ATME College of Engineering

Course Name: Data Structures and Applications

Course Code: BCS304

Course Learning Objectives

- Explain fundamentals of data structures and their applications essential for programming / problem solving.
- Illustrate linear representation of data structures: Stack, Queues, Lists, Trees and Graphs.
- Demonstrate sorting and searching algorithms.
- Find suitable data structure during application development/Problem Solving.

Course Outcomes

- Use different types of data structures, operations and algorithms
- Apply searching and sorting operations on files
- Use stack, Queue, Lists, Trees and Graphs in problem solving
- Implement all data structures in a high-level language for problem solving.

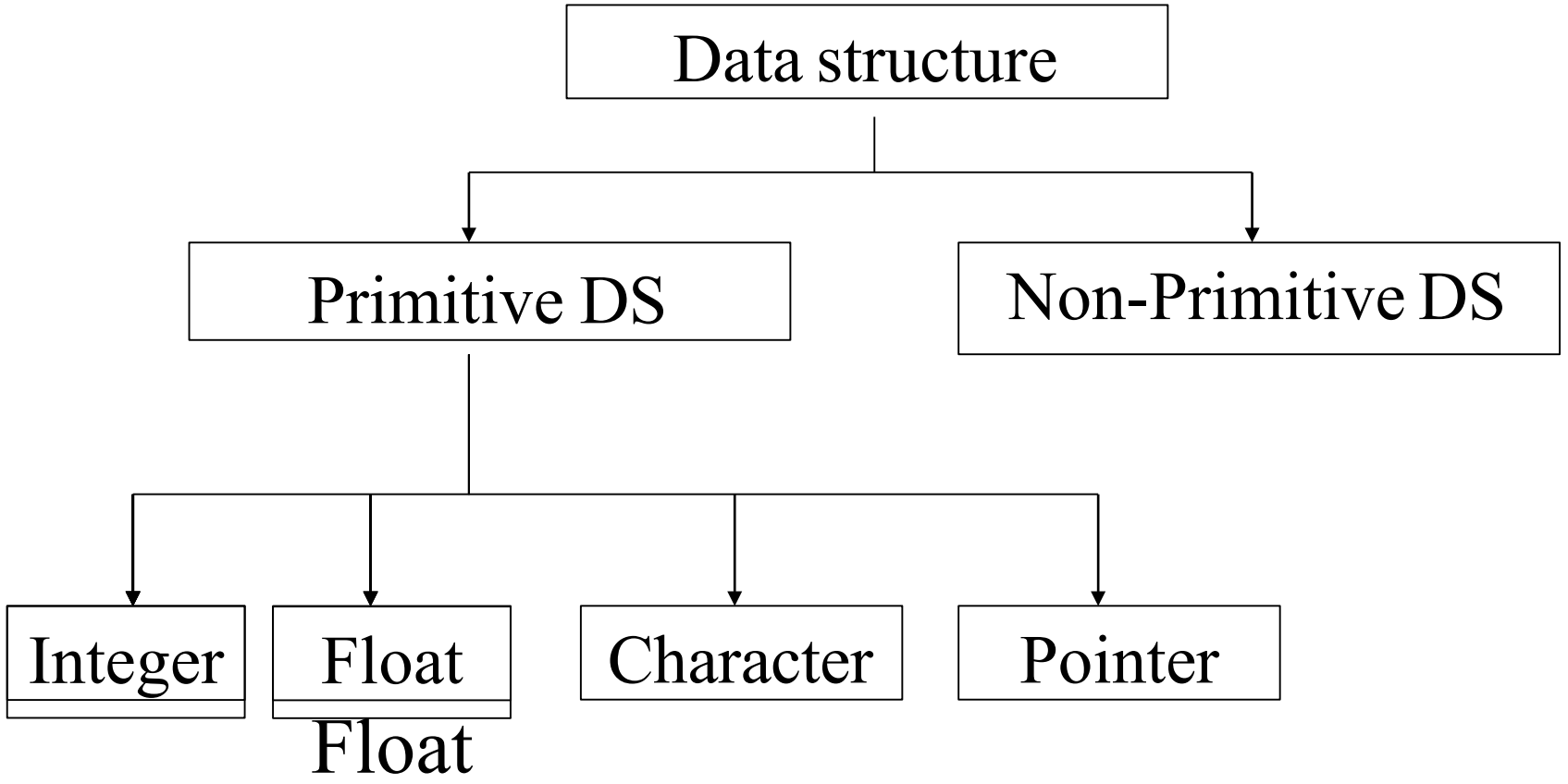
Introduction

- Data structure affects the design of both structural & functional aspects of a program.

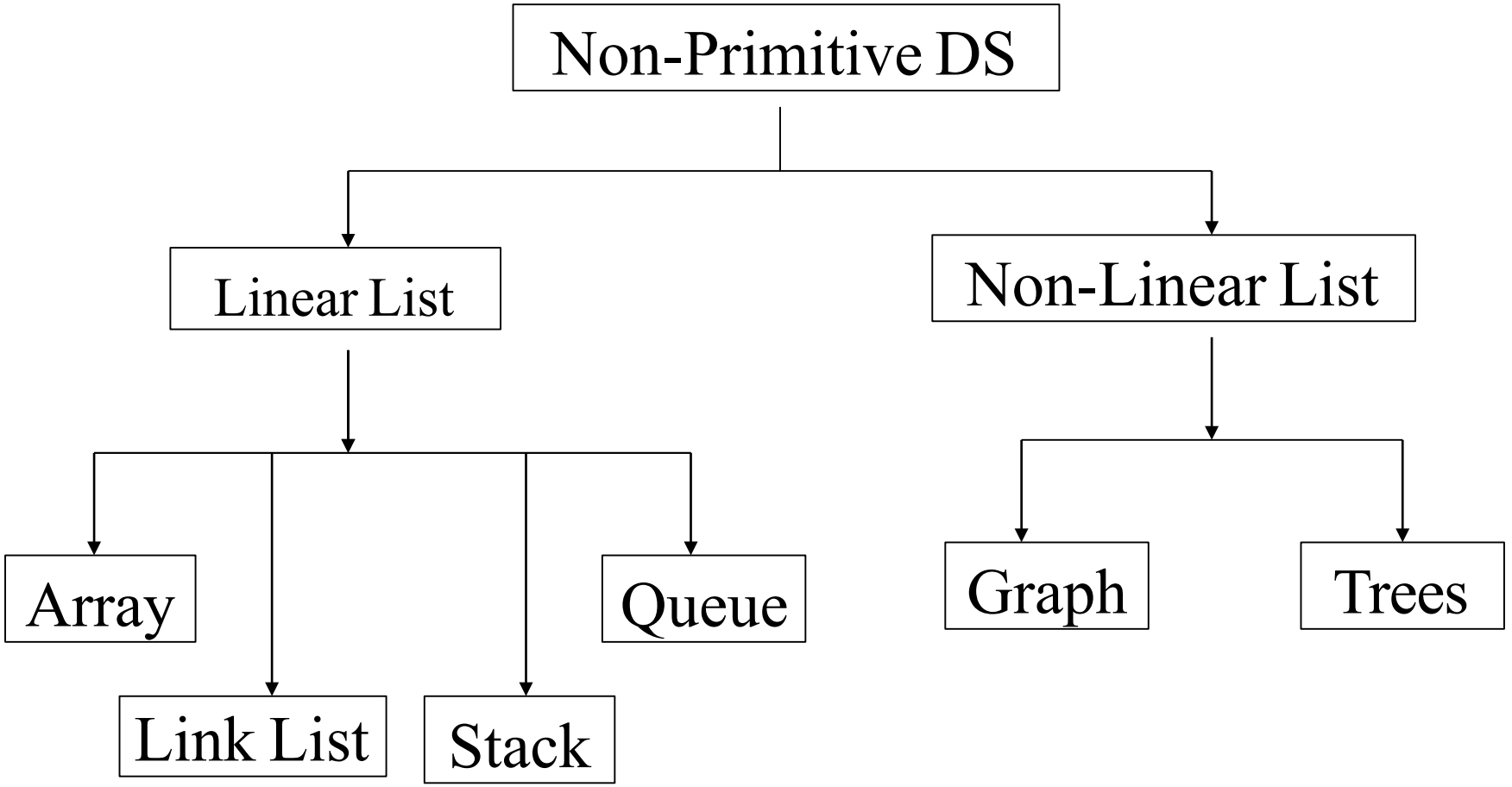
Program=algorithm + Data Structure

- Algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

Classification of Data Structure



Cont...


Non-Primitive DS
Linear List
Non-Linear List
Array
Link List
Stack
Queue
Graph
Trees

Linear Data Structure

- A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.
  - One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.
  - The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.
- The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

Non-linear Data Structure

- A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.
- Based on the memory allocation, non-primitive data structures is further classified into
  - Static Data Structure
  - Dynamic Data Structure

Static Data Structure

- In these type of data structure, the memory is allocated at compile time. Therefore maximum size is fixed.
- Advantages: Fast access
- Disadvantages: Slower insertion and deletion
- Example: Array

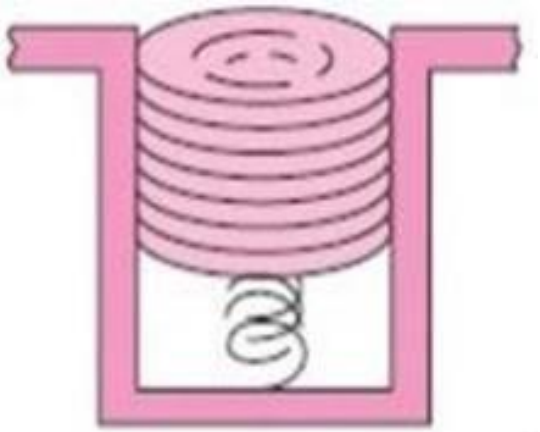
Dynamic Data Structure

- In these type of data structure, the memory is allocated at run time. Therefore maximum size is flexible.
- Advantages: Slower access
- Disadvantages: Faster insertion and deletion
- Example: Linked list

Arrays

	STUDENT
1	John Brown
2	Sandra Gold
3	Tom Jones
4	June Kelly
5	Mary Reed
6	Alan Smith

Stack

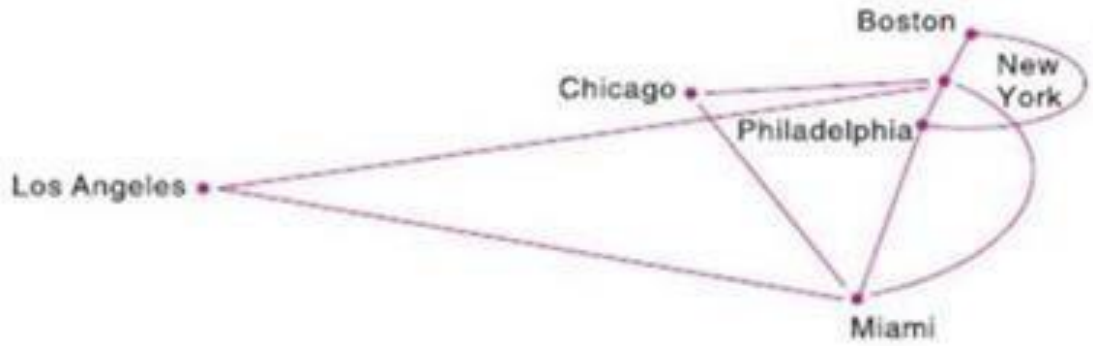


(a) Stack of dishes

Queue

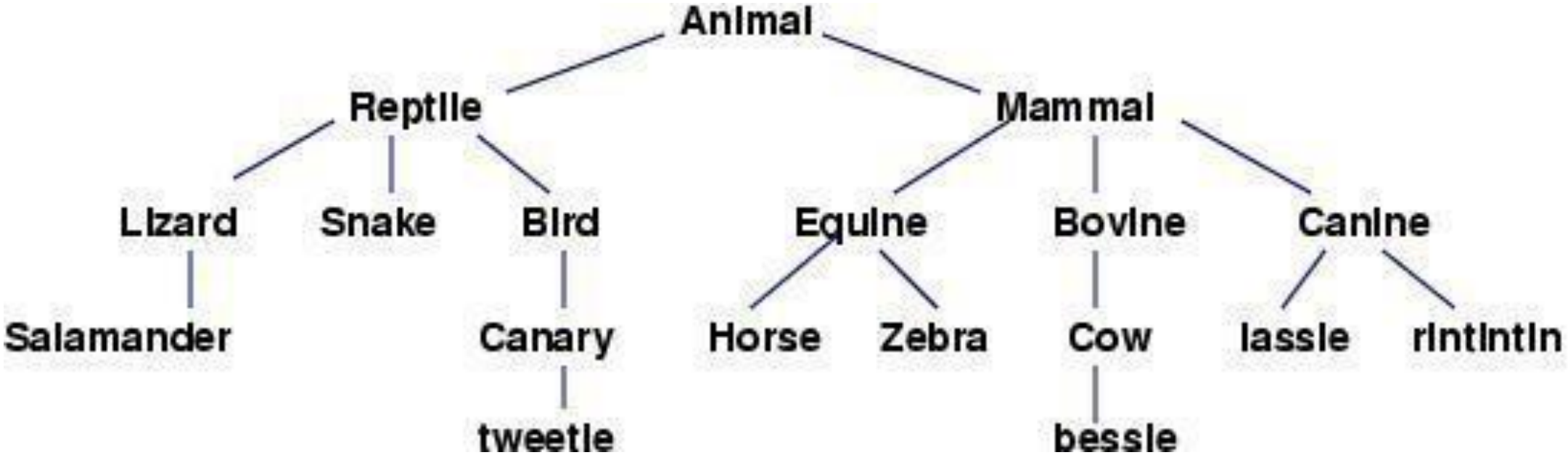
Queue waiting for a bus

Graph



Airline flights

Tree


Animal
Reptile
Mammal
Lizard
Snake
Bird
Equine
Bovine
Canine
Salamander
Canary
Horse
Zebra
Cow
lassie
rintintin
tweetie
bessie

Operations performed on DS

- **Traversing or Visiting: -** Accessing or visiting of each data element present in data structure is known as Traversing.
- **Insertion: -** To add a new element in a given data structure is known as Insertion.
- **Deletion: -** To remove an element from a given data structure is known as Deletion.
- **Searching: -** to find a given item into data structure is known as Searching. When searching is successful then it will provide location or position of data element .Otherwise, it will return zero.
- **Sorting:-** To arrange data elements in ascending or descending order on the basis of alphabet or number is known as Sorting
- **Merging: -** merging is a process of combining the data items of two or more different data structure into a new data structure of same type. the memory size of resultant data structure is the sum of memory sizes of all combined data structures

Structures

A structure (a record) is a collection of data items, where each item is identified as to its type and name.

Syntax: struct

```
{       data_type member 1;
        data_type member 2;
        ……….
        ……….
        data_type  member n;
} variable_name;
```

Ex:

```
struct {
 char name[10];
 int age;
float salary;
} Person;
```

Self-Referential Structures

- Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Example:

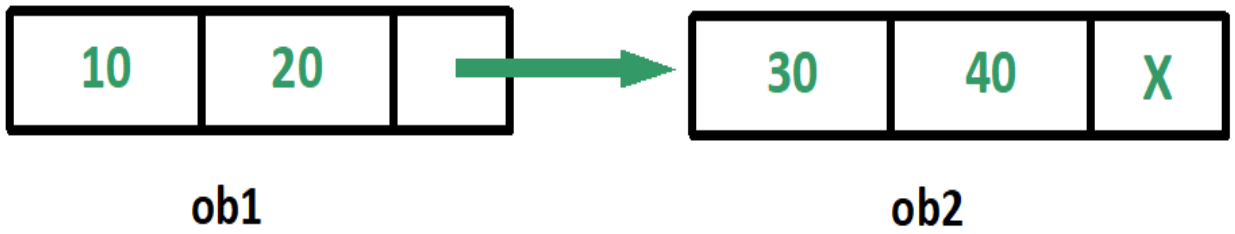
```
struct node {
        int data1;
        char data2;
        struct node* link;
         };
int main()
{
         struct node ob;
         return 0;
}
```

Types of Self Referential Structures

- Self Referential Structure with Single Link
- Self Referential Structure with Multiple Links

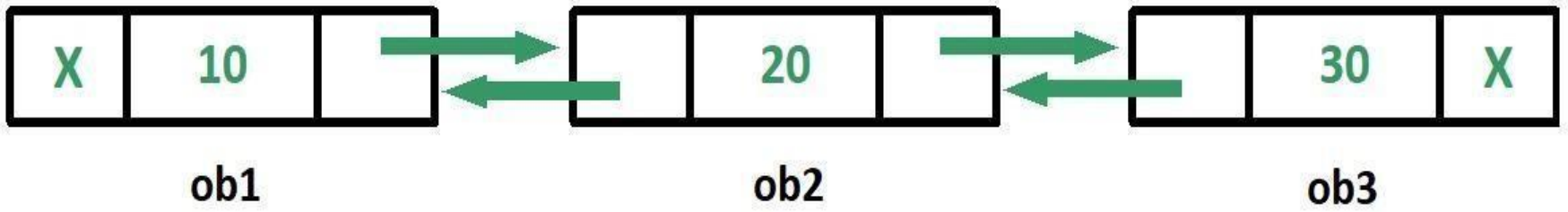
Self Referential Structure with Single Link

These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members.



Self Referential Structure with Multiple Links

- Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one node at a time. The following example shows one such structure with more than one links.



