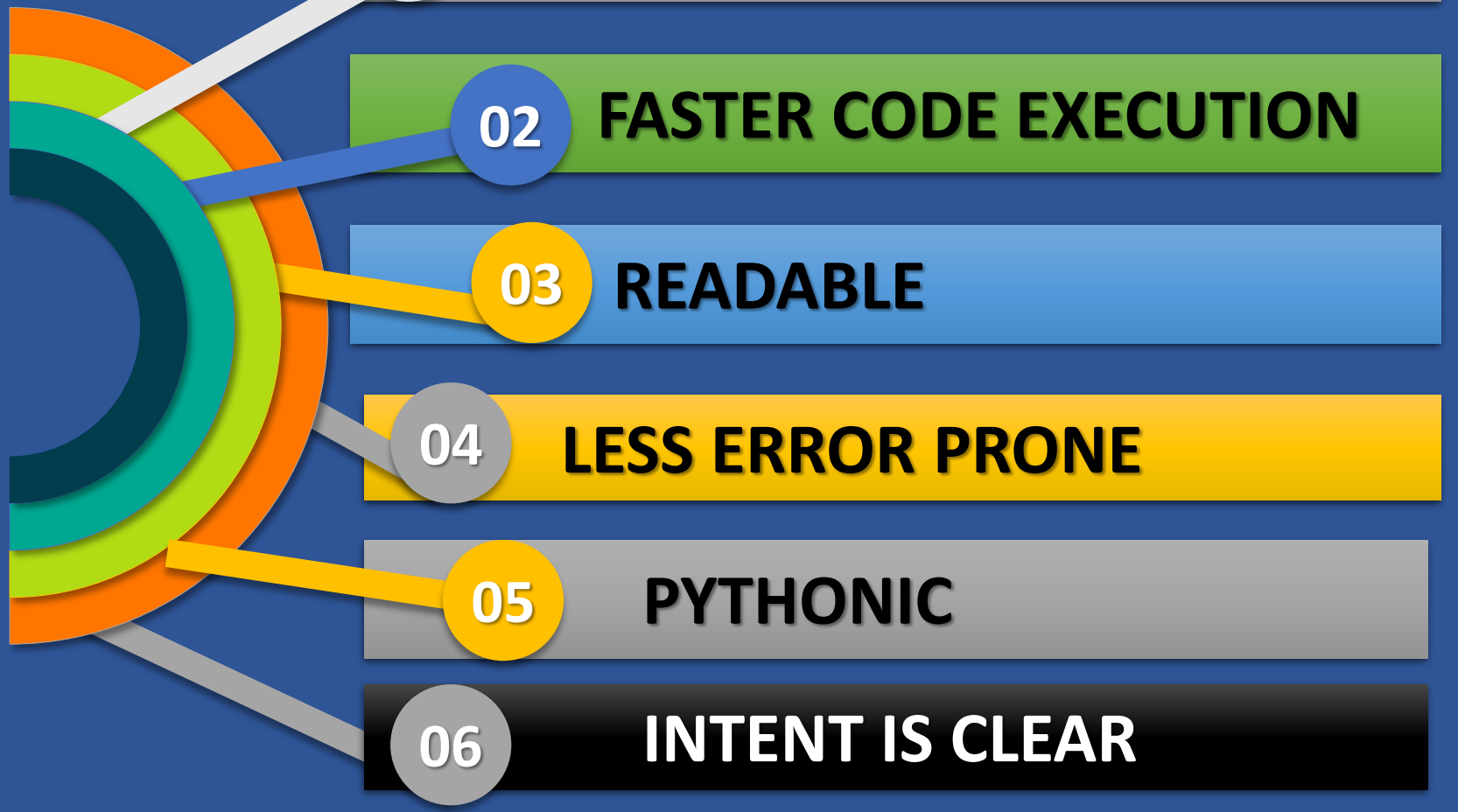
Consider the following code and write the output.

OUTPUT



LIST COMPREHENSIONS - Advantages

LIST COMPREHENSIONS - Advantages



LIST COMPREHENSIONS - Advantages

1. CODE REDUCTION

A code of multiple statements gets reduced to single line of code.

2. FASTER CODE EXECUTION

LC are executed faster than loops and other equivalent statements.

LIST COMPREHENSIONS - Advantages

2. FASTER CODE EXECUTION

There are two reasons for faster execution:

i) Python will allocate memory first before adding elements to it.

ii) Calls append() function get avoided, reducing function overhead time.

LIST COMPREHENSIONS - Advantages

3. READABLE

It is more readable (*when you get used to it*).

4. LESS ERROR PRONE

Once you are familiar with LC, there is less chance to error prone.

LIST COMPREHENSIONS - Advantages

5. PYTHONIC

LC are more PYTHONIC as they truly represent the python code, hence generate interest among developers to write concise and accurate code.

LIST COMPREHENSIONS - Advantages

6. INTENT IS CLEAR

Intent is clear: to initialise a list. A loop could be doing anything, you have to read it to check that it doesn't have any other side effects. A list comprehension just sets up the list, you know it isn't up to anything else.

NESTED LISTS or TWO DIMENSIONAL LISTS

NESTED LISTS or TWO DIMENSIONAL LISTS

A list is a container which holds comma-separated values (items or elements) between square brackets where items or elements need not all have the same type.

A nested list is a list that appears as an element in another list. In this list.

