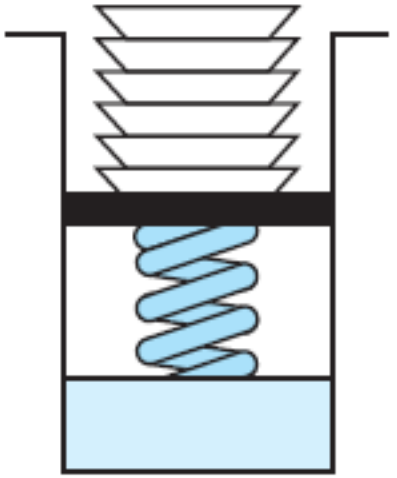
EECE 2560:
Fundamentals of Engineering Algorithms

Stacks

The Abstract Data Type Stack

- Operations on a stack
  - Last-in, first-out (LIFO) behavior.
  - Example: A stack of cafeteria plates.



- Applications demonstrated
  - Evaluating algebraic expressions
  - Searching for a path between two points
  - Storage of local variables between function calls.

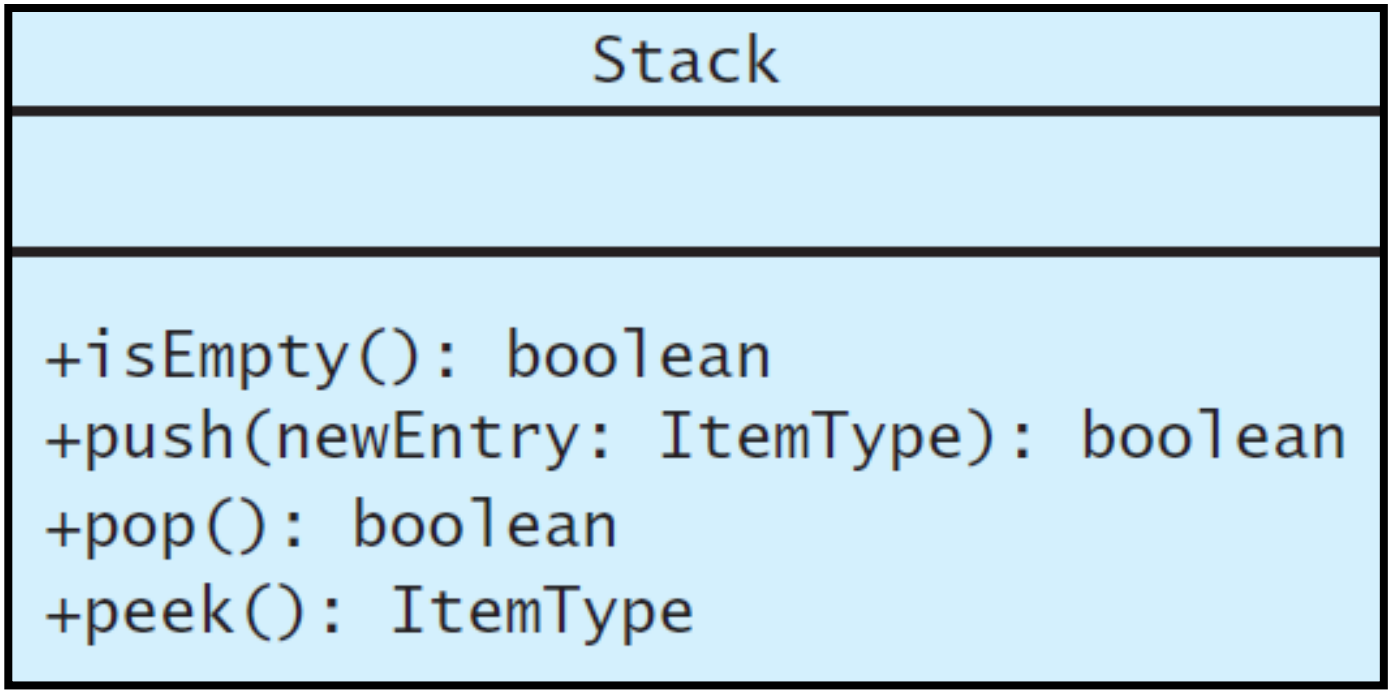
ADT for Typing a Line of Text (1 of 2)

- Consider typing a line of text on a keyboard
  - Use of backspace key to make corrections
    - You type `abcc←ddde←←←eg←fg`
    - Corrected input will be `abcdefg`
- Must decide how to store the input line.

