Chapter 7: Stacks
1. The abstract data type stack
2. Simple applications of the ADT stack
3. Implementation of the ADT stack
4. Application: algebraic expressions/ a search problem
5. The relationship between stacks and recursion
6. Summary
The abstract data type stack
The abstract data type stack

- Applications in real life

```
abcc ← ddde ← ef ← fg
```

Output: abcdefg
The abstract data type stack

- Last-in, first-out (LIFO) ADT stack
  The last item placed on the stack will be the first item removed

- 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

All operations are only allowed on the top of the stack
Bonus question

1. Step 1:
   - S.pop();
   - S.push(c);
   - S.push(b);

2. Step 2:
   - S.pop();
   - S.peek();
   - S.push(a);

3. Step 3:
   - S.peek();
   - S.push(b);
   - S.pop();
   - S.pop();
   - S.push(c);
Simple applications of the ADT stack
Simple applications of the ADT stack

Checking the 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

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 “{”

The exception StackException

- Take precautions to avoid an exception
- Provide try and catch blocks to handle a possible exception
Simple applications of the ADT stack

Checking the balanced braces

```java
aStack.createStack();
balancedSoFar = true;
i=0;
While (balancedSoFar and i<length of aString){
    ch=character at position i in aString;
    ++i;
    If(ch is '{')// push an open brace
    {
        aStack.push('{');
    }else if (ch is '}')//close brace
    {
        if(!aStack.isEmpty())
        {
            openBrace=aStack.pop(); //pop a matching open brace
        }
        else{
            balancedSoFar=false;
        }
    }
}
```
## Simple applications of the ADT stack

### Checking the balanced braces

<table>
<thead>
<tr>
<th>Input string</th>
<th>Stack as algorithm executes</th>
<th>Procedure</th>
</tr>
</thead>
</table>
| `{a{b}c}`    | { { } }                      | 1. push "{"
              | 2. push "{"                | 2. push "{"
              | 3. pop                     | 3. pop |
              | 4. pop                     | 4. pop |
              |                             | Stack empty $\Rightarrow$ balanced |
| `{a{bc}`     | { { } }                      | 1. push "{"
              | 2. push "{"                | 2. push "{"
              | 3. pop                     | 3. pop |
              |                             | Stack not empty $\Rightarrow$ not balanced |
| `{ab}c`      | { }                         | 1. push "{"
              | 2. pop                      | 2. pop |
              |                             | Stack empty when last "}" encountered $\Rightarrow$ not balanced |
Recognizing strings in a Language

Language L

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

e.g. $abba$, $aaccctaa$, $acbca$

A stack can be used to determine whether a given string is in L

- Traverse the first half of the string, pushing each character onto a stack
- Once you reach the $, for each character in the second half of the string, pop a character off the stack, match the popped character with the current character in the string

This string is in L
Simple applications of the ADT stack

- Recognizing strings in a Language
  - **Bonus question**
    - L = \{w$w: w is a possible empty string of characters other than \$\}
    - e.g. ab$ab, aacc$aacc, aacc$ccaa
      - Can we use stack to verify the string by scanning the string only once? Why?
    - L = \{strings contains one more a then b\}
      - e.g. aaabb, ababa, a
        - How can we check if a string is in L or not?
        - Scanning the string only once?
        - The terminate conditions?

Order matters or not
Implementations of the ADT Stack
Implementations of the ADT stack

- **Implementations**
  - Array-based
  - Reference-based
  - ADT list-based

- **StackInterface**
  - Provides a common specification for the three implementations

- **StackException**
  - Used by StackInterface
  - Extends java.lang.RuntimeException

<table>
<thead>
<tr>
<th>StackInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>popAll()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>push()</td>
</tr>
<tr>
<td>pop()</td>
</tr>
<tr>
<td>peek()</td>
</tr>
</tbody>
</table>
Implementations of the ADT stack

- **Array-based implementation of the ADT stack**
  
  - **StackArrayBased class**
    - Implements StackInterface
    - Instances Stacks
    - **Private data fields**
      - An array of Objects called items
      - The index top
    - **Constructor replaces the ADT operation createStack**

  ![Diagram of stack array-based implementation](image)
Implementations of the ADT stack

- A reference-based implementation of the ADT stack
  - Required when the stack needs to grow and shrink dynamically
  - StackReferenceBased
    - Implements StackInterface
    - top is a reference to the head of a linked list of items
Implementations of the ADT stack

