COMPUTER SCIENCE With python

Textbook for Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

9 Data Structures – I : Linear Lists

In This Chapter

9.1 INTRODUCTION

The computer system is used essentially as data manipulation system where '*Data*' are very important thing for it. The data can be referred to in many ways *viz.* data, data items, data structures etc. The *data* being an active participant in the organizations' operations and planning, data are aggregated and summarized in various meaningful ways to form *information*. The *data* must be *represented, stored, organized, processed* and *managed* so as to support the user environment. All these factors very much depend upon the way data are aggregated. The *Data structures* are an effective and reliable way to achieve this.

First part of this chapter introduces data structures, basic terminology used for them, and different commonly used data structures.

The *second* part of this chapter talks about list data structures. You have learnt about list data type in your previous class, but here you shall see them in data structure avatar.

9.2 ELEMENTARY DATA REPRESENTATION

Elementary representation of data can be in forms of *raw data, data item, data structures.*

- **Raw data** are raw facts. These are simply values or set of values.
- **Data item** represents single unit of values of certain type.

Data items are used as per their associated data types.

DATA STRUCTURE

A *Data Structure* is a named group of data of different data types which is stored in a specific way and can be processed as a single unit. A data structure has well-defined operations, behaviour and properties.

While designing data structures, one must determine the logical picture of the data in a particular program, choose the representation of the data, and develop the operations that will be applied on.

Data Type vs. Data Structure

A ***Data type*** defines a set of values along with well-defined operations stating its input-output behaviour *e.g.*, you cannot put decimal point in an integer or two strings cannot not be multiplied etc.

On the other hand, a ***Data structure*** is a physical implementation that clearly defines a way of storing, accessing, manipulating data stored in a data structure. The data stored in a data structure has a specific work pattern *e.g.*, in a stack, all insertions and deletions take place at one end only.

For example, Python supports **lists** as a collection data type which can store heterogeous types of elements, but when we implement *List as a data structure*, its behaviour is clearly predecided *e.g.*, it will store all elements of same type ; for searching, list elements will be presorted and so forth.

9.3 DIFFERENT DATA STRUCTURES

Data Structures are very important in a computer system, as these not only allow the user to combine various data types in a group but also allow processing of the group as a single unit thereby making things much simpler and easier. The data structures can be classified into following *two* types :

1. ***Simple Data Structures***. These data structures are normally built from primitive data types like integers, reals, characters, boolean. Following data structures can be termed as simple data structures :
   - Array or Linear Lists
2. ***Compound Data Structures***. Simple data structures can be combined in various ways to form more complex structures called *compound data structures*. Compound data structures are classified into following *two* types :
   - *Linear data structures*. These data structures are single level data structures. A data structure is said to be linear if its elements form a sequence. Following are the examples of linear data structures : (*a*) Stack (*b*) Queue (*c*) Linked List
   - *Non-Linear data structures*. These are multilevel data structures. Example of non-linear data structure is *Tree*.

> **NOTE**
>
> Python's built-in linear data structures are : *lists, tuples, dictionaries, sets.*

Following figure (Fig. 9.1) shows all the data structures.

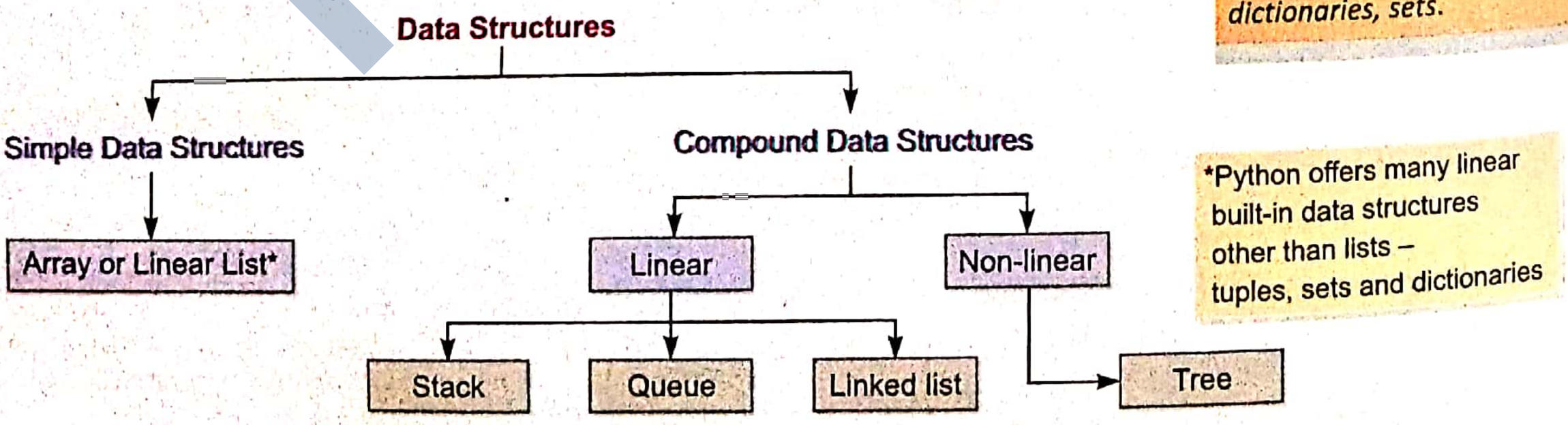


Figure 9.1 Different Data Structures

9.3.1 Linear Lists Arrays

Linear Lists or Arrays refer to a named list of a finite number n of **similar data elements.** Each of the data elements can be referenced respectively by a set of consecutive numbers, usually 0, 1, 2, 3,...n. If the name of a linear list of 10 elements is LIL, then its elements will be referenced as shown : LIL[0], LIL[1], LIL[2], LIL[3], ... LIL[9]

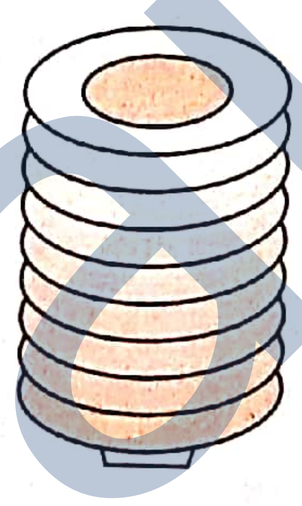
Arrays can be *one dimensional, two dimensional* or *multi dimensional*. In Python, arrays are implemented through List data types as *Linear Lists* or through NumPy arrays.

9.3.2 Stacks

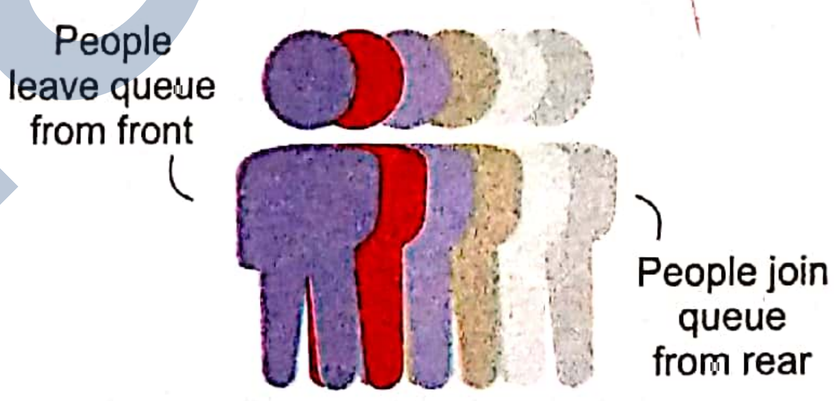
Stacks data structures refer to the lists stored and accessed in a special way, where LIFO (*Last In First Out*) technique is followed. In Stacks, insertions and deletions take place only at one end, called the top. Stack is similar to a stack of plates as shown in Fig. 9.2(*a*). Note that plates are inserted or removed only from the top of the stack.

9.3.3 Queues

Queues data structures are FIFO (*First In First Out*) lists, where insertions take place at the "rear" end of the queues and deletions take place at the "front" end of the queues. Queue is much the same as a line of people [shown in Fig. 9.2(*b*)] waiting for their turn to vote. First person will be the first to vote and a new person can join the queue, at the rear end of it.



(*a*) Stack of plates



(*b*) Queue of people waiting for their turn

Figure 9.2 A stack and a queue.

9.3.4 Linked Lists

Linked lists are special lists of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a *node*, which has two parts. The *INFO* part which stores the information and the reference-pointer part, which points to *i.e.*, stores the reference of the next element. Figure 9.3 shows both types of lists (singly linked lists and doubly linked lists).

In singly linked lists, nodes have one reference-pointer (*next*) pointing to the next node, whereas nodes of doubly linked lists have two reference-pointers (*prev* and *next*). *Prev* points to the previous node and *next* points to the next node in the list.

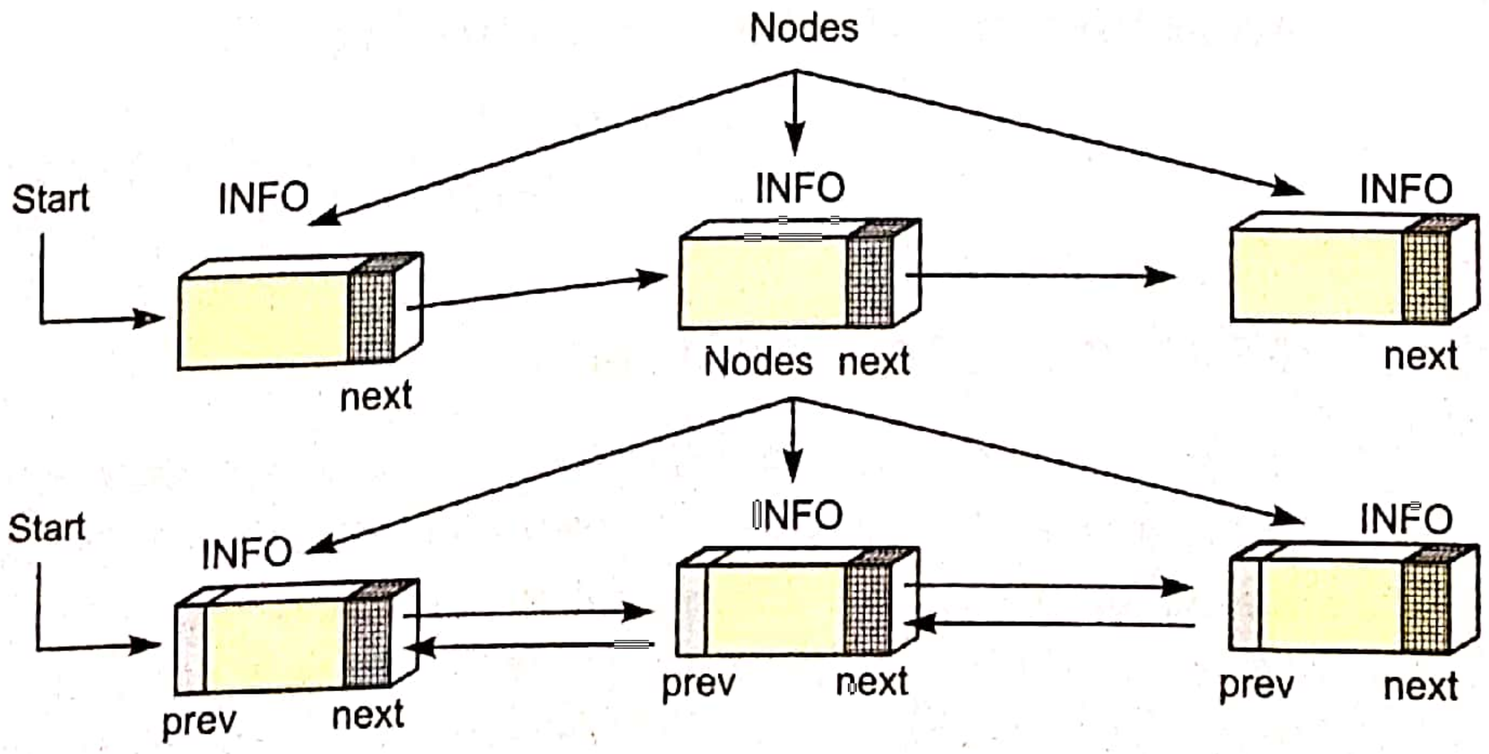


Figure 9.3 Singly and doubly linked lists.

9.3.5 Trees

Trees are multilevel data structures having a hierarchical relationship among its elements called *nodes. Topmost node is called the* **root** *of the tree and bottommost nodes are called* **leaves** *of the tree.*

Each of the nodes has some reference-pointers, pointing to (*i.e.*, storing the reference of) the nodes below it.

