CS 171: Introduction to Computer Science II

Stacks and Queues

Li Xiong
Announcements/Reminders

• Last day to turn in Hw1 with 2 late credits
• Hw2 due next Monday
• Hw3 to be assigned next Tuesday
• Midterm 3/29
Today

• Stacks
  – Operations
  – Implementation using resizable array
  – Implementation using generics

• Applications using stacks

• Queues
  – Operations
  – Implementation
  – Applications
Stacks and queues

Fundamental data types.
- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

**Stack.** Examine the item most recently added. LIFO = "last in first out"

**Queue.** Examine the item least recently added. FIFO = "first in first out"
Stacks

• A stack stores an array of elements but with only two main operations:
  
  **Push**: add an element to the top of the stack
  
  **Pop**: remove the top element of the stack.

• Pop always removes the last element that’s added to the stack. This is called **LIFO** (Last-In-First-Out).
Stack API

Warmup API. Stack of strings data type.

```
public class StackOfStrings

    StackOfStrings()              // create an empty stack
    void push(String s)           // insert a new item onto stack
    String pop()                  // remove and return the item most recently added
    boolean isEmpty()             // is the stack empty?
    int size()                    // number of items on the stack
```

Warmup client. Reverse sequence of strings from standard input.
Stack: resizing-array implementation

Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

```java
public ResizingArrayStackOfStrings()
{
    s = new String[1];
}

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

Consequence. Inserting first $N$ items takes time proportional to $N$ (not $N^2$).
Generic stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public StackOfStrings(int capacity) {
        s = new String[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(String item) {
        s[N++] = item;
    }

    public String pop() {
        return s[--N];
    }
}
```

The way it should be

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = new Item[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```

@#$%^ generic array creation not allowed in Java
Generic stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(String item) {
        s[N++] = item;
    }

    public String pop() {
        return s[--N];
    }
}
```

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = (Item[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```

the ugly cast

the way it is
Stack: Implementations

• Stack of strings using fixed-capacity array: 
  *FixedCapacityStackOfStrings.java*

• Generic stack using fixed-capacity array: 
  *FixedCapacityStack.java*

• Generic stack using a resizing array: 
  *ResizingArrayStack.java*

• Generic stack using a linked list (next lecture): 
  *Stack.java*
Stack: Applications

- Application 1: Reverse a list of integers
- Application 2: Delimiter matching
- Application 3: Expression evaluation
- Other applications
  - Undo/redo history
  - Browsing history (back button in browser)
  - Call stack
Application 1: Reverse a list of integers

• Reads a sequence of integers, prints them in reverse order
Application 1: Reverse a list of integers

- Reads a sequence of integers, prints them in reverse order
  - Push the integers to a stack one by one
  - pop and print them one by one

Reverse.java
Application 2 – Delimiter Matching

• You want to make sure if the parentheses in an mathematical expression is balanced: 
  \[(w \ast (x + y) / z - (p / (r - q)))\]

• It may have several different types of delimiters: braces\{\}, brackets[], parentheses().
  – Each opening on the left delimiter must be matched by a closing (right) delimiter.
  – Left delimiters that occur later should be closed before those occurring earlier.
Application 2 – Delimiter Matching

• Examples:

  c[d]
  a{b[c]d}e
  a{b(c)d}e
  a[b{c}d]e}
  a{b(c)
Application 2 – Delimiter Matching

• Examples:

```
c[d]       // correct
a{b[c]d}e  // correct
a{b(c)d}e  // not correct; ] doesn't match ( 
a[b{c}d]e} // not correct; nothing matches final }
a{b(c)     // not correct; nothing matches opening {
```
Application 2 – Delimiter Matching

• It’s easy to achieve matching using a stack:
  – Read characters from the string.
  – Whenever you see a left (opening) delimiter, push it to the stack.
  – Whenever you see a right (closing) delimiter, pops the opening delimiter from the stack and match.
  – If they don’t match, report error.
Application 2 – Delimiter Matching

• It’s easy to achieve matching using a stack:
  – Read characters from the string.
  – Whenever you see a left (opening) delimiter, push it to the stack.
  – Whenever you see a right (closing) delimiter, pops the opening delimiter from the stack and match.
  – If they don’t match, report error.
  – What happens if the stack is **empty** when you try to match a closing delimiter?
  – What happens if the stack is **non-empty** after all characters are read?
Application 2 – Delimiter Matching

• Example:
  a{b(c[d]e)f}

• Code: ~cs171000/share/code/Brackets/brackets.java

• Why does this work?
  – Delimiters that are opened last must be closed first.
  – This conforms exactly with the LIFO property of the stack.
public void check()
{
    int stackSize = input.length();  // get max stack size
    StackX theStack = new StackX(stackSize);  // make stack
    for(int j=0; j<input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j);  // get char
        switch(ch)
        {
            case '{':          // opening symbols
case '[':
case '(':  
                theStack.push(ch);  // push them
                break;
            case '}':  // closing symbols
case ']':
case ')':
                if( !theStack.isEmpty() )  // if stack not empty,
                {
                    char chx = theStack.pop();  // pop and check
                    if( (ch=='}' && chx!='{') ||
                        (ch==']' && chx!='[') ||
                        (ch==')' && chx!='(') )
                        System.out.println("Error: "+ch+" at "+j);
                }
                else  // prematurely empty
                        System.out.println("Error: "+ch+" at "+j);
                break;
            default:    // no action on other characters
                break;
        }  // end switch
    }  // end for
    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
}  // end check()
public void check()
{
    int stackSize = input.length();       // get max stack size
    StackX theStack = new StackX(stackSize); // make stack

    for(int j=0; j<input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j);       // get char
        switch(ch)
        {
            case '{':                      // opening symbols
            case '[': case '(':        // opening symbols
            case '(':                      // opening symbols
            case '(':                      // opening symbols
                theStack.push(ch);          // push them
                break;
            case '}':                      // closing symbols
            case ']': case ')':        // closing symbols
                if( !theStack.isEmpty() )   // if stack not empty,
                {
                    char chx = theStack.pop();  // pop and check
                    if( (ch=='}' && chx!='{') ||
                        (ch==']' && chx!='[') ||
                        (ch==')' && chx!='(') )   // if stack not empty,
                        System.out.println("Error: "+ch+" at "+j);
                }
                else                        // prematurely empty
                    System.out.println("Error: "+ch+" at "+j);
                break;
            default:    // no action on other characters
                break;
        }  // end switch
    }  // end for
    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
}  // end check()
public void check()
{
    int stackSize = input.length();  // get max stack size
    StackX theStack = new StackX(stackSize);  // make stack

    for(int j=0; j<input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j);  // get char
        switch(ch)
        {
            case '{':  // opening symbols
                case '[':
                case '(':
                    theStack.push(ch);  // push them
                    break;
            case '}':  // closing symbols
                case ']':
                case ')':
                    if( !theStack.isEmpty() )  // if stack not empty,
                    {
                        char chx = theStack.pop();  // pop and check
                        if( (ch=='}' && chx!='{') ||
                            (ch==']' && chx!='[') ||
                            (ch==')' && chx!=')' )
                            System.out.println("Error: "+ch+" at "+j);
                    }
                    else  // prematurely empty
                        System.out.println("Error: "+ch+" at "+j);
                    break;
            default:  // no action on other characters
                break;  // end switch
        }  // end switch
    }  // end for

    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
}  // end check()
