Department of Information Technology :: VRSEC

III/IV B. Tech - VI th Semester

20IT6205C - INTRODUCTION TO DATA STRUCTURES

A.Y: 2023-2024

Active Learning in Programming & Coding

Active learning is a teaching and learning strategy that involves students actively participating and being motivated. In programming and coding, active learning can include techniques like: Writing code, Coding challenges, explaining concepts to others, Coding projects, and Code executable blocks.

Active learning can also be used in machine learning, where it's an iterative process that uses a machine learning model to select examples to label. The model is retrained on the new dataset after annotation, and then selects more data to label until it reaches a stopping criterion. This process can use fewer training examples to achieve better optimization.

ACTIVE LEARNING

Name of the Faculty: Dr.Y.Sangeetha	**Designation:** Associate Professor	**Subject:** Introduction to Data Structures
Year/ Semester: 3rd Year, VIth Sem	**Branch:** ECE, EIE, & Civil	**Topics:**
Name of the Activity: Active Learning in Programming & Coding	**Date:** 24-01-2024	**No. of students attended:** 40

Objective of the Activity:

Active Learning in Programming & Coding classroom model helps to improve critical thinking and strong technical competency in their profession.

Execution Plan:

Time management: Class time: 50mins

- Student construction coding : 45 mins
- Teacher summary : 5 mins

Expected Outcomes:

The IT industry demands for graduates with sound technical competency in programming domain. This course is very important which play vital role for creating the interest among students to learn the programming languages.

Introduction to Data Structures

24-01-24

Activity - I

Topic - Stack with Infix to prefix notations.

Batch no - 5.

→ **Stack:** A stack is a Linear data structure that follows the LIFO (Last in, First out) principle that allows operations like insertion and deletion from one end of the stack i.e. Top.

conversion of an Infix expression to a prefix expression:

Step 1:

Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.

step 2:

obtain the post-fix expression of the infix expression.

step3:

Reverse the postfix expression to prefix expression.

Example for infix to prefix conversion

(1) Input infix expression : a * (b+c+d)

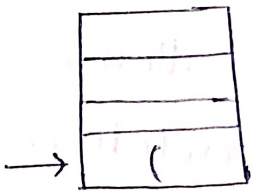
reverse the expression :)d+c+b(* a

change the parenthesis : (d+c+b) * a

(2) '(' is open paranthesis push it into the stack

(d+c+b) * a
↑

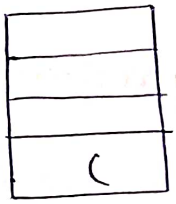
stack



postfix

(3) (d+c+b) * a
↑
'd' is operand
append it to postfix expression.

stack

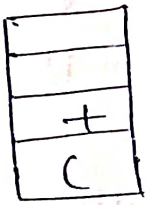


postfix

d

(4) (d+c+b) * a
↑
'+' is an operator
PUSH it into stack

stack

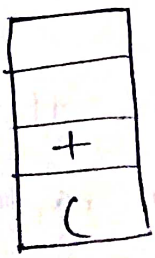


postfix

d

(5) (d+c+b) * a
↑
'c' is an operand
append it into postfix expression

stack



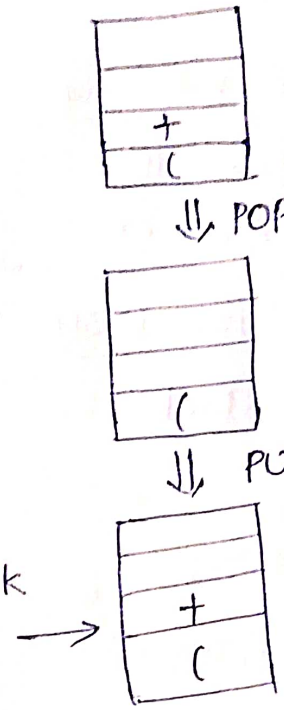
postfix

dc

(6) (d+c+b)*a

↑

'+' is an operator. top of stack is '+' which has equal precedence, so POP it and append to the postfix expression. PUSH '+' in the stack



