DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

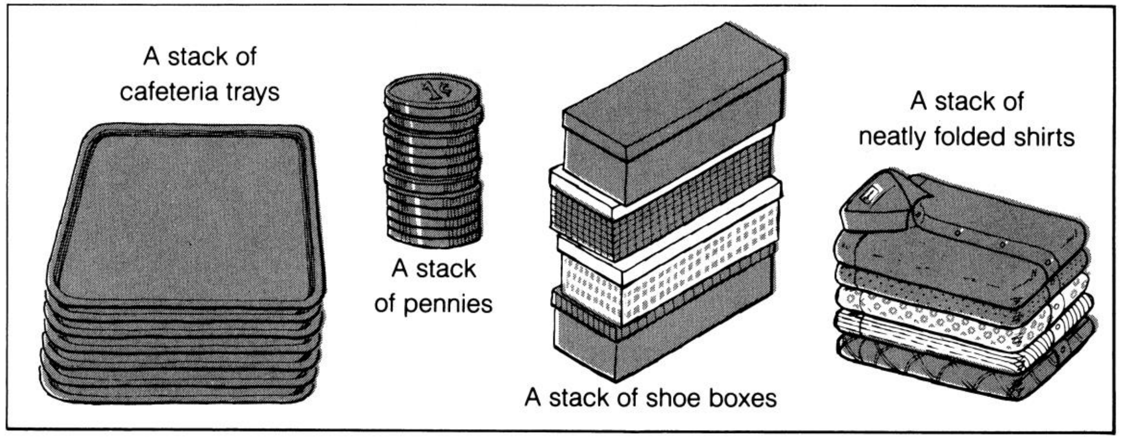
BS (AI)

What is a stack?

It is an ordered group of homogeneous items of elements.

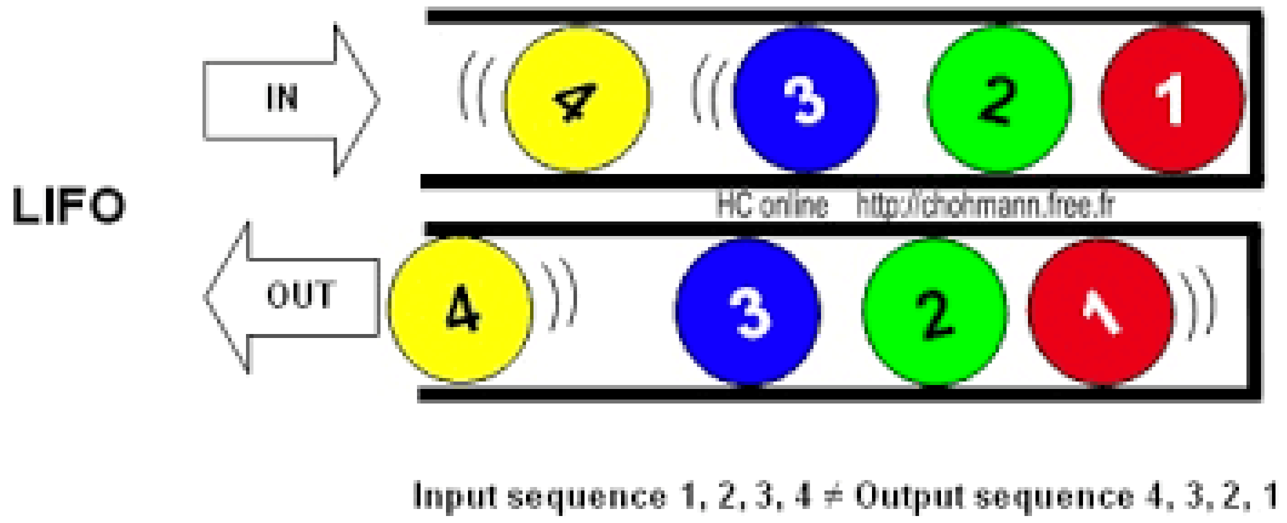
Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).

The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Overview

Last-In-First-Out Data Structure


LIFO
IN
OUT
4
3
2
1
HC online http://chohmann.free.fr
Input sequence 1, 2, 3, 4 ≠ Output sequence 4, 3, 2, 1

Implementations of the ADT Stack

The ADT stack can be implemented using

- An array
- A linked list
- The ADT list

Stack Specification

Definitions: (provided by the user)

- *MAX_ITEMS*: Max number of items that might be on the stack
- *ItemType*: Data type of the items on the stack

Operations

- MakeEmpty
- Boolean IsEmpty
- Boolean IsFull
- Push (ItemType newItem)
- Pop (ItemType& item)

Push (ItemType newItem)

Function: Adds newItem to the top of the stack.

