Title : Assignment 2 : Expression Conversion

Aim : To implement expression conversion using stack data structure

Problem statement : Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix and prefix expression

Objectives :

- To study data structures and their implementations.
- To learn implementation of singly linked list

Theory :

Concept of linear data structure :

- Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called linear data structure.
- In single level is involved. Therefore, we can traverse all the elements in single run only.
- Linear datastructures are easy to implement because computer memory is arranged in a linear way.

Examples :
Array, stack, queue, linked list

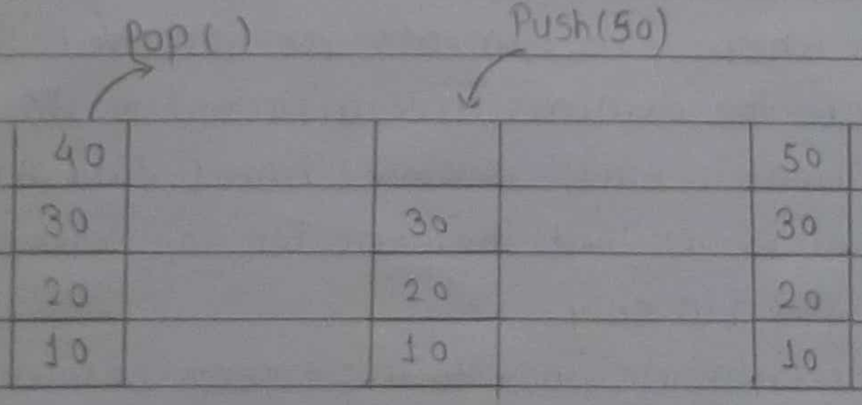
- Stack:

- A Stack is a linear data structure which follows a particular order in which the operations are performed.
- So it is restricted linear data structure as insertion & deletion is restricted to a particular end.
- It works on the principle "Last In First Out".
- Insertion & deletion are made only by ~~end~~ one end called as 'top'

Operations on stack:

i) push () - Insertion in Stack in top
ii) pop () - Deletion in Stack from top
iii) peek () - get top data element of the stack, without removing

Stack representation:



Stack as Abstract Data Type :

The stack of elements of any particular type is a finite sequence of elements of that type together with the following operations :

```
bool isEmpty ()
boll isFull()
void push ()
element pop ()
element peek ()
```

- Realization of stack using array ;

In array implementation, the stack is formed using the array. All the operations regarding the stack are performed using array.

1) Algorithm for push () operation

```
begin
    if top=n then stack full  //n is maximum size of array
    else
      top = top+1
      stack [top]= item
end
```

2) Algorithm for pop operation

```
begin
    if top=-1 then empty stack
    item = stack [top]
    top=top-1
end
```

3) Algorithm for peek operation

```
begin
    If top=-1 then empty stack
    item = stack [top]
    return item
end
```

- Example

1) Create an empty array & initialize top to -1

0	1	2	3	4

size of stack = 5

top=-1

2) push (20)
push (30)
push (50)

0	1	2	3	4
20	30	50		

↑
top=2

3) pop ()

0	1	2	3	4
20	30			

↑
top= 1

push (60)
push (100)
push (101)

0	1	2	3	4
20	30	60	100	101
				↑ top

push (200)
as top = max_size , this will be overflow condition.

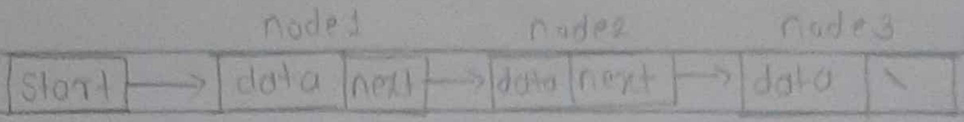
When ~~st~~ top = -1 & we call pop ( ) on stack , then it will be underflow condition.

- Realization of stack using linked list :

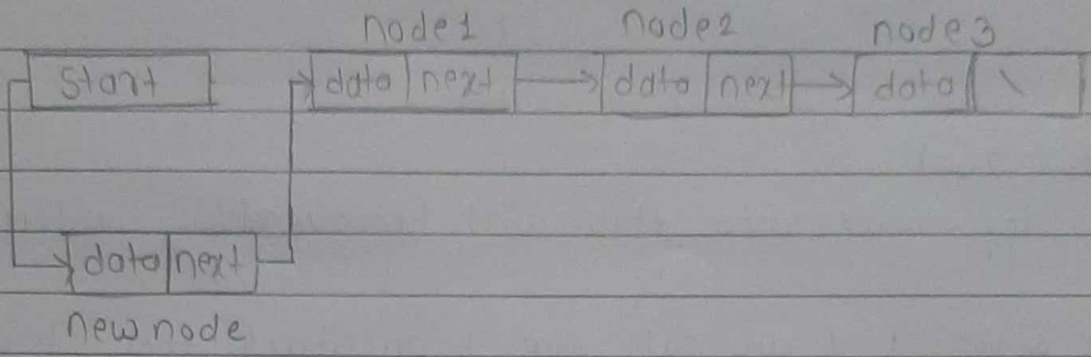
In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack.

1) Push ( ) :
Steps :
1) create a node first & allocate memory to it.
2) If the list is empty then the item is to be pushed as the start node of the list.
This includes assigning value to the data part of the node and assign null to the address part of the node.
3) If there are some nodes in the list already, then we have to add the new element in the beginning of list.

- overflow condition occurs when space left in memory heap is not enough to create a node



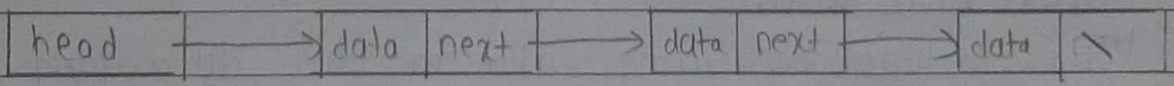
push (newNode)



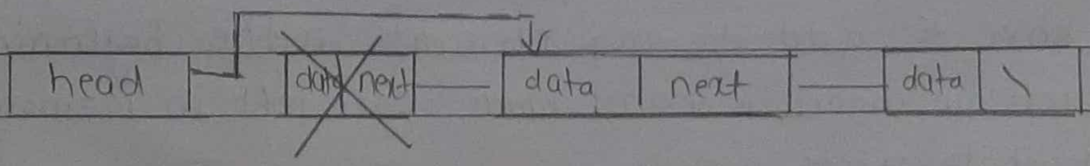
2) Algorithm for pop

1) check for underflow condition:
When head points to null, ~~underf~~ & we call pop(), then underflow condition occurs

2) Adjust head pointer accordingly:
For deletion, the value stored in head pointer is deleted & node must be deleted. The next node of head node becomes head node.



pop()



- Applications of Stack:

1) Expression evalution
2) To check parenthesis matching in an expression
3) Expression conversion
4) Memory management
5) Recursion

- Expression conversion and stack

Need for expression conversion:

- Evolution of infix expression using computer needs proper code generation by compiler without any ambiguity and is difficult due to various aspects such as operators priority and associativity.
- This problem can be overcome by converting infix to alternate notations such as prefix or postfix.

Types of polish notations

- An arithmetic expression can be written in three different but equivalent notations i.e. without changing the essence ~~of~~ output of an expression.
- These notations are:

Infix notation
Prefix (polish) notation ~~,~~
postfix (Reverse polish) notations.

1) Infix Notation:

- Operators are written in between operands.

Eg. a-b+c

2) Prefix notation:
Operator is prefixed to operand i.e. operator is written ahead of operands.
Eg. +ab

3) Postfix notation:
Operator is postfixed to operand i.e. operator is written after operands
eg. ab+

Advantages of polish notations:
- Expression can be shown without parenthesis.
- It is convinient to evaluate formula by precedence
- The complete expression can be passed in one traversal.

• Algorithm / pseudo~~de~~code:

1) Infix to postfix conversion:

```
  // Postfix = "  "
~~1)~~ //Scan infix expression from left to right.
   while infix[i] ≠ '\0'
        If infix[i] is operand
             //append it to postfix
        Else If infix[i] = '('
               push('(')
        Else If infix[i] = ')'
             while ~~pop~~ stack[top] ≠ '(' or top ≠ -1
             x = pop()
             // append x to postfix
```

```
Else If infix[i] is operator
      If top=-1 or stack[top]='('
            push(infix[i])
      Else if precedence(infix[i]) > precedence(stack[top])
            push(infix[i])
      Else
          while precedence(infix[i] <= precedence(stack[top])
                x=pop()
              //append x to postfix
          push(infix[i])
     End while
     while &top ≠ -1
         x= pop()  //append x to postfix
   Return postfix
```

- Example:
  (a+ b) * (c+d)

Symbol scan	Stack	Expression
(	(	
a	(	a
+	(+	a
b	(+	ab
)		ab+
*	*	ab+
(	*(	ab+
c	*(	ab+c
+	*(+	ab+c
d	*(+	ab+cd
)	*	ab+cd

postfix => ab+ cd + *

2) Infix to prefix

1) Reverse the infix expression. While reversing each '(' will become ')' and each ')' become '('

2) obtain postfix expression for modified expression
3) Reverse the postfix expression.

Example:
(a+b) * (c+d)
Reverse => (d+c) * (b+a)

Symbol scan	Stack	Expression
(	(	
d	(	d
+	(+	d
c	(+	dc
)		dc+
*	*	dc+
(	*(	dc+
b	*(	dc+b
+	*(+	dc+b
a	*(+	dc+ba
)	*	dc+ba+

=> dc+ba+*
Prefix => *+ab+cd

3) Postfix evaluation

```
Postfix _ evalution (postfix [ ])
    while ( postfix [i] ≠ '\0')
        symbol = postfix (i]
        If Symbol isoperand
            push (symbol)
        Else
            op2 = pop ( )
            op21= pop ( )
            result = op21 symbol op2
            push (result)
    End while
    Return pop( )
```

Example :

10 , 2 , + , 12 , 4 , + , *

Symbol scan	stack	operation
10	10	
2	$\overset{op1}{10}$ 2^{op2}	
+		10 + 2 = 12
	12	
12	12 12	
4	12 12 4	
+		12+4 = 16
	12 16	
*		12 * 16 = 19 2
	192	

∴ Result = 192

4) Prefix evalution:

```
Prefix evaluation (prefix [])
  // Scan from right to left
  ~~while (i ≠ 0~~ i= prefix. length()-1
  while (i>=0)
    If prefix [i] is operand
        push (prefix [i])
    Else
        symbol = prefix [i]
        op1 = pop()
        op2 = pop()
        result = op1 symbol op2
        push (result)
  End while
  Return pop
```

Example:

*, +, 10, 2, +, 12, 4

Symbol	Stack	operation
4	4	
12	4 12	
+		12+4=16
	16	
2	16 2	
10	16 2 10	
+		10+2=12
	16 12	
*		16*12 = 192

192

- Test cases /validations:

Validations:

1) Number of operands and operators relationship
2) Well formed parenthesis matching

Test cases:

Based on precedence of operators

Sr. No.	Infix	Postfix	Prefix
1.	A+B*C	ABC*+	+A*CB
2.	A*B-C	AB*C-	-*ACB
3.	A^B-C	AB^C-	-^ABC
4.	A+B*C^E	ABCE^*+	+A*B^CE
5.	A-B*C+A	ABC*-A+	-+A*BCA

- Conclusion

Prefix and postfix expressions can be evaluated faster than infix expression. We can convert infix to prefix or postfix using stack. Evaluation of expression can also be done by stack.