Data Structures

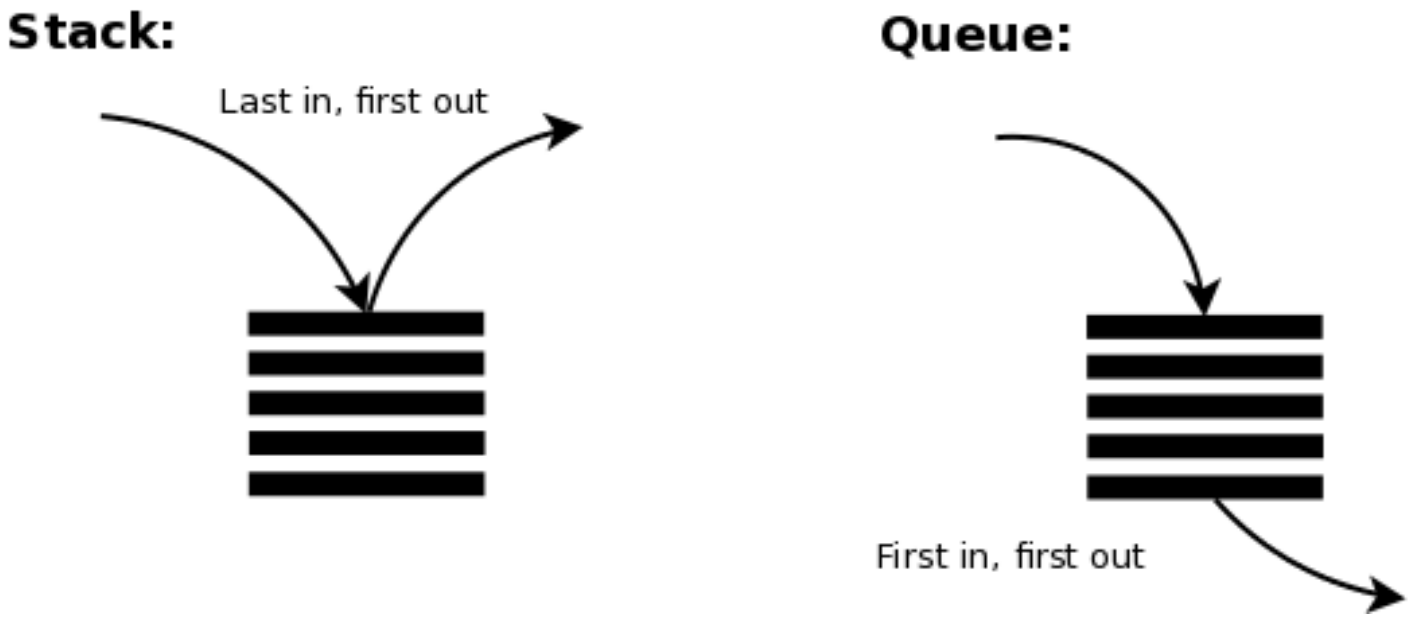
Augustin Cosse.

Spring 2021

March 16, 2021

Stacks, Queues and Deques

- A stack is a collection of objects that are inserted and removed according to the last in first out (LIFO) principle
- A user may insert objects into a stack at any time, but it will only be able to access or remove the most recently inserted element (i.e the element at the top of the stack)



Stacks, Queues and Deques

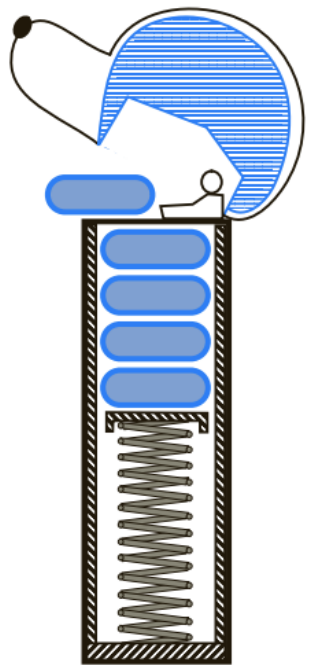
- The "stack" is derived from the metaphor of a stack of plates in a spring loaded cafetaria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack.
- When we need a new plate, we "pop" the top plate off the stack and when we add a plate, we "push" it down on the stack to become the new top plate