postfix

dc

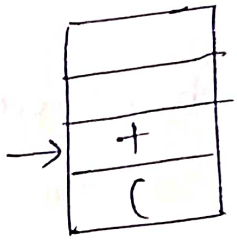
dc+

dc+

(7) (d+c+b)*a

↑

'b' is operand. append into post-fix



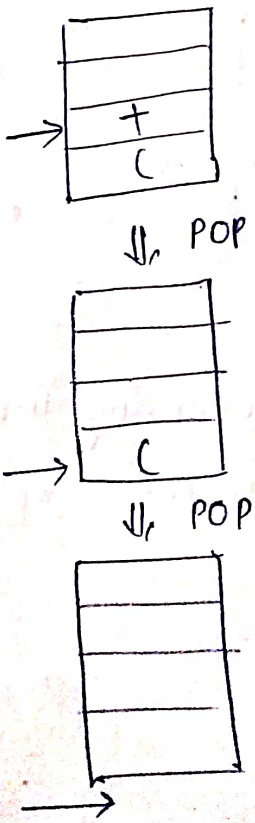
postfix

dc+b

(8) (d+c+b)*a

↑

')' is close paranthesis. pop all operators form stack until matching '(' is found & append to the postfix.



postfix

dc+b

dc+b+

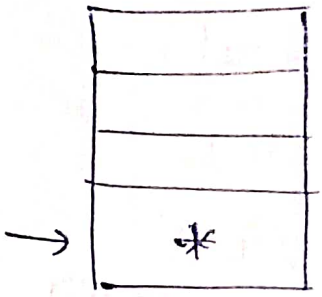
(9) (d+c+b) * a
↑

stack

postfix

dc+b+

"*" is an operator. so, pop all operators from the stack with equal or higher precedence. PUSH '*' into the stack.



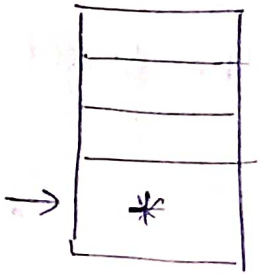
(10) (d+c+b) * a
↑

stack

postfix

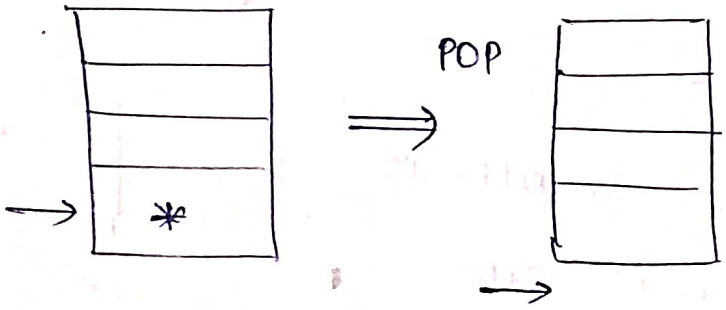
dc+b+a

`a' is operand. append it to the postfix expression.



(11) All symbols have been Scanned . pop from stack and append to the postfix expression.