ADT for Typing a Line of Text (2 of 2)

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else if (the ADT is not empty)
        Remove from the ADT the item that was added most recently
    else
        Ignore the '←'
}
```

UML diagram for the class Stack



Stack Interface (1 of 2)

```
template<class ItemType>
class StackInterface
{
 public :
  /** Sees whether this stack is empty.
  @return True if the stack is empty, or false if not. */
  virtual bool isEmpty() const = 0;

  /** Adds a new entry to the top of this stack.
  @post If the operation was successful, newEntry is at the top
     of the stack.
  @param newEntry The object to be added as a new entry.
  @return True if the addition is successful or false if not. */
 virtual bool push( const ItemType& newEntry) = 0;
```

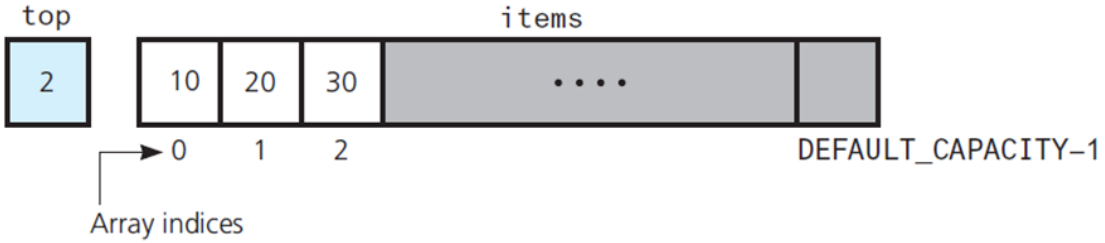
Stack Interface (2 of 2)

```
   /** Removes the top of this stack.
   @post If the operation was successful, the top of the stack has
      been removed.
   @return True if the removal is successful or false if not. */
  virtual bool pop() = 0;

   /** Returns the top of this stack.
   @pre The stack is not empty.
   @post The top of the stack has been returned, and the stack is
      unchanged.
   @return The top of the stack. */
  virtual ItemType peek() const = 0;
}; // end StackInterface
```

An Array-Based Implementation (1 of 3)

- Header File:




```
#include "StackInterface.h"
template<class ItemType>
class ArrayStack : public StackInterface<ItemType> {
private:
   static const int DEFAULT_CAPACITY = 50;
   ItemType items[DEFAULT_CAPACITY]; // Array of stack items
   int      top;                     // Index to top of stack

public:
         ArrayStack();                    // Default constructor
         bool isEmpty() const;
         bool push(const ItemType& newEntry);
         bool pop();
         ItemType peek() const;
}; // end ArrayStack
```

An Array-Based Implementation (2 of 3)

- *The implementation file:*

```
template<class ItemType>
ArrayStack<ItemType>::ArrayStack() : top(-1)
{ }  // end default constructor

template<class ItemType>
bool ArrayStack<ItemType>::isEmpty() const{
       return top < 0;       }  // end isEmpty

