Part 2. Stacks

Instructor: Sangmi Pallickara (sangmi@cs.colostate.edu)
Department of Computer Science
Colorado State University

Outline

• Stacks?
• Applications using Stacks
• Implementing Stacks
• Relationship between Stacks and Recursive Programming

Linear, time ordered structures

• Data Structures that reflects a temporal relationship
  – Order of removal based on the order of insertion
• We will consider
  – “First come, first serve”
    • First Come, First Out: FIFO (queue)
  – “take from the top of the pile”
    • Last In, First Out: LIFO (stack)

What can we do with Coin dispenser?

• “push” a coin into the dispenser.
• “pop” a coin from the dispenser.
• “see” the coin on top.
• “check” whether this dispenser is empty or not.

Stacks

• Last In First Out (LIFO) structure
• Add/Remove from the same end
**Stack Operations**

- **isEmpty():** determine whether stack is empty
- **push():** add a new item to the stack
- **pop():** remove the item added most recently
- **peek():** retrieve the item added most recently
- **createStack():** create an empty stack
- **removeAll():** remove all the items

**Outline**

- Stacks?
- Applications using Stacks
- Implementing Stacks
- Relationship between Stacks and Recursive Programming

**Applications – Call Stack**

- Stack data structure to store information about active subroutines or computer program.
- Memory allocation for tracking the execution of functions such as a nested function.
  - First method that returns is the last one invoked
- Element of call stack – activation record
  - Parameters
  - Local variables
  - Return address: pointer to the next instruction to be executed in calling method

**Applications – Backtracking**

- Looking for a solution based on a guess.
- Keeping the guesses (temporary answers) in the stack.
- When the process faces deadend, go back one step (visit the first one in the stack) and repeat the process.
Checking for Balanced Braces

1. Each time you encounter a ")", it matches an already encountered "(".
2. When you reach the end of the string, you have matched all of the "{" you have encountered.

Peudocode

```
while ( not at the end of the string){
    if (the next character is a "{"{
        aStack.push("{"{
    }else if (the character is a "}"") {
        openBrace = aStack.pop()
    }
}
```

Algebraic Expressions

• Evaluating Postfix Expressions
  – Assumptions:
    • String is a syntactically correct postfix expression
    • No unary operator
    • No exponentiation operators
    • Operands are single lowercase letters
Peudocode

```
for (each character ch in the string){
    if(ch is an operand){
        push value that operand ch represents onto stack
    } else{
        operand2 = Pop the top of the stack
        operand1 = Pop the top of the stack
        result = operand1 op operand2
        push result onto stack
    }
}
```

Outline

• Stacks?
• Applications using Stacks
• Implementing Stacks
• Relationship between Stacks and Recursive Programming

String

```
  2 3 4+3*
```

```
2
3
4
+ 3*4
```

WHY?

```
2*(3+4)
```

Implementing Stacks

• Should perform the **stack operations**

• **Array-based** implementation
• **Reference-based** implementation
• **List-based** implementation

Array-Based Implementation

• Use an array of Objects (called items)

```
public class StackArrayBased implements StackInterface{
    final int MAX_STACK = 50;
    private Object items[];
    private int top;
    public StackArrayBased(){
        items = new Object[MAX_STACK];
        top = -1;
    } // end default constructor
```

isEmpty()

```
public boolean isEmpty(){
    return top < 0;
}
```

push()

```
public void push (Object newItem) throws StackException{
    if (isFull()) {
        item[++top] = newItem;
    } else{
        throw new StackException(your_error_message);
    }
}
```
pop()

```java
public Object pop() throws StackException{
    if (!isEmpty){
        return items[top--];
    } else {
        throw new StackException(your_error_message);
    }
}
```

peek()

```java
public Object peek() throws StackException{
    if(!isEmpty){
        return items[top];
    }else {
        throw new StackException(your_error_message);
    }
}
```

Reference-Based Implementation

```
private Node top;
public void push (Object newItem){
    top = new Node(newItem,top)
}

public Object pop() throws StackException{
    if (!isEmpty){
        Node temp = top;
        top = top.next;
        return temp.item;
    } else { ... exception handling }
}
```

Implementation that uses List

```
public void push(Object newItem){
    list.add(0, newItem);
}

public Object pop() throws StackException{
    if (!list.isEmpty){
        Object temp = list.get(0);
        list.remove(0);
        return temp;
    } else { ... exception handling }
}

public void popAll(){
    list.removeAll();
}
```
Comparison of Implementation

- What are the pros and cons for different implementation styles?
  - Array based?
  - Reference based?
  - List based?

  -- Array based implementation can provide fixed size stack
  -- List based: Why not just List?

How about java.util.Stack?

- Derived from Vector.
- empty(), peek(), pop(), push(), search(Object o)

Relationship between Stacks and Recursion

- Most implementations of recursion can maintain a stack of activation records.
- Backtracking example
- Within recursive calls, the most recently executed call is stored at the top of the stack.
- Store and access the same point of the data structure.

Next Reading

- Chap. 8 Queues