Stacks, Queues and Deques

- Stacks are a fundamental data structures many application including Web browsers and text editors
  - Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to pop back to previously visited sites using the "back" button
  - Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states state of a document. This undo operation can be accomplished by keeping text changes into a stack

Stacks, Queues and Deques

- Another example is the PEZ candy dispenser which stores mint candies in a spring loaded container that pops out the topmost candy in the stack when the top of the dispenser is lifted



The stack abstract data type

- Stacks are the simplest of all data structures, et they are also among the most important, as they are used in a host of different applications and as a tool for many more sophisticated data structures and algorithms
- Formally a stack is an abstract data type that supports the following two update methods:
  - push(e): Adds element e to the top of the stack
  - pop(e): Removes and returns the top element from the stack (or null if the stack is empty)

The stack abstract data type

- Additionally, a stack supports the following accessor methods for convenience:
  - top(): Returns the top element of the stack without removing it (or null if the stack is empty)
  - size(): Returns the number of elements in the stack
  - isEmpty(): Returns a boolean indicating whether the stack is empty

  By convention we assume that elements added to the stack can have arbitrary type and that a newly created stack is empty.

The stack abstract data type

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

The stack abstract data type

- In order to formalize our abstraction of a stack, we will first define what is known as its application programming interface (API) in the form of a Java interface which describes the names of the methods that the ADT supports and how they are to be declared and used
- We will again rely on Java's generic framework, allowing the elements stored in the stack to belong to any object type ¡E¿
- As an example, a stack of integers could be declared with type `Stack<Integer>`
- Recall that in Java, an interface serves as a type definition but that it cannot be instantiated.
- For the ADT to be of any use, we must provide one or more concrete classes that implements the methods of the interface.

The stack abstract data type

- The Stack interface is given below

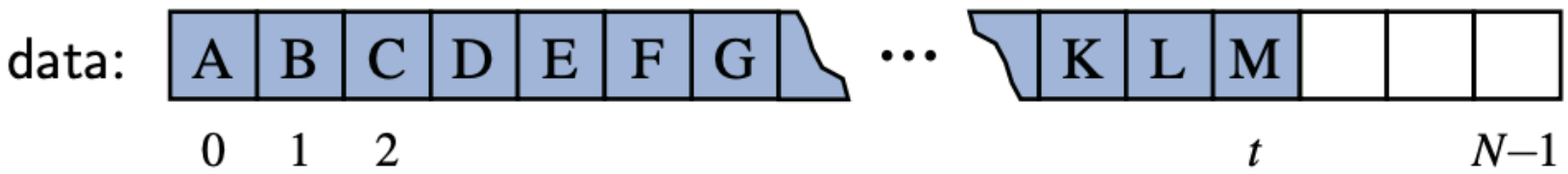
```
public interface Stack<E> {
/**
* Returns the number of elements in the stack.
*/
int size( );
/**Tests whether the stack is empty.*/
boolean isEmpty( );
/**Inserts an element at the top of the stack.*/
void push(E e);
/**Returns, but does not remove top of stack.*/
E top( );
/** Removes and returns top of stack.*/
E pop( );}
```

The stack abstract data type

- Because of the importance of the stack ADT, java has included, since its original version, a concrete class named `java.util.Stack` which implements the LIFO semantics of a stack
- However, Java's stack remains only for historic reasons and the interface is not consistent with most other data structures in the Java library
- In fact the current documentation for the `Stack` class recommends that it not be used as LIFO functionality is provided by a more general data structure known as a double ended queue
- In this course, we will consider two implementation of a stack: the first one using an array for storage and the second one that uses a linked list.