stack



postfix : dc+b+a*

Reversing the postfix expression gives

Prefix expression — * a+b+cd

```
def isOperator(c):
    return (not (c >= 'a' and c <= 'z') and not(c >= '0' and c <= '9') and not(c >= 'A' and c <= 'Z'))


def getPriority(C):
    if (C == '-' or C == '+'):
        return 1
    elif (C == '*' or C == '/'):
        return 2
    elif (C == '^'):
        return 3
    return 0



def infixToPrefix(infix):
    operators = []
    operands = []

    for i in range(len(infix)):

        if (infix[i] == '(' ):
            operators.append(infix[i])

        elif (infix[i] == ')'):
            while (len(operators)!=0 and (operators[-1] != '(' )):
                op1 = operands[-1]
                operands.pop()
                op2 = operands[-1]
                operands.pop()
                op = operators[-1]
                operators.pop()
                tmp = op + op2 + op1
```

```
            operands.append(tmp)
        operators.pop()
    elif (not isOperator(infix[i])):
        operands.append(infix[i] + "")

    else:
        while (len(operators)!=0 and getPriority(infix[i]) <= getPriority(operators[-1])):
            op1 = operands[-1]
            operands.pop()

            op2 = operands[-1]
            operands.pop()

            op = operators[-1]
            operators.pop()

            tmp = op + op2 + op1
            operands.append(tmp)
        operators.append(infix[i])

while (len(operators)!=0):
    op1 = operands[-1]
    operands.pop()

    op2 = operands[-1]
    operands.pop()

    op = operators[-1]
    operators.pop()

    tmp = op + op2 + op1
```

```
        operands.append(tmp)
    return operands[-1]


while(1):
    s = input("Infix Expression : ")
    print("Prefix Expression : ", infixToPrefix(s))
```

```
Infix Expression : (A+B)*(C+D)
Prefix Expression :  *+AB+CD
Infix Expression : A+B*C+D
Prefix Expression :  ++A*BCD
Infix Expression : A*B+C*D
Prefix Expression :  +*AB*CD
Infix Expression : A+B+C+D
Prefix Expression :  +++ABCD
```


19:12 24 0124
ZAMF/191/22 150380

Stacks using linked lists :-

→ storage requirement of linked list representation of stack with n elements is O(n). and time requirement is O(1).

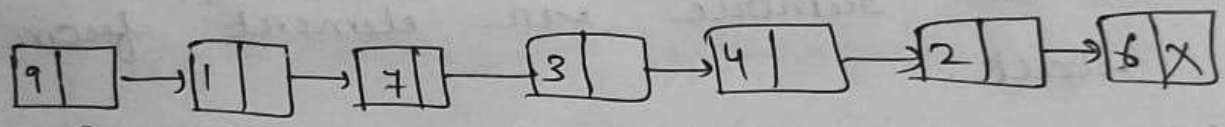
→ Node

stores data

stores address of next node

→ Start pointer of linked list is called TOP. If TOP = NULL, then stack is empty.

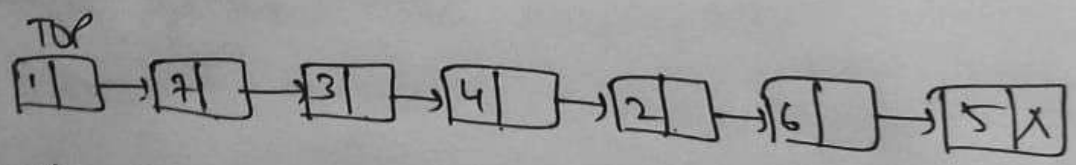
Linked stack



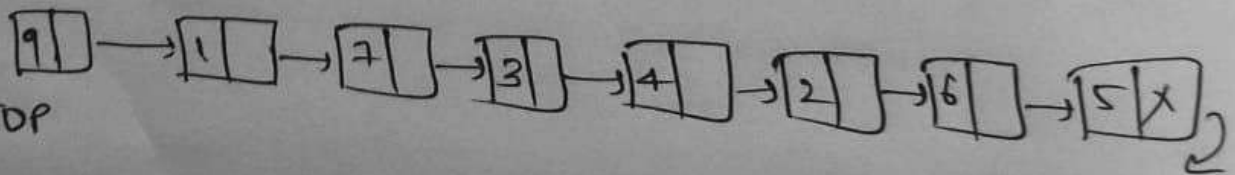
→ Push operation :

Element is added to topmost position of stack.

Ex



To insert 9,

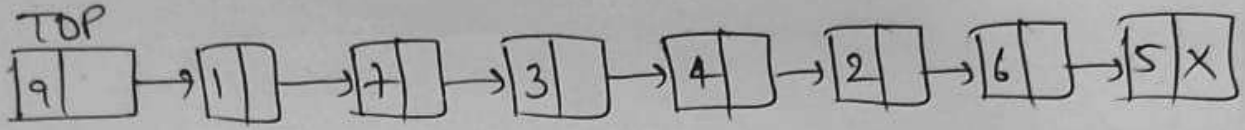


linked stack after inserting a new node(9).

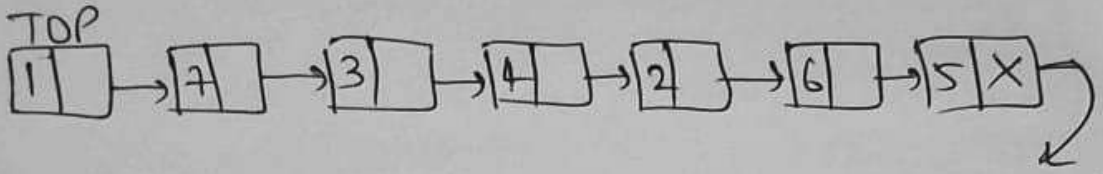
→ Pop operation :

If stack is empty, no more deletions can be done. (underflow).

Ex:



To remove 9,



linked stack after deletion of topmost element

Algorithm :-

Push operation :-

```
begin
if stack is full
   return
endif
else
   increment top
   stack(top) assign value
endelse
end procedure
```

Pop operation :-

```
begin
if stack is empty
   return
endif
else
 store value of stack(top)
 decrement top
 return value
endelse
end procedure
```

⇒ Advantages of Stack:

* Easy implementation :- Stack data structure is easy to implement using arrays (or) Linked Lists
* Efficient memory utilization:-
  Stack uses a contiguous block of memory
* Fast access time :- Stack data structure, provides fast access time for adding and removing as the elements.
* Supports backtracking:- stack data structure supports backtracking algorithm.