template<class ItemType>
bool ArrayStack<ItemType>::push(const ItemType& newEntry) {
  bool result = false;
  if (top < DEFAULT_CAPACITY - 1) {  // Does stack have room for newEntry?
                top++;
                items[top] = newEntry;
                result = true;
       }  // end if
  return result;
}  // end push
```

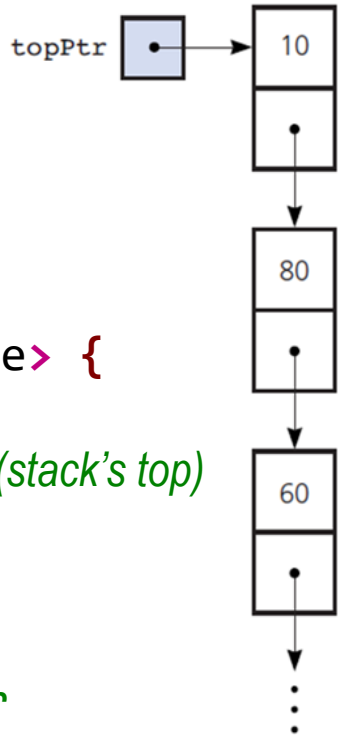
An Array-Based Implementation (3 of 3)

```
template<class ItemType>
bool ArrayStack<ItemType>::pop() {
        bool result = false;
        if (!isEmpty()) {
                top--;
                result = true;
        }  // end if
        return result;
}  // end pop

template<class ItemType>
ItemType ArrayStack<ItemType>::peek() const {
   assert(!isEmpty()); // If empty, the program is terminated and display an error message
   // Stack is not empty; return top
   return items[top];
}  // end peek
// End of implementation file.
```

A Link-Based Implementation (1 of 4)

- *The header file:*




```
#include "StackInterface.h"
#include "Node.h"

template<class ItemType>
class LinkedStack : public StackInterface<ItemType> {
private:
  Node<ItemType>* topPtr; // Pointer to first node in the chain (stack's top)

public:
// Constructors and destructor:
        LinkedStack();      // Default constructor
        ~LinkedStack();     // Destructor
// Stack operations:
        bool isEmpty() const;
        bool push(const ItemType& newItem);
        bool pop();
        ItemType peek() const;
}; // end LinkedStack
```

A Link-Based Implementation (2 of 4)

- *The implementation file:*

```
template<class ItemType>
LinkedStack<ItemType>::LinkedStack() : topPtr(nullptr)
      { }  // end default constructor

template<class ItemType>
LinkedStack<ItemType>::~LinkedStack() {
       // Pop until stack is empty
       while (!isEmpty()) pop();
}  // end destructor

template<class ItemType>
bool LinkedStack<ItemType>::push(const ItemType& newItem) {
 Node<ItemType>* newNodePtr = new Node<ItemType>(newItem, topPtr);
                                  // Node class is implemented in Lecture 6
   topPtr = newNodePtr;
   newNodePtr = nullptr;
  return true;
}  // end push
```

A Link-Based Implementation (3 of 4)

```
template<class ItemType>
bool LinkedStack<ItemType>::pop() {
  bool result = false;
  if (!isEmpty()) {
      // Stack is not empty; delete top
    Node<ItemType>* nodeToDeletePtr = topPtr;
    topPtr = topPtr->getNext();

     // Return deleted node to system
    nodeToDeletePtr->setNext(nullptr); //defensive step but not needed
    delete nodeToDeletePtr;
    nodeToDeletePtr = nullptr; //defensive step but not needed
    result = true;
  }  // end if

  return result;
}  // end pop
```

A Link-Based Implementation (4 of 4)

```
template<class ItemType>
ItemType LinkedStack<ItemType>::peek() const
{
  assert(!isEmpty()); // If empty, the program is terminated and display an error message

  // Stack is not empty; return top
  return topPtr->getItem();
}  // end getTop

template<class ItemType>
bool LinkedStack<ItemType>::isEmpty() const
{
   return topPtr == nullptr;
}  // end isEmpty

// End of implementation file.
```

Checking for Balanced Braces (1 of 4)

- Example of curly braces in C++ language
  - Balanced `abc{defg{ijk}{l{mn}}op}qr`
  - Not balanced `abc{def}}{ghij{kl}m`
- Requirements for balanced braces
  - For each **}**, must match an already encountered **{**
  - At end of string, must have matched each **{**

Checking for Balanced Braces (2 of 4)

Initial draft of a solution.

