Solution for Internal Assessment Test 1 – September 2019

Sub:	Data Structures & Algorithms					Sub Code:	18CS32	Branch:	CSE	
Date:	09/09/2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	3/A,B &C			OBE

Answer any FIVE FULL Questions MARKS

1 (a) Differentiate structure and union with examples. [04]

Solution:

S.no	Structure	Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies larger memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.
4	Structure example: struct student { int mark; double average; };	Union example : union student { int mark; double average; };
5	For above structure, memory allocation will be like below. int mark – 2B double average – 8B Total memory allocation = 2+8 = 10 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

(b) Explain dynamic memory allocation & de-allocation functions in C with syntax & example. [06]

Solution:

The process of allocating memory during program execution is called dynamic memory allocation.

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can‟t be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

C language offers 4 dynamic memory allocation functions. They are,

1. malloc() : malloc (number * sizeof(int));
2. calloc() : calloc (number, sizeof(int));
3. realloc() : realloc (pointer_name, number * sizeof(int));
4. free() : free (pointer_name);

1. MALLOC():

- Is used to allocate space in memory during the execution of the program.
- Does not initialize the memory allocated during execution. It carries garbage value.
- Returns null pointer if it couldn‟t able to allocate requested amount of memory.

Syntax: ptr = (cast-type*) malloc(byte-size)

Example: ptr = (int*) malloc(100 * sizeof(float));

2. CALLOC():

- calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn‟t.

Syntax: ptr = (cast-type*)calloc(n, element-size);

Example: ptr = (float*) calloc(25, sizeof(float));

3. REALLOC():

- Realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- If enough space doesn‟t exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

Syntax: ptr = realloc(ptr, x);
Example: ptr = realloc(ptr, n2 * sizeof(int));

4. FREE():

- Free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

Syntax: `free(ptr);`

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	**calloc()**
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int)); For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn‟t initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

2 Write the Knuth Morris Pratt pattern matching algorithm and apply the same to search the pattern 'abcdabcy' in the text 'abcxabcdabxabcdabcdabcy'. Demonstrate steps also. [10]

Solution:

Knuth Morris Pratt Pattern Matching Algorithm:

algorithm kunth_morris_pratt

input:

S // array of characters (String/text to be searched)

W // array of characters (sub-string/pattern to find)

output:

an integer (starting position where W is found in S)

```
1.  m=0 // the beginning of the current match in S
2.  i=0  // the position of the current character in W)
3.  T[]  //  array of integers (to track and record position)

4.  while m + i < length(S) do
        4.1 if W[i] = S[m + i] then
              if i = length(W) - 1 then
                return m
              endif
              i = i+1
        4.2 else if T[i] > -1 then
                  m = m + i - T[i]
                  i =  T[i]
        4.3       else
                  m = m + 1
                  i = 0
        4.4     endif
        4.5 endif
5.    Endwhile
6     return the length of S  // if reached here
```

Applying the algorithm to **search the pattern 'abcdabcy' in the text 'abcxabcdabxabcdabcdabcy'**

```
                    1         2
m:      012345678901234567890 12
S:      abcxabcdabxabcdabcdabcy
W:      abcdabcy
i:      01234567
```

//Mismatches on 3 so, next comparison after overlapping from 3

```
                    1         2
m:      012345678901234567890 12
S:      abcxabcdabxabcdabcdabcy
W:        abcdabcy
i:        01234567
```

//Mismatches on 3(first char) so, next comparison from 4

```
                    1         2
m:      012345678901234567890 12
S:      abcxabcdabxabcdabcdabcy
W:          abcdabcy
i:          01234567
```

//Mismatches on 10 but next ab starts at 8 so, next comparison after overlapping from 8

```
                    1         2
m:      012345678901234567890 12
S:      abcxabcdabxabcdabcdabcy
W:              abcdabcy
i:              01234567
```

//Mismatches on 10 so, next comparison after overlapping from 10

```
                    1               2
m:      01234567890123456789012
S:      abcxabcdabxabcdabcdabcy
W:                  abcdabcy
i:                  01234567
```

