CMPT 225

Lecture 9 – Stack
Last Lecture

- We did an ...
  - activity about Stack
Learning Outcomes

- At the end of this lecture (and the activity), a student 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 to solve the problem
  - Solve problems using Stack ADT
Today’s menu

- Going over our Stack activity
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 that allows the following:
  - Add a new element
  - Remove most recently added element
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
- Access to other elements in the stack is not allowed
- LIFO / FILO
- Linear data collection
- Not a “general-purpose” ADT
Step 2 – Design - Stack operations

- **isNullOrEmpty**: 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 - 1

**NOTE:** Expressed in C++ and using template

**Class invariant:** LIFO / FILO

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

```cpp
// 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 - 2

// 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);
Step 2 – Design – Stack public interface – Contract - 3

// 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 an ADT

- Array-based implementation
Step 3 - Implementing Stack as an ADT

- Link-based implementation
Step 3 - Implementing Stack as an ADT - 1

- List ADT-based implementation – refer to List.h (List public interface) posted on our course web site – we shall use it to implement a Stack:

```cpp
class Stack {
private:
    List * elements = new List( );
public: /* Stack public interface */
    bool isEmpty( ) const;
    bool push(const ElementType& newElement);
    bool pop( );
    bool popAll( );
    ElementType peek( ) const throw(EmptyStackException);
};
```
List ADT-based implementation

```cpp
bool isEmpty() const {
    return elements->getElementCount() == 0;
}

bool push(const ElementType& newElement) {
    // If we consider the “top of the Stack” being the “front of the List”
    return elements->insert(1, newElement);

    OR

    // If we consider the “top of the Stack” being the “end of the List”
    return elements->insert(elements->getElementCount() + 1, newElement);
}
```
Step 3 - Implementing Stack as an ADT - 3

```cpp
bool pop() {
    // If we consider the “top of the Stack” being the “front of the List”
    return elements->remove(1);

    OR

    // If we consider the “top of the Stack” being the “end of the List”
    return elements->remove(elements->getElementCount() + 1);
}

bool popAll( ) {
    return elements->clear();
}
```
ElementType peek() const throw(EmptyStackException) {
    // If we consider the “top of the Stack” being the “front of the List”
    return elements->getElement(1);

    OR

    // If we consider the “top of the Stack” being the “end of the List”
    return elements->getElement(elements->getElementCount() + 1);
}
Step 3 - Implementing Stack as an ADT - 5

- List ADT-based implementation
  - Advantages:
    - Simple implementation
    - Using code (the List ADT class) that has already been tested
  - Disadvantage:
    - Unless the List ADT class public interface states the time efficiency of its public methods, we will not be able to guarantee that the Stack public methods (calling the List ADT class public methods) will execute in $O(1)$
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 Stack
  - 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
Learning Check

- We can now ...
  - 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 to solve the problem
  - Solve problems using Stack ADT