Array based stack implementation

- As a first implementation, we store the elements of the list in an array named `data` with capacity N for some fixed N
- We orient the stack so that the bottom element of the stack is always stored in cell `data[0]` and the top element of the stack is in cell `data[t]` for index t (where t is equal to one less than the current size of the stack)



Array based stack implementation

- A java implementation for such an array based stack can be found below

```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;
//  array capacity
  private E[ ] data; // generic array
  private int t = 1; // top index
  public ArrayStack( ) { this(CAPACITY); }
  public ArrayStack(int capacity) {
    data = (E[ ]) new Object[capacity];
// safe cast; compiler may give warning
}
 /* see next slide */
}}
```

Array based stack implementation

```
public class ArrayStack<E> implements Stack<E> {
/*see previous slide */
 public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == -1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length)
          throw new IllegalStateException("Stack is full");
    data[++t] = e;
}
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
 }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;
    t--;
    return answer;}}
```

Array based stack implementation

- The array implementation of a stack is simple and efficient. Nevertheless this implementation has one negative aspect: it relies on a fixed capacity array which limites the ultimate size of the stack
- For convenience, you can see from the implementation that we let the user specify the capacity as a parameter to the constructor (and also offer a constructor with a default capacity of 1000)
- In case a user has a good estimate of the number of items to go in the stack, the array based implementation is hard to beat

Array based stack implementation

- The correctness of the array based implementation relies on the index t. When pushing an element, t is incremented before placing the new element so that it uses the first available cell

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Implementing a stack with a singly linked list

- Unlike the array based implementation, the linked list approach has memory usage that is always proportional to the number of actual elements currently in the stack yet without an arbitrary capacity limit
- For a design based on linked lists, we need to decide if the top of the stack will be at the from or back of the list
- Since we can insert and delete elements only at the front, this clearly seems to be the best choice
- With the top of the stack stored at the front, all methods execute in constant time

The adapter design pattern

- The adapter design pattern applies to any context where we effectively want tomodify an existing class so that its methods match those of a related, but different class or interface.
- One general way to apply the adapter pattern is to define a new class in such a way that it contains an instanceof the existing class as a hidden field and then implement each method of the new class using methods of this hidden instance variable
- By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class but repackaged in a more convenient way