//Mismatches on 10(first char) so, next comparison from 11

```
                    1               2
m:      01234567890123456789012
S:      abcxabcdabxabcdabcdabcy
W:                   abcdabcy
i:                  0123456 7
```

//Mismatches on 18 but next ab starts at 15 so, next comparison after overlapping from 15

```
                    1               2
m:      01234567890123456789012
S:      abcxabcdabxabcdabcdabcy
W:                        abcdabcy
i:                       01234567
```

//All characters of pattern match starting from 15 – Found

3 (a) Demonstrate bubble sort with an array of 5 integers. [05]

Solution:

Algorithm for Bubble sort: Bubble_Sort(A[], N)

```
Step1 : Repeat for p = 1 to N-1
        Begin
Step2 :     Repeat for j = 1 to N-p
            Begin
Step3 :         if (A[j] < A[j-1])
                    Swap ( A[j], A[j-1]);
    End for
    End for
```

Exit

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

(b) Write a fast transpose algorithm or function to transpose a sparse matrix using triplets. [05]

Solution:

Fast Transpose Algorithm to transpose the given sparse matrix:

1. Input Matrix is A[6][5]

2. For r= 1 to 6
   2.1 For c = r+1 to 5 // for replacing values of A[r][c] with A[c][r]
   temp=A[r][c]
   A[r][c]=A[c][r]
   A[c][r]=temp
   2.2 Next c
3. Next r

In given sparse matrix, Total rows = 6, Total columns = 5 & Total non-zero elements = 8

Converting in triplets:

First/header row of triplets will contain 6, 5 and 8 and rest of the triplets will have corresponding row number, column number of non-zero element and non-zero element itself

6	5
8	
0	0
10	
0	3
25	
1	1
23	
1	4

Now Transposed sparse matrix: in 5 rows and 6 columns

10	0	0	42	0
0				
0	23	0	0	0
0				
0	0	0	0	0

Again transposing for triplets

5	6
8	
0	0
10	
0	3
42	
1	1
23	
2	5

4 (a) Define recursion. What are the properties of recursive procedure? [05]

Solution:

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion though simpler alternatives are available. It is because recursion is elegant to use though it is costly in terms of time and space.

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base case** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **General case** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

(b) Write a C program to solve the "Tower of Hanoi" problem using recursion. [05]

Solution:

```
int main ()
{
        int   n;
        printf("Enter number of discs: ");
        scanf("%d",&n);
        towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
        return 0;
}
void towers_of_hanoi (int n, char *a, char *b, char *c)
{
        if (n == 1)
        {
                ++cnt;
                printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
                return;

        }
        else
        {

                towers_of_hanoi (n-1, a, c, b);  ++cnt;
                printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
                 towers_of_hanoi (n-1, b, a, c);
                 return;
        }
 }
```

Output of the program:

```
Enter the number of discs: 3
1: Move disk 1 from tower 1 to tower  3.
2: Move disk  2  from tower  1  to tower  2.
3: Move disk  1  from tower  3  to tower  2.
4: Move disk 3 from tower 1 to tower  3.
5: Move disk 1 from tower 2 to tower  1.
6: Move disk  2  from tower  2  to tower  3.
7: Move disk  1  from tower  1  to tower  3.
```

5 Define stack. Write a C program demonstrating the various stack operations, including cases for overflow and underflow of stacks. [10]

Solution:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists. As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.
The two basic operations associated with stacks are:

- **Push**: is the term used to insert an element into a stack.
- **Pop**: is the term used to delete an element from a stack.

.
All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

```
#include<stdio.h>
#include<process.h>
#include<stdlib.h>
#define MAX 5          //Maximum number of elements that can be stored
int top=-1,stack[MAX];
void push();
void pop();
void display();

void main()
{
        int ch;
        while(1)        //infinite loop, will end when choice will be 4
        {
                printf("\n*** Stack Menu ***");
                printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
                printf("\n\nEnter your choice(1-4):");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: push();
```

```
                                        break;
                        case 2: pop();
                                        break;
                        case 3: display();
                                        break;
                        case 4: exit(0);

                        default: printf("\nWrong Choice!!");
                }
        }
}

void push()
{
        int val;

        if(top==MAX-1)
        {
                printf("\nStack is full!!");
        }
        else
        {
                printf("\nEnter element to push:");
                scanf("%d",&val);
                top=top+1;
                stack[top]=val;
        }
}
 void pop()
{
        if(top==-1)
        {
                printf("\nStack is empty!!");
        }
```

```
        else
        {
                printf("\nDeleted element is %d",stack[top]);
                top=top-1;
        }
}

void display()
{
        int i;

        if(top==-1)
        {
                printf("\nStack is empty!!");
        }
        else
        {
                printf("\nStack is...\n");
                for(i=top;i>=0;--i)
                        printf("%d\n",stack[i]);
        }
}
```

