Week2

Arrays

Linked Lists

Equivalence Testing

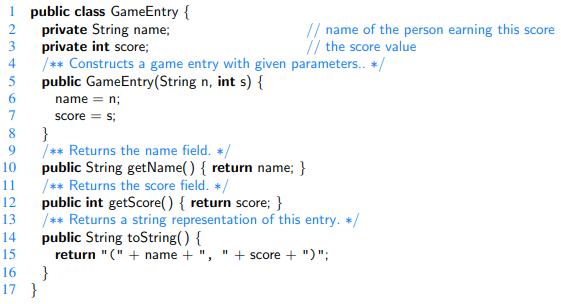
Cloning

Stacks

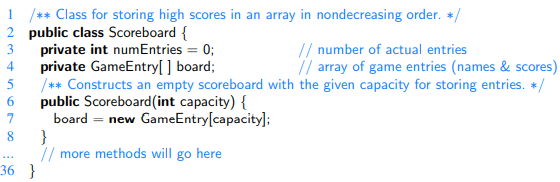
Queues

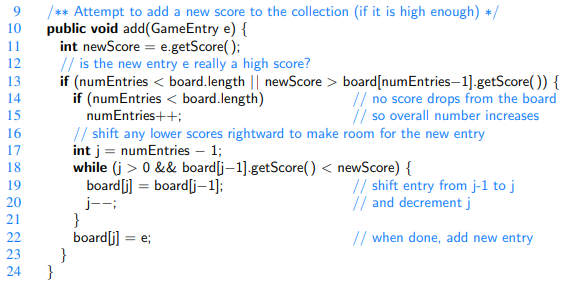
Deques

Storing Game Entries in an Array

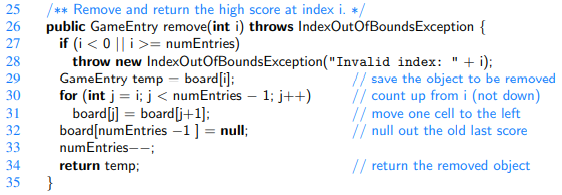


A Class for High Scores

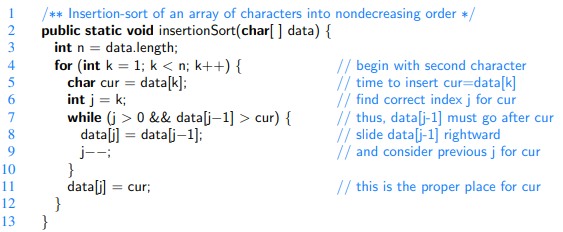




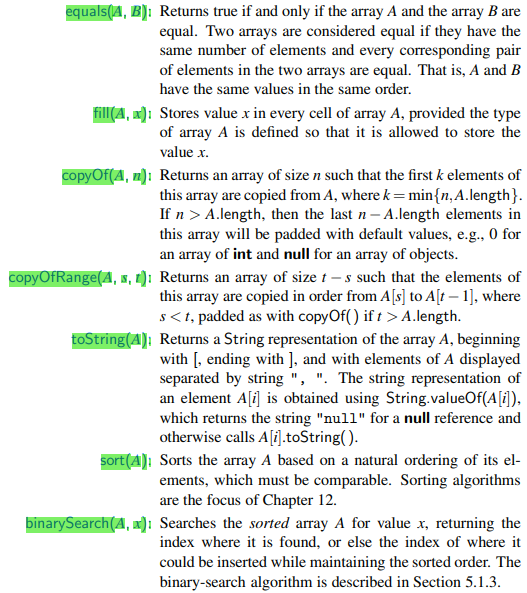
The Scoreboard.remove Operation



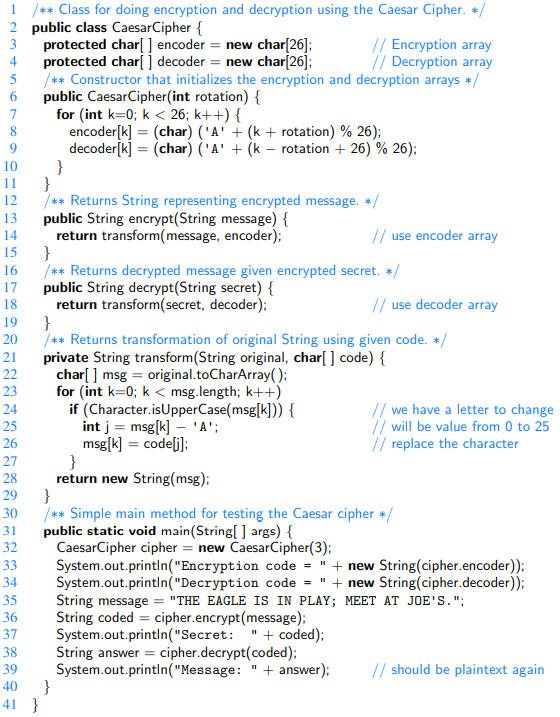
Insertion Sort



Most Commonly Used Methods of java.util.Arrays:



The Caesar Cypher

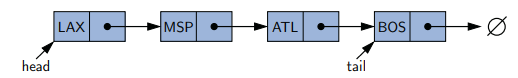


Tic-Tac-Toe