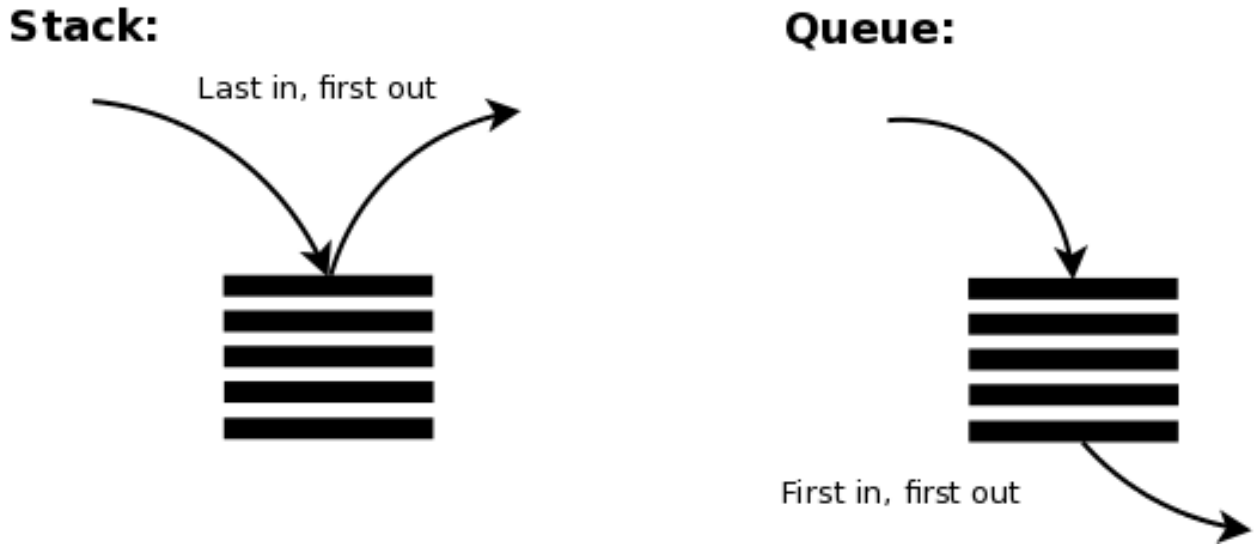
The adapter design pattern

- In the context of the Stack ADT, we can adapt our `SinglyLinkedList` class to define a new `LinkedStack` class shown below. The class declares a `SinglyLinkedList` named list as a private field.

```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>( );
  public LinkedStack( ) { } // new stack
  public int size( ) { return list.size( ); }
  public boolean isEmpty( ) { return list.isEmpty( ); }
  public void push(E element) { list.addFirst(element); }
  public E top( ) { return list.first( ); }
  public E pop( ) { return list.removeFirst( ); }
}
```

Queues

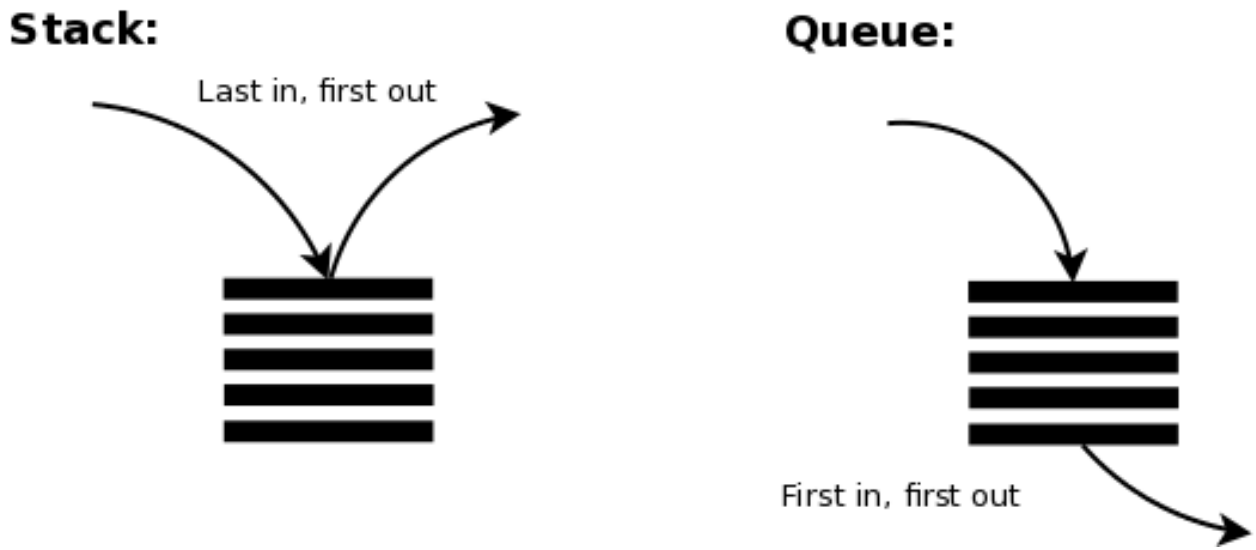
- Another fundamental data structure is the queue. A queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle
- Elements can be inserted at any time, but only the element that has been in the queue the longest can be removed first
- Elements enter a queue at the back and are removed from the front.



Queues

- A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the from of the line

- A queue is therefore a logical choice for a data structure to handle calls to a customer service or a wait-list at a restaurant. FIFO queues are also used by many computing devices such as networked printers or Web servers responding to requests.



Queues

- Formally the queue abstract defines a collection that keeps objects in a sequence, where elements access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence
- This restriction enforces the rule that are inserted and deleted in the queue according to the First in First out principle
- The queue abstract data type supports the following two update methods:
  - enqueue(e): Adds element e to the back of the queue
  - dequeue(): Removes and returns the first element from the queue (or null if the queue is empty)

Queues

- The queue ADT also includes the following accessor methods (with first being analogous to the stack's top method)
  - first(): Returns the first element of the queue, without removing (or null if the queue is empty)
  - size(): Returns the number of elements in the queue
  - isEmpty(): Returns a boolean indicating whether the queue is empty
- By convention, we will again assume that elements added to the queue can have arbitrary type and that a newly created queue is empty

Queues

- The queue ADT is formalized through the Java Interface shown below

```
public interface Queue<E> {
/**  Returns number of elem*/
int size( );
/** Tests whether queue is empty. */
boolean isEmpty( );
/** Inserts an element at the rear of the queue. */
void enqueue(E e);
/** Returns, but does not remove, first elem */
E first( );
/** Removes and returns first elem*/
E dequeue( );}
```

Queues

Method	**Return Value**	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size( )	2	(5, 3)
dequeue( )	5	(3)
isEmpty( )	false	(3)
dequeue( )	3	( )
isEmpty( )	true	( )
dequeue( )	null	( )
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first( )	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

Queues

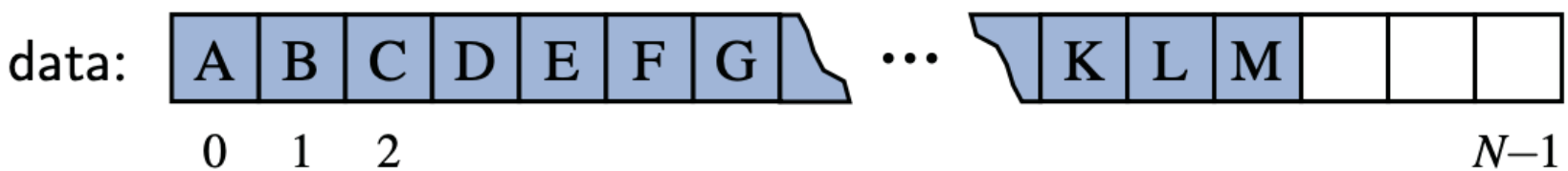
- As for the stacks, Java provides a type of queue interface, `java.util.Queue` which has functionalities similar to the ADT given in the previous slide.
- The `java.util.Queue` ADT supports two styles for most operations wich vary in the way that treat exceptional cases
- When a queue is empty, the `remove()` and `element()` methods throw a `NoSuchElementException` while the corresponding `poll()` and `peek()` return **null**
- For implementation with a bounded capacity, the `add` method will throw an `IllegalStateException` when full, while the `offer` method ignores the new element and returns **false** to signal that this new element was not accepted

Queues

Our Queue ADT	**Interface java.util.Queue**	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Array based Queue implementation

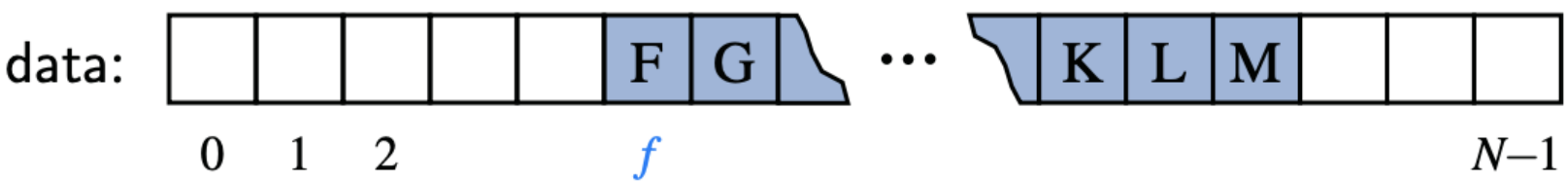
- Just as we implemented the LIFO semantics of the stack ADT using an array, we can also use arrays to efficiently support the FIFO semantics of the Queue ADT.
- Let us assume that elements are inserted into a queue. We store them in an array such that the first element is at index 0, the second is at index 1 and so on
- With such a convention, how do we implement the `dequeue` operation?