Preconditions: Stack has been initialized and is not full.

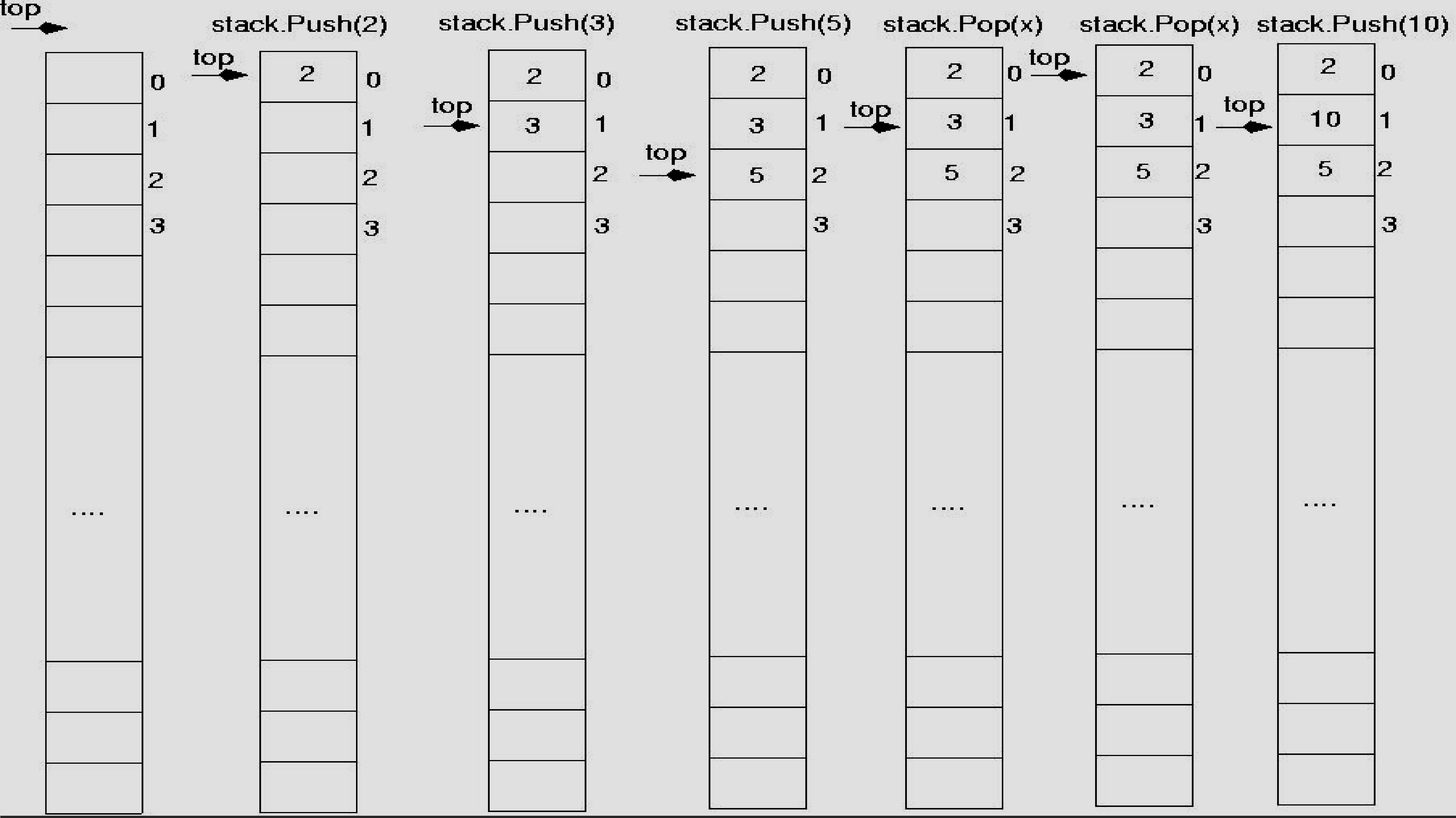
Postconditions: newItem is at the top of the stack.

Pop (ItemType& item)

Function: Removes topItem from stack and returns it in item.

Preconditions: Stack has been initialized and is not empty.