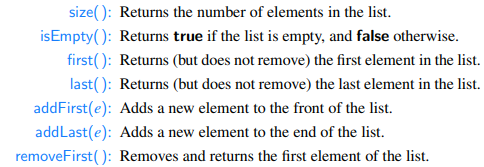




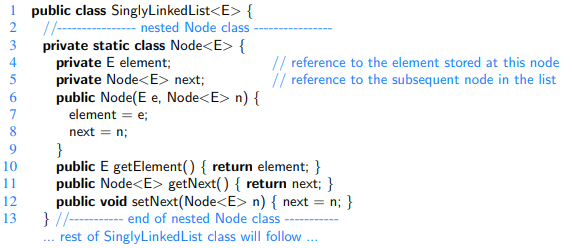
Singly Linked Lists



Methods:

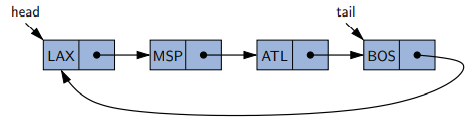


We will use nested classes inside the SinglyLinkedList class.





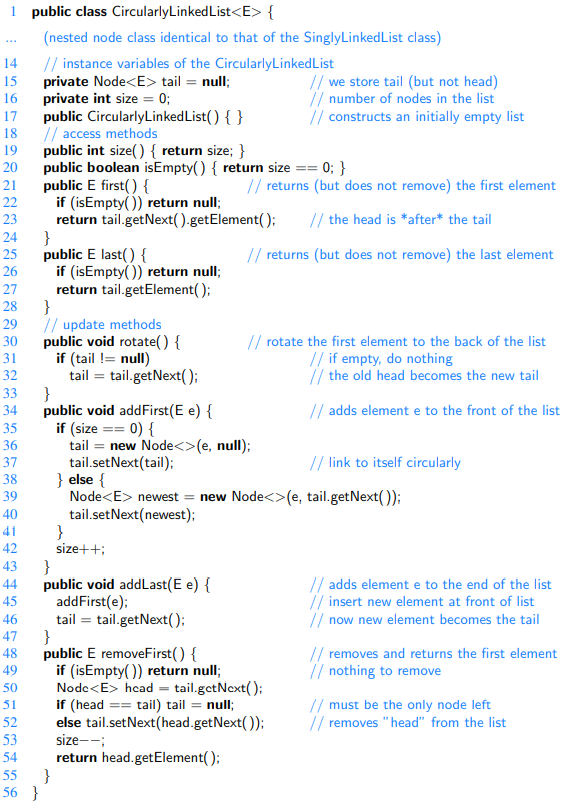
Circularly Linked Lists



Additional Method:

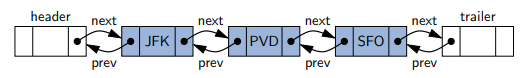


We no longer explicitly maintain the head reference. So long as we maintain a reference to the tail, we can locate the head as tail.getNext( ).



[Fontos, hogy alkalmazni is tudjam valamilyen gyakorlati peldara. Pl. Round-Robin Scheduling]

Doubly Linked Lists



Header, trailer sentinels.

