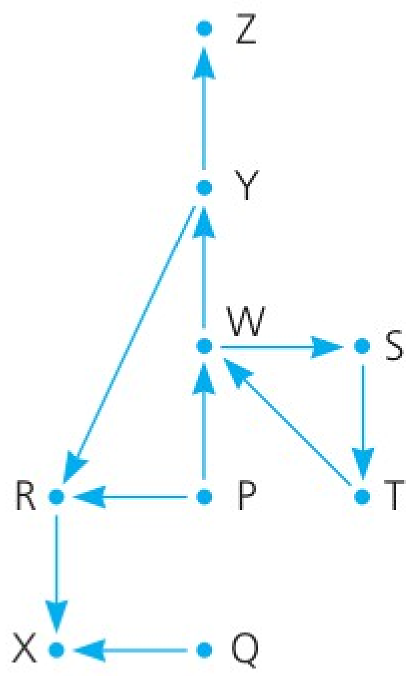
```
// Checks the string aString to verify that braces match.
// Returns true if aString contains matching braces, false otherwise.
checkBraces(aString: string): boolean

   aStack = a new empty stack
   balancedSoFar = true
   i = 0

   while (balancedSoFar and i < length of aString)
   {
      ch = character at position i in aString
      i++

      // Push an open brace
      if (ch is a '{')
         aStack.push('{')

      // Close brace
      else if (ch is a '}')
      {
         if (!aStack.isEmpty())
            aStack.pop()     // Pop a matching open brace
         else                // No matching open brace
            balancedSoFar = false
      }
      // Ignore all characters other than braces
   }

   if (balancedSoFar and aStack.isEmpty())
      aString has balanced braces
   else
      aString does not have balanced braces
```

```
for (each character ch in the string)
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack

        operand1 = top of stack
        Pop the stack

        result = operand1 op operand2
        Push result onto the stack
    }
}
```

```
for (each character ch in the infix expression)
{
    switch (ch)
    {
        case operand:           // Append operand to end of postfix expression—step 1
            postfixExp = postfixExp • ch
            break
        case '(':               // Save '(' on stack—step 2
            aStack.push(ch)
            break
        case operator:          // Process stack operators of greater precedence—step 3
            while (!aStack.isEmpty() and aStack.peek() is not a '(' and
                    precedence(ch) <= precedence(aStack.peek()))
            {
                Append aStack.peek() to the end of postfixExp
                aStack.pop()
            }
            aStack.push(ch) // Save the operator
            break
        case ')':               // Pop stack until matching '('—step 4
            while (aStack.peek() is not a '(')
            {
                Append aStack.peek() to the end of postfixExp
                aStack.pop()
            }
            aStack.pop()        // Remove the open parenthesis
            break
    }
}

// Append to postfixExp the operators remaining in the stack—step 5
while (!aStack.isEmpty())
{
    Append aStack.peek() to the end of postfixExp
    aStack.pop()
}
```

```
// Searches for a sequence of flights from originCity to destinationCity
searchS(originCity: City, destinationCity: City): boolean

   aStack = a new empty stack
   Clear marks on all cities

   aStack.push(originCity) // Push origin onto the stack
   Mark the origin as visited

   while (!aStack.isEmpty() and destinationCity is not at the top of the stack)
   {
      // Loop invariant: The stack contains a directed path from the origin city at
      // the bottom of the stack to the city at the top of the stack
      if (no flights exist from the city on the top of the stack to unvisited cities)
         aStack.pop() // Backtrack
      else
      {
         Select an unvisited destination city C for a flight from the city on the top of  the stack
         aStack.push(C)
         Mark C as visited
      }
   }
   if (aStack.isEmpty())
      return false // No path exists
   else
      return true  // Path exists
```



Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z