Postconditions: Top element has been removed from stack and item is a copy of the removed element.


top
stack.Push(2)
stack.Push(3)
stack.Push(5)
stack.Pop(x)
stack.Pop(x)
stack.Push(10)
0
1
2
3
2
3
5
10
....

Stack Implementation

```
#include "ItemType.h"
// Must be provided by the user of the class
// Contains definitions for MAX_ITEMS and ItemType
 class StackType {
 public:
        StackType();
        void MakeEmpty();
        bool IsEmpty() const;
        bool IsFull() const;
        void Push(ItemType);
        void Pop(ItemType&);
private:
        int top;
        ItemType items[MAX_ITEMS];
};
```

Stack Implementation (cont.)

```
StackType::StackType()
{
 top = -1;
}
```

```
void StackType::MakeEmpty()
{
 top = -1;
}
```

Stack Implementation (cont.)

```
bool StackType::IsEmpty() const
{
 return (top == -1);
}

bool StackType::IsFull() const
{
 return (top == MAX_ITEMS-1);
}
```

Stack Implementation (cont.)

```
 void StackType::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}
 void StackType::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Stack overflow

The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
   stack.Push(item);
```

Stack underflow

The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
   stack.Pop(item);
```

Implementing stacks using templates

Templates allow the compiler to generate multiple versions of a class type or a function by allowing parameterized types.

Implementing stacks using templates

(cont.)

```
template<class ItemType>
class StackType {
 public:
    StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

Example using templates

```
// Client code
StackType<int> myStack;
StackType<float> yourStack;
StackType<StrType> anotherStack;

myStack.Push(35);
yourStack.Push(584.39);
```

The compiler generates distinct class types and gives its own internal name to each of the types.

Implementing stacks using dynamic array allocation

```
template<class ItemType>
class StackType {
 public:
    StackType(int);
    ~StackType();
  void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
```

Implementing stacks using dynamic array allocation (cont.)

```
template<class ItemType>
StackType<ItemType>::StackType(int max)
{
 maxStack = max;
 top = -1;
 items = new ItemType[max];
}

template<class ItemType>
StackType<ItemType>::~StackType()
{
 delete [ ] items;
}
```

Example: postfix expressions

Postfix notation is another way of writing arithmetic expressions.

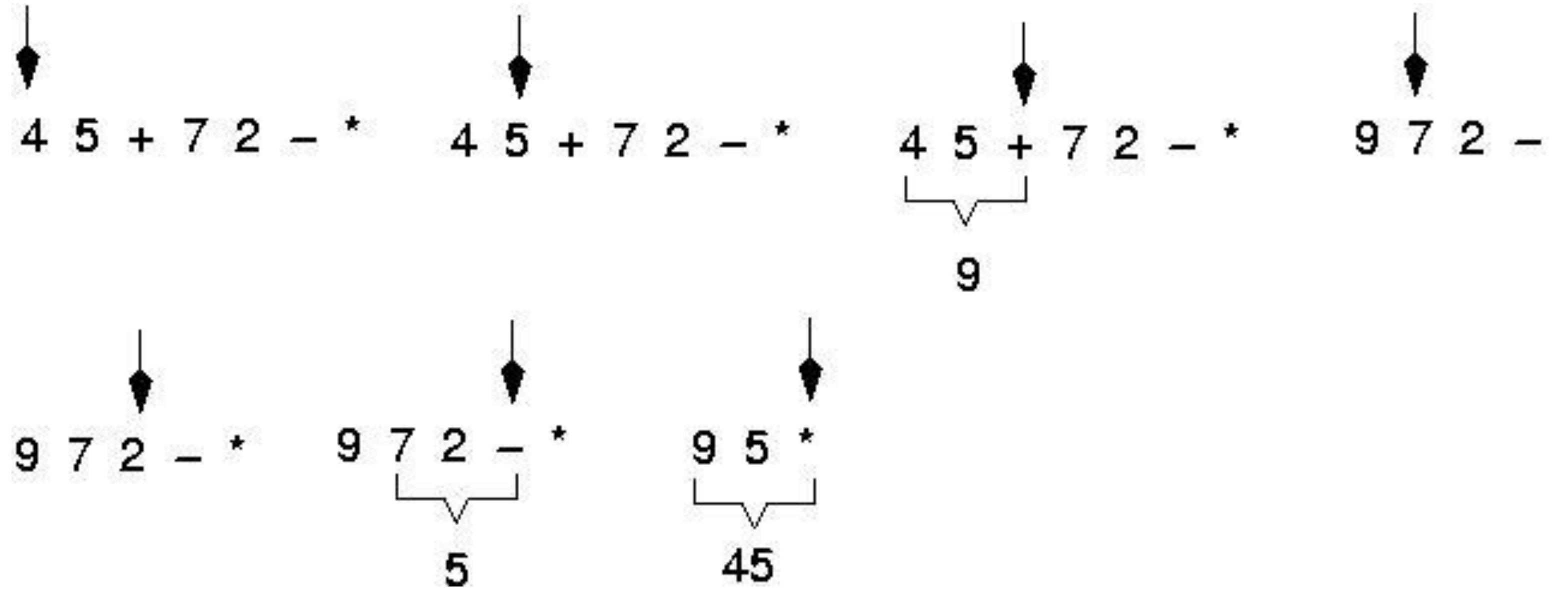
In postfix notation, the operator is written after the two operands.