NESTED LISTS or TWO DIMENSIONAL LISTS

```
nums = [[1, 2], [3, 4], [5, 6]]
print(nums[0])
print(nums[1])
print(nums[2])
print(nums[0][0])
print(nums[0][1])
print(nums[1][0])
print(nums[2][1])
```

Creating 2D List

Creating 2D List

```
L1=[]
r=int(input("How many rows?"))
c=int(input("How many Columns?"))
for i in range(r):
    row = []
    for j in range( c):
        ele=int(input("element", i , " , " , j ))
    row.append(ele)
     L1.append(row)
print("List is " ,L1)
```

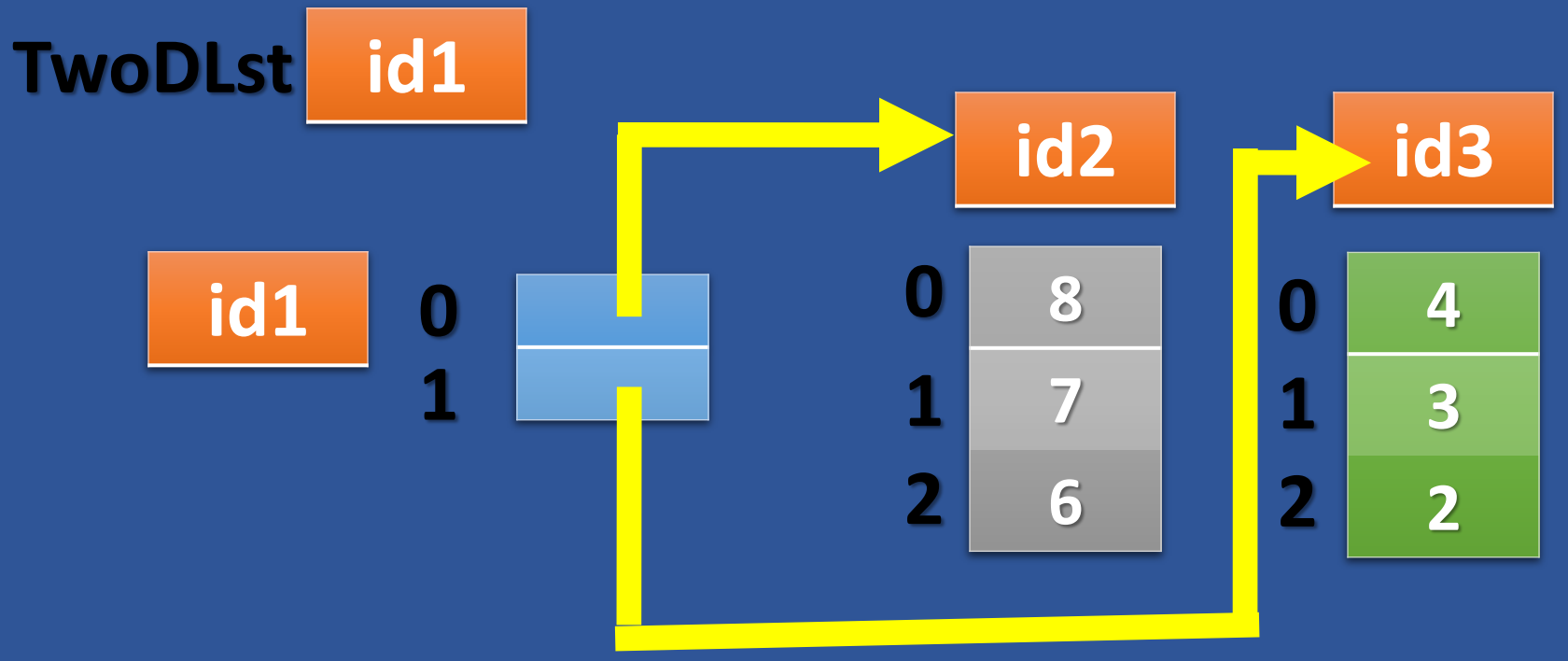
TRAVERSING 2D LIST

```
print("List is " ,L1)
for i in range(r):
    print("\t[",end=" ")
    for j in range( c):
        print(L1[i][j], end=" ")
     print("]")
    print("\t ] " )
```

REPRESENTATION OF 2D LIST IN MEMORY

REPRESENTATION OF 2D LIST IN MEMORY

TwoDLst= [[8 , 7 , 6] , [4 , 3 , 2]]



REPRESENTATION OF 2D LIST IN MEMORY

What is Regular 2D List?

If row size and column size are same then it is said to be **Regular 2D List.**

What is Ragged List?

If row size and column size are not same then it is said to be **Ragged List.**

SLICES IN 2D LISTS

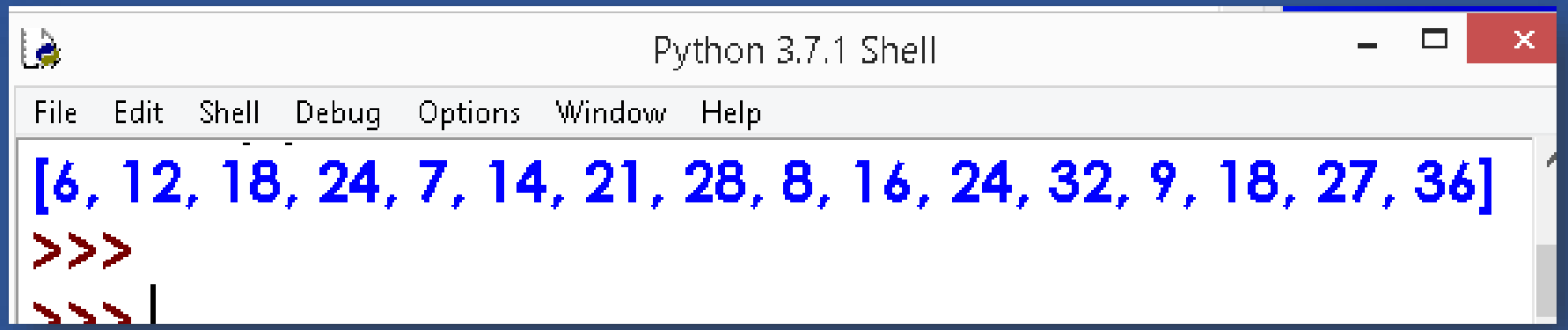
```
TwoDLst= [ [ 8 , 7 , 6 ] , [ 4 , 3 , 2 ] , [ 7 ,3 , 1 ]  ]

>>>TwoDLst[:1]
[ [ 8 , 7 , 6 ] , [ 4 , 3 , 2 ]]
>>> TwoDLst[1:]
[[ 4 , 3 , 2 ] , [ 7 ,3 , 1 ]  ]
```