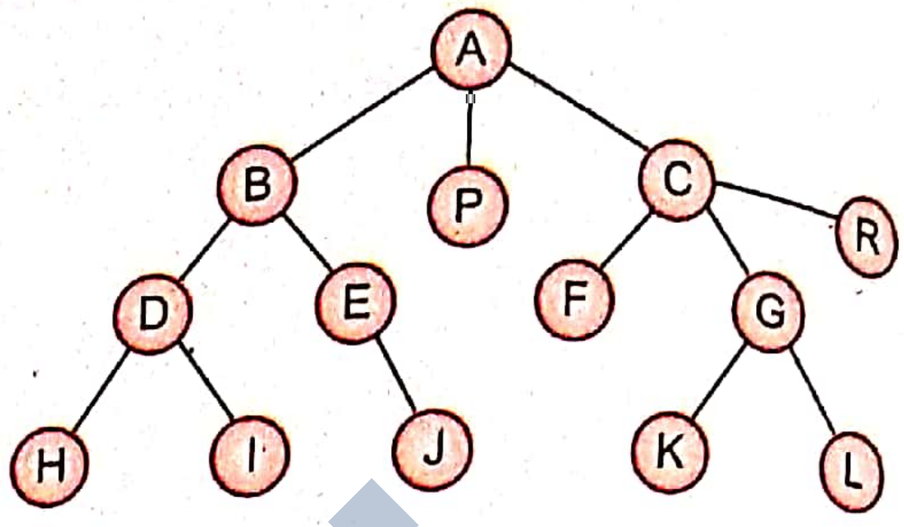


Figure 9.4 Trees.

9.4 OPERATIONS ON DATA STRUCTURES

The basic operations that are performed on data structures are as follows :

1. ***Insertion.*** Insertion means addition of a new data element in a data structure.
2. ***Deletion.*** Deletion means removal of a data element from a data structure. The data element is searched for before its removal.
3. ***Searching.*** Searching involves searching for the specified data element in a data structure.
4. ***Traversal.*** Traversal of a data structure means processing all the data elements of it, one by one.
5. ***Sorting.*** Arranging data elements of a data structure in a specified order is called sorting.
6. ***Merging.*** Combining elements of two similar data structures to form a new data structure of same type, is called *merging*.

9.5 LINEAR LISTS

A linear data structure is that whose elements form a sequence. When elements of linear structures are homogeneous and are represented in memory by means of sequential memory locations, these linear structures are called *arrays*. Linear Lists or **Arrays** are one of the simplest data structures and are very easy to traverse, search, sort etc. *An array stores a list of finite number (n) of homogeneous data elements* (*i.e., data elements of the same type*). The number *n* is called length or size or range of a linear list. When upper bound and lower bound of a linear list are given, its size is calculated as follows :

$$\textit{Linear list size (length)} = UB - LB + 1$$

(UB - *Upper Bound,* LB - *Lower Bound*)

For instance, if a linear list or an array has elements numbered as $-7, -6, -5 \ldots .0, 1, 2, \ldots .15$, then its UB is 15 and LB is -7 and array length is

$$= 15 - (-7) + 1$$
$$= 15 + 7 + 1 = 23$$

LINEAR LIST

A **linear list** is a sequence of $n \geq 0$ elements, $E_1, E_2, \ldots E_n$ where each element E_i has the same data type T.

9.6 LINEAR LIST DATA STRUCTURE

You have learnt about **list** datatype in class XI, but when we talk about lists as data structures, it means that all its implementation functions along with storage details (*e.g.*, size) are clearly defined at one place. In the following lines, we shall first talk about various operations performed on linear list and then finally have a complete program implementing linear lists. Here, please note that we shall implement linear list data structure based on what you have learnt in lists in class XI.

9.6.1 Searching in a Linear List

There are many different searching algorithms : *linear search* and *binary search*.

In linear search, each element of the array/linear list is compared with the given *Item* to be searched for, one by one.

Linear Search

In linear search, each element of the array/linear list is compared with the given *Item* to be searched for, one by one. This method, which traverses the array/linear list sequentially to locate the given *Item*, is called *linear search* or *sequential search*.

Algorithm *Linear Search in Linear List*

```
#Initialise counter by assigning lower bound value of the linear list
Step 1.   Set ctr = L                          # L (Lower bound) is 0 (zero).
          # Now search for the ITEM
Step 2.       Repeat steps 3 through 4 until ctr > U.   #U (Upper bound) is size-1
Step 3.       IF AR[ctr] == ITEM then
              {   print("Search Successful")
                  print(ctr, "is the location of", ITEM)
                                                  # go out of loop
                  break
              }
Step 4.       ctr = ctr + 1
              # End of Repeat
Step 5.       IF ctr > U then
                  print("Search Unsuccessful !")
Step 6.       END
```

The above algorithm searches for ITEM in linear list AR with lower bound *L* and upper bound *U*. As soon as the search is successful, it jumps out of the loop (*break* statement), otherwise continues till the last element.

Program 9.1 Linear Searching in an array (linear list)

```
# linear search
def Lsearch(AR, ITEM) :
    i = 0
    while i < len(AR) and AR[i] != ITEM :
        i += 1
    if i < len(AR) :
        return i
    else :
        return False

# ---- main ----
N = int(input("Enter desired linear-list size (max. 50)... "))
print("\nEnter elements for Linear List\n")
AR = [0] * N                          # initialize List of size N with zeros
```

```
for i in range(N) :
    AR[i] = int(input("Element" + str(i)+ ":"))

ITEM = int(input("\nEnter Element to be searched for ..."))
index = Lsearch(AR, ITEM)

if index :
    print("\nElement found at index :", index, ", Position : ",(index + 1))
else :
    print("\nSorry!! Given element could not be found.\n")
```

Sample run of above code :

```
Enter desired linear-list size (max. 50) ... 7
Enter elements for Linear List
Element 0 :   88
Element 1 :   77
Element 2 :   44
Element 3 :   33
Element 4 :   22
Element 5 :   11
Element 6 :   10
Enter Element to be searched for ... 11
Element found at index : 5, Position : 6
Enter Element to be searched for ... 78
Sorry!! Given element could not be found.
```

NOTE

In this chapter, we are alternatively using words linear-lists or arrays. All linear-lists being considered here are assumed to store homogenous elements, just like arrays.

The above program reads a linear-list and asks for the item to be searched for. Then it calls a function called **Lsearch( )** which receives linear-list and search-item as parameters. It then searches for given item in the passed linear-list. If the item is found, then it returns the *index* of found element otherwise it returns ***False***.

The above search technique will prove the worst, if the element to be searched is one of the last elements of the linear list as so many comparisons would take place and the entire process would be time-consuming. *To save on time and number of comparisons, binary search is very useful.*

Binary Search

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires the array, to be scanned, must be sorted in any order (for instance, say ascending order). In binary search, the *ITEM* is searched for in smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the middle element, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned. The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the ITEM is not found (search unsuccessful).

NOTE

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

Algorithm *Binary Search in Linear List*

Case I : Array AR[L : U] is stored in ***ascending*** **order**

```
# Initialise segment variables
1.  Set beg = L, last = U                    # L is 0 and U is size-1.
2.  while beg <= last, perform steps 3 to 6  # INT( ) is used to extract integer part
3.      mid = INT ( (beg + last)/2)
4.      if AR [mid] == ITEM then
            print("Search Successful")
            print(ITEM, "found at", mid)
            break                                    # go out of the loop
5.      if AR[mid] < ITEM then
            beg = mid + 1
6.      if AR[mid] > ITEM then
            last = mid – 1
    # End of while
7.  if beg ≠ last
        print("Unsuccessful Search")
8.  END.
```

Case II : Array AR[L : U] is stored in ***descending*** order

```
# Initialize
:
if AR[mid] = =ITEM then
:
if AR[mid] < ITEM then
     last = mid – 1
if AR[mid] > ITEM then
     beg = mid + 1
:    # Rest is similar to the algorithm in Case I.
```

Program 9.2 **Binary Searching in an array.**

```
def Bsearch( AR, ITEM) :
    beg = 0
    last = len(AR) - 1
    while (beg <= last) :
        mid = (beg + last )/2
        if (ITEM == AR[mid]) :
            return mid
        elif (ITEM > AR[mid]) :
            beg = mid + 1
        else :
            last = mid-1
    else :
        return False                          #when ITEM not found
```

```
# __main__
N = int(input("Enter desired linear-list size (max. 50)..."))
print("\nEnter elements for Linear List in ASCENDING ORDER\n")
AR = [0] * N                          # initialize List of size N with zeros
for i in range(N) :
    AR[i] = int (input("Element" + str(i)+" :"))

ITEM = int(input("\nEnter Element to be searched for ..."))
index = Bsearch(AR, ITEM)

if index :
    print("\nElement found at index : ", index, ", Position : ", (index + 1))
else :
    print("\nSorry!! Given element could not be found.\n")
```

Sample run of the above code as shown on the right.

```
Enter desired linear-list size (max. 50)... 8
Enter elements for Linear List in ASCENDING ORDER
Element 0 :   11
Element 1 :   15
Element 2 :   18
Element 3 :   21
Element 4 :   23
Element 5 :   25
Element 6 :   27
Element 7 :   29
Enter Element to be searched for ... 29
Element found at index : 7, Position :  8
```

9.6.2 Insertion in a Linear List

Insertion of new element in array can be done in *two* ways : (*i*) if the array is unordered, the new element is inserted at the end of the array, (*ii*) if the array is sorted then new element is added at appropriate position without altering the order and to achieve this, rest of the elements are shifted.

For instance, 35 is to be added in an array as shown in Fig. 9.5(*a*).

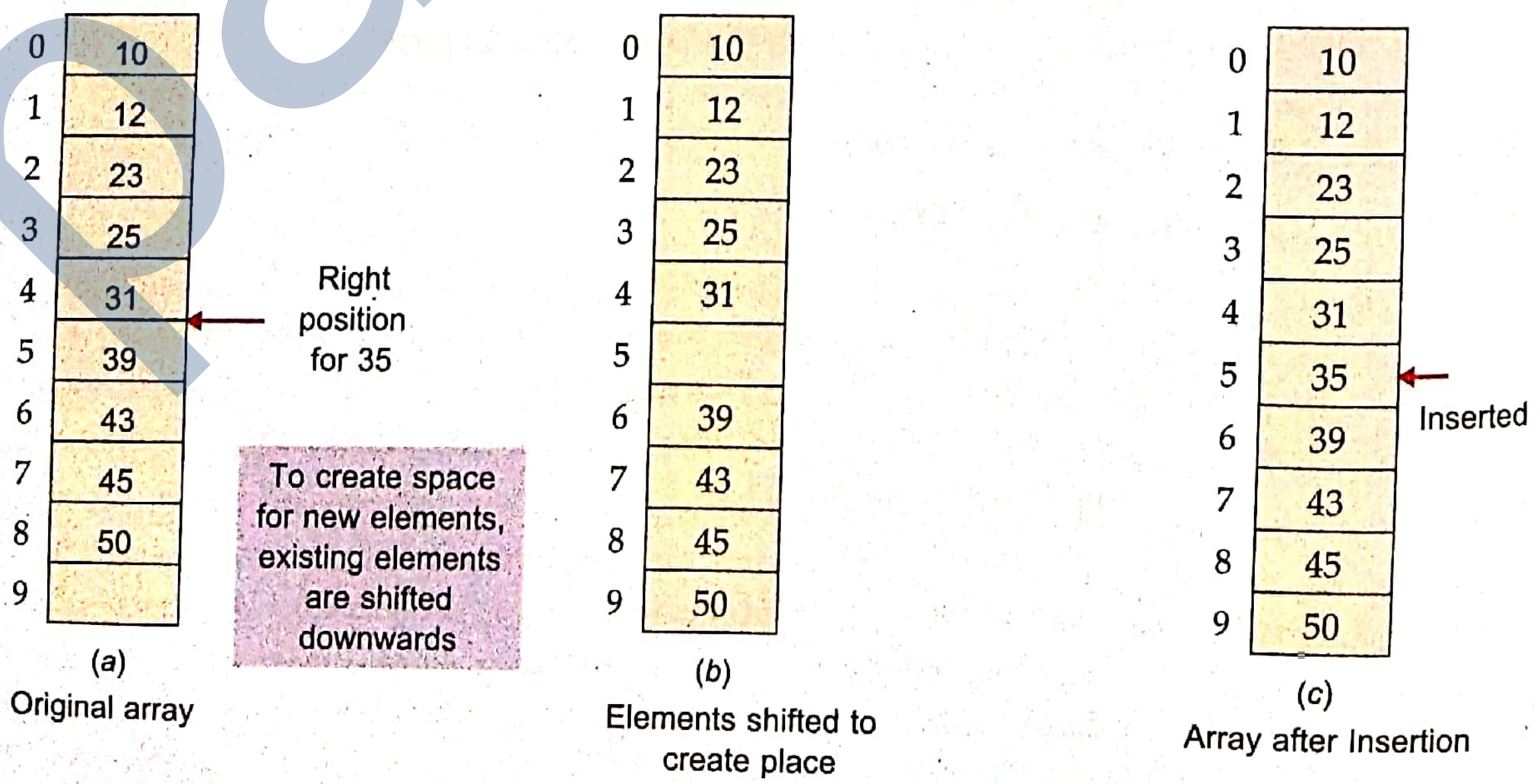


Figure 9.5 Insertion in a sorted array.

