

Chapter One

Course objectives

This course aims to make the student capable of understanding and writing different data structures as:

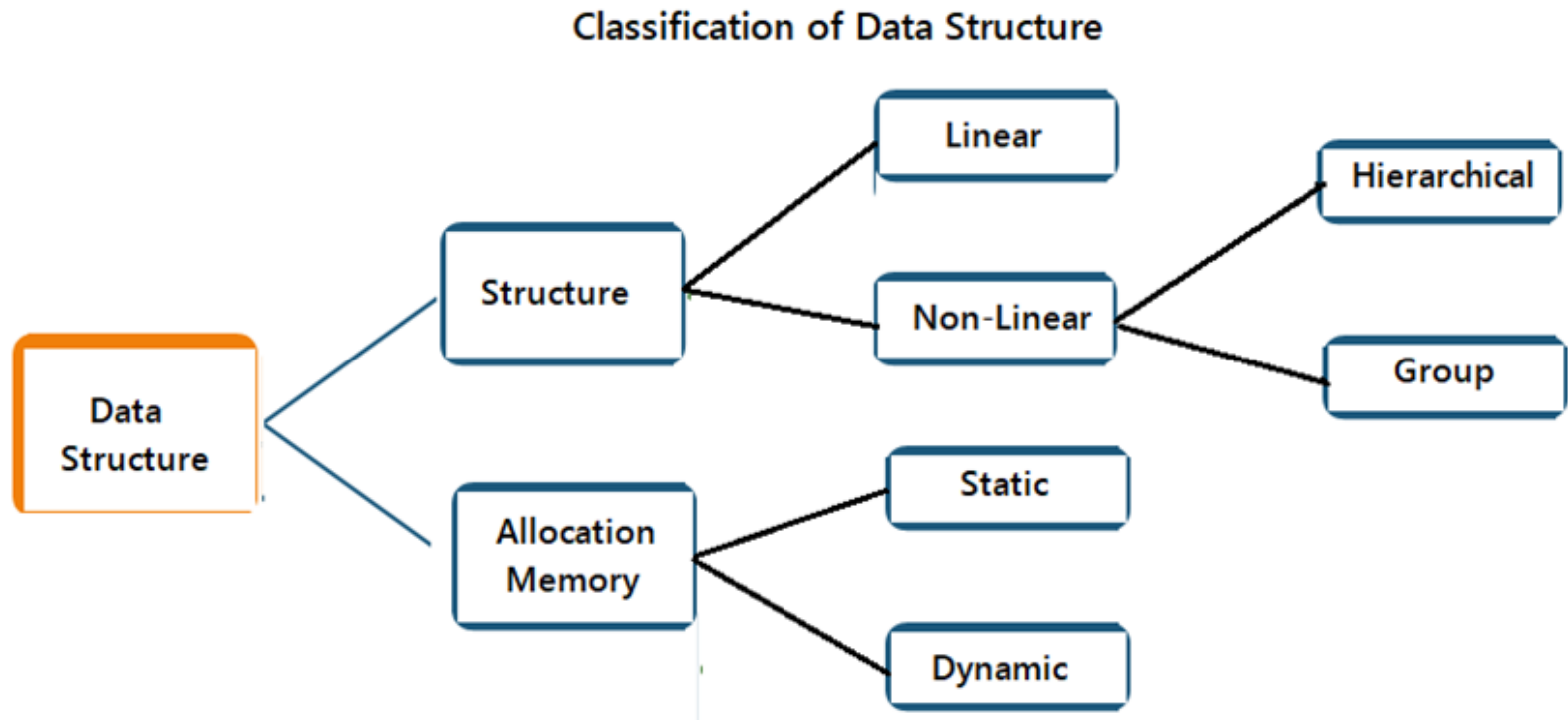
- Array, ArrayList
- String
- Linked List
- Stack
- Queue

Introduction

What is Data Structures ?

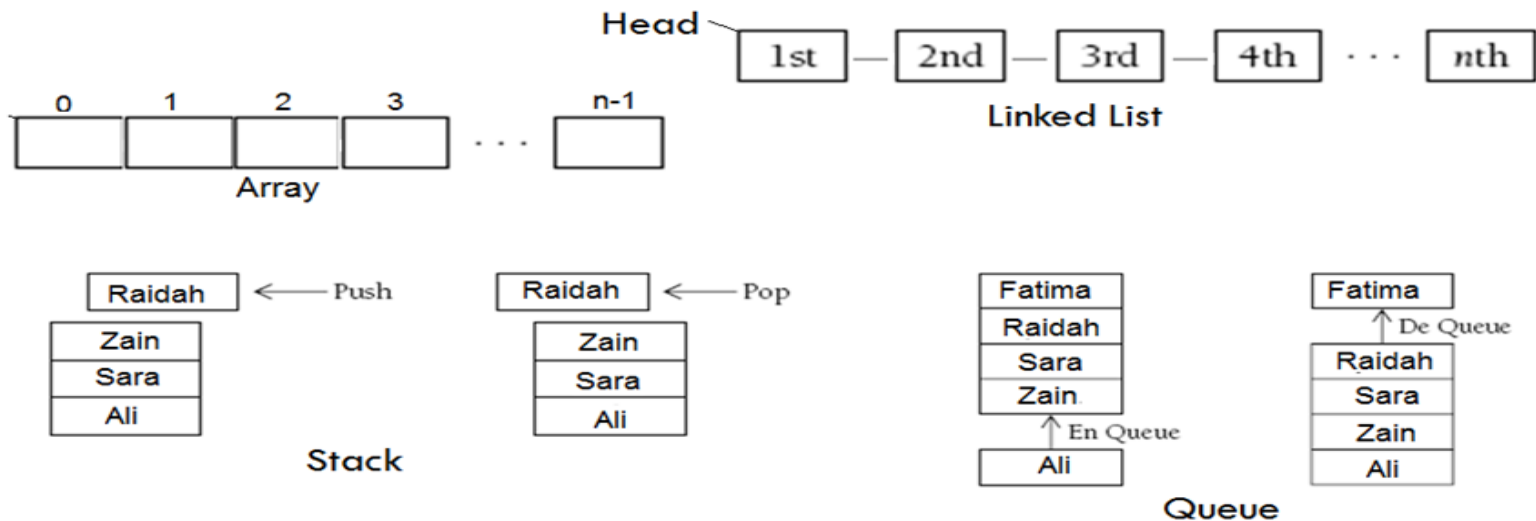
- A collection of basic data types is called data structure.
- A collection of data elements whose logical organization reflects a relationship among the elements.
- Abstract way to organize information.

Classification of data structure



Classification according to structure

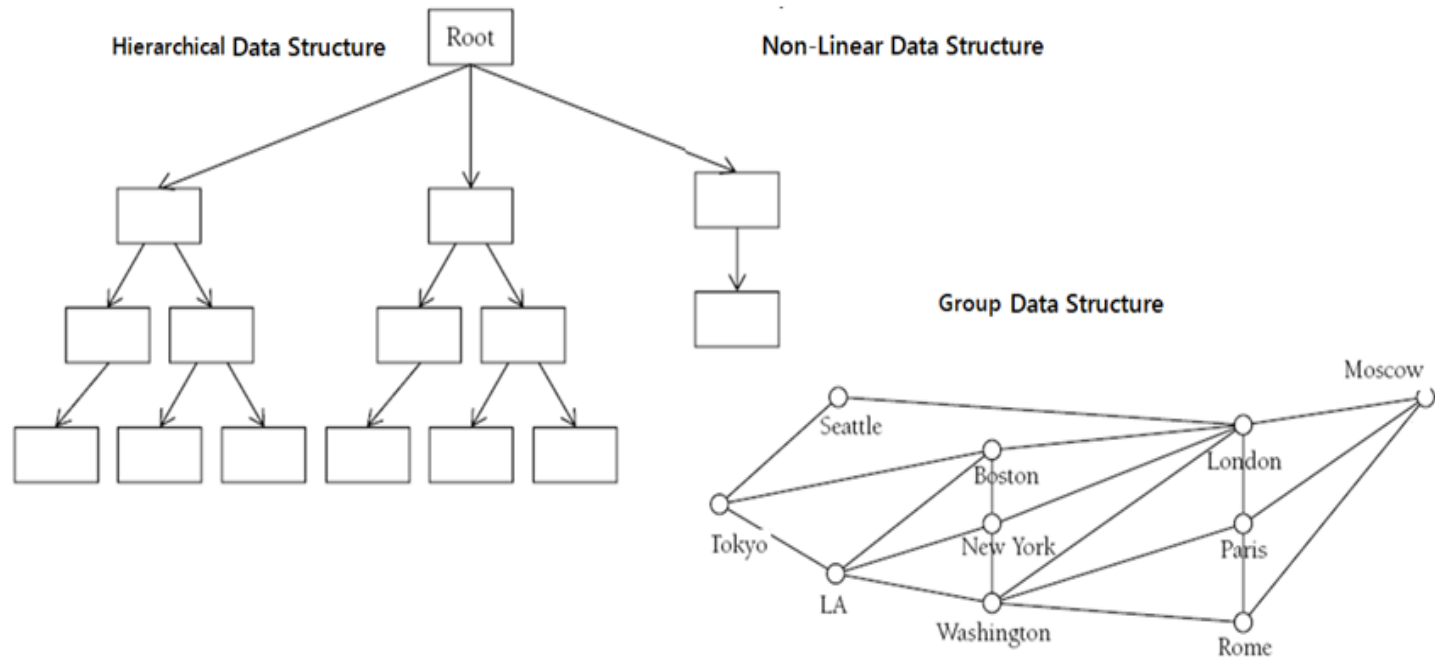
1. **Linear data structures:** In these data structures the elements form a sequence, such as array, Linked Lists, Stacks and Queues .



2. **Non-Linear** data structures: In these data structures the elements do not form a sequence, such as Trees(example of **hierarchical collection**) and Graphs(example of **Group**):
3. **A hierarchical collection** is a group of items divided into levels. An item at one level can have successor items located at the next lower level.Example: Tree
2. **Group**: A nonlinear collection of items that are unordered is called a **group**. Example : Sets and Graphs.

Classification according to Allocation memory

1. **Static** memory allocation means the program must obtain its space before the execution and cannot obtain more while or after execution. Example: Arrays
2. **Dynamic** memory allocation is the ability for a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed .Example : Array lists, Linked Lists, Stacks, Queues and Trees



The data structures can be viewed in two ways, physically and logically.

- The physical data structure refers to the physical arrangement of the data on the memory.
- The logical data structure concerns how the data "seem" to be arranged and the meanings of the data elements in relation to one another.

Data Structures Operations

Following the operations that can be performed on the data structures:

1. **Traversing:** It is used to access each data item exactly once.
2. **Searching:** It is used to find out the location of the data item.
3. **Inserting:** It is used to add a new data item in the given collection of data items.
4. **Deleting:** It is used to delete an existing data item from the given collection of data items.
5. **Sorting:** It is used to arrange the data items in some order i.e. in ascending or descending order.

Array Data Structure

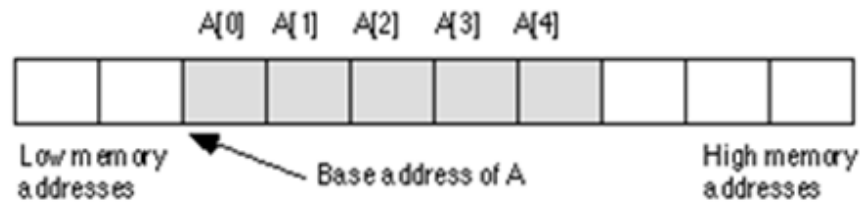
An array is :

- **Linear data structure:** means organize in memory as linear order.
- **homogeneous structure:** all components in the structure are of the same data type.
- **finite structure :** indicates that there is a last element.
- **fixed size structure:** mean that the size of the array must be known at compile time.
- **contiguously data structure:** means that there is a first element, a second element, and so on.
- **The component selection mechanism of an array is direct access,** which means we can access any element directly (by using specific equation), without first accessing the preceding elements. The desired element is specified using an index, which gives its relative position in the collection.

- A one-dimensional array is homogeneous structure, it can be visualized as a list. Logical structure for one-dimensional array with 15 elements as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	7	9	12	15	23	30	45	70	77	80	88	90	99	

- Physical structure for one-dimensional array with five elements will appear in memory as shown below:



Linear Array A			Base Address
0	10	1000	↖
1	100	1002	
2	20	1004	
3	500	1006	
4	600	1008	
<div> <div>Index</div> <div>Value</div> <div>Address (in bytes)</div> </div>			

Linear Array A			Base Address
1	10	1000	↖
2	100	1002	
3	20	1004	
4	500	1006	
5	600	1008	
<div> <div>Index</div> <div>Value</div> <div>Address (in bytes)</div> </div>			

We can find out the location of any element by using following formula:

$$\text{Loc (ArrayName [k])} = \text{Loc (ArrayName [1])} + (k - \text{LB}) * w$$

where:

Loc (ArrayName [k]): is the address of the kth element of ArrayName.

Loc (ArrayName [1]): is the base address or address of first element of ArrayName.