FIND THE OUTPUT

FIND THE OUTPUT

OUTPUT

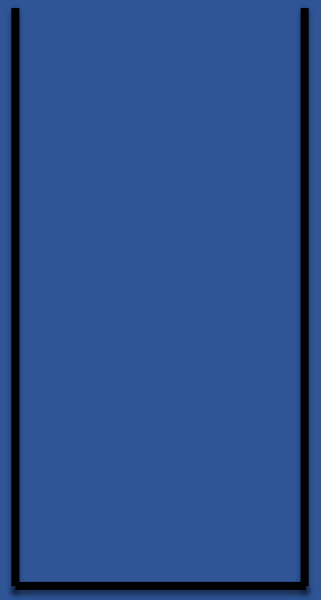


STACK

Stack

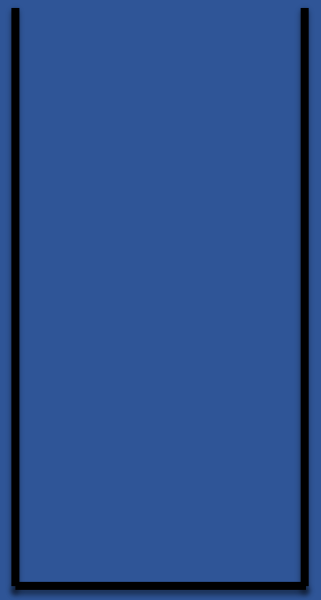
- It is an abstract data type (interface)
- Basic operations: pop(), push() and peek()
- **LIFO** structure: **L**ast **I**n **F**irst **O**ut
- In most high level languages, a stack can be easily implemented either with arrays or linked lists
- A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack

Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)



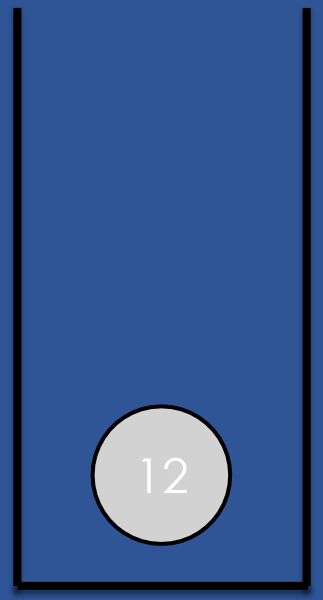
Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)

stack.push(12);



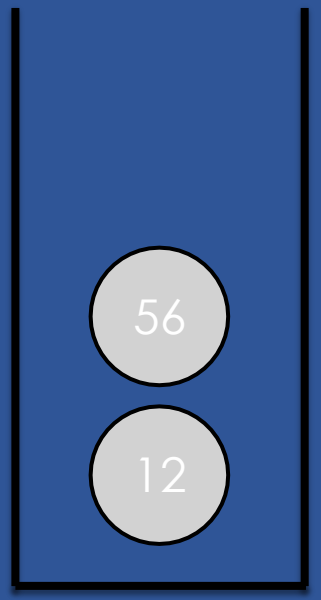
Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)

stack.push(56);



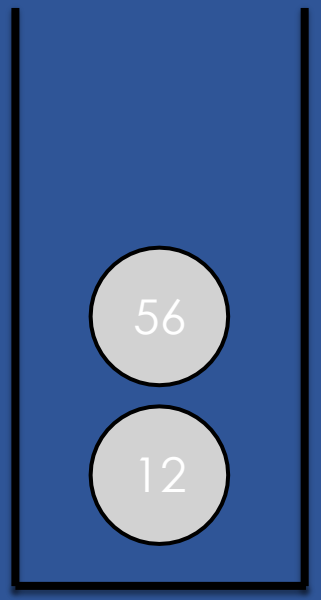
Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)

stack.push(56);



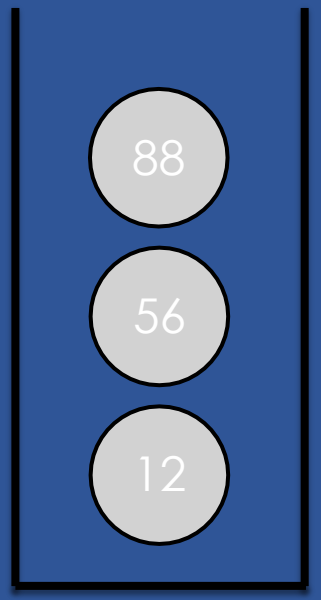
Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)

stack.push(88);