If the array is already full, then insertion of an element into its results into *OVERFLOW*.

Algorithm *Insertion in Linear List*

```
# First the appropriate position for ITEM is to be determined i.e., if the
appropriate position is I+1 then AR[ I ] <= ITEM <= AR[I + 1], LST specifies
maximum possible index in array, u specifies index of last element in Array
    1.      ctr = L                                  # Initialise the counter
    2.      If LST = U then
                Print("Overflow :")
                Exit from program
    3.      if AR[ctr] > ITEM then
                pos = 1
            else
            {
    4.          while ctr < U perform steps 5 and 6
    5.              if AR[ctr] <= ITEM and ITEM <= AR[ctr + 1] then
                            pos = ctr + 1
                            break
    6.              ctr = ctr + 1
    7.          if ctr = U then
                        pos = U + 1
            }                                        # end of if step 3
            # shift the elements to create space
    8.      ctr = U                                  # Initialise the counter
    9.      while ctr >= pos perform steps 10 through 11
    10.     {   AR[ctr + 1 ] = AR[ctr]
    11.         ctr = ctr – 1
            }
    12.     AR[pos] = ITEM                           # Insert the element
    13.     END.
```

You have just now read the traditional algorithm of inserting an element in a sorted list, given above. If you notice, the above algorithm involves shifting of elements. Shifting of elements is an expensive operation in a programming language. Expensive in the sense that it consumes much of CPU time.

To understand this, consider the following example :

Given array is :

0	1	2	3	4	5	6
22	33	55	77	88	99	100

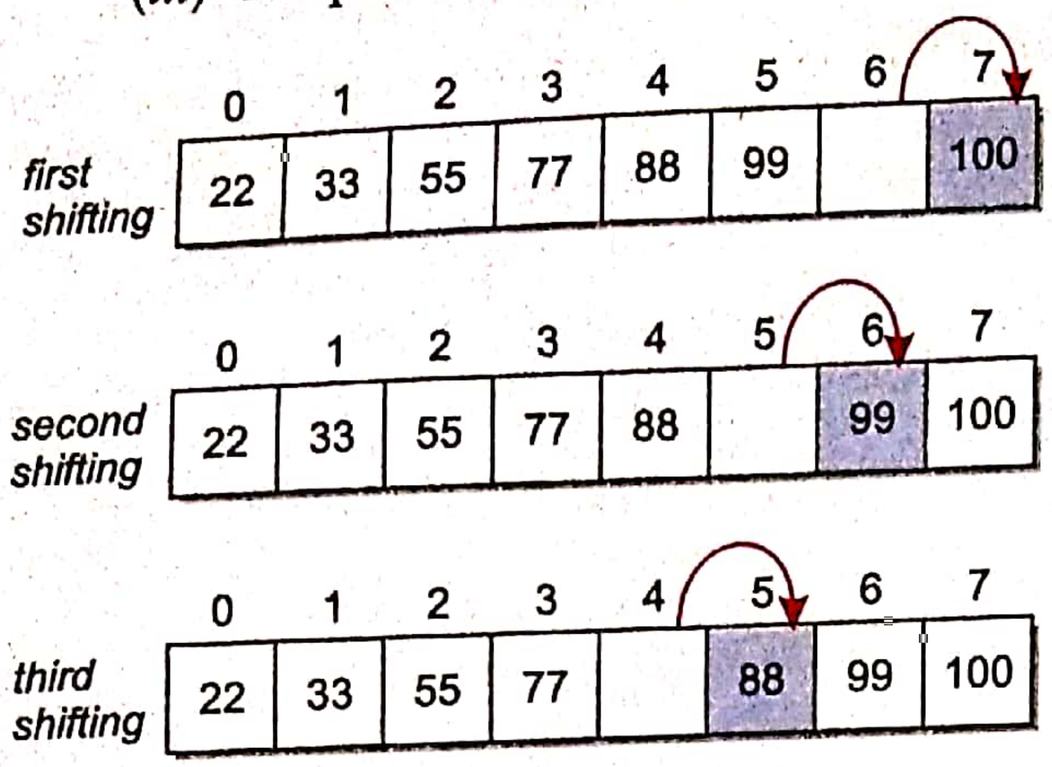
Element to be inserted : 44

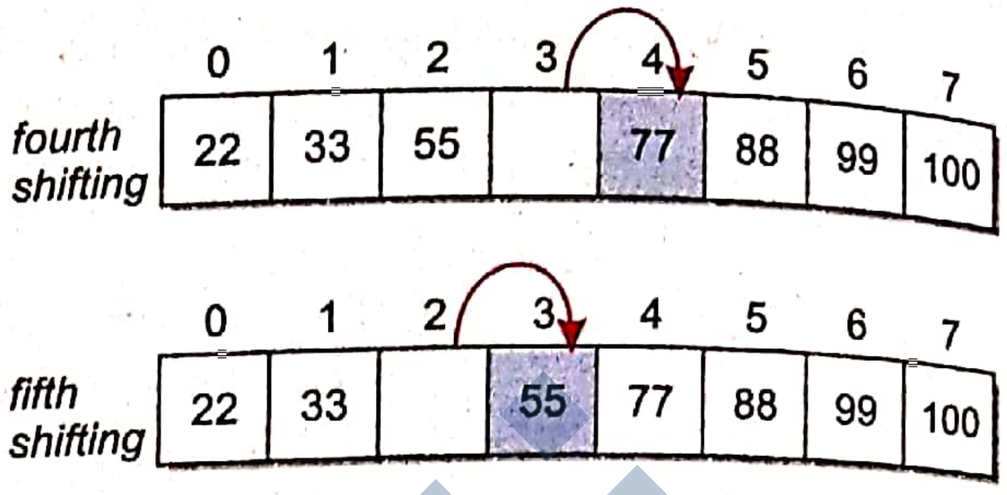
(*i*) Determine the appropriate position for new element : it is **index 2**, after element 33

(*ii*) Create space in the end of array for one element.

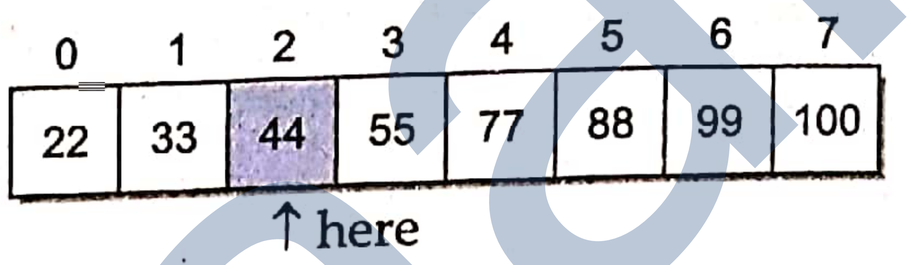
0	1	2	3	4	5	6	7
22	33	55	77	88	99	100	

(*iii*) Keep shifting elements from the right end till index 2 *i.e.*,





(*iv*) Now insert the new element :



As you can make out that insertion of a new element requires the shifting of elements to make room for new element.

Shifting of elements is an expensive process and should be avoided if a better algorithm is available. Python makes available a better algorithm called **bisect,** available in **bisect** module. So, if you use following statement after importing bisect module :

```
bisect.insort(list, <newelement>)
```

The **insort( )** function of bisect module inserts an item in the sorted sequence, keeping it sorted. The above function offers a better algorithm to insert in a sorted sequence.

But one thing that you must remember is that the **bisect** module works on a sequence arranged in **ascending order only**. The **bisect** module also offers another function **bisect( )** that returns the appropriate *index* where the new item can be inserted to keep the order maintained, *e.g.,*

```
bisect.bisect(list, <element>)
```

will give you the index where the new element should be inserted.

In the following lines we are giving two programs for insertion of an element — **program 9.3** that uses traditional algorithm given above by finding position and shifting elements and **program 9.4** that uses efficient **bisect** algorithm of Python **bisect** module.

Program 9.3 Inserting an element in a sorted array using traditional algorithm.

```
def FindPos (AR, item) :
    size = len(AR)
    if item < AR[0] :
        return 0
    else :
        pos = -1
```

```
for i in range(size –1) :
    if (AR[i] <= item and item < AR[i + 1] ) :
        pos = i + 1
        break
if (pos == –1 and  i <= size –1 ):
    pos = size
return pos

def Shift( AR, pos ):
    AR.append(None)                    # add an empty element at the end
    size = len (AR)

    i = size –1
    while i >= pos :
        AR[i] = AR[i-1]
        i = i-1
# __ main__
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
ITEM = int(input("Enter new element to be inserted :"))
position = FindPos( myList, ITEM)
Shift (myList, position)
myList[position] = ITEM
print("The list after inserting", ITEM, "is")
print(myList)
```

The output produced by above program is :

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter new element to be inserted : 80
The list after inserting 80 is
[10, 20, 30, 40, 50, 60, 70, 80]
```

Following program uses **bisect** module of Python for insertion of new element in a sorted array.

Program 9.4 Insertion in sorted array using bisect module.

```
import bisect
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
```

```
ITEM = int(input("Enter new element to be inserted :"))

ind = bisect.bisect( myList, ITEM )
bisect.insort (myList, ITEM )

print(ITEM, "inserted at index", ind)
print("The list after inserting new element is")
print(myList)
```

*The bisect( ) function would return the correct index value for **ITEM**. And the insort( ) would insert the **ITEM** in the list **myList** maintaining the order of the elements.*

The output produced by above program as shown on the right.

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter new element to be inserted : 45
45 inserted at index 4
The list after inserting new element is
[10, 20, 30, 40, 45, 50, 60, 70]
```

IMPORTANT

But one thing, you must be aware of, which is that the **bisect** module's functions work with sequences arranged in **ascending order** of their elements. In order to insert an element in a descending order sequence, you may tweak it as listed below :

- firstly reverse the list as follows to change it to *ascending order* list :

  `<list>.reverse( )`

- Add the new element using bisect module as explained above.
- Reverse the list again so that it is back to descending order.

```
>>> l2 = [80, 70, 60, 50, 40, 30, 20, 10]
>>> l2.reverse()
>>> l2
[10, 20, 30, 40, 50, 60, 70, 80]
>>> bisect.insort(l2, 45)
>>> l2.reverse()
>>> l2
[80, 70, 60, 50, 45, 40, 30, 20, 10]
>>>
```

9.6.3 Deletion of an Element from a Sorted Linear List

The element to be deleted is first searched for in the array using one of the search techniques *i.e.*, either *linear search* or *binary search*. If the search is successful, the element is removed and rest of the elements are shifted so as to keep the order of array undisturbed. Let us consider element 22 is to be deleted from array X shown in Fig. 9.6(*a*).