W : is the number of bytes taken by one element(element size).

LB : is the lower bound.

Example1 : Suppose we want to find out Loc (A [3]) that store as the following figure in two case (LB=1 and LB=0), for it, we have: Base(A)=1000, w = 2 bytes.

in Case LB = 1

After putting these values in the given formula, we get:

$$\begin{aligned}\text{LOC}(A[3]) &= 1000 + 2 (3 - 1) \\ &= 1000 + 2 (2) \\ &= 1000 + 4 \\ &= 1004\end{aligned}$$

If Case LB = 0

After putting these values in the given formula, we get:

$$\begin{aligned}\text{LOC}(A[3]) &= 1000 + 2 * 3 \\ &= 1000 + 6 \\ &= 1006\end{aligned}$$

Two-dimensional array

- A two-dimensional array is also homogeneous structure, it can be visualized as a table consisting of rows and columns.
- The element in a two-dimensional array is accessed by specifying the row and column indexes of the item in the array.
- **Logical Level** for two-dimensional array with 3(0-2) rows and 4(0-3)column elements will appear as shown below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0, 0]	a[0, 1]	a[0, 2]	a[0, 3]
Row 1	a[1, 0]	a[1, 1]	a[1, 2]	a[1, 3]
Row 2	a[2, 0]	a[2, 1]	a[2, 2]	a[2, 3]

Diagram illustrating array indexing for a 2D array `a`.

The array is represented as a table with rows and columns. The row indices are 0, 1, and 2. The column indices are 0, 1, 2, and 3.

The elements are labeled as `a[row index, column index]`.

Labels and arrows pointing to the elements:

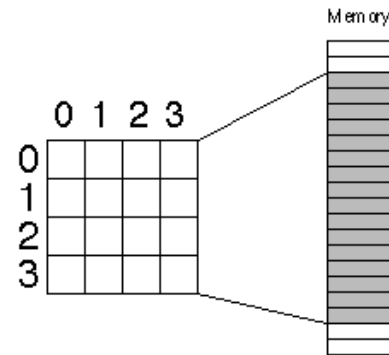
- Column index (or subscript) points to the second index (column index) in the subscript.
- Row index (or subscript) points to the first index (row index) in the subscript.
- Array name points to the variable `a`.

In the computer memory, all elements are stored linearly using contiguous addresses. Therefore, in order to store a two-dimensional matrix, two dimensional address space must be mapped to one-dimensional address space.

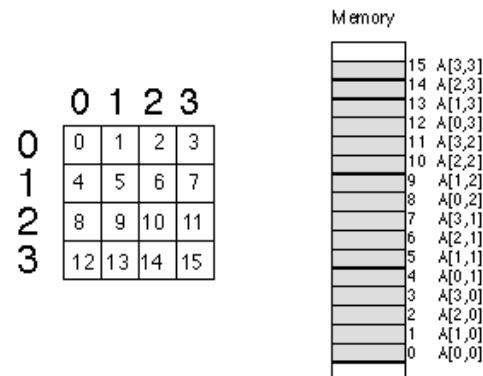
There are two methods for arranging two or multidimensional [arrays](#) in [memory](#):

1. Row-major order
2. Column-major order

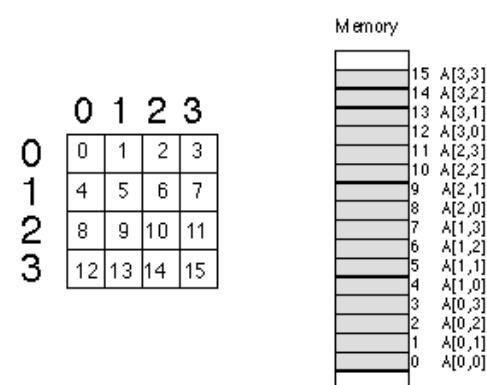
For example the physical structure for array with 3X3 elements will appear in memory as shown below in two methods:



Column Major Ordering



Row Major Ordering



- **In row-major order:**

consecutive elements of the **rows** of the array are contiguous in memory. Used in [C/C++](#), [PL/I](#), [Pascal](#), [Python](#) , [C# .Net](#), [Java](#).

- **in column-major order:**

consecutive elements of the columns are contiguous.Used in [Fortran](#), [OpenGL](#) , [MATLAB](#)

- **Example:** The following array: A would be stored as follows in the Two orders:

Column-major order
e.g., Fortran

Address	Value
0	a_{11}
1	a_{21}
2	a_{12}
3	a_{22}
4	a_{13}
5	a_{23}

Row-major order
e.g., C++

Address	Value
0	a_{11}
1	a_{12}
2	a_{13}
3	a_{21}
4	a_{22}
5	a_{23}

We can find out the location of any element in array (NXM) by using following formulas:

1. In case of Row Major Order:

$$\text{Loc (A [i, j])} = \text{Loc (A [1,1])} + ([i-1]*M + [j-1]) * w$$

where

Loc(A[[i,j]) : is the location of the element in the ith row and jth column.

Loc (A [1,1]) : is the base address or address of the first element of the array A.

w : is the number of bytes required to store single element of the array A.

M : is the total number of columns in the array.

•

Example: finding the location (address) of element in 2D

Suppose $A_{3 \times 4}$ (N=3 and M=4) integer array A is shown as below and base address = 1000 and number of bytes=2. find the location of $A[3,2]$:

A:

10	20	50	60
90	40	30	80
75	55	65	79

Address Elements

1000	10
1002	20
1004	50
1006	60
1008	90
1010	40
1012	30
1014	80
1016	75
1018	55
1020	65
1022	79

$$\begin{aligned}\text{LOC}(A[3,2]) &= 1000 + 2[4(3-1) + (2-1)] \\ &= 1000 + 2[4(2) + 1] \\ &= 1000 + 2[8 + 1] \\ &= 1000 + 2[9] \\ &= 1000 + 18 \\ &= 1018\end{aligned}$$

2. In case of Column Major Order:

$$\text{Loc (A [i,j])} = \text{Loc (A [1, 1])} + ([j-1]*N + [i-1])*w$$

where

Loc(A[i,j]): is the location of the element in the ith row and jth column.

Loc (A [1,1]): is the base address or address of the first element of the array A.

w: is the number of bytes required to store single element of the array A.

N: is the total number of rows in the array.

Example: finding the location (address) of element in 2D

Suppose $A_{3 \times 4}$ (N=3 and M=4) integer array A and base address =1000 and number of bytes=2. find the location of A [3,2]:

Address	Elements	
1000	10	$\begin{aligned}\text{LOC}(A[3,2]) &= 1000 + 2[3(2-1) + (3-1)] \\ &= 1000 + 2[3(1) + 2] \\ &= 1000 + 2[3 + 2] \\ &= 1000 + 2[5] \\ &= 1000 + 10 \\ &= 1010\end{aligned}$
1002	90	
1004	75	
1006	20	
1008	40	
1010	55	
1012	50	
1014	30	
1016	65	
1018	60	
1020	80	
1022	79	

Note: if the value of w not determine, it suppose equal to 1.

Three Dimensional Arrays

In case three Dimensional Arrays, memory-address of the element $A[i,j,K]$ with dimension $(N \times M \times R)$ is given by:

where:

R : number of levels

N: number of rows

M: number of column

In case of Row Major Order:

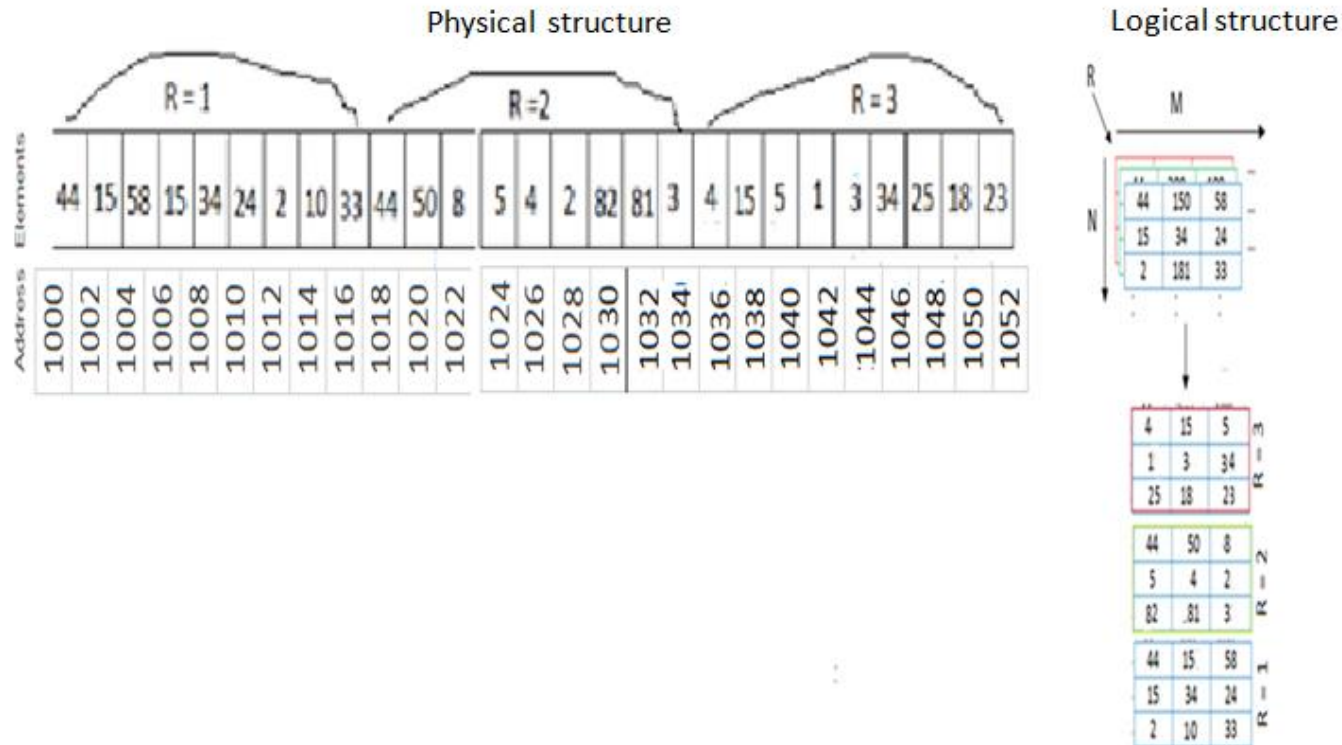
$$\text{Loc (A [i,j,k])} = \text{Loc (A [1, 1, 1])} + ([k-1]*N*M + [j-1]*N + (i-1))*W$$

In case of Column Major Order:

$$\text{Loc (A [i,j,k])} = \text{Loc (A [1, 1, 1])} + ([k-1]*N*M + [i-1]*M + (j-1))*W$$

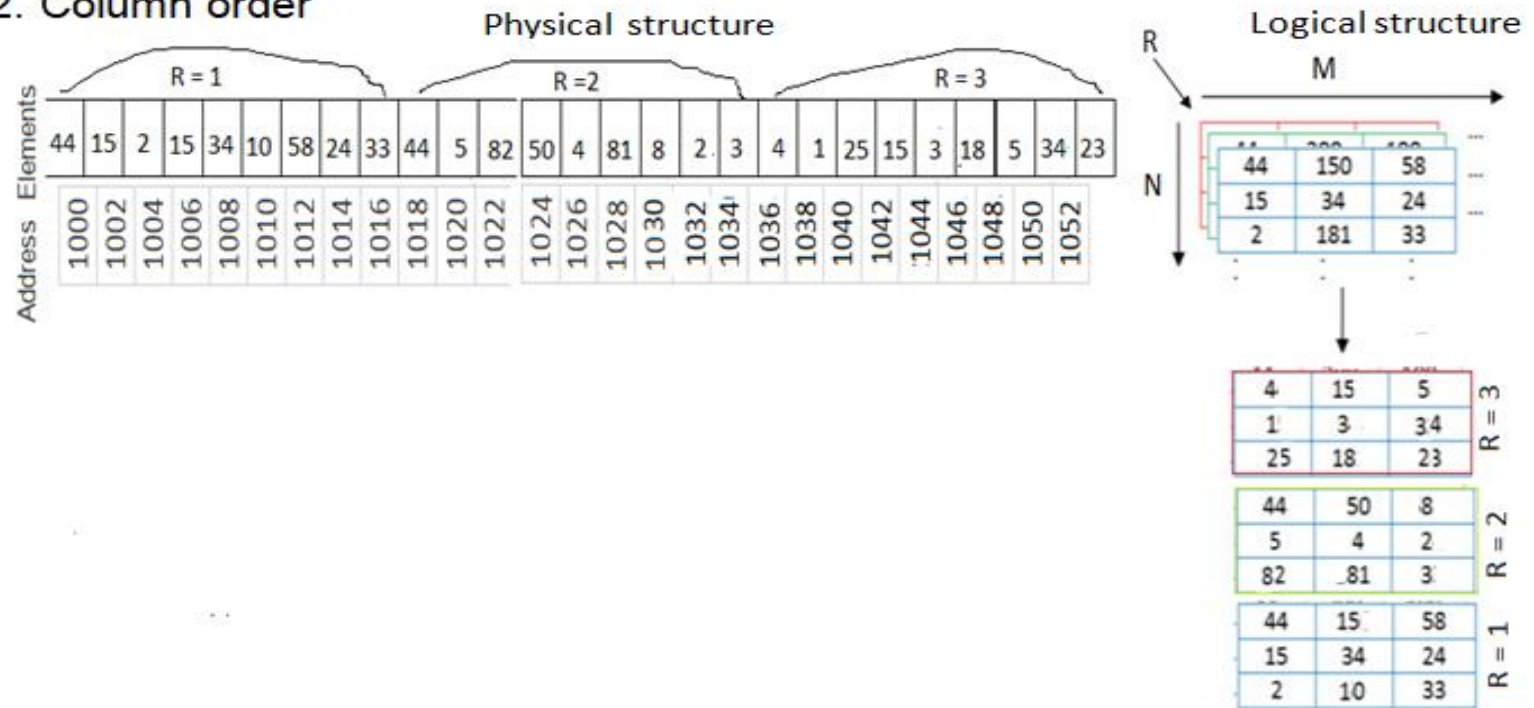
Example : Suppose A3 x 4(N=3, M=3 and R=3) integer array A and base address =1000 and number of bytes(w)=2. find the location of A [3,2,2] by using two method of arrange matrix in memory:

1. Row order



$$\begin{aligned}
 \text{Loc}(A[3,2,2]) &= 1000 + 2(3 \times 3 \times (2-1) + 3 \times (3-1) + 2-1) \\
 &= 1000 + 2(9+6+1) \\
 &= 1032
 \end{aligned}$$

2. Column order



$$\begin{aligned}
 \text{Loc}(A[3,2,2]) &= 1000 + 2(3 \times 3 \times (2-1) + 3 \times (2-1) + 3-1) \\
 &= 1000 + 2(9+3+2) \\
 &= 1028
 \end{aligned}$$

Triangular Matrix

A triangular matrix is a special kind of square matrix. A square matrix is called lower triangular if all the entries above the main diagonal are zero. Similarly, a square matrix is called upper triangular if all the entries below the main diagonal are zero.

Logical Structure

Upper Triangular Matrix

44	150	58
0	34	24
0	0	33

if $i > j$ then $A_{ij} = 0$

Lower Triangular Matrix

44	0	0
58	34	0
150	24	33

if $i < j$ then $A_{ij} = 0$

Physical Storage

Address Elements

1000	44
1002	150
1004	58
1006	34
1008	24
1010	33

Address Elements

1000	44
1002	150
1004	34
1006	58
1008	24
1010	33

Row Major order

Column Major order

Address Elements

1000	44
1002	58
1004	34
1006	150
1008	24
1010	33

Address Elements

1000	44
1002	58
1004	150
1006	34
1008	24
1010	33

We can find out the location of a[2,2] by using following formulas:

1. Upper triangular

In case of Row Major Order:

$$\text{Loc (A [i, j])} = \text{Base(A)} + w((i-1)*M - (i-1)*i/2 + (j-1))$$

Address Elements

1000	44
1002	150
1004	58
1006	34
1008	24
1010	33

suppose Base (A)=1000 , i=2, j=2 and w=2
 $\text{LOC (A [2,2])} = 1000 + 2*((2-1)*3-(2-1)*2/2+2-1)$
 $= 1000 + 2(3-1+1)$
 $= 1000 + 2*3$
 $= 1000 + 6$
 $= 1006$

In case of Column Major Order:

$$\text{Loc (A[i, j])} = \text{Base(A)} + w((j-1) *j / 2 + (i-1))$$

Address Elements

1000	44
1002	150
1004	34
1006	58
1008	24
1010	33

suppose Base (A)=1000 , i=2, j=2 and w=2
 $\text{LOC (A [2,2])} = 1000 + 2*((2-1)*2/2+2-1)$
 $= 1000 + 2(1+1)$
 $= 1000 + 2*2$
 $= 1000 + 4$
 $= 1004$

2. Lower triangular

In case of Row Major Order:

$$\text{Loc (A [i,j])} = \text{Base(A)} + w((i-1) * i / 2 + ([j-1]))$$

Address Elements

1000	44	suppose Base (A)=1000 , i=2, j=2 and w=2 LOC (A [2,2]) = $1000 + 2*((2-1)*2/2+2-1)$ = $1000 + 2(1+1)$ = $1000 + 2*2$ = $1000 + 4$ = 1004
1002	58	
1004	34	
1006	150	
1008	24	
1010	33	

In case of Column Major Order:

$$\text{Loc (A [i,j])} = \text{Base(A)} + w(((j-1) * N - (j-1)*j/2) + (i-1))$$

Address Elements

1000	44	suppose Base (A)=1000 , i=2, j=2 and w=2 LOC (A [2,2]) = $1000 + 2*((2-1)*3 - (2-1)*2/2+2-1)$ = $1000 + 2(3-1+1)$ = $1000 + 2*3$ = $1000 + 6$ = 1006
1002	58	
1004	150	
1006	34	
1008	24	
1010	33	

Q: How determine the number of array elements?

Ans.: To determine the number of any array elements by applying the following equation:

$$M_{i=1}^n ((U_i - L_i) + 1)$$

where:

n is dimensions of the array

U : upper bound for dimension i

L: lower bound for dimension i

**Example1: Find the number of positions required to store the array: A [5]
=5-0+1=6**

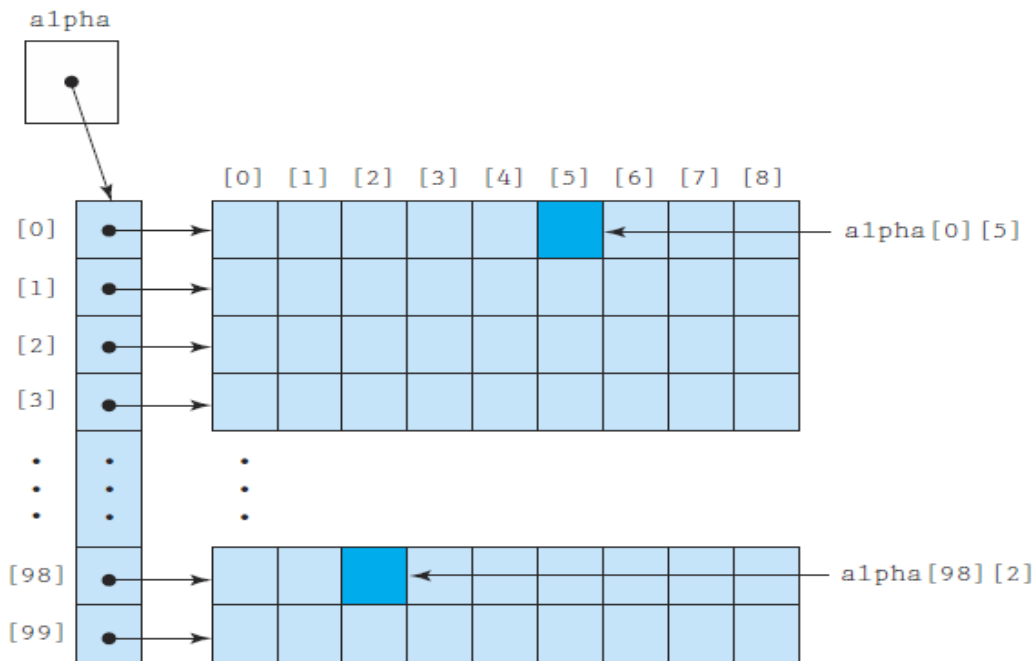
$$M_{i=1}^2 ((U_1 - L_1) + 1) * ((U_2 - L_2) + 1)$$

**Example2: Find the number of positions required to store the matrix: A [5, 6]
(5-0+1)*(6-0+1)=6*7=42**

**Example3: Find the number of positions required to store the matrix:
A[2..5, 6...8]
= (5-2+1)*(8-6+1) = 4*3 =12**

Arrays in Java

- In Java each row of a two-dimensional array is itself a one dimensional array.

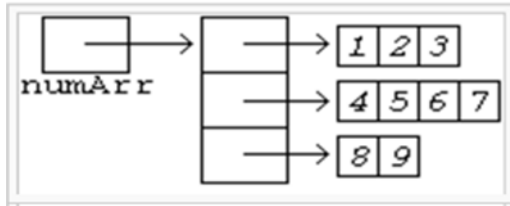


- Arrays of arrays in languages such as Java, Python (multidimensional lists), Visual Basic.NET, Perl, PHP, JavaScript are implemented as Iliffe vectors and they can also be used to implement jagged arrays.
- The Iliffe vector for a 2-dimensional array is simply a vector of pointers to vectors of data, i.e., the Iliffe vector represents the columns of an array where each column element is a pointer to a row vector.
- In general, an Iliffe vector for an n -dimensional array (where $n \geq 2$) consists of a vector (or 1-dimensional array) of pointers to an $(n - 1)$ -dimensional array. They are often used to avoid the need for expensive multiplication operations when performing address calculation on an array element.
- While a 2D array is a 1D array of references to 1D arrays, each of these 1D arrays (rows) can have a different length, this 2D array is called jagged arrays.

Example to initialize [jagged array](#):

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

In the following memory layout of a jagged array numArr.



And jagged arrays can be created with the following code:

```
int [][]c;  
c=new int[2][];  
c[0]=new int[5];  
c[1]=new int[3];
```

Example : suppose the following declaration:

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

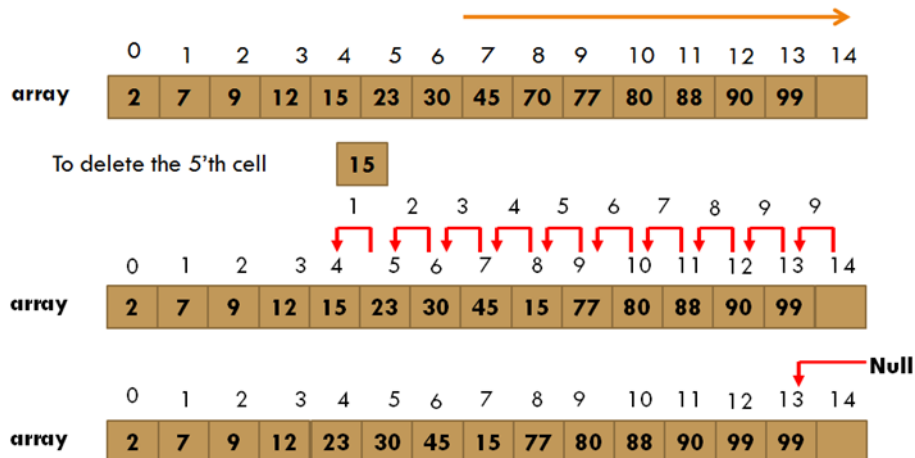
3	5	7	9
4	2		
5	7	8	6
6			

Q: Show output of the following segment?

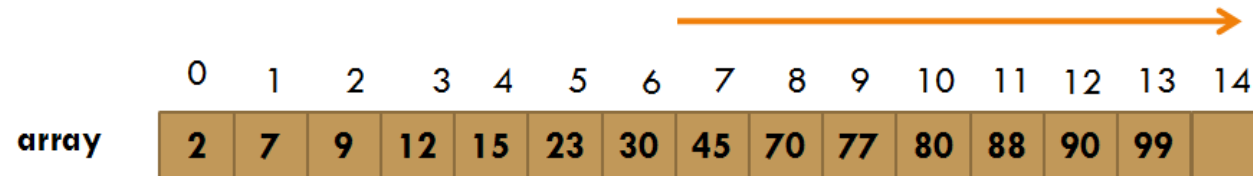
```
int count = 0; double sum = 0;
for (int row = 0; row < jagged.length; row++)
    if (jagged[row].length > 3){
        sum += jagged[row][3];
        count++; }
if (count > 0)
    System.out.println((double) sum / count);
```

Q:How to delete and insert an element from array?

Ans.: Insertion and deletion at particular position is complex, it require shifting as in the following examples.



Insert item to sorted array



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
array	2	7	9	12	15	23	30	45	70	77	80	88	90	99	

To insert value

13

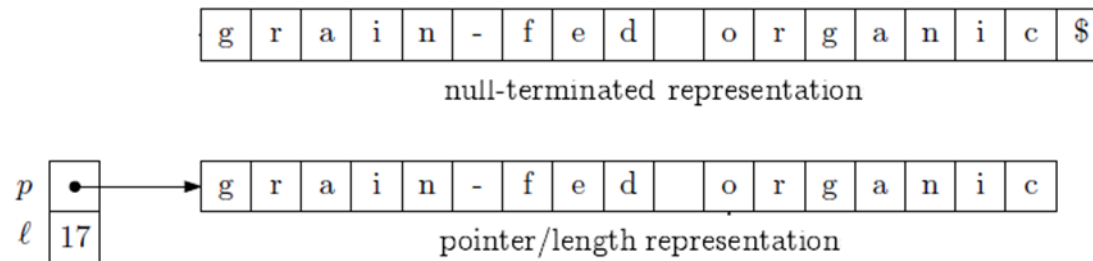
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
array	2	7	9	12	13	15	23	30	45	70	77	80	88	90	99

Chapter Two

String

What is a String?

- A string is collection of characters grouped together. For example, "hello" is a string consisting of the characters 'h', 'e', 'l', 'l', and 'o'.
- In most programming languages, strings are generally understood as a data type, they are built in as part of the language and they take one of two basic representations as illustrated in following figures:
 1. null-terminated representation, in this, strings are represented as an array of characters that ends with the special null terminator `\0`.
 2. pointer/length representation, in this representation a string is represented as (a pointer to) an array of characters along with a integer that stores the length of the string. The pointer/length representation is more efficient for some operations. For example, in the pointer/length representation, determining the length of the string takes constant time, since it is already stored.



What is String in java?

In java, string is basically an object that represents sequence of char values. it is often implemented as:

- an array of bytes that stores a sequence of elements or as a reference type , mean string variable holds the address of the memory location where the actual body of the string is stored.

For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};  
String s=new String(ch);
```

- Java Strings can't change the characters, but string variables can point to different strings:

```
String s;
```

```
s = "java language";
```

```
s = "java is oop";
```

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

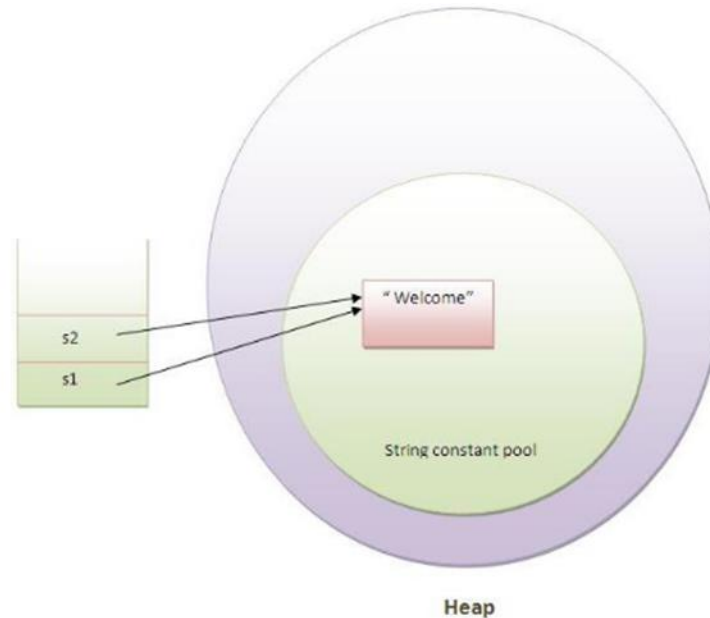
String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



- In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.
- Note: String objects are stored in a special memory area known as string constant pool.

Q: Why java uses concept of string literal?

Ans.: To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

new keyword

Example

CODE:

```
String st = new String ("hello");
```

MEMORY:

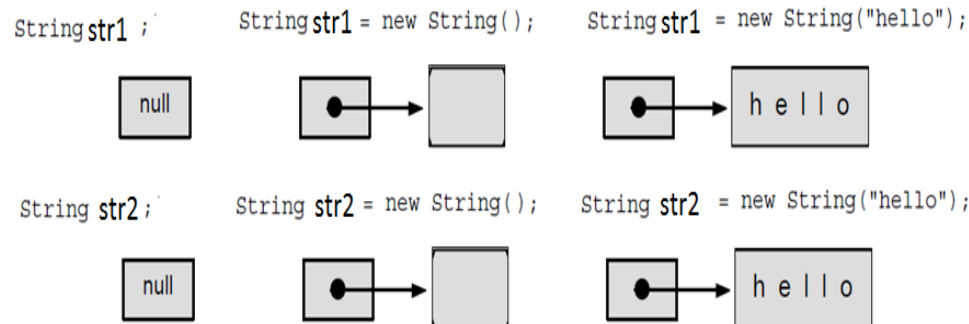


Each time a new declaration a compiler would create different object in memory

Example:

```
String str1 = new String("Hello");
```

```
String str2 = new String("Hello");
```



In this case compiler would create two different object in memory having the same text.

Q: What is the output of the following program?

```
public class StringExample{  
    public static void main(String args[]){  
        String s1="java";                //creating string by java string literal  
        char ch[]={'s','t','r','i','n','g','s'};  
        String s2=new String(ch);        //converting char array to string  
        String s3=new String("example"); //creating java string by new keyword  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println(s3); }  
}
```

Java String class methods

The `java.util.String` class provides many useful methods to perform operations on sequence of char values.

Assignment of string:

While primitive types are *value types*:

The memory of x and y both contain "5".

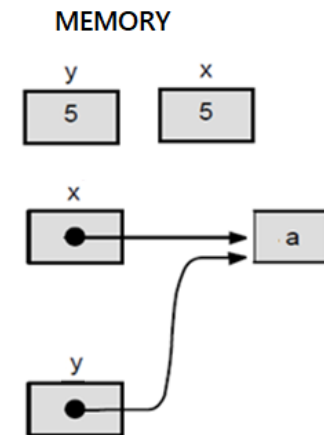
```
int x = 5; int y = x;
```

But with *reference type*:

The memory of x and y both contain a *pointer* to the character "a".

```
String x = "a";
```

```
String y = x;
```



No.	Method	Description
1	<u>char charAt(int index)</u>	returns char value for the particular index
2	<u>int length()</u>	returns string length
3	<u>String substring(int beginIndex)</u>	returns substring for given begin index
4	<u>String substring(int beginIndex, int endIndex)</u>	returns substring for given begin index and end index
5	<u>boolean equals(Object another)</u>	checks the equality of string with object
6	<u>boolean isEmpty()</u>	checks if string is empty
7	<u>String concat(String str)</u>	concatenates specified string
8	<u>String replace(char old, char new)</u>	replaces all occurrences of specified char value
9	<u>String[] split(String regex)</u>	returns splitted string matching regex
10	<u>String[] split(String regex, int limit)</u>	returns splitted string matching regex and limit
11	<u>String intern()</u>	returns interned string
12	<u>int indexOf(int ch)</u>	returns specified char value index
13	<u>int indexOf(int ch, int fromIndex)</u>	returns specified char value index starting with given index
14	<u>int indexOf(String substring)</u>	returns specified substring index
15	<u>int indexOf(String substring, int fromIndex)</u>	returns specified substring index starting with given index
16	<u>String toLowerCase()</u>	returns string in lowercase.
17	<u>String toLowerCase(Locale l)</u>	returns string in lowercase using specified locale.
18	<u>String toUpperCase()</u>	returns string in uppercase.
19	<u>String toUpperCase(Locale l)</u>	returns string in uppercase using specified locale.
20	<u>String trim()</u>	removes beginning and ending spaces of this string.
21	<u>static String valueOf(int value)</u>	converts given type into string. It is overloaded.

Chapter Three

Linked lists data structure

Static and Dynamic data structures


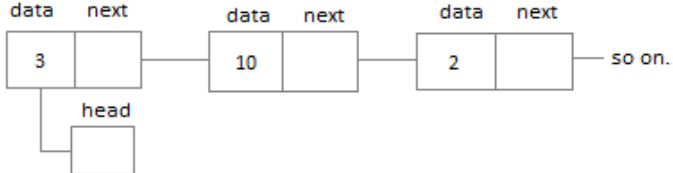
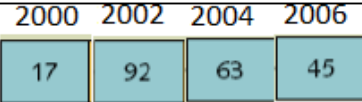
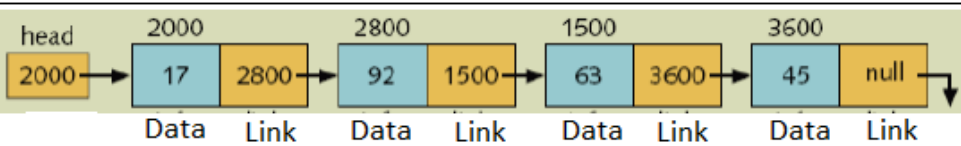
A static data structure is an organization of collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later.

Arrays are example of static data structure.

A dynamic data structures , where in with the latter the size of the structure can dynamically grow or shrink in size as needed.

Linked lists are example of dynamic data structure.

What are different between linked lists and arrays?

Arrays	Linked lists
Have a pre-determined fixed size	No fixed size; grow one element at a time
Array items are store contiguously	Element can be stored at any place
Easy access to any element in constant time	No easy access to <u>i-th</u> element , need to hop through all previous elements start from start
Size = $n \times \text{sizeof}(\text{element})$	Size = $n \times \text{sizeof}(\text{element}) + n \times \text{sizeof}(\text{reference})$
Insertion and deletion at particular position is complex(shifting)	Insertion and deletion are simple and faster
Only element store	Each element must store element and pointer to next element (extra memory for storage pointer).
	
	

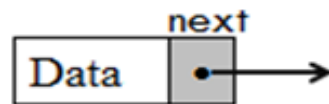
What is linked list?

A linked list is a collection of objects called nodes. Each node is linked to its successor node in the list using a reference to the successor node.

What is linked list element?

The linked list is sequence of nodes arranged one after another, with each node connected to the next by a "link" (like a chain) each node contain:

- 1) a single element - stored object or value(We call this part Data)
- 2) links - reference to one or both neighboring nodes(We call this next)

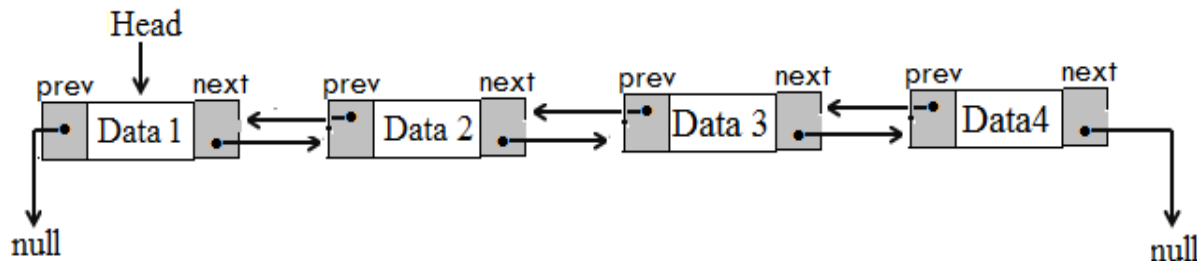


Types of linked lists

- **Singly Linked List** : Singly linked lists contain nodes which have a data part as well as an address part (next) which points to the next node in sequence of nodes.

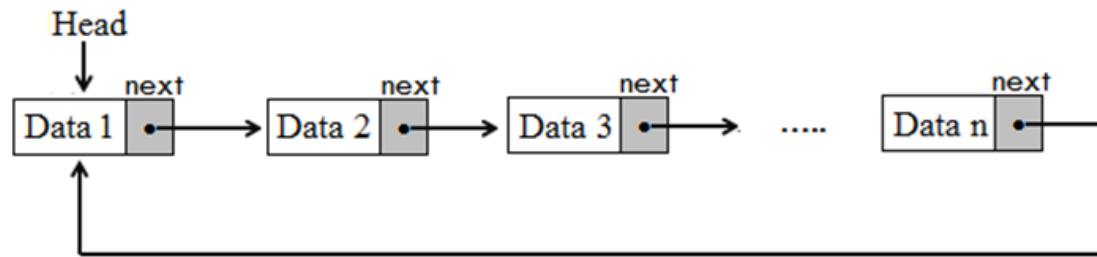


- **Doubly Linked List** : In a doubly linked list, each node (except the first and last node) contains two links: the first link points to the previous node and the next link points to the next node in the sequence.



Types of linked lists

- **Circular Linked List** : In the circular linked list the last node of the list contains the address of the first node.



Basic Operations

Following are the basic operations supported by an list.

- 1.Insert in beginning: add an element at the beginning of the list.
- 2.Insert in the end : add an element in the end of the list.
- 3.Insert after element : add an element after an item of the list.
- 4.Insert before element: add an element before an item of the list.
- 5.Delete first element: delete the element at the beginning of the list.
- 6.Delete last element: delete the element from the end of the list.
- 7.Delete and insert specific element.
- 8.Display forward: displaying complete list in forward manner.
- 9.Display backward: displaying complete list in backward manner.

Singly linked lists

To construct linked list we need define two class:

1. node class
2. singly linked list class(SLL)

```
class Node {  
    int data;  
    node next;  
    node(int data) {  
        this.data = data; } }  
class SLL {  
    node head;  
  
    public SLL() {  
        this.head = null;  
    }  
    // methods  
    ...  
}
```

In the following steps for create linked list with data(22, 33, 44, 55,66)

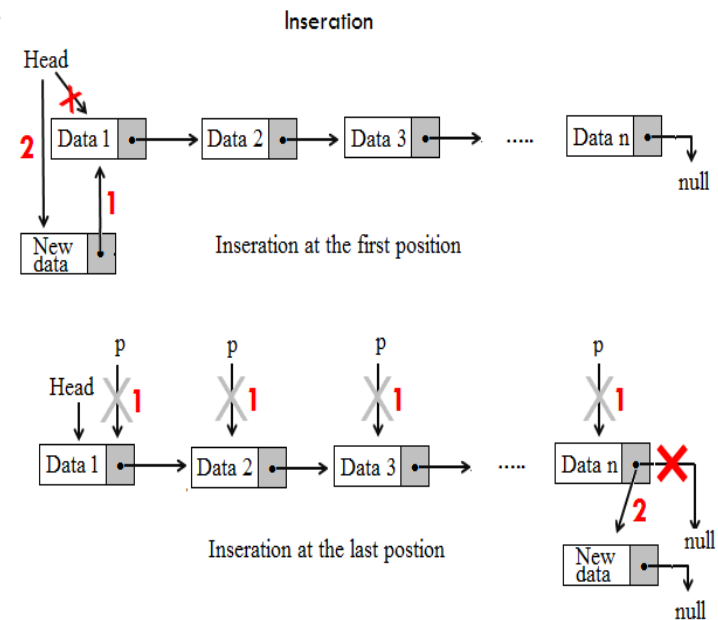
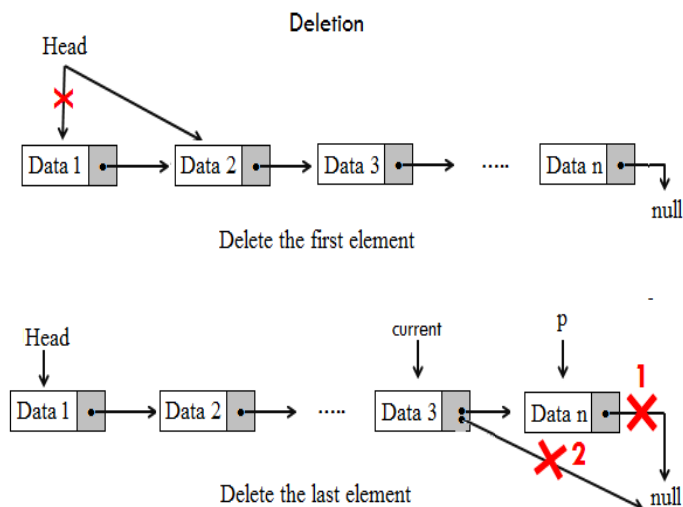
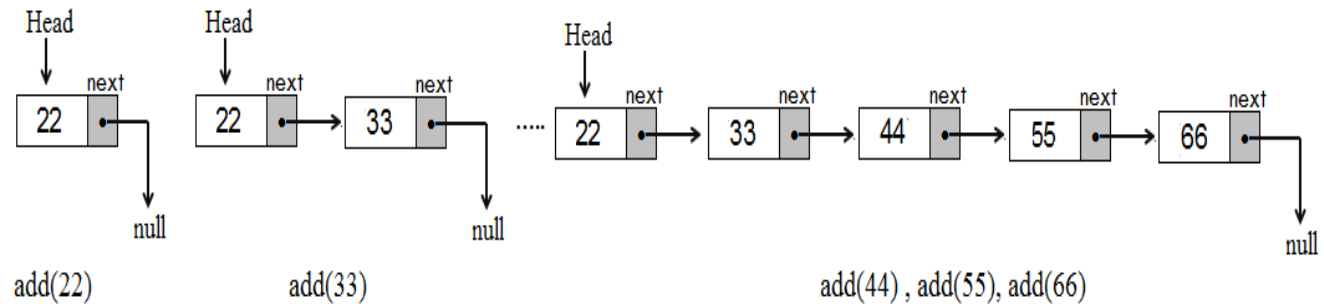
```
node head = new node(22);  
head.next = new node(33);  
head.next.next = new node(44);  
head.next.next.next = new node(55);  
head.next.next.next.next = new node(66);
```

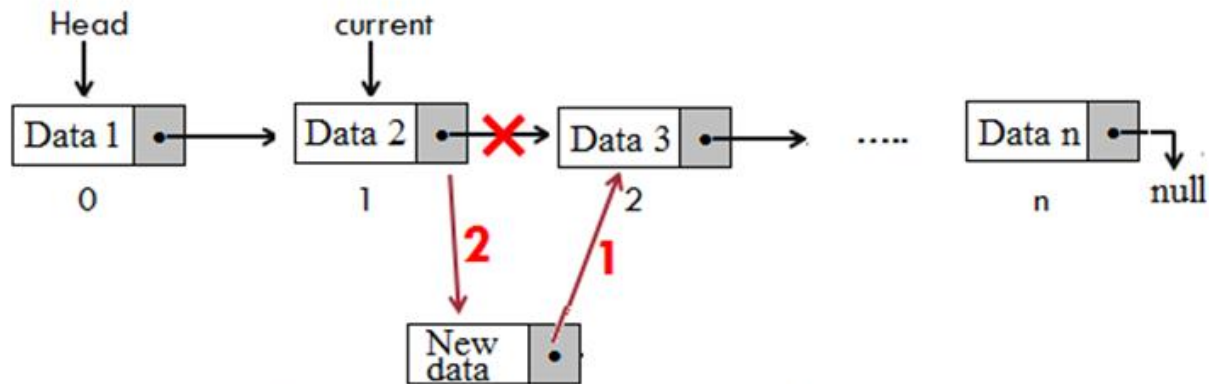
or

```
head = new node(22);  
node p= head;  
p.next = new node(33);  
p = p.next;  
p.next = new node(44);  
p = p.next;  
p.next = new node(55);  
p = p.next;  
p.next = new node(66);
```

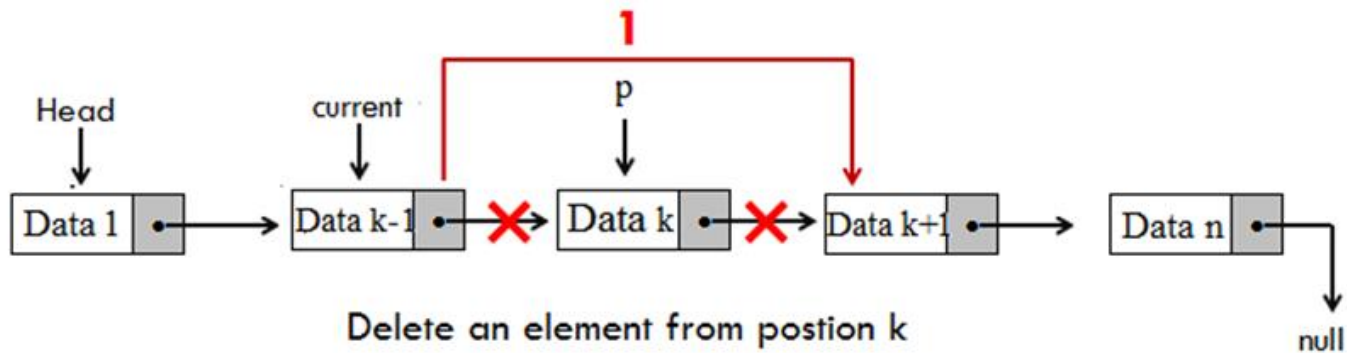
In the following figure shown some operation on singly linked list :

Example





Insert an element at position 2
(the third element)



Delete an element from position k

Some code segments in java to processing singly linked list

// method to print elements of singly linked list

```
public void printSLL() {  
    for (node p = head; p != null; p = p.next) {  
        System.out.print(p.data + " ");  
    }  
}
```

// method for adding new node

void addLast1 (int value){

```
    node newNode = new node(value);  
    node p = head;  
    if (head == null)  
        head = newNode;  
  
    else {  
        while (p.next != null)  
            p = p.next;  
        p.next = newNode;  
    }  
}
```

```
void addLast2(int value){
```

```
    node p;  
    node newNode = new node(value);  
    if (head==null)  
        head=newNode;  
    else {  
        for (p = head; p.next!= null; p=p.next)  
            {}  
        p.next=newNode;  
    }  
}
```

```
// method to delete first element
```

```
public void deleteFirst() {
```

```
    head = head.next;  
}  
or  
public void deleteFirst() {  
    node curr = head;  
    head = head.next;  
    curr.next = null;  
    curr = null;  
}
```

//method to delete element after element in the list

public void deleteAfter(int value) {

node p = head;

while (p.data != value)

p = p.next;

node curr=p.next;

p.next= curr.next;

}

// method to add an element after an item of the list

public void addAfter(int value, int newValue) {

node p = head;

while (p.data != value)

p = p.next;

node newNode = new node(newValue);

newNode.next = p.next;

p.next = newNode;

}

Doubly linked lists

In this type of linked list each node contains a pair of references. One reference points to the node that precedes the node (prior) and the other points to the node that follows the node (next).

To construct double linked list we need define two class:

```
class DListNode {  
    int data;  
    DListNode prior, next;  
} // DListNode
```

