# Stacks and Queues

## COSC 1P03 – Lecture 06 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University

Tuesday June 04, 2024

Total slides: 19

# Lecture Outline

01 Introduction to Stacks

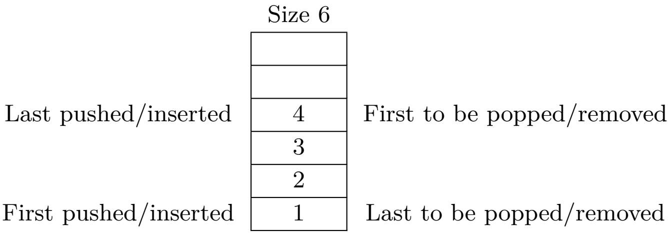
- Stacks as an Array Implementation
- Stacks as a Linked List Implementation
- The Stack Interface
- The Stack Implementation Classes

02 Introduction to Queues

- Queues as an Array Implementation
- Queues as a Linked List Implementation
- The Queue Interface
- The Queue Implementation Classes

# What is a Stack

- A *stack* is a data structure that allows insertion (push) at the back and removal (pop) from the back (LIFO, last-in-first-out).
- It contains a `size` property.
- There are two ways to encounter an exception:
  - `StackOverflowError` which is when you try to insert/push an element but the stack is full.
  - `StackUnderflowError` which is when you try to remove/pop an element but the stack is empty.
- You implement a stack using an array or a linked list (insertion at the back and removal from the back).
- A simple diagram of a stack that is implemented through an array:



Size 6

Last pushed/inserted | 4 | First to be popped/removed

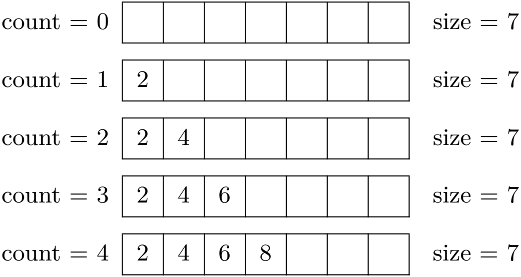
3

2

First pushed/inserted | 1 | Last to be popped/removed

# Pushing/Insertion in a Stack (Array)

- We could represent the array as a horizontal diagram where the left is the place of the first element pushed/inserted and right is the place of the last element pushed/inserted
- Let's place the values 2 (first), 4 (second), 6 (third) and 8 (fourth) in a stack of size 7. This is the order, we cannot insert at, say the middle.
- It is helpful to have a counter associated with the number of elements inserted as we are dealing with variable-sized array



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 0 | | | | | | | | size = 7 |
| count = 1 | 2 | | | | | | | size = 7 |
| count = 2 | 2 | 4 | | | | | | size = 7 |
| count = 3 | 2 | 4 | 6 | | | | | size = 7 |
| count = 4 | 2 | 4 | 6 | 8 | | | | size = 7 |

- We would get a `StackOverflowError` if we have pushed/inserted seven elements (which means we have a full array) and wanted to insert the eighth element

# Popping/Removing from a Stack (Array)

- To pop/remove from a stack, remove the last element inserted.
- Suppose we have the following stack:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 4 | 2 | 4 | 6 | 8 | | | | size = 7 |

- Popping will remove the value 8 and the count becomes 3, like so:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 3 | 2 | 4 | 6 | | | | | size = 7 |

- We must pop the last element, that is what a stack is, we cannot pop the element in the middle, or first element. We must pop the element inserted last. Otherwise, we don't have a stack, something *similar* to a stack but not a stack.

# Stacks as a Linked List Implementation

- We will not have a fixed size when it comes to linked list implementation as linked lists are a dynamic data structure
- To push/insert an element, perform an insertion at the front
- To pop/remove an element, perform removal at the front
- This will ensure that both pushing/insertion and popping/removal is $\mathcal{O}(1)$
- Again, we cannot push/insert, say at the middle, nor the end (if we have multiple elements). We must insert at the front
- When popping/removing, we cannot remove at the middle or end (if we have multiple elements), we must remove at the front
- We will not implement any other operations in terms of insertion/removal (no circular linked list, nor doubly linked list, etc)

# The Stack Interface I

- We have two ways to implement a stack, either using an array implementation or linked list implementation
- It would make sense that we would have the same structure but different implementation
- We will store `int`egers as the elements (so, the array is of type `int` and the item in the `Node` class is of type `int` too)
- How about we have something similar to the following when initializing:

```
IntStack a = new ArrayStack(); //array implemented stack
IntStack b = new LinkedStack();//linked list implemented stack
```

