CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

9: Stacks, Queues, and Iterators

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators

Some slides adopted from Algorithms 4th Edition and Oracle tutorials
Stacks

- Dynamic linear data structures.
- Items are inserted and removed following the LIFO paradigm.
- **LIFO**: Last In, First Out.
- Similar to lists, there is a sequential nature to the data.
- Remove the *most* recent item.

- Metaphor of cafeteria plate dispenser.
  - Want a plate? **Pop** the top plate.
  - Add a plate? **Push** it to make it the new top.
  - Want to see the top plate? **Peek**.
- We want to make push and pop as time efficient as possible
### Example of stack operations

<table>
<thead>
<tr>
<th>push</th>
<th>To</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>-</th>
<th>be</th>
<th>-</th>
<th>-</th>
<th>that</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stack Order:**

- **Last In, First Out**
- **Push to Top:**
- **Pop from Top:**

**Operations:**

- **Push to Top:**
- **Pop from Top:**
Implementing stacks with ArrayLists

- Where should the top go to make push and pop as efficient as possible?
- The end/rear represents the top of the stack.
- To push an item `add(Item item)`.
  - Adds at the end. Average $O(1)$.
- To pop an item `remove()`.
  - Removes and returns the item from the end. Average $O(1)$.
- To peek `get(size()-1)`.
  - Retrieves the last item. $O(1)$.
- If the front/beginning were to represent the top of the stack, then:
  - Push, pop would be $O(n)$ and peek $O(1)$.
Implementing stacks with singly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The *front* represents the top of the stack.
- To push an item `add(Item item)`.
  - Adds at the head. \(O(1)\).
- To pop an item `remove()`.
  - Removes and retrieves from the head. \(O(1)\).
- To peek `get(0)`.
  - Retrieves the head. \(O(1)\).
- If the *end* were to represent the top of the stack, then:
  - Push, pop, peek would all be \(O(n)\).
Implementing stacks with doubly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The front represents the top of the stack.
- To push an item `addFirst(Item item)`.
  - Adds at the head. \(O(1)\).
- To pop an item `removeFirst()`.
  - Removes and retrieves from the head. \(O(1)\).
- To peek `head.item`.
  - Retrieves the head. \(O(1)\).
- Unnecessary memory overhead with extra pointers.
- If the end were to represent the top of the stack, we’d need to use `addLast(Item item), removeLast()`, and `tail.item` to have \(O(1)\) complexity.
Textbook implementation of stacks

- `ResizingArrayStack.java`: for implementation of stacks with ArrayLists.
- `LinkedStack.java`: for implementation of stacks with singly linked lists.
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators
Queues

- Dynamic linear data structures.
- Items are inserted and removed following the FIFO paradigm.
  - **FIFO**: First In, First Out.
- Similar to lists, there is a sequential nature to the data.
- Remove the *least* recent item.

- Metaphor of a line of people waiting to buy tickets.
- Just arrived? **Enqueue** person to the end of line.
- First to arrive? **Dequeue** person at the top of line.
- We want to make enqueue and dequeue as time efficient as possible.
### Example of stack operations

<table>
<thead>
<tr>
<th>enqueue</th>
<th>To</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>-</th>
<th>be</th>
<th>-</th>
<th>-</th>
<th>that</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td>to</td>
<td>be</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **enqueue**: To be or not to - be - that - - - is
- **dequeue**: To be or not to be

**First In, First Out**

- **enqueue at end**: To be or not to be
- **dequeue from beginning**: To be or not to be
Implementing queue with ArrayLists

- Where should we enqueue and dequeue items?
- To enqueue an item `add()` at the end of `ArrayList`. Average $O(1)$.
- To dequeue an item `remove(0)`. $O(n)$.
- What if we add at the beginning and remove from end?
  - Now dequeue is cheap ($O(1)$) but enqueue becomes expensive ($O(n)$).
Implementing queue with singly linked list

- Where should we enqueue and dequeue items?
- To enqueue an item `add()` at the head of SLL ($O(1)$).
- To dequeue an item `remove(size()-1)` ($O(n)$).
- What if we add at the beginning and remove from end?
  - Now dequeue is cheap ($O(1)$) but enqueue becomes expensive ($O(n)$).
- $O(1)$ if we have a tail pointer.
  - Simple modification in code, big gains!
  - Version that textbook follows.
Implementing queue with doubly linked list

