Digital Assignment - I

Problem solving with Data Structures and Algorithms

1) Algorithm for infix to postfix.

Ans)

Step1: Scan the infix expression from left to right.

Step2: If the scanned character is an opperand, append it with the Infix to Postfix string.

Step 3: If the scanned character is not an opperand,

Step 3.1: If the precedence order of the scanned operator is greater than the precedence order of the operator in the stack or the stack is empty or it contains a '(' or '[' or '{'), push it on stack.

Step 3.2: Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that, push the scanned operator to the stack. (If ~~p~~ paranthesis is encountered while popping then stop there and push the scanned operator in the stack.

Step 4: If the scanned character is an ')' or ']' or '}', then pop the stack until a '(' or '[' or '{' respectively is encountered, and discard both the paranthesis.

step5: Repeat step 2 to step 5 until infix expression is scanned.

step 7: Give the output string.

Algorithm for infix to prefix

infix = reverse(infix)
loop i = 0 to infix.length
if infix[i] is operand → prefix += infix[i]
else if infix[i] is '(' → stack.push(infix[i])
else if infix[i] is ')' → pop and print values of stack till the symbol ')' is not found.
else if infix[i] is an operator →

if the stack is empty then push infix[i] on the top of the stack

Else →

If (precedence(infix[i]) > precedence(stack.top))
→ Push infix[i] on the top of the stack
else if (infix[i] == precedence(stack.top) && infix[i] == '^')
→ pop and print the top values of the stack till the condition ~~true~~ is true
→ Push infix[i] into the stack
else if (infix[i] == precedence(stack.top))
→ push infix[i] on ~~to~~ to the stack
Else
→ Pop the stack values and print them till the stack is not empty and infix[i] < precedence(stack.top)
→ push infix[i] on ~~t~~ to the stack

END LOOP
Pop and print remaining elements of stack
Prefix = revers(prefix)

1) a) A^B*C - D + E/F /(G + H)

Infix to Post fix

Scan	Stack	Output
A		A
^	^	A
B	^	AB
*	*	AB^
C	*	AB^C
-	-	AB^C*
D	-	AB^C*D
+	+	AB^C*D-
E	+	AB^C*D-E
/	+/	AB^C*D-E
F	+/	AB^C*D-EF
/	+/	AB^C*D-EF/
(	+/(	AB^C*D-EF/
G	+/(	AB^C*D-EF/G
+	+/(+	AB^C*D-EF/G
H	+/(+	AB^C*D-EF/GH
)	+/	AB^C*D-EF/GH+
Empty	Empty	AB^C*D-EF/GH+/+

Postfix = AB^C*D-EF/GH+/+

1) b) Infix to Postfix

$A - B / (C * D ^ E)$

Scan	Stack	Output
A		A
-	-	A
B	-	AB
/	-/	AB
(	-/(	AB
C	-/(	ABC
*	-/(*	ABC
D	-/(*	ABCD
^	-/(*^	ABCD
E	-/(*^	ABCDE
)	-/	ABCDE^*
Empty	Empty	ABCDE^*/-

Postfix = ABCDE^*/-

1) a) $A ^ B * C - D + E / F / (G + H)$

Infix to Prefix

After reversing :) H + G (/ F / E + D - C * B ^ A

Scan	Stack	Output
)	)	-
H	)	H
+	) +	H
G	) +	H G
(	+	H G
/	+ /	H G
F	+ /	H G F
/	+ /	H G F /
E	+ /	H G F / E
+	+	H G F / E / +
D	+	H G F / E / + D
-	-	H G F / E / + D +
C	-	H G F / E / + D + C
*	- *	H G F / E / + D + C
B	- *	H G F / E / + D + C B
^	- * ^	H G F / E / + D + C B
A	- * ^	H G F / E / + D + C B A
Empty	Empty	H G F / E / + D + C B A ^ * -

~~Prefix: H G F / E / + D + C B A ^ * -~~

Prefix: - * ^ A B C + D + / E / F G H

1) b) Infix to Prefix

A - B / (C * D ^ E)

After reversing :) E ^ D * C (/ B - A

Scan	Stack	Output
)	)	
E	)	E
^	) ^	E
D	) ^	E D
*	) *	E D ^
C	) *	E D ^ C
(		E D ^ C *
/	/	E D ^ C *
B	/	E D ^ C * B
-	-	E D ^ C * B /
A	-	E D ^ C * B / A
Empty	Empty	E D ^ C * B / A -

Prefix = - A / B * C ^ D E

2) Algorithm to add 2 polynomials represented as circular list with header node.