public void check()
{
    int stackSize = input.length(); // get max stack size
    StackX theStack = new StackX(stackSize); // make stack

    for(int j=0; j<input.length(); j++) // get chars in turn
    {
        char ch = input.charAt(j); // get char
        switch(ch)
        {
            case '{': // opening symbols
            case '[':
            case '(': // opening symbols
                theStack.push(ch); // push them
                break;

            case '}': // closing symbols
            case ']':
            case ')': // closing symbols
                if( !theStack.isEmpty() ) // if stack not empty,
                {
                    char chx = theStack.pop(); // pop and check
                    if( (ch=='}' && chx!='{') ||
                        (ch==']' && chx!='[') ||
                        (ch==')' && chx!='(') )
                        System.out.println("Error: \"+ch\" at \"+j\";
                    }
                    else // prematurely empty
                        System.out.println("Error: \"+ch\" at \"+j\",
                break;
            default: // no action on other characters
                break; // no action on other characters
        } // end switch
    } // end for

    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
} // end check()
Application 3 – Arithmetic Expression Evaluation

- Task: evaluate arithmetic expressions.
- Familiar arithmetic expressions:
  
  2+3
  
  2*(3+4)
  
  ...

- The **operators** are placed between two **operands**. This is called **infix** notation.
Arithmetic expression evaluation

Goal. Evaluate infix expressions.

\[(1 + ((2 + 3) \times (4 \times 5)))\]
Arithmetic expression evaluation

**Goal.** Evaluate infix expressions.

\[(1 + ((2 + 3) \times (4 \times 5)))\]

**Two-stack algorithm.** [E. W. Dijkstra]

- **Value:** push onto the value stack.
- **Operator:** push onto the operator stack.
- **Left parenthesis:** ignore.
- **Right parenthesis:** pop operator and two values; push the result of applying that operator to those values onto the operand stack.
Application 3 – Arithmetic Expression Evaluation

• Code
  Evaluate.java
• Demo
Postfix (RPN) Notation

• For computers to parse the expressions, it’s more convenient to represent expressions in **postfix** notation, also known as reverse polish notation (RPN)

• **Operators** are placed after **operands**.
  
  \[
  23^+ \\
  AB^/ 
  \]

• Postfix notation is parenthesis-free as long all operators have fixed # operands
Evaluating Postfix Expressions

• Example: \(345 + * 612 + / -\)

• This is equivalent to the infix expression: 
  \(3 * (4 + 5) - 6 / (1 + 2)\)

How do we evaluate this postfix expression?
Implementation Idea

- Whenever we encounter an operator, we apply it to the last two operands we’ve seen.

<table>
<thead>
<tr>
<th>Item Read from Postfix Expression</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>Push it onto the stack.</td>
</tr>
<tr>
<td>Operator</td>
<td>Pop the top two operands from the stack and apply the operator to them. Push the result.</td>
</tr>
</tbody>
</table>

345+*612+/-
Summary

• **Stack**: a useful abstraction
• **Implementation**: can be done with arrays
• Key operations: **push**, **pop**
• Applications: reversing, matching, expression evaluation
Today

• Stacks
  – Operations
  – Implementation using resizable array
  – Implementation using generics
• Applications using stacks
• Queues
  – Operations
  – Implementation
  – Applications
Queues

• The word *Queue* is British for *Line*.
  – The first person that enters the queue gets served.
Queue Data Structure

• Queue is *usually* stored as a continuous list of elements.

• The Queue data structure is similar to the Stack data structure, except it’s *First-in-First-Out (FIFO)*

• The first element is referred to as the **Front** (or **head**)

• The last element is referred to as the **Rear** (or **tail**)

2/16/2012
Queue Applications

• Queues are very useful in a computer:
  1. Printer queue
  2. Keyboard buffer
  3. Network buffer
  4. ...
Queue API

```java
public class QueueOfStrings {
    QueueOfStrings()  // create an empty queue
    void enqueue(String s)  // insert a new item onto queue
    String dequeue()  // remove and return the item least recently added
    boolean isEmpty()  // is the queue empty?
    int size()  // number of items on the queue
}
```
Queue: resizing array implementation

Array implementation of a queue.

- Use array $q[]$ to store items in queue.
- $enqueue()$: add new item at $q[tail]$.
- $dequeue()$: remove item from $q[head]$.
- Update $head$ and $tail$ modulo the capacity.
- Add resizing array.
Queue: Implementations

• Generic Queue using resizing array
  ResizingArrayQueue.java
• Generic queue using linked list (next lecture)
  Queue.java
Queue: applications

• Josephus problem

N people agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to N-1) and proceed around the circle, eliminating every Mth person until only one person is left.

Print out the order in which people are eliminated
Today

• Stacks
  – Operations
  – Implementation using resizable array
  – Implementation using generics
• Applications using stacks
• Queues
  – Operations
  – Implementation
  – Applications