Stacks

孫雅麗
台灣大學 資管所
Outline

- What is a “stack”?
  - Concept
  - Operations – pop and push
- Simple applications of “stack”
**Stack**

- A **STACK** is a special case of an ordered list in which "insertion" and "deletion" are made only at the first position of the list called "TOP".

\[
S = (a_0, \ldots, a_{n-1})
\]

- \(a_{i-1}\) is on top of \(a_i\), \(n > i > 0\)
- \(a_{n-1}\) : bottom
- \(a_0\) : top

---

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Stack (cont’d)

- **insertion**($S, X$): “push”
  
  $S = (X, a_0, \ldots, a_{n-1});$

- **deletion**($S$): “pop”
  
  $S = (X, a_0, \ldots, a_{n-1}); \text{ return } (X)$
Stack (cont’d)

- Stacks are also referred to as **LAST-IN-FIRST-OUT (LIFO)** lists.
Pseudocode for ADT Stack Operations (1/2)

`createStack()`
// Creates an empty stack.

`destroyStack()`
// Destroys a stack.

`stackIsEmpty()`
// Determines whether a stack is empty.

`push(in newItem) throw StackException`
// Adds `newItem` to the top of a stack.
// Throws Stack Exception if the insertion is not successful
ADT Stack Operations (2/2)

pop() throw StackException;

pop(out stackTop) throw StackException
// Retrieves into stackTop and then removes the top of a stack. That is, retrieves and removes the item that was added most recently.

getTop(out stackTop) throw StackException
// Retrieves into stackTop the top of a stack. That is, retrieves the item that was added most recently.
// Retrieval does NOT change the stack.
while ( not at the end of the string )
{
    if ( the next character is a '{' )
        s.push( '{' )
    else if ( the character is a '}' )
        s.pop()
}
Application #1: Balanced Braces (2/5)

Input string | Stack as algorithm executes
--- | ---
\{a\{b\}c\} | 1. Push "{"
| | 2. Push "{"
| | 3. Pop
| | 4. Pop
| | Stack empty $\Rightarrow$ balanced
\{a\{bc\} | 1. Push "{"
| | 2. Push "{"
| | 3. Pop
| | Stack not empty $\Rightarrow$ not balanced
\{ab\}c\} | 1. Push "{"
| | 2. Pop
| | Stack empty when last "}" encountered $\Rightarrow$ not balanced

Figure 6-2
Traces of the algorithm that checks for balanced braces

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
s.createStack()
balancedSoFar = true
i = 0

while (balancedSoFar and i < length of String)
{
    ch = string[i];
    ++i;

    // push an open brace
    if (ch is '{')
        s.push('{')
Application #1: Balanced Braces

// close brace
else if (ch is '}]')
    if (!s.isEmpty())
        s.pop() // pop a matching open brace
    else // no matching open brace
        balancedSoFar = false

// ignore all characters other than braces
} // end while
Application #1: Balanced Braces (5/5)

```
if (balancedSoFar and s.stackIsEmpty())
    String has balanced braces
else
    String does not have balanced braces
```

//end of detailed pseudocode solution

\[ O(n) \quad n : \text{length of the string} \]
Application #2: Recognizing Strings in a Language - problem (1/3)

$L = \{w$w': $w$ is a possibly empty string of characters other than $, $w' = \text{reverse}(w)\}$

e.g. $A$A, ABC$CBA, $ -> yes

e.g. AB$AB, ABC$ABC -> no
Application #2: Recognizing Strings in a Language - implementation (2/3)

// w$w', e.g., ABC$CBA

s.createStack()
// push the characters before $, that is, the
// characters in w, onto the stack
i = 0
while (string[i] is not '$')
{
    s.push(string[i])
    ++i
}
// end while

// skip the $
++i
Application #2: Recognizing Strings in a Language (3/3)