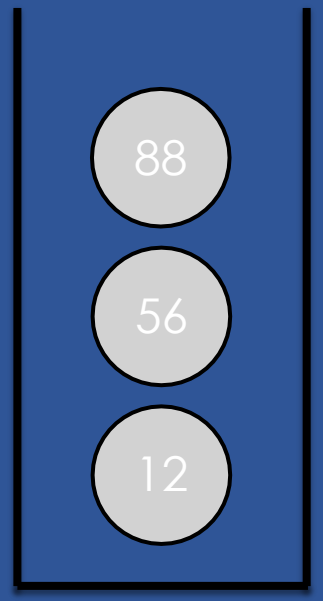


Push operation: put the given item to the top of the stack Very simple operation, can be done in O(1)

stack.push(88);

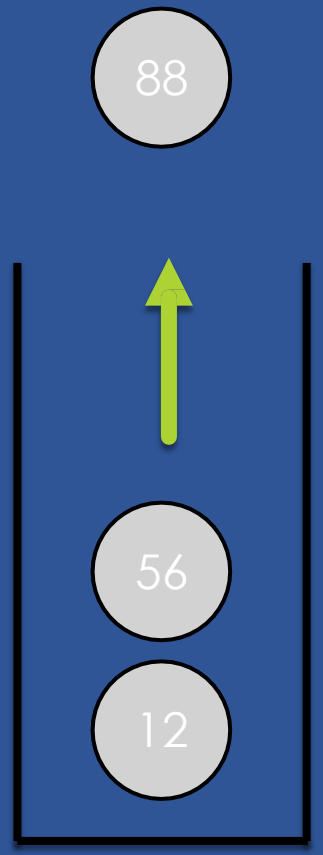


Pop operation: we take the last item we have inserted to the top of the stack (LIFO) Very simple operation, can be done in O(1)

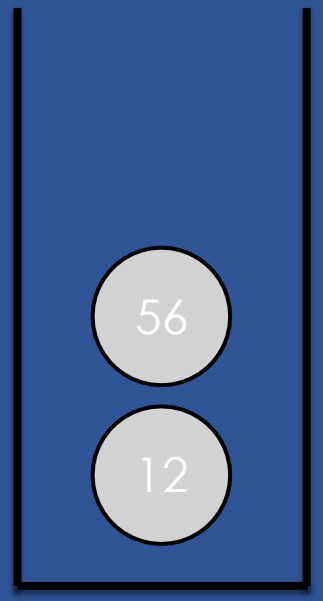


Pop operation: we take the last item we have inserted to the top of the stack (LIFO) Very simple operation, can be done in O(1)

stack.pop();

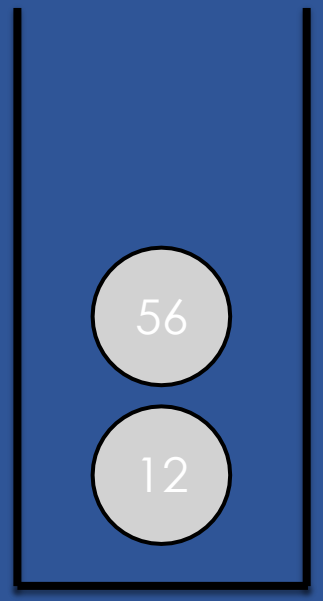


Pop operation: we take the last item we have inserted to the top of the stack (LIFO) Very simple operation, can be done in O(1)



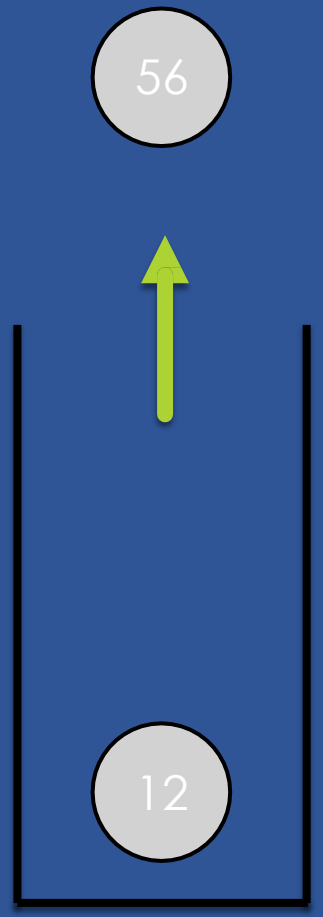
Pop operation: we take the last item we have inserted to the top of the stack (LIFO) Very simple operation, can be done in O(1)

stack.pop();

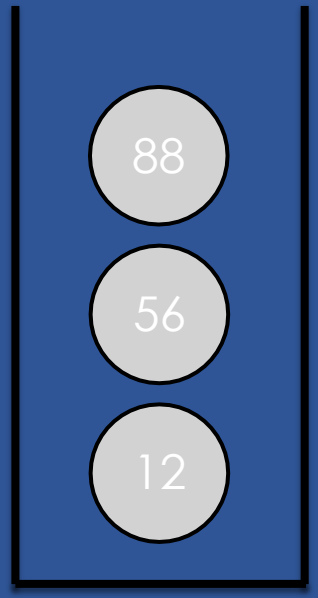


Pop operation: we take the last item we have inserted to the top of the stack (LIFO) Very simple operation, can be done in O(1)

stack.pop();

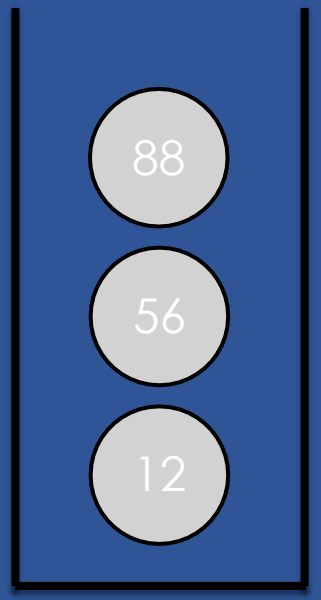


Peek operation: return the item from the top of the stack without removing it Very simple operation, can be done in O(1)