- Where should we enqueue and dequeue items?
- To enqueue an item `addFirst()` at the head of DLL \(O(1)\).
- To dequeue an item `removeLast()` \(O(1)\).
- What if we add at the beginning and remove from end?
  - Both are \(O(1)\)!
Textbook implementation of queues

- **ResizingArrayQueue.java**: for implementation of queues with ArrayLists.
- **LinkedQueue.java**: for implementation of queues with singly linked lists.
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators
Stack applications

- Java Virtual Machine.
- Basic mechanisms in compilers, interpreters (see CS101).
- Back button in browser.
- Undo in word processor.
- Infix expression evaluation (Dijkstra’s algorithm with two stacks).
- Postfix expression evaluation.
1.3 Dijkstra's 2-Stack Demo
Postfix expression evaluation example

Example: $(52 - ((5 + 7) * 4)) \Rightarrow 52 5 7 + 4 * -$
Queue applications

- Spotify playlist.
- Data buffers (netflix, Hulu, etc.).
- Asynchronous data transfer (file I/O, sockets).
- Requests in shared resources (printers).
- Traffic analysis.
- Waiting times at calling center.
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators
The Java Collections Framework

Collections

Deque in Java Collections

- Do not use **Stack**.
- **Queue** is an interface...
- It’s recommended to use **Deque** instead.
- Double-ended queue (can add and remove from either end).

```java
java.util.Deque;

public interface Deque<E> extends Queue<E>
- You can choose between **LinkedList** and **ArrayDeque** implementations.

  - Deque deque = new ArrayDeque(); //preferable
```

[https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html)
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators
**Iterator Interface**

- Interface that allows us to traverse a collection one element at a time.

```java
public interface Iterator<E> {
    // returns true if the iteration has more elements
    // that is if next() would return an element instead of throwing an exception
    boolean hasNext();

    // returns the next element in the iteration
    // post: advances the iterator to the next value
    E next();

    // removes the last element that was returned by next
    default void remove(); // optional, better avoid it altogether
}
```

[https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)
Iterator Example

List<String> myList = new ArrayList<String>();
//... operations on myList

Iterator listIterator = myList.iterator();
while(listIterator.hasNext()){
    String elt = listIterator.next();
    System.out.println(elt);
}
Java 8 introduced lambda expressions

- **Iterator** interface now contains a new method.

  ```java
default void forEachRemaining(Consumer<? super E> action)
```

- Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

```
listIterator.forEachRemaining(System.out::println);
```
Iterables Interface

- Interface that allows an object to be the target of a for-each loop:

```java
for(String elt: myList){
    System.out.println(elt);
}
```

```java
interface Iterable<E>{
    //returns an iterator over elements of type E
    Iterator<E> iterator();

    //Performs the given action for each element of the Iterable until all elements
    //have been processed or the action throws an exception.
    default void forEach(Consumer<? super E> action);
}
```

```java
myList.forEach(elt-> {System.out.println(elt)});  
myList.forEach(System.out::println);
```
ITERATORS

How to make your data structures iterable?

1. Implement `Iterable` interface.

2. Make a private class that implements the `Iterator` interface.

3. Override `iterator()` method to return an instance of the private class.
Example: making ArrayList iterable

```java
public class ArrayList<Item> implements Iterable<Item> {
    //...
    public Iterator<Item> iterator() {
        return new ArrayListIterator();
    }

    private class ArrayListIterator implements Iterator<Item> {
        private int i = 0;
        public boolean hasNext() {
            return i < n;
        }
        public Item next() {
            return a[i++];
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```
Traversing `ArrayList`

- All valid ways to traverse `ArrayList` and print its elements one by one.

```java
for (String elt: a1) {
    System.out.println(elt);
}

a1.forEach(System.out::println);
a1.forEach(elt->{System.out.println(elt);});

a1.iterator().forEachRemaining(System.out::println);
a1.iterator().forEachRemaining(elt->{System.out.println(elt);});
```
Lecture 9: Stacks, Queues, and Iterators

- Stacks
- Queues
- Applications
- Java Collections
- Iterators
ASSIGNED READINGS AND PRACTICE PROBLEMS

Readings:

- Oracle’s guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Deque: https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
  - Iterator: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
  - Iterable: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
- Textbook:
  - Chapter 1.3 (Page 126–157)
- Website:
  - Stacks and Queues: https://algs4.cs.princeton.edu/13stacks/

Practice Problems:

- 1.3.2–1.3.8, 1.3.32–1.3.33