CMPT 225
Stack
Last Lecture

- Effective sort algorithm
  - Merge sort
Learning Outcomes

At the end of this lecture, students will be able to:

- Describe Stack
- Define public interface of Stack ADT
- Design and implement Stack ADT using various data structures
- Compare and contrast these various implementations using Big O notation
- Give examples of real-life applications (problems) where we could use Stack ADTs to solve the problem
- Solve problems using Stack ADT
Today’s Menu

- Introducing another position-oriented data collection
  - Stack

which we shall design and implement as an abstract data type (ADT) class
Step 1 - Problem Statement

Web Browser Back and Forward buttons.

Sample run:

- I did some searching on Google and I am currently looking at Google’s search results
- I click on “wiki” search result and go to “wiki” web page
- From “wiki” web page, I click on Back button, i.e., go back to “Google” web page
- I click on Forward button, i.e., go forward to “wiki” web page
Step 2 - Design

• Solution requires a data collection ADT that allows the following:
  ◦ Add a new element
  ◦ Remove most recently added element
What is a Stack?

- What can we do with a stack?
What characterizes a Stack?

- A stack only allows elements to be inserted and removed at one end -> top
  - The other end -> bottom
- Access to other elements in the stack is not allowed
- Position-oriented: LIFO / FILO
- Linear data collection
- Objects not necessarily distinct
- Not a “general-purpose” ADT
Step 2 – Design - Stack Operations

- **isEmpty**: Is the stack empty?
- **push**: Insert an element onto the top of the stack
- **pop**: Remove the topmost element of the stack
- **peek**: Retrieve the topmost element of the stack (but do not remove the element)
- **popAll**: Remove all elements of the stack
Step 2 – Design –
Stack Public Interface – Contract

NOTE: Expressed in C++ and using template
Class invariant: LIFO / FILO

// Description: Returns true if this stack is empty otherwise false.
// Time Efficiency: O(1)
bool isEmpty() const;

// Description: Adds a new element to the top of this stack.
// Returns true if the addition is successful otherwise false.
// Time Efficiency: O(1)
bool push(const ElementType& newElement);
Step 2 – Design – Stack Public Interface – Contract

// Description: Removes the top element of this stack.
// Returns true if the removal is successful otherwise false.
// Precondition: The stack is not empty.
// Time Efficiency: O(1)
bool pop();

Alternative:
// Description: Removes and returns the top element of this stack.
// Precondition: The stack is not empty.
// Exceptions: Throws EmptyStackException if this stack is empty.
// Time Efficiency: O(1)
ElementType pop() throw(EmptyStackException);
// Description: Removes all elements from this stack.
// Returns true if the removal is successful otherwise false.
// Precondition: The stack is not empty.

bool popAll();

// Description: Returns the top of this stack.
// Precondition: The stack is not empty.
// Postcondition: This stack is unchanged.
// Exceptions: Throws EmptyStackException if this stack is empty.
// Time Efficiency: O(1)

ElementType peek() const throw(EmptyStackException);
Let’s Test the Stack Public Interface

Using our “Web Browser Back and Forward buttons” problem statement

◦ Currently looking at “google” -> `currentURL`

◦ Click on “wiki” -> `newURL` -> `open(newURL )`
  • if ( ! back.push(`currentURL`) ) throw exception
  • `currentURL = newURL`
  • if ( ! forward.popAll( ) ) throw exception

◦ Click on Back button, i.e., go back to “google”, `currentURL` -> “wiki” -> `back( )`
  • if ( back.isEmpty( ) ) throw exception
  • if ( ! forward.push(`currentURL`) ) throw exception
  • `currentURL = back.pop( )`

◦ Click on Forward button, i.e., go forward to “wiki”, `currentURL` -> “google” -> `forward( )`
  • if ( forward.isEmpty( ) ) throw exception
  • if ( ! back.push(`currentURL`) ) throw exception
  • `currentURL = forward.pop( )`  ( why not forward.peek( )? )
Step 3 - Implementing Stack as a data collection ADT class

- Array-based implementation
Step 3 - Implementing Stack as a data collection ADT class

- Link-based implementation
Step 3 - Implementing Stack as a data collection ADT class

- List ADT-based implementation
Stack ADT - Comparing both its implementations

- Time efficiency of Stack ADT’s operations (worst case scenario) expressed using the Big O notation

<table>
<thead>
<tr>
<th>Operations</th>
<th>array-based</th>
<th>link-based</th>
<th>List ADT-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peek</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popAll</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When Stack ADT is appropriate

- Examples of problem statements that would be solved using a data collection Stack ADT class
  - Compiler: checking for balanced braces while parsing the code in order to verify the syntax
  - Evaluating Postfix expressions
  - Finding our way through a maze
  - Simulating the execution of recursive operations by displaying the call stack, i.e., the activation records (or stack frames) kept in memory and their content
  - Text editing application: Undo and Redo buttons
Summary

• Introduced another position-oriented data collection
  ◦ Stack

designed and implemented as an abstract data type (ADT) class
Next Lecture

- Moving on to yet other position-oriented data collection!!!