Applications

Self referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

Unions

- A union is similar to a structure, it is collection of data similar data type or dissimilar. But only one field is active.

```
typedef union human_being
{
        char name[10];
        int age;
        float salary;
        date dob;
        sex_type sex_info;
}h1;
```

h1.name="swetha"; h1.salary=20000; h1.age=20;

Difference between Structure and Union

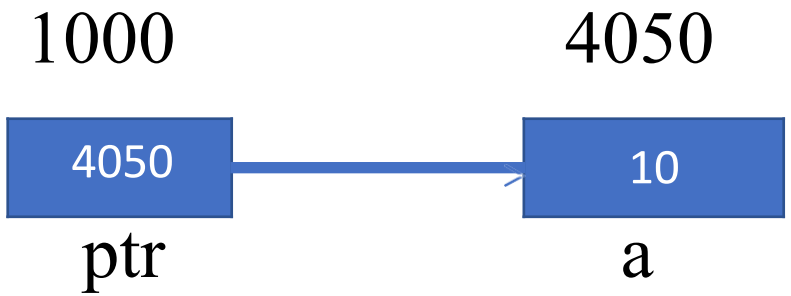
	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Pointers

- A pointer is a variable which contains the address in memory of another variable.
- The two most important operator used with the pointer type are
- & - The unary operator ***&*** which gives the address of a variable
- * - The indirection or dereference operator * gives the content of the object pointed to by a pointer.

Pointer Declaration and Initialization

```
int *ptr,a=10;
  ptr=&a;
```



Identify the output of the following statements

- printf("%d\n",a);
- printf("%d",&a);
- printf("%d",ptr);
- printf("%d",*ptr);

Pointers and Functions

```
void swap(int a,int b)
{
int temp;
temp=a;
a=b;  b=temp;
}
void main()
{
int x=10,y=20;
swap(x,y);
printf("after swapping x=%d\t  y=%d\n",x,y);
}
```

- It is used in functions when the parameters are by reference
- Consider the example of swapping

Pointers and arrays

- The operations performed using array can also be done using pointers
- Syntax

  data_type *ptr,array[size];

  ptr=&array; or ptr=a;

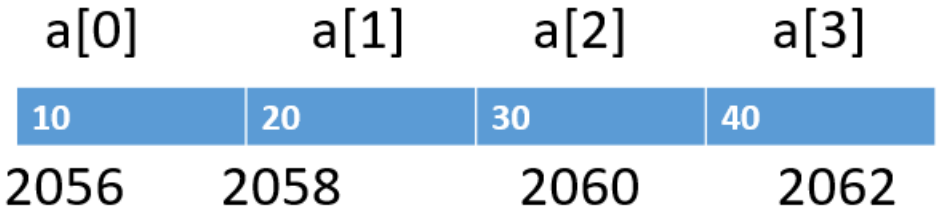
- Example:

  int *ptr, a[20]={11,12,13,14};

  ptr=a; //ptr points to address of a[0] location i.e. base address

Program

```
#include<stdio.h>
void main()
{
   int a[10]={10,20,30,40},*ptr;
   ptr=&a;
   for(int i=0;i<4;i++)
   {
            printf("%d",a[i]);
            printf("%d",&a[i]);
            printf("%d",*ptr);
            printf("%d",ptr);
   }
}
```



Output ????

Pointers to Strings

- The pointer points to the first character of the string
- Syntax

```
data_type     *ptr,str;
ptr=str;
```

- Example

```
char *char_ptr,str;
str="DSA";
char_ptr=str;
```

Program

```
Void main()
{
        int i;
        char str1[20]="Data structure";
        char str2[20],*ptr;
        ptr=str1;
        for(i=0;str1[i]!='\0';i++)
        {
                str2[i]=*ptr;
                ptr++;
        }
        str2[i]='\0';
        printf("String 2 after copying is %s\n",str2);
}
```

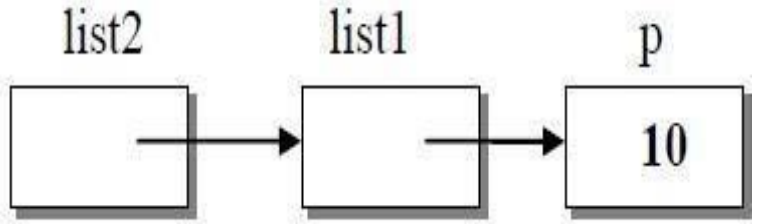
OUTPUT ???

Pointer to Pointer or Double Pointer

- A variable which contains address of a pointer variable is called pointer-to-pointer

Example:

```
int p;
int *list1, * *list2;
p=10;
list1=&p;
list2=&list1;
printf("%d, %d, %d", a, *list1, **list2);
```



Output: 10 10 10

Memory Allocation

- Definition:

  reserving/allocating memory to the data used by a program using variables

- Types of allocation
  - **Static Memory allocation** : allocating memory at compilation time
  - **Dynamic memory allocation** : allocating memory at execution time

Dynamic Memory Allocation Functions

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
- There are 4 library functions under "stdlib.h" for dynamic memory allocation.

Cont..

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc( )

- The function *malloc* allocates a user-specified amount of memory and a pointer to the start of the allocated memory is returned.
- If there is insufficient memory to make the allocation, the returned value is NULL.

Where,

data_type *x;

x= (data_type *) malloc(size);

x is a pointer variable of data_type

size is the number of bytes

Ex: int *ptr;

ptr = (int *) malloc(100*sizeof(int));

calloc( )

- The function *calloc* allocates a user-specified amount of memory and initializes the allocated memory to **0** and a pointer to the start of the allocated memory is returned.
- If there is insufficient memory to make the allocation, the returned value is NULL.
- **Syntax:**

Where,

data_type *x;

x= (data_type *) calloc(n, size);

x is a pointer variable of type int

n is the number of block to be allocated

size is the number of bytes in each block

Example:

int *x

x= calloc (10, sizeof(int));

The above example is used to define a one-dimensional array of integers. The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

realloc( )

- Before using the realloc( ) function, the memory should have been allocated using malloc( ) or calloc( ) functions.
- The function relloc( ) resizes memory previously allocated by either ***malloc*** or ***calloc***, which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.
- If the existing allocated memory cannot be extended, the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.
- When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL

- **Syntax**:

```
data_type *x;
x= (data_type *) realloc(p, s );
```

free( )

- Dynamically allocated memory with either malloc( ) or calloc ( ) does not return on its own. The programmer must use free( ) explicitly to release space.

Syntax:

free(ptr);

- This statement cause the space in memory pointer by ptr to be deallocated

Representation of Array in a Memory

- In general, index function:

Loc (X[i]) = Loc(X[LB]) + w*(i-LB);

where w is length of memory location required.

For real number: 4 byte, integer: 2 byte and character: 1 byte.

➢ Example:

If LB = 5, Loc(X[LB]) = 1200, and w = 4, find Loc(X[8]) ?

Loc(X[8])= Loc(X[5]) + 4*(8 – 5)

= 12128

Representation of Linear Arrays in Memory

- A linear array is a list of a finite number *'n'* of homogeneous data element .The elements of the array are reference respectively by an index set consisting of n consecutive numbers and in successive memory locations.
- The number *n* of elements is called the length or size of the array. The length or the numbers of elements of the array can be obtained from the index set by the formula .

•When LB = 0 and LB = 1,

Where,

Length = UB – LB + 1 Length = UB

- UB is the largest index called the Upper Bound
- LB is the smallest index, called the Lower Bound

DYNAMICALLY ALLOCATED 1D ARRAYS

```
int i, n, *list;
        printf("Enter the number of numbers to generate:");
        scanf("%d", &n);
        if(n<1)
        {
        fprintf (stderr, "Improper value of n \n");
        exit(EXIT_FAILURE);
        }
```

MALLOC (list, n*sizeof(int));

The programs fail only when n<1 or insufficient memory to hold the list of numbers that are to be sorted

DYNAMICALLY ALLOCATED 2D ARRAYS

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4]=6;
int ** make2dArray(int rows, int cols)
{ /* create a two dimensional rows X cols array */ int **x,
        i;
        MALLOC(x, rows * sizeof (*x)); /*get memory for row pointers */ for (i=
        0;i<rows; i++)/* get memory for each row */
        MALLOC(x[i], cols * sizeof(**x)); return x;
}
```

Array Operations

- Traversing
- Inserting
- Deleting
- Searching
- Sorting

Traversing Algorithm

- Traversing operation means visit every element once.

e.g. to print, etc.

Algorithm 1: (Traversing a Linear Array)

1. Hear LA is a linear array with the lower bound LB and upper bound UB.
2. This algorithm traverses LA applying an operation PROCESS to each element of LA using while loop.
3. [Initialize Counter] set K:= LB
4. Repeat step 3 and 4 while K ≤ UB
5. [Visit element] Apply PROCESS to LA[K]
6. [Increase counter] Set K:= K + 1 [End of step 2 loop]
7. Exit

Inserting

- Inserting an element at the “end” of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.

Algorithm:

1. INSERT (LA, N, K, ITEM)
2. Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. This algorithm inserts an element ITEM into the Kth position in LA.
3. [Initialize counter] set J:= N
4. Repeat step 3 and 4 while J ≥ K
5. [Move J^{th} element downward] set LA[J+1] := LA[J]
6. [Decrease counter] set J:= J – 1 [End of step 2 loop]
7. [Insert element] set LA[K]:= ITEM
8. [Reset N] set N:= N+1
9. Exit

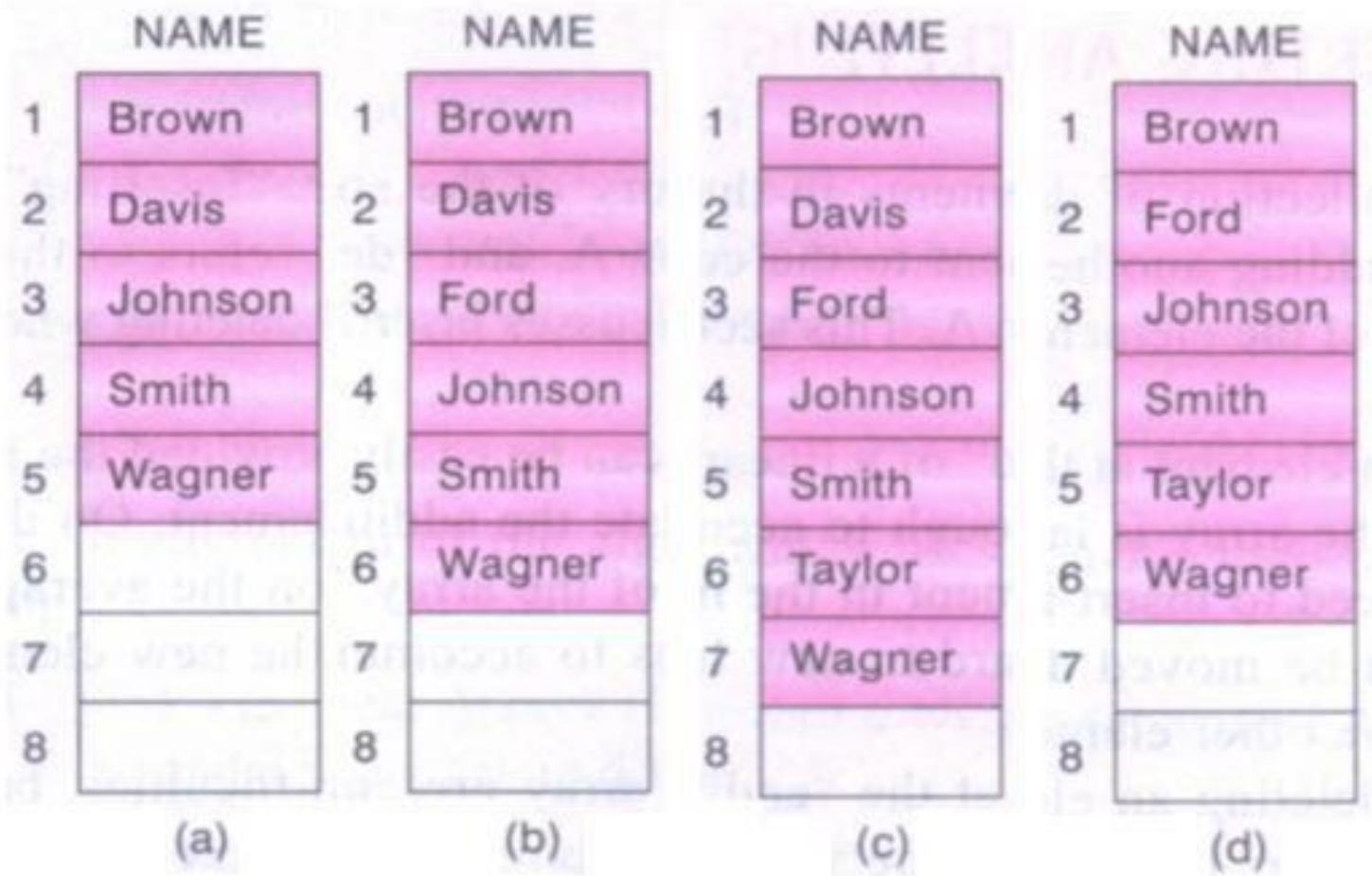
Deleting

Deleting refers to the operation of removing one element to the collection A

Algorithm

1. DELETE (LA, N, K, ITEM)
2. Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. this algorithm deletes the Kth element from LA
3. Set ITEM:= LA[K]
4. Repeat for J = K to N – 1
5. [Move J + 1 element upward] set LA[J]:= LA[J+1] [End of loop]
6. [Reset the number N of elements in LA] set N:= N – 1
7. Exit

Example for Insertion and Deletion



Sorting

Sorting refers to the operation of rearranging the elements of a list.

- Ex: suppose A is the list of n numbers. Sorting A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that,

A[I] <A[2] <A[3] < ... <A[N]

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

Bubble Sort

Algorithm: BUBBLE (DATA, N)

Here DATAis an array with N elements. This algorithm sorts the elements in DATA.

- Repeat Steps 2 and 3 for K = 1 to N - 1.
- Set PTR: = 1.[Initializes pass pointer PTR.]
- Repeat while PTR ≤ N - K: [Executes pass.]
  - If DATA[PTR] > DATA[PTR + 1], then:

Interchange DATA[PTR] and DATA[PTR + 1]. [End of If structure.]

- Set PTR: = PTR + 1. [End of inner loop.]

[End of Step 1 outer loop.]

Exit

Searching

Searching refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there.

Algorithm: (Linear Search) LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC: = 0 if the search is unsuccessful.

[Insert ITEM at the end of DATA.] Set DATA[N + 1]: = ITEM.

[Initialize counter.] Set LOC: = 1.

[Search for ITEM.]

Repeat while DATA[LOC] ≠ ITEM: Set LOC:= LOC + 1.

[End of loop.]

[Successful?] If LOC = N + 1, then: Set LOC:= 0

Exit.

Binary Search

- Algorithm: (Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)
- Here DATAis a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, the beginning, end and middle locations of a segment of elements of DATA.

Cont..

- This algorithm finds the location LOC of ITEM in DATAor sets LOC = NULL.
- [Initialize segment variables.]
- Set BEG: = LB, END := UB and MID = INT((BEG + END)/2).
- Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
- If ITEM < DATA[MID], then: Set END := MID - 1.
- Else:

Cont..

- Set BEG := MID + 1.
- [End of If structure.]
- Set MID := INT((BEG + END)/2). [End of Step 2 loop.]
- If DATA[MID] = ITEM, then:
- Set LOC := MID.
- Else:
- Set LOC := NULL.
- [End of If structure.]
- Exit.

Polynomials

A polynomial is a sum of terms, where each term has a form **axe** , where **x** is the variable, **a** is the coefficient and **e** is the exponent.

Two example polynomials are:

$A(x) = 3x20 + 2x5 + 4$

$B(x) = x4 + 10x3 + 3x2 + 1$

Polynomial Representation

```
#define MAX-DEGREE   101     /*Max degree of polynomial+1*/
typedef struct{
int degree;
float coef[MAX-DEGREE];
} polynomial;
```

Sparse Matrices

	col0	col1	col2
row 0	-27	3	4
row 1	6	82	-2
row 2	109	-64	11
row 3	12	8	9
row 4	48	27	47

Figure A

	col0	col1	col2	col3	col4	col 5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Figure B

Implementation of Sparse Matrix using arrays

```
typedef struct
{
    int col;
    int row;
    int value;
}terms;
terms a[MAX_TERMS];
int avail;
void InsertElement(int r,int c,int val)
{
    a[avail].col=c;
    a[avail].row=r;
    a[avail].value=val;
    a[0].value++;
    avail++;
}
```

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Implementation of Sparse Matrix using arrays

```
    printf("\n The elements of matrix\n");
    for(i=0;i<avail;i++)
    {
        printf("\na[%d].row=%d\ta[%d].col=%d\ta[%d].val=%d",i,a[i].row,i,a[i].col,i,a[i].value);
    }
    k=1;
    printf("\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            if(a[k].row==i && a[k].col==j)
            {
                printf("\t%d",a[k].value);
                k++;
            }
            else
                printf("\t0");
        }
        printf("\n");
    }
}
```

Cont..