➢ The ADT list can be used to represent the items in a stack. If the item in position 1 of a list represents the top of the stack:

- **push(newItem) operation** is implemented as:
  - `add(0, newItem)`

- **pop() operation** is implemented as:
  - `get(0)`
  - `remove(0)`

- **peek() operation** is implemented as:
  - `get(0)`

<table>
<thead>
<tr>
<th>List</th>
<th>createList()</th>
<th>destroyList()</th>
<th>isEmpty()</th>
<th>getLength()</th>
<th>insert()</th>
<th>remove()</th>
<th>Retrieve()</th>
</tr>
</thead>
</table>

List position

```
1 | 10
2 | 80
3 | 60
...
list.size() | 5
```
JCF contains an implementation of a stack class called Stack (generic)

- `java.util.Stack`
- Derived from `Vector`
- Includes methods: `empty`, `peek`, `pop`, `push`, and `search`
- `search` returns the 1-based position of an object on the stack
Application: Algebraic Expressions/ A Search Problem
Algebraic Expressions

- **Algebraic expression**
  - Infix expression: every binary operator appears between its operands
    - a+b
    - a+b*c (ambiguous)
  - Prefix expression: an operator precedes its operands
    - +ab
    - +a*bc
  - Postfix expression: an operator follows its operands
    - ab+
    - abc**+
Algebraic Expressions

Converting Infix Expressions to Equivalent Postfix form

Initialize postfixExp to the null string

for (each character ch in the infix expression) {
    switch (ch) {
        case ch is an operand:
            append ch to the end of postfixExp break;
        case ch is '(':
            aStack.push(ch) break;
        case ch is ')':
            While (top of stack is bot '('){
                postfixExp = postfixExp + aStack.pop();
            }
            openParen = aStack.pop(); //remove the open parenthesis
            break;
        case ch is an operator:
            while (!aStack.isEmpty() and top of stack is not '(' and precedence(ch) <= precedence(top of stack)) {
                postfixExp = postfixExp + aStack.pop();
            }
            aStack.push(ch) //save new operator break;
    }
}
while (!aStack.isEmpty()) {
    postfixExp = postfixExp + aStack.pop();
}
Algebraic Expressions

Converting Infix Expressions to Equivalent Postfix Expressions \( a-(b+c*d)/e \)

<table>
<thead>
<tr>
<th>ch</th>
<th>stack (bottom to top)</th>
<th>postfixExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td></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>a</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></td>
</tr>
<tr>
<td>/</td>
<td>−/</td>
<td>abcd*+e</td>
</tr>
<tr>
<td>e</td>
<td>−/</td>
<td>abcd*+e/−</td>
</tr>
</tbody>
</table>

Move operators from stack to postfixExp until " ( "

Copy operators from stack to postfixExp
Algebraic Expressions

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
Algebraic Expressions

evaluate an infix expressions
- 2*(3+4) hard!
- Convert the infix expression to postfix form
- Evaluate the postfix expression 2,3,4,+,*

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
## Algebraic Expressions

<table>
<thead>
<tr>
<th>Key entered</th>
<th>Calculator action</th>
<th>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 (4)</td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack (3)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>result = operand1 + operand2 (7)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>push result</td>
<td>2 7</td>
</tr>
<tr>
<td>*</td>
<td>operand2 = pop stack (7)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>result = operand1 * operand2 (14)</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>push result</td>
<td>14</td>
</tr>
</tbody>
</table>
Bonus question

1. $12 \times 34 + -$ 
2. $1234 + -*$
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

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

- 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 choice of a city
Stack-based solution

- Stack-based solution

(a) P
(b) R P
(c) X R P
(d) R P
(e) P
(f) W P

Diagram:

- Z
- Y
- W
- S
- R
- P
- T
- X
- Q
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)
  }
 }

Recursive-based 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
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.
Summary
ADT stack operations have a last-in, first-out (LIFO) behavior

Algorithms that operate on algebraic expressions are an important application of stacks

A stack can be used to determine whether a sequence of flights exists between two cities

A strong relationship exists between recursion and stacks
Algebraic Expressions

- Bonus question (infix expression to postfix form)
  1. $a + (b - c)$
  2. $a * (b + c - d)$