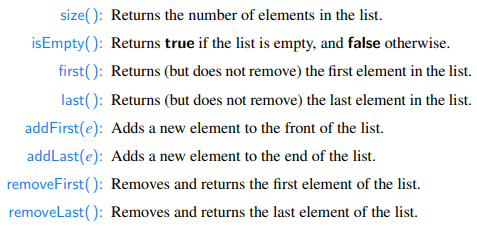
Advantages of using sentinels:

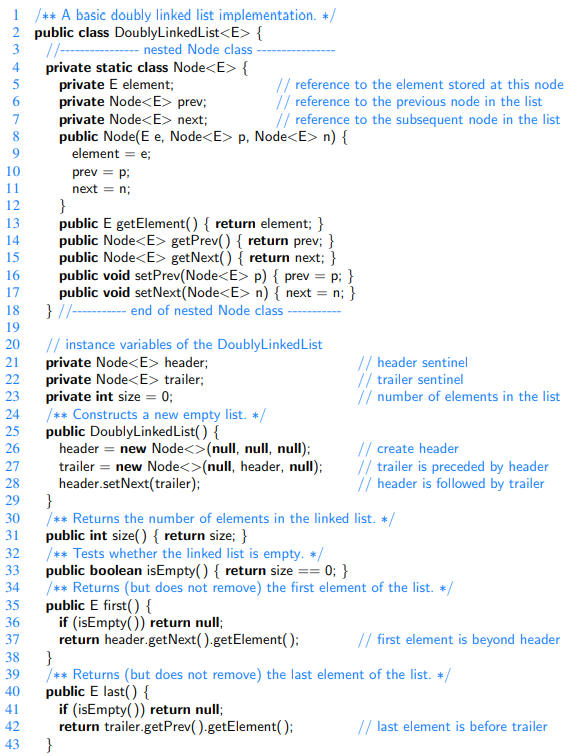
Simplifies the logic of our operations:

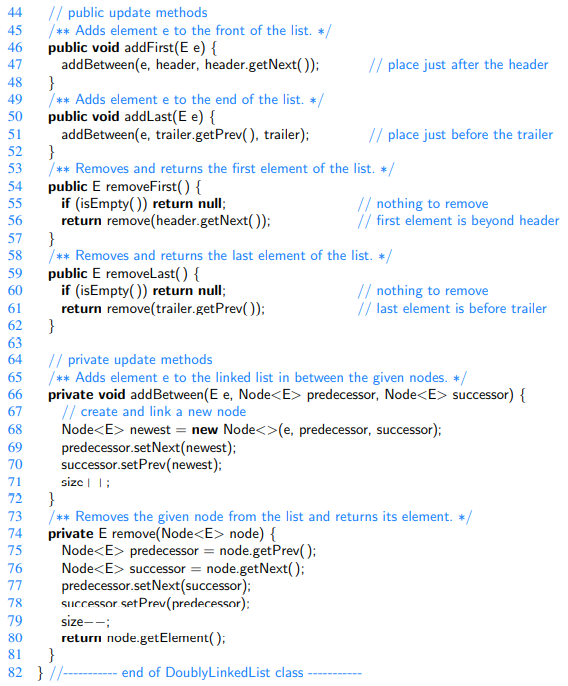
- header and trailer nodes never change—only the nodes between them change.

- we can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes.

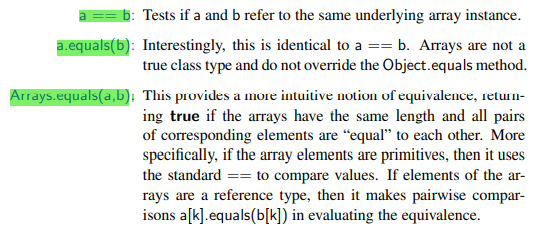
Methods:

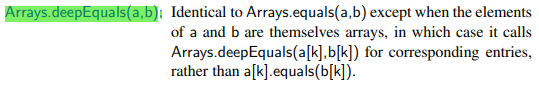




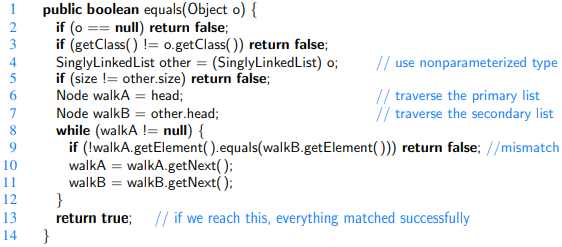


Equivalence Testing with Arrays





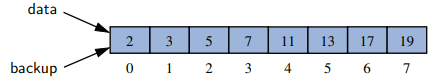
Equivalence Testing with Linked Lists



Cloning Arrays

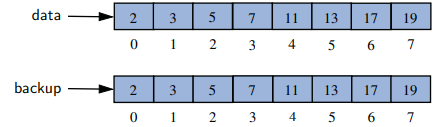
Not a copy, just an alias:





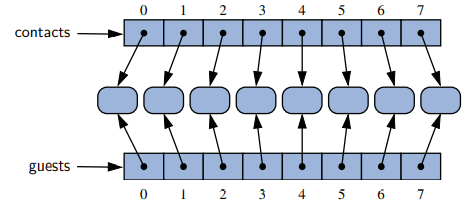
An actual copy:





The array stored primitive types. What about reference types?

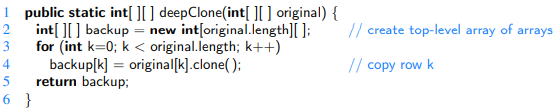
The clone( ) method produces a **shallow copy** of the array, producing a new array whose cells refer to the same objects referenced by the first array:



A **deep copy** of the contact list can be created by iteratively cloning the individual elements, as follows, but only if the Person class is declared as **Cloneable:**



A deep copy of a 2D array:

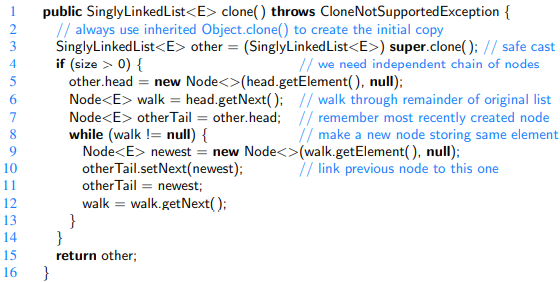


Cloning Linked Lists

We modify the first line:

,

and add the method clone():

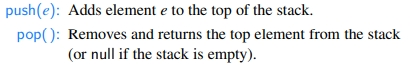


Stacks

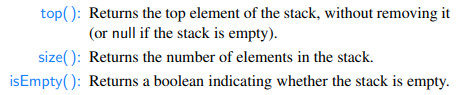
LIFO: last-in, first-out

Methods:

Update methods:

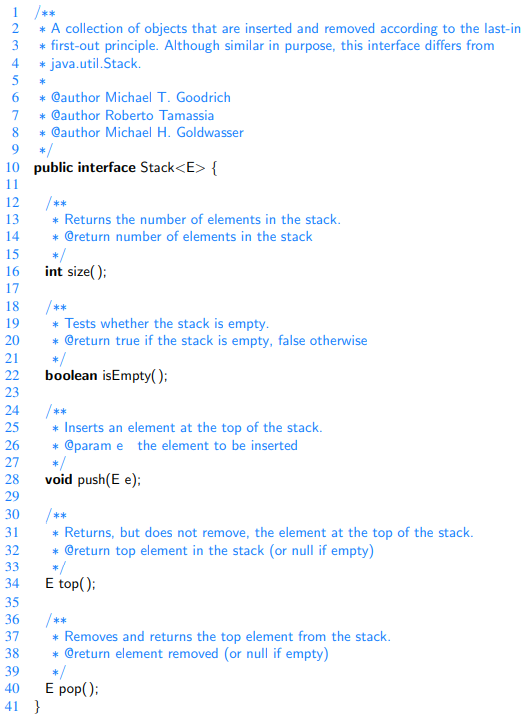


Accessor methods:

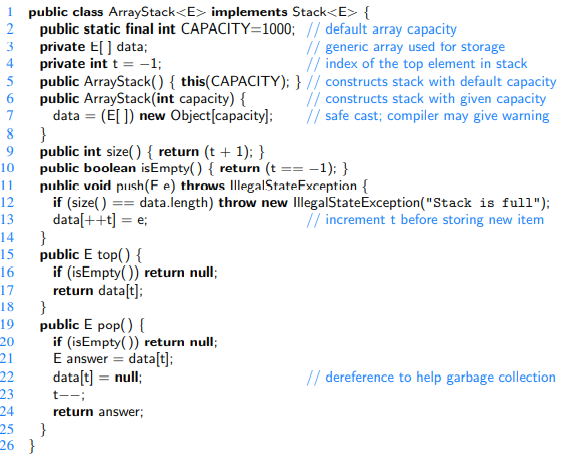


We will do a Stack **interface** (as an API), which describes the names of the methods and they should be declared and used.