Output

```
*** Stack Menu ***
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1
Enter element to push:3
```

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):1

Enter element to push:6

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):3

Stack is…

6

3

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):2

Deleted element is 6

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):3

Stack is…

3

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):2

Deleted element is 3

*** Stack Menu ***

1.Push

2.Pop

3.Display

4.Exit

Enter your choice(1-4):2

Stack is empty!!

6 (a) Write a function to evaluate postfix expression [05]

Solution:

```
#include<stdio.h>
#include<ctype.h>
# define MAXSTACK 100
# define POSTFIXSIZE
int stack[MAXSTACK];
int top = -1
void push(int item)
{
        if(top >= MAXSTACK -1)
        {
                printf("stack over flow");
                return;
        }
        else
        {
                top = top + 1 ;
```

```
                stack[top]= item;
        }
}

int pop()
{
        int item;
        if(top <0)
        {
                printf("stack under flow");
        }
        else
        {
                item = stack[top];
                top = top - 1;
                return item;
        }
}

void EvalPostfix(char postfix[])
{

        int i ;
        char ch;
        int val;
        int A, B ;

        /* evaluate postfix expression */
        for (i = 0 ; postfix[i] != ')'; i++)
        {
                ch = postfix[i];
                if (isdigit(ch))
                {
```

```
                push(ch - '0');
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/')
        {
                A = pop();
                B = pop();

                switch (ch) /* ch is an operator */
                {
                        case '*':
                        val = B * A;
                        break;
                        case '/':
                        val = B / A;
                        break;
                        case '+':
                        val = B + A;
                        break;
                        case '-':
                        val = B - A;
                        break;
                }
                        push(val);
        }
    }
    printf( " \n Result of expression evaluation : %d \n", pop()) ;
}
int main()
{
    int i ;
    char postfix[POSTFIXSIZE];
```

```
        for (i = 0 ; i <= POSTFIXSIZE - 1 ; i++)
        {
                scanf("%c", &postfix[i]);
        }
                EvalPostfix(postfix);
        return 0;
}
```

Output:

Enter postfix expression : 456*+

Result of expression evaluation : 34

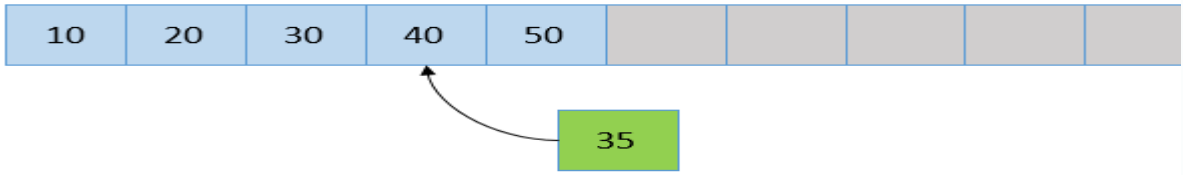
(b) Explain with example insertion and deletion at a valid position in an array. [05]

Solution:

To insert element in array

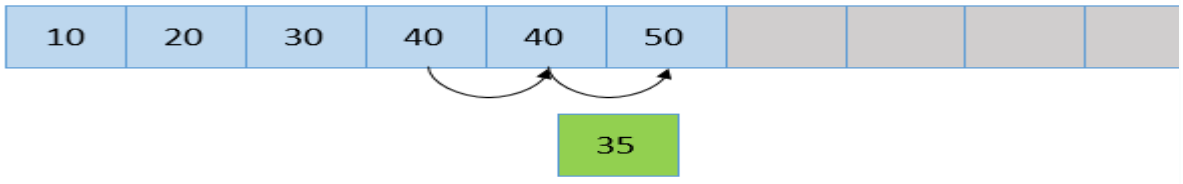
1. Input size and elements in array. Store it in some variable say size and arr.