```
for (each character in the string)
{
    if (the character is a '{')
        aStack.push('{')
    else if (the character is a '}')
        aStack.pop()
}
```

Checking for Balanced Braces (3 of 4)

Detailed pseudocode solution:

```
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++
 if (ch is a '{') aStack.push('{') // Push an open brace
 else if (ch is a '}') {// Close brace
  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
```

Checking for Balanced Braces (4 of 4)

Traces of algorithm that checks for balanced braces

Input string	Stack as algorithm executes	

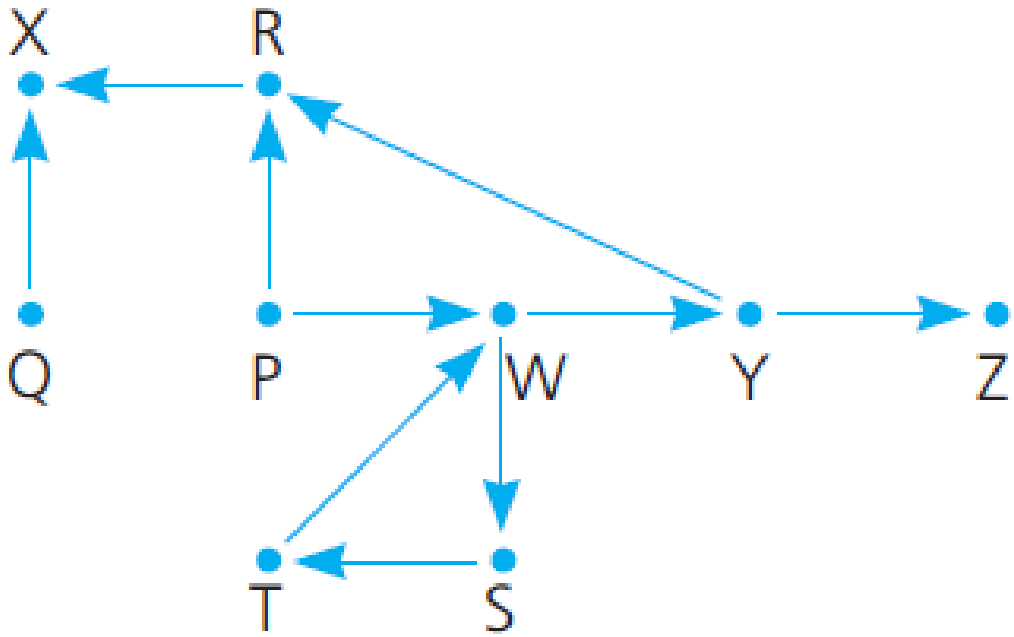
{a{b}c}	1. {
	 2. { {
	 3. {
	 4.
		1. push {
		 2. push {
		 3. pop
		 4. pop
		 Stack empty ⟹ balanced

{a{bc}	1. {
	 2. { {
	 3. {
		1. push {
		 2. push {
		 3. pop
		 Stack not empty ⟹ not balanced

{ab}c}	1. { 2.	1. push { 2. pop Stack empty when "}" encountered ⟹ not balanced

Search a Flight Map (1 of 8)

A flight map



Search a Flight Map (2 of 8)

Recall recursive search strategy.

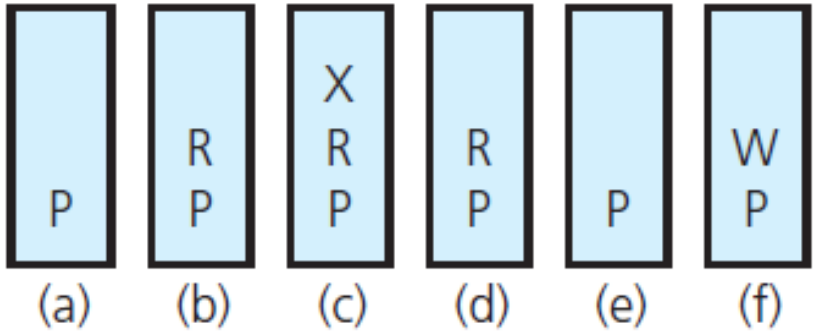
```
To fly from the origin to the destination
{
    Select a city C adjacent to the origin
    Fly from the origin to city C
    if (C is the destination city)
        Terminate— the destination is reached
    else
        Fly from city C to the destination
}
```

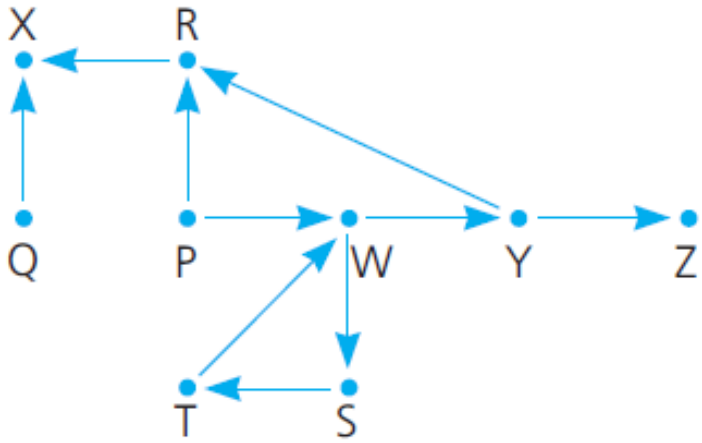
Search a Flight Map (3 of 8)

- Possible outcomes of exhaustive search strategy
  1. Reach destination city, decide possible to fly from origin to destination
  2. Reach a city, *C* from which no departing flights
  3. You go around in circles