```
Enter the order of the Matrix
6 6

 Enter the elements of matrix
15 0 0 22 0 -15
0 11 3 0 0 0
0 0 0 -6 0 0
0 0 0 0 0 0
91 0 0 0 0 0
0 0 28 0 0 0

 The elements of matrix

a[0].row=6      a[0].col=6      a[0].val=8
a[1].row=0      a[1].col=0      a[1].val=15
a[2].row=0      a[2].col=3      a[2].val=22
a[3].row=0      a[3].col=5      a[3].val=-15
a[4].row=1      a[4].col=1      a[4].val=11
a[5].row=1      a[5].col=2      a[5].val=3
a[6].row=2      a[6].col=3      a[6].val=-6
a[7].row=4      a[7].col=0      a[7].val=91
a[8].row=5      a[8].col=2      a[8].val=28
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Cont..

```
 The elements of matrix

a[0].row=5      a[0].col=6      a[0].val=6
a[1].row=0      a[1].col=4      a[1].val=9
a[2].row=1      a[2].col=1      a[2].val=8
a[3].row=2      a[3].col=0      a[3].val=4
a[4].row=2      a[4].col=3      a[4].val=2
a[5].row=3      a[5].col=5      a[5].val=5
a[6].row=4      a[6].col=2      a[6].val=2
 The elements of Transpose matrix

b[0].row=6      b[0].col=5      b[0].val=6
b[1].row=0      b[1].col=2      b[1].val=4
b[2].row=1      b[2].col=1      b[2].val=8
b[3].row=2      b[3].col=4      b[3].val=2
b[4].row=3      b[4].col=2      b[4].val=2
b[5].row=4      b[5].col=0      b[5].val=9
b[6].row=5      b[6].col=3      b[6].val=5
```

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

0	0	4	0	0
0	8	0	0	0
0	0	0	0	2
0	0	2	0	0
9	0	0	0	0
0	0	0	5	0

Row	Col	Value
6	5	6
0	2	4
1	1	8
2	4	2
3	2	2
4	0	9
5	3	5

- A matrix which contains many zero entries or very few non-zero entries is called as Sparse matrix.
- In the **figure B** contains only 8 of 36 elements are nonzero and that is sparse.

SparseMatrix Create(maxRow, maxCol) ::=

```
#define MAX_TERMS   101      /* maximum number of terms +1*/
typedef struct {
int col; int row; int value;
} term; term a[MAX_TERMS];
```

Basic Terminology

- **String:** A finite sequence S of zero or more Characters is called string.
  Length: The number of characters in a string is called length of string.
  Empty or Null String: The string with zero characters.
- **Concatenation:** Let S1 and S2 be the strings. The string consisting of the characters of S1 followed by the character S2 is called Concatenation of S1 and S2.
- Ex: ‘THE’ // ‘END’ = ‘THEEND’ ‘THE’ // ‘ ’ // ‘END’ = ‘THE END’
- **Substring:** A string Y is called substring of a string S if there exist string X and Z such that S = X // Y // Z
- If X is an empty string, then Y is called an Initial substring of S, and Z is an empty string then Y is called a terminal substring of S.
- Ex: ‘BE OR NOT’ is a substring of ‘TO BE OR NOT TO BE’ ‘THE’ is an initial substring of ‘THE END’

Strings

STRINGS IN C

- In C, the strings are represented as character arrays terminated with the null character \0.

Declaration 1:

```
#define MAX_SIZE  100/* maximum size of string */
char s[MAX_SIZE] = {“dog”};
char t[MAX_SIZE] = {“house”};
```

s[0]	s[1]	s[2]	s[3]		t[0]	t[1]	t[2]	t[3]	t[4]	t[4]
d	o	g	\0		h	o	u	s	e	\0

Storing Strings

Strings are stored in three types of structures

- Fixed length structures
- Variable length structures with fixed maximum
- Linked structures

Character DATA type

Constants

- Many programming languages denotes string constants by placing the string in either single or double quotation marks.

Ex: 'THE END'

"THE BEGINNING"

The string constants of length 7 and 13 characters respectively.

Variables

- Each programming languages has its own rules for forming character variables. These variables fall into one of three categories
- **Static:** In static character variable, whose length is defined before the program is executed and cannot change throughout the program
- **Semi-static:** The length of the variable may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed.
- **Dynamic:** The length of the variable can change during the execution of the program

String Operation

Substring

- Accessing a substring from a given string requires three pieces of information:
- The name of the string or the string itself
- The position of the first character of the substring in the given string
- The length of the substring or the position of the last character of the substring.

Syntax:SUBSTRING (string, initial, length)

The syntax denote the substring of a string S beginning in a position K and having a length L.

Ex: SUBSTRING ('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'

SUBSTRING ('THE END', 4, 4) = ' END'

Indexing

- Indexing also called pattern matching, refers to finding the position where a string pattern P first appears in a given string text T. This operation is called INDEX

Syntax: INDEX (text, pattern)

- If the pattern P does not appears in the text T, then INDEX is assigned the value 0. The arguments "text" and "pattern" can be either string constant or string variable.

Concatenation

- Let S1 and S2 be string. The concatenation of S1 and S2 which is denoted by S1 // S2, is the string consisting of the characters of S1 followed by the character of S2.

Ex:

(a) Suppose S1 = 'MARK' and S2= 'TWAIN' then S1 // S2 = 'MARKTWAIN'

Concatenation is performed in C language using ***strcat*** function as shown below

strcat (S1, S2);

Concatenates string S1 and S2 and stores the result in S1

strcat ( ) function is part of the ***string.h*** header file; hence it must be included at the time of pre- processing

Length

- The number of characters in a string is called its length.

Syntax:

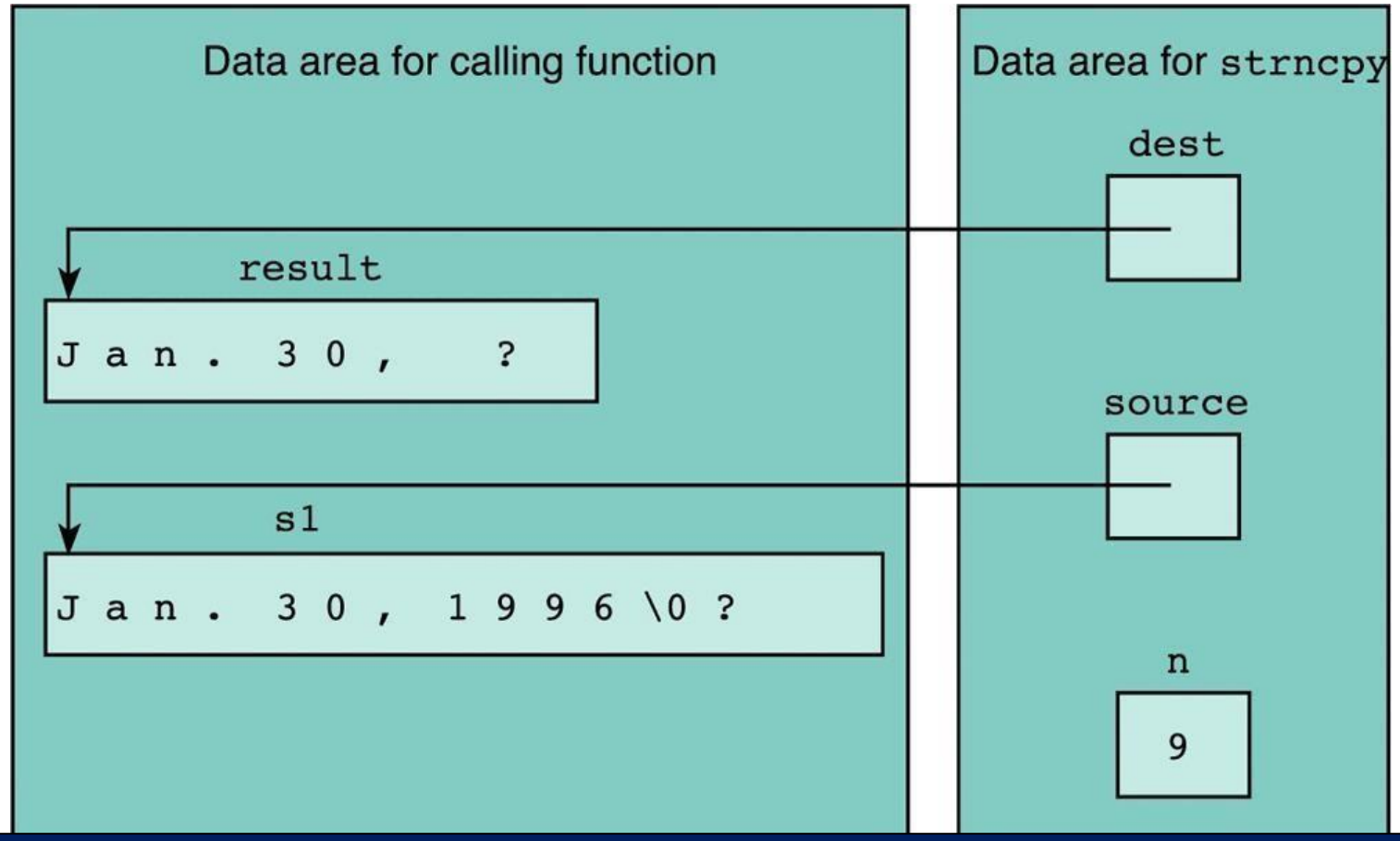
LENGTH (string)

Ex: LENGTH (‘computer’) = 8

- String length is determined in C language using the ***strlen( )*** function, as shown below: X = strlen ("sunrise");
- strlen function returns an integer value 7 and assigns it to the variable X
- Similar to ***strcat, strlen*** is also a part of string.h, hence the header file must be included at the time of pre-processing.

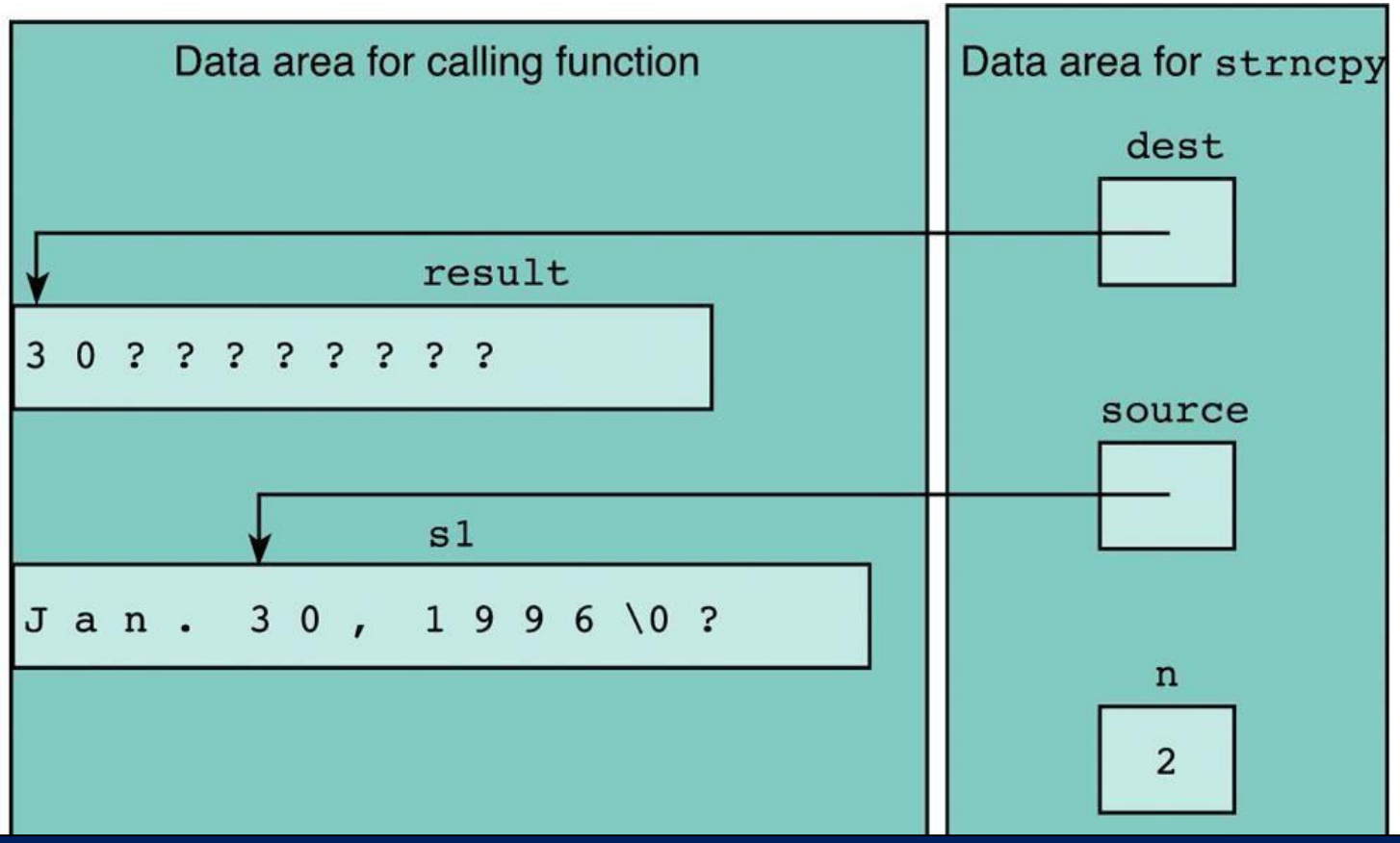
Extracting Substring of a String (1/2)

- We can use strncpy to extract substring of one string.
  - e.g., strncpy(result, s1, 9);



Extracting Substring of a String (2/2)

- e.g., strncpy(result, **&s1[5]**, 2);



Functions strcpy and strncpy

- Function strcpy copies the string in the second argument into the first argument.
  - e.g., strcpy(dest, “test string”);
  - The **null character** is appended at the end automatically.
  - If source string is longer than the destination string, the overflow characters may occupy the memory space used by other variables.
- Function strncpy copies the string by specifying the number of characters to copy.
  - You have to place the null character manually.
  - e.g., strncpy(dest, “test string”, 6); **dest[6] = ‘\0’;**
  - If source string is longer than the destination string, the overflow characters are discarded automatically.

Pattern Matching Algorithms

- Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. The length of P does not exceed the length of T.

Algorithm: (Pattern Matching)

P and T are strings with lengths R and S, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

[Initialize.] Set K: = 1 and MAX: = S - R + 1

Repeat Steps 3 to 5 while K ≤ MAX

Repeat for L = 1 to R: [Tests each character of P]

If P[L] ≠ T[K + L – l], then: Go to Step 5 [End of inner loop.]

[Success.] Set INDEX = K, and Exit

Set K := K + 1

[End of Step 2 outer loop]

[Failure.] Set INDEX = O

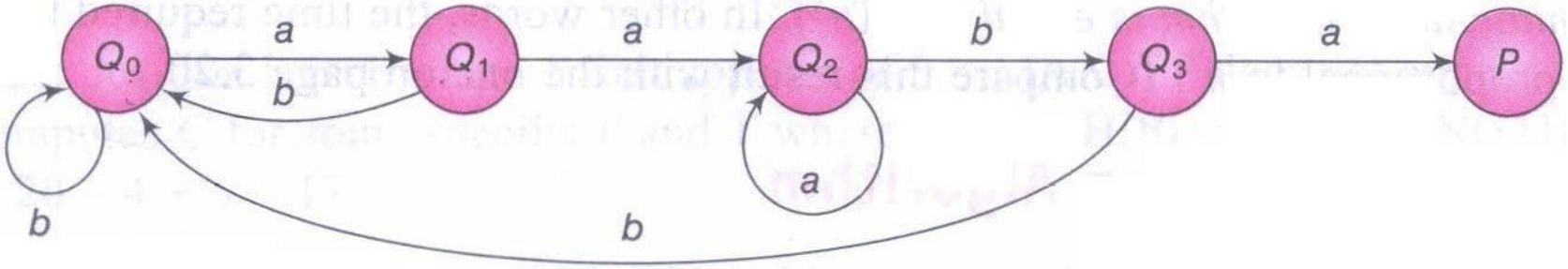
Exit

Second Pattern Matching Algorithm

- The second pattern matching algorithm uses a table which is derived from a particular pattern P but is independent of the text T.
- For definiteness, suppose
- P = aaba
- This algorithm contains the table that is used for the pattern P = *aaba*.

	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

(a) Pattern matching table



b Pattern matching graph

Algorithm : (PATTERN MATCHING)

- The pattern matching table F(Q1, T) of a pattern P is in memory, and the input is an N-character string T = T1 T2 T3 …… TN.
- The algorithm finds the INDEX of P in T.

- [Initialize] set K: =1 ans S1 = Q0
- Repeat steps 3 to 5 while SK ≠ P and K ≤ N
- Read TK
- Set SK+1 : = F(SK, TK) [finds next state]
- Set K: = K + 1 [Updates counter] [End of step 2 loop]
- [Successful ?] If SK = P, then
- INDEX = K – LENGTH (P)
- Else
- INDEX = 0
- [End of IF structure]
- Exit.

