CMPT 225

Lecture 7 – Data collection **Stack** as an ADT Class

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 class using various data structures (CDTs)
  - 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

Last lecture

- Compared the various implementations of our List ADT class:
  - Position-oriented (Table 1) versus value-oriented (Table 2)
  - Array-based implementation versus link-based implementation
- We can now …
  - Create class documentation (contract) for our class
  - **Step 4 – Compilation and Testing** of **List** ADT
    - Describe ***white box testing strategy***
    - Create ***test cases*** based on this strategy
    - Implement a ***test driver*** for a class based on these test cases
- Started our Stack activity

Today's menu

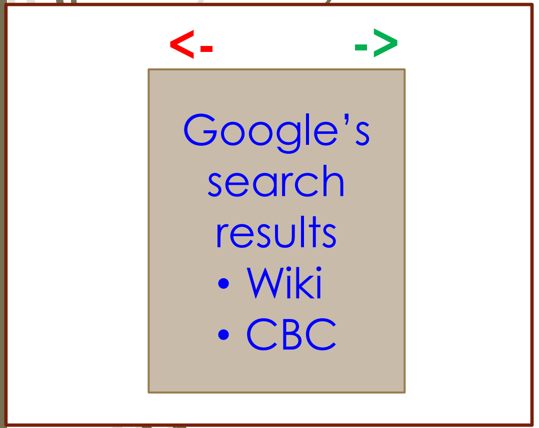
- Solve a problem using a Stack ADT class
  - Design and implement a Stack ADT class
    - Define its Public Interface
    - Design (and draw) and implement Stack ADT using various data structures (CDTs)
    - Compare and contrast these various implementations using Big O notation
    - Give examples of real-life applications (problems) where we could use Stack to solve the problem
- *Perform complexity analysis and use the Big O notation to represent time and space efficiency*

Step 1 - Problem Statement + Requirements

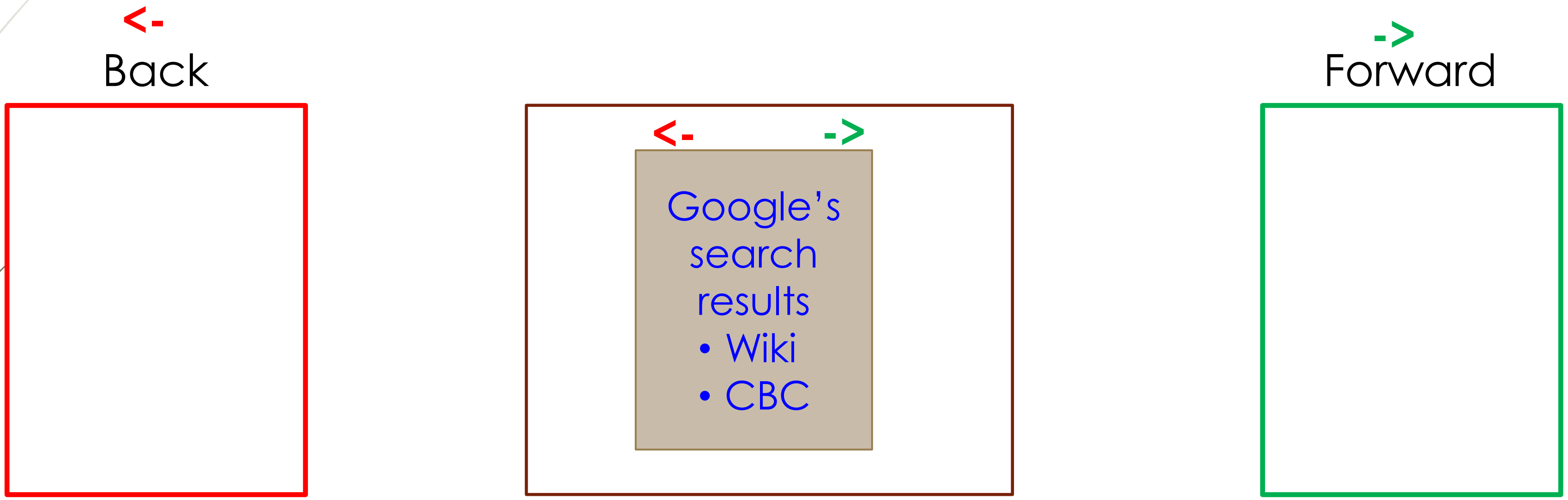
We are asked to develop s/w to implement *Back* and *Forward* buttons on a web browser.

Sample run:

- I do some searching on Google and I get a page full of Google's search results
- I click on "wiki" search result and go to "wiki" web page (within the same tab)
- From "wiki" web page, I click on *Back* button, i.e., go back to "Google" web page
- I click on "CBC" search result and go to "CBC" web page
- I click on some ad on the "CBC" web page and go this ad
- From this ad web page, I click on *Back* button and go to "CBC" web page
- I click on *Forward* button, i.e., go forward to the ad web page



Step 1 - Problem Statement + Requirements



Step 2 - Design

This is the
public interface
of the data
collection we
are looking for!

