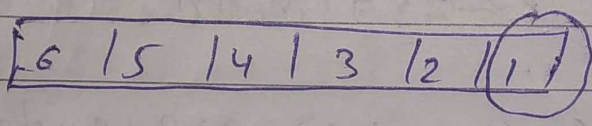
:- Introduction to Stack :-

- A stack is a list of elements in which insertions and deletions can take place only at one end. This end is called top of stack top.
- Only top element can be accessed.
- It has LIFO property
- A stack is an example of a linear data structure.



- one side is ~~cotes~~ closed, another side is open.

⇒ Common operations of stack :-

- push() → insert new element into stack
- pop() → deletes the top element.
- peek() → gives the element at top
- isFull → checks whether stack is full
- isEmpty → checks if stack is empty or not.

⇒ Stack Implementation :-

(i) Implemented using an array.
2. Implemented using a linked list :-
The most important advantage is that the size of the stack is not needed to be specified statically. Push and pop operations are just insertion and deletion in the front of the linked list.

⇒ peek() function in C programming:-

```
int peek(){
  return stack[top];
}
```

⇒ isfull() function in C programming:-

```
bool isfull() {
  if ( top == MAXSIZE -1)
   return True;
else
    return false;
 }
```

⇒ isEmpty() function in C programming:-

```
bool isEmpty () {
   if (top == -1)
      return true;
  else
    return false;
}
```

⇒ Implementation of Push:-

> Check the status of stack, if it full or not.
> If it is stack is full returns an error and exit from the function.
> If the stack is not full, increments top to point next empty slot.
> Adds data element to the stack location, where is top is pointing.

```
void push(){
    int x;
    if(top == n-1){
      printf("\n Stack is overflow");
      exit(1);
    }
    else{
      printf("Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      Stack[top]=x;
    }
}
```

⇒ Implementation of pop():-

Algorithm:-

> Checks if the stack is empty or not.
> If the stack is empty, it produces an error and exit.
> If the stack is not empty, delete the data element at which top is pointing.
> Decrement the value of top by 1.
> Return the deleted element.

Code:-

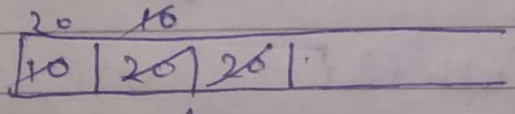
```
Void pop(){
  if(top==-1){
     printf("\n Stack is Underflow");
     exit(1);
  }
```

```
    else {
        printf("\n Popped elements is %d", stack[top]);
        top--;
    }
}
```

⇒ Code to print the element of the stack:-

```
void display(){
    if (top >= 0){
        printf("\n The elements of the stack are \n");
        for(i=top; i>=0; i--)
            printf("\n%d", stack[i]);
    }
    else {
        printf("\n Stack is empty");
    }
}
```

Question: The following operations is performed on a stack: PUSH(10), PUSH(20), POP, PUSH(10), PUSH(20), POP, POP, POP, PUSH(20), POP. The sequence of value popped out:



Ans: 20, 20, 10, 10, 20 Ans

⇒ Multiple Stack Implementation:-

Size of an array (M) = 9
No. of Stack (N) = 3

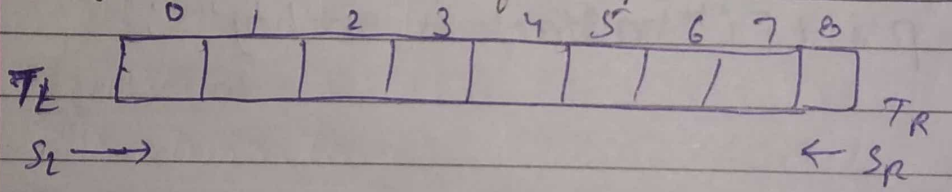
Empty condition T_i for the stack:

$$T_i = (M/N) * i - 1$$

Full condition

$$T_i = (i+1)\left(\frac{M}{N}\right) - 1$$

⇒ Efficient way of implementing stack:-



$$\begin{cases} T_L = T_R - 1 \\ T_R = T_L + 1 \end{cases} \rightarrow \text{full condition}$$

Implementation of Push:-

```
Void Push () {
  int x;
  int (ti == (i+1)(M/N)-1) {
        printf ("\n Stack is overflow");
        exit (1);
  }
  else{
    printf ("Enter a value to be pushed:");
```

```
        scanf("%d", &x);
        ti++;
        stack[ti] = x;
    }
}
```

⇒ Implementation of pop(-):-

```
void pop() {
    if (Ti == -1(M/N)-1) {
        printf("\n Stack is underflow");
        exit(1);
    }
    else {
        printf("\n Popped elements is %d", stack[Ti]);
        Ti--;
    }
}
```

⇒ Applications of Stack:-

> Used in "Undo" mechanism at many places.
> Compiler's syntax check for matching braces.
> Support for recursion
> String reversal.

⇒ Applications of stack:-

> Recursion
> Infix to postfix conversion
> Prefix to postfix conversion
> Infix to prefix conversion
> Postfix evaluation
> Tower of HANOI
> Fibonacci series.

: Recursion :-

> Recursion functions typically implement recurrence relations, which are mathematical formula.
> The function written in terms of itself is a recursive case.
> Recursion is a repetitive process in which a function calls itself.
> A function is called recursive if a statement within the body of a function calls the same function.
> Also known as circular definition.

Tail Recursion :-

> If in the given recursive program, recursive call is at the end of the program, that recursion is also known as tail recursion.
> Disadvantage :- Unnecessarily wasting disk space.
> Advantage :- can be easily convert to an equivalent non-recursive program.

Ex:

```
Tail (int i) {
    if (i<=1)
        return;
    else {
        1. printf("%d", i);
        2. Tail (i-1);
    }
}
```

Output → 5 4 3 2

Tail (5)
1. 5
2. Tail (4)
1. 4
2. Tail (3)
1. 3
2. Tail (2)
1. 2
2. Tail (1)
↳ Terminate

Non Tail Recursion:-

> If in the given program after the recursive call some statements are there then it is known as non-tail recursion.

> Disadvantage :- Difficult to write an equivalent non-recursive program.

> Advantage :- Space is utilized efficiently.

Ex:-

```
Non Tail (int i) {
    if (i<=1){
        return;
    } else {
        Non Tail (i-1);
        printf ("%d", i);
        Non Tail (i-1);
    }
}
```

Out :- 2, 3, 2, 4, 2, 3, 2

Nested Recursion:- A recursion is nested if a recursive function includes one of the argument to another recursive function.

Eg:- Ackermann's Recurrence Relation / Ackerma$_{nn}$ function:-

$$A(m,n) = \begin{cases} n+1 & \text{if } m == 0 \\ A(m-1,1) & \text{if } m == 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

Calculate

(1) A (1, 3) = 5

(2) A (2, 2) = 7

:- Indirect Recursion :-

```
Aorder (int i){
  if (i<=1){
    return i
  }
  else {
    Border (i-1);
    printf ("%d", i);
    Border (i-1);
  }
}
```

```
Border (int i){
  if (i<=1){
    return;
  } else {
    printf ("%d", i);
    Aorder (i-1);
    printf ("%d", i);
  }
}
```

» An expression can be represented in :-

> Prefix notation (Polish notation) :-
operand1 operator operand2 operand2

> Infix notation :-
operand1 operator operand2

> Postfix notation :- (Reverse Polish notation)
operand1 operand2 operator

Symbol	Priority	Associativity
(), [], {}	1 (Highest)	~~R-L~~
^(exp)	2	R-L
*, /	3	L-R
+, -	4 (lowest)	L-R

⇒ **Infix to Postfix Conversion:**

A + B → AB+

A + B * C → ABC*+

A + B - C → AB+C-

~~AB~~ A^B^C → ABC^^

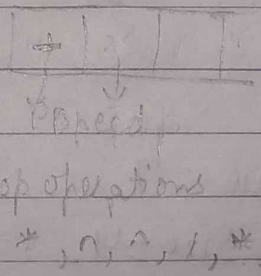
Note :- While conversion only order of operations is going to change not the order of operand.

⇒ **Infix to Postfix Conversion rules using Stack:-**

Top of Stack	Next Coming operator	Stak operation
Lower Priority	Higher	push ()
Higher	Lower	pop ()
Same Priority	L-R (Asso.)	pop ()
	R-L (Asso.)	push ()

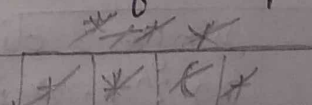
Ex:- Infix → A + B * C / D ^ E ^ F * D - C + B

AB C * D E F ^^ / D * + C - B +



GATE-1995 The postfix expression for the infix expression A + B * (C + D) / F + D * E is :

Ans AB C D + * F / + D E * +



Note:- An operator stack is essential for converting an infix expression to the postfix form efficiently.

⇒ Evaluation of Postfix expression:-

Step-1 Read the postfix expression from left to right character by character

Step-2 If character = operant then push in stack.

Step-3 If the character is an operator pop two element from the stack op2 = pop() // top element
op1 = pop() // Next element.

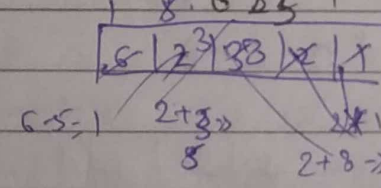
Step-4 move the pointer towards left and go to ~~set~~ step-2 as long as there are characters left to be scanned in the expression.

Step-5 The result is stored at the top of the stack, return it.

Q. What is the maximum size of operand stack while evaluating the postfix expression?

6 5 2 3 + 8 * + 3 8 2 / + *

Sol =



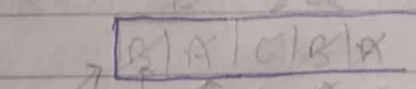
Value = 7

Stack Size = 4.

Postfix to Postfix Conversion :

Step-1 Read the prefix expression from right to left
Step-2 If char is operand push it into the stack.
Step-3 If char is an operator pop two elements from the stack op1 = pop() // top element
op2 = pop() // next element to top. Display it as op1 and op2 and push it back to the stack.

Q. Prefix : * - / ↑ ABCD - ↑ / * ABCD + AB
Read the prefix expression Right to left



AB↑CD/-AB*D↑AB+-* Ans.

Summary : Infix to postfix conversion (L-R)

→ Operator stack
operand → display it
operand → follow rules

TOS	Next Coming	Operator
Lower	Higher	push
Higher	Lower	pop
Same	L-R	pop
Priority	R-L	push

Prefix to Postfix Conversion: (R-L)

operand → push it in stack

operator → op1 = pop()
op2 = pop()
push(op1 op2 operator).

Postfix evaluation: (L-R)

→ operand stack

operand → push it in the stack

operator → op2 = pop() } Need to get two
op1 = pop() } operands.

Evaluate → op1 operator op2 and push() back to the stack.

⇒ Tower of HANOI:

Assumptions:

① At a time, only one disc can be moved
② one a larger disc, smaller disc can be placed but vice-versa is not possible.
③ All the disc should be present in 3 towers only.

Goal: To move all the disc from one tower (source) to other tower (destination).

n-discs → (n+1) func. calls

No. of moves → $2^n - 1$ with n discs

No. of functions calls with n discs = $2^{n+1} - 1$

Complexity = $2T(n-1) + 1$

Infix to Prefix Conversion Rules:-

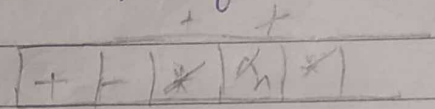
Top of Stack	Next Coming operator	Stack operation
Lower	Higher	push()
Higher	Lower	pop()
Same priority	L-R Assoc.	push()
	R-L Assoc.	pop()

Ex:- A+B*C/D^E^F*D-C+B

Step-1 Reverse the ~~string~~ given infix expression

B+C-D*F^E^D/C*B+A

Step-2 Find the ~~postfix~~ Prefix expression for string obtained:



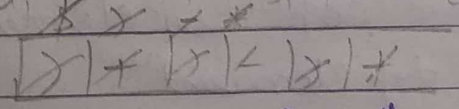
BCDFE^D^CB*/*A+-+

Step-3 Reverse the string obtained from step-2

+-+A*/*BC^D^EFDCB

2. ((A+B)*C-(D-E))$(F+G)

)G+F($))E-D(-C*)B+A((



GF+ED-CBA+*-$

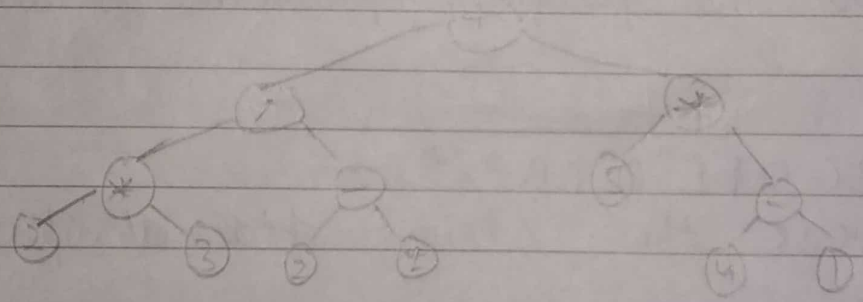
Step-3 $-*+ABC-DE+FG

⇒ Binary Expression Tree:- An expression tree is a strictly binary tree in which leaf nodes contains operands and non-leaf

nodes contain operator that is applied to result of left subtree and right subtree. Once we obtain the expression tree for a particular expression its conversion into different notations and evaluation become a matter of traversing the exp. tree.