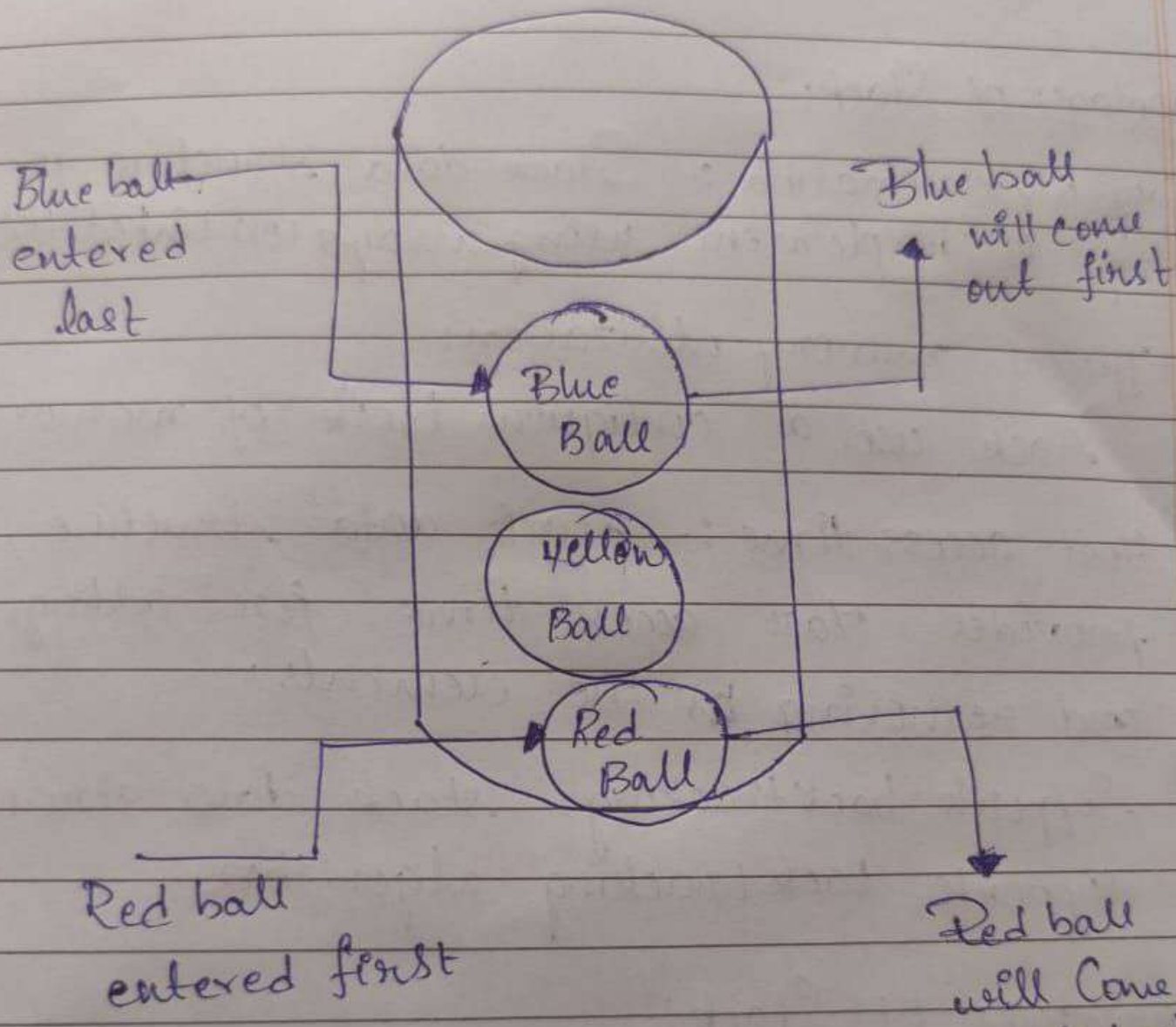
⇒ Disadvantages of Stack:-

* Limited Capacity
* No random access
* Memory management

⇒ LIFO :- (Last in first Out):-

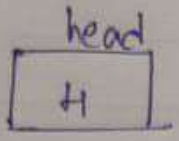
The element that is inserted last will come out first.

Ex:- Take a pile of plates kept on top of each others. So if we want the last placed plate in the middle plate should be removed.

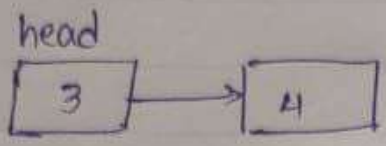

Blue ball entered last