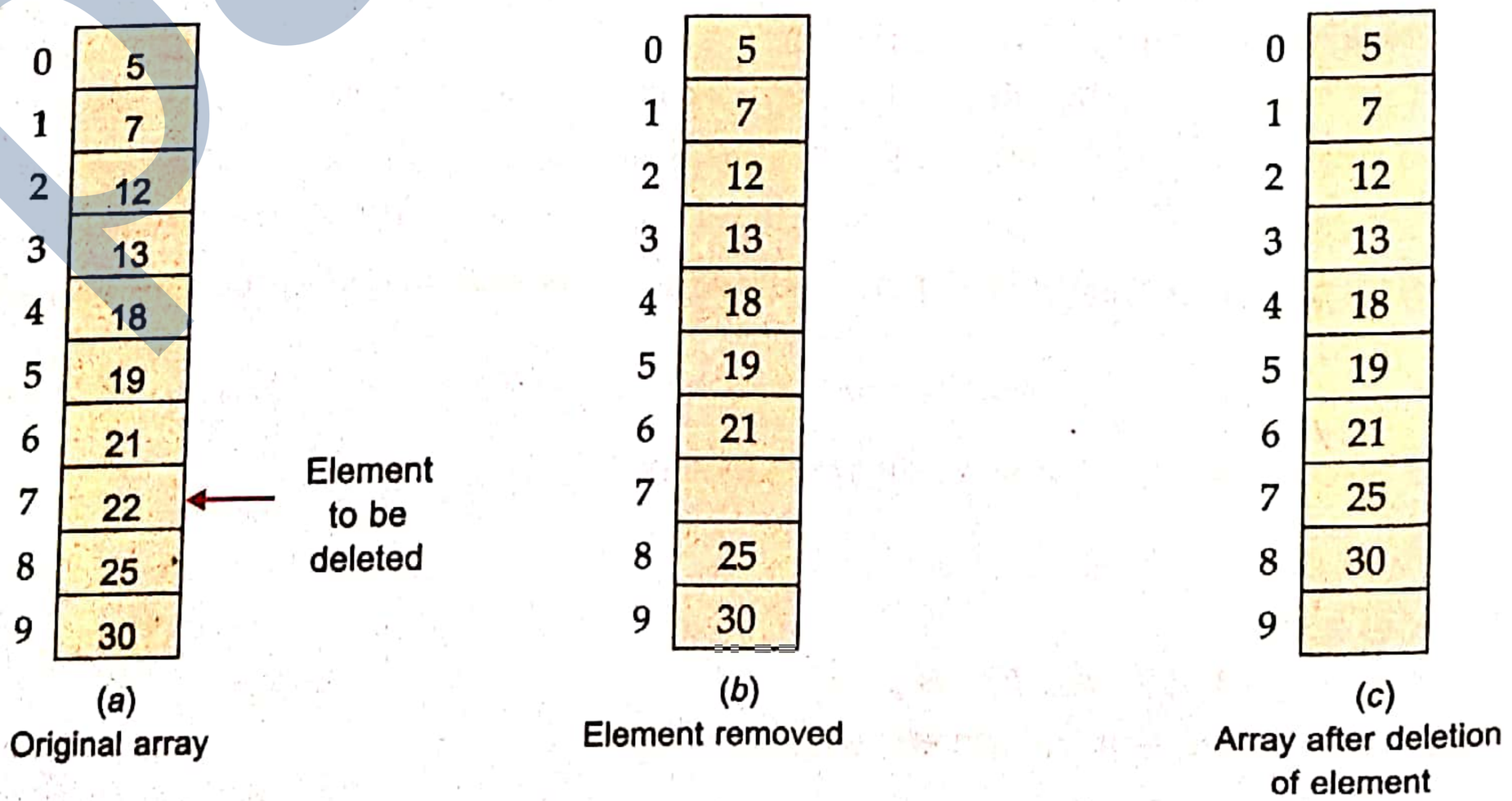


Figure 9.6 Deletion of an element in array *X*.

If the elements are shifted downwards or towards right side, then unused elements (free spaces) are available in the beginning of the fixed-size array otherwise free space is available at the end of the array. However, if you are implementing arrays through lists, then there would not be any such issue as Python lists are very flexible and can grow/shrink easily.

Algorithm *Deletion in Linear List*

Considering that *ITEM*'s search in array AR is successful at location pos

Case I *Shifting upwards (or in left side)*

```
# Initialise counter
1.   ctr = pos
2.   while ctr < U perform steps 3 and 4
3.        AR[ctr] = AR[ctr + 1]
4.        ctr = ctr + 1
# End of while
```

Case II *Shifting downwards (or in right side)*

```
1.   ctr = pos
2.   while ctr > 1 perform steps 3 and 4
3.        AR[ctr] = AR[ctr – 1]
4.        ctr = ctr – 1
# End of while
```

Python provides a remove( ) method that itself takes care of shifting of elements. You have worked with **remove( )** method and **del<item>** statements of lists in class XI. The same method/statement you can use for removing an element from a sorted array/list.

Following program deletes an element from a sorted array.

Program 9.5 Deletion of an element from a sorted linear list.

```
def Bsearch( AR, ITEM) :
    beg = 0
    last = len(AR) - 1
    while (beg <= last) :
        mid = (beg + last)/2
        if(ITEM == AR[mid]) :
            return mid
        elif (ITEM > AR[mid] ) :
            beg = mid + 1
        else :
            last = mid - 1
    else :                          # else of loop
        return False                # when ITEM not found
```

```
# __main__
myList = [10, 20, 30, 40, 50, 60, 70]
print("The list in sorted order is")
print(myList)
ITEM = int(input("Enter element to be deleted :"))
position = Bsearch(myList, ITEM)
if position :
    del myList[position]
    print("The list after deleting", ITEM, "is")
    print(myList)
else :
    print("SORRY! No such element in the list")
```

Determining the position of element to be deleted

Statement deleting the element at index position

The output produced by above code is :

```
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter element to be deleted : 45
SORRY! No such element in the list
>>> ========== RESTART ==========
The list in sorted order is
[10, 20, 30, 40, 50, 60, 70]
Enter element to be deleted : 40
The list after deleting 40 is
[10, 20, 30, 50, 60, 70]
```

The above program firstly reads an array and then asks for the element to be deleted. It then searches for the element's position in the array, using Binary search. And if found, then the element at determined position is deleted.

9.6.4 Traversal of a Linear List

Traversal in one-dimensional arrays involves processing of all elements (*i.e.*, from very first to last element) one by one. For instance, traversal of array shown in Fig. 9.6(*a*) would process in the following order :

X[0], X[1], X[2], X[3].....X[10]

Algorithm

Traversal in Linear List

```
# Considering that all elements have to be printed
1.  ctr = L                              # Initialising the counter
2.  Repeat steps 3 through 4 until ctr > U
3.      print(AR[ctr])
4.      ctr = ctr + 1
    # End of Repeat
5.  END.
```

Program 9.6 Traversing a linear list.

```
def traverse(AR) :
    size = len(AR)
    for i in range(size) :
        print(AR[i], end = ' ')
# __main__
size = int (input("Enter the size of Linear list to be input :"))
AR = [None] * size                    # create an empty list of the given size
print("Enter elements for the Linear List")
for i in range(size) :
    AR[i] = int (input("Element" + str(i) + ":"))
print("Traversing the list :")
traverse(AR)
```

Loop traversing the elements in the linear list

traversal function invoked

```
Enter the size of Linear list to be input : 6
Enter elements for the Linear List
Element 0 : 12
Element 1 : 23
Element 2 : 34
Element 3 : 45
Element 4 : 56
Element 5 : 67
Traversing the list :
12 23 34 45 56 67
```

PriP SEARCHING, INSERTION, DELETION IN THE LIST — Progress In Python 9.1

This PriP session aims at practice of these concepts : searching, insertion and deletion of elements in an array.

⋮

>>>❖<<<

9.6.5 Sorting a Linear List

Sorting refers to arranging elements of a list in ascending or descending order. There are many sorting algorithms that you can apply to sort elements of a linear list. You have learnt about two sorting algorithms, *bubble sort* and *insertion sort*, in your previous class. You can choose any of these algorithms to sort your linear list.

We are not giving below any program to sort a linear list as you already know this. In fact, we expect you to add a sorting function to following program (create a need) that implements a linear list data structure. Before we do that, let us talk about an efficient way of creating lists– **List comprehensions**. Data structures always demand efficient implementation methods, thus, it is important for you to know **List comprehensions** – an efficient way of creating lists.

9.6.6 List Comprehensions

You know already how to create lists. You have been creating lists either by directly assigning comma separated values to a name or by employing a for loop to do this, *e.g.*,

```
lst = [2, 4, 6, 8, 10]
```

Or

```
lst = [ ]
for i in range(1,6) :
    lst.append(i * 2) # lst is now = [ 2, 4, 6, 8, 10]
```

Both above codes create the lists with the same elements.

There is another way of creating lists – in fact, a concise way called **list comprehensions.**

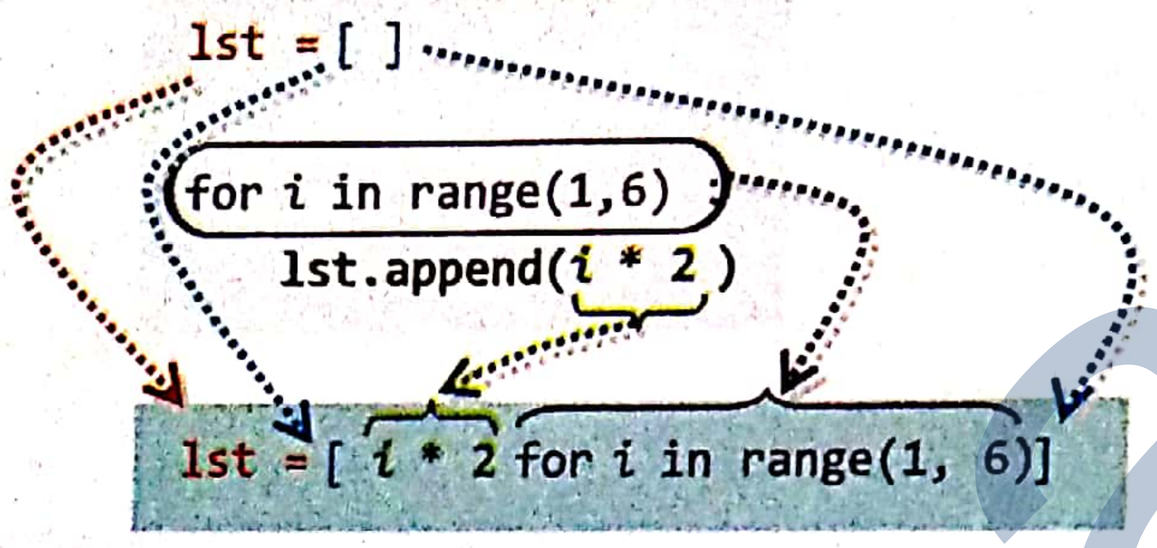
A list comprehension is a concise description of a list that shorthands *the list creating for loop* in the form of a single statement.

Let us see how it works. The above list creating *for loop* can be written as :

```
lst2 = [ i * 2   for i in range(1, 6) ]          # it is a list comprehension
```

Now if you print ***lst2,*** it will also be containing the same elements *i.e.,* [2, 4, 6, 8, 10]

Carefully look at following figure that illustrates how list comprehension is created from above loop :



NOTE

A list comprehension is short-hand for a loop that creates a list.

A list comprehension is usually shorter, more readable, and more efficient.

The above conversion of for loop to list comprehension can be summarized as :

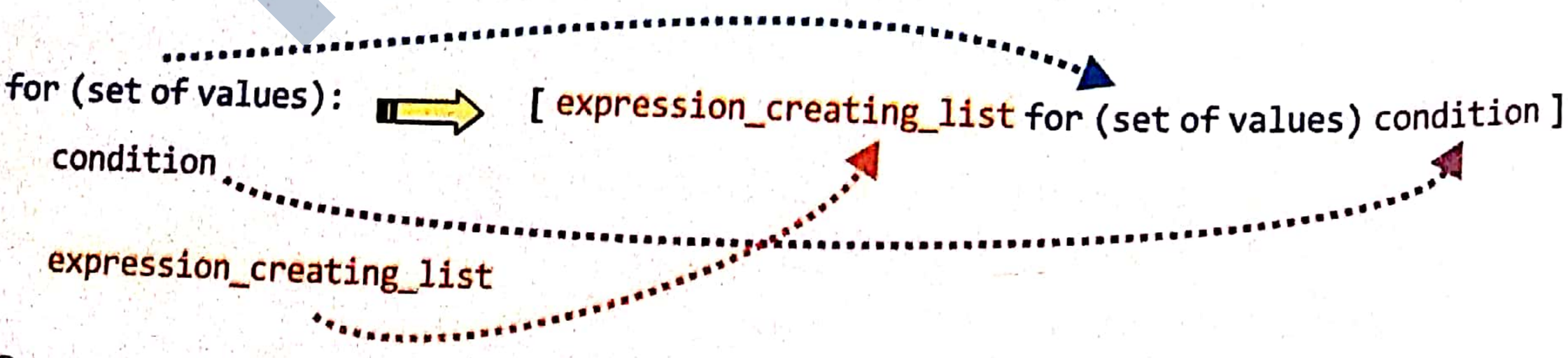
Now let us consider another for loop (more detailed) that is creating a list based on a condition.

```
lst3 = []
for num in range(1, 50):
    if num % 7 == 0 :
        lst3.append(num)
```

The for loop
condition
Expression

As you can see that above loop carries out expression_creating_list only if the given condition is true. The list produced by above code is **[7, 14, 21, 28, 35, 42, 49].**

To create a list comprehension from above for loop, you need to use following conversion rule :



So, our list comprehension for above for loop will be :

```
lst3 = [num for num in range(1,50) if num % 7 == 0]
```

Please note that colon (:) is not required with ***if*** *and* ***else*** *keywords in List Comprehension*

Consider another list comprehension.

```
A = [num if num < 5 else num*2 for num in range(2, 9)]
```

The above list comprehension will produce list A as

```
[ 2, 3, 4,  10, 12, 14, 16 ]
```

Values < 5 are stored as it is — *Values > 5 are stored through expression num*2*

For more clarity, you can enclose entire if else part in parenthesis, *e.g.*,

```
A = [(num if num < 5 else num * 2) for num in range (2, 9)]
```

Notice again that with **if** and **else** we do not give colon in list comprehensions otherwise Python gives error, *e.g.*, following code will give an error.

```
A = [num if num < 5 :    else : num * 2 for num in range(2, 9)]
```

Not to be given in list comprehensions

LIST COMPREHENSION

A **List Comprehension** is a concise description of a list that shorthands *the list creating for loop* in the form of a single statement.