infix: 2+5   *postfix*: 2 5 +

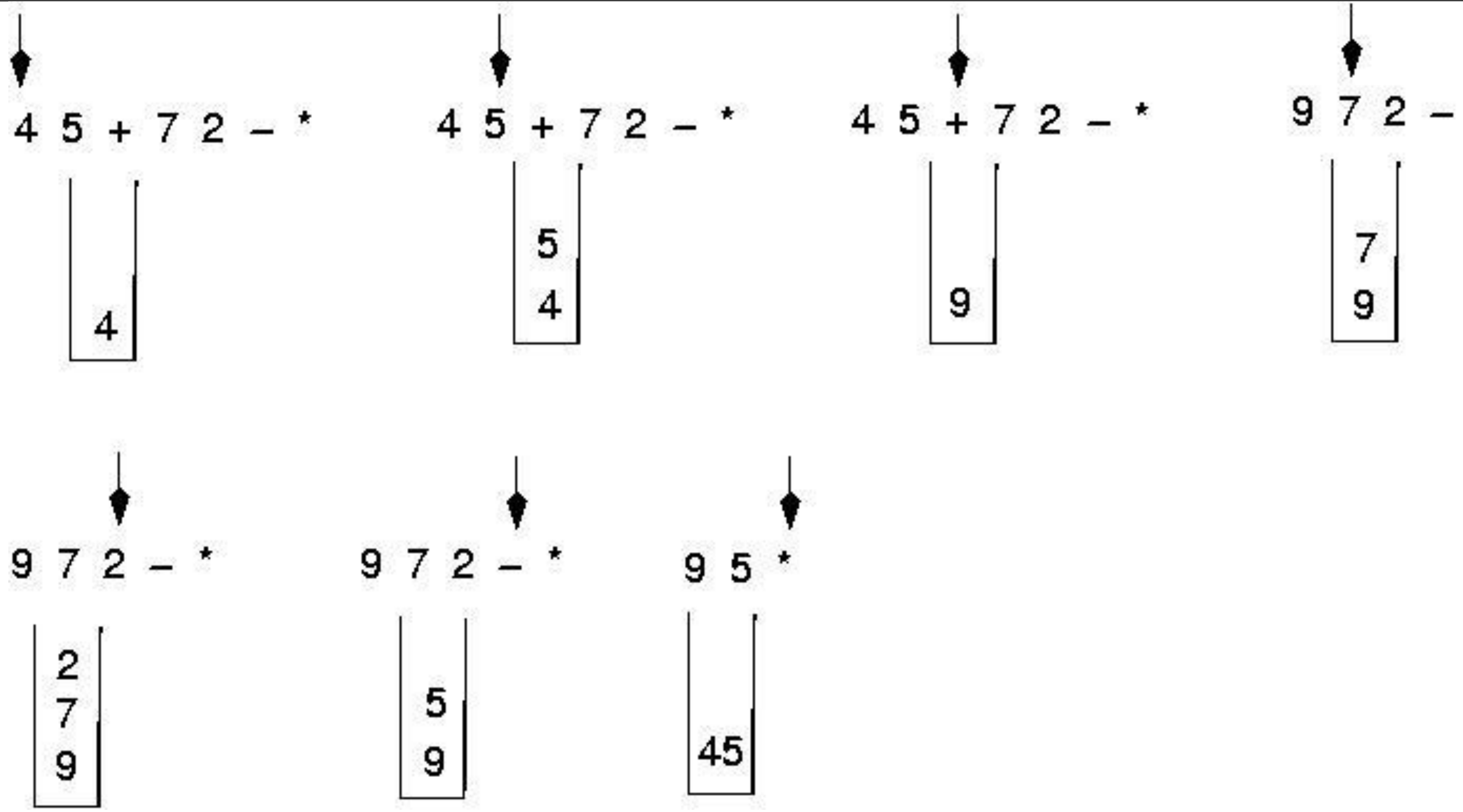
Expressions are evaluated from left to right.