// match the reverse of w
inLanguage = true  // assume string is in language
while (inLanguage and i < length of String)
{
  s.pop(stackTop)
  if (stackTop equals string[i])
    ++i  // characters match
  else  // stack is empty (first half of string is shorter than second half)
    // or top of stack is not string[i] (characters do not match)
    inLanguage = false  // reject string
}
// end while

if (inLanguage and s.stackIsEmpty()) then string is in language
else string is not in language
Implementation of the ADT Stack

Array

Linked List

Position 1
Array-Based ADT Stack (1/8)

Figure 6-4
An array-based implementation
Array-Based ADT Stack (2/8)

// **************************************************
// Header file StackA.h for the ADT stack.
// Array-based implementation.
// **************************************************
#include "StackException.h"
const int MAX_STACK = maximum-size-of-stack;
typedef desired-type-of-stack-item stackItemType;

class Stack
{
public:
    // constructors and destructor:
    stack();   // default constructor
    // copy constructor and destructor are supplied by the compiler
Array-Based ADT Stack (3/8)

// stack operations:
bool isEmpty() const;
// Determines whether a stack is empty.
// Precondition: None.
// Postcondition: Returns true if the stack is empty;
// otherwise returns false.

void push(stackItemType newItem) throw (StackException);
// Adds an item to the top of a stack.
// Precondition: newItem is the item to be added.
// Postcondition: If insertion was successful, newItem
// is on the top of the stack.
// Exception: Throws StackException if the item cannot be placed
// on the stack, i.e. array is full.
Array-Based ADT Stack (4/8)

void pop() throw (StackException);
   // Removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, the item that was
   // added most recently is removed. However, if the stack was empty,
   // deletion is impossible.

void pop(stackItemType & stackTop) throw (StackException);
   // Retrieves and removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, StackTop contains the
   // item that was added most recently, the item is removed.
   // However, if the stack was empty, deletion is
   // impossible, stackTop is unchanged.
void getTop(stackItemType & stackTop) const
    throw (StackException);
    // Retrieves the top of a stack.
    // Precondition: None.
    // Postcondition: If the stack was not empty, stackTop contains the
    // item that was added most recently and Success is true. However,
    // if the stack was empty, the operation fails, stackTop is unchanged,
    // and Success is false. The stack is unchanged.

private:
    stackItemType items [MAX_STACK];  // array of stack items
    int top;  // index to top of stack
};  // end class
// End of header file.
Array-Based ADT Stack (6/8)

// ****************************************************
// Implementation file StackA.cpp for the ADT stack.
// Array-based implementation.
// ****************************************************
#include "Stack.h" // header file

Stack:: Stack(): top(-1)
{
} // end default constructor

bool Stack :: isEmpty() const
{
    return top < 0;
}
} // end Stack::isEmpty
Array-Based ADT Stack (7/8)

void Stack :: push(stackItemType newItem)
{
    if (top >= MAX STACK - 1) // if stack has no more room for another item
        throw StackException("StackException: stack full on push");
    else {
        ++top;
        items [top] = newItem;
    } // end if
} // end push

void Stack :: pop()
{
    if (isEmpty())
        throw StackException("StackException: stack empty on pop");
    else
        --top;  // pop top
} // end pop
Array-Based ADT Stack (8/8)

```cpp
void Stack::pop(stackItemType& stackTop)
{
  if (isEmpty())
    throw StackException("StackException: stack empty on pop");
  else // if stack is not empty,
  {
    stackTop = items[top]; // retrieve top
    --top; // pop top
  } // end if
} // end Pop

void Stack ::getTop(stackItemType& StackTop) const
{
  if (isEmpty())
    throw StackException("StackException: stack empty on pop");
  else // if stack is not empty,
    stackTop = items[top]; // retrieve top
} // end getTop // End of implementation file.
```
//*********************************************************
// Header file StackP.h for the ADT stack.
// Pointer-based implementation.
//
//******************************************************************************
**
#include “StackException.h”
typedef desired-type-of-stack-Item StackItemType;
class Stack
{
Public:
// constructors and destructor;
    stack();                           // default constructor
    stack(const stack & aStack);      // copy constructor
    ~stack();                         // destructor