Consider some more examples given below :

Example 9.1 *Create a list myList with these elements (i) using for loop, (ii) using list comprehension.*

Solution *(i)*

```
myList = []
for i in range(11):
    myList.append( 2 ** i)
```

(ii)

```
myList = [2 ** i for i in range(11)]
```

Example 9.2 *Given an input list* ***Vals*** *below, produce a list namely* ***Mul3****, using a list comprehension having the numbers from* ***Vals*** *that are multiples of 3.*

```
Vals = [31, 15, 42, 12, 5, 39, 21, 61, 25]
```

Solution

```
Mul3 = [num for num in Vals if num % 3 == 0]
```

The list, *Mul3*, will store **[15, 42, 12, 39, 21]**

Example 9.3 *Consider the code below. What will the list NL be storing ?*

```
Lst = [('a', 11), ('b', 12), ('c', 13)]
NL = [ n * 3 for (x, n) in Lst if x == 'b' or x == 'c']
```

Solution

```
[36, 39]
```

Example 9.4 *Consider the following code. What will the list* ***Res*** *be storing ?*

```
Res = ["Ev" if i % 2 == 0 else "Od" for i in range(10,20)]
print(Res)
```

Solution

```
['Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od', 'Ev', 'Od']
```

You can also form a list comprehension for a nested for loop.

```
for loop1:          ⇒   [expression_creating_list for loop1 for loop2 ]
    for loop2
        expression_creating_list
```

For example, consider the following nested loop :

```
result= []
for x in [10, 5, 2]:
    for y in [2, 3, 4]:
        result.append( x ** y)
print(result)
```

It produced result as : [100, 1000, 10000, 25, 125, 625, 4, 8, 16]

The list comprehension for above nested loop will be like :

```
result = [ x ** y for x in [10, 5, 2] for y in [2, 3, 4]]
```

So the equivalent code for above code will be :

```
result= []
result = [ x ** y for x in [10, 5, 2] for y in [2, 3, 4]]
print(result)
```

NOTE

Please note that list comprehensions work with square brackets only. Try using it with ( ) and Python will give you an error. An assignment question is based on this, check *Type B*.

And the result will just be the same.

The nested for loops may have optional condition(s) and the list comprehension will include optional condition just in the same way as you have included conditions in single for loop.

For instance, look at the following list comprehension :

```
[(x,y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
```

Can you write its equivalent nested for loop ? You surely can, I bet ! Please check if mine (given below) is correct ? ☺

```
L1 = []
for x in range(5) :
    if x % 2 == 0 :
        for y in range(5) :
            if y % 2 == 1 :
                L1.append((x, y))
print(L1)
```

NOTE

List comprehensions are although useful but you should not use them when you need to check multiple conditions.

Advantages of List Comprehensions

List comprehensions are considered more Pythonic as they truly represent Python coding style and also offer these advantages :

(*i*) **Code reduction.** A code of 3 or more lines (for loop with or without a condition) gets reduced to a single line of code.

(*ii*) **Faster code processing.** List comprehensions are executed faster than their equivalent for loops for these *two* reasons :

(*a*) Python will allocate the list's memory first, before adding the elements to it, instead of having to resize on runtime.

(*b*) Also, calls *to append( )* function get avoided, reducing function overhead time (*i.e.*, additional time taken to call and return from a function) which may be cheap but add up.

Program 9.5

Linear List Implementation. A charity organization conducts camps in various locations of a city. Each camp is located at a location on a specific date. The organization maintains a monthly list of camps planned in this format :

03 New Cly, 18 T Nagar, 25 K Pura

Where first two digits of every entry signify the date on which the camp is to be conducted. Write a program to implement this along with following points :

(*i*) A linear list, **planned**, stores the camps that are to be conducted. A camp's details are added to **planned** list once NOC is obtained.

(*ii*) As soon as a camp is conducted, its details are moved to list **conducted** and removed from **planned** list.

(*iii*) Each camp servers some people.

(*iv*) The program should be able to provide options for adding to planned list, getting conducted camp's details, searching for a camp, a report of how many camps were conducted so far and how many people were served and display the linear lists **planned** and **conducted**.

```
def addloc(cmp):
    dd = cmp[0:2]
    ln = len(planned)
    if ln == 0:
        planned.append(cmp)
    else:
        last = planned[ln-1]
        if int(dd) >= int(last[0:2]):
            planned.append(cmp)
        else:
            for i in range(ln):
                cp = planned[i]
                if  int(dd) <= int(cp[0:2]):
                    planned.insert(i,cmp)
                    break
```

```
def conductCamp(cmp):
    conducted.append(cmp)
    planned.remove(cmp)
def search(cmp, lst):                    # linear search technique
    ln = len(lst)
    for i in range(ln):
        if cmp in lst[i]:
            return lst[i]
    else:
        return False
def Report() :
    lenp = len(planned)
    lenc = len(conducted)
    print("\t R E P O R T")
    print("————————————")
    print("Camps conducted so far:", lenc)
    print("People served so far", ppl)
    print("Camps to be conducted:", lenp)
    print("————————————")
def display():
    print("\nCamps Planned : ", end = ' ')
    for i in planned:
        print(i, end=', ')
    print("...!!")
    print("\nCamps Conducted so far : ", end = ' ')
    for i in conducted:
        print(i, end=', ')
    print("...!!")
#__main__
planned = []
conducted = []
ppl = 0
ch = 0
while (ch != 6):
    print("\t——")
    print("\tMENU")
    print("\t——")
    print("1. Add Camp Location")
    print("2. Camp Conducted")
    print("3. Look for a Camp")
    print("4. Report")
    print("5. Display List")
    print("6. Exit")
    ch = int(input("Enter your choice (1-6):"))
```

```
if ch == 1 :
    cm = input("Enter Camp location : ")
    dd = input("Enter date of the month (only dd) : ")
    cmp = dd + cm
    addloc(cmp)
elif ch == 2 :
    cm = input("Camp conducted at location?")
    p = int(input("How many people are served at this camp?"))
    ppl = ppl + p
    result = search(cm, planned)
    if result == False :
        print("Sorry no such camp in the list")
    else :
        conductCamp(result)
elif ch == 3:
    cm = input("Enter camp location : ")
    r1 = search(cm, planned)   # result1
    if r1 == False :
        r2 = search(cm, conducted) # result2
        if r2 == False:
            print("Sorry no such camp in our list")
        else:
            dd = r2[0:2]
            print(cm, "was conducted on date", dd, "of this month")
    else :
        dd = r1[0:2]
        print(cm, "camp is to be conducted on date", dd, "of this month")
elif ch == 4 :
    Report()
elif ch == 5:
    display()
elif ch != 6 :
    print("Wrong choice! Enter choice from 1 to 6 only")
else:
    print("THANK YOU")
```

* Please note above program is based on certain assumptions such that :

(*i*) **dd** part is always 2 digits long *i.e.*, enter single digit dates preceeded with a zero *e.g.*, 03, 06 etc.

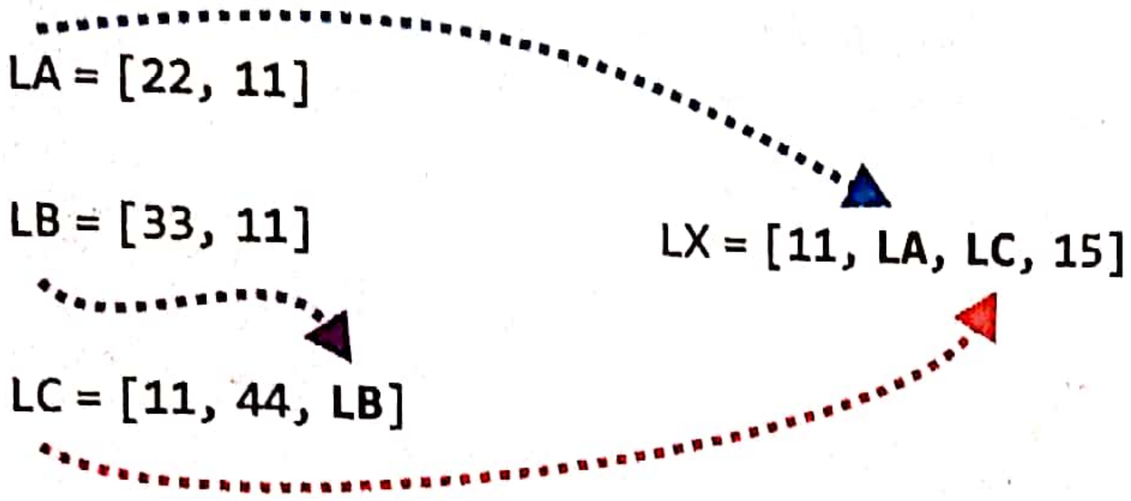
(*ii*) there is no check to ensure that location name is not repeated. If you want, you can add this functionality.

Sample run of above program is as shown below :

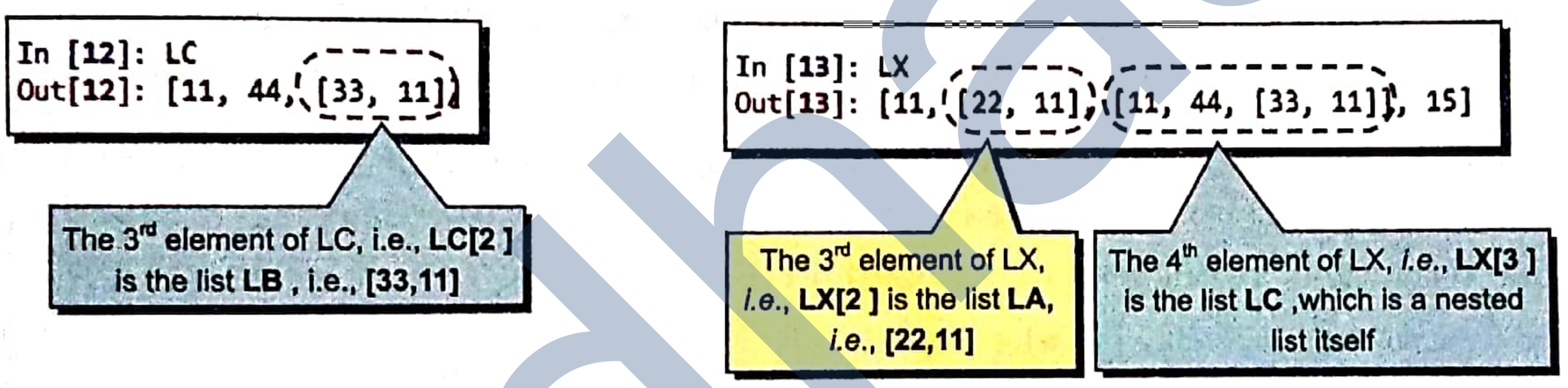
```
        MENU
1. Add Camp Location
2. Camp Conducted
3. Look for a Camp
4. Report
5. Display List
6. Exit
Enter your choice (1-6): 1
Enter Camp location : K Pur
Enter date of the month (only dd) : 09

        MENU

:
Enter your choice (1-6): 1
Enter Camp location : D Pur
Enter date of the month (only dd) : 27

        MENU

:
Enter your choice (1-6): 1
Enter Camp location : B Nagar
Enter date of the month (only dd) : 03

        MENU

:
Enter your choice (1-6): 1
Enter Camp location : Y Vihar
Enter date of the month (only dd) : 15

        MENU

:
Enter your choice (1-6): 5
Camps Planned : 03 B Nagar, 09 K Pur, 15 Y
Vihar, 27 D Pur, ...!!
Camps Conducted so far : ...!!

        MENU

:
Enter your choice (1-6): 3
Enter camp location : D Pur
D Pur camp is to be conducted on date 27
of this month

        MENU

:
Enter your choice (1-6): 4
        R E P O R T
-------------------------------
Camps conducted so far: 0
People served so far 0
Camps to be conducted: 4
-------------------------------

        MENU

:
Enter your choice (1-6): 2
Camp conducted at location? B Nagar
How many people are served at this camp? 320

        MENU

:
Enter your choice (1-6): 2
Camp conducted at location? K Pur
How many people are served at this camp? 412

        MENU

:
Enter your choice (1-6): 4
        R E P O R T
-------------------------------
Camps conducted so far: 2
People served so far 732
Camps to be conducted: 2
-------------------------------

        MENU

:
Enter your choice (1-6):6
THANK YOU
```

9.7 NESTED/TWO DIMENSIONAL LISTS IN PYTHON

You know that lists are objects that can hold objects of any other types as its elements. Since *lists* are objects themselves, they can also hold other list(s) as its element(s). Carefully go through the code shown below that creates four lists namely LA, LB, LC and LX :