```
class DLL {  
    node head;  
    public DLL() {  
        this.head = null;  
    }  
    // methods  
    ...  
}
```

Advantages & Disadvantages

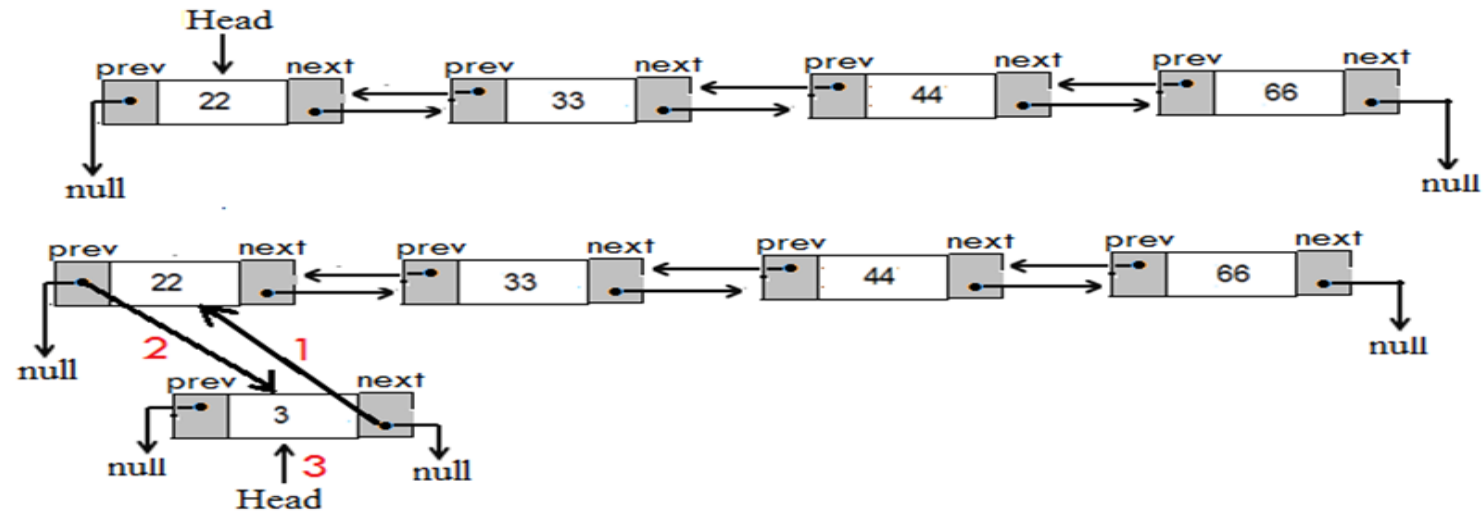
Advantages:

- Can be traversed in either direction
- Some operations, such as deletion and inserting before a node, become easier

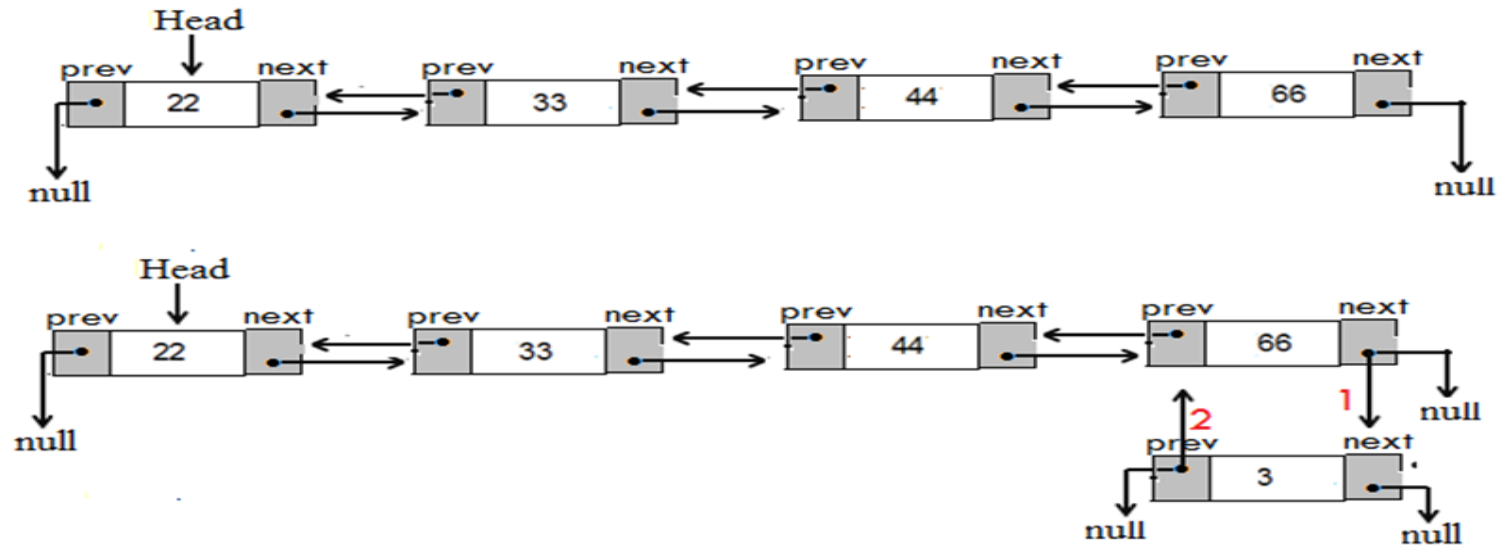
Disadvantages:

- Requires more space
- List manipulations are slower (because more links must be changed)
- Greater chance of having bugs (because more links must be manipulated)

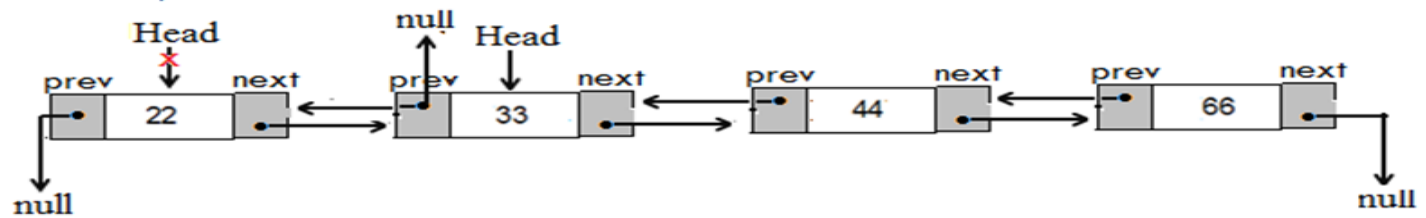
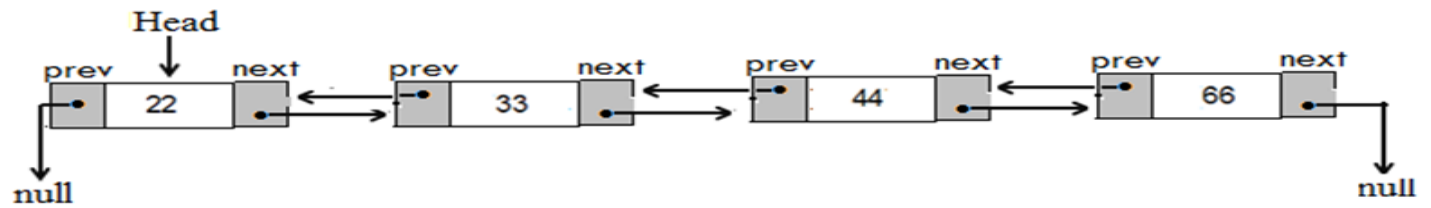
insert in the first position



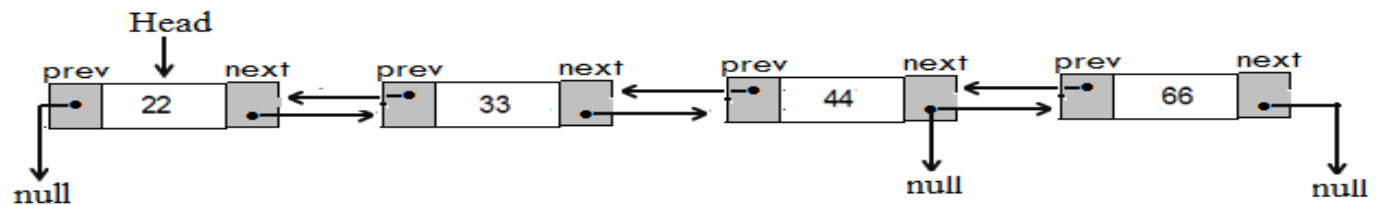
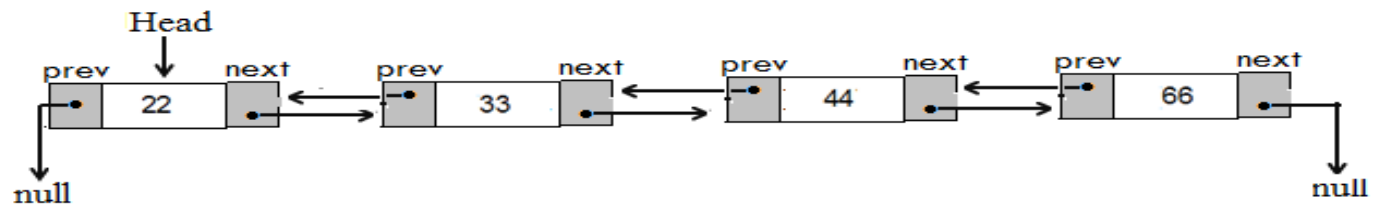
insert in last position



delete first element



delete last element



H.W. Write code segments in java for the following operations:

- 1.Add an element at the beginning of the list.
- 2.Add an element in the end of the list.
- 3.Add an element after an item of the list.
- 4.Add an element before an item of the list.
- 5.Delete the element at the beginning of the list.
- 6.Delete the element from the end of the list.

Circular linked lists

in this type of linked list:

Last node references the first node

Every node has a successor

No node in a circular linked list contains *null*

Both singly linked list and doubly linked list can be made into as circular linked list:

Singly linked list as circular, the next pointer of the last node points to the first node.

Doubly linked list as circular, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.

To construct circular linked list we need define two class:

1. node class
2. circular linked list class(CLL)

```

class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
    }
}

```

```

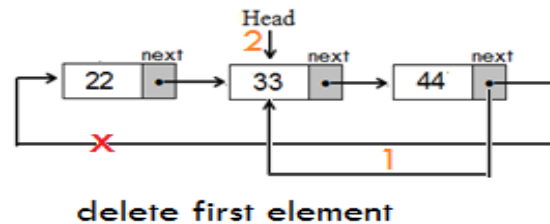
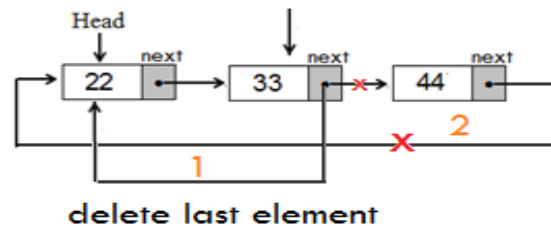
class CLL {
    Node head;

    public CLL() {
        this.head = null;
    }

    // methods
    ...
}

```

Delete



H.W. Write code segments in java for the following operations:

- Add an element at the beginning of the list.
- Add an element in the end of the list.
- Add an element after an item of the list.
- Add an element before an item of the list.
- Delete the element at the beginning of the list.
- Delete the element from the end of the list.

Single linked list:

```
public class Node {  
    int data;  
    Node next;  
  
    Node(int data){  
        this.data=data;  
        next=null;}  
}
```

```
public class SingleLL {
```

```
    Node head;
```

```
    SingleLL(){
```

```
        head=null;
```

```
    }
```

```
void printList(){
```

```
    for (Node p=head; p!=null; p=p.next ) {
```

```
        System.out.print(p.data+" ");
```

```
    }
```

```
    System.out.println();
```

```
}
```

```
void addItem( int data){
```

```
    Node p= new Node(data);
```

```
    Node c=head;
```

```
    if (head==null)
```

```
        head=p;
```

```
    else{
```

```
        for (c=head; c.next!=null; c=c.next)
```

```
            c.next=p;}
```

```
}
```

```
void addBefore(int data, int item){
```

```
    Node p=new Node(data);
```

```
    Node cur=head;
```

```
    Node prv=head;
```

```
    if (head.data==item){
```

```
        p.next=head;
```

```
        head=p;}
```

```
    else{
```

```
        while (cur.data!=item && cur.next!=null){
```

```
            prv=cur;
```

```
            cur=cur.next; }
```

```
    if (cur.next==null)
```

```
        addItem(data);
```

```
    else{
```

```
        p.next=cur;
```

```
        prv.next=p;}
```

```
    }
```

```
}
```

```

void addAfter(int data, int item){
    Node p=new Node(data);
    Node cur=head;
    while (cur.data!=item &&
cur.next!=null)
        cur=cur.next;
    if (cur.next==null) addItem(data);
    else {
        p.next=cur.next;
        cur.next=p;}
}

```

```

void deleteItem(int item){
    Node cur=head;
    Node prv=head;
    if (head.data==item)
        head=cur.next;
    else{
        while (cur.data!=item &&
cur.next!=null){
            prv=cur;
            cur=cur.next;
        }
        if (cur.next==null &&
cur.data==item) prv.next=null;
        else if (cur.next==null)
            System.out.println(item+ "
not found");
        else
            prv.next=cur.next;
        }
    }
}

```

```

void printDLL(){
    for (DNode p=head; p!=null; p=p.next)
        System.out.print(p.data + " ");
    System.out.println();
}

```

```

void RprintDLL(){
    DNode p;
    for (p=head; p.next!=null; p=p.next){}
    for (; p!=null; p=p.prev)
        System.out.print(p.data + " ");
    System.out.println();
}

```

```

void addItem(int data){
    DNode p = new DNode(data);
    DNode c;
    if (head==null)
        head=p;
    else{
        for (c=head; c.next!=null; c=c.next){}
        c.next=p;
        p.prev=c;}
}

```

Double linked list

```

public class DNode {
    int data;
    DNode next, prev;
}

```

```

DNode(int data){
    this.data=data;
    next = null;
    prev = null; }
}

```

```

public class DoubleLL {
    DNode head=null;
    DoubleLL(){
        head=null;
    }
}

```

```

void addAfter(int data, int item){
    DNode p= new DNode(data);
    DNode c=head;
    while (c.data!=item && c.next!=null)
        c=c.next;
    if (c.next==null && c.data!=item)
        System.out.println(item + " not
found");
    else if (c.next==null)
        addItem(data);
    else{
        p.next=c.next;
        c.next.prev=p;
        c.next=p;
        p.prev=c;}
}

```

```

void addBefore(int data, int item){
    DNode p=new DNode(data);
    DNode c=head;
    if (head.data==item){
        p.next=head;
        head.prev=p;
        head=p;}
    else {
        while (c.data!=item &&
c.next!=null)
            c=c.next;
        if (c.data!=item && c.next==null)
            System.out.println(item + " not
found");

        else{
            p.next=c;
            DNode q=c.prev;
            q.next=p;
            p.prev=q;
            c.prev=p;}
    }
}

```