Brute-Force String Matching

- ***Pattern***: a string of *m* characters to search for
- ***Text***: a (longer) string of *n* characters to search in
- **Problem**: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until
- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

STACKS

INTRODUCTION

- The linear lists and linear arrays allows one to insert and delete elements at any place in the list- at the beginning, at the end or in the middle
- There are certain frequent situations when one wants to restricts insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle
- They are STACKS and QUEUES

STACKS

- Stack is a linear list in which addition and deletion of elements are restricted to one end called the **top.**
- A linear list is a list in which each element has unique successor.
- A linear list may be a restricted list or a general list.
- Stack is also called as
  - LIFO Data structure
  - Push down list

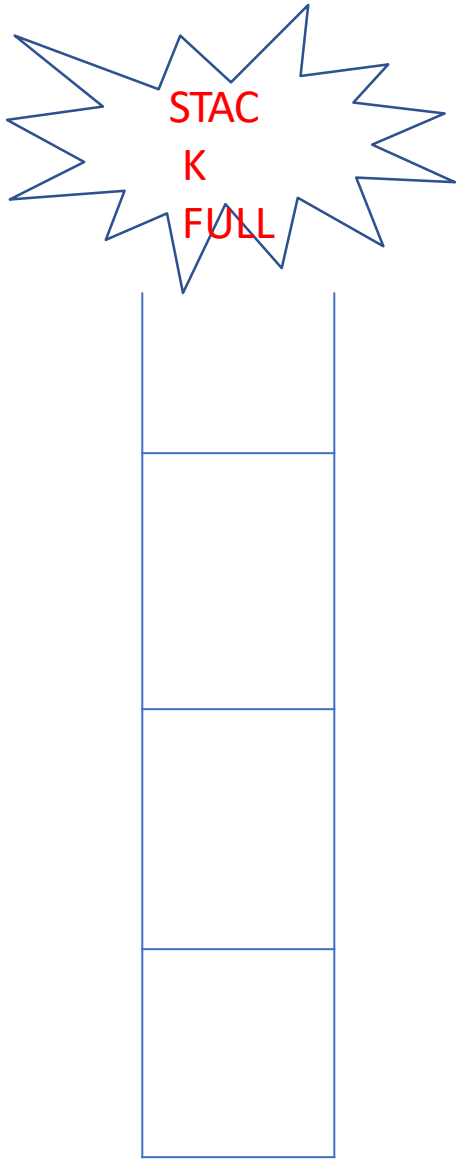
Stack Operations

- PUSH – Used to insert an element into Stack.
- POP – Used to delete an element from Stack.
- Stack top – used to copy the item at the top of Stack but does not delete it.

Underflow and Overflow

- Underflow is a condition occurs if we try to POP item from the empty STACK
- Overflow is a condition occurs if we try to PUSH an ITEM to the STACK which is already full

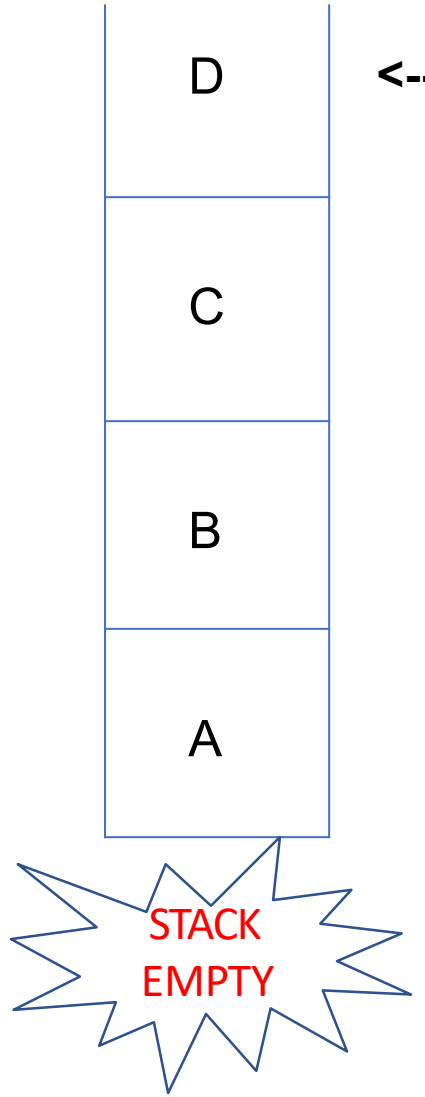
E D C B A



STACK OVERFLOW

<--TOP

POP

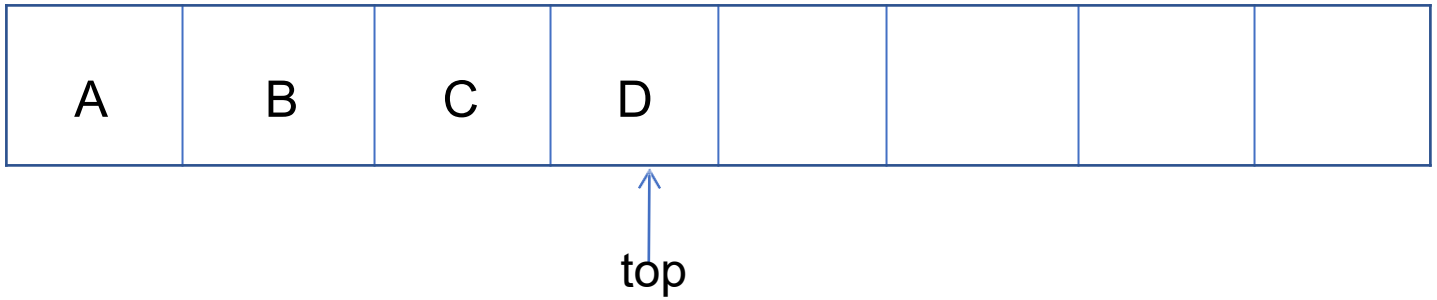


STACK UNDERFLOW

Array representation of Stack

- **Array can be used as home to the stack**
- **Data structure**

```
#define MAX STK 25
typedef struct {
        int top;
        char item[MAXSTK];
        }STACK;
STACK s;
s.top= -1;
```



Stack after – PUSH(‘A’), PUSH(‘B’), PUSH(‘C’), PUSH(‘D’)

PUSH(STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack

1.[Stack already filled?]

If TOP = MAXSTK, then: print: OVERFLOW, and return

2. Set TOP := TOP + 1 [Increases TOP BBY 1]

3.Set STACK[TOP] := ITEM [Inserts ITEM in new TOP position]

4. Return

POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM

1. [Stack has an item to be removed?]

   If TOP = 0, then: Print : UNDERFLOW, and Return

2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM]

3. Set TOP := TOP – 1. [Decreases TOP by 1]

4. Return

Stack representation using arrays in C

```
#define MAXSTK 10
typedef struct {
        int item[MAXSTK];
        int top;
        }STACK;

  STACK s;
  s.top = -1;
```

Stacks using dynamic arrays

```
#define MAX 25
typedef struct {
        int  key;
        /*other fields*/
        }Element;
Element * stack;
int top= -1;
capacity = 1;
stack = (Element *) malloc (sizeof (*stack));

int IsFull( ) {
if ( top >= capacity – 1)
return 1;
return 0;
}
```

```
Void Push(Element item)
{
If(IsFull( ))
Stackfull( );
Top++;
* stack=item;
return;
}
Stackfull( )
{
Stack=realloc(2*capacity*size of (*stack));
Capacity*=2;
}
```

```
    int Isempty( ){
    If(top <0)
                return 1;
                return 0;
                                }
Element Pop( )
                {
 Element x;
    If(!Is empty(s)
    {
X=*stack;
Stack--;
    return x;
    }
                }
```

Array doubling

```
void stackFull()
{
  realloc(s , 2*capacity*sizeof(*s));
  capacity * = 2;
}
```

Stack Applications

- Postponing data usage
- Reversing Data
- Parsing Data
- Backtracking Steps.

Postponing data usage

- A stack can be used in applications that requires that the use of data be postponed for a while.
- Examples:

1. Infix to postfix transformation
2. Evaluation of postfix expression

Evaluation of expression

- Three levels of precedence for the 5 binary operations
- Highest : Exponentiation ( ) or ($) or (^) or (**)
- Next Highest : Multiplication (*) and Division(/)
- Lowest: Addition (+) and Subtraction (-)

- 2 3 + 5 * 2 2 - 12 / 6
- To evaluate expression is traversed three times, each time corresponding to the level of precedence of the operations
- After first traversal expression reduces to 8 + 5 * 4 - 12 / 6
- After Second traversal expression reduces to 8 + 20 – 2
- After Third traversal expression reduces to 26

Arithmetic Expression Representation

- INFIX
- PREFIX – POLISH
- POSTFIX – REVERSE POLISH or SUFFIX

INFIX	POLISH	REVERSE POLISH
A+B*C	+A*BC	ABC*+
(A+B)*C	*+ABC	AB+C*

- Note
- The order of operators and operands in an INFIX determine the order in which operations to be performed
- In POLISH and REVERSE POLISH notation the order in which the operations to be preformed is completely determined by the position of the operators and the operands in the expression
- No parenthesis required to determine the order
- Computer evaluates arithmetic expressions in INFIX notation in two steps
- 1) Convert INFIX expression to POSTIX
- 2) Evaluate the POSTFIX expression

Converting infix to postfix and Prefix - Examples

INFIX

1. A+B*C : A+[BC*] ABC*+ is POSTFIX

   A+[*BC] +A*BC is PREFIX

2 (A+B)*C : [AB+]*C AB+C* is POSTFIX

   [+AB]*C *+ABC is PREFIX

3 A$B*C-D+E/F/G+H: [AB$]*C-D+E/F/G+H

(Converting to postfix) [AB$C*]-D+[EF/G/]+H

AB$C*D-EF/G/+H+ is POSTFIX

(Converting to prefix) [$AB]*C-D+E/F/G+H

[*$ABC]-D+[//EFG]+H

++-*$**ABCD//EFGH is** PREFIX

Converting infix to postfix and Prefix - INFIX to postfix conversion

Given Infix Expression: ((A+B)*C-(D-E))$(F+G)

([AB+]*C-[DE-])$[FG+]

([AB+C*]-[DE-])$[FG+]

[AB+C*DE--]$[FG+]

AB+C*DE--FG+$ is Post expression

INFIX to PREFIX Conversion

Given Infix Expression: ((A+B)*C-(D-E))$(F+G)

([+AB]*C-[-DE])$[+FG]

([*+ABC]-[DE-])$[+FG]

[-*+ABC-DE]$[+FG]

$-*+ABC-DE+FG is Prefix expression

Evaluation of postfix expression

To evaluate postfix expression, create an empty stack

- Scan from left to right to get a token
- If token is operand place on the stack
- If token is an operator remove two operands from the stack
- Perform the operation on that operands corresponding to the operator
- Result is placed back on the stack
- Continue the procedure until we reach end of the expression
- Remove the answer from the stack.

Algorithm to evaluate POSTFIX expression

- EVALUATE (P)

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation

1. Append a right parenthesis ")" at the end of P.[this acts as a sentinel]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered
3. If an operand is encountered, place it on to STACK
4. If an operator is encountered, then,
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element
   (b) Evaluate B operator A
   (c) Place the result of (b) back on stack

   [End of If structure]

   [End of Step 2 loop]

5. Set VALUE equal to the top element on STACK
6. Exit

Evaluating Given Postfix Expression :

Example: 6 2/ 3– 42*+

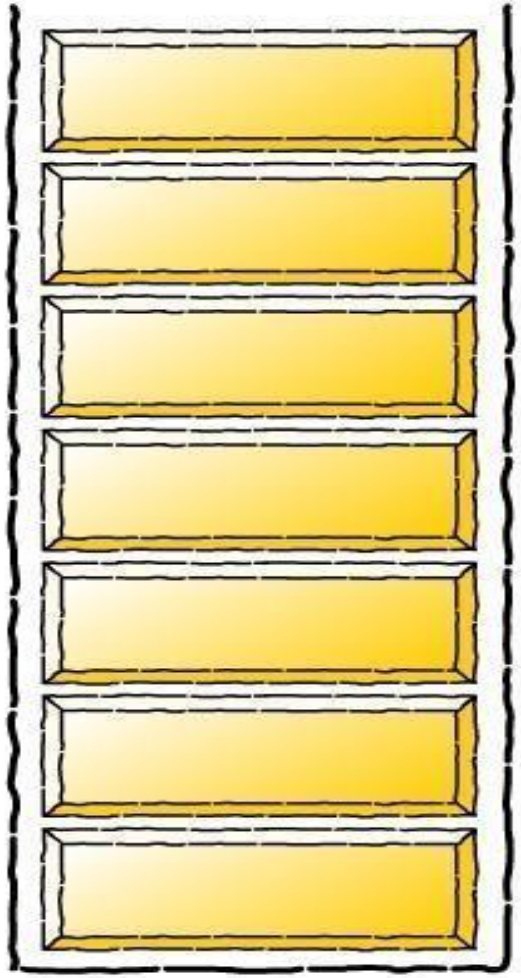
Token	Stack [0] [1] [2]	top
Initially		-1
6	6	0
2	6 2	1
/	3	0
3	3 3	1
-	0	0
4	0 4	1
2	0 4 2	2
*	0 8	1
+	8	0
eos		-1 **Answer=**