Precedence rules and parentheses are never needed!!

Example: postfix expressions (cont.)



Postfix expressions: Algorithm using stacks (cont.)



Postfix expressions: Algorithm using stacks

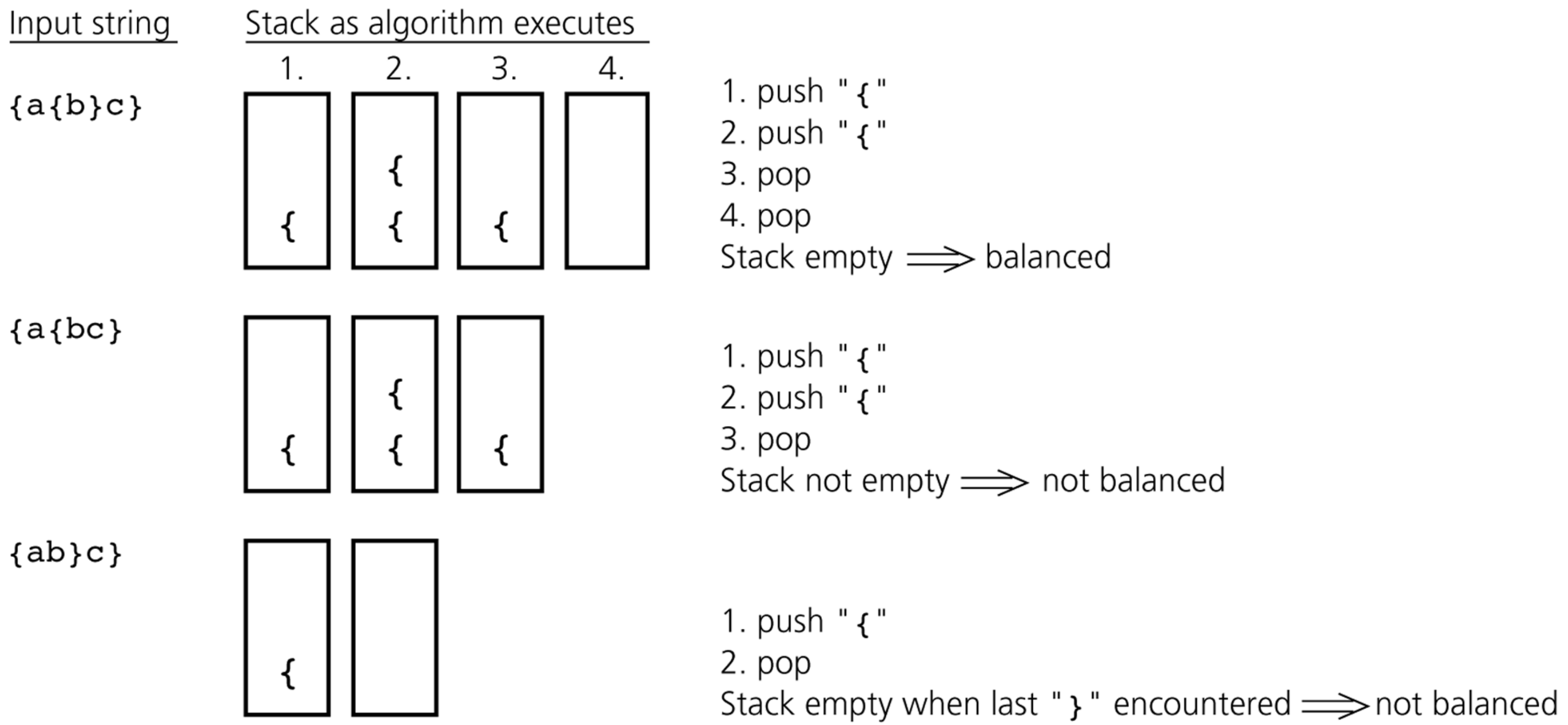
```
WHILE more input items exist
        Get an item
        IF item is an operand
    stack.Push(item)
 ELSE
    stack.Pop(operand2)
    stack.Pop(operand1)
    Compute result
    stack.Push(result)
stack.Pop(result)
```

Checking for Balanced Braces

Requirements for balanced braces

- Each time you encounter a “}”, it matches an already encountered “{”
- When you reach the end of the string, you have matched each “{”