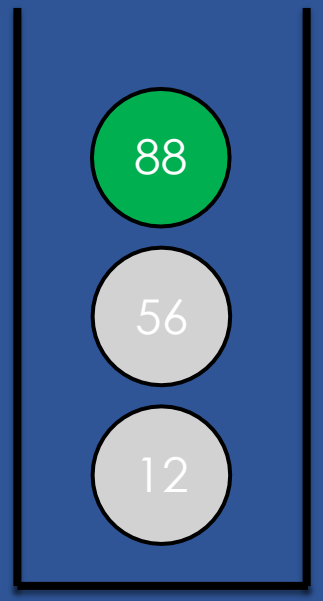
Peek operation: return the item from the top of the stack without removing it Very simple operation, can be done in O(1)

stack.peek();



Peek operation: return the item from the top of the stack without removing it Very simple operation, can be done in O(1)

stack.peek();



The peek() method will return 88 but the structure of the stack remains the same !!!

```
#Q1.WAP to implement operations of stack without using predefined
functions
stack=[];

def getChoice():
    print("Menu\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.EXIT")
    choice = int(input("Enter your choice"));
    return choice;

def pushitem(item):
    stack.append(item);

def popitem():
    global stack;
    item=stack[-1];
    del stack[-1];
    return item;

def display():
    print 'Elements of stack are',stack;
```

```
#Main function starts from here
print("Program starts")
choice = getChoice();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to push"));
        pushitem(item);
    elif choice==2:
        if(len(stack)!=0):
            item=popitem();
            print("Popped item= ",item);
        else:
            print("Stack Underflow");
    elif choice==3:
        if(len(stack)!=0):
            display();
        else:
            print("Stack Underflow");
    else:
        print("Wrong Choice");
    choice=getChoice();
print("Stack Operations are Over");
```

```
#Q2.WAP to implement stack using built-in functions

stack=[];
def getChoice():
    print("Menu\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.EXIT")
    choice = int(input("Enter your choice"));
    return choice;

#Main function starts from here
print("Program starts")
choice = getChoice();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to push"));
        stack.append(item);
    elif choice==2:
        if(len(stack)!=0):
            item=stack.pop();
            print("Popped item= ",item);
        else:
            print("Stack Underflow");
    elif choice==3:
        if(len(stack)!=0):
            print 'Elements of stack are',stack;
        else:
            print("Stack Underflow");
    else:
        print("Wrong Choice");
    choice=getChoice();
print("Stack Operations are  Over");
```

#Q3.WAP to implement stack using class and object

```
class Stack:
    def__init__(self):
        self.stackarr=[];
    def push(self,item):
        self.stackarr.append(item);
    def pop(self):
        item=self.stackarr[-1];
        del self.stackarr[-1];
        return item;
    def display(self):
        print('Values of stack are ',self.stackarr);

def getChoice():
    print("Menu\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.Exit")
    choice = int(input("Enter your choice"));
    return choice
```

```
#Main function starts from here
print("Program starts")
choice = getChoice();
ob = Stack();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to push"));
        ob.push(item);
    elif choice==2:
        if(len(ob.stackarr) != 0):
            item=ob.pop();
            print("Popped item is ", item);
        else:
            print('Stack Underflow');
    elif choice==3:
        if(len(ob.stackarr) != 0):
            ob.display();
        else:
            print('Stack Underflow');
    else:
        print("Invalid choice, please choose again")
        print("\n")
    choice = getChoice();
```

APPLICATIONS OF STACKS

APPLICATIONS OF STACKS

1. EXPRESSION EVALUATION
2. EXPRESSION CONVERSION
3. SYNTAX PARSING
4. BACKTRACKING

Continued….

APPLICATIONS OF STACKS

5. PARENTHESIS CHECKING

6. STRING REVERSAL

7. FUNCTION CALL

APPLICATIONS OF STACKS

1. EXPRESSION EVALUATION

Stack is used to evaluate prefix, postfix and infix expressions. (infix, Prefix and postfix expressions you will study in next topic.)

APPLICATIONS OF STACKS

2. EXPRESSION CONVERSION

An expression can be represented in prefix, postfix or infix notation.

Stack can be used to convert one form of expression to another.

APPLICATIONS OF STACKS

3. SYNTAX PARSING

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

What is parsing?

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language.

APPLICATIONS OF STACKS

4. BACKTRACKING

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

APPLICATIONS OF STACKS

4. BACKTRACKING

What is Maze problem?

A maze is a path or collection of paths, typically from an entrance to a goal.

OR

Find the total number of unique paths which the robot can take in a given maze to reach the destination from given source.

APPLICATIONS OF STACKS

4. BACKTRACKING

The Problem

A robot is asked to navigate a maze. It is placed at a certain position (the *starting* position) in the maze and is asked to try to reach another position (the *goal* position). Positions in the maze will either be open or blocked with an obstacle. Positions are identified by (x,y) coordinates.

APPLICATIONS OF STACKS

5. PARENTHESIS CHECKING

Stack is used to check the proper opening and closing of parenthesis.

6. STRING REVERSAL

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

APPLICATIONS OF STACKS

7. FUNCTION CALL

Stack is used to keep information about the active functions or subroutines.

QUEUE

- It is an abstract data type (interface)
- Basic operations: enqueue() and dequeue() , peek()
- FIFO structure: first in first out
- It can be implemented with dynamic arrays as well as with linked lists

Insert operation: we just simply add the new item to the end of the queue

Insert operation: we just simply add the new item to the end of the queue

queue.insert(10);

Insert operation: we just simply add the new item to the end of the queue

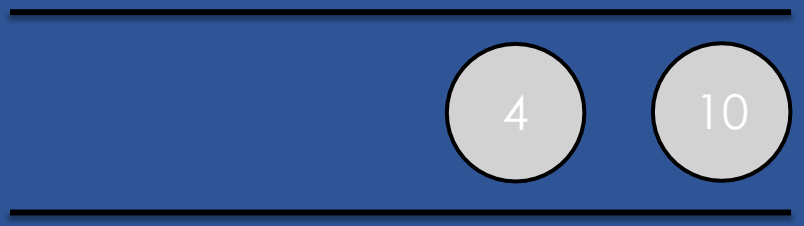
queue.insert(10);

Insert operation: we just simply add the new item to the end of the queue

queue.insert(4);

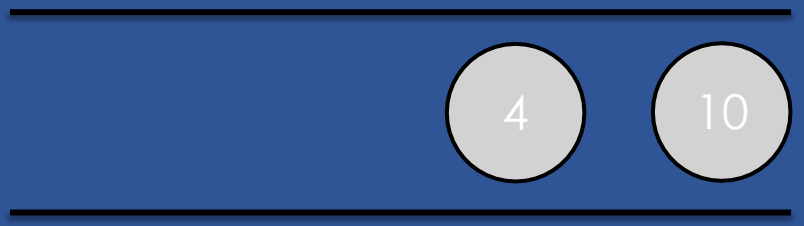
Insert operation: we just simply add the new item to the end of the queue

queue.insert(4);



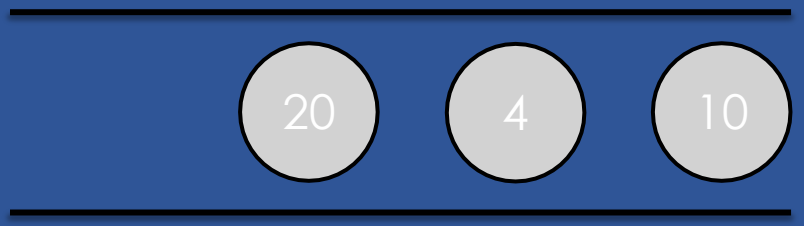
Insert operation: we just simply add the new item to the end of the queue

queue.insert(20);

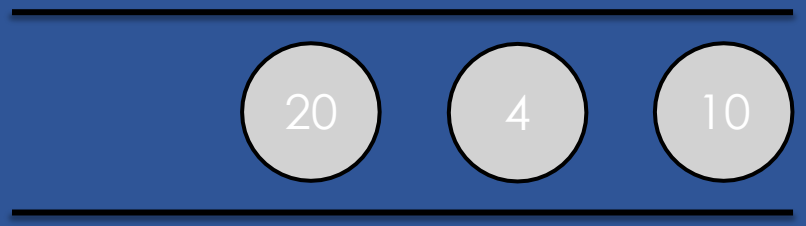


Insert operation: we just simply add the new item to the end of the queue

queue.insert(20);

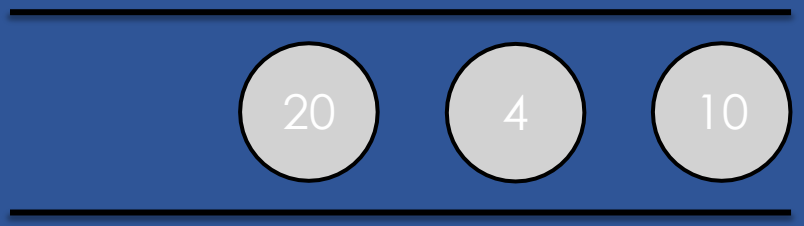


Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure



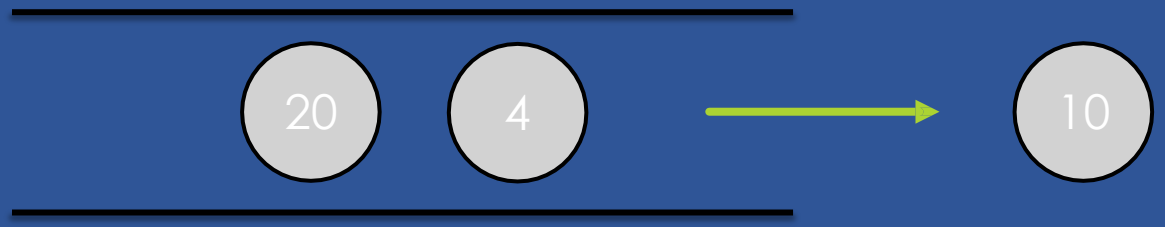
Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

queue.delete();



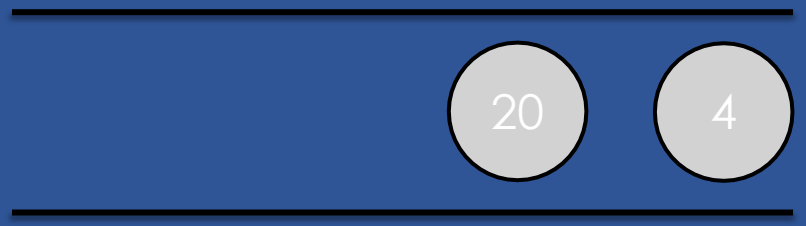
Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

queue.delete();