Blue ball will come out first
Blue Ball
Yellow Ball
Red Ball
Red ball entered first
Red ball will Come out last

PUSH :-

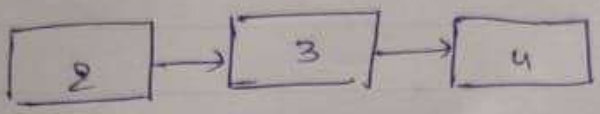
Inserting 4



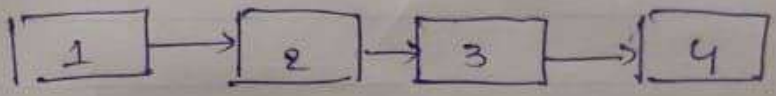
Inserting 3



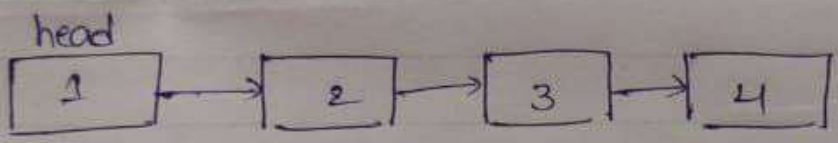
Inserting 2



Inserting 1

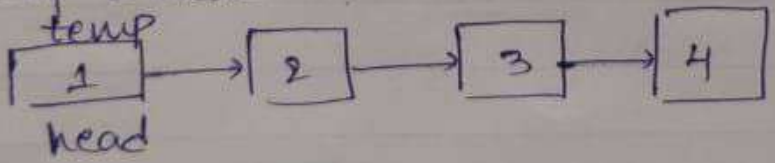


POP :-

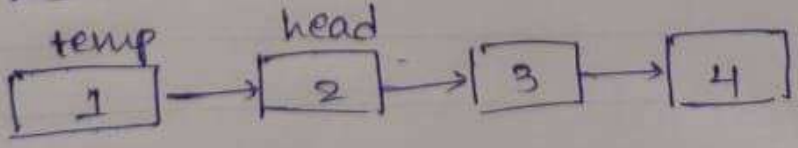


As the head is not null and head → next is not NULL, base case fails and we proceed.