Checking for Balanced Braces



1. push "{"
2. push "{"
3. pop
4. pop
Stack empty $\Longrightarrow$ balanced

1. push "{"
2. push "{"
3. pop
Stack not empty $\Longrightarrow$ not balanced

1. push "{"
2. pop
Stack empty when last "}" encountered $\Longrightarrow$ not balanced

Traces of the algorithm that checks for balanced braces

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

To evaluate an infix expression

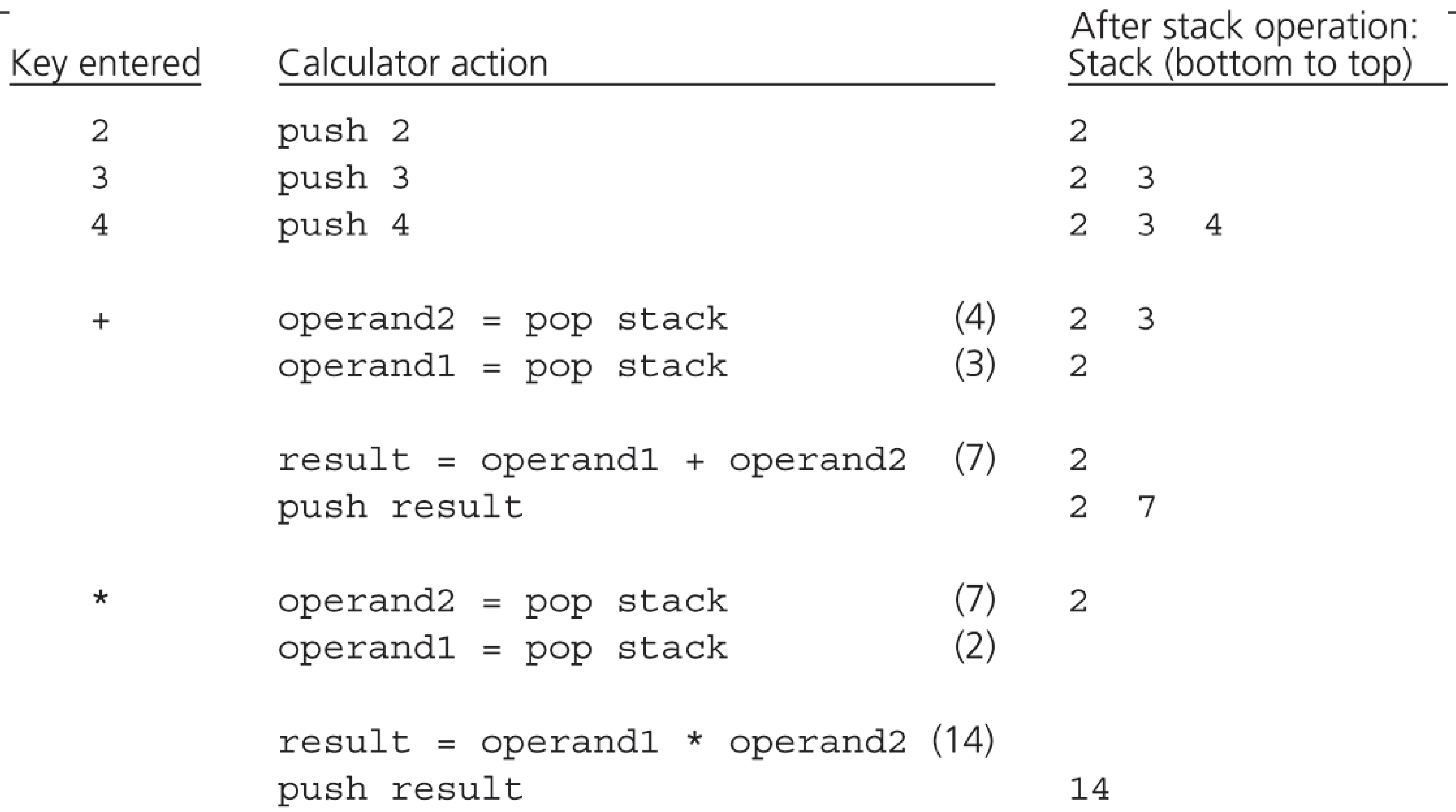
- Convert the infix expression to postfix form
- Evaluate the postfix expression

Evaluating Postfix Expressions

A postfix calculator

- 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

Evaluating Postfix Expressions



Key entered	Calculator action		After stack operation: Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

