Computer Science II
Handout 12

Stacks - LIFO

Another abstract data structure!

- *Stacks* are yet another collection of sequential data
- This time, elements are added/removed in a *Last In First Out* (LIFO) manner
  - So, the last element to be added will be the first one to be removed

Stacks - LIFO

- Imagine a stack of textbooks
  - While reading one book, you find a reference to a second
  - You open the second book, placing it on top of the first, and start reading
  - There you find a reference to a third book!
  - You open the third book, placing it on top of the first, and start reading
  - … and so on …
- Once you are finished reading each book, you remove it from the top of the *stack* as LIFO
  - This is the same idea as embarking/disembarking an airplane, parking cars in a narrow driveway, …

Stacks - LIFO

- Stacks are used *everywhere* in computer science!

- We already saw that *the stack* refers to the organization of frames within Java memory
    - These frames are added/removed as LIFO

- Stacks are also used in …
    - Parsing text (parenthesis matching, arithmetic expressions)
    - Solving graph traversal problems
    - Representing recursion (methods calling themselves repeatedly)

Stacks - LIFO

- Like with LinkedLists, there is a Java standard library class (Java.util.Stack), but we will implement our own to learn more!

- Stacks consist of nodes (elements) arranged in sequence

- A stack should support at least two operations:
  - Push
  - Pop

- It’s helpful to have a third:
  - Peek

Stacks - LIFO

- Push
  - Adds a new element to the top of the Stack

- Pop
  - Removes the top element from the Stack and returns it

- Peek
  - Returns the top element from the Stack, but does *not* remove it

Stacks - LIFO

- Recall that stacks are an example of an *abstract* data structure: they are independent of the specific implementation
  - For now, we will keep using Strings to represent the data
- We could use an array to store each element
  - We would be responsible for re-sizing when needed
- We could also use an ArrayList to store each element
  - We would be responsible for knowing the relevant methods

Stacks - LIFO

```
public class Stack {
      private ArrayList<String> stack;

      public Stack() {
            stack = new ArrayList<String>();
      }

      public boolean isEmpty() {
            return (stack.size() == 0);
      }

      // ...

}
```

Stacks - Push

Add an element to the top of the stack

```
public void push(String s) {



}
```

Stacks - Pop

Remove the top element from the Stack and return it

```
public String pop() {
    String top = “”;






    return top;
}
```

Stacks - Peek

Remove the top element from the Stack and return it

```
public String peek() {
    String top = “”;







    return top;
}
```

Stacks - Demo

```
public static void main(String args[]) {
        Stack s = new Stack();
        String tmp;

        s.push(“Anne”);
        s.push(“Bob”);
        s.push(“Carol”);
        System.out.println(s.pop());
        System.out.println(s.peek());
        System.out.println(s.pop());
        s.push(“Dwight”);
        s.push(“Ernie”);
        s.pop();
        System.out.println(s.pop());
        System.out.println(s.peek());
}
```

```
>
```

Stacks – LIFO

- Each of the primary methods has a specific *pre-condition* and *post-condition*
  - These are states that are guaranteed to be true before and after (respectively) the method executes

	Pre-condition	**Post-condition**
`push`	Stack is not full	Stack has new element on top
`pop`	Stack is not empty	Stack has top element removed
`peek`	Stack is not empty	None

Stacks – LIFO

- Establishing pre-conditions (true before), post-conditions (true after), and invariants (true always) is helpful for designing new programs
  - It can help break down a complicated problem for yourself
  - It can make your “black box” method understandable for someone else

Queues - FIFO

- Queues are another abstract data structure

- They are similar to stacks, except this time operating with First In First Out
  - This is exactly like a line-up (or *queue*, in UK English) of people

- Queues have both a *front* and an *end* (or *rear*)

- Items are added to the rear, removed from the front

Queues - FIFO

- Instead of push and pop, queues use *enqueue* and *dequeue*

- Enqueue
  - Add an element to the rear

- Dequeue
  - Remove and return the front element

- Peek
  - Return the front element without removing

Queues – FIFO

- What are the pre- and post-conditions for these Queue operations?

	Pre-condition	Post-condition
`enqueue`	Queue is not full	Queue has new element at rear
`dequeue`	Queue is not empty	Queue has front element removed
`peek`	Queue is not empty	None

Queues – FIFO

```
public class Queue {
    private ArrayList<String> q;

    public Queue() {
        q = new ArrayList<String>();
    }

    public boolean isEmpty() {
        return (q.size() == 0);
    }

    // ...

}
```

Queues - Enqueue

Add an element to the top of the stack

```
public void enqueue(String s) {
    // Uses index 0 as the ‘rear’
    q.add(s);

}
```

Queues - Enqueue

Remove the top element from the Stack and return it

```
public String dequeue() {
    String front = “”;






    return front;
}
```

Queues - Enqueue

Remove the element from the Stack and return it

```
public String peek() {
    String front = “”;







    return front;
}
```

Queues - Demo

```
public static void main(String args[]) {
        Queue s = new Queue();
        String tmp;

        s.enqueue(“Anne”);
        s.enqueue(“Bob”);
        s.enqueue(“Carol”);
        System.out.println(s.dequeue());
        System.out.println(s.peek());
        System.out.println(s.dequeue());
        s.enqueue(“Dwight”);
        s.enqueue(“Ernie”);
        s.dequeue();
        System.out.println(s.dequeue());
        System.out.println(s.peek());
}
```

```
>
```

More LinkedLists

- Both Stacks and Queues open to multiple implementations
  - We used ArrayLists throughout

- What would have been different had we used arrays?
  - For Stacks?
  - For Queues?

- How could we implement the same three “primary” methods if we decided to use LinkedLists instead of ArrayLists?