2. Input new element and position to insert in array. Store it in some variable say num and pos.

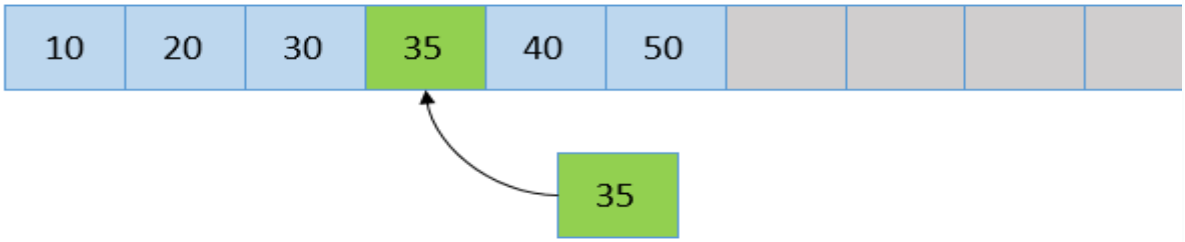


3.To insert new element in array, shift elements from the given insert position to one position right. Hence, run a loop in descending order from size to pos to insert. The loop structure should look like for(i=size; i>=pos; i--).

Inside the loop copy previous element to current element by arr[i] = arr[i - 1];.



4.Finally, after performing shift operation. Copy the new element at its specified position i.e. arr[pos-1]=num;.



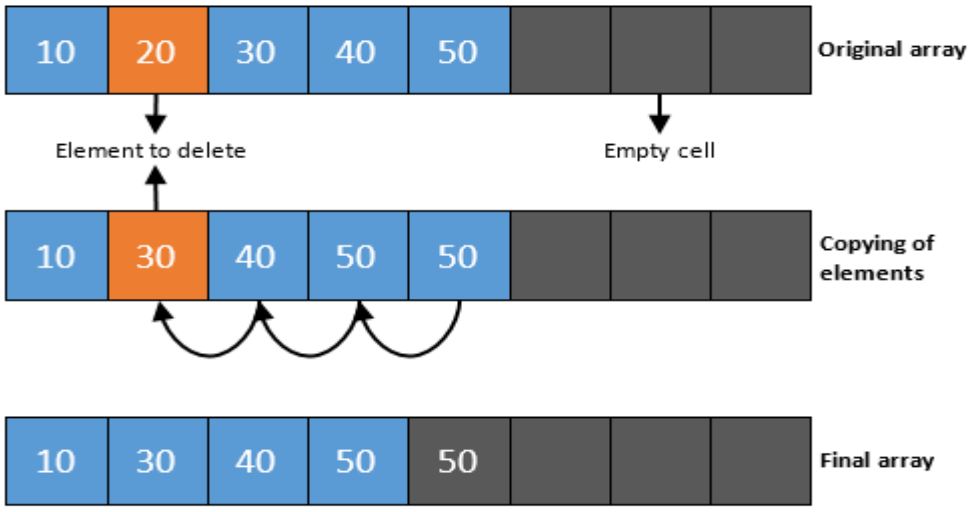
To delete element from an array

1. Move to the specified location which you want to remove in given array.

2. Copy the next element to the current element of array. Which is you need to perform array[i] = array[i + 1].

3. Repeat above steps till last element of array.

4. Finally decrement the size of array by one.

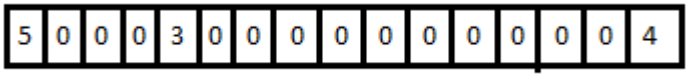


7 (a) Consider two polynomials, [05]
$A(x) = 4x^{15} + 3x^4 + 5$ and $B(x) = x^4 + 10x^2 + 1$
Show diagrammatically how these two polynomials can be stored in a 1- D array. Also give its C
representation for initialization.
Solution:

Diagrammatic representation of given polynomial in one-dimensional array:

Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the index/subscript (element number of the array) of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array.