Evaluating Postfix Expressions

To evaluate a postfix expression which is entered as a string of characters

- Simplifying assumptions
  - The string is a syntactically correct postfix expression
  - No unary operators are present
  - No exponentiation operators are present
  - Operands are single lowercase letters that represent integer values

Converting Infix Expressions to Equivalent Postfix Expressions

An infix expression can be evaluated by first being converted into an equivalent postfix expression

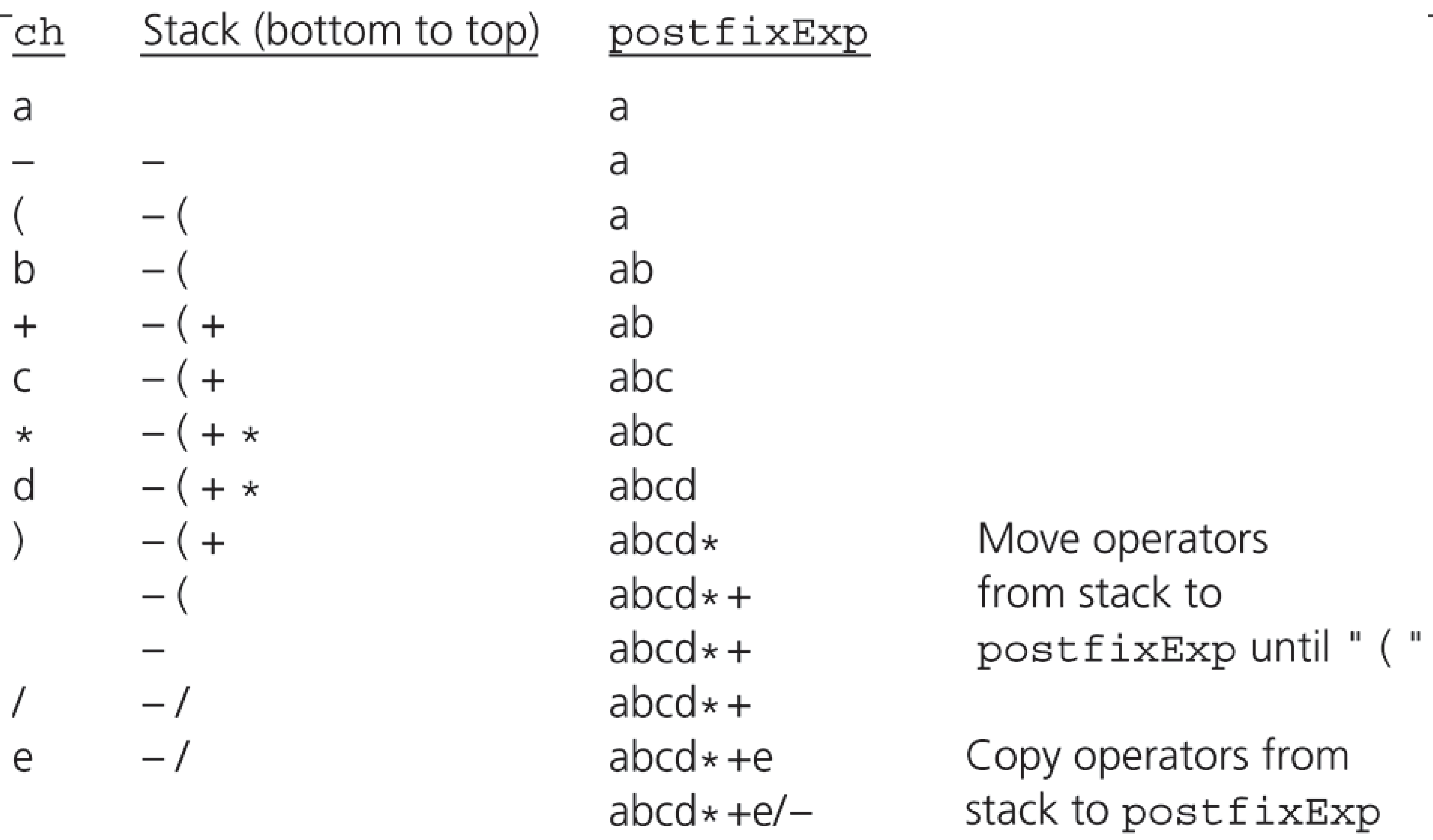
Facts about converting from infix to postfix

- 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

Algorithm to Convert Infix to Postfix Expression Using Stack

1. Initialize the Stack.
2. Scan the operator from left to right in the infix expression.
3. If the leftmost character is an operand, set it as the current output to the Postfix string.
4. And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
5. If the scanned operator has higher precedence than the existing **precedence** operator in the Stack or if the Stack is empty, put it on the Stack.
6. If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
7. If the scanned character is a left bracket '(', push it into the Stack.
8. If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
9. Repeat all steps from 2 to 8 until the infix expression is scanned.
10. Print the Stack output.
11. Pop and output all characters, including the operator, from the Stack until it is not empty.

