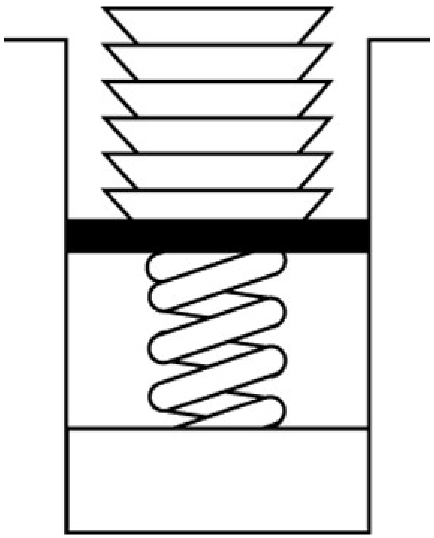
# Stacks

# Developing an ADT During the Design of a Solution

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria



# Stack

- ADT stack operations
  - Create an empty stack
  - Determine whether a stack is empty
  - Add a new item to the stack
  - Remove from the stack the item that was added most recently
  - Remove all the items from the stack
  - Retrieve from the stack the item that was added most recently

# The Abstract Data Type: Developing an ADT During the Design of a Solution

- Specifications of an abstract data type for a particular problem
  - Can emerge during the design of the problem’s solution
  - Examples
    - `SolveMaze` algorithm
    - `displayBackward` algorithm

# Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

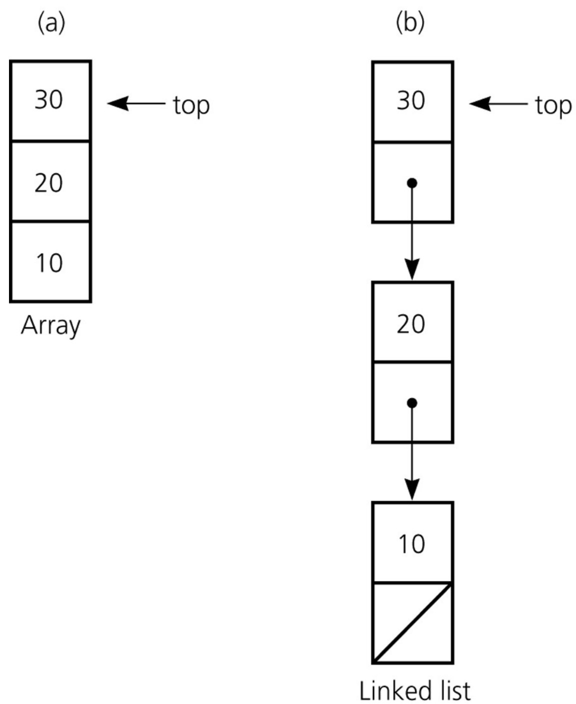
    `abc{defg{ijk}{l{mn}}op}qr`
  - An example of unbalanced braces

    `abc{def}}{ghij{kl}m`

# Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list

# Implementations of the ADT Stack




The ADT stack can be implemented using

a) An array
b) A linked list

# Implementing Stacks (ArrayList or LinkedList

## Using an array list to implement Stack

- Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list.

# Design of the Stack

There are two ways to design the Stack class

- Using inheritance: You can define the stack class by extending the array list class, or the stack class by extending the linked list class.


(a) Using inheritance

– Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.


(b) Using composition

# MyStack and MyQueue

| GenericStack<E> | |
|---|---|
| -list: java.util.ArrayList<E> | An array list to store elements. |
| +GenericStack() | Creates an empty stack. |
| +getSize(): int | Returns the number of elements in this stack. |
| +peek(): E | Returns the top element in this stack. |
| +pop(): E | Returns and removes the top element in this stack. |
| +push(o: E): void | Adds a new element to the top of this stack. |
| +isEmpty(): boolean | Returns true if the stack is empty. |

# The Java Collections Framework Class **`Stack`**

- JCF contains an implementation of a stack class called `Stack` (generic)
- Derived from `Vector`
- Includes methods: `peek, pop, push, and search`
- `search` returns the 1-based position of an object on the stack

# Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

# Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

# Evaluating Postfix Expressions

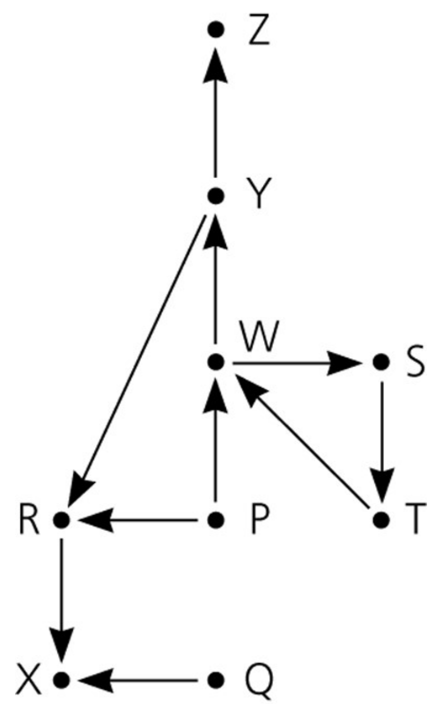
- A postfix calculator
  - Requires you to enter postfix expressions
    - Example: 2, 3, 4, +, *
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

# Application: A Search Problem

- High Planes Airline Company (HPAir)
  - Problem
    - For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

# Representing the Flight Data

- The flight map for HPAir is a graph
  - Adjacent vertices
    - Two vertices that are joined by an edge
  - Directed path
    - A sequence of directed edges