// stack operations:
    bool isEmpty() const;
    void push(stackItemType NewItem) throw (StackException);
    void pop() throw (StackException);
    void pop(stackItemType& stackTop) throw (StackException);
    void getTop(stackItemType& stackTop) const throw (StackException);
private:

struct stackNode
{
    stackItemType item;
    stackNode *next;
}; // end struct

typedef stackNode* topPtr; // points to top of stack
}; // end class

// End of header file.
#include "stackP.h"  // header file
#include <cstddef>  // for NULL
#include <cassert>  // for assert
Pointer-Based ADT Stack (5/11)

```
stack:: stack() : topPtr(NULL) 
{}  // end default constructor

stack :: stack(const stack & aStack) // copy constructor
{
    if (aStack.topPtr == NULL)  // original list is empty
        topPtr = NULL;
    else {  // copy first node
        topPtr = new StackNode; // allocate a new space
        assert(topPtr != NULL);   // see if such space valid
        topPtr->item = aStack.topPtr->item;
```


// copy rest of list
stackNode * newPtr = topPtr;  // new list pointer
for (stackNode * origPtr = aStack.topPtr->next;
     origPtr != NULL; origPtr = origPtr->next)
{
    newPtr->next = new stackNode;
    assert(newPtr->next != NULL);
    newPtr = newPtr->next;
    newPtr->item = origPtr->item;
}
// end for
newPtr->next = NULL;  // mark the bottom of the stack

O(n)
Stack Destructor

```cpp
stack::~stack()
{
    bool success;
    // pop until stack is empty (Success is false)
    while(!isEmpty())
        pop();
    // Assertion: topPtr == NULL
}
    // end destructor
```

Stack Is Empty

```cpp
bool stack::isEmpty() const
{
    return topPtr == NULL);
}
    // end isEmpty
```
void stack::push(stackItem newItem)  
{  // create a new node  
    stackNode *newPtr = new stackNode;  
    if (newPtr == NULL);  // check allocation  
        throw StackException(  
            "StackException: stack push cannot allocate memory");  
    else  {  // allocation successful; set data portion of new node  
        newPtr->item = newItem;  
        // insert the new node  
        newPtr->next = topPtr;  
        topPtr = newPtr;  
    }  // end if  
}  // end Push
void Stack::pop()
{
    if (isEmpty())
        throw StackException(
            "StackException: stack empty on pop");
    else   {  // stack is not empty; delete top
        StackNode *temp = topPtr;
        topPtr = topPtr->next;

        // return deleted node to system
        temp->next = NULL;  // safeguard
        delete temp;
    }  // end if
}  // end pop
void stackClass::pop(StackItemType & stackTop)
{
    if (isEmpty())
        throw StackException(
            "StackException: stack empty on pop");
    else  {  // stack is not empty; delete top
        stackTop = topPtr->item;
        stackNode *temp = topPtr;
        topPtr = topPtr->next;
        // return deleted node to system
        temp->next = NULL;  // safeguard
        delete temp;
    }  // end if
}  // end Pop
void Stack::getTop(StackItemType & stackTop) const
{
    if (isEmpty())
        throw StackException(
            "StackException: stack empty on getTop");
    else
    // stack is not empty; retrieve top
        stackTop = topPtr->item;
} // end getTop

// End of implementation file.
ADT List Implementation (1/6)

List position

1 10
2 80
3 60
... ...
ListLength() 5

Top of stack

top is at position 1
ADT List Implementation (2/6)

// ****************************************************
// Header file StackL.h for the ADT stack.
// ADT list implementation.
// ******************************************************
#include "stackException.h"
#include "listP.h"    // list operations
typedef ListItemType StackItemType;

class stack
{
    public:
        // constructors and destructor:
        stack();             // default constructor
        stack(const stack & s);    // copy constructor
        ~ stack();          // destructor

ADT List Implementation (3/6)

// stack operations:
bool isEmpty() const;
void push(stackItemType newItem) throw (StackException);
void pop() throw (StackException);
void pop(stackItemType& stackTop) throw (StackException);
void getTop(stackItemType& stackTop) const throw (StackException);

private:
    List aList;  // list of stack items
};  // end class
// End of header file.
ADT List Implementation (4/6)

// **************************************************
// Implementation file StackL.cpp for the ADT stack.
// ADT list Implementation.
// **************************************************
#include "StackL.h"  // header file
stack:: stack()
{ }  // end default constructor

stack :: stack(const stack & aStack): aList(aStack.aList)
{ }  // end copy constructor

stack :~ stack()
{ }  // end destructor
See page 294

bool Stack::isEmpty() const
{  return aList.isEmpty();
}  // end isEmpty

void Stack::push(StackItemType newItem)
{  try
    {
        aList.insert(1, newItem); // insert new item at position 1
    } // end try
    catch (ListException e)
    {
        throw StackException("StackException: cannot push item");
    } // end catch
}  // end push
void stackClass::pop(stackItemType& stackTop, bool& success)
{
    L.listRetrieve(1, stackTop, success);
    L.listDelete(1, success);
    // Assertion: If list was empty at entry, success is false.
} // end Pop

void stackClass::GetStackTop(stackItemType& stackTop, bool& success) const
{
    L.listRetrieve(1, stackTop, success);
    // Assertion: If list was empty at entry, Success is false.
} // end GetStackTop
// End of implementation file.
Application #3: Read & Correct

readAndCorrect(sS)
// Reads the input line. For each character read, either enters it
// into stack S or, if it is '<-', corrects the contents of S.

s.createStack;
Read newChar;
while (newChar is not the end-of-line symbol)
{
  if (newChar is not '<-')
    s.push(newChar, success)
  else if (!s.stackIsEmpty())
    s.pop(success)
  Read newChar
}
  // end while

• ls eol
• la<-d<-s eol

1. read l, push (l): l
2. read a, push (a): la
3. read <-, pop(a): l
4. read d, push(d): ld
5. read <-, pop(d): l
6. read s, push(s): ls
Advanced Applications

- #4: Algebraic expressions
  - Converting *infix* to *postfix*
  - Evaluating *postfix* expressions

- #5: Search problems

- Relationship between stacks and recursion
Application #4: Evaluating Algebraic Expressions

- **Example:** *infix* notation
  - if \((i < \text{max\_size})\) \(k = i + j;\)

- **Postfix** notation: if \((i\ \text{max\_size} < )\) \(i\ j\ +;\)

- **Prefix**: if \((< i\ \text{max\_size})\) \(+ i\ j;\)

- **Evaluate algebraic expressions**
  - Convert an *infix* expression into an equivalent *postfix* expression.
  - Evaluate *postfix* expression
Expression: Introduction

- An expression is made up by operands, operators, delimiters.
- Example: $X = A / B - C + D * E - A * C$
  - Operands: A, B, C, D, E
- In programming languages, operands can be any legal variable names or constant.
- In any expression, the values that variables take must be consistent with the operations performed on them.
Expression: Introduction (cont’d)

- Operations are described by the operators
- In most programming languages, there are several kinds of operators that correspond to the different kinds of data a variable can hold.

**Arithmetic operators**

- +, -, *, /
- Unary minus (-), logical not (!), &, * and aList.insert()
Expression: Introduction (cont’d)

- **Relational operators**
  - <, <=, ==, <>, >=, and >
  - The result of an expression that contains relational operators is either true (1) or false (0)
  - Such an expression is called **boolean** (named after the mathematician George Boole, the father of symbolic logic.)

- **Logical operators**
  - &&, ||, and !.
Expression Evaluation

- Determine the **order** the operations are carried out.

- Every programming language must uniquely define such an order.
  - e.g., multiplication/division have higher order than plus/minus. $4+5*7$
## Priority (precedence order) of Operators in C++

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unary minus, !</td>
<td>Unary operators</td>
</tr>
<tr>
<td>2</td>
<td>*, /, %</td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>3</td>
<td>+, -</td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>4</td>
<td>&lt;, &lt;=, &gt;=, &gt;</td>
<td>Relational operators</td>
</tr>
<tr>
<td>5</td>
<td>==, !=</td>
<td>Relational operators</td>
</tr>
<tr>
<td>6</td>
<td>&amp;&amp;</td>
<td>Logical operators</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Precedence and Associativity

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
</tr>
<tr>
<td>[ ]</td>
<td>right to left</td>
</tr>
<tr>
<td>-&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>.</td>
<td>left to right</td>
</tr>
<tr>
<td>(type)</td>
<td>left to right</td>
</tr>
<tr>
<td>sizeof</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + * &amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt; &lt;= &gt;= == != &amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^ &amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^=</td>
<td>left to right</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Unary +, -, and * have higher precedence than the binary forms.

**Associativity** \( x + (y + z) = (x + y) + z \)
Evaluating Postfix Expressions: Example (1/3)

- $2*(3+4) \rightarrow 2 \ 3 \ 4 \ + \ *$

- In a postfix expression: a binary operator applies to the two operands immediately precede it.
Evaluating Postfix Expressions: Algorithm (2/3)

for (each character ch in the string)
{
    if (ch is an operand)
        push value that operand ch represents onto stack;
    else //ch is an operator named op; evaluate and push the result
    {
        operand2 = top of stack; // get the value of the top item
        pop the stack; //remove the top item from the stack
    }
    operand1 = top of stack;
    pop the stack;
    result = operand1 op operand2;
    push result onto stack;
} //end for
Evaluating Postfix Expressions:

\[
2 \ 3 \ 4 \ + \ * \ (3/3)
\]

<table>
<thead>
<tr>
<th>Key entered</th>
<th>Calculator action</th>
<th>After stack operation: Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>push 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>push 3</td>
<td>2 3</td>
</tr>
<tr>
<td>4</td>
<td>push 4</td>
<td>2 3 4</td>
</tr>
<tr>
<td>+</td>
<td>operand2 = pop stack</td>
<td>(4) 2 3</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack</td>
<td>(3) 2</td>
</tr>
<tr>
<td></td>
<td>result = operand1 + operand2</td>
<td>(7) 2 7</td>
</tr>
<tr>
<td></td>
<td>push result</td>
<td>2 7</td>
</tr>
<tr>
<td>*</td>
<td>operand2 = pop stack</td>
<td>(7) 2</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>result = operand1 * operand2</td>
<td>(14) 14</td>
</tr>
<tr>
<td></td>
<td>push result</td>
<td>14</td>
</tr>
</tbody>
</table>
Converting an infix expression to postfix form: Algorithm (1/4)

Initialize postfixExp to the null string; // output: postfixExp

for (each character ch in the infix expression)
{
    switch (ch)
    {
        case operand: //append operand to end of postfixExp
            postfixExp = postfixExp + ch;
            break;
        case "(": //save '(' on stack
            aStack.push(ch);
            break;
        case ")": //pop stack until matching '(
            while (top of stack is not "(" ) {
                postfixExp = postfixExp + (top of aStack)
                aStack.pop()
            } //end while
            aStack.pop(); //remove "(" parenthesis
            break;
    }
}

2*(3+4) -> 2 3 4 + *
Converting an infix expression to postfix form: Algorithm (2/4)

```
case operator:  //process stack operators of greater precedence
    while(!aStack.isEmpty() and top of stack is not '(' and
          precedence(ch) <= precedence(top of aStack))
        // the operator inside the stack must be performed
        // before the operator outside
        // associativity is left-to-right
        {
            postfixExp = postfixExp + (top of aStack)
            aStack.pop()
        } //end while
    aStack.push(ch);  //save new operator
    break;
} //end for
```

`2*(3+4) -> 2 3 4 + *`
Converting an *infix* expression to *postfix* form: Algorithm (3/4)

// after scan all the characters in the infix expression,
// append to postfixExp the operators remaining in the stack

while (!aStack.isEmpty()) {
    postfixExp = postfixExp + (top of aStack)
    aStack.pop()
} //end while
### Converting an infix expression to postfix form: Example (4/4)

**Input:** infix expression \( a-(b+c*d)/e \)

<table>
<thead>
<tr>
<th>ch</th>
<th>Stack (bottom to top)</th>
<th>postfixExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>(</td>
<td>-()</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>-()</td>
<td>ab</td>
</tr>
<tr>
<td>+</td>
<td>-()</td>
<td>ab</td>
</tr>
<tr>
<td>c</td>
<td>-()</td>
<td>abc</td>
</tr>
<tr>
<td>*</td>
<td>-()</td>
<td>abc</td>
</tr>
<tr>
<td>d</td>
<td>-()</td>
<td>abcd</td>
</tr>
<tr>
<td>)</td>
<td>-()</td>
<td>abcd*</td>
</tr>
<tr>
<td>+</td>
<td>-()</td>
<td>abcd*+</td>
</tr>
<tr>
<td>/</td>
<td>-()</td>
<td>abcd*++</td>
</tr>
<tr>
<td>e</td>
<td>-()</td>
<td>abcd*++</td>
</tr>
</tbody>
</table>

- ‘(‘ before entering the stack, it has the highest priority
- ‘(‘ in the stack has the lowest priority

Move operators from stack to postfixExp until ‘(‘

Copy operators from stack to postfixExp
Application#5: A “Search” Problem

- A type of problem that stacks are very useful – “probe” and “backtrack”

- Compare solutions
  - Using stacks in a non-recursion version of the search algorithm
  - Using stacks in a recursive search algorithm
Non-recursive Solution: Depth First Search

- An exhaustive search
  - Try *every possible sequence* until the target one is found.
  - STACK is useful in organizing this search.
  - Stack helps to keep track of what has been “visited” for the backtracking purpose.

- Approach
  - Select an arbitrary path
  - “Backtrack” – go back to the “most recently” visited node
    - When hit a dead-end, or
    - Go around in cycles
Example Application #5: find a travel itinerary

- Consider an airline company called The High Planes Airline (HPAir).

- It requires a program to process customer requests to fly from some origin city to some destination city
  - i.e. a path finding (search) problem
  - e.g., routing in networks, scheduling in supply-chain management, etc.
Flight Map

- The “Flight map” is represented by a directed graph $DG=(V, DE)$
  - City - vertex (node)
  - Flight Route - directed edge
- Wish to find a directed path from source $P$ to destination $Z$
- Solutions
  - #1: exhaustive search using stack
  - #2: recursion
Exhaustive Search using Stack (1/2)

- **Algorithm** *(depth-first search)*
  - Begin at the origin city
  - At an intermediate city, *try every possible sequence of flights*
  - Until either a path is found or it determines no such sequence exists. *(dead-end, cycle)*
Flight Map 
from p to z

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.
Exhaustive Search using Stack (2/2)

- Backtracking is necessary when
  - the city on the top of the stack has NO flights out
  - Or all of its next hops have been "visited" – cycle.
Exhaustive Search using Stack - code (1/4)

```cpp
bool Map::isPath(int originCity, int destinationCity)
{// Determine whether a sequence of flights between two
cities exists. Nonrecursive stack version.

// Precondition: originCity and destinationCity are the city
// numbers of the origin and destination cities, respectively.

// Postcondition: Returns true if a sequence of flights exists
// from originCity to destinationCity; otherwise returns
// false. Cities visited during the search are marked as
// visited in the flight map. Implementation notes:
// Uses a stack for the city numbers of a potential path,
// Calls unvisitAll, marVisited, and getNextCity.
```
Exhaustive Search using Stack - code (2/4)

Stack aStack;
int topCity, nextCity;
bool success;
unvisitAll();  //clear marks on all cities

aStack.push(originCity);  //push origin city onto aStack
markVisited(originCity);  //mark it visited

aStack.getTop(topCity);
Exhaustive Search using Stack - code (3/4)

```java
while (!aStack.isEmpty() &&
        (topCity != destinationCity)) {
    // 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
    // find an unvisited city adjacent to the city on the top of the stack
    success = getNextCity(topCity, nextCity);
    if (!success)
        aStack.pop(); //no city found; backtrack
    else {
        aStack.push(nextCity);
        markVisited(nextCity);
    } //end if
    aStack.getTop(topCity);
} //end while
```

This code shows how to exhaustively search for a path using a stack. It iterates through the stack while there are cities left to visit and the current city is not the destination. It finds the next city adjacent to the top city and marks it as visited. If it cannot find a valid next city, it backtracks by popping the current city from the stack. The loop invariant ensures that the stack contains a directed path from the origin city to the current city in the stack.
Exhaustive Search using Stack - code (4/4)

```java
if (aStack.isEmpty())
    return false; //no path exists
else
    return true; //path exists
```
```
Example: Fig 6-13

<table>
<thead>
<tr>
<th>Action</th>
<th>Reason</th>
<th>Contents of stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push P</td>
<td>Initialize</td>
<td>P</td>
</tr>
<tr>
<td>Push R</td>
<td>Next unvisited adjacent city</td>
<td>P R</td>
</tr>
<tr>
<td>Push X</td>
<td>Next unvisited adjacent city</td>
<td>P R X</td>
</tr>
<tr>
<td>Pop X</td>
<td>No unvisited adjacent city</td>
<td>P R</td>
</tr>
<tr>
<td>Pop R</td>
<td>No unvisited adjacent city</td>
<td>P</td>
</tr>
<tr>
<td>Push W</td>
<td>Next unvisited adjacent city</td>
<td>P W</td>
</tr>
<tr>
<td>Push S</td>
<td>Next unvisited adjacent city</td>
<td>P W S</td>
</tr>
<tr>
<td>Push T</td>
<td>Next unvisited adjacent city</td>
<td>P W S T</td>
</tr>
<tr>
<td>Pop T</td>
<td>No unvisited adjacent city</td>
<td>P W S</td>
</tr>
<tr>
<td>Pop S</td>
<td>No unvisited adjacent city</td>
<td>P W</td>
</tr>
<tr>
<td>Push Y</td>
<td>Next unvisited adjacent city</td>
<td>P W Y</td>
</tr>
<tr>
<td>Push Z</td>
<td>Next unvisited adjacent city</td>
<td>P W Y Z</td>
</tr>
</tbody>
</table>
A Recursion Solution: code

```cpp
bool Map::isPath(int originCity, int destinationCity) {
    int nextCity;
    bool success, done;
    //mark the current city as visited
    markVisited(originCity);
    //base case: the destination is reached
    if (originCity == destinationCity)
        return true;
    return false;
}
```
A Recursion Solution: code (cont’d)

```c
else  //try a flight to each unvisited city
{ done = false;
  success = getNextCity(originCity, nextCity);
  while (success && !done)
  { done = isPath(nextCity, destinationCity);
    if (!done)
      success = getNextCity(originCity, nextCity);
  } //end while
return done;
} //end if
```
The End 😊
Homework 2008

- Pages 334-337: #2, #13 (b, d, f), #14 (a, c, e) and #17.
Homework 2006

- Pages 334-337: #2, #13 (b, d, f), #14 (a, c, e) and #17.