```
void deleteItem(int item){
```

```
    DNode p=head;
```

```
    if (head.data==item){
```

```
        head.next.prev=null;
```

```
        head=head.next;
```

```
    }
```

```
    else {
```

```
        while (p.data!=item && p.next!=null)
```

```
            p=p.next;
```

```
        if (p.data!=item && p.next==null)
```

```
            System.out.println(item + " not  
found");
```

```
        else if (p.data==item && p.next==null){
```

```
            DNode q=p.prev;
```

```
            q.next=null;}
```

```
        else {
```

```
            DNode q=p.prev;
```

```
            q.next=p.next;
```

```
            p.next.prev=q;
```

```
        }
```

```
    }
```

```
circular linked list
```

```
public class CirclLL {
```

```
    Node head;
```

```
    CirclLL(){
```

```
        head=null;    }
```

```
void printList(){
```

```
    Node p;
```

```
    for (p=head; p.next!=head; p=p.next ) {
```

```
        System.out.print(p.data+" ");
```

```
    }
```

```
    System.out.println(p.data+"\n");}
```

```
void addItem( int data){
```

```
    Node p= new Node(data);
```

```
    Node c=head;
```

```
    if (head==null){
```

```
        head=p;
```

```
        p.next=head;}
```

```
    else{
```

```
        for (c=head; c.next!=head; c=c.next){}
```

```
        c.next=p;
```

```
        p.next=head;}
```

```
void addBefore(int data, int item){
```

```
    Node p=new Node(data);
```

```
    Node cur=head;
```

```
    Node prv=head;
```

```
    if (head.data==item){
```

```
        for (cur=head; cur.next!=head;
```

```
            cur=cur.next ) {}
```

```
        p.next=head;
```

```
        head=p;
```

```
        cur.next=head;}
```

```
    else{
```

```
        while (cur.data!=item && cur.next!=head){
```

```
            prv=cur;
```

```
            cur=cur.next; }
```

```
        if ( cur.next==head && cur.data==item){
```

```
            p.next=cur;
```

```
            prv.next=p;}
```

```
        else if (cur.next==head)
```

```
            addItem(data);
```

```
        else{
```

```
            p.next=cur;
```

```
            prv.next=p;}
```

```
    }
```

```
}
```

```
void addAfter(int data, int item){
```

```
    Node p=new Node(data);
```

```
    Node cur=head;
```

```
    while (cur.data!=item && cur.next!=null)
```

```
        cur=cur.next;
```

```
    if (cur.next==head) addItem(data);
```

```
    else {
```

```
        p.next=cur.next;
```

```
        cur.next=p;}    }
```

```
void deleteItem(int item){
```

```
    Node cur=head;
```

```
    Node prv=head;
```

```
    if (head.data==item){
```

```
        for (cur=head; cur.next!=head; cur=cur.next ) {
```

```
            head=head.next;
```

```
            cur.next=head;}
```

```
    else{
```

```
        while (cur.data!=item && cur.next!=head){
```

```
            prv=cur;
```

```
            cur=cur.next;        }
```

```
        if (cur.next==head && cur.data==item)
```

```
            prv.next=head;
```

```
        else if (cur.next==head)
```

```
            System.out.println(item+ " not found");
```

```
        else
```

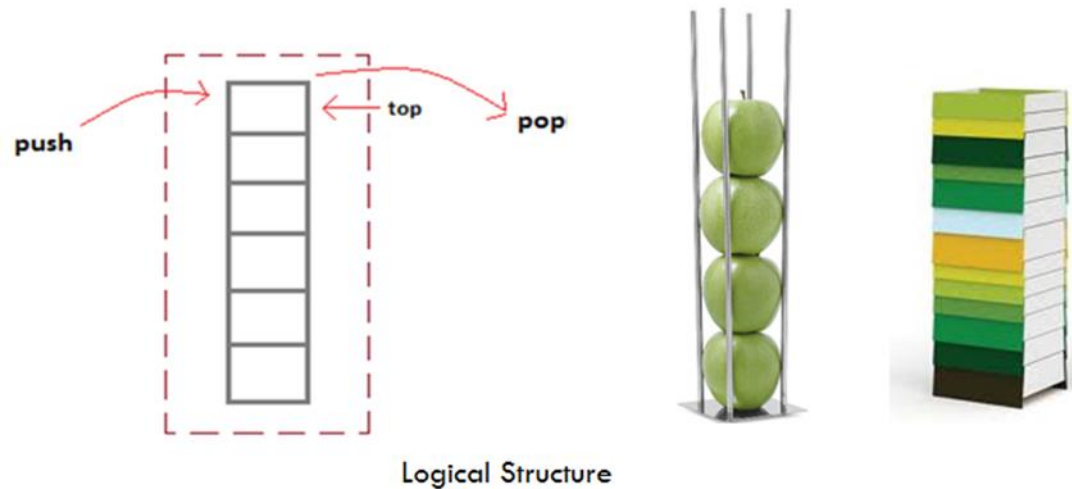
```
            prv.next=cur.next;    }    }
```

Chapter Four

Stacks

What is the Stack?

- Stack is a dynamic linear data structure.
- Data in a stack are added and removed from only one end of the list.



- We define a stack as a list of homogeneous items that are accessible only from the end of the list, which is called the top of the stack.
- Elements are always removed from the top, and inserted on the top also.
- A stack is known as a Last-in, First-out (LIFO) data structure

Basic Features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. There are two operation : push function is used to insert new elements into the Stack and pop is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Storage Structure

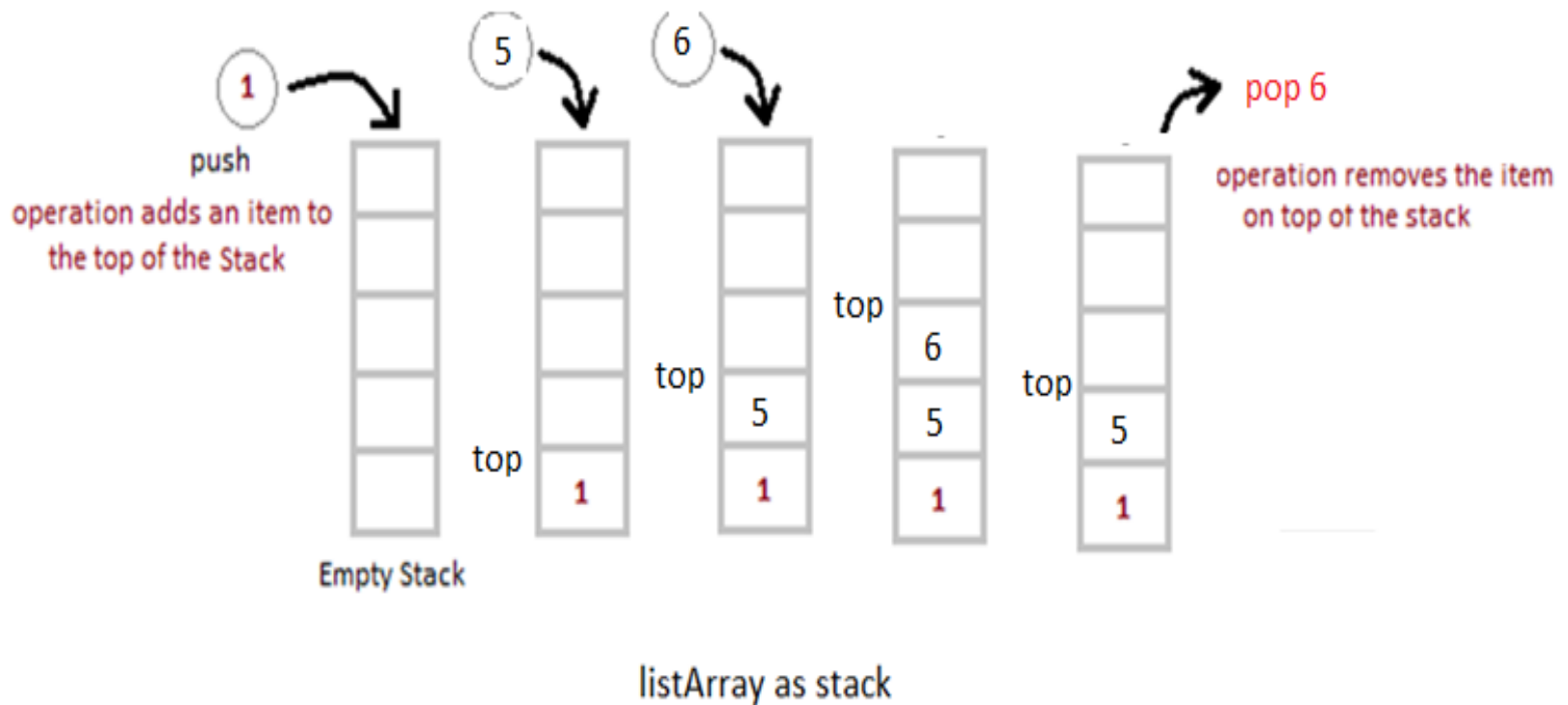
Storage structure depends on the implementation of stack, array or linked list structure.

Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link but is not limited in size.

Implementation of stack using Array

The following figure shows implementation for stack using array:



In the following stack class:

```
class stack{
    int top;
    int listArray[]=new int[10]; //Maximum size of Stack
    stack() {
        top = -1;}

    void push(int x){
        if ( top >= 10)
            System.out.println( "Stack Overflow");
        else{
            top++;
            listArray[top] = x;
            System.out.println( "Element Inserted");}
    }

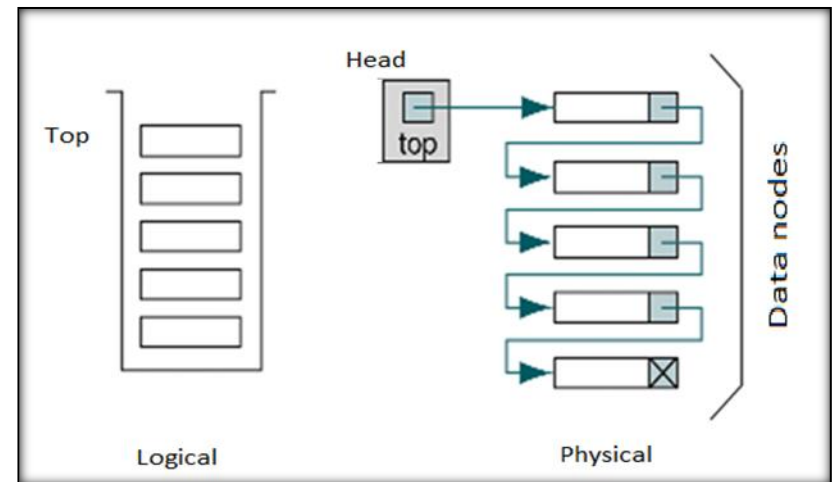
    int pop(){
        if (top < 0) {
            System.out.println( "Stack Underflow");
            return 0; }
        else {
            int d = listArray[top];
            top--;
            return d; }
    }
}
```

Stack implementation using a linked list

The following figure shows implementation for stack using linked list:

In the following stack class:

```
class StackLinkedList {  
  
    node top = null;  
  
    void push(int data) {  
        node p = new node(data);  
        if (top == null)  
            top = p;  
        else{  
            p.next = top;  
            top = p; }  
    }  
}
```



```
node pop() {  
    if (top == null){  
        System.out.println( " Stack  
is Empty.... ");  
        return top;}  
    else{  
        node p = top;  
        top = top.next;  
        return p;}  
}
```

```
void peek(){  
    if (top == null)  
        System.out.println("The  
Stack is Empty....");  
    else  
        System.out.println  
(top.data);}  
  
void clear() {  
    if (top == null)  
        System.out.println("The Stack is  
Empty....");  
    else  
        top = null; }
```

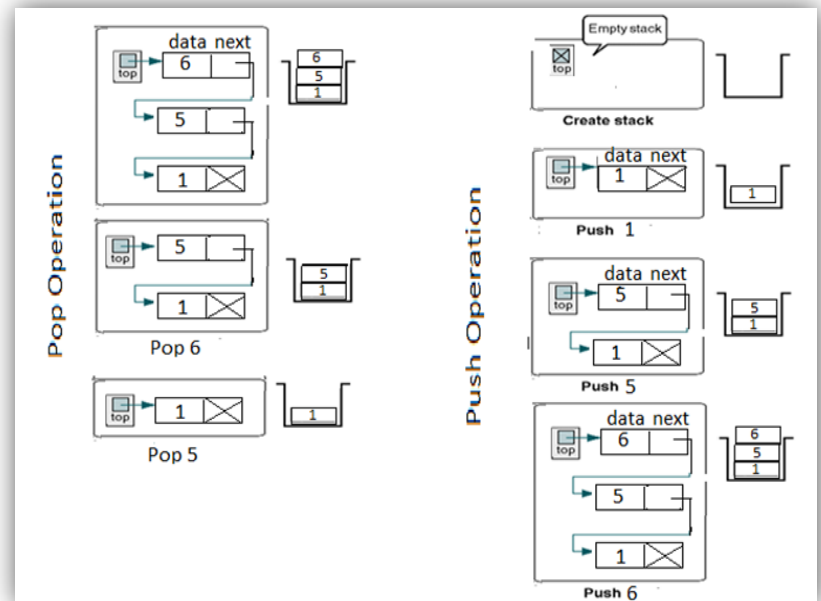
```