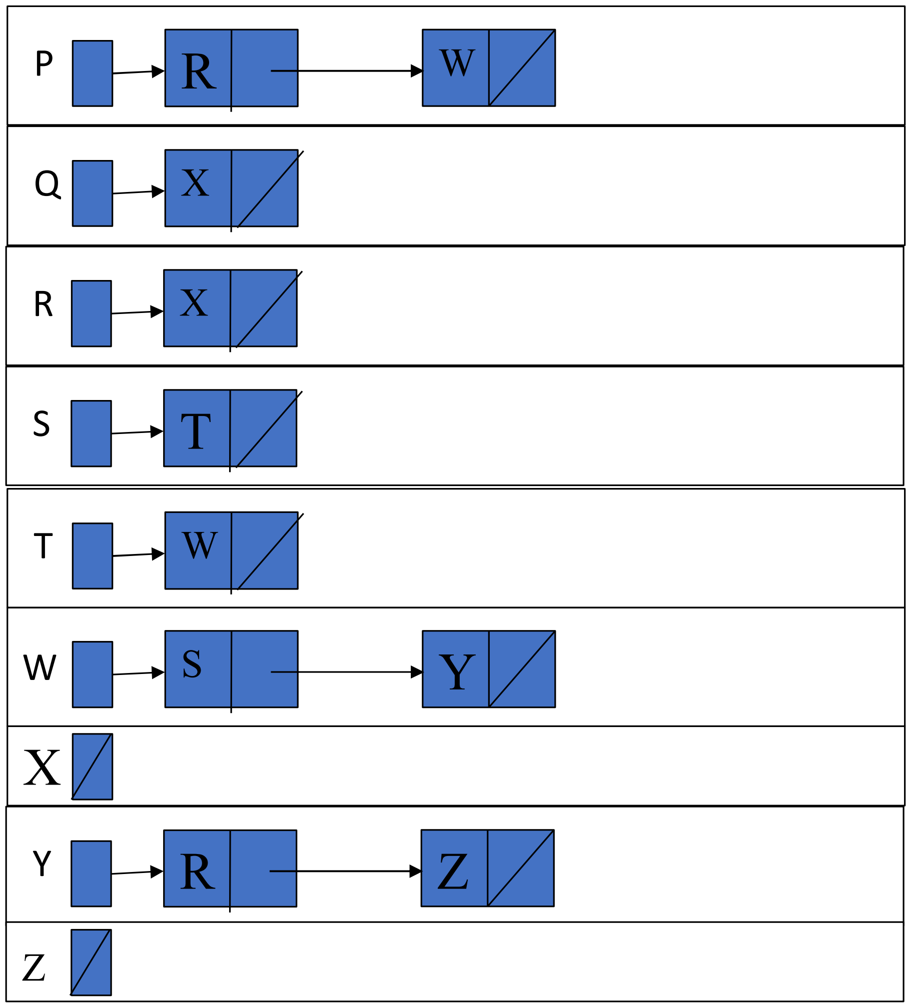



Flight map for HPAir

# A Nonrecursive Solution that Uses a Stack

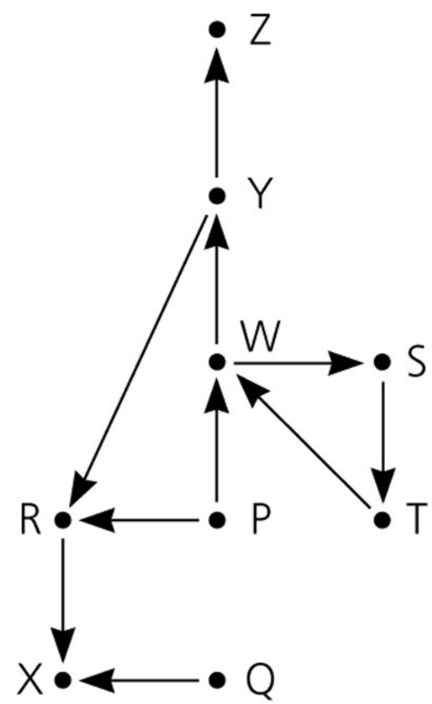
- The solution performs an exhaustive search
  - Beginning at the origin city, the solution will try every possible sequence of flights until either
    - It finds a sequence that gets to the destination city
    - It determines that no such sequence exists
- The ADT stack is useful in organizing an exhaustive search
- Backtracking can be used to recover from a wrong choice of a city

# Representing the Flight Data



# A Non-Recursive Solution

```
Create a stack (aStack)
Create a visited list (LinkedList)
aStack.push(originalCity)
add the originalCity to visited
While ( !aStack.isEmpty()) {
 if top of aStack is endcity, then
   print the content of aStack in reverse order
Else {
if ( no flight exist from city on top of stack to unvisited cities)
     temp = aStack.pop()
  else {
     select an unvisited dest. city c for a flight from the city on the top of the stack
      aStack.push(c)
      mark c as visited
    }
 }
}
```



# A Recursive Solution

- Possible outcomes of the recursive search strategy
  - You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
  - You reach a city C from which there are no departing flights
  - You go around in circles

# A Recursive Solution

- A refined recursive search strategy

```
searchR(originCity, destinationCity)
  Mark originCity as visited
  if (originCity is destinationCity) {
    Terminate -- the destination is reached
  }
  else {
    for (each unvisited city C adjacent to originCity) {
      searchR(C, destinationCity)
    }
  }
```

# The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm