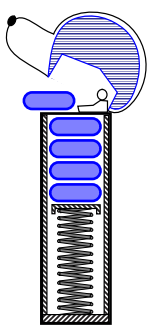
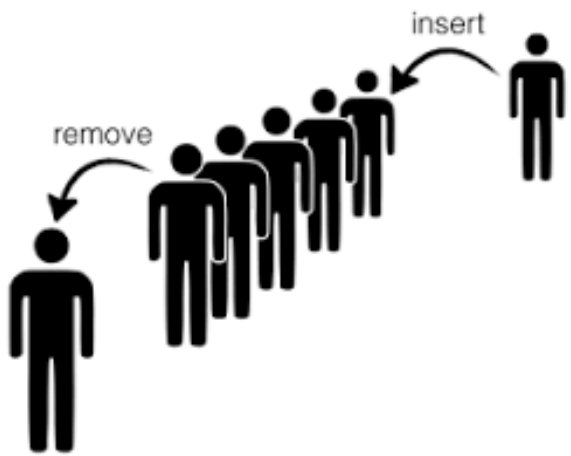
Stacks and Queues

Jianguo Lu

February 5, 2025



vs.



Overview

Data Type and Abstract Data Type

- Data type
  - Data values

Data Type and Abstract Data Type

- Data type
  - Data values
  - Operations on the data
- Abstract

Data Type and Abstract Data Type

- Data type
  - Data values
  - Operations on the data
- Abstract
  - Focus on some details while ignore others.
  - Simplified description of objects
  - Emphasis significant information only
  - Suppress irrelevant information
- Abstract Data Type

Data Type and Abstract Data Type

- Data type
  - Data values
  - Operations on the data
- Abstract
  - Focus on some details while ignore others.
  - Simplified description of objects
  - Emphasis significant information only
  - Suppress irrelevant information
- Abstract Data Type
  - Focus on operations, ignore the concrete data representation.

ADT

- Important programming concepts introduced in the 1970s.
- Separation of the **use** of the data type from its **implementation**
- Users are concerned with the interface, but not the implementation
  - Implementation can change in the future.
- This supports the principle of information hiding.
- Protecting the program from design decisions that are subject to change.

Example Stack ADT

boolean empty() Tests if this stack is empty.

E peek() Looks at the object at the top of this stack without removing it.

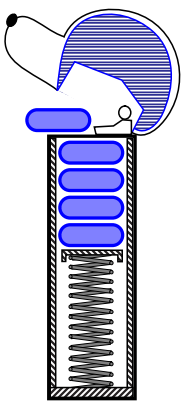
E pop() Removes the object at the top of this stack and returns it.

E push(E item) Pushes an item onto the top of this stack.

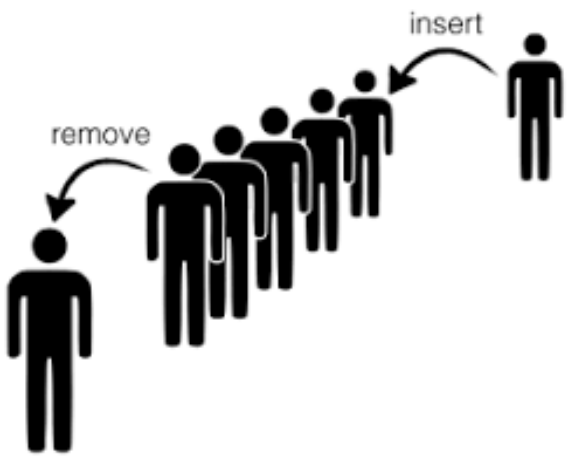
- For user of the ADT: ADT determines what operations can be done to a variable
- For implementers of the ADT: ADT can be implemented in different ways.

Stack vs. Queue

- Two closely-related data types for manipulating arbitrarily large collections of objects
- Stacks and queues are special cases of the idea of a collection.
- Each is characterized by four operations:
  - create the collection,
  - insert an item,
  - remove an item, and
  - test whether the collection is empty.



Stack

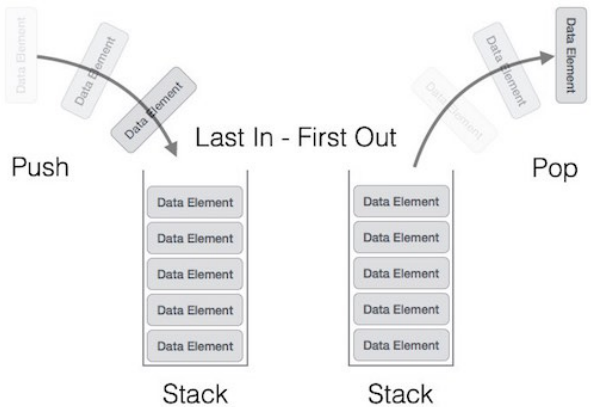


Queue

Stack

Stack

A stack is a collection that is based on the last-in-first-out (LIFO) policy.



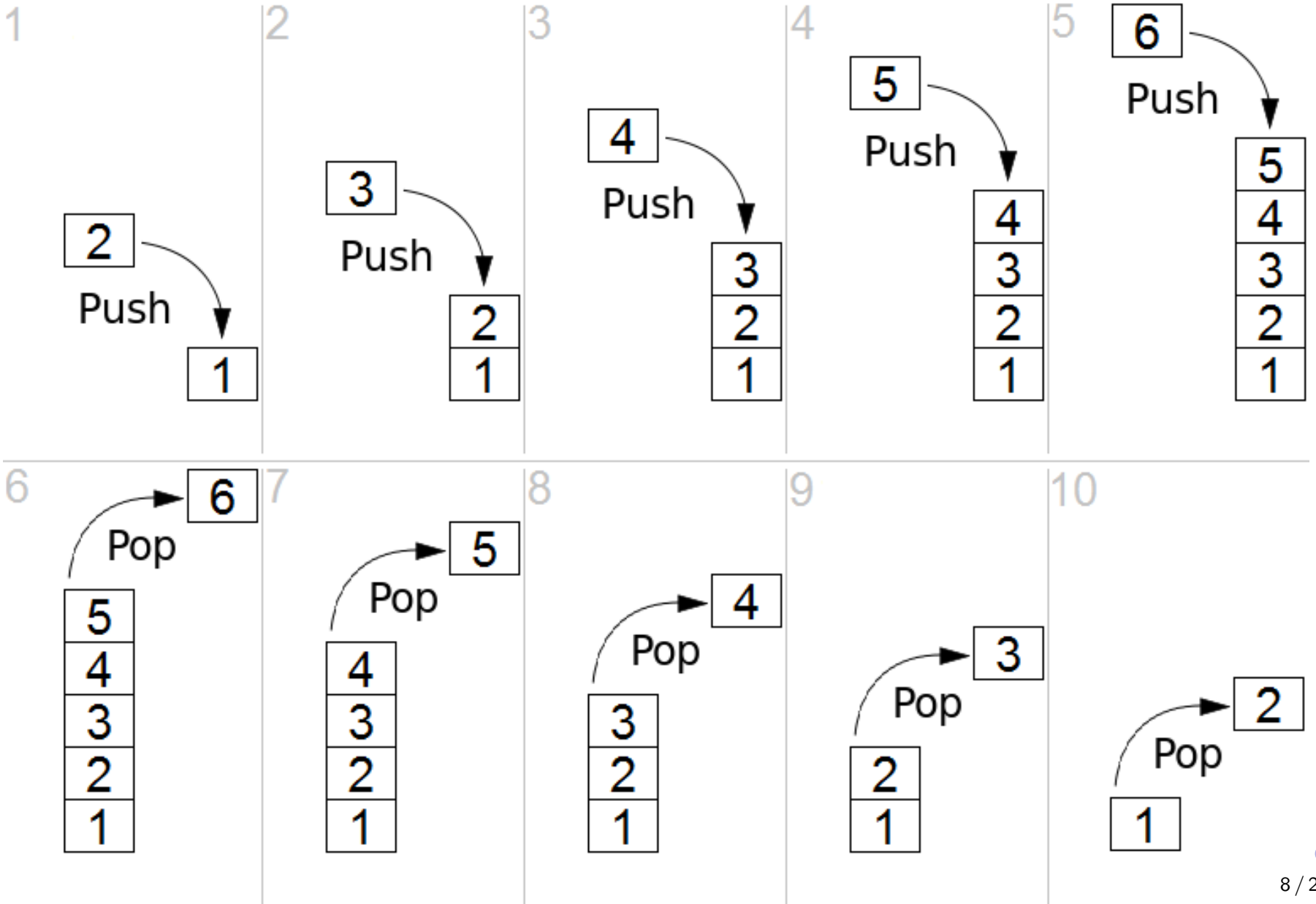

```
public class Stack<Item> implements Iterable<Item>
```

	`Stack()`	*create an empty stack*
`boolean`	`isEmpty()`	*is the stack empty?*
`void`	`push(Item item)`	*push an item onto the stack*
`Item`	`pop()`	*return and remove the item that was inserted most recently*
`int`	`size()`	*number of items on stack*