- In order for us to achieve that, we need to have an interface to implement, called `IntStack` (note that `Stack` is something Java has, so we will not use that name)
- We will have two implementation classes (they will throw exceptions), called `ArrayIntStack` and `LinkedIntStack`, which both `implements` the `IntStack` interface (and the `Serializable` interface to read/write the entire data structure to file, but we will not read/file from/to files)
- One test class, where we use `try`/`catch` blocks to handle the exceptions

# The Stack Interface II

- The `IntStack` has five methods to be implemented:

```
public interface IntStack {

  public void push (int item);

  public int top (); //returns last element added

  public int pop ();//returns last element added AND removes it

  public boolean empty ();

  public int size (); //extra for fun!

}
```

# The Stack Implementation Classes

- We will have two implementation classes that both implements the `IntStack` interface:
- The `ArrayIntStack` class will have to have two constructors and the default constructor (the one that doesn't accept parameters) will call the other constructor that accepts one parameters
  - This is referred to as *constructor chaining*
  - Suppose that we are currently in the default constructor and want to call another constructor that accepts an `int`, then we will use the `this` keyword and pass the some integer in parenthesis, as such: `this(100);`
  - The default constructor will initialize our array to 100 elements.
  - The other constructor accepts an `int`eger as a parameter and then initializes the array to that `int`eger passed.
- The linked list implementation will not deal with default size because linked lists are dynamic data structure and their size will increase/decrease
  - Only default constructor (*i.e.*, doesn't accept anything). We need a `Node` class (which also `implements` `Serializable`) containing `item` and `next`
- Both classes shouldn't have the `main` method. They are created to implement the logic, not test it.

# Custom Exceptions

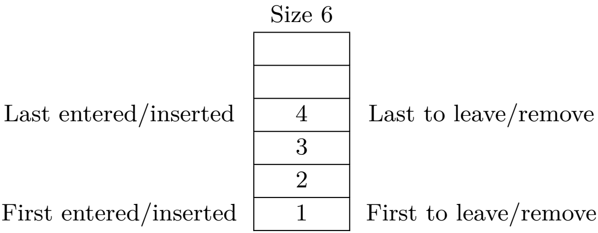
- We will have two custom exceptions:
  - `IntStackOverflowException`: when inserting in a full array (only in `ArrayIntStack` class)
  - `IntStackUnderflowException`: when removing but we don't have elements yet (both `ArrayIntStack` and `LinkedIntStack`)

```java
public class IntStackOverflowException extends RuntimeException {
}
```

```java
public class IntStackUnderflowException extends RuntimeException {
}
```

# What is a Queue

- A *queue* is a data structure that allows insertion (enter) at the back and removal (leave) from the front (FIFO, first-in-first-out).
- It contains a `size` property.
- There are two ways to encounter an exception:
  - `NoSpaceException` which is when inserting/entering an element but the queue is full.
  - `NoItemException` which is when you try to removing/leaving an element but the queue is empty.
- You implement a queue using an array or a linked list (insertion at the back and removal at the front).
- A simple diagram of a queue that is implemented through an array:



Size 6

Last entered/inserted 4 Last to leave/remove

3

2

First entered/inserted 1 First to leave/remove

# Entering/Insertion in a Queues (Array) I

- We could represent the array as a horizontal diagram where the left is the place of the first element entered/inserted and right is the place of the last element entered/inserted
- Let's place the values 2 (first), 4 (second), 6 (third) and 8 (fourth) in a queue of size 7. This is the order, we cannot insert at, say the middle.
- It is helpful to have a counter associated with the number of elements inserted as we are dealing with variable-sized array

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 0 | | | | | | | | size = 7 |
| count = 1 | 2 | | | | | | | size = 7 |
| count = 2 | 2 | 4 | | | | | | size = 7 |
| count = 3 | 2 | 4 | 6 | | | | | size = 7 |
| count = 4 | 2 | 4 | 6 | 8 | | | | size = 7 |

- We would get a `NoSpaceException` if we have entered/inserted seven elements (which means we have a full array) and wanted to insert the eighth element

# Entering/Insertion in a Queues (Array) II

- In the previous slide, we inserted the elements inside a queue and ended with

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 4 | 2 | 4 | 6 | 8 | | | | size = 7 |

- Let us perform leave/remove elements. We will remove the first element inserted, like so:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| count = 3 | | 4 | 6 | 8 | | | | size = 7 |
| count = 2 | | | 6 | 8 | | | | size = 7 |
| count = 1 | | | | 8 | | | | size = 7 |
| count = 0 | | | | | | | | size = 7 |

- We would get a `NoItemException` in case we tried to leave/remove again.