Out of the four lists created above, two are nested lists, *i.e.*, *LC* and *LX* are nested lists as they contain one or more lists as their elements. So when you display or print the contents of lists *LC* and *LX*, it will be like :



So with *LX* as :

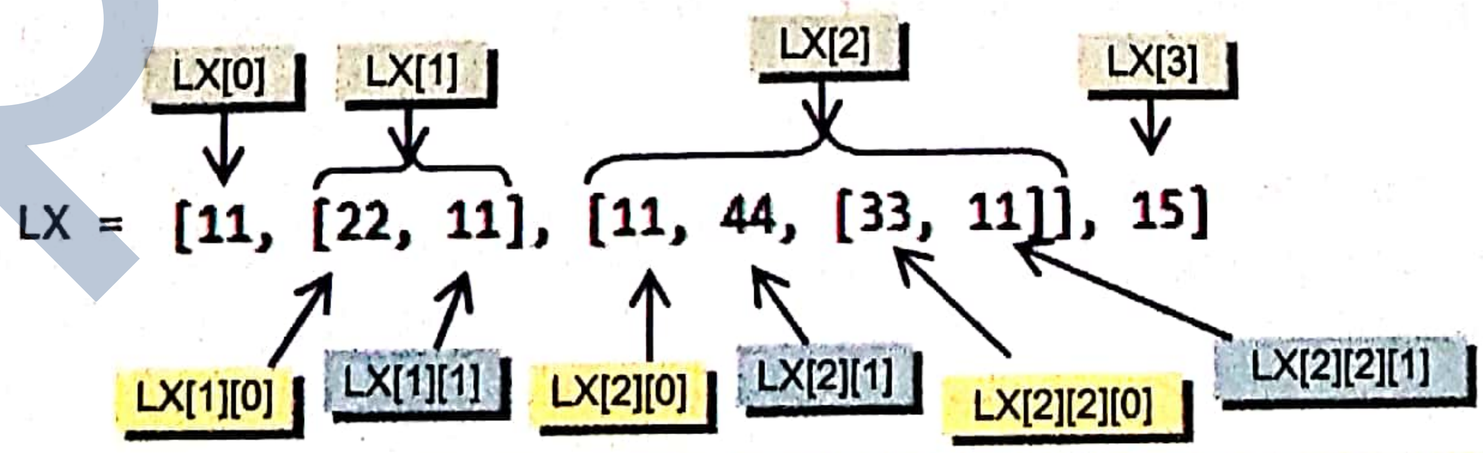
```
LX = [11, [22, 11], [11, 44, [33, 11]], 15]
```

What will be shown if you display :

(*i*) LX[1], (*ii*) LX[2], (*iii*) LX[1][1], (*iv*) LX[2][0], (*v*) LX[2[2], (*vi*) LX[2][2][1] ?

Well, I already knew that you knew it . 😉

So the values displayed are like :



(*i*) [22, 11] (*ii*) [11, 44, [33, 11]] (*iii*) 11 (*iv*) 11 (*v*) [33, 11] (*vi*) 11

So, you can say that a list that has one or more lists as its elements is a nested list. A two dimensional list is also a nested list. Let us see how.

NESTED LIST

A list that has one or more lists as its elements is a **nested list.**

9.7.1 Two Dimensional Lists

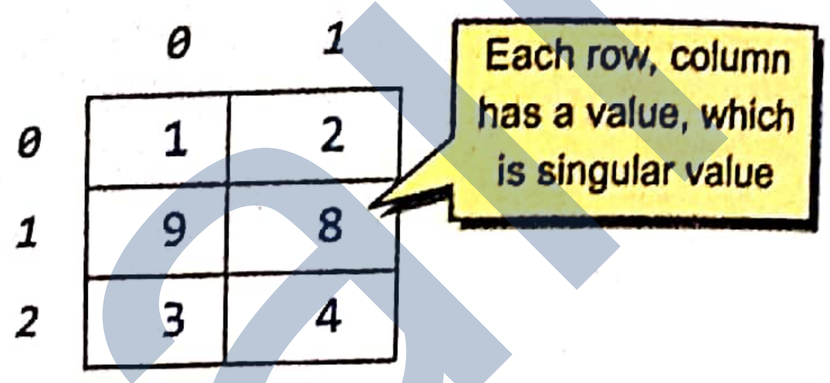
A two dimensional list is a list having all its elements as lists of same shapes, *i.e.*, a two dimensional list is ***a list of lists***, *e.g.*,

```
L1 = [ [1, 2], [9, 8], [3, 4] ]
```

L1 is a two dimensional list as it contains *three lists*, each having same shape, *i.e.*, all are singular lists (*i.e.*, non-nested) with a length of 2 elements. Following representation will make it clearer.

```
L1 = [ [1, 2],
       [9, 8],
       [3, 4] ]
```

You can visualize it as:



Regular two-dimensional lists are the nested lists with these properties :

(*i*) All elements of a 2D list have same shape (*i.e.*, same dimension and length)

(*ii*) The length of a 2D list tells about **Number of Rows** in it (*i.e.*, **len(list)**)

(*iii*) The length of single row gives the **Number of Columns** in it (*i.e.*, **len(list[n])**)

Ragged list. A list that has lists with different shapes as its elements is also a 2d list but it is an irregular 2d list, also known as a *ragged list.*

For instance, following list ( ) is a ragged list :

```
L2 = [ [1, 2, 3], [5, 6] ]
```

as its one element has a length as 3 while its second element is a list with a length of 2 elements.

Let us quickly learn how you can use a two dimensional list. Please note that in the following examples, we shall mostly be using regular 2D lists unless specified explicitly.

REGULAR TWO DIMENSIONAL LIST

A **regular two dimensional list** is a list having lists as its elements and each element-list has the same shape i.e., same number of elements (length).

9.7.1A Creating a 2D List

To create a 2D list by inputting element by element, you can employ **a nested loop** as shown below: one loop for rows and the other loop for reading individual elements of each row.

```
Lst = []
r = int(input("How many rows?"))
c = int(input("How many columns?"))
for i in range(r) :
    row = []
    for j in range(c) :
        elem = int(input("Element"+ str(i) +", "+str(j)+ ": "))
        row.append(elem)
    Lst.append(row)
print("List created is :", Lst)
```

Each row is initialized as an empty list and then elements are added to this row list

Integers being added to row list

row list is added as an element of 2d list

The sample run of above program is given below :

```
How many rows? 3
How many columns? 3
Element 0, 0: 2
Element 0, 1: 3
Element 0, 2: 4
Element 1, 0: 22
Element 1, 1: 32
Element 1, 2: 43
Element 2, 0: 50
Element 2, 1: 60
Element 2, 2: 70
List created is : [[2, 3, 4], [22, 32, 43], [50, 60, 70]]
```

As you can see that above code has successfully created a regular 2D list having a size of 3×3.

9.7.1B Traversing a 2D List

To traverse a 2D list element by element, you can employ a nested loop as earlier : one loop for rows and another for traversing individual elements of each row (see below).

```
# We have added following nested loop to previous code of 2d list creation
print("List created is :")
print("Lst = [ ")
for i in range(r) :
    print("\t[", end = " ")
    for j in range(c) :
        print( Lst[i][j], end = "  " )
    print("]")
print("\t]")
```

The output produced is like :

```
List created is :
Lst = [
        [ 2  3  4  ]
        [ 22  32  43 ]
        [ 50  60  70 ]
    ]
```

9.7.1C Accessing/Changing Individual Elements in a 2D List

You can access individual elements in a 2D list by specifying its indexes in square brackets, *e.g.*,

To access 3rd row's 3rd element from the 2D list namely *Lst* that we created in previous sections, you will write :

```
Lst[2][2]
```

Indexes begin with *0* and go till $n-1$.

Using the same syntax, you can also change a specific value in a 2D list as lists are mutable types, *i.e.*, following statement will change the 3rd element of 2nd row to 345 :

```
Lst[1][2] = 345
```

9.7.1D How a Two-dimensional List is Stored

You know that Python variables are not like storage containers rather they store or point to the address where a certain value is located. You have read in class XI that lists also store reference (memory location) of each individual item stored in them.

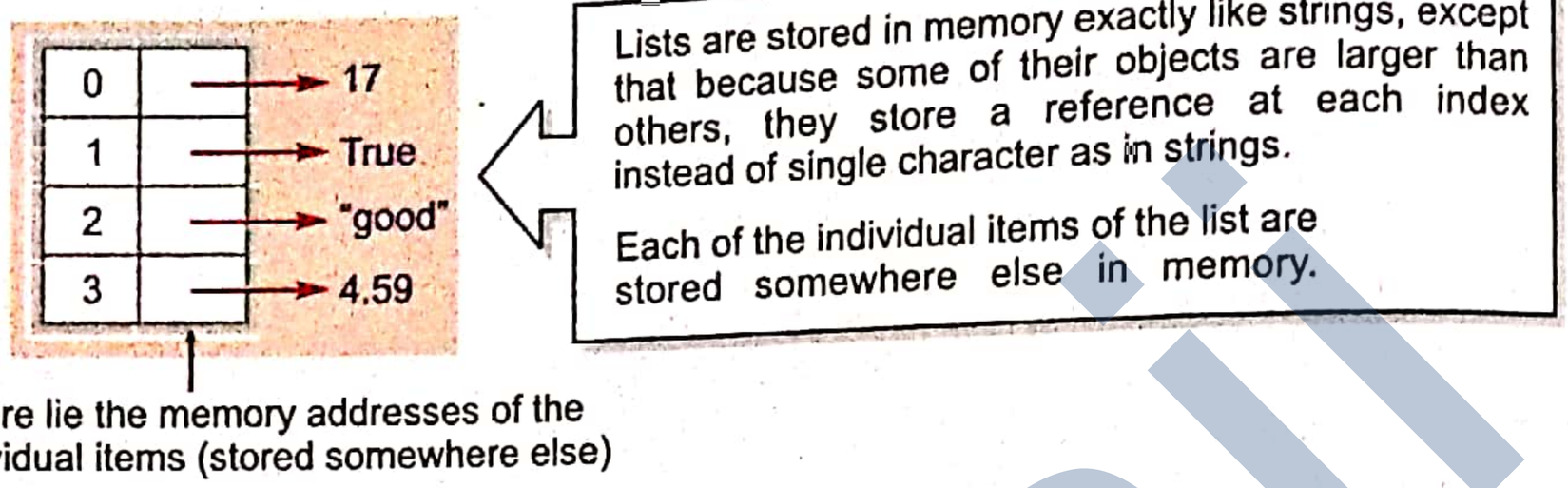


Figure 9.7 How lists are internally stored.

On the same lines, a 2D list stores references of its element lists at its indexes. Following figure 9.8 will make it clearer.

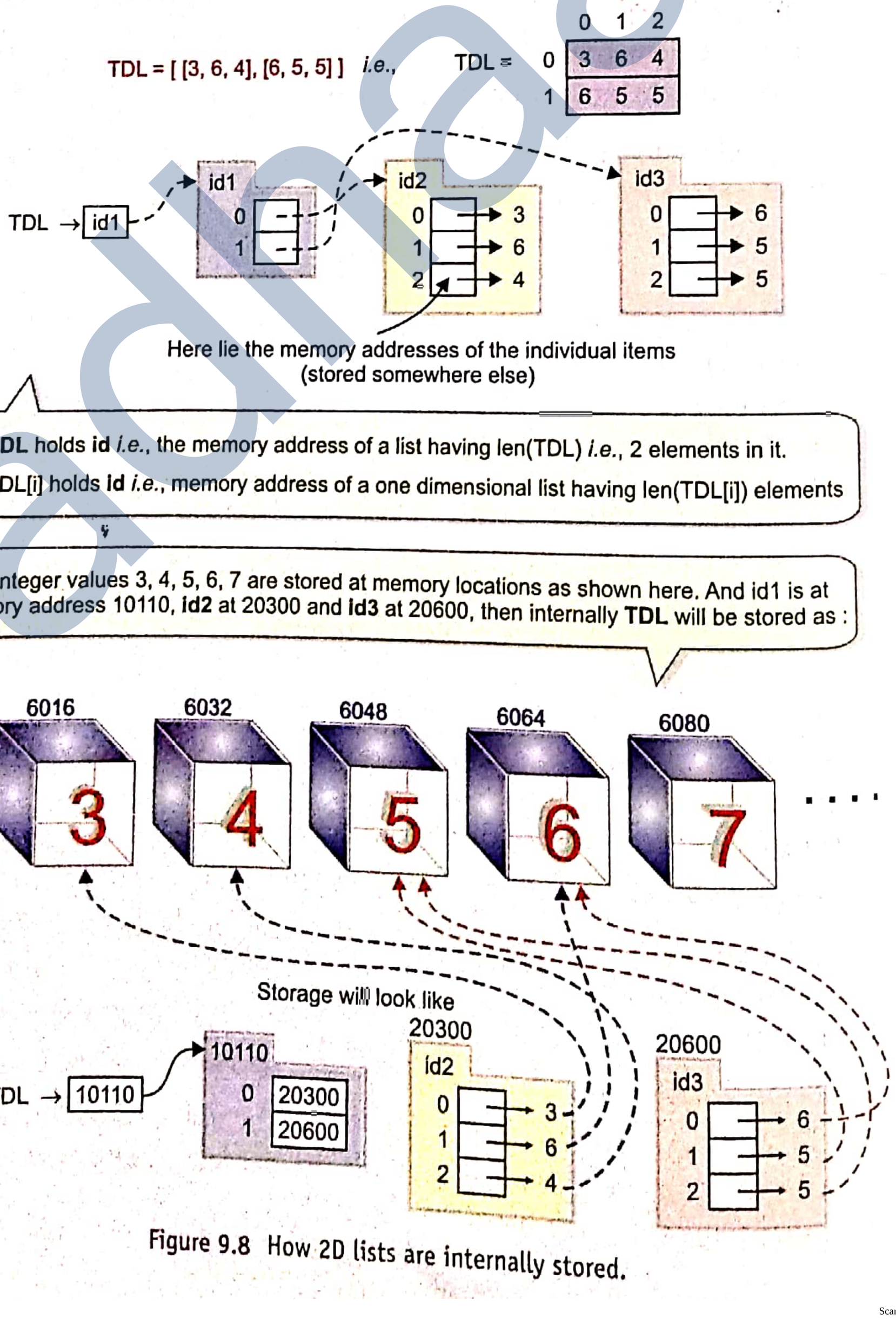


Figure 9.8 How 2D lists are internally stored.