Stack operations

Method	Return Value	Stack content
push(5)	–	(5)
push(3)	–	(5,3)
size()	2	(5,3)
pop()	3	(5)
pop()	5	()
push()	7	(7)
push()	9	(7,9)

Another tracing of the operations–LIFO



$\emptyset$

Applications of Stack data structure

- Direct applicationsø
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Recursion stack
- Indirect applications
  - Auxiliary data structure for algorithms
    - e.g., Depth First Search
  - Component of other data structures

String Reverse Example

The task: Transform an array of strings from

```
["Jack", "Kate", "Hurley", "Jin", "Michael"]
```

into

```
["Michael", "Jin", "Hurley", "Kate", "Jack"]
```

String Reverse Example

The task: Transform an array of strings from

```
["Jack", "Kate", "Hurley", "Jin", "Michael"]
```

into

```
["Michael", "Jin", "Hurley", "Kate", "Jack"]
```

```
static void reverse(String[] a) {
   Stack<String> stack = new Stack<String>();
   for (int i=0; i < a.length; i++)
     stack.push(a[i]);
   for (int i=0; i < a.length; i++)
     a[i] = stack.pop();
 }
}
```

Stack application example: Matching parentheses

```
validStrings=
    "[()]",
    "( ) ( ( ) )  ",
    "((()(())){([()])}))",
    "[(5+x)-(y+z)]"

  invalidStrings =
    "([)]",
    "({[])}",
    "("
```

Steps	Stack content
push('[')	[
push('(')	[, (
')' matches the top, pop()	[
']' matches the top, pop()	empty

Matching parentheses

```
boolean isMatched(String expression) {
    final String opening  = "({[";
    final String closing  = ")}]";
    Stack<Character> buffer = new Stack<>();
    for (char c : expression.toCharArray()) {
      if (opening.indexOf(c) != -1)   buffer.push(c);
      else if (closing.indexOf(c) != -1) {
        if (buffer.isEmpty()) return false;
        if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
          return false;
      }
    }
    return buffer.isEmpty();
  }
```

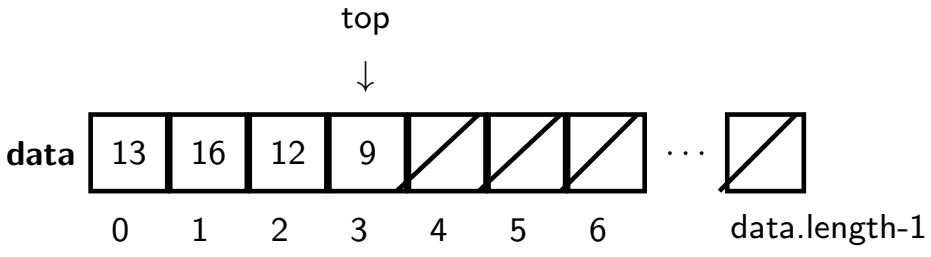
Steps	Stack content
push('[')	[
push('(')	[, (
')' matches the top, pop()	[
']' matches the top, pop()	empty