leaf node → operand
internal node → operator

Ex: Construct expression tree for the given infix expression:- 2*3/(2-1)+5*(4-1)



⇒ Algorithm for Prefix to Postfix Conversion:-

Step 1. Read prefix expression right to left char by char.
Step-2. If char == operand push it into stack
Step-3 If char == operator pop two element from stack
op1 = pop() // top element
op2 = pop() // Next to top

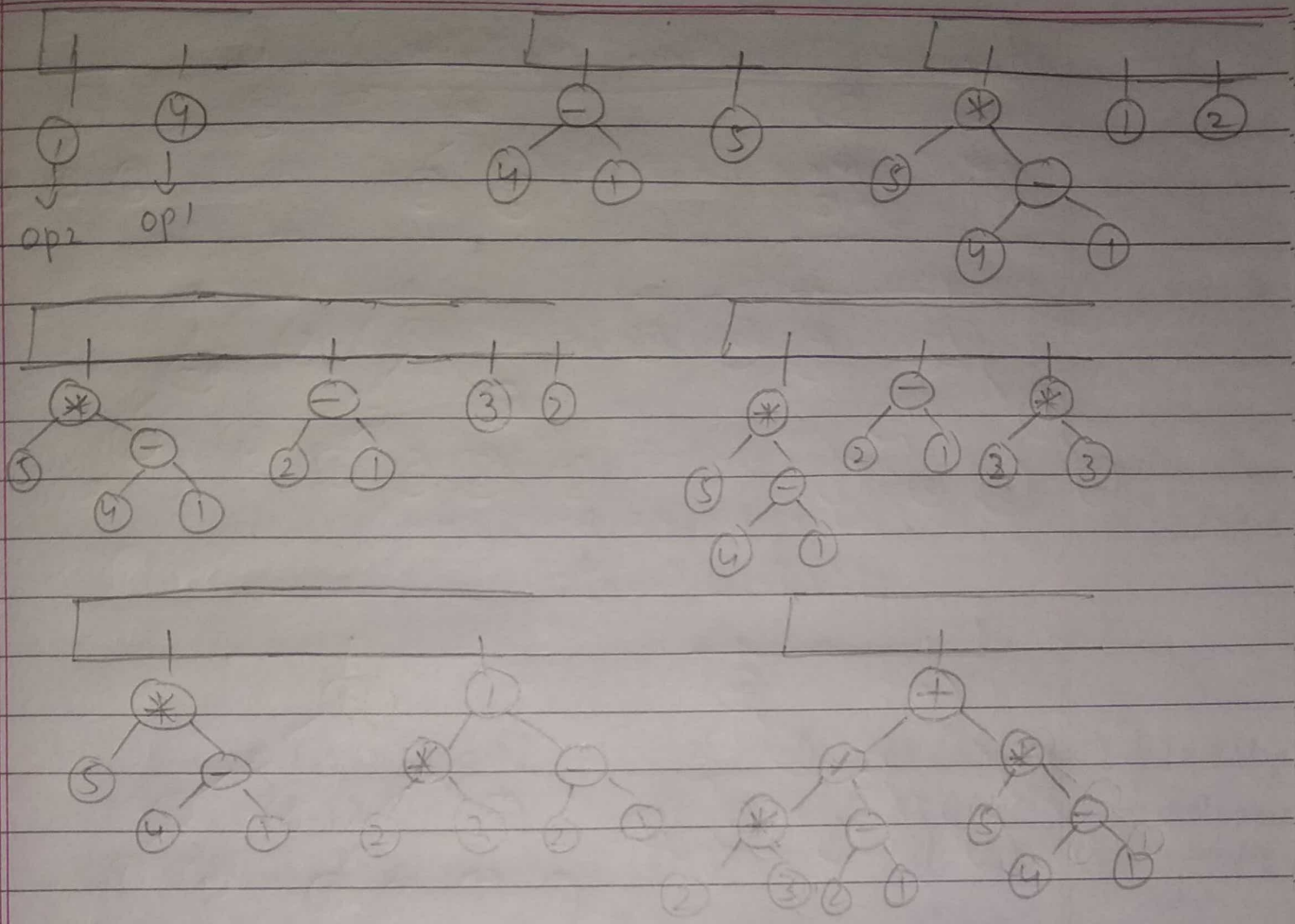
Q. Construct the expression tree for the given prefix expression:- +/*23-21*5-41