Array based Queue implementation

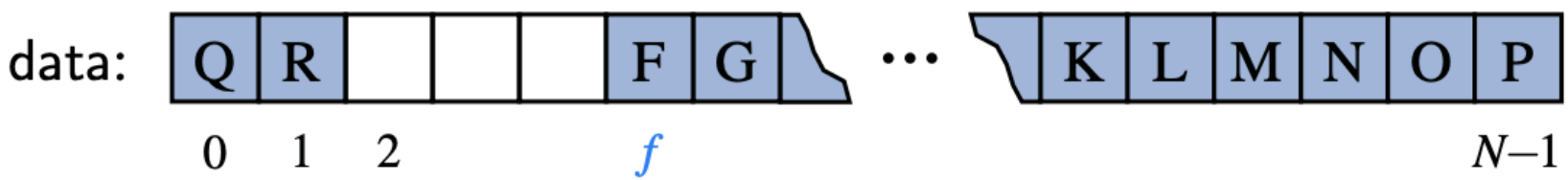
- We can decide to store the element to be removed at position 0 in the array. In this case a strategy is then to execute a loop to shoft all other elements of the queue one cell to the left so that the front of the queue is again aligned with the cell 0 of the array. The use of such a for loop would result in $O(n)$ complexity for the dequeue method though

- An alternative is to replace the dequeued element in the array with a **null** reference and maintain a f variable to represent the element that is currently at the front of the queue. Such an algorithm for dequeue would run in $O(1)$



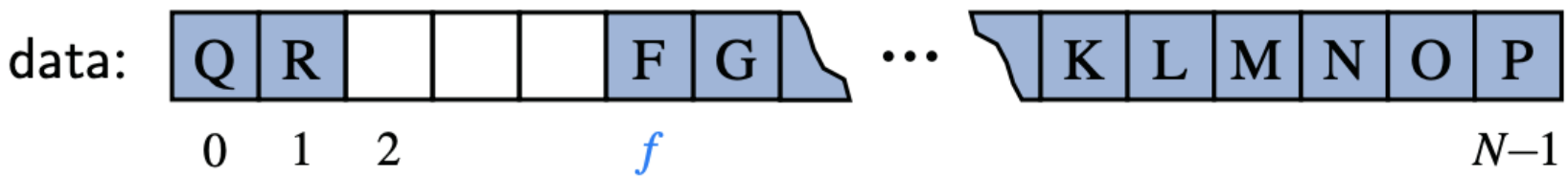
Array based Queue implementation

- There remains a challenge with the revised approach though: with an array of capacity N, we should be able to store up to N elements before reaching any exceptional case.
- If we repeatedly let the front of the queue drift rightward over time, the back of the queue will reach the end of the underlying array even when there are fewer than N elements currently in the queue.
- We must decide how to store additional elements using such a configuration



Array based Queue implementation

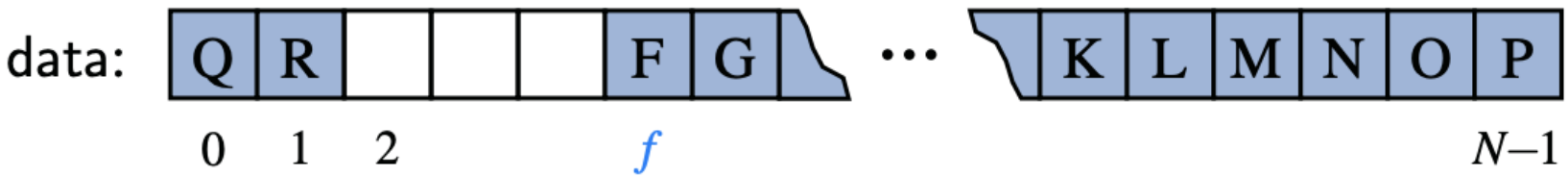
- A possible solution could be to allow both the front and the back of the queue to drift backward with the back of the queue "wrapping around" the en of the array
- Assuming that the array has length N, new elements are enqueued toward the "end" of the current queue, progressing from the front to index $N-1$ and continuing at index 0 then index 1
- This idea is illustrated below for a queue with first element F and last element R