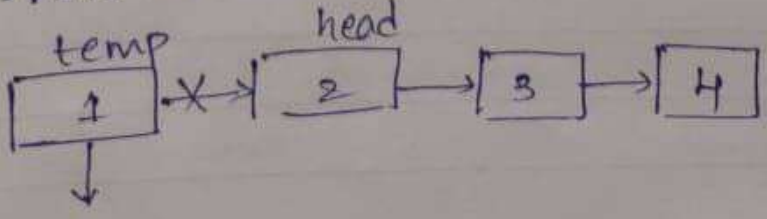
Node * temp = head



head = head → next



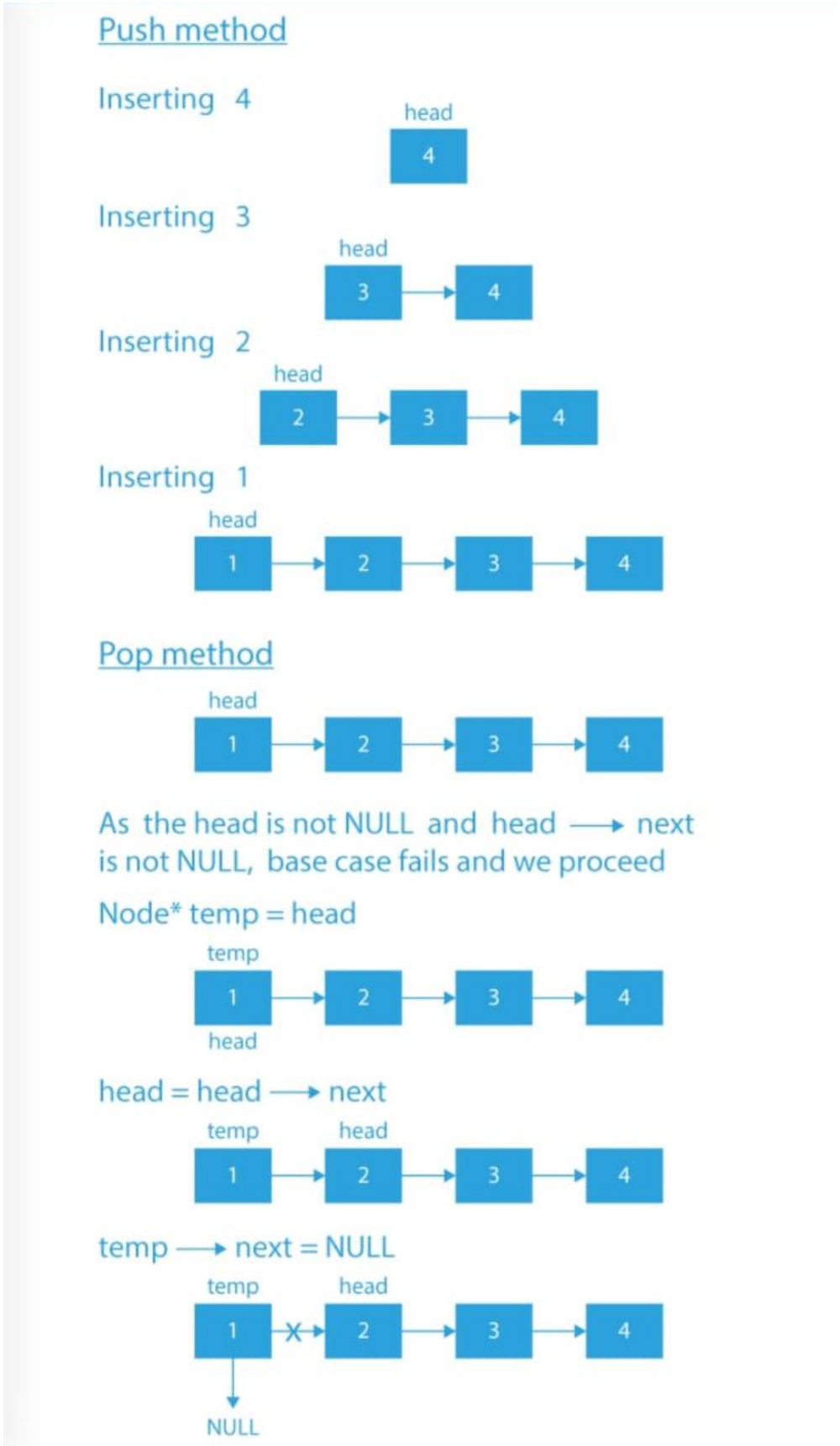
temp → next = NULL



```
#include<stdio.h>
#include<conio.h>
struct Node
{
    int data;
    struct Node *next;
} *top=NULL;
void push (int);
void pop();
void display();
void main()
{
   int choice,value;
   clrscr();
   printf("\n:: Stack using Linked List ::\n");
   while(1) {
        printf("\n*** MENU *** \n");
        printf("1. Push \n 2. Pop\n3. Display \n 4. Exit\n");
        printf("Enter your choice:");
        scanf("%d",& choice);
        switch(choice) {
          case1:printf("Enter value to be Insert:");
                scanf("%d",& value);
                push(value);
                break;
          case 2: pop();
                  break;
          case 3: display();
                  break;
          case 4: exit(0);
          default: printf("\nWrong selection!! try again!!!\n");
          }
        }
   }
   void push(int value)
   {
```

```
struct Node *newNode;
newNode = (struct Node*) malloc(sizeof (struct Node));
newNode -> data = value;
if (top == NULL)
        newNode ->next =NULL;
else
        newNode -> next = top;
top = newNode;
   printf("\n Insertion is Success!!! \n");
}
void pop()
{
   if (top==NULL)
      printf ("\n stack is Empty !!! \n");
   else{
      struct Node *temp=top;
      printf ("\n Deleted element : %d", temp -> data);
      top = temp -> next;
      free(temp);
      }
}
void display()
{
    if (top==NULL)
       printf ("\n stack is Empty !!! \n");
    else{
       struct Node *temp = top;
       while (temp -> next != NULL) {
       printf ("%d --->", temp -> data);
       temp = temp -> next;
       }
       printf ("%d ---> NULL", temp -> data);
  }
}
```