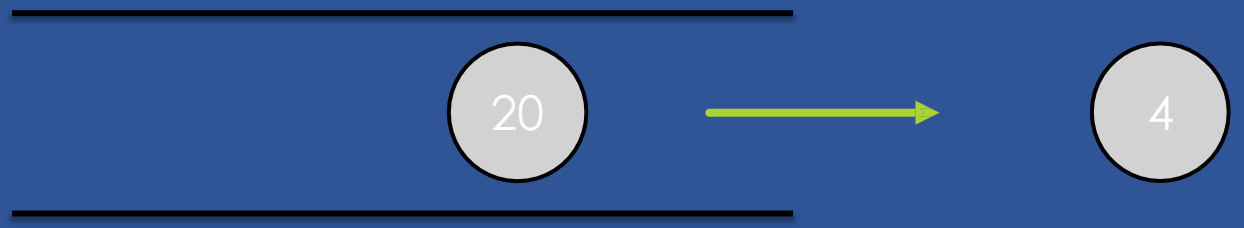
Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.delete();
```



Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.delete();
```



Delete operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

Applications

- When a resource is shared with several consumers (threads): we store them in a queue
- For example: CPU scheduling
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
- For example: IO buffer

```
#Q4.WAP to implement the operations of Linear Queue
Queue=[];

def getChoice():
    print("Menu\n 1.INSERT\n 2.DELETE\n 3.DISPLAY\n
4.EXIT")
    choice = int(input("Enter your choice"));
    return choice;

def insertitem(item):
    Queue.append(item);

def deleteitem():
    global Queue;
    item=Queue[0];
    Queue = Queue[1:len(Queue)];
    return item;

def display():
    print 'Elements of Queue are',Queue;
```

```
#Main function starts from here
print("Program starts")
choice = getChoice();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to insert"));
        insertitem(item);
    elif choice==2:
        if(len(Queue) != 0):
            item=deleteitem();
            print("Deleted item= ",item);
        else:
            print("Queue Underflow");
    elif choice==3:
        if(len(Queue) != 0):
            display();
        else:
            print("Queue Underflow");
    else:
        print("Wrong Choice");
    choice=getChoice();
print("Queue Operations are Over");
```

```
#Q5.WAP to implement Linear Queue using built-in functions
Queue=[];

def getChoice():
    print("Menu\n 1.INSERT\n 2.DELETE\n 3.DISPLAY\n 4.EXIT")
    choice = int(input("Enter your choice"));
    return choice;

#Main function starts from here
print("Program starts")
choice = getChoice();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to insert"));
        Queue.append(item);
    elif choice==2:
        if(len(Queue) != 0):
            #item=Queue.get();
            item=Queue.pop(0);
            print("Deleted item= ",item);
        else:
            print("Queue Underflow");
    elif choice==3:
        if(len(Queue) != 0):
            print 'Elements of Queue are',Queue;
        else:
            print("Queue Underflow");
    else:
        print("Wrong Choice");
    choice=getChoice();
print("Queue Operations are  Over");
```

#Q6.WAP to implement operations of Linear Queue using class and object

```
class Queue:
    def __init__(self):
        self.queuearr = [];
    def insert(self,item):
        self.queuearr.append(item);
    def deletequeue(self):
        item = self.queuearr[0];
        del self.queuearr[0];
        return item;
    def display(self):
        print 'Elements of Queue are',self.queuearr;

def getChoice():
    print("Menu\n 1.INSERT\n 2.DELETE\n 3.DISPLAY\n 4.EXIT")
    choice = int(input("Enter your choice"));
    return choice;
```

```
print("Program starts")
choice = getChoice();
ob = Queue();
while choice!=4:
    if choice==1:
        item=int(input("Enter value to insert"));
        ob.insert(item);
    elif choice==2:
        if(len(ob.queuearr) != 0):
            item=ob.deletequeue();
            print("Deleted item= ",item);
        else:
            print("Queue Underflow");
    elif choice==3:
        if(len(ob.queuearr) != 0):
            ob.display();
        else:
            print("Queue Underflow");
    else:
        print("Wrong Choice");
    choice=getChoice();
print("Queue Operations are Over");
```

Assignments

- Implement operations of Stack using array
- Implement operations of Linear Queue using array
- Implement Circular Queue
- Implement Linear Queue using two stacks

APPLICATIONS OF QUEUES

APPLICATIONS OF QUEUES

1. Serving requests on a single shared resource, like a printer.

2. CPU task scheduling.

3. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

APPLICATIONS OF QUEUES

4.When data is transferred asynchronously between two processes. eg. IO Buffers.

5.When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

6.In recognizing palindrome.

APPLICATIONS OF QUEUES

7.In shared resources management.

8.Keyboard buffer.

9.Round robin scheduling.

10.Job scheduling.

11.Simulation

TYPES OF QUEUES

TYPES OF QUEUES

Queue is an important structure for storing and retrieving data and hence is used extensively among all the data structures. Types of Queues are:-

1. SIMPLE QUEUE

2. CIRCULAR QUEUE

TYPES OF QUEUES

1. SIMPLE QUEUE

As studied in previous topic.