Array based Queue implementation

- Implementing such a circular queue is relatively easy with the modulo operator (denoted with the symbol $\%$ in java). Recall that the modulo operator is computed by by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2 so that $14\%3$ evaluates to the remainder 2

- The modulo operator is ideal for treating an array circularly. When we dequeue an element, we use the arithmetic $f = (f + 1)\%N$. As a concrete example, if we have an array of length 10, and a front index 7, we can advance the front by formally computing $7 + 1\%10$ which is simply 8 as 8 divided by 10 is 0 with a remainder of 8



Array based Queue implementation

- A complete implementation of a Java queue ADT is given below

```
public class ArrayQueue<E> implements Queue<E> {
// instance variables
  private E[ ] data; // storage array
  private int f = 0; // front element
  private int sz = 0; // number of elements
  // constructors
  public ArrayQueue( ) {this(CAPACITY);} // cons. 1
  public ArrayQueue(int capacity) { // cons. 2
    data = (E[ ]) new Object[capacity];
   // safe cast; warning}

/* see next slide */
}
```

Array based Queue implementation

```
public class ArrayQueue<E> implements Queue<E> {
/* see previous slide */
public int size( ) { return sz; }
/** Tests whether the queue is empty. */
public boolean isEmpty( ) { return (sz == 0); }
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws
       IllegalStateException {
  if (sz == data.length) throw
          new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;
  data[avail] = e;
  sz++;}
/** Returns, but does not remove, first elem. */
public E first( ) {
  if (isEmpty( )) return null;
  return data[f];}}
```

Array based Queue implementation

- Internally the queue class maintains three variables: data: a reference to the underlying array, f, a integer that represents the index, within array data of the first element of the queue. sz: an integer representing the current number of elements stored in the queue.

```
public class ArrayQueue<E> implements Queue<E> {
/* see previous slides */
public E dequeue( ) {
  if (isEmpty( )) return null;
  E answer = data[f];
  data[f] = null; // dereference
  f = (f + 1) % data.length;
  sz--;
  return answer;}
```

Implementing a queue with a singly linked list

- As we did for the stack ADT, one can easily adapt a singly linked list to implement the queue ADT while supporting worst case O(1) tiime for all operations, and without any artificial limit for the capacity.

```
/** FIFO queue as SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list
          = new SinglyLinkedList<>( ); // empty list
  public LinkedQueue( ) { } // new queue = empty list
  public int size( ) { return list.size( ); }
  public boolean isEmpty( ) { return list.isEmpty( ); }
  public void enqueue(E element) { list.addLast(element); }
  public E first( ) { return list.first( ); }
  public E dequeue( ) { return list.removeFirst( ); }
}
```

Implementing a queue with a singly linked list

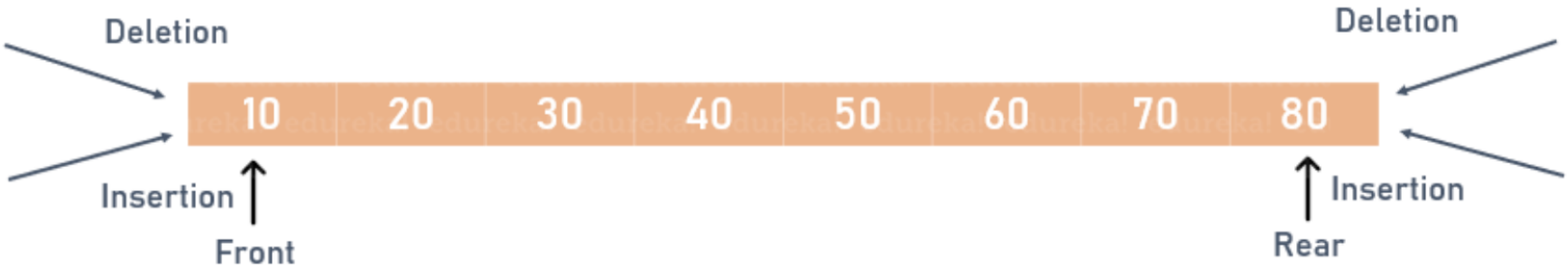
- The natural orientation for a queue is to align the from of the queue with the from of the list and the back of the queue with the tail of the list because the only update operation that singly linked lists support at the back end is an insertion.

```
/** FIFO queue as SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list
          = new SinglyLinkedList<>( ); // empty list
  public LinkedQueue( ) { } // new queue = empty list
  public int size( ) { return list.size( ); }
  public boolean isEmpty( ) { return list.isEmpty( ); }
  public void enqueue(E element) { list.addLast(element); }
  public E first( ) { return list.first( ); }
  public E dequeue( ) { return list.removeFirst( ); }
}
```