Converting Infix Expressions to Equivalent Postfix Expressions



ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

A trace of the algorithm that converts the infix expression *a* - (*b* + *c* * *d*)/*e* to postfix form

Application: A Search Problem

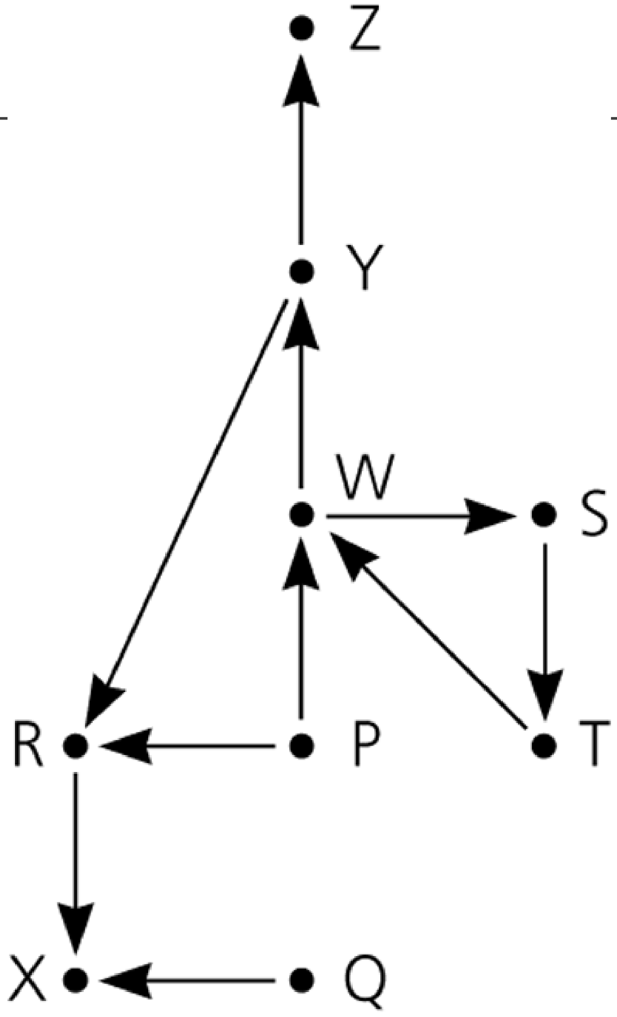
High Planes Airline Company (HPAir)

- For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

The flight map for HPAir is a graph

- Adjacent vertices are two vertices that are joined by an edge
- A directed path is a sequence of directed edges

Application: A Search Problem



Flight map for HPAir

A Nonrecursive Solution That Uses a Stack

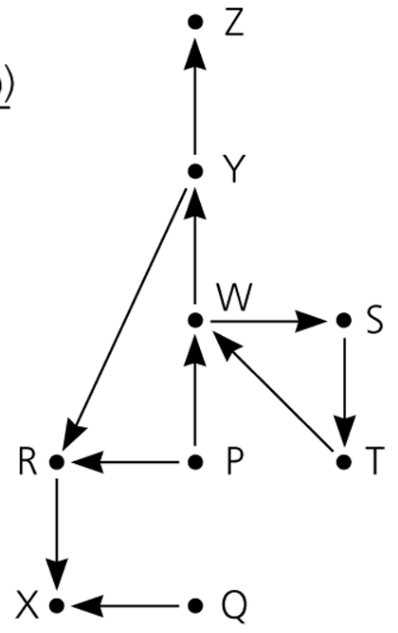
The solution performs an exhaustive search

- 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

Backtracking can be used to recover from a wrong choice of a city

A Nonrecursive Solution That Uses a Stack

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



A trace of the search algorithm, given the flight map

Credits and Acknowledgements

Lectures by Prof. Yung Yi, KAIST, South Korea.

Lectures by **Selim Aksoy**, Bilkent University, Ankara, Turkey

Lecture slides by Dept of Computer Science, Boston University