In the same way as regular 2D lists, **ragged lists** are also stored (*see* Fig. 9.9)

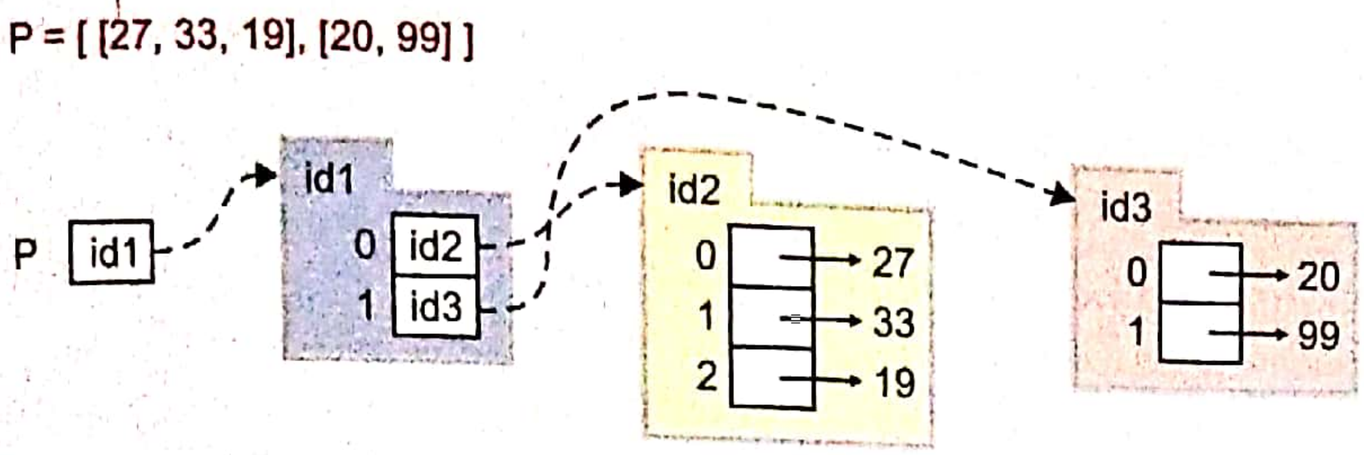


Figure 9.9 Internal Storing of Ragged lists.

NOTE

The regular 2D lists are the lists having lists of same sizes as its elements and that means its rows have equal sizes. On the other hand, the ragged lists are also nested lists with rows of different sizes.

9.7.1E Slices of Two-dimensional Lists

The slicing rules of Python are applicable to 2D lists in the same way as you have applying to lists, tuples and other sequences. In a 2D array, you just need to be always aware that the top level list's elements are lists themselves. Following examples will make it clearer.

```
>>> Lst = [[2, 3, 4], [22, 32, 43], [50, 60, 70], [9, 10, 11]]
>>> Lst[ : 2]
[[2, 3, 4], [22, 32, 43]]
>>> Lst[2: ]
[[50, 60, 70], [9, 10, 11]]
```

Elements of Lst with index < 2

Elements of Lst with index >= 2

Since the slice itself is a list, you can further apply slicing on it :

```
>>> Lst[2: ] [ :1]
[[50, 60, 70]]
```

Following figure (Fig. 9.10) illustrates it.

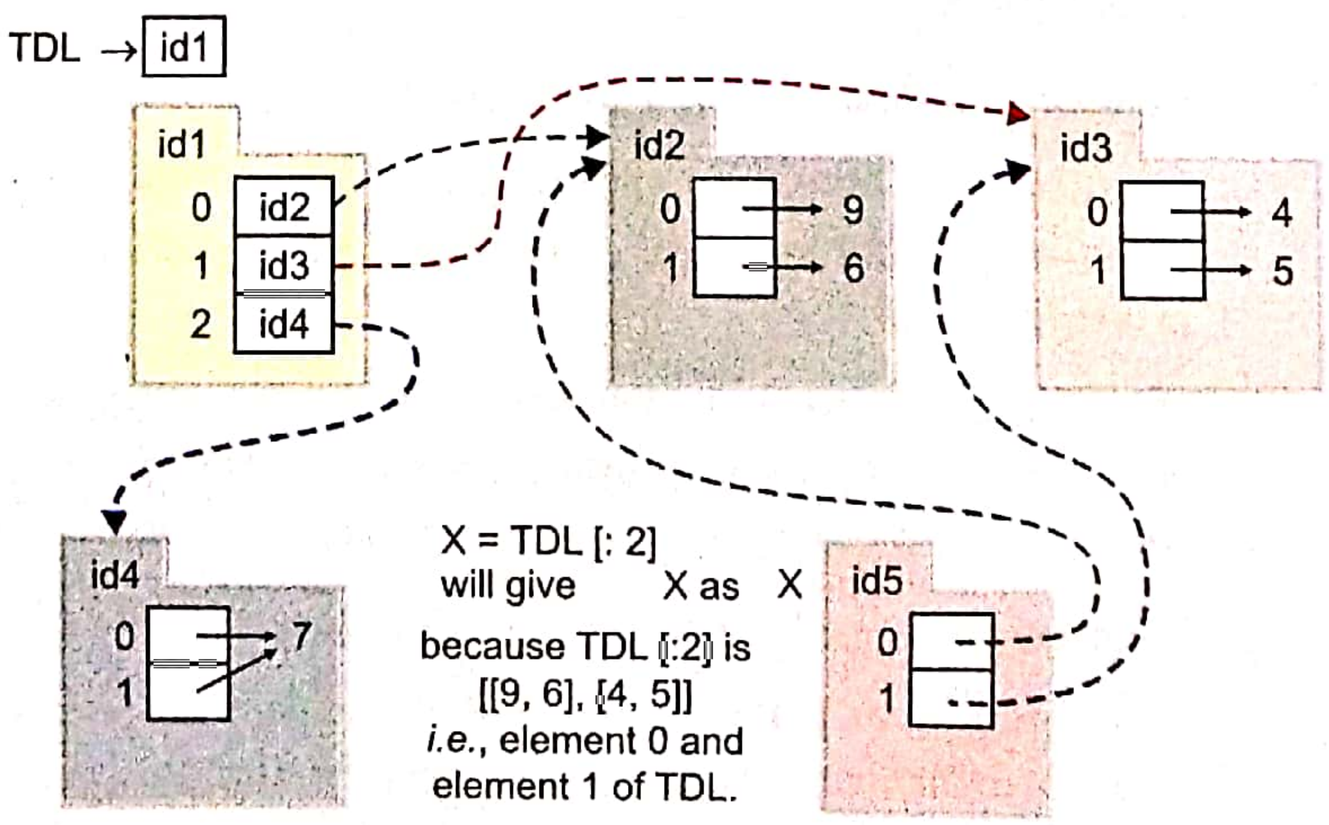


Figure 9.10 Slicing a Multi-dimensional list.

Check Point 9.1

1. What do you mean by the following terms ?
   (*i*) raw data (*ii*) data item
   (*iii*) data type (*iv*) data structure
2. What do you understand by the following :
   (*i*) simple data structures
   (*ii*) compound data structures
   (*iii*) linear data structures
   (*iv*) non-linear data structures ?
3. When would you go for linear search in an array and why ?
4. State condition(s) when binary search is applicable.
5. Name the efficient algorithm offered by Python to insert element in a sorted sequence.
   State condition(s) when binary search is applicable.
6. What is a list comprehension ?
7. What is a 2D list ?
8. What is a nested list ?
9. Is Ragged list a nested list ?

LET US REVISE

- *A data structure is a named group of data of different data types which can be processed as a single unit.*
- *Simple data structures are normally built from primitive data types.*
- *Simple data structures can be combined in various ways to form compound data structures. The compound data structures may be linear (whose elements form a sequence) and non-linear (which are multi-level).*
- *A Linear list or an array refers to a named list of a finite number n of similar data elements whereas a structure refers to a named collection of variables of different data types.*
- *Stacks are LIFO (Last In First Out) lists where insertions and deletions take place only at one end.*
- *Queues are FIFO (First In First Out) lists where insertions take place at " rear" end and deletions take place at the " front" end.*
- *In linear search, each element of the array is compared with the given Item to be searched for, one by one.*
- *A* ***List Comprehension*** *is a concise description of a list that shorthands the list creating for loop in the form of a single statement.*
- *A list that has one or more lists as its elements is a nested list.*
- *A regular two dimensional list is a list having lists as its elements and each element-list has the same shape i.e., same number of elements (length).*
- *A list that has lists with different shapes as its elements is also a 2d list but it is an irregular 2d list, also known as a ragged list.*
- *Internally a 2D list stores references of its element lists at its indexes.*

Solved Problems

1. *What is a Data Structure ?*

Solution. A data structure is a logical way of organizing data that makes them efficient to use. Different data structures are suited to different types of applications, and some are highly specialized to specific tasks. For instance, stacks are best suited for reversal applications or recursive applications.

2. *Compare a data type with a Data structure.*

Solution. A Data type defines a set of values along with well-defined operations stating its input-output behavior *e.g.*, you cannot put decimal point in an integer or two strings cannot not be multiplied etc.

On the other hand, a Data structure is a physical implementation that clearly defines way of storing, accessing, manipulating data stored in a data structure. The data stored in a data structure has a specific data type *e.g.*, in a stack, all insertions and deletions take place at one end only.

3. *What are linear and nonlinear data Structures ?*

Solution. **Linear Data structure.** A data structure is said to be linear if its elements form a sequence or a linear list, *e.g.*, Arrays or linear lists, Stacks and Queues etc.

Non-Linear Data structure. A data structure is said to be non-linear if traversal of nodes is nonlinear in nature, *e.g.*, Graphs, Trees etc.

4. *In general, what common operations are performed on different Data Structures ?*

Solution. Some commonly performed operations on data structures are :

(*i*) **Insertion.** To add a new data item in the given collection of data items.

(*ii*) **Deletion.** To delete an existing data item from the given collection of data items.

(*iii*) **raversal.** To access each data item exactly once so that it can be processed.

(*iv*) **Searching.** To find out the location of the data item if it exists in the given collection of data items.

(*v*) **Sorting.** To arrange the data items in some order *i.e.*, in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

5. *What purpose Linear lists data structures are mostly used for ?*

Solution. Linear lists data structures are used to store and process elements that are similar in type and are to be processed in the same way. For example, to maintain a shopping list, a linear list may be used where items to be shopped are inserted to it one by one and as soon as an item is shopped, it is removed from the list.

6. *Consider the following similar codes (carefully go through these) and predict their outputs.*

(*i*)
```
NList = [ 60, 32, 13, 'hello' ]
print(NList[1], NList[-2])
NList.append( 15 )
print( len(NList) )
print( len(NList[3]) )
NList.pop(3)
NList.sort()
NList.insert(2, [14, 15] )
NList[3] += NList[4]
NList[3] += NList[2][1]
print(NList[3])
NList.pop()
NList[2].remove(14)
print(NList)
```

(*ii*)
```
NList = [ 60, 32, 13, 'hello' ]
print(NList[1], NList[-2])
NList.append( 15 )
print( len(NList) )
print( len(NList[3]) )
NList.pop(3)
NList.insert(2, [14, 15] )
NList[3] += NList[4]
NList[3] += NList[2][1]
print(NList[3])
NList[2].remove(14)
print(NList)
```

Solution.