Postfix Evaluation

Postfix expression

6 2 / 3 – 4 2 * +

stack



Postfix Expression

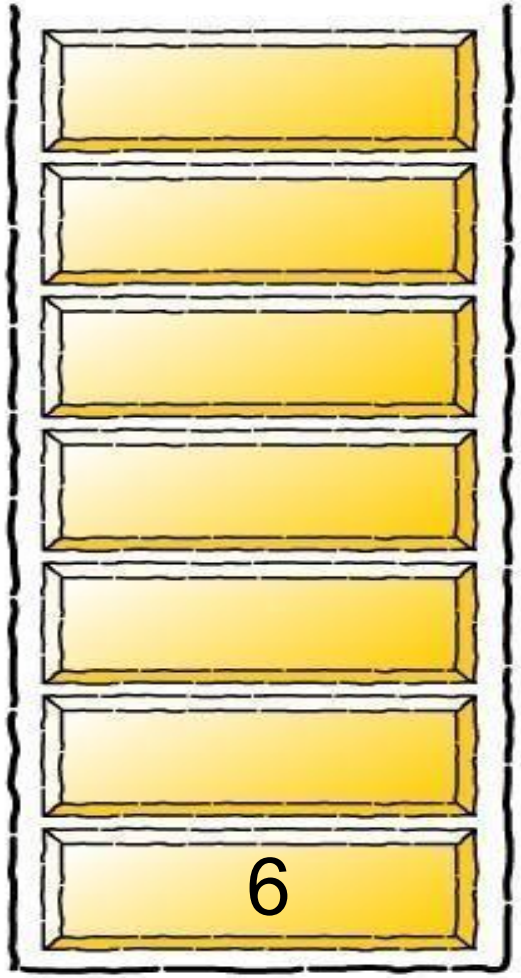
2 / 3 – 4 2 * +

Token

6

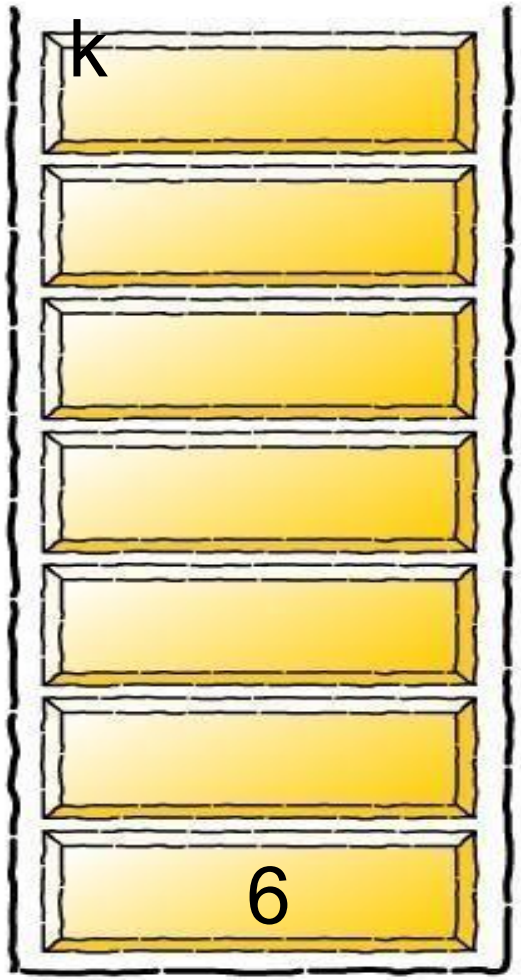
Operand, PUSH onto stack

stack



Postfix Expression

2 / 3 – 4 2 * +



Postfix Expression

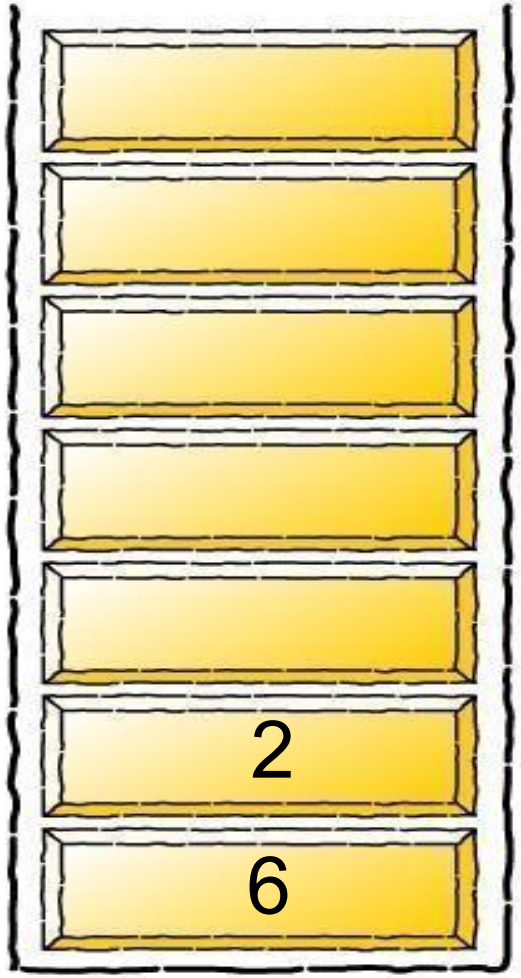
/ 3 – 4 2 * +

Token

2

Operand, PUSH onto stack

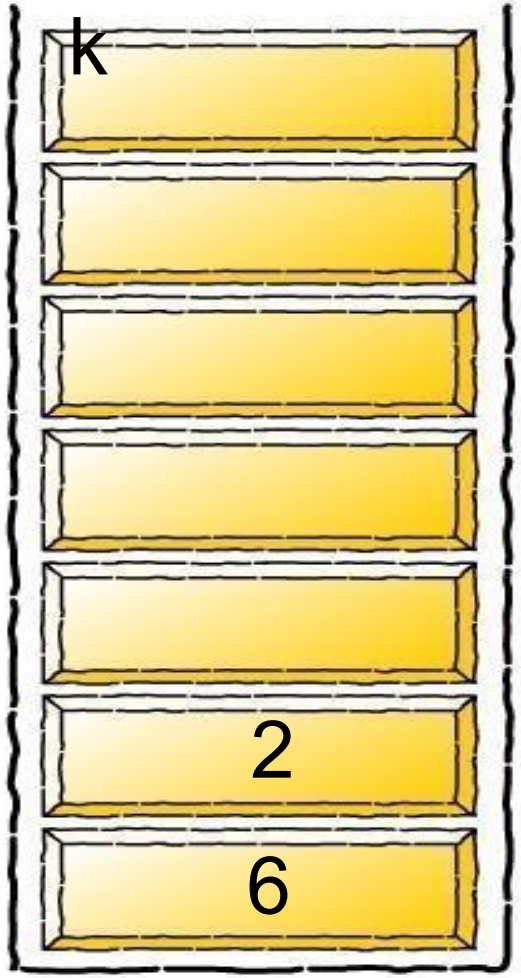
stack



Postfix Expression

/ 3 – 4 2 * +

stac
k



Postfix Expression

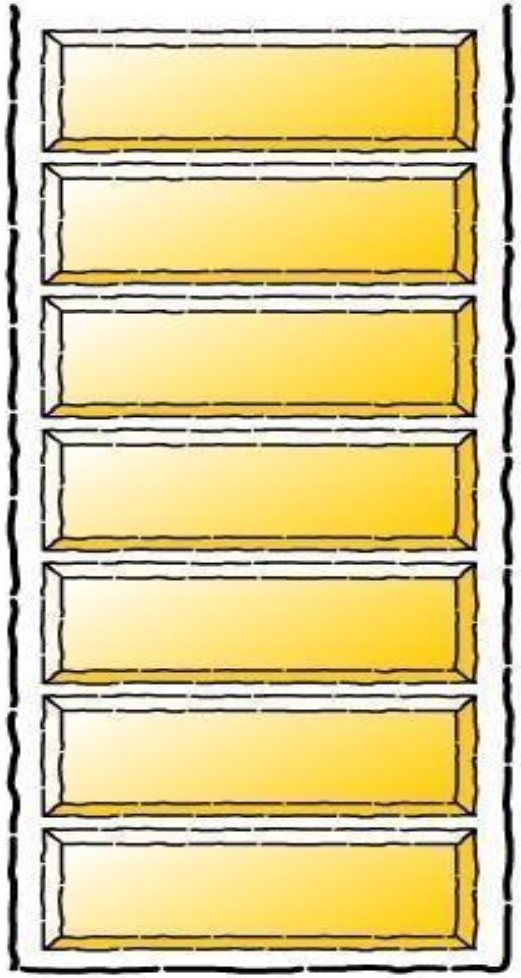
3 – 4 2 * +

Token

/

Operator, POP 2 top elements from the stack and store it in opnd2 and opnd1

stack



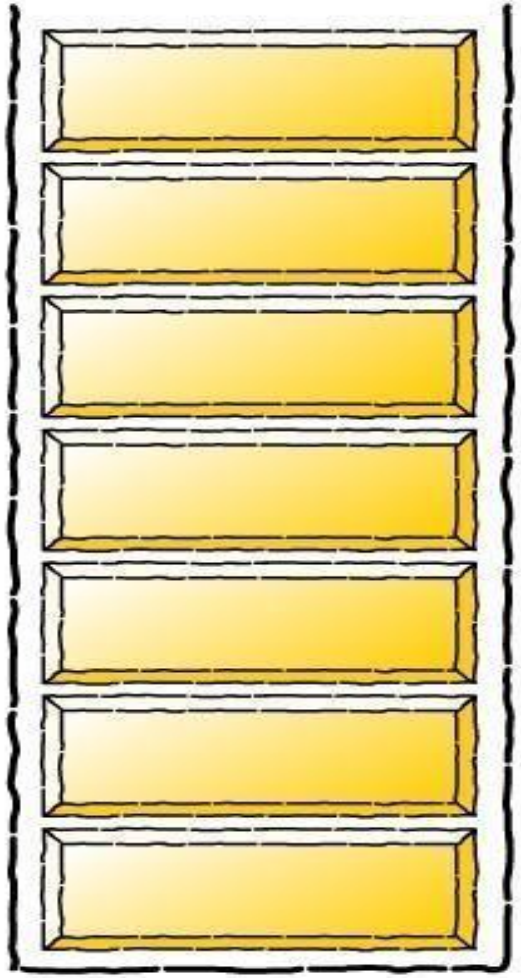
Postfix Expression

3 – 4 2 * +

opnd1 opnd2

6 2

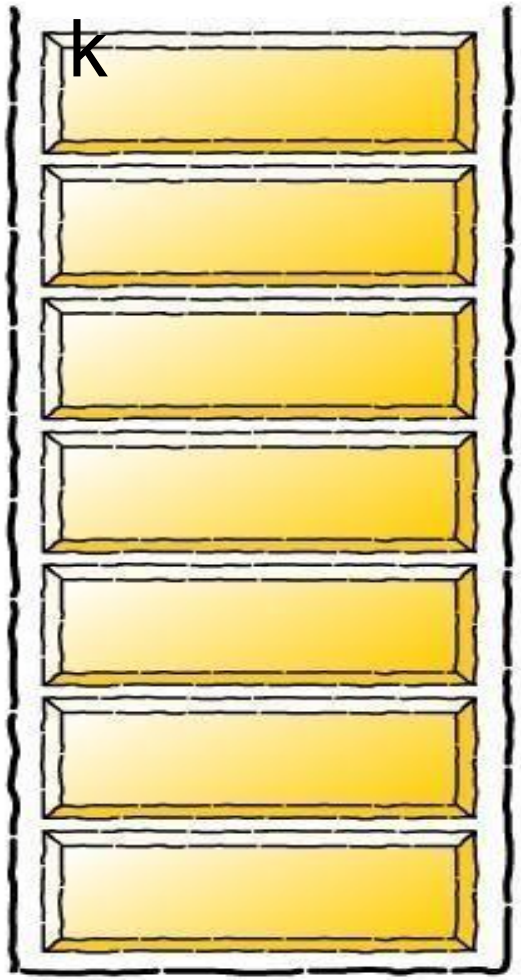
stack



Postfix Expression

3 – 4 2 * +

opnd1	opnd2	Evaluate opnd1 operator opnd2
6	2	



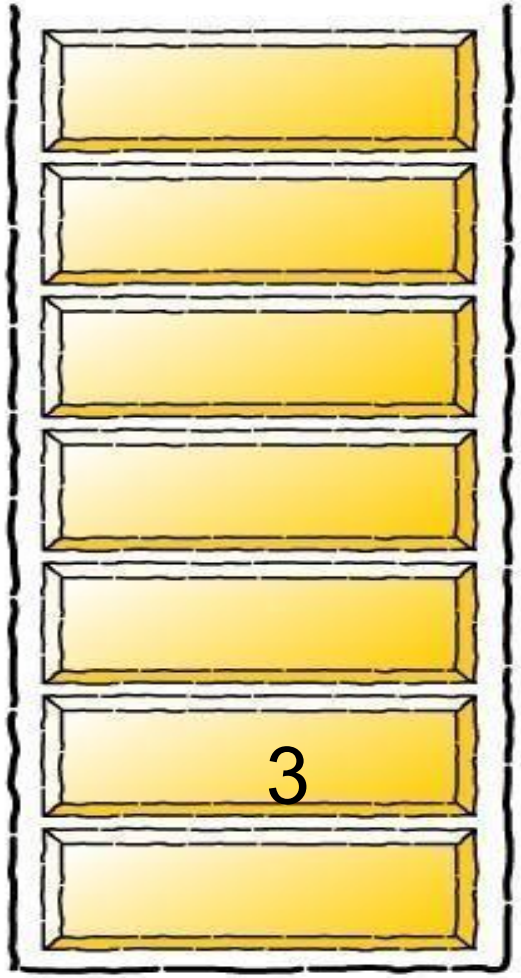
Postfix Expression

3 – 4 2 * +

opnd1		opnd2	Res
6	/	2	3

PUSH Res onto stack

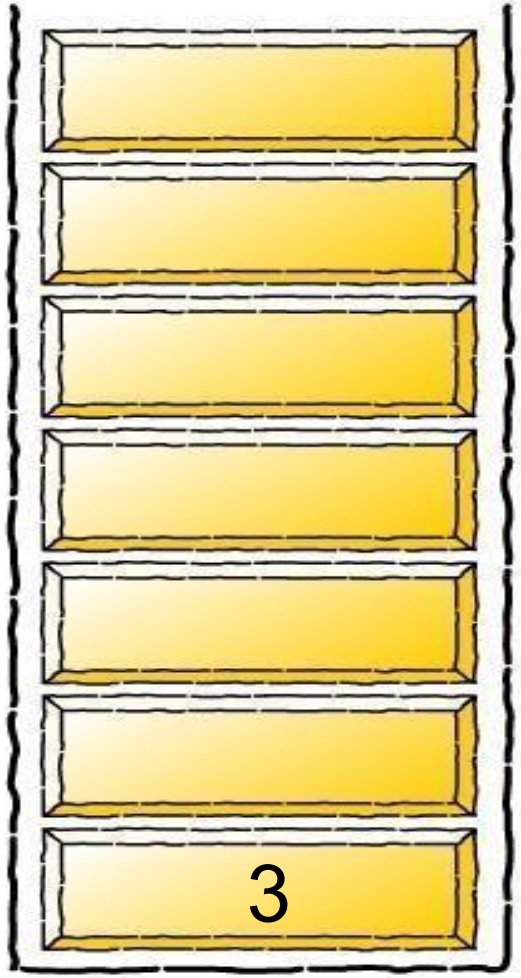
stack



Postfix Expression

3 – 4 2 * +

stack



Postfix Expression

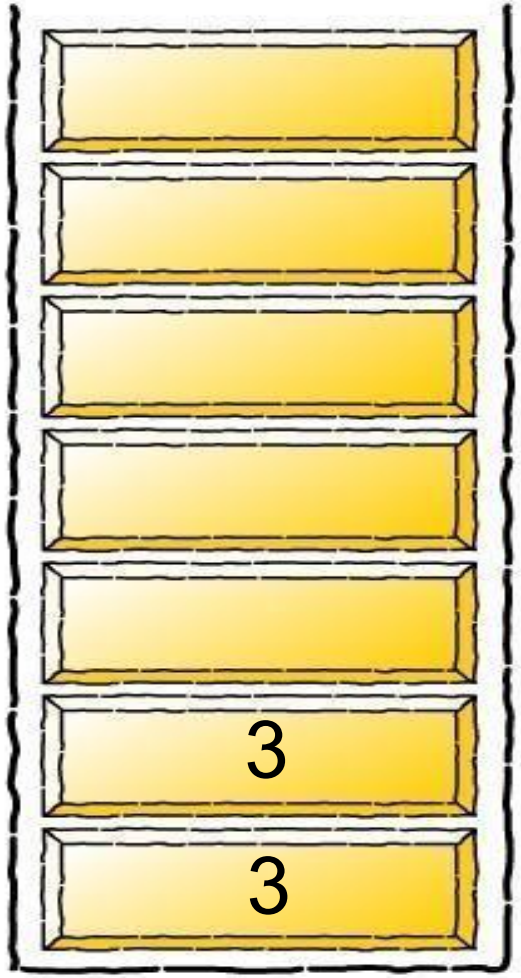
– 4 2 * +

Token

3

Operand, PUSH onto stack

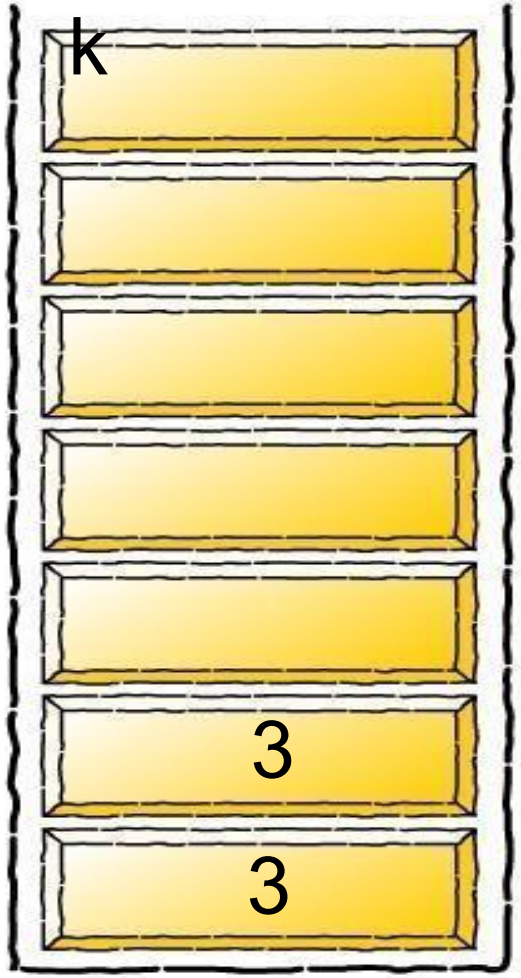
stack



Postfix Expression

– 4 2 * +

stac
k



Postfix Expression

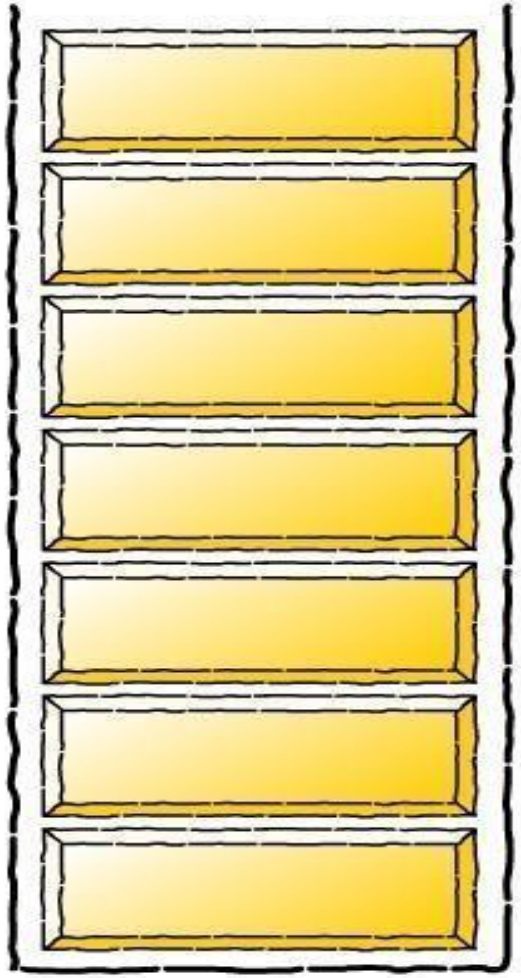
4 2 * +

Token

-

Operator, POP 2 top elements from the stack and store it in opnd2 and opnd1

stack

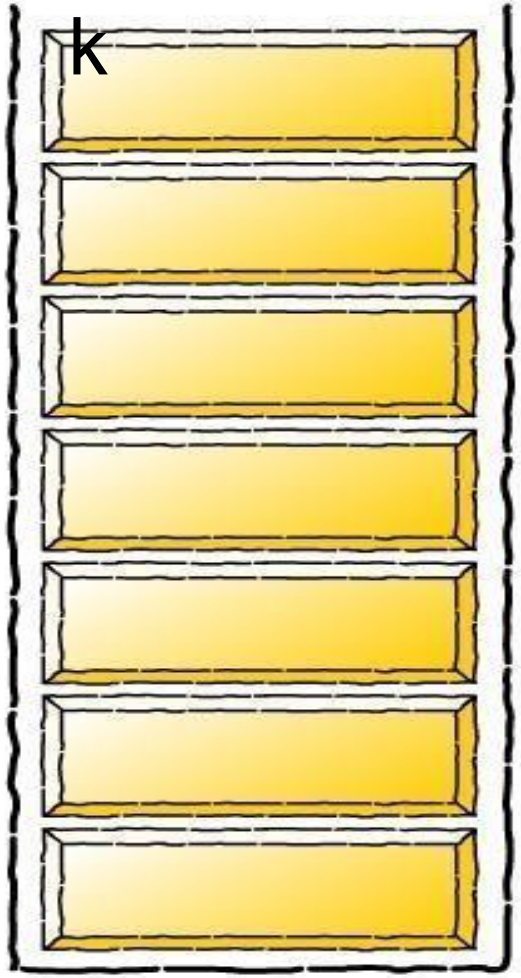


Postfix Expression

4 2 * +

opnd1	opnd2	Evaluate opnd1 operator opnd2
3	3	

stac
k



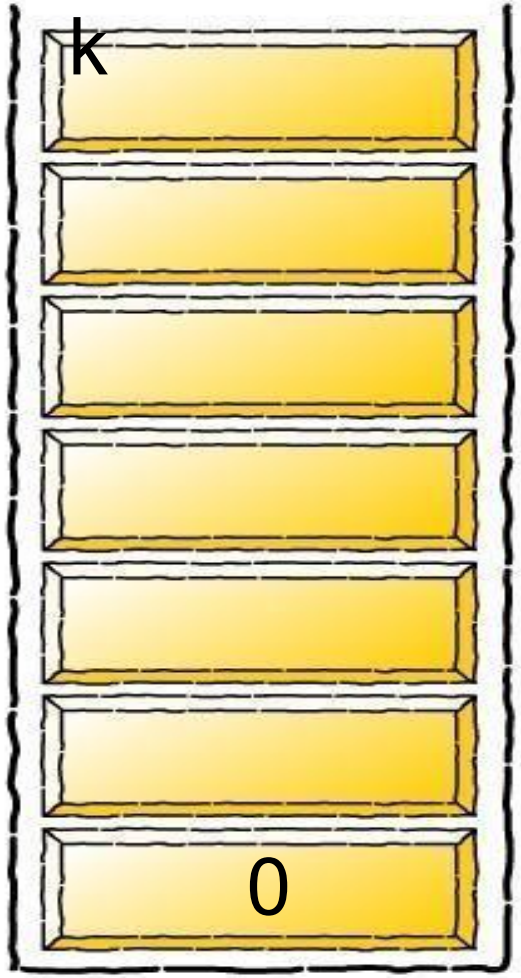
Postfix Expression

4 2 * +

opnd1		opnd2	Res
3	-	3	0

PUSH Res onto stack

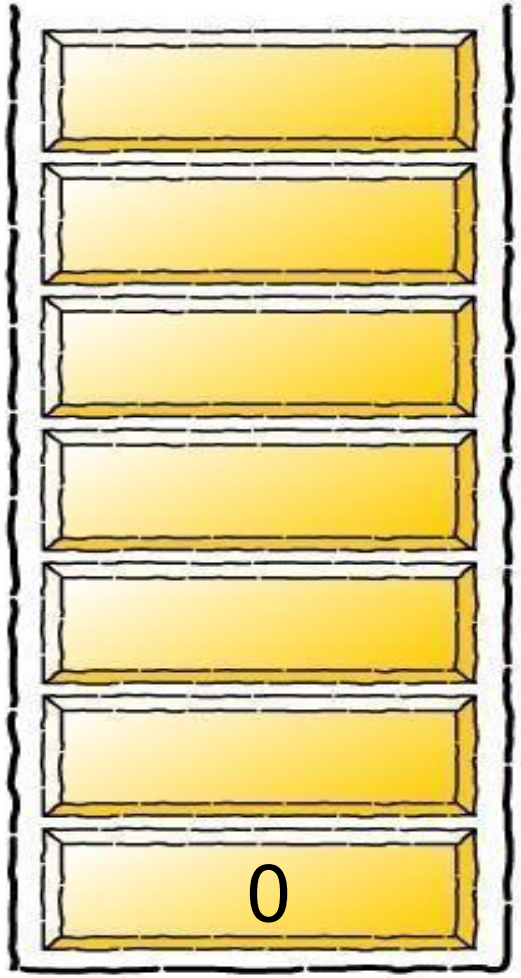
stac
k



Postfix Expression

4 2 * +

stack



Postfix Expression

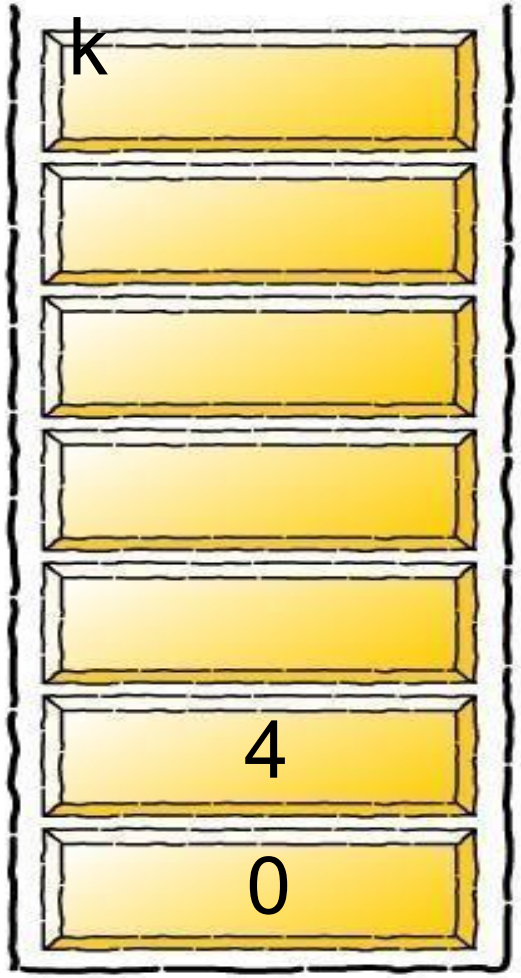
2 * +

Token

4

Operand, PUSH onto stack

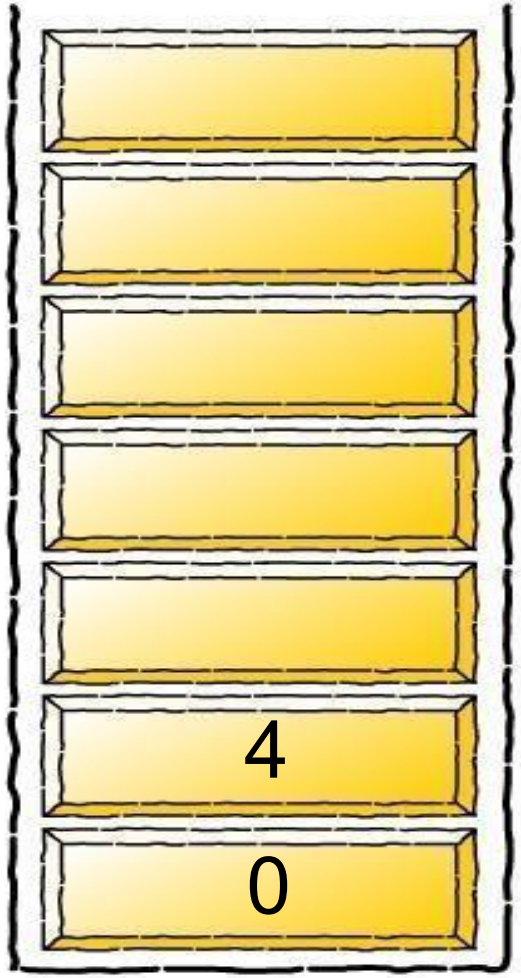
stac
k



Postfix Expression

2 * +

stack



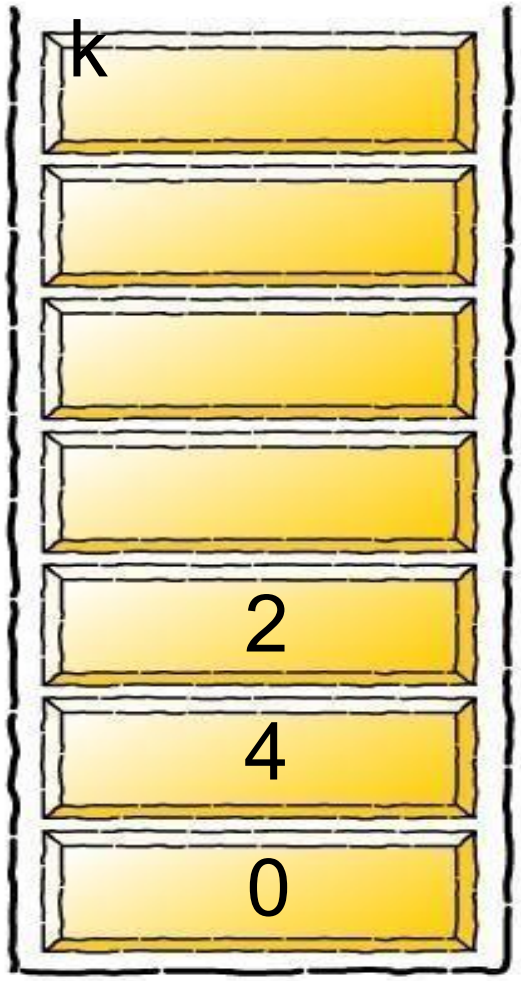
Postfix Expression

* +

Token

2

