Homework Two : Stacks

Question One

> a) Use the operations push, pop, peek and empty to construct an operation which sets i to the bottom element of the stack, leaving the stack unchanged. (Hint: use an auxiliary stack.)

```
stack a = {initial_values}
stack b = {}
while stack a.peek != null {
    b.push(a.pop)
}
i = b.peek()
while stack b.peek() != null {
    a.push(b.pop)
}
```

> b) Use the operations push, pop, peek and empty to construct an operation which sets i to the third element from the bottom of the stack. The stack may be left changed

stack a = {initial_values} stack b = {} while stack a.peek != null { b.push(a.pop) } repeat(3): a.push(b.pop) i = a.peek()

Question Two:

> ```
> 2. Simulate the action of the algorithm for checking delimiters for
> each of these strings by using a stack and showing the contents of the
> stack at each point. Do not write an algorithm.
> ```

> a. {[A+B]-[(C-D)]

```
0. push { to stack: stsack = {
1. push [ to stack: stack = {[
2. ignore A
2.5 ignore +
3. ignore B
4. pop [ bc of match : stack {
5. ignore -
6. push [ to stack: stack {[
7. push ( to stack: stack {[(
8. ignore C
8.5 ignore -
9. ignore D
```

```
10. pop ( because of ): stack {[
11. pop [ because of ]: stack {
```

> b. ((H) * {([J+K])})

```
0. push (:  stack (
1. push (:  stack ((
2. ignore H
3. pop ( bc ): stack (
4. ignore *
5. push {: stack ({
6. push (: stack ({(
7. push [: stack ({([
8. ignore J
9. ignore +
10. ignore K
11. pop [ bc ]: stack ({(
12. pop ( bc ): stack ({
13. pop { bc }: stack (
14. pop ( bc )L stack:
```

Question Three

> Write an algorithm to determine whether an input character string is of the form x C y where x is a string consisting only of the letters 'A' and 'B' and y is the reverse of the x (i.e. if x="ABABBA" then y must equal "ABBABA"). At each point you may read only the next character in the string, i.e. you must process the string > on a left to right basis. You may not use string functions

```
i = 0
stack = []
while char[i] != C {
    if char[i] != "A" or char[i] != "B"
        return False

    else {
        stack.push(char[i])
        i += 1
    }
}
// This will leave us with a stack containing x
i += 1 // index to hypothetical start of y
while char[i] != null { \\ assuming null term string
    if stack.pop() != char[i]:
        return False
}

if stack.size
    return False
```

```
else
    return True
```

Question Four

> Write an algorithm to determine whether an input character string is of the form a D b D c D … D z where each string a, b, …z is of the form of the string defined in problem 3. (Thus a string is in the proper form if it consists of any number of such strings from problem 3, separated by the character 'D', e.g. > ABBCBBADACADBABCBABDAABACABAA.) At each point you may read only the next character in the string, i.e. you must process the string on a left to right basis. You may not use string functions.

the basic approach is to just run the algo but terminate on a char == D, and then just loop that until we hit a null terminator. This could also just be done recursively if we were allowed to shove the string onto a substring and then pass it to the original function

```

i = 0
stack = []
while char[i] != null {
  while char[i] != C {
      if char[i] != "A" or char[i] != "B"
          return False

      else {
          stack.push(char[i])
          i += 1
      }
  }
  // This will leave us with a stack containing x
  i += 1 // index to hypothetical start of y
  while char[i] != D { \\ assuming null term string
      if stack.pop() != char[i]:
          return False
  }

  if stack.size
      return False
  i += 1; // Get us over the D
}
```

Question Five

> Consider a language that does not have arrays but does have stacks defined as a data type. That is, one can declare stack s; The push, pop, empty, and peek operations are defined. Show how a one-dimensional array can be implemented by using these operations on two stacks. In particular, show how you can insert into and read from such an array

Given the language with a stack type implemented, we can implement an array by stacking all elements of the array on the first stack, and then using the second stack as a holding ground for the chaff while we're working on the other:

```
stack a = {initial values, with the top of the stack being the 0 index}
stack b = {empty}
array = new(a, b) // assume array is build by passing location to two
stacks
// Read Implementation - return value at index i if it exists, else, return
NULL
array.read(i);

function stack_val read(idx):
    stack_val return_val;
    for i = 0; i <idx-1; i++ {
        if !a.empty()
            b.push(a.pop)
        else
            while !b.empty {a.push(b.pop)}
            return NULL
    }
    return_val = a.peek()
    while !b.empty {a.push(b.pop)}

function insert(idx, stack_val val):
    for i = 0; i < idx-1; i++ {
        if !a.empty()
            b.push(a.pop)
        else
            while !b.empty { a.push(b.pop)}
            return False // failed to insert
    }

    a.pop() // discard top
    a.push(val) // push new val
    while !b.empty{ a.push(b.pop)}

    return True
```

Question Six

> Design a method for keeping two stacks within a single linear array s[SPACESIZE] so that neither stack overflows until all of memory is used and an entire stack is never shifted to a different location within the array. Write methods push1, > push2, pop1, and pop2 to manipulate the two stacks. (Hint: the two stacks grow toward each other.)

