Q1

(a)

```
while( !main.isEmpty() ) // move all items to aux stack from main stack
   temp = main.pop()
   aux.push( temp )
i = temp // set i to temp, which is equal to the last (bottom) item of stack.
while( !aux.isEmpty() ) // move all items to main stack from aux stack
   main.push( aux.pop() )
```

(b)

```
while( !mainstack.isEmpty() ) // move all items to aux stack from main stack
   aux.push( temp )
for ( a=0; a <= 2; a++ ) // move first three items to main stack from aux stack
   temp = aux.pop() // set temp variable to popped values
   main.push( temp )
i = temp // set i to temp, which is equal to the third item from bottom of stack
while( !aux.isEmpty() ) // move all remaining items to main stack from aux stack
   main.push( aux.pop() )
```

Q2

(a) Since there is still an item remaining in the stack at the end, the delimiters are not correct.

Item	Description	Stack
{	push { onto stack	{
[	push [onto stack	{[
]	pop [from stack	{
[	push [onto stack	{[
(	push (onto stack	{[(
)	pop (from stack	{[
]	pop [from stack	{

(b) Since the stack is empty, the delimiters are valid.

Item	Description	Stack
(	push (onto stack	(
(	push (onto stack	((
)	pop (from stack	(
{	push { onto stack	({
(	push (onto stack	({(
[	push [onto stack	({([
]	pop [from stack	({(
)	pop (from stack	({
}	pop { from stack	(
)	pop (from stack	empty

Q3

```
/* Java/pseudocode algorithm to check if str follows form x C y */
if( str=null OR str.length=0 OR charAt(0)=C OR charAt(str.length-1)=C ) // check if
    string is valid
   return false
else // string is valid, proceed
   i = 0
   while( charAt(i) != C ) // push all chars before C to aux stack
      aux.push( charAt(i) )
      i++
   temp = charAt(i) // set temp equal to C
   i++
   while( charAt(i) = aux.peek() ) // check if chars after C mimic/symmetric to chars
       before C
      aux.pop() // if so, pop from aux stack and check next char
      i++
      if( i = str.length) // break if end of string is reached
         break
   if( aux.isEmpty() ) // if stack is empty, string follows form
      return true
   else
      return false // if stack is not empty, string does not follow form
```

Q4

```
/* Java/pseudocode algorithm to check if str follows form 'a D b D c D ... D z' */
if( str=null OR str.length=0 OR charAt(0)=D OR charAt(str.length-1)=D ) // check if
    string is valid
   return false
else // string is valid, proceed
   start = 0
   for( int i = 0; i < str.length; i++ ) // iterate through every character in string
      if( charAt(i) = D )
         end = i
         valid = Problem3Solution( str.substring(start,end) ) // run the substring
             (string before/between/after D) through previous question solution
         if( valid = false ) // if Problem3Solution returns false, then this will return
             false to indicate string does not follow form
            return false
         start = end + 1 // set new substring start
   valid = Problem3Solution( str.substring(start,str.length) // run the last substring
       (following last D) through Problem3Solution method.
   if( valid = true )
      return true
   else
      return false
```

Q5

The requirement is to "design and implement a stack in which each item on the stack is a varying number of integers." I interpret this to mean that each item in the stack is an array of integers. Thus, the stack can be implemented by the int[][] data type, also known as a 2D int array or an array of int arrays. In Java, the size of the arrays must be declared upon creation (or specify the array literals); however, each element in the 'primary' array can be explicitly set to a different size after initialization.

```
/* Java/pseudocode to implement stack with int[] elements */
size = 10 // set static stack size
top = -1 // set top of empty stack to -1 (since first element will be at index 0)
int[][] stack = new int[size][1]; // creates a stack of size ten, initialized with ten
    int[1] arrays (which initialize to zero by default)

/* push method */
if( top < size-1 )
   top++
   stack[top] = element // element is an int[]
else
   no room to add!

/* pop method */
if( top > -1 )
   temp = stack[top]
   top--
   return temp
else
   nothing to pop!

/* empty method */
if( top == -1 )
   return true
else
   return false
```

Q6

There are four primary functions for the 1D array - insert to end, delete from end, insert at i^{th} element, and delete i^{th} element. The main stack is representative of the array, and the auxiliary stack is used for insert and delete functions.

- insert to end $\rightarrow$ push to main stack
- delete from end $\rightarrow$ pop from main stack
- insert at i^{th} $\rightarrow$
    1. move all items to aux stack
    2. move i-1 items back to main stack
    3. push insert item to main stack
    4. move remainder of items back to main stack
- delete i^{th} $\rightarrow$
    1. move all items to aux stack
    2. move i-1 items back to main stack
    3. pop i^{th} item
    4. move remainder of items back to main stack

Q7

This can be accomplished by having one stack start at 0 and increase in size (top of stack i++). The other stack would start at array.length-1 and increase in size by moving the top of the stack towards 0 (top of stack i–).


```
/* Java/pseudocode to implement two stacks in single array */
s = new int[size]
top1 = -1
top2 = size

/* push1 method Java/pseudocode */
if( top1 < top2-1 )
   top1++
   s[top1] = item
else
   no room to push!

/* pop1 method Java/pseudocode */
if( top1 >= 0 )
   item = s[top1]
   top1--
   return item
else
   nothing to pop!

/* push2 method Java/pseudocode */
if( top1 < top2-1 )
   top2--
   s[top2] = item
else
   no room to push!

/* pop2 method Java/pseudocode */
if( top2 < size )
   item = s[top2]
   top2++
   return item
else
   nothing to pop!
```

Q8

(a) Prefix

$$(A+B)*(C\$(D-E)+F)-G$$
$$(+AB)*(C\$(-DE)+F)-G$$
$$(+AB)*(\$C-DE)+F)-G$$
$$(+AB)*(+\$C-DEF)-G$$
$$\boxed{\text{-*+AB+\$C-DEFG}}$$

Postfix

$$(A+B)*(C\$(D-E)+F)-G$$
$$(AB+)*(C\$(DE-)+F)-G$$
$$(AB+)*(CDE-\$)+F)-G$$
$$(AB+)*(CDE-\$F+)-G$$
$$(AB+CDE-\$F+*)-G$$
$$\boxed{\text{AB+CDE-\$F+*G-}}$$

(b) Prefix

$$A+(((B-C)*(D-E)+F)/G)\$(H-J)$$
$$A+(((-BC)*(-DE)+F/G)(-HJ)$$
$$A+\$((*(-BC-DE)+F)/G)(-HJ)$$
$$A+\$((+*-BC-DEF)/G)(-HJ)$$
$$A+\$/+*-BC-DEFG-HJ$$
$$\boxed{\text{+A\$/+*-BC-DEFG-HJ}}$$

Postfix

$$A+(((B-C)*(D-E)+F)/G)\$(H-J)$$
$$A+(((BC-DE-*)+F)/G)\$(HJ-)$$
$$A+((BC-DE-*F+)/G)\$(HJ-)$$
$$A+(BC-DE-*F+G/)\$(HJ-)$$
$$A+(BC-DE-*F+G/HJ-\$)$$
$$\boxed{\text{ABC-DE-*F+G/HJ-\$+)}}$$

Q9

(a)

Item	Stack
I	I
H	I,H
G	I,H,G
*	I,G*H
F	I,G*H,F
E	I,G*H,F,E
+	I,G*H,E+F
/	I,E+F/G*H
D	I,E+F/G*H,D
C	I,E+F/G*H,D,C
B	I,E+F/G*H,D,C,B
$	I,E+F/G*H,D,B$C
*	I,E+F/G*H,B$C*D
-	I,B$C*D-E+F/G*H
A	I,B$C*D-E+F/G*H,A
+	I,A+B$C*D-E+F/G*H
+	A+B$C*D-E+F/G*H+I

(b)

Item	Stack
G	G
F	G,F
E	G,F,E
*	G,E*F
*	E*F*G
D	E*F*G,D
*	D*E*F*G
C	D*E*F*G,C
B	D*E*F*G,C,B
A	D*E*F*G,C,B,A
$	D*E*F*G,C,A$B
-	D*E*F*G,A$B-C
+	A$B-C+D*E*F*G

(c)

Item	Stack
A	A
B	A,B
-	A-B
C	A-B,C
+	A-B+C
D	A-B+C,D
E	A-B+C,D,E
F	A-B+C,D,E,F
-	A-B+C,D,E-F
+	A-B+C,D+E-F
$	A-B+C$D+E-F

(d)

Item	Stack
A	A
B	A,B
C	A,B,C
D	A,B,C,D
E	A,B,C,D,E
-	A,B,C,D-E
+	A,B,C+D-E
$	A,B$C+D-E
*	A*B$C+D-E
E	A*B$C+D-E,E
F	A*B$C+D-E,E,F
*	A*B$C+D-E,E*F
-	A*B$C+D-E-E*F

Q10

(a)

$$AB + C - BA + C\$ - \quad A = 1;\ B = 2;\ C = 3$$
$$(1 + 2) - 3 - (2 + 1)^3$$
$$3 - 3 - 3^3$$
$$0 - 3^3$$
$$0 - 27$$
$$\boxed{\text{-27}}$$

(b)

$$ABC + *CBA - + * \quad A = 1;\ B = 2;\ C = 3$$
$$1 * (2 + 3) * (3 + (2 - 1))$$
$$(1 * 5) * (3 + 1)$$
$$5 * 4$$
$$\boxed{20}$$

Q11

The general algorithm is the following:

- add string to main stack, left to right
- run the following until main stack is empty
  - if top of main stack is operand, add to final stack
  - if top of main stack is ‘)’ or operator with equal or higher precedence than top of opstack, add to opstack
  - if top of main stack is ‘(’, pop ‘(’ from main stack, move operators from opstack to final stack until ‘)’ is encountered, then pop the ‘)’ from opstack
  - if top of main stack is operator with lower precedence than top of opstack, pop opstack to final stack until top of main stack is no longer lower precedence, then pop top of main stack to opstack.
- finally, move remainder of opstack to final stack
- convert final stack back to string in regular order to obtain prefix expression string

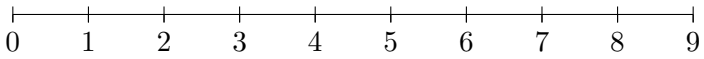
```
/* Java/pseudocode for conversion of infix string to prefix string */
for( i=0; i< str.length; i++) // move all chars in string to main stack
   main.push( charAt(i) )

while( !main.isEmpty ) // run until main stack is empty
   if( main.peek is operand) // if top of main stack is operand, move to final stack
      final.push( main.pop() )

   elseif( main.peek = ')' OR operatorHierarchy( main.peek ) >= operatorHierarchy(
       opstack.top ) ) // if top of main stack is ')' or has higher priority than top of
       opstack, add to opstack
      opstack.push( main.pop() )

   elseif( main.peek = '(' ) // if top of main stack is '(' then add all from opstack to
       final stack until reaching a ')' in opstack
      main.pop()
      while( opstack.peek != ')' )
         final.push( opstack.pop() )
      opstack.pop() // pop ')' from opstack

   elseif( operatorHierarchy( main.peek ) < operatorHierarchy( opstack.top ) ) // if top
       of main stack is has lower priority than top of opstack, add top of opstack to
       final stack and top of main stack to opstack
      while( !opstack.isEmpty AND (operatorHierarchy( main.peek ) < operatorHierarchy(
          opstack.top )) )
         final.push( opstack.pop() )
         opstack.push( main.pop() )

while( !opstack.isEmpty ) // main stack is empty, add what's left of opstack to final
    stack
   final.push( opstack.pop() )
```

Q12

The answer is lg(n) rounded up to the nearest whole number. This is given by:

$$\frac{n}{2^{max}} < 1$$

For example, in the following figure if our search is for 0, we could halve (round up) and guess element 5. 0 is less than five so it must be between elements 0 and 4. We could halve again and guess element 2. We could halve again and guess element 1. Finally, the only remaining item is element 0. Thus, depending on how the search is executed, one could say it took at most three calls (lg(n) rounded down to nearest whole) if you go by process of elimination, or four calls (lg(n) rounded up to nearest whole) if you actually use the search algorithm to 'land' on the desired value.



Q13

Three cases are provided. One of them serves as a base case that returns an integer. The other two are already described as calling the gcd method again. Thus, the recursive method is as follows:

```
/* Java/pseudocode for recursive GCD method */
GCD(int x, int y){
if( y <= x AND x%y == 0 )
   return y
else if( x < y )
   return GCD(y,x)
else
   return GCD(y,x%y)
}
```