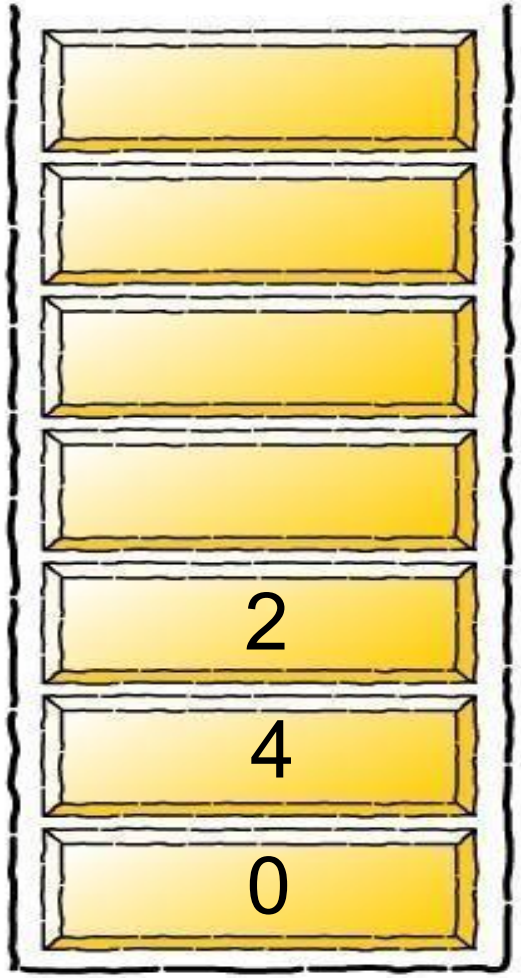
Operand, PUSH onto stack



Postfix Expression

* +

stack



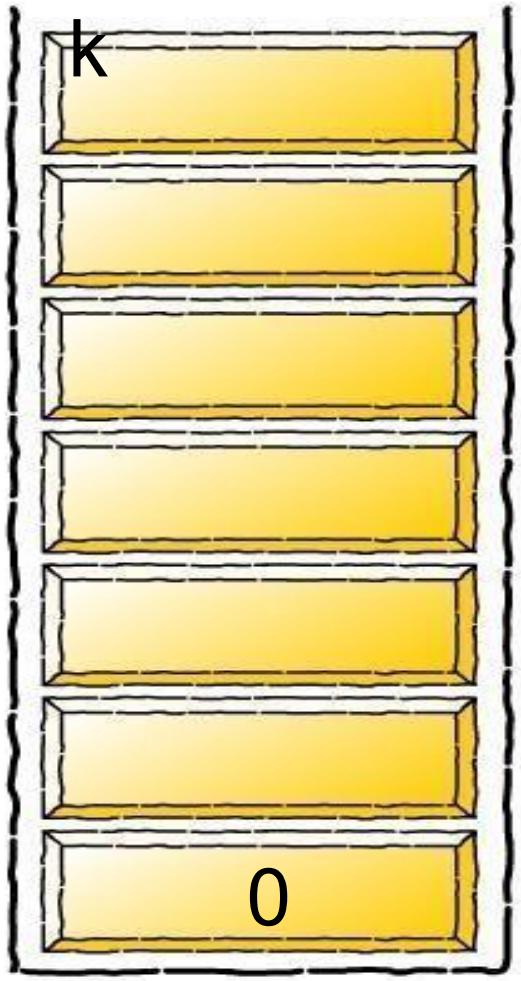
Postfix Expression

+

Token

*

Operator, POP 2 top elements from the stack and store it in opnd2 and opnd1



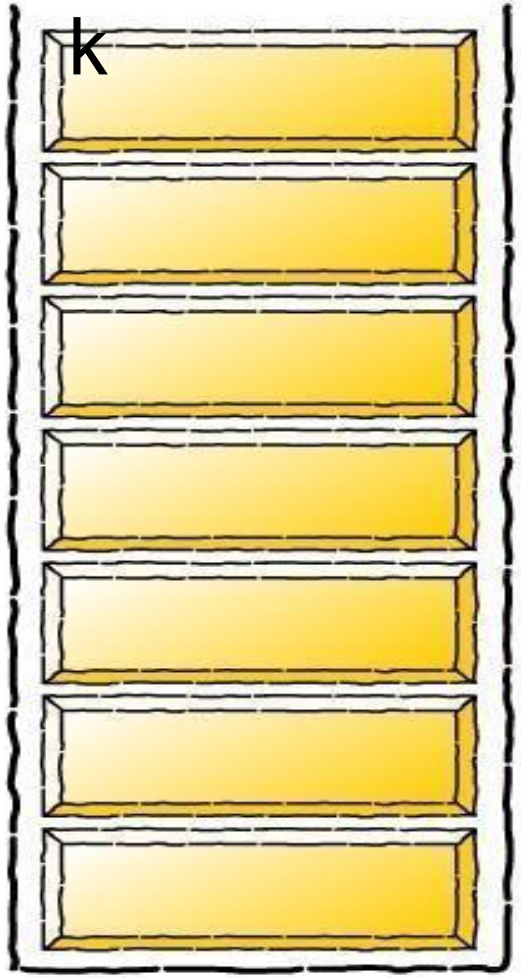
Postfix Expression

+

opnd1 opnd2 Evaluate opnd1 operator opnd2

4 2

stac
k



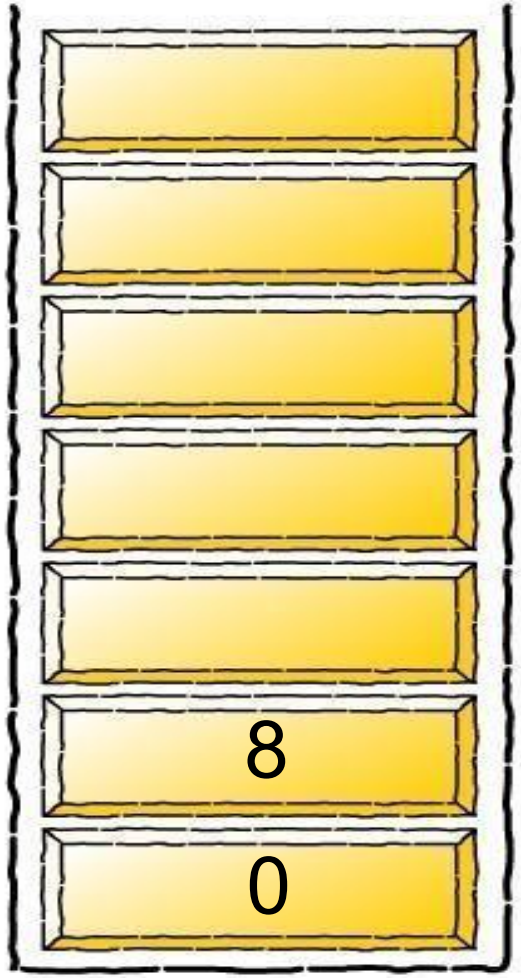
Postfix Expression

+

opnd1		opnd2	Res
4	*	2	8

PUSH Res onto stack

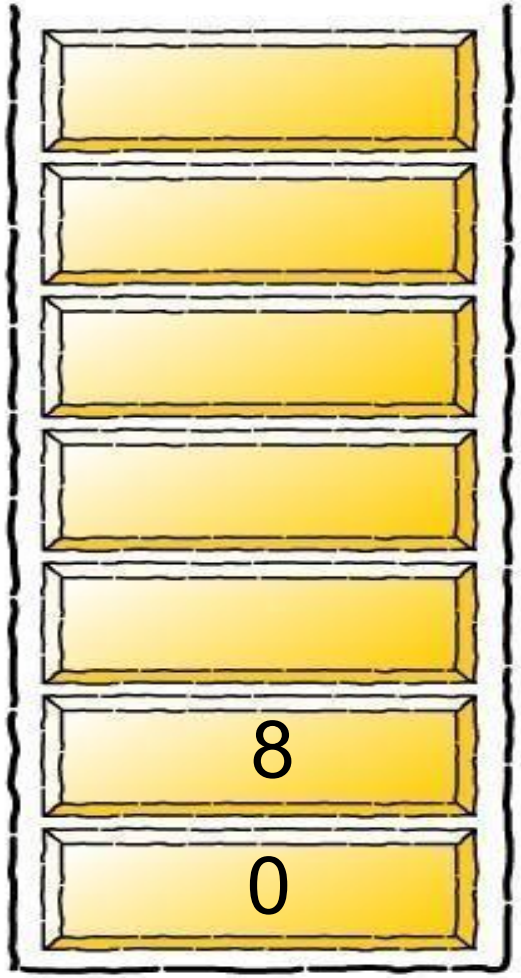
stack



Postfix Expression

+

stack



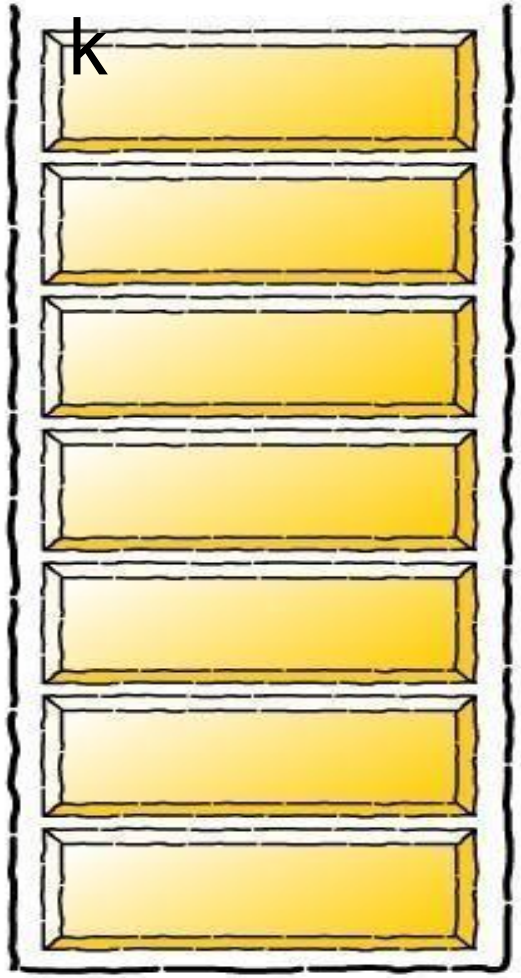
Postfix Expression

Token

+

Operator, POP 2 top elements from the stack and store it in opnd2 and opnd1

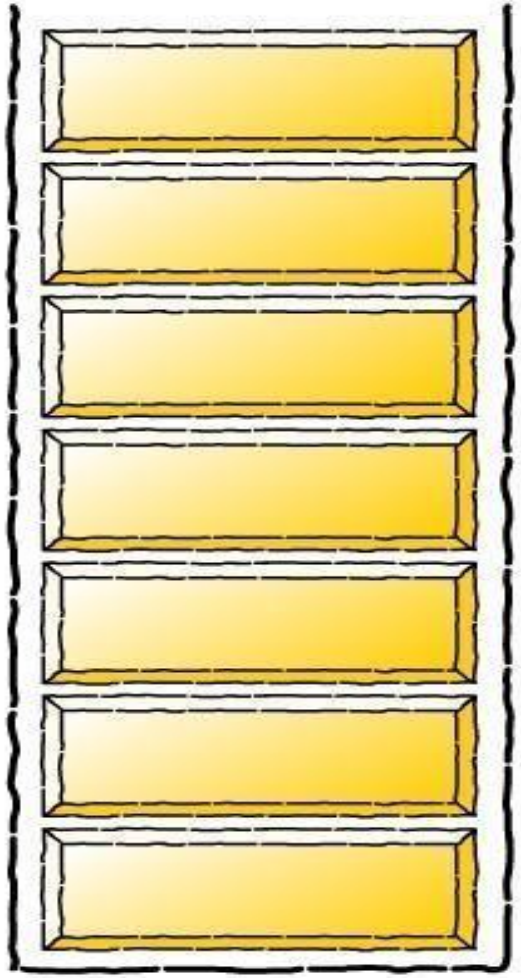
stac
k



Postfix Expression

opnd1	opnd2	Evaluate opnd1 operator opnd2
0	8	

stack

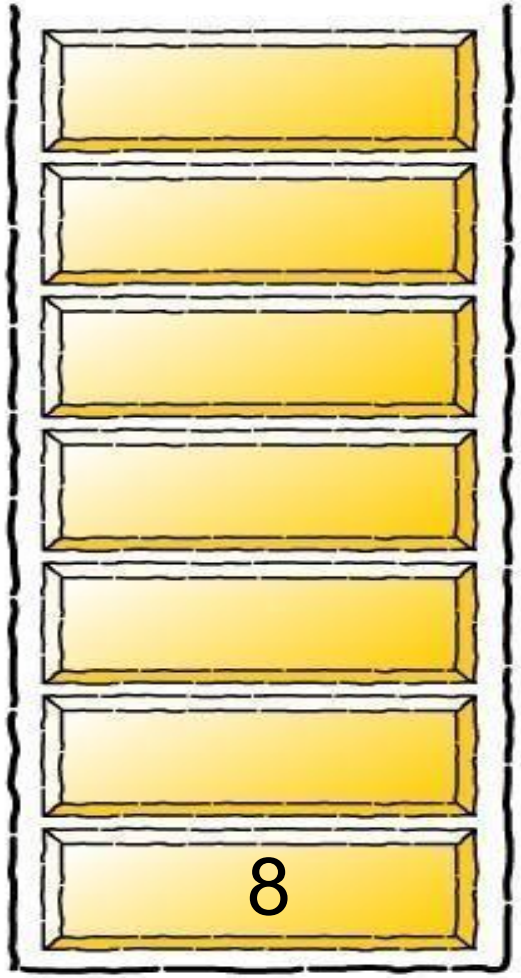


Postfix Expression

opnd1		opnd2	Res
0	+	8	8

PUSH Res onto stack

stack



Postfix Expression

Infix to Postfix -Conversion process

- Since the order of operands in the infix and postfix expression is same
- Scan the input infix expression from left to right
- During the scan ,operands are passed directly to the postfix expression as they are encountered.
- When the operators are encountered it is placed in the postfix expression depending on its precedence
- To get the higher precedence first in the postfix expression, we save the operators until we know their correct placement. To do this we follow the following procedure
  - Check the precedence of the incoming operator with the stack top.

    if precedence of incoming operator is not higher than the operator at the stack top repeatedly pop the stack and place the popped item to the postfix expression.
  - place incoming operator on the stack.
- Once you reach end of infix expression repeatedly pop the stack and place the popped item to the postfix expression till stack is empty

- In a parenthesized infix expression the expression inside the parenthesis should be converted first.