To do this, we'll need a top pointer of each stack as an index of the array. This has a weird behavior where we're initializing things such that the size of each stack is always one, technically, with the indication that its empty being that the value is set to null.

We could also use values outside of SPACESIZE in order to fully allow the stack to operate, but I don't know the parameters of the system so I chose not to .

```
// Initialize array
array s[SPACESIZE]; // assume initialized to NULL
// Initialize pointers
stack_one_top = 0;
stack_two_top = SPACESIZE-1; // Assuming 0 indexing

function push1(stack_val val)
    if(stack_one_top == stack_two_top):
        return False // unable to push because we're full memory
    else if(stack_one_top == SPACESIZE-1)
        return False
    else
        s[stack_one_top] = val;
        stack_one_top++;
        return True

function push2(stack_val val)
    if(stack_two_top == stack_one_top):
        return False
    else if(stack_two_top == 0)
        return False
    else
        s[stack_two_top] = val;
        stack_top_top--;
        return True

function pop1()
    if(s[stack_one_top] != NULL)
        return s[stack_one_top]
        s[stack_one_top] = NULL
        if(stack_one_top) stack_one_top--;
    else
        return NULL

function pop2()
    if(s[stack_two_top] !+ NULL)
        return s[stack_two_top]
        s[stack_two_top] = NULL
        if(stack_two_top != SPACESIZE -1) stack_two_top++;
    else
        return NULL
```

Question Seven

> Transform each of the following expressions to prefix and postfix expressions. $ is exponentiation. a. (A+B)*(C$(D-E)+F)-G

```
prefix:
+ A B
-*+AB-+$C - DEFG

postfix:
AB+
AB+CDE-$F+*G-
```

> b. A+(((B-C)*(D-E)+F)/G)$(H-J) prefix: A+((-BC)*(-DE)+F)/G)$(-HJ) A+(((/+-*BC-DEFG))) +A#/+*-BC-DEFG-HJ

postfix: A+(((BC-)*(DE-)+F/G)$(HJ-)) A B C - D E - * F + G /H J-$ +

Question Eight

> Transform each of the following expressions to infix expressions. a. ++A-*$BCD/+EF*GHI (E/F)+(G*H))+ I ((B*C)$D) A + (((B*C)$D)-(E/F)+(G*H) + I) b. +-$ABCD**EFG ((A$B)-C)+(D*(EFG) c. AB-C+DEF-+$ (A-B)+C, (F(D-E))$ ((D-E) + F) $ ((A-B) + C)

> d. ABCDE-+$*EF- ((D-E)+C)$B)*A -(E*F)

Question Nine

> Apply the evaluation algorithm in the text to evaluate the following postfix expressions, where A=1, B=2, and C=3. a. AB+C-BA+C$- [1] [1,2] [3] [3,3] [0] [0,2,1] [0,3] [0,3,3] [0,27] -27 b. ABC+*CBA-+*
> [1] [1,2] [1,2,3] [1,5] [5] [5,3] [5,3,2] [5,3,2,1] [5,3,1] [5,4] [20

Question Ten

> Write a prefix function to accept an infix string and create the prefix form of that string, assuming that the string is read from right to left and that the prefix string is created from right to left. Handle variables, +,-,/,*,$, (,).

```
function infix_to_prefix(infix_string)
    // flip infix string
    char arr[len(infix_string)]
    stack s;
    for(int i = 0; i < len(infix_string); i++) {
        arr[len(infix_string) -1 -i] = infix_string[i] if not ( or ) else
if ( gets ) or if _) gets (
    }
    char out[$] // Queue
    for(int i = 0; i < len(infix_string); i++ {
        if arr[i] not "(,),+,-,*,/,$:
            out.push(arr[i])
        else if arr[i] == (:
            s.push(arr[i])
        else if arr[i] == ):
            while !s.empty && stack.top != ) {
                out.push(stack.pop())
            }
            stack.pop() // resolve everything in the stack
        else if arr[i] inside +, -, *, /, $L
            while !s.empty and stack.top > precedence(arr[i])
                out.push(stack.pop)
            stack.push(arr[i]
        )
    }
    // Empty stack
    while !s.empty():
        q.push(stack.pop)
    reverse(out) and return out
}
```