$A(x) = 4x^{15} + 3x^4 + 5$

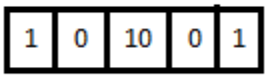


Exponents/Indices 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Now C representation:

int Ax [16] = { 5,0,0,0,3,0,0,0,0,0,0,0,0,0,0,4};

$B(x) = x^4 + 10x^2 + 1$



Exponents/Indices 0 1 2 3 4

And C representation:

int Bx [5] = { 1,0,10,0,1};

(b) Write an algorithm for Ackermann's function. Evaluate A(1,2) using Ackermann's function. [05]

Solution:

Ackerman's function is a classic example of recursive function. **It is well defined total computable function but not primitive recursive.** Earlier before 1995 the concept was that all computable functions are primitive recursive also.
Ackerman's function grows exponentially:

Ackerman's A(x,y) function is defined for integer x and y:

$$a(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ a(x-1,1) & \text{if } y = 0 \\ a(x-1, a(x,y-1)) & \text{otherwise} \end{cases}$$

We can implement in C function:

```
int ackermann(int x, int y)
{
        if(x==0)  return y+1;
        if(y==0) return ackermann(x-1,1);
        return ackermann(x-1,ackermann(x,y-1));
}
```

If x=1 and y = 2, the result will be 4 and total function calls in recursion will be 10 times.

8 Convert the infix expression ((a/(b-c+d))*(e-a)*c) to postfix expression and evaluate that postfix expression for given data a=6, b=3, c=1, d=2, e=4 (using stack representation). [10]

Solution:

Convert the infix expression ((a/(b-c+d))*(e-a)*c) to postfix expression:

Stack	**Top of the stack**	**Symbol**	**Postfix**	**Operation**
#	#	(		Push into stack.
# (	(	(		Push into stack.
# ((	(	a	a	Place it in postfix.
# ((	(	/	a	/ has higher precedence push into stack.
# ((/	/	(	a	Push into stack.
# ((/ (	(	b	a b	Place it in postfix.
# ((/ (	(	-		- has higher precedence push into stack.
# ((/ (-	-	c	a b c	Place it in postfix.
# ((/ (-	-	+	a b c -	- has equal precedence to + pop from stack and place it in postfix, push + into stack.
# ((/ (+	+	d	a b c - d	Place it in postfix.
# ((/ (+	+	)	a b c – d +	Pop until we get matching '(' and decrement top.
# ((/	/	)	a b c – d + /	Pop until we get matching '(' and decrement top.
# (	(	*	a b c – d + /	* has higher precedence push into stack.
# (*	*	(	a b c – d + /	Push into stack.
# (* (	(	e	a b c – d + / e	Place it in postfix.
# (* (	(	-	a b c – d + / e	- has higher precedence push into stack.
# (* (-	-	a	a b c – d + / e a	Place it in postfix.
# (* (-	-	)	a b c – d + / e a -	Pop until we get matching '(' and decrement top.
# (*	*	*	a b c – d + / e a - *	* has equal precedence to * pop from stack and place it in postfix, push + into stack.
# (*	*	c	a b c – d + / e a - * c	Place it in postfix.
# (*	*	)	a b c – d + / e a - * c*	Pop until we get matching '(' and decrement top.

Postfix Expression: a b c – d + / e a - * c*

Evaluation of postfix expression: a b c – d + / e a - * c* = 631-2+/46-*1*

Symbol	**OP2**	**OP1**	**Stack Contents**	**Operation**
6			6	Push into stack.
3			6 3	Push into stack.
1			6 3 1	Push into stack.
-	1	3	6 2	3-1=2 ,push into stack.
2			6 2 2	Push into stack.
+	2	2	6 4	2+2=4 , push into stack.
/	4	6	1	6/4=1 , push into stack.
4			1 4	Push into stack.
6			1 4 6	Push into stack.
-	6	4	1 -2	4 – 6 = -2, push into stack.
*	-2	1	-2	1*-2= -2, push into stack.
1			-2 1	Push into stack.
*	1	-2	-2	-2 * 1 = -2, push into stack.

Result of postfix expression: a b c – d + / e a - * c* = 631-2+/46-*1*= -2.