(*i*)
```
32 13
5
5
107
[13, 15, [15], 107]
```

(*ii*)
```
32 13
5
5
43
[60, 32, [15], 43, 15]
```

7. *What will be the output produced by following code ?*

```
text = ['h', 'e', 'l', 'l', 'o']
print(text)
vowels = "aeiou"
newText= [ x.upper() for x in text if x not in vowels ]
print(newText)
```

Solution.

```
['h', 'e', 'l', 'l', 'o']
['H', 'L', 'L']
```

8. *What is a list comprehension ? How is it useful ?*

Solution. A List Comprehension is a concise description of a list that shorthands the list creating for loop in the form of a single statement.

List comprehensions make the code compact and are faster to execute.

9. *Predict the output.*

(*i*)
```
LA = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
LB = [num/3 for num in LA if num % 3 == 0]
print(LB)
```

(*ii*)
```
[x + y for x in 'ball' for y in 'boy']
```

(*iii*)
```
li = [1, 2, 3, 4, 5, 6, 7, 8, 9]
k = [elem1*elem2 for elem1 in li if (elem1 - 4) >1 for elem2 in li[:4] ]
print(k)
```

Solution.

(*i*) `[3.0, 12.0, 27.0]`

(*ii*) `['bb', 'bo', 'by', 'ab', 'ao', 'ay', 'lb', 'lo', 'ly', 'lb', 'lo', 'ly']`

(*iii*) `[6, 12, 18, 24, 7, 14, 21, 28, 8, 16, 24, 32, 9, 18, 27, 36]`

10. *What is the difference between a regular 2D list and a ragged List ?*

Solution. A regular two dimensional list is a list having lists as its elements and each element-list has the same shape *i.e.*, same number of elements (length).

On the other hand, a list that contains lists with different shapes as its elements is also a 2D list but it is an irregular 2D list, also known as a ragged list.

For instance, in the example code below *List1* is a regular 2D list while *List2* is a ragged list :

```
List1 = [ [1, 2, 3], [7, 9, 8] ]
List2 = [ [4, 5, 6], [1, 7] ]
```

11. *The Investiture Ceremony is a prestigious event in every school's calendar wherein the school formally entrusts responsibilities on the 'young student leaders'.*

At MySchool, the list STL stores the names of all the students of class XI who have applied for the various School Leaders posts. Out of these, school leaders will be selected.

Students need to register their name through an online application form available on school's website. Ideally the name should be in the form such that First name and Last name have their first letter capitalized and rest of the letters in lowercase.

But not all students are careful when entering their names, so the names can appear with incorrectly capitalized letters. For example :

```
STL = ['Meesha Jain', 'khushi khan', 'Raman Singh', 'yashi Sheril']
```

Write a program that provides functions for

(*i*) selecting only those correctly entered entries where the first letters of the first name and last name are capitalized.

(*ii*) selecting only the incorrectly entered names.

(*iii*) returning a list with corrected names.

Solution.

```
def select_errors(STL):
        newList = []
        for record in STL:
            name_surname = record.split(' ')
            name = name_surname[0]
            surname = name_surname[1]
            if name[0].islower() or surname[0].islower():
                newList.append(record)
        return newList

def select_correct(STL):
        newList = []
        for record in STL:
            name_surname = record.split(' ')
            name = name_surname[0]
            surname = name_surname[1]
            if not name[0].islower() and not surname[0].islower():
                newList.append(record)
        return newList

def correct_entries(STL):
        newList = []
        for record in STL:
            name_surname = record.split(' ')
            name = name_surname[0]
            surname = name_surname[1]
            newList.append(name.capitalize() + ' ' + surname.capitalize())
        return newList

#__main__
STL = []
ch = 0
while (ch != 4):
    print("\t----")
    print("\tMENU")
    print("\t----")
    print("1. Apply for the School Post")
    print("2. List of all applicants")
    print("3. Correct the Incorrect Entries")
    print("4. Exit")
    ch = int(input("Enter your choice (1-4):"))
    if ch == 1 :
        name = input("Enter your name : ")
        STL.append (name)
    elif ch == 2 :
        print("Students applied so far:")
        print(STL)
```

```
elif ch == 3:
    ok_entries = select_correct(STL)
    error_entries = select_errors(STL)
    corrected_entries = correct_entries(STL)
    print("Correctly entered names:", ok_entries)
    print("Incorrectly entered names:", error_entries)
    print("Corrected names :", corrected_entries)
  elif ch != 4 :
      print("valid choices are 1..4:")
else:
      print("THANK YOU")
```

Sample run of above program is :

```
____
MENU
____
1. Apply for the School Post
2. List of all applicants
3. Correct the Incorrect Entries
4. Exit
Enter your choice (1-4): 1
Enter your name : Meesha Jain

____
MENU
____
:
Enter your choice (1-4): 1
Enter your name : khushi khan

____
MENU
____
:
Enter your choice (1-4): 2
Students applied so far:
['Meesha Jain', 'khushi khan', 'Raman Singh', 'yashi Sheril']

____
MENU
____
:
Enter your choice (1-4): 3
Correctly entered names: ['Meesha Jain', 'Raman Singh']
Incorrectly entered names: ['khushi khan', 'yashi Sheril']
Corrected names : ['Meesha Jain', 'Khushi Khan', 'Raman Singh', 'Yashi Sheril']

____
MENU
____
:
Enter your choice (1-4): 1
Enter your name : Raman Singh

____
MENU
____
1. Apply for the School Post
2. List of all applicants
3. Correct the Incorrect Entries
4. Exit
Enter your choice (1-4): 1
Enter your name : yashi Sheril
```

GLOSSARY

Data item	Single unit of values of certain type.
Data structure	Named group of data of some data types.
Linear data structures	Single level data structures representing linear relationship among data.
Linked list	A list of linked elements by means of pointers.
Queue	A FIFO (First In First Out) list.
Simple data structures	Data structures normally built from primitive data types.
Sorting	Arranging elements of a data structure in some order (ascending or descending).
Stack	A LIFO (Last In First Out) list.
Traversal	Processing each and every element of a data structure.
Tree	A multilevel data structure representing hierarchical relationship among its data.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What are data structures ? Name some common data structures.
2. Is data structure related to a data type ? Explain.
3. What do you understand by linear and non-linear data structures ?
4. Name some linear data structures. Is linked list a linear data structure ?
5. What is the working principle of data structures stack and queues ?
6. What is a linear list data structure ? Name some operations that you can perform on linear lists.
7. Suggested situations where you can use these data structures: (*i*) linear lists , (*ii*) stacks, (*iii*) queues.
8. What is a list comprehension ? How is it useful ?
9. Enlist some advantages of list comprehensions.
10. In which situations should you use list comprehensions and in which situations you should not use list comprehensions ?
11. What is a nested list ? Give some examples.
12. What is a two dimensional list ? How is it related to nested lists ?
13. Suggest a situation where you can use a regular two dimensional list.
14. What are ragged lists ? How are these different from two dimensional lists ?
15. Suggest a situation where you can use ragged list ?
16. How are lists internally stored ? How are 2D lists internally stored ?
17. Consider the following code and show how internally the memory is assigned to these

```
List1 = [2, 3]
List2 = [3, 4, 5]
List3 = [List1, List2]
```

Assume that the numbers 1..10 are stored in the memory addresses 16000 onwards, each consuming 16 bytes of memory.

Type B : Application Based Question

1. Create a list **SqLst** that stores the doubles of elements of another list **NumLst**. Following code is trying to achieve this. Will this code work as desired ? What will be stored in **SqLst** after following code ?

```
NumLst = [2, 5, 1, 7, 3, 6, 8, 9]
SqLst = NumLst * 2
```

2. Change the above code so that it works as stated in previous question.
3. Modify your previous code so that SqLst stores the doubled numbers in ascending order.
4. Consider a list ML = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]. Write code using a list comprehension that takes the list ML and makes a new list that has only the even elements of this list in it.
5. Write equivalent list comprehension for the following code :

```
target1 = []
for number in source:
  if number & 1:
     target1.append(number)
```

6. Write equivalent for loop for the following list comprehension :

```
gen = (i/2 for i in [0, 9, 21, 32])
print(gen)
```

7. **Predict the output** of following code if the input is :

(*i*) 12, 3, 4, 5, 7, 12, 8, 23, 12 (*ii*) 8, 9, 2, 3, 7, 8

Code :

```
s = eval(input("Enter a list : "))
n = len(s)
t = s[1:n-1]
print(s[0]==s[n-1] and t.count(s[0])==0)
```

8. Predict the output :

```
def h_t(NLst):
    from_back = NLst.pop()
    from_front = NLst.pop(0)
    NLst.append(from_front)
    NLst.insert(0,from_back)
NLst1 = [ [21, 12], 31 ]
NLst3 = NLst1.copy()
NLst2 = NLst1
NLst2[-1] = 5
NLst2.insert(1,6)
h_t(NLst1)
print(NLst1[0], NLst1[-1], len(NLst1))
print(NLst2[0], NLst2[-1], len(NLst2))
print(NLst3[0], NLst3[-1], len(NLst3))
```

9. Predict the output :

```
ages = [11, 14, 15, 17, 13, 18, 25]
  print(ages)
Elig = [x for x in ages if x in range(14, 18)]
print(Elig)
```

10. Predict the output :

```
L1 = [x ** 2 for x in range(10) if x % 3 == 0]
L2 = L1
L1.append(len(L1))
print(L1)
print(L2)
L2.remove(len(L2) - 1 )
print(L1)
```

11. Predict the output :

```
def even(n):
    return n % 2 == 0
list1 = [1,2,3,4,5,6,7,8,9]
ev = [n for n in list1 if n % 2==0]
evp = [n for n in list1 if even(n) ]
print(evp)
```

12. Predict the output.

(*i*)
```
b = [[9,6],[4,5],[7,7]]
x = b[:2]
x.append(10)
print(x)
```

(*ii*)
```
b = [[9,6],[4,5],[7,7]]
x = b[:2]
x[1].append(10)
print(x)
```

13. Find the Error. Consider the following code, which runs correctly at times but gives error at other times. Find the error and its reason.

```
Lst1 = [23, 34, 12, 77, 34, 26, 28, 93, 48, 69, 73, 23, 19, 88]
Lst2 = []
print("List1 originally is:", Lst1)
ch = int(input("Enter 1/2/3 and predict which operation was performed?"))
if ch == 1:
    Lst1.append(100)
    Lst2.append(100)
elif ch == 2:
    print(Lst1.index(100))
    print(Lst2.index(100))
elif ch == 3:
    print(Lst1.pop())
    print(Lst2.pop())
```

14. Suggest the correction for the error(s) in previous question's code.

15. Find the error. Consider the following code(s) and predict the error(s):

(*i*) y for y in range(100) if y % 2 == 0 and if y % 5 == 0

(*ii*) (y for y in range(100) if y % 2 == 0 and if y % 5 == 0)

16. Find the error in the following list comprehension :

```
["good" if i < 3: else: "better" for i in range(6)]
```

17. Suggest corrections for the errors in both the previous questions.

Type C : Programming Practice/Knowledge based Questions

1. Write a program that uses a function called find_in_list( ) to check for the position of the first occurrence of v in the list passed as parameter (lst) or -1 if not found. The header for the function is given below :

```
def find_in_list (lst, v ):
""" lst - a list
v - a value that may or may not be in the list """
```

2. Implement the following function for a linear list, which find outs and returns the number of unique elements in the list

```
def unique(lst):
"""passed parameter lst is a list of integers (could be empty)."""
```

After implementing the above function, test it with following lists and show the output produced by above function along with the reason for that output.

(*i*) lst = [ ] (*ii*) lst = [1,2,3] (*iii*) lst = [1,2,2] (*iv*) lst = [1,2,2,3,3]

3. Use a list comprehension to create a list, **CB4**. The comprehension should consist of the cubes of the numbers 1 through 10 only if the cube is evenly divisible by four. Finally, print that list to the console. Note that in this case, the cubed number should be evenly divisible by 4, not the original number.

4. Take two lists, say for example these two :

```
a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

and write a program that returns a list that contains only the elements that are common between the lists (without duplicates). Make sure your program works on two lists of different sizes. Write this in one line of Python using at least one list comprehension. Run the complete program and show output.

5. Suppose we have a list V where each element represents the visit dates for a particular patient in the last month. We want to calculate the highest number of visits made by any patient. Write a function **MVisit(Lst)** to do this.

A sample list (V) is shown below :

```
V = [[2, 6], [3, 10], [15], [23], [1, 8, 15, 22, 29], [14]]
```

For the above given list V, the function *MVisit( )* should return the result as [1, 8, 15, 22, 29].