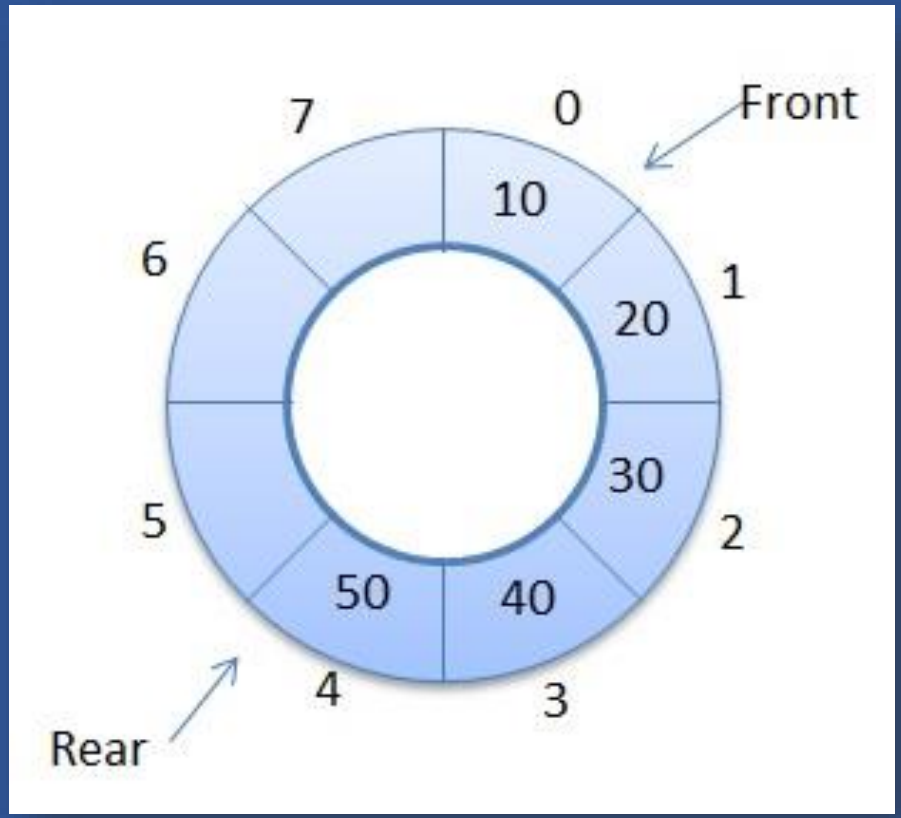
TYPES OF QUEUES

2. CIRCULAR QUEUE

Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node's pointer is also connected to the first node's address. Hence, the last node and the first node also gets connected making a circular link overall.

TYPES OF QUEUES

2. CIRCULAR QUEUE



EXPRESSION CONVERSION METHOD

EXPRESSION CONVERSION METHOD

Converting from Infix to Postfix

Infix expressions can be easily converted by hand to postfix notation.

The expression :

A + B - C

would be written as AB+C- in postfix form

EXPRESSION CONVERSION METHOD

Converting from Infix to Postfix

STEP 1

A*B+C/D

1. Place parentheses around every group of operators in the correct order of evaluation. There should be one set of parentheses for every operator in the infix expression.

   ((A * B) + (C / D))

Converting from Infix to Postfix

STEP 2

2. For each set of parentheses, move the operator from the middle to the end preceding the corresponding closing parenthesis. ((A B *) (C D /) +)

Converting from Infix to Postfix

STEP 3

3. Remove all of the parentheses, resulting in the equivalent postfix expression.

A B * C D / +

CONVERSION METHOD

CONVERSION – Example 1

For example convert the infix expression

(A+B)*(C-D)/E

into postfix expression showing stack status after every step.

Symbol scanned	Stack status	Postfix expr
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	
*	(*	
(	(*(	
C	(*(	AB+C
-	(*(-	AB+C
D	(*(-	AB+CD
)	(*	AB+CD-
/	(/	AB+CD-*
E	(/	AB+CD-*E
)		AB+CD-*E/

Answer : AB+CD-*E/

EVALUATION – Example 2

Evaluate the following postfix expression showing stack status after every step

8, 2, +, 5, 3, -, *, 4 /

EVALUATION – Example 2

8, 2, +, 5, 3, -, *, 4 /

scanned	Stack status	Operation performed
8	8	Push 8
2	8, 2	Push 2
+	10	Op2=pop() i.e 2 Op1=pop() i.e 8 Push(op1+op2) i.e. 8+2
5	10, 5	Push(5)
3	10, 5, 3	Push(3)
-	10, 2	Op2=pop() i.e. 3 Op1=pop() i.e. 5 Push(op1-op2) i.e. 5-3

EVALUATION – Example 2

8, 2, +, 5, 3, -, *, 4 /

scanned	Stack status	Operation performed
*	20	Op2=pop() i.e. 2 Op1=pop() i.e. 10 Push(op1-op2) i.e. 10*2
4	20, 4	Push 4
/	5	Op2=pop() i.e. 4 Op1=pop() i.e. 20 Push(op1/op2) i.e. 20/4
NULL	Final result 5	Pop 5 and return 5

Answer : 5

EVALUATION – Example 3

Evaluate the following Boolean postfix expression showing stack status after every step.

True, False, True, AND, OR, False, NOT, AND

EVALUATION – Example 3

True, False, True, AND, OR, False, NOT, AND

scanned	Stack status	Operation performed
True	True	Push True
False	True, False	Push False
True	True, False, True	Push True
AND	True, False	Op2=pop() i.e. True Op1=pop() i.e. False Push(Op2 AND Op1) i.e. False AND True=False
OR	True	Op2=pop() i.e. False Op1=pop() i.e. True Push(Op2 OR Op1) i.e. True OR False=True

EVALUATION – Example 3

True, False, True, AND, OR, False, NOT, AND

Scanned	Stack status	Operation Performed
False	True, False	Push False
NOT	True, True	Op1=pop() i.e. False Push(NOT False) i.e. NOT False=True
AND	True	Op2=pop() i.e. True Op1=pop() i.e. True Push(Op2 AND Op1) i.e. True AND True=True
NULL	Final result True	Pop True and Return True

Answer :True

EXPRESSION TRANSLATION CLASSWORK/HOME WORK

INFIX EXPRESSIONS INTO POSTFIX

Translate each of the following infix expressions into postfix.

(a) (A * B) / C
(b) (A - (B * C) + D / E
(c) (X - Y) + (W * Z) / V
(d) V * W * X + Y - Z
(e) A / B * C - D + E