- Use backtracking to recover from a wrong choice (2 or 3)

Search a Flight Map (4 of 8)

- Strategy requires information about order in which it visits cities.
- The stack of cities as you travel from P to W:





Search a Flight Map (5 of 8)

- Stack will contain directed path from
  - Origin city at bottom to …
  - Current visited city at top

- When to backtrack
  - No flights exist from the city on the top of the stack to unvisited cities.

Search a Flight Map (6 of 8)

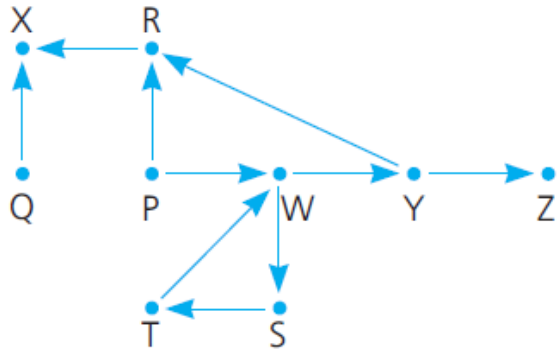
Final draft of algorithm:

```
// 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
```

Search a Flight Map (7 of 8)

A trace of `searchS(P,Z)`:

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



Search a Flight Map (8 of 8)

C++ implementation of **searchS**:

```
bool Map::isPath(City originCity, City destinationCity){
   bool success;
   Stack aStack;
   unvisitAll(); // Clear marks on all cities
   aStack.push(originCity); // Push origin city onto aStack and mark it as visited
   markVisited(originCity);
   City topCity = aStack.peek();
   while (!aStack.isEmpty() && (topCity != destinationCity)) {
   // 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. Find an unvisited city adjacent to the city on the top of the stack
      City nextCity = getNextCity(topCity);
      if (nextCity == NO_CITY) aStack.pop(); // No city found; backtrack
      else // Visit city {
         aStack.push(nextCity);
         markVisited(nextCity);
      } // end if
      if (!aStack.isEmpty())  topCity = aStack.peek();
   } // end while
   return !aStack.isEmpty();
} // end isPath
```