Sol 2 Revel the given prefix expression:-

14-5*12-32*/+

op1 = POP() // ~~Next~~ top element, node → left_child

op2 = pop() // Next to top, node → right_child



⇒ Algorithm for evaluation of postfix expression:-

Step-1 Read postfix expression from left to right char by char

Step-2 If char == operand push it into the stack

Step-3 If char == operator pop two elements from stack

op2 = pop() // top element

op1 = pop() // Next to top

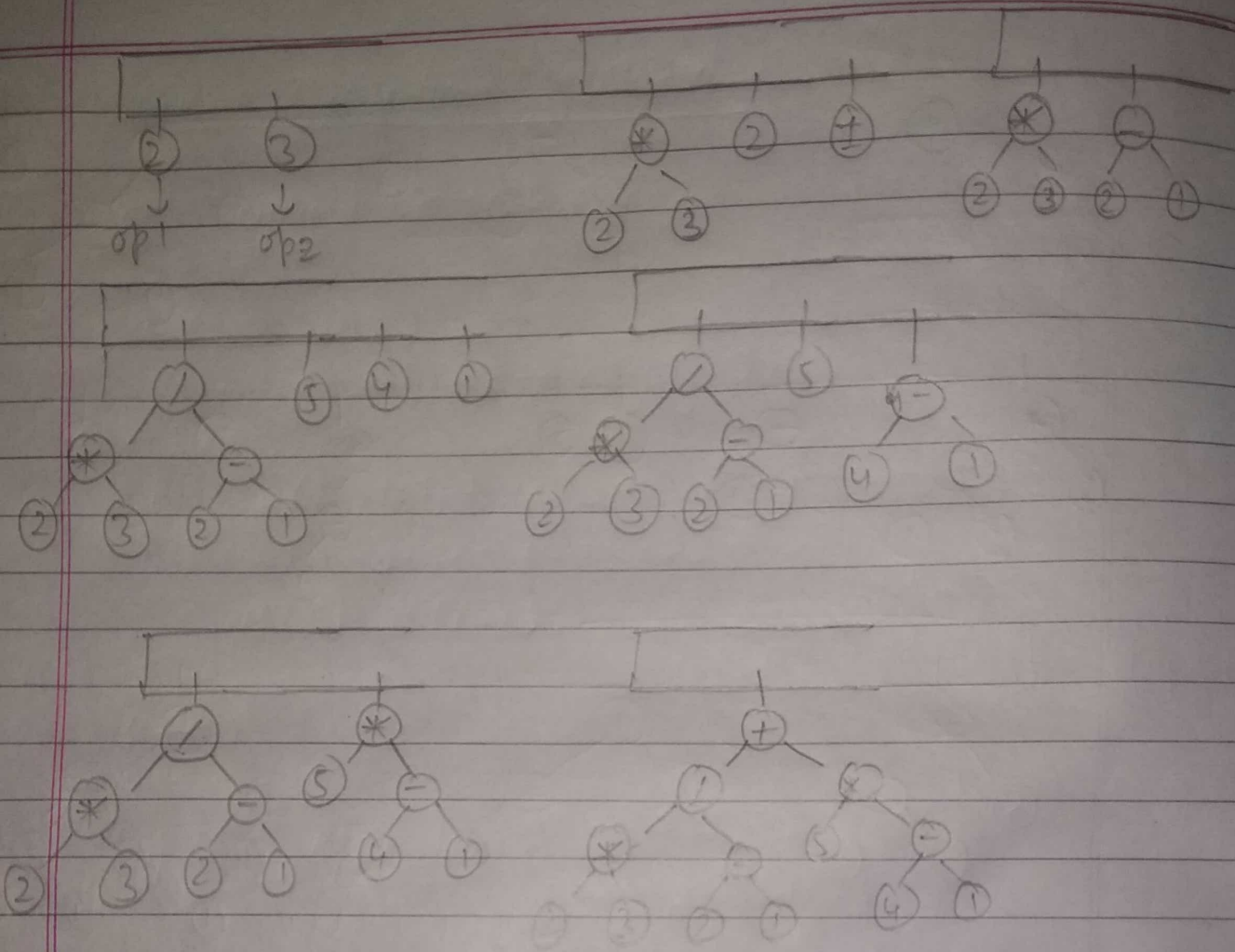
op2 = node → right_child

op1 = node → left_child

Q. Construct the expression tree for the given postfix expression : 23* 21- 1541- *+

Sol Read the postfix expression left to right

23* 21- 1541- *+



⇒ Fibonacci Series:-

$$fib(n) = \begin{cases} 0 & \text{if } n==0 \\ 1 & \text{if } n==1 \\ fib(n-1)+f(n-2) & \text{otherwise if } n \geq 2 \end{cases}$$

} termination condition

n → after (n+1) function calls 1st addition will take place.

No. of additions in fib(n) = f(n+1) − 1
Value