- Hence we stack the operators until we reach the right parenthesis
- As we encounter left parenthesis during scanning, we must place it on the stack what ever may be the stack top. Once left parenthesis becomes the stack top, what ever may be the operators other than the right parenthesis ,the operator should be placed on the stack
- Hence left parenthesis need to behave like high precedence when it is out from the stack and low precedence operator when it is on the stack.

Algorithm converts INFIX expression Q into POSTFIX expression P

1 push “(“ onto STACK, and append “)” to the end of Q

2 scan Q left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty

3 If an operand is encountered place it in the POSTFIX expression P

4 If an left parenthesis is encountered, push it on to the STACK

5 If an operator is encountered, call it as incoming operator, then:

a) Repeatedly pop from the STACK and place it in the P which has the same precedence or higher precedence than the incoming operator

b) b) Place the incoming operator to the STACK

[End of IF]

6 If a right parentheses is encountered

a)Repeatedly pop from the STACK and place it in the P until left parenthesis is encountered

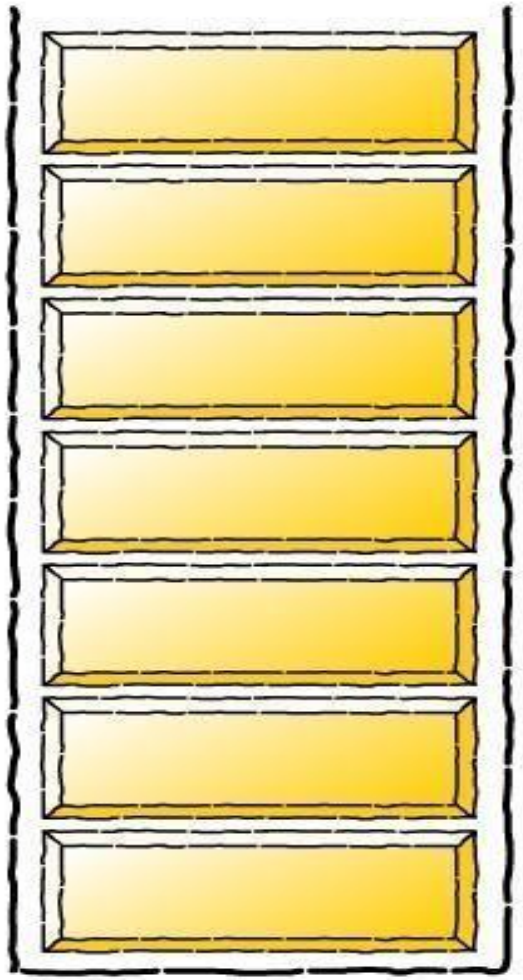
b)Remove the left parenthesis

[End of If]

[End of Step 2 loop]

- Complexity – Ø(n), where n be the number of tokens in the expression

Infix to postfix



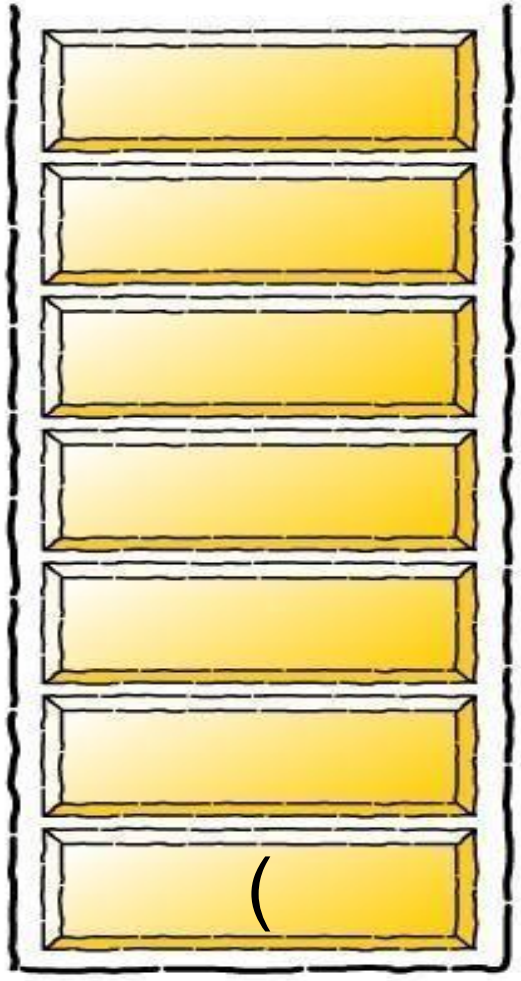
infixVect

(a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion

stackVect

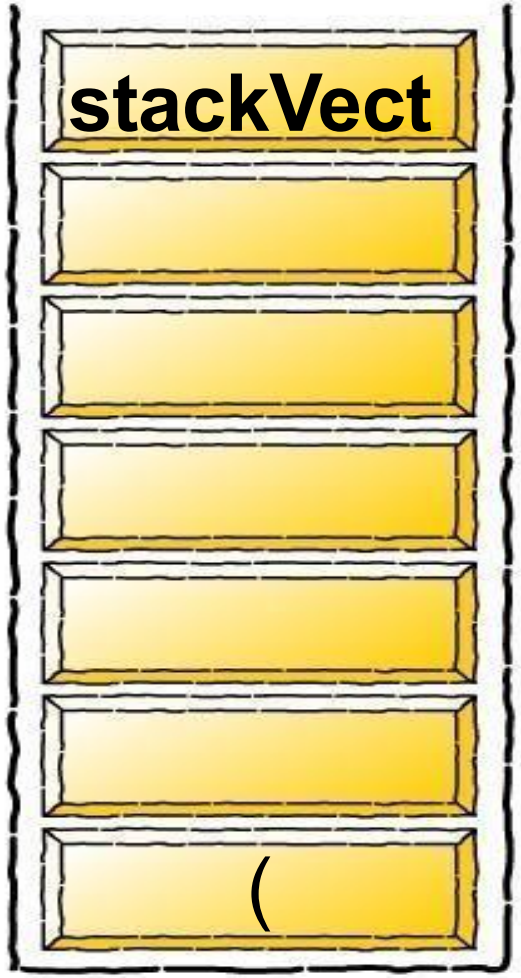


infixVect

a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion



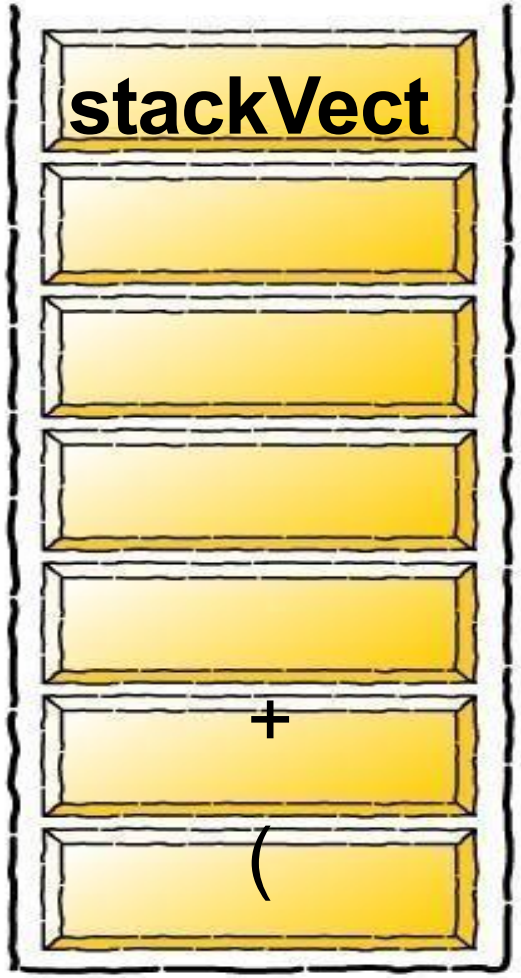
infixVect

+ b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion



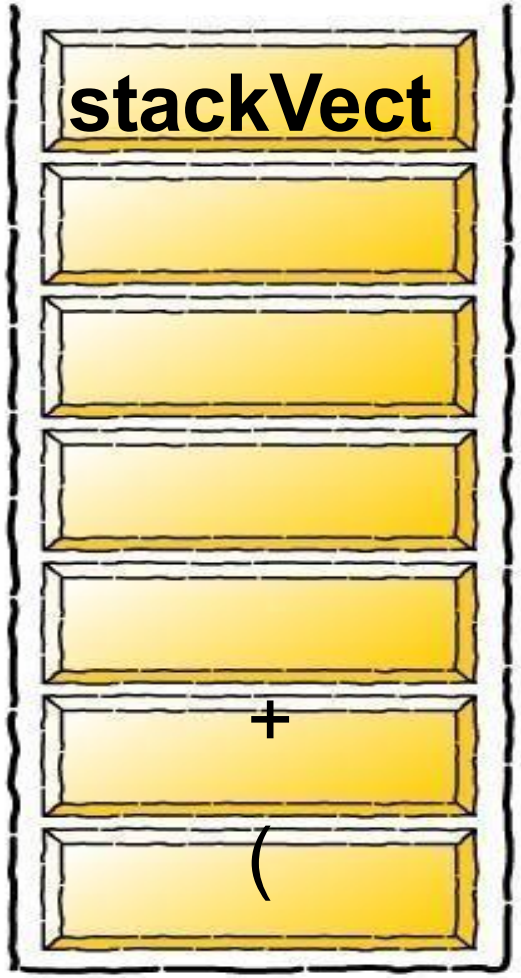
infixVect

b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion



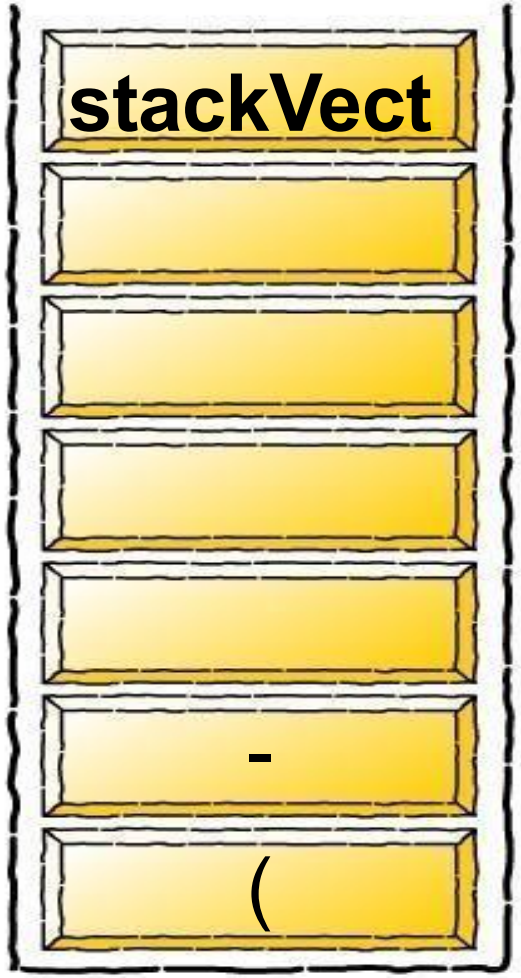
infixVect

- c) * d – (e + f)

postfixVect

a b

Infix to postfix conversion



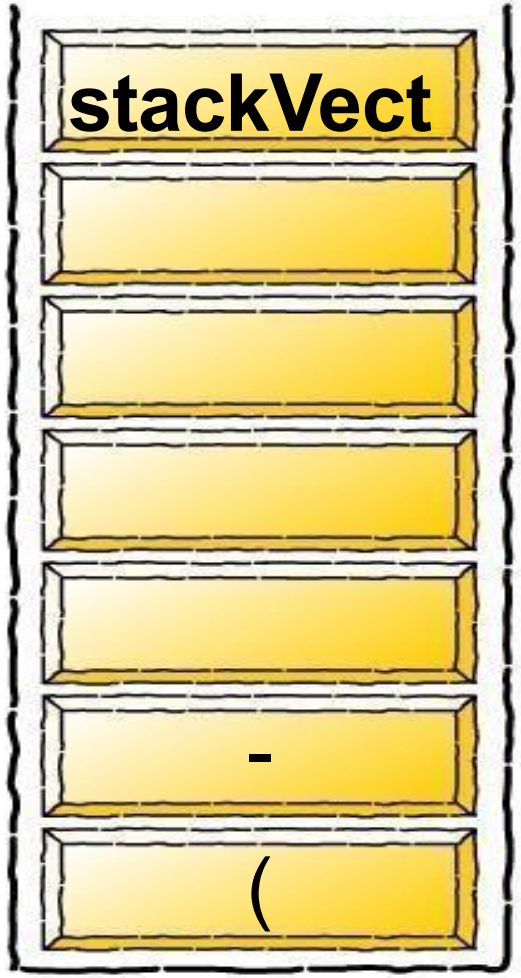
infixVect

c) * d – (e + f)