We will use Java’s **generics framework** allowing the elements stored in the stack to belong to any object type <E>. E.g. Stack<Integer>



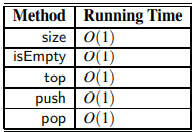
An Array-based Implementation of the Stack Interface



Note: When the stack is empty, t = -1.

Disadvantage: fixed-capacity array.

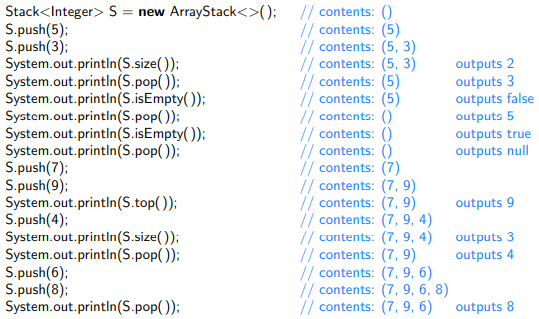
Time Complexity of the Array-based Stack



Space Complexity of the Array-based Stack

The space usage is O(N), where N is the size of the array, determined at the time the stack is instantiated, and independent from the number n ≤ N of elements that are actually in the stack.

Sample Usage:



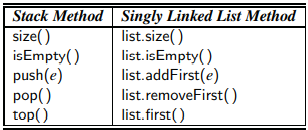
An ‘Singly Linked List’ -based Implementation of the Stack Interface

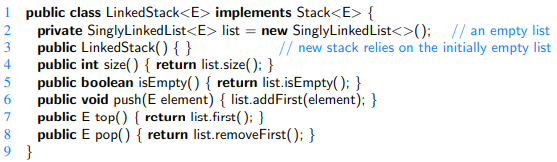
No capacity limit,and the memory usage is always proportional to the number of actual elements currently in the stack.

We need to decide if the top of the stack is at the *front* or *back* of the list. There is clearly a best choice here, however, since we can insert and delete elements in constant time only at the front. With the *top* of the stack stored at the *front* of the list, all methods execute in **constant time**.

We will use the *Adapter* design pattern.

We can adapt our (previously implemented) SinglyLinkedList class to define a new LinkedStack class.   
[Tehat egy Stack-et implementalunk, ugy, hogy a SinglyLinkedList-et hasznaljuk storage-nak.]

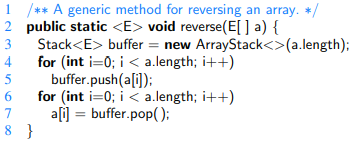




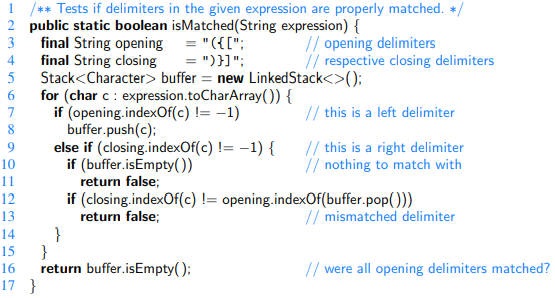
Reversing an Array Using a Stack

If we push: 1, 2 and 3,

We can pop 3, 2 and 1.



