Stack

Important Question

1. Define Data Structure. Differentiate linear and non-linear data structures with example [3M] (IT).
2. What is recursion ? state its advantage. * (IT) 3M
3. Explain Linear and Non linear data structure *** (CO/IT) [5M]
4. Define stack. Give its application (IT)(3M)
5. Write algorithm to convert infix expression to postfix **(IT)(10 M)
6. Write a postfix form of following infix expression (IT) (3M)
   A+(B*(C-D)/E)
   (f-g)*((a+b)*(c-d))/e
7. Write algorithm to implement stack using array** [10 M]
8. Write a program to evaluate postfix expression using stack** [10 M].
9. Write a program to convert infix expression to postfix expression * [10 M].
10. Write an algorithm for postfix evaluation. Demonstrate step by step.
    9 6 7 * 2 / -
11. Use of stack data structure to check Well-formedness of parentheses in an algebraic expression.

What is recursion ? state its advantage. * (IT) 3M

Advantage

1. The code may be easier to write.
2. To solve such problems which are naturally recursive such as tower of Hanoi.
3. Reduce unnecessary calling of function.
4. Extremely useful when applying the same solution.
5. Recursion reduce the length of code.
6. It is very useful in solving the data structure problem.
7. 7. Stacks evolutions and infix, prefix, postfix evaluations etc.
8. Reduces system resource usage and performs better than the traditional for loop.

Disadvantage

1. Recursive functions are generally slower than non-recursive function.
2. It may require a lot of memory space to hold intermediate results on the system stacks.
3. Hard to analyze or understand the code.
4. It is not more efficient in terms of space and time complexity.
5. The computer may run out of memory if the recursive calls are not properly checked.

Applications of Stack [5M]

- Expression Conversion
  - a) Infix to postfix
  - b) Infix to prefix
  - c) postfix to infix
  - d) prefix to infix
- Expression Evaluation
- Parsing
- Simulation of recursion
- Function Call

1. ***Algorithm for PUSH() operation in Stack using Array:***

```
Step 1: Start

Step 2: Declare  Stack[MAX];    //Maximum size of Stack

Step 3: Check if the stack is full or not by comparing top with (MAX-1)
        If the stack is full, Then print "Stack Overflow" i.e, stack is full
and cannot be pushed with another element

Step 4: Else, the stack is not full
        Increment top by 1 and Set,
        a[top] = x
        which pushes the element x into the address pointed by top.
        // The element x is stored in a[top]
Step 5: Stop
```

1. ***Algorithm for POP() operation in Stack using Array:***

```
Step 1: Start

Step 2: Declare  Stack[MAX]

Step 3: Push the elements into the stack

Step 4: Check if the stack is empty or not by comparing top with base of
array i.e 0
        If top is less than 0, then stack is empty, print "Stack
Underflow"

Step 5: Else, If top is greater than zero the stack is not empty, then store
the va
```

1. We first define a stack of max size
2. **PUSH():** First, we check if the stack is full, if the top pointing is equal to (MAX-1), it means that stack is full and no more elements can be inserted we print overflow. Otherwise, we increment the top variable and store the value to the index pointed by the top variable
3. **POP():** First, we check if the stack is empty, if the top variable has a value less than 0, it means the stack is empty. So, we can't delete any more elements and print underflow condition. Otherwise, we decrement the top variable.
4. **PEEK():** The PEEK() function returns the topmost value stored in stack, it's done by returning the element pointed by the top variable.

Algorithm to convert an Infix expression to a Postfix expression. Check below example.

Step 1 : Start
Step 2: Input the infix expression. i.e Store each element i.e (operator / operand / parentheses) of an infix expression into a list.
Step 3. If the token is an operand, append it to the postfix expression. (Positions of the operands do not change in the postfix expression so append an operand as it is.)
Step 4. If the token equals "**(**", push it onto the top of the stack.
Step 5. If the token equals "**)**", pop out all the operators from the stack and append them to the postfix expression till an opening bracket i.e "**(**" is found.
Step 6. If the token equals "*****" or "/" or "**+**" or "**-**" or "^", pop out operators with higher precedence at the top of the stack and append them to the postfix expression. Push current token onto the stack.
Step 7. If the token is an operand, append it to the postfix expression. (Positions of the operands do not change in the postfix expression so append an operand as it is.)
Step 8. Repeat step 3 to step 7 till input in null.
Step 9. pop out all the operators from the stack and append them to the postfix expression if token empty.
Step 10.Stop

Algorithm : Evaluating the postfix expression

Step 1. For each element (operator or operand) of tokenized postfix expression stored in the list/queue do
Step 2 and Step 3.
Step 2. If the token is an operand i.e a number between (0 .. 9), push it into the stack.
Step 3. Else i.e if the token is an **operator** (+, -, ^, *, /), pop the top two elements from the stack
x = element at top, and y = element below the top element.
push result of operation (y **operator** x) into the stack.
Step 4. Element at the top of the stack is the result of the evaluation.