# Array Implemenation of a Queue

- We will need to use two (instance) `int`eger variables `front` and `rear`, to mark the front and last of the queue.
- We can see that our queue has blanks at the left indices once leaving/removal occurred.
- This means that we could either shift all the elements once to the left each removal (which costs $\mathcal{O}(n)$ moves), or use clever mathematics to find where is the beginning and end indices of our elements.
- Suppose we have our queue in an array called `data`:
  - To move `front` after leaving/removing, use

    ```
    front = (front + 1) % data.length; //shift once to right
    ```
  - To move `rear` after entering/insertion, use

    ```
    rear = (rear + 1) % data.length; //shift once to right
    ```

# Queues as a Linked List Implementation

- We will not have a fixed size when it comes to linked list implementation as linked lists are a dynamic data structure
- To enter/insert an element, perform an insertion at the rear
- It would be helpful to add a tail references to the far-right node to ensure insertion at the rear is $\mathcal{O}(1)$ than $\mathcal{O}(n)$.
- To leave/remove an element, perform removal at the front
- This will ensure that both entering/insertion and leaving/removal is $\mathcal{O}(1)$
- Again, we cannot enter/insert, say at the middle, we must insert at the rear.
- When leaving/removing, we cannot remove at the middle or end (if we have multiple elements), we must remove at the front
- We will not implement any other operations in terms of insertion/removal (*e.g.*, no circular linked list, nor doubly linked list, etc)

# The Queue Interface I

- We have two ways to implement a queue, either using an array implementation or linked list implementation
- It would make sense that we would have the same structure but different implementation
- We will store `int`egers as the elements (so, the array is of type `int` and the item in the `Node` class is of type `int` too)
- How about we have something similar to the following when initializing:

```
IntQueue a = new ArrayIntQueue(); //array implemented queue
IntQueue b = new LinkedIntQueue();//linked list implemented queue
```

- In order for us to achieve that, we need to have an interface to implement, called `IntQueue` (note that `Queue` is something Java has, so we will not use that name)
- We will have two implementation classes (they will throw exceptions), called `ArrayIntQueue` and `LinkedIntQueue`, which both `implements` the `IntQueue` interface (and the `Serializable` interface to read/write the entire data structure to file, but we will not read/file from/to files)
- One test class, where we use `try`/`catch` blocks to handle the exceptions

# The Queue Interface II

- The `IntQueue` has five methods to be implemented:

```
public interface IntQueue {

  public void enter (int item); //insertion

  public int front (); //returns first element added/to remove

  public int leave ();//returns first element added AND removes it

  public boolean empty ();//whether the queue is empty or not

  public int size (); //should take O(1), not O(n) to find size

}
```

# The Queue Implementation Classes

- We will have two implementation classes that both implements the `IntQueue` interface:
- The `ArrayIntQueue` class will have to have two constructors and the default constructor (the one that doesn't accept parameters) will call the other constructor that accepts one parameters
  - This is referred to as *constructor chaining*
  - Suppose that we are currently in the default constructor and want to call another constructor that accepts an `int`, then we will use the `this` keyword and pass the some integer in parenthesis, as such:
    `this(100);`
  - The default constructor will initialize our array to 100 elements.
  - The other constructor accepts an `int`eger as a parameter and then initializes the array to that `int`eger passed.
- The linked list implementation will not deal with default size because linked lists are dynamic data structure and their size will increase/decrease
  - Only default constructor (*i.e.*, doesn't accept anything). We need a `Node` class (which also `implements` `Serializable`) containing `item` and `next`
- Both classes shouldn't have the `main` method. They are created to implement the logic, not test it.

# Custom Exceptions

- We will have two custom exceptions:
  - `NoSpaceException`: when inserting in a full array (only in `ArrayIntQueue` class)
  - `NoItemException`: when removing but we don't have elements yet (both `ArrayIntQueue` and `LinkedIntQueue`)

```java
public class NoSpaceException extends RuntimeException {
}
```

```java
public class NoItemException extends RuntimeException {
}
```