Matching parenthesis



Matching HTML tags

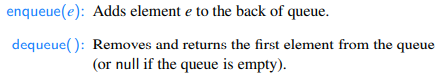


Queues

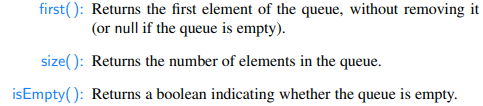
FIFO: first-in, first-out

Methods:

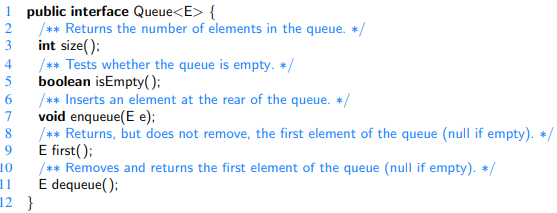
Update methods:



Accessor methods:



The Queue interface:



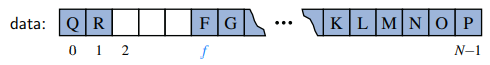
Array-Based Queue Implementation

Allowing the front of the queue to drift away from index 0. In this representation, index f denotes the location of the front of the queue:



However, there remains a challenge with the revised approach. With an array of capacity N, we should be able to store up to N elements before reaching any exceptional case. Solution: using it circularly.

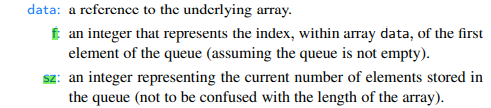
Using an Array Circularly

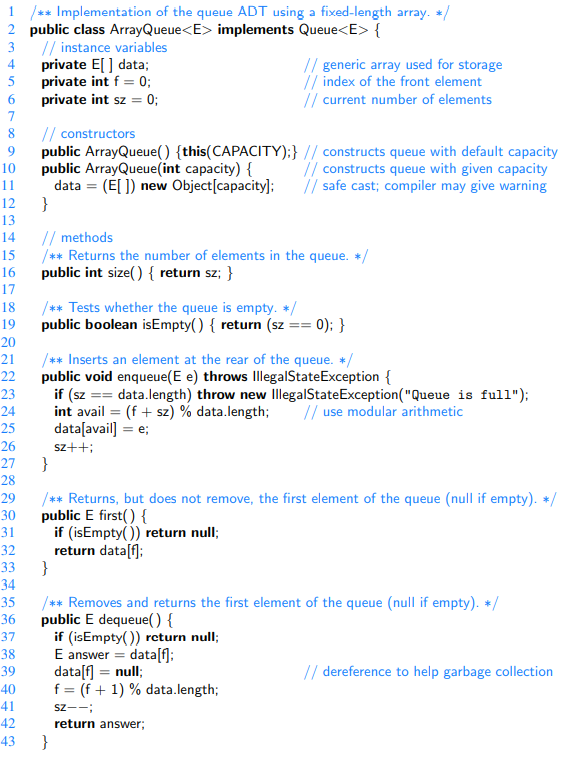


When we dequeue an element and want to “advance”[step to the next] the front index, we use the arithmetic f = (f +1) % N.

The Queue Implementation with an Array

Variables:





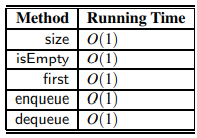
Adding and Removing Elements

We compute the index of the next opening based on the formula:

avail = (f + sz) % data.length;

When the dequeue method is called, the current value of f designates the index of the value that is to be removed and returned. We keep a local reference to the element that will be returned, before setting its cell of the array back to null, to aid the garbage collector. Then the index f is updated to reflect the removal of the first element, and the presumed promotion of the second element to become the new first. In most cases, we simply want to increment the index by one, but because of the possibility of a wraparound configuration, we rely on modular arithmetic, computing f = (f+1) % data.length, as described earlier.

Queue Time Complexity

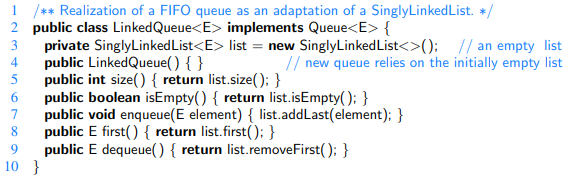


Note: With Singly Linked List implementation it is slightly different.

Queue Space Complexity

The space usage is O(N), where N is the size of the array, determined at the time the queue is created, and independent from the number n < N of elements that are actually in the queue.

The Queue Implementation with a Singly Linked List

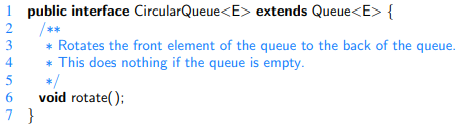


All methods run with O(1) worst-case time.