Implementation of Stacks Using Linked List :



Program :

```
#include<stdio.h>
#include<conio.h>

struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();

void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(value);
                break;
        case 2: pop(); break;
```

```
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Please try again!!!\n");
        }
    }
}
void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}
void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
void display()
```

```
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    while(temp->next != NULL){
        printf("%d--->",temp->data);
        temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
}
```

Output :

```
*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
25--->NULL

*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted element: 25
*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack is Empty!!!
```

```
*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 25
Insertion is Success!!!

*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
25--->NULL

*** MENU ***
1. Push
2. Pop
3. Display
4. Exit
Enter your choice:
```

Data Structures

Stack from infix to prefix.

Algorithm

1. Create an empty stack and an empty output string.
2. Reverse the infix expression: Reverse the order of all elements in the infix expression, including operands and operators.
3. Iterate through the reversed infix expression from left to right.

Infix: An expression is called the infix expression if the operator appears in b/w the operands in the expression.

Ex:- (A+B)*(C-D).

Prefix: An expression is called the prefix expression if the operator appears in the expression before the operands.

Ex:- i/p → a*(b+c+d)

o/p → *a+b+cd

Ex-2:- i/p → b*c

o/p → *bc

Code for convertion from infix to prefix:

```
#include<string.h>
#include<limits.h>
#include<stdio.h>
#include<stdlib.h>

#define MAX 100

// A structure to represent a stack
struct Stack
{
  int top;
  int maxSize;

  int *array;
};

struct Stack *create (int max)
{
  struct Stack *stack = (struct Stack *) malloc (sizeof (struct Stack));
  stack->maxSize = max;
  stack->top = -1;
  stack->array = (int *) malloc (stack->maxSize * sizeof (int));
  return stack;
}

// Checking with this function is stack is full or not
// Will return true is stack is full else false
//Stack is full when top is equal to the last index
int isFull (struct Stack *stack)
{
  if (stack->top == stack->maxSize - 1)
    {
      printf ("Will not be able to push maxSize reached\n");
    }
  // Since array starts from 0, and maxSize starts from 1
  return stack->top == stack->maxSize - 1;
}

// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty (struct Stack *stack)
{
  return stack->top == -1;
}

// Push function here, inserts value in stack and increments stack top by 1
void push (struct Stack *stack, char item)
{
  if (isFull (stack))
    return;
```

```
  stack->array[++stack->top] = item;
}

// Function to remove an item from stack.  It decreases top by 1
int pop (struct Stack *stack)
{
  if (isEmpty (stack))
    return INT_MIN;
  return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek (struct Stack *stack)
{
  if (isEmpty (stack))
    return INT_MIN;
  return stack->array[stack->top];
}

// A utility function to check if the given character is operand
int checkIfOperand (char ch)
{
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// Fucntion to compare precedence
// If we return larger value means higher precedence
int precedence (char ch)
{
  switch (ch)
    {
    case '+':
    case '-':
      return 1;

    case '*':
    case '/':
      return 2;

    case '^':
      return 3;
    }
  return -1;
}

// The driver function for infix to postfix conversion
int getPostfix (char *expression)
{
  int i, j;

  // Stack size should be equal to expression size for safety
  struct Stack *stack = create (strlen (expression));
```

```
 if (!stack)                                    // just checking is stack was created or not
  return -1;

 for (i = 0, j = -1; expression[i]; ++i)
  {

    if (checkIfOperand (expression[i]))
              expression[++j] = expression[i];


    else if (expression[i] == '(')
              push (stack, expression[i]);


    else if (expression[i] == ')')
              {
               while (!isEmpty (stack) && peek (stack) != '(')
                expression[++j] = pop (stack);
               if (!isEmpty (stack) && peek (stack) != '(')
                return -1;                          // invalid expression
               else
                pop (stack);
              }
    else                                  // if an opertor
              {
               while (!isEmpty (stack)
                           && precedence (expression[i]) <= precedence (peek (stack)))
                expression[++j] = pop (stack);
               push (stack, expression[i]);
              }

  }

 // Once all inital expression characters are traversed
 // adding all left elements from stack to exp
 while (!isEmpty (stack))
  expression[++j] = pop (stack);

 expression[++j] = '\0';

}

void reverse (char *exp)
{

 int size = strlen (exp);
 int j = size, i = 0;
 char temp[size];

 temp[j--] = '\0';
 while (exp[i] != '\0')
  {
```

```
    temp[j] = exp[i];
    j--;
    i++;
   }
  strcpy (exp, temp);
}

void brackets (char *exp)
{
  int i = 0;
  while (exp[i] != '\0')
   {
    if (exp[i] == '(')
              exp[i] = ')';
    else if (exp[i] == ')')
              exp[i] = '(';
    i++;
   }
}

void InfixtoPrefix (char *exp)
{

  int size = strlen (exp);

  // reverse string
  reverse (exp);
  //change brackets
  brackets (exp);
  //get postfix
  getPostfix (exp);
  // reverse string again
  reverse (exp);
}

int main ()
{
  printf ("The infix is: ");

  char expression[] = "((a/b)+c)-(d+(e*f))";
  printf ("%s\n", expression);
  InfixtoPrefix (expression);

  printf ("The prefix is: ");
  printf ("%s\n", expression);

  return 0;
}
```

Output:

```
The infix is: ((a/b)+c)-(d+(e*f))
The prefix is: -+/abc+d*ef
```