Step1: Create 2 circular linked list with the following attributes each:

(i) coefficient of x
(ii) coefficiet of y
(iii) power of x
(iv) power of y
(v) pointer to the next node.

Step2: Traverse both polynomials. (Loop start)

If power of x of first polynomial is greater than that of second polynomial
→ store node of first Polynomial in result and increase iterator of first polynomial.

Else If power of x of first polynomial is less than that of second polynomial
→ store node of second polynomial in result and increase the iterator of second polynomiall

Else (both equal)
→ IF power of y of 1st polynomial is greater than that of 2nd polynomial
→ store the node of first polynomial in result and increase iterator of polynomial 1

Else
→ store the ~~#~~ sum of coefficient of both polynomial in result and increase iterator of both polynomials.

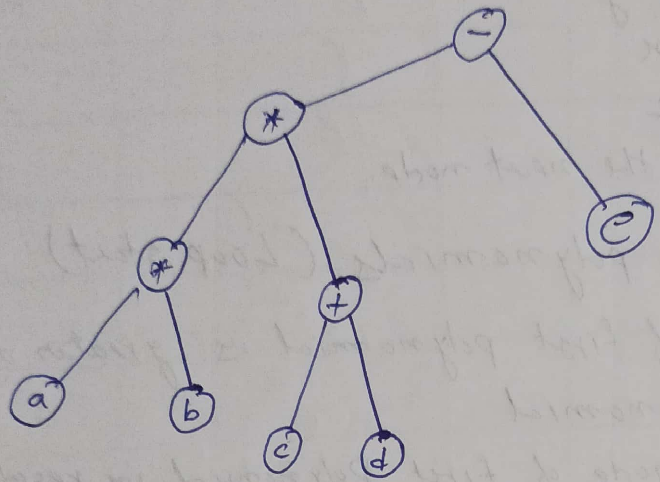
Loop END

Step3: Append the remaining parts of the longer polynomial in the result.

Step4: Give output

---

3)



Prefix : - * * a b + c d e

Infix : a * b * c + d - e

Postfix : a b * c d + * e -

---

4)

```
#include<iostream>
using namespace std;

~~int p~~
int stack[10], top = -1;
void push(int a)
{
    stack[++top] = a;
}
```

```
void pop()
{
    while(top!=-1)
    {   cout<<stack[top--];
    }
}
int main()
{
    cout<<"Enter a decimal number: ";
    int n;
    cin>>n;
    while(n>0)
    {
        push(n%2);
        n/=2;
    }
    pop();
    return 0;
}
```

5) (i) Counting number of bits in binary representation of a number.

(ii) basic operation is to divide the value of n by 2 till it is greater than 1, i.e, the loop stops when n decreases to 1 or less than that (theoretically).

(iii) basic operation is executed $(m-1)$ times where 'm' is the number of digits in the binary representation.

(iv) Efficiency class → logarithmic [$O(\log n)$]

(v) This algorithm has time complexity of $O(\log n)$. A more efficient algorithm for this will be an algorithm which runs in constant time ($O(1)$). For this particular instance, we need to apply a loop to find the number of bits, the loop runs for ~~are less times~~ $(m-1)$ times where m is the number of bits. This case makes it impossible to solve in constant time as we do not know the value of 'm' for the input n, i.e., 'm' is a variable, this means the loop runs variable times with respect to the input n.

Therefore it is impossible to make it an $O(1)$ algorithm.