[But in practice it is a little slower and uses little more memory than the one with array-based implementation]

Circular Queue

It has the behaviors (methods) of a singly linked list, plus a rotate() method, to move the first element to the end of the list.

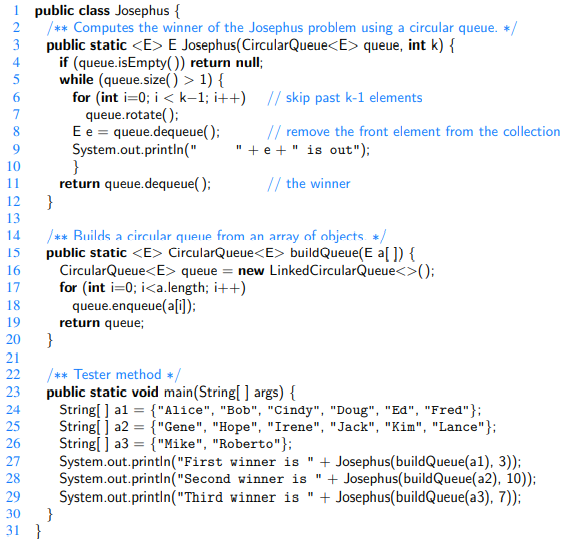


Josephus (hot potato) problem

[Kids, hot potato, they give it around, when bell, the kid who has the potato: out. ]

[Josephus: fixed number k, every k-th kid removed. Determining the winner is the J. problem.]

Using a Circular Queue: rotate k-times, remove the element at the front of the queue.



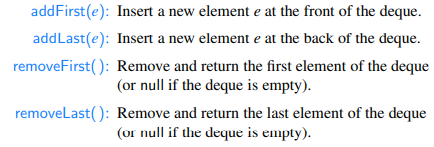
Time complexity: O(nk). [Nem eliras]

Double-Ended Queues (Deque)

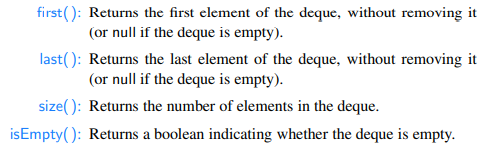
A Deque is a Queue with two ends; supports deletion and insertion at the front and the back of the Queue.

Methods:

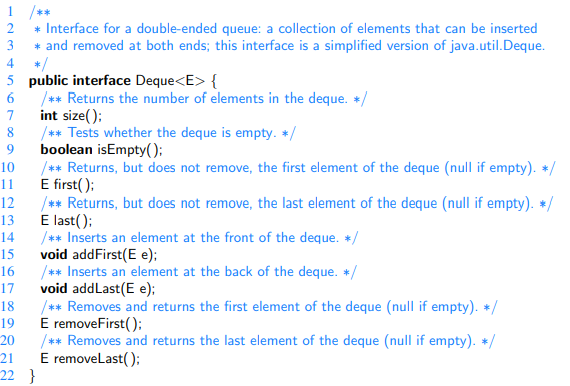
Update methods:



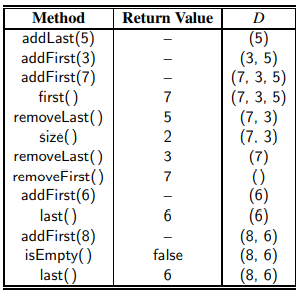
Accessor methods:



Deque interface



Example:



Implementing a Deque

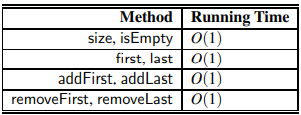
Array-based implementation

Similarly as we did with the queue. Although we need to avoid negative numbers this time, therefore instead of f = (f - 1) % N we use f = (f - 1 + N) % N.

‘Doubly Linked List’ -based implementation

It is basically the same as the Doubly Linked List implementation, we just need to add **implements** Deque<E>.

Running times of the ‘Doubly Linked List’ -based implementation



[Azt se felejtsuk el, hogy mindezek meg vannak irva mar a Java-ban (java.util.\*), tehat ha nincs kikotve, hogy azt nem hasznalhatod, akkor erdemesebb(gyorsabb) azt hasznalni.]