- How to implement *Back* button and *Forward* button?
- We need a data structure that allows the following operations:
  - Insert a new element (i.e., the URL of a web page)
  - Remove *most recently inserted* element
- And of course, we want to perform these operations as fast as possible -> **O(1)**

Stack

- A way of structuring data along with a rule dictating how this data is accessed
- Rule?
- What can we do with a Stack in the real world?
- How does a Stack behave in the real world?
  - Duplications allowed?
  - Are plates in a Stack sorted?
  - Must we keep track of the number of plates in a Stack? Or must we simply ascertain whether the Stack is empty?

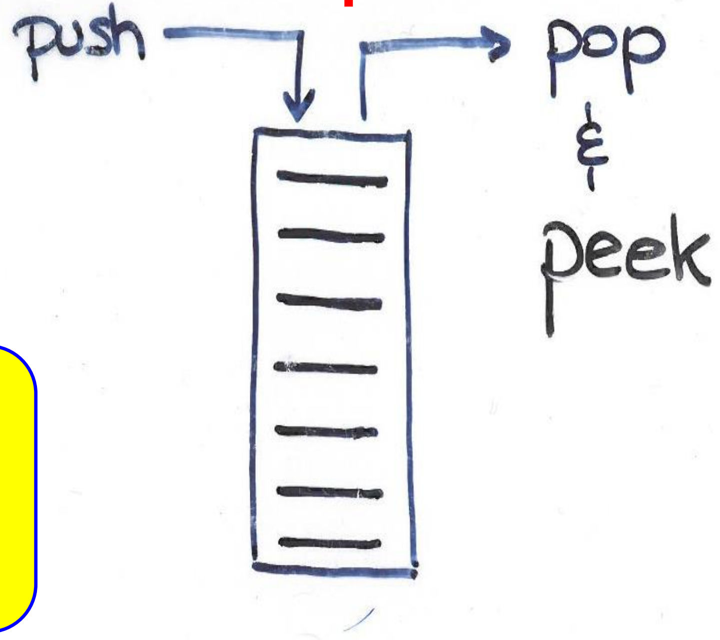
Source: http://i.istockimg.com/file_thumbview_approve/6443810/3/stock-photo-6443810-stack-of-plates.jpg

What characterizes a Stack?

- **Rule:** A Stack only allows elements to be inserted and removed at **one end -> top**
- Access to other elements in the Stack is **not allowed**
  - Nothing happens at the bottom of a Stack
- This **rule** is called:
  - LIFO -> Last in first out
  - FILO -> First in last out
- Stack is a linear data collection
- Stack is **not** a *general-purpose (flexible)* data collection

This **rule** becomes the the *Stack* ***class invariant***

top



Step 2 – Design - Stack operations

To become the **public interface** of our Stack ADT class

- **`push`**: Insert an element onto the top of the Stack
- **`pop`**: Remove the topmost element of the Stack
- **`peek`**: Gives access to the topmost element of the Stack (but does not remove it from the Stack)
- **`isEmpty`**: Is the Stack empty?

About Stack operations and class invariant

- It is the implementation of the methods of our *Stack* class that ensures that the **class invariant** (LIFO or FILO) holds true!
  - We write our code such that …
    - **`push`** method **only** pushes an element at the **top** of the Stack
    - **`pop`** method **only** pops the element on the **top** of the Stack
    - **`peek`** method **only** peeks at the element on the **top** of the Stack
- Therefore it is important to clearly define and indicate where the **top** of the Stack is located

Step 3 – Implementation – `Stack.h`
Stack public interface – Contract

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

Class invariant: LIFO / FILO

```
// Description: Returns true if this Stack is empty otherwise false.
// Postcondition:  This Stack is unchanged.
// Time Efficiency: O(1)
bool isEmpty( ) const;

// Description:  Adds a new element to the top of this Stack.
// Exception: Throws PushFailedException if push unsuccessful.
// Time Efficiency: O(1)
void push(ElementType& newElement);
```

Step 3 – Implementation – `Stack.h`
Stack public interface – Contract

```
// Description: Removes the top element of this Stack.
// Precondition:  The Stack is not empty.
// Exception: Throws EmptyStackException if this Stack is empty.
// Time Efficiency: O(1)
void pop( );
```

Alternative:

```
// Description: Removes and returns the top element of this Stack.
// Precondition:  The Stack is not empty.
// Exception: Throws EmptyStackException if this Stack is empty.
// Time Efficiency: O(1)
ElementType & pop();
```

Step 3 – Implementation – `Stack.h`
Stack public interface – Contract

// Description: Removes all elements from this Stack.

// Postcondition: Stack is in same state as when constructed.