postfixVect

a b +

Infix to postfix conversion



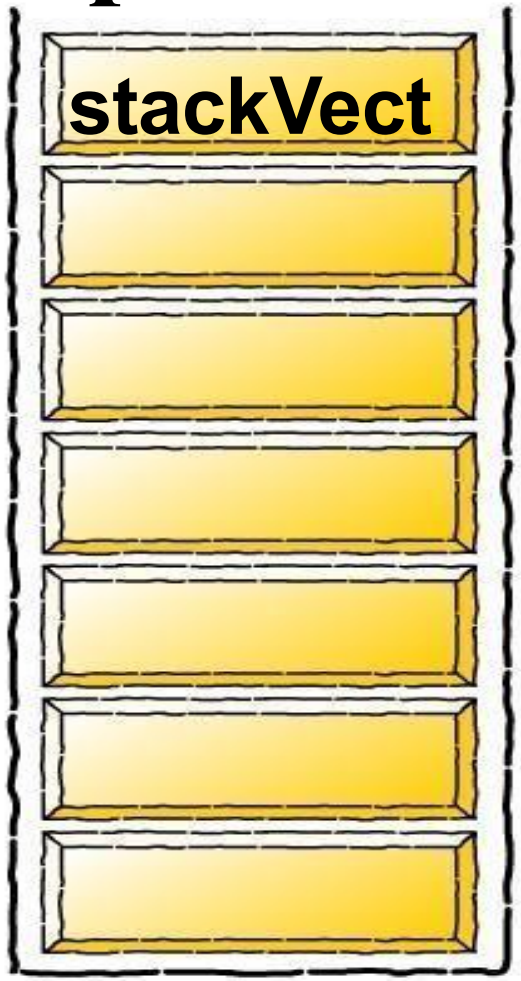
infixVect

) * d – (e + f)

postfixVect

a b + c

Infix to postfix conversion



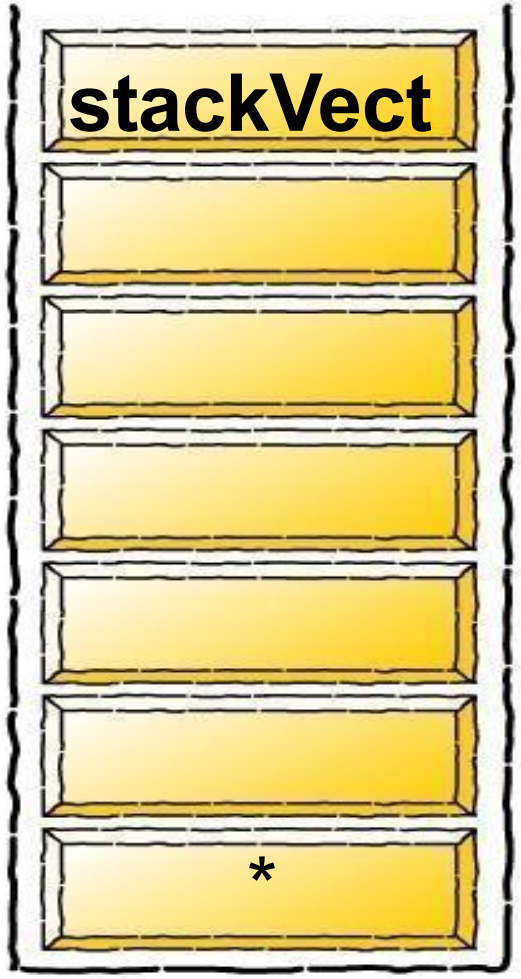
infixVect

* d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion



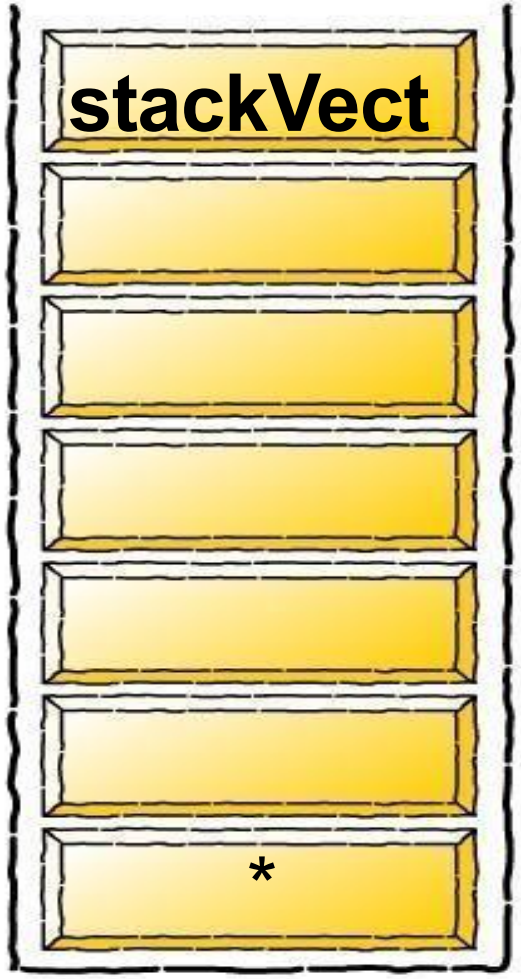
infixVect

d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion



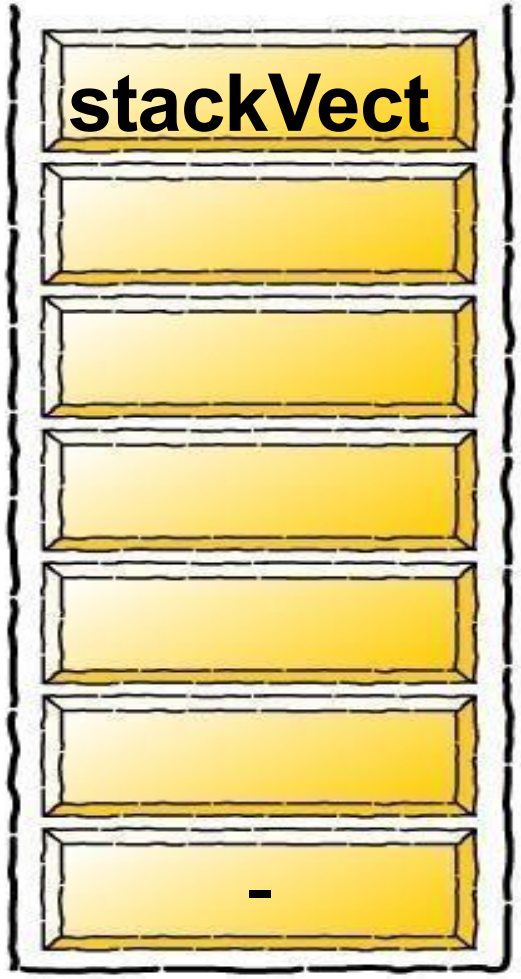
infixVect

– (e + f)

postfixVect

a b + c - d

Infix to postfix conversion



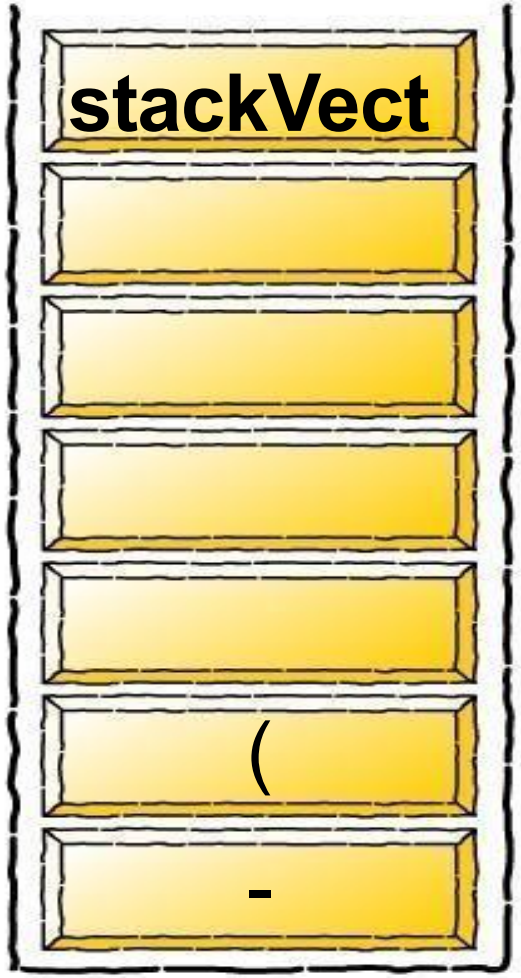
infixVect

(e + f)

postfixVect

a b + c – d *

Infix to postfix conversion



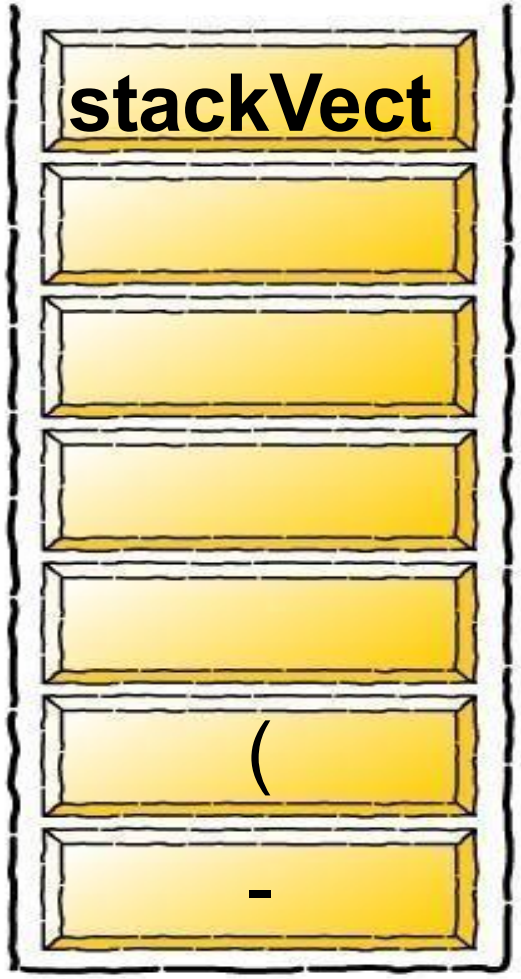
infixVect

e + f)

postfixVect

a b + c – d *

Infix to postfix conversion



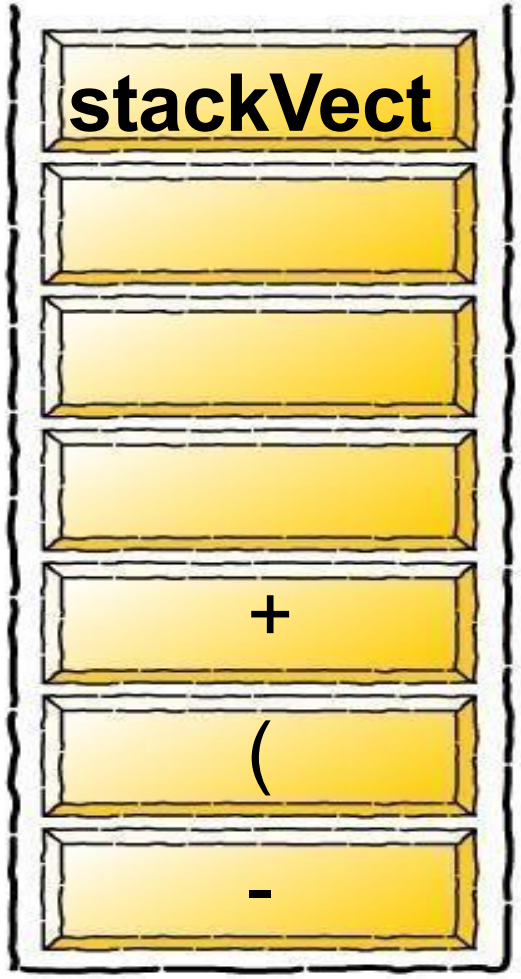
infixVect

+ f)

postfixVect

a b + c – d * e

Infix to postfix conversion



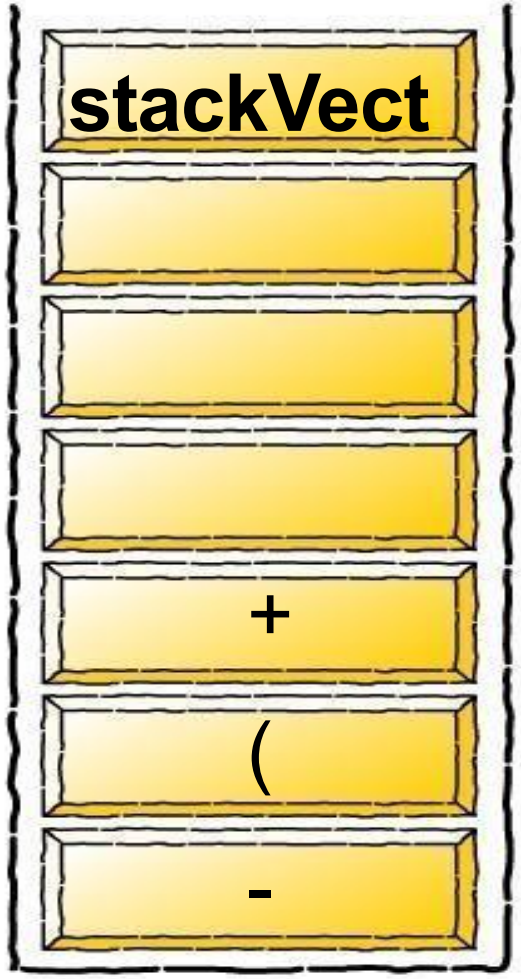
infixVect

f)

postfixVect

a b + c – d * e

Infix to postfix conversion



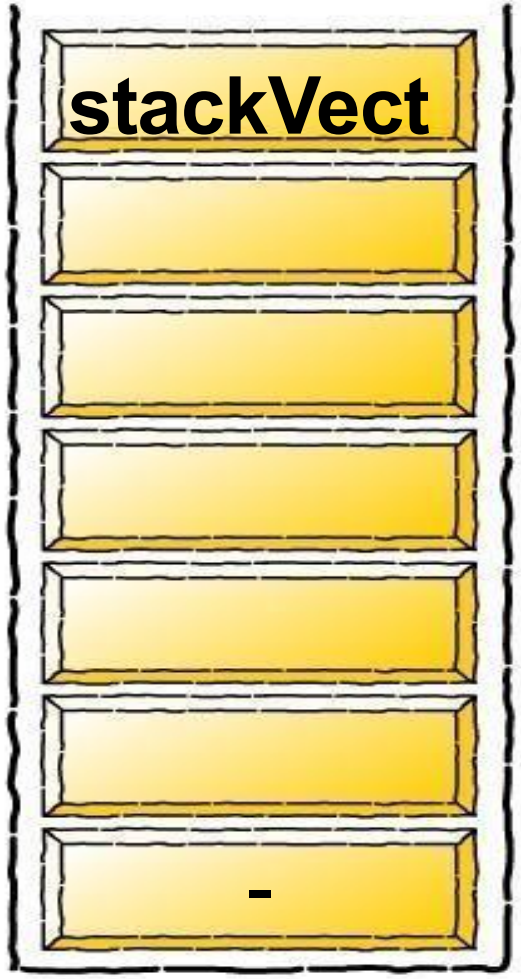
infixVect

)

postfixVect

a b + c – d * e f

Infix to postfix conversion

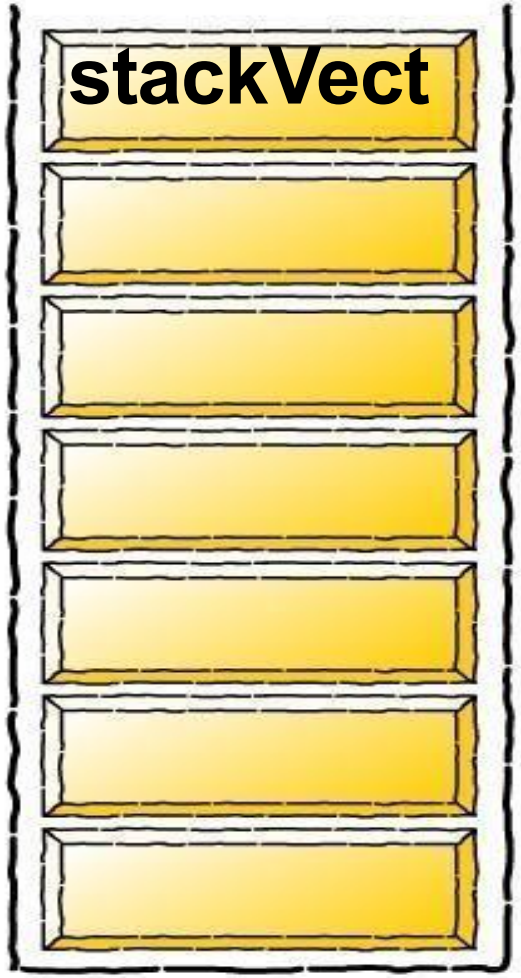


infixVect

postfixVect

a b + c – d * e f +

Infix to postfix conversion



infixVect

postfixVect

a b + c – d * e f + -

Infix to Postfix Conversion using Stack:Example

Infix Expression : A *(B + C) /D

Token	Stack [0] [1] [2]	top	Postfix
A		-1	A
*	*	0	A
(	* (	1	A
B	* (	1	A B
+	* (+	2	A B
C	* (+	2	A BC
)	*	0	ABC+
/	/	0	ABC+*
D	/	0	ABC+*D
eos		-1	ABC+*D/