Double ended queues

- Double Ended Queues (or Deque) are queue like structures that support insertion and deletion at both the front and the back of the queue

- Note that Deque in this framework is usually pronounced "deck" to avoid confusion with the dequeue method of the regular queue ADT which is pronounced like the abbreviation "D.Q"

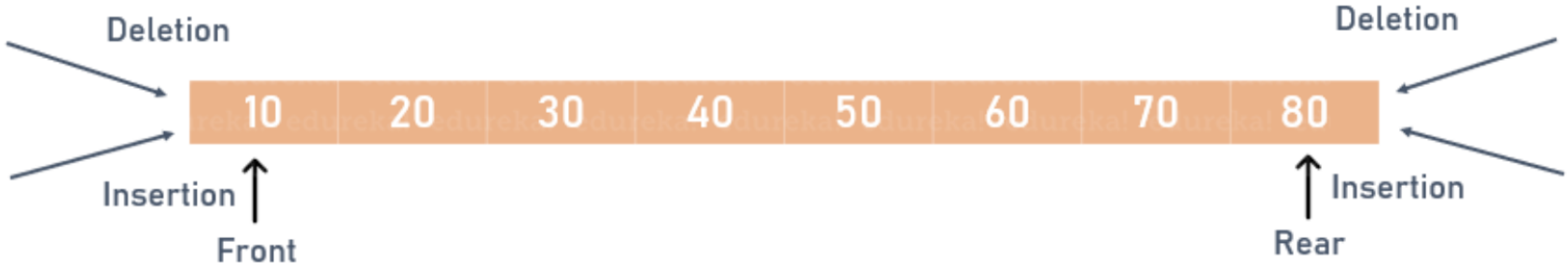


Double ended queues

- The Deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications. For example we might want to describe a restaurant using a queue to maintain a waitlist.

- Occasionally, the first person might be removed fom the queue only to find that a table was not available; typically, the restaurant will reinsert the person at the first position in the queue.

- It may also be that a customer at the end of the queue may grow impatient and leave the restaurant. (We will need an even more general data structure if we want to model customers leaving the queue from other positions)



The Deque abstract data type

- One can formalize the deque ADT with the following Java interface

```
public interface Deque<E> {
/** Returns num of elem. in deque. */
int size( );
/** Tests whether the deque is empty. */
boolean isEmpty( );
/** Returns first/last elem of deque */
E first( );
 E last( );
/** Inserts an element at the front/back */
void addFirst(E e);
void addLast(E e);
/** Removes and returns first/last elem */
E removeFirst( );
E removeLast( );
}
```

The Deque abstract data type

- The deque abstract data type is usually defined to support the following update methods:

addFirst(e)	Insert a new element e at the front of the queue
addLast(e)	Insert a new element e at the back of the deque
removeFirst()	Remove and return first element of the deque
removeLast()	Remove and return last element of the deque
first()	Return first element of the deque without removing
last()	Return last element of the deque without removing
size()	Returns num of elements in deque
isEmpty	returns a boolean indicating whether deque is empty