~~// Precondition: The Stack is not empty.~~

~~// Exception: Throws EmptyStackException if this Stack is empty.~~

void 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;

Question 4 of our Stack Activity

Step 3 – Implementation of CDT underlying our Stack ADT class

1. Array-based implementation

index:	0	1	2	3	4	5	6	7	8	9
elements ->	3	7	5	8						

top? **top?**

Is the **top** of the Stack at the front or at the end of the array. How to decide?
Answer: Select the location of **top** such that the operations of our Stack class are as time efficient as possible -> **O(1)**.

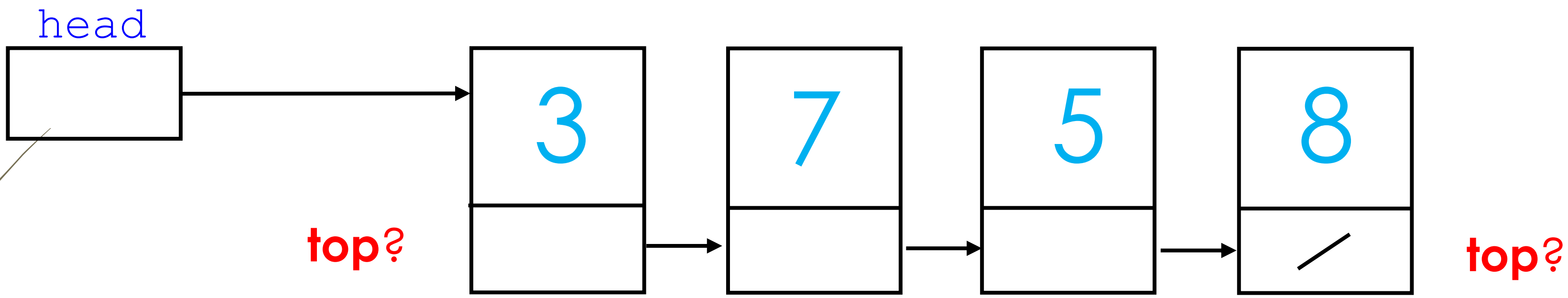
Private data members:

```
const unsigned int SIZE;
ElementType * elements;
unsigned int top; (if top >= 0)
int top: (if top starts at -1)
```

Question 5 of our Stack Activity

Step 3 – Implementation of CDT underlying our Stack ADT class

2. Linked list-based implementation



Is the **top** of the Stack at the front or at the end of the linked list. How to decide?
Answer: Select the location of **top** such that the operations of our Stack class are as time efficient as possible -> **O(1)**.

Private data member:
`StackNode * head;`

Step 3 - Implementation of Stack ADT class

3. List-based implementation

```
class Stack {
private:
   List * elements = nullptr;
public:  /* Stack public interface */
   bool isEmpty( ) const;
   bool push(ElementType& newElement);
   bool pop( );
   bool popAll( );
   ElementType & peek( ) const;
 };
```

we can use our **position-oriented List** ADT class (posted on our course web site) to implement our Stack

Step 3 - Implementation of Stack ADT class

3. List-based implementation

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

bool push(ElementType & newElement) {
   if ( elements == nullptr ) elements = new List(); ...
   // 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 - Implementation of Stack ADT class

```
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( ));
}


bool popAll( ) {
    return elements->removeAll();
}
```

Step 3 - Implementation of Stack ADT class

```
ElementType & peek( ) const {
    // 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( ));
}
```

Step 3 - Implementation of Stack ADT class

- List-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 guaranty that the Stack public methods (calling the List ADT class public methods) will execute in O(1)

Comparing various implementations of the Stack ADT class using Big O notation

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

Operations	array-based	linked list-based	List-based
`isEmpty`			
`push`			
`pop`			
`peek`			

Question 8 of our Stack Activity

Step 4 – Compilation and Testing – Let's test the Stack public interface

Stack object Stack object

Using our "Web Browser *Back* and *Forward* buttons" problem statement

- Currently looking at "google" -> **`currentURL`** (**`back.isEmpty()`** **`forward.isEmpty()`** true!) empty)
- Click on "wiki" -> **`newURL`** -> **open(newURL):**
  - `if ( ! back.push(currentURL) ) throw exception`
  - `currentURL  = newURL`
  - `forward.popAll( ) // Make sure forward Stack is empty`
- Click on *Back* button, i.e., go back to "google", "wiki" -> **`currentURL`** - **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", "google" -> **`currentURL`** - **forward( ):**
  - `if ( forward.isEmpty( ) ) throw exception`
  - `if ( ! back.push(currentURL) ) throw exception`
  - `currentURL = forward.pop( )` **(why not `forward.peek( )`?)**

When Stack ADT class 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 of statements
  - Evaluating Postfix expressions
  - Finding our way through a maze
  - Text editing application: Undo and Redo buttons
  - Simulating the execution of recursive operations by displaying the call Stack, i.e., the Stack frames kept in memory and their content

√ Learning Check

- We can now …
  - Describe Stack
  - Define public interface of Stack ADT class
  - Design, draw and implement Stack ADT using various data structures (CDTs)
  - 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

Next Lecture

- Introduce our **next linear data** collection -> **Queue**