void displayStack() {

    node current = top;
    while (current != null) {
        System.out.print(current.data);
        System.out.print(" ");
        current = current.next;
    }
}

```

Example:



Applications of Stack

1. **Reverse a word**: You push a given word to stack - letter by letter - and then pop letters from the stack.
2. **Expression Conversion and evaluating expressions.**
3. **Call subprogram and recursion processing**

Converting and Evaluating Expressions

Arithmetical operations like addition, subtraction, multiplication, and division are called **binary operations** because they each combine two operands:

Operand operator operand

There are three type for operator notations : **Infix, prefix and postfix** (also called *reverse Polish notation, or RPN*) . for example consider the simple binary operation **a + b** Equivalent prefix and postfix forms are shown below:.

Prefix: + a b

operator first

Postfix: a b +

operator last

Notes:

1. **Postfix** expressions are **easier to process by machine** than are **infix** expressions. and it used in stack to evaluate the expressions.
2. Each operator has precedence as shown in the following table.

Operator	Precedence
() Exponentiation ^	Highest ↓ Lowest
Multiplication (*) & Division (/)	
Addition (+) & Subtraction (-)	

Convert Infix to Postfix Algorithm (in case expression NOT contain parentheses)

Step 1: For each term in expression

Step 2: If term is an operator

 Compare it with the operator on the top, if have the same or higher precedence , Pop it, otherwise Push this operator into stack

 Else Copy operand to output

 end if

Step 3: Pop remaining operators and copy to output

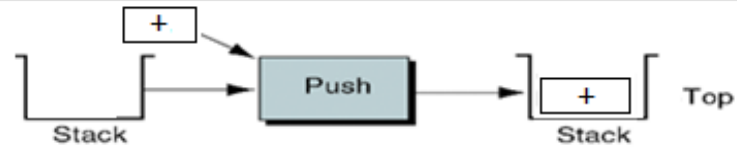
Example 1: By using stack data structure convert Infix expression $a + b / c$ to Postfix expression

Step 1

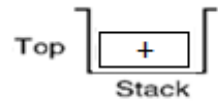


Output: a

Step 2

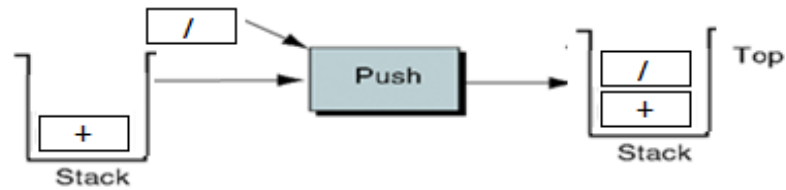


Step 3



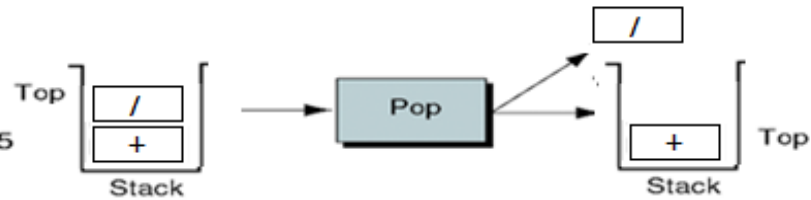
Output: ab

Step 4



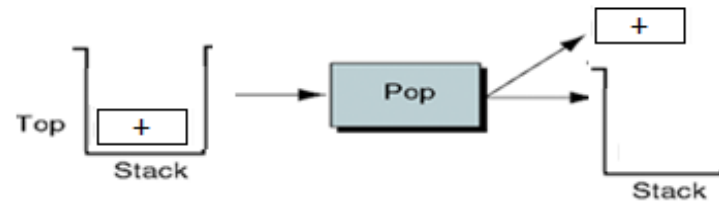
Output: abc

Step 5

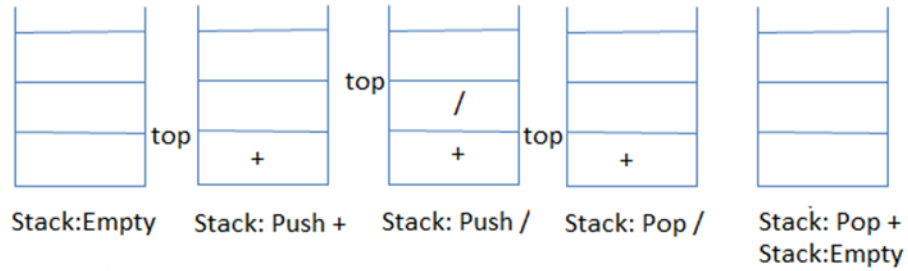


Output: abc /

Step 6

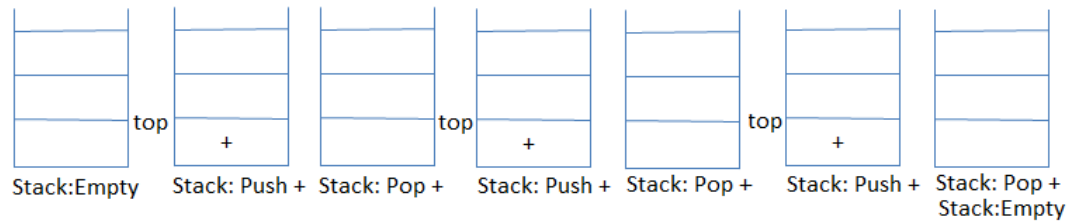


Output: abc / +



Expression	Stack Operator	Output (RPN)	Action
a+b/c	Empty	-	
+b/c	Empty	a	
b/c	+	a	Push +
/c	+	ab	
c	+/	ab	Push /
Empty	+/	abc	
Empty	+	abc/	Pop /
Empty	Empty	abc/+	Pop +

Example 2: : By using stack data structure convert Infix expression $a + b + c + d$ to Postfix expression.



Output : $ab+c+d+$

Expression	Stack Operator	Output (RPN)	Action
$a+b+c+d$	Empty	-	
$+b+c+d$	Empty	a	
$b+c+d$	+	a	Push +
$+c+d$	+	ab	
$+c+d$	Empty	ab+	Pop +
$c+d$	+	ab+	
$+d$	+	ab+c	
$+d$	Empty	ab+c+	Pop +
d	+	ab+c+	
Empty	+	ab+c+d	
Empty	Empty	ab+c+d+	Pop +

Exercises: Convert these infix expressions to postfix expressions:

1. $a + b * c * d + e$

2. $a * b + c * d * e * f$

3. $a / b / c + d * e * f$

4. $a + b * c^d / e - f * g$

5. $a - b + c * d / e$

Algorithm to Convert an infix expression to postfix notation(in case expression contain parentheses)

Suppose Q is an arithmetic expression(contain parentheses) in infix notation. We will create an equivalent postfix expression P by adding items to on the right of P.

Start with an empty stack. We scan Q from left to right.

While (we have not reached the end of Q)

 If (an operand is found)

 Add it to P

 end if

 If (a left parenthesis is found)

 Push it onto the stack

 end if

 If (a right parenthesis is found)

 While (the stack is not empty AND the top item is not a left
 parenthesis)

 Pop the stack and add the popped value to P

 end while

 Pop the left parenthesis from the stack and discard it

End-If

If (an operator is found)

 If (the stack is empty or if the top element is a left parenthesis)

 Push the operator onto the stack

 Else

 While (the stack is not empty AND the top of the stack
 is not a left parenthesis AND precedence of the
 operator \leq precedence of the top of the stack)

 Pop the stack and add the top value to P

 End-While

 Push the latest operator onto the stack

 End-If

End-If

End-While

 While (the stack is not empty)

 Pop the stack and add the popped value to P

 End-While

Note : At the end, if there is still a left parenthesis at the top of the stack, or if we find a right parenthesis when the stack is empty, then Q contained unbalanced parentheses and is in error.

Example 3 : Convert the infix expressions to postfix:(2+3)*4

Expression	Stack Operator	Output (RPN)	Action
(2+3)*4	Empty	-	
2+3)*4	(-	Push (
+3)*4	(2	
3)*4	(+	2	Push +
)*4	(+	2 3	
*4	(2 3 +	Pop +
*4	Empty	2 3 +	Pop (
4	*	2 3 +	
Empty	*	2 3 + 4	
Empty	Empty	2 3 + 4 *	Pop *

Example 4 : Convert the infix expressions to postfix: $2+(3*4)$

Expression	Stack Operator	Output (RPN)	Action
$2+(3*4)$	Empty	-	
$+(3*4)$	Empty	2	
$(3*4)$	+	2	Push +
$3*4)$	+(2	Push(
$*4)$	+(2 3	
$4)$	+(*	2 3	Push*
$)$	+(*	2 3 4	
Empty	+(*	2 3 4 *	Pop *
Empty	+(2 3 4 *	Pop (
Empty	Empty	2 3 4 * +	Pop +

Example 5: Convert the infix expressions to postfix: $(3*2+4)^2$

Expression	Stack Operator	Output (RPN)	Action
$(3*2+4)^2$	Empty	-	
$3*2+4)^2$	(-	Push(
$*2+4)^2$	(3	
$*2+4)^2$	(*	3	Push *
$+4)^2$	(*	3 2	
$+4)^2$	(3 2 *	Pop *
$4)^2$	(+	3 2 *	Push +
$)^2$	(+	3 2 * 4	
2	(3 2 * 4 +	Pop +
2	Empty	3 2 * 4 +	Pop (
2	^	3 2 * 4 +	Push ^
Empty	^	3 2 * 4 + 2	
Empty	Empty	3 2 * 4 + 2 ^	Pop ^

Algorithm to Evaluate a postfix expression

Suppose P is an arithmetic expression (contain parentheses)in postfix notation. We will evaluate it using a stack to hold the operands.

Start with an empty stack. We scan P from left to right.

While (we have not reached the end of P)

 If an operand is found

 push it onto the stack

 End-If

 If an operator is found

 Pop the stack and call the value A

 Pop the stack and call the value B

 Evaluate $B \text{ op } A$ using the operator just found.

 Push the resulting value onto the stack

 End-If

End-While

Pop the stack (this is the final value)

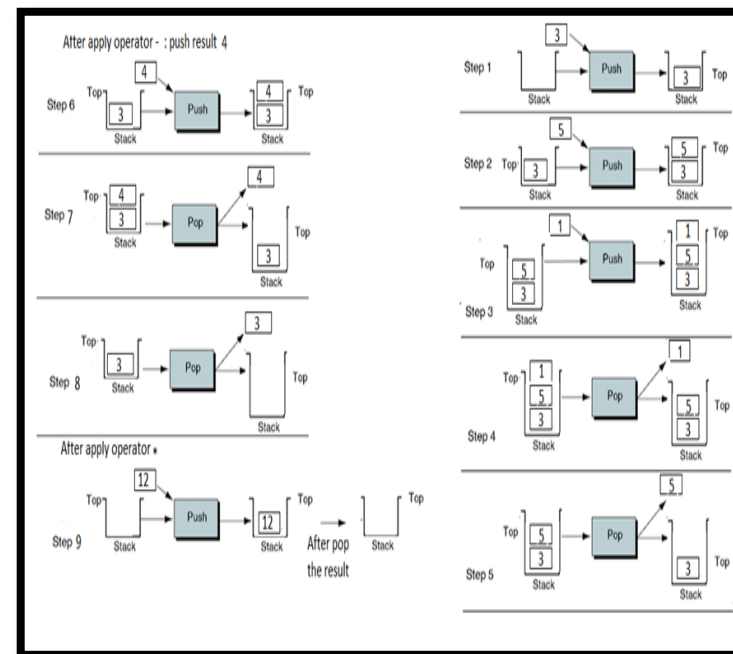
Notes:

At the end, there should be only one element left on the stack.

This assumes the postfix expression is valid.

Example 6 : consider the postfix expression : 3 5 1 - *

Expression	Execute Stack	Action
3 5 1 - *	Empty	
5 1 - *	3	Push 3
1 - *	3 5	Push 5
- *	3 5 1	Push 1
*	3 4	Pop 1 and 5, Execute $5 - 1 = 4$, Push 4
Empty	12	Execute $3 * 4 = 12$, Push 12
Empty	Empty	Pop the result 12



Example 7 : consider the postfix expression : 5 4 + 3 / 1 6 * +

Expression	Execute stack	Action
5 4 + 3 / 1 6 * +	Empty	
4 + 3 / 1 6 * +	5	Push 5
+ 3 / 1 6 * +	5 4	Push 4
3 / 1 6 * +	9	Pop 5 and 4, Execute $5+4=9$, Push 9
1 6 * +	9 3	Pop 9 and 3, Execute $9/3=3$, Push 3
1 6 * +	3	Pop 6 and 1, Execute $6/1=6$, Push 6
6 * +	3 1	
* +	3 1 6	Pop 1 and 6, Execute $6*1=6$, Push 6
+	3 6	Pop 3 and 6, Execute $3+6=9$, Push 9
Empty	9	
Empty	Empty	Pop the result 9

Example8 : Evaluate the postfix expression : 3 2 * 4 + 2 ^

Expression	Execute stack	Action
3 2 * 4 + 2 ^	Empty	
2 * 4 + 2 ^	3	Push 3
* 4 + 2 ^	3 2	Push 2
4 + 2 ^	6	Pop 3 and 2, Execute $2*3=6$, Push 6
+ 2 ^	6 4	Push 4
2 ^	10	Pop 6 and 4, Execute $6+4=10$, Push 10
2 ^	10 2	Push 2
^	100	Pop 10 and 2, Execute $10^2=100$, Push 100
Empty	Empty	Pop the result 100

Example9 : Evaluate the postfix expression : 2 3 4 * + 2 ^

Expression	Execute stack	Action
2 3 4 * + 2 ^	Empty	
3 4 * + 2 ^	2	Push 2
4 * + 2 ^	2 3	Push 3
* + 2 ^	2 3 4	Push 4
+ 2 ^	2	Pop 3 and 4, Execute $3*4=12$, Push 12
+ 2 ^	2 12	
2 ^	14	Pop 2 and 12, Execute $2+12=14$, Push 14
^	14 2	Push 2
Empty	196	Pop 14 and 2, Execute $14^2=196$, Push 196
Empty	Empty	Pop the result 196

Exercises: Convert the following infix expressions to postfix then evaluates these postfix expressions, giving the stack contents after each step:

1. $a/b+c/d$
2. $(a + b) * (c + d)$
3. $((a + b) * c) - d$
4. $(80 - 30) * (40 + 50 / 10)$
5. $(a + b) - (c / (d + e))$
6. $a / ((b / c) * (d - e))$
7. $(a / (b / c)) * (d - e)$
8. $a * b + c) / d - e)$
9. $(a - b) / (c * (d + e))$
10. $a / (b + (c * (d - e)))$
11. $((2 * 5) - (1 * 2)) / (11 - 9)$
12. $(2 * 5 - 1 * 2) / (11 - 9)$
13. $A + B * C / D - E$
14. $A + B * (C - D) / E$