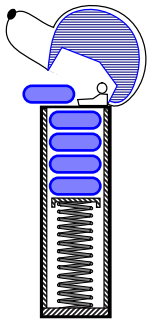
Stack implementation

Stack can be implemented using

- Array
- LinkedList

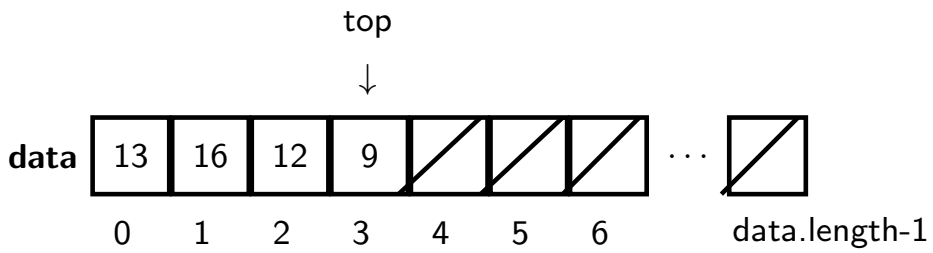
Stack implementation using Array





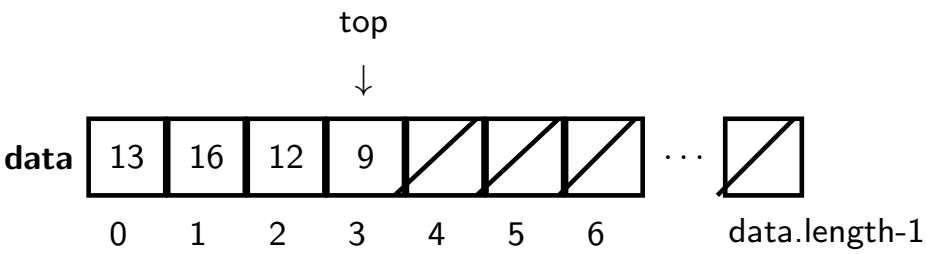
```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000; // default array capacity
  private E[ ] data; // generic array used for storage
  private int t=-1;
      ...
   public int size( ) { return (t + 1); }
   public boolean isEmpty() {return (t==-1); }


   public void push(E e)  { ...}
   public E pop() { ...}
   public E top() {...}
 }
```

Operation top()–Peek the top element



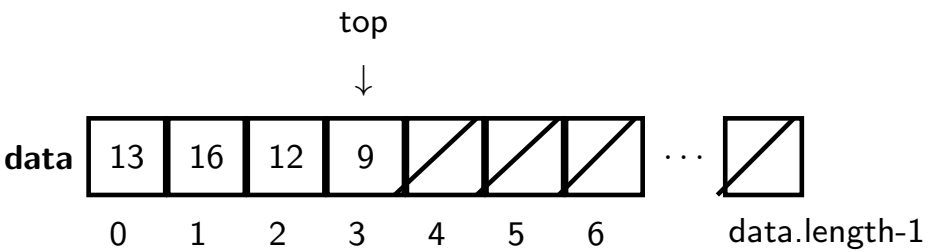

```
public E top( ) {
   if (isEmpty( )) return null;
    return data[t];
 }
```

pop() operation: remove the top element




```
public E pop() {
  if (isEmpty()) return null;
  E answer = data[t];
  data[t] = null;
  t--;
  return answer;
}
```

Implementation of Stack: PUSH operation




```
public void push(E e) throws IllegalStateException {
  if (size() == data.length) throw new
       IllegalStateException("Stack is full");
  top++;
  data[top] = e;
}
```

What if stack length is longer than array length?

- Implement stack using LinkedList
- Dynamically resize the array

Resize array

- Increase the array size:

```
public void push(E e) {
        if (size() == data.length) resize(2*data.length);
        top++;
        data[top]=e;
}

 private void resize(int capacity) {
        assert capacity >= n;
        String[] temp = new String[capacity];
        for (int i = 0; i < n; i++)
            temp[i] = data[i];
        data = temp;
  }
```

- Decrease the array size:

```
public String pop() {
   if (isEmpty()) throw new Exception("Stack underflow");
   E e = data[t];
   data[t] = null;
   t--;
   if (t > 0 && t == data.length/4) resize(data.length/2);
   return e;
}
```

Resize array

- Increase the array size:

```
public void push(E e) {
        if (size() == data.length) resize(2*data.length);
        top++;
        data[top]=e;
}

 private void resize(int capacity) {
        assert capacity >= n;
        String[] temp = new String[capacity];
        for (int i = 0; i < n; i++)
            temp[i] = data[i];
        data = temp;
  }
```

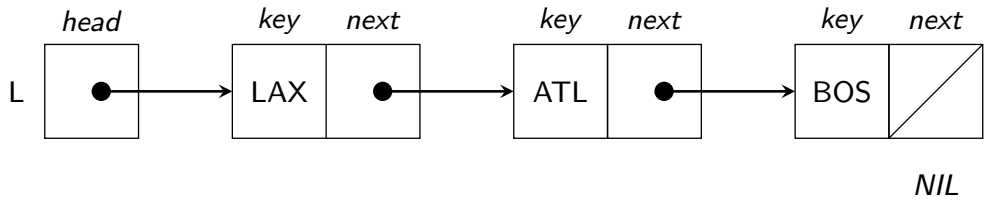
- Decrease the array size:

```
public String pop() {
   if (isEmpty()) throw new Exception("Stack underflow");
   E e = data[t];
   data[t] = null;
   t--;
   if (t > 0 && t == data.length/4) resize(data.length/2);
   return e;
}
```

- What is the complexity of resizing?

LinkedList Implementation

- LinkedList implementation
- Adapted from SinglyLinkedList
- so-called Adapter design pattern




```
public class LinkedStack<E> implements Stack<E> {
 private SinglyLinkedList<E> list = new SinglyLinkedList<>( );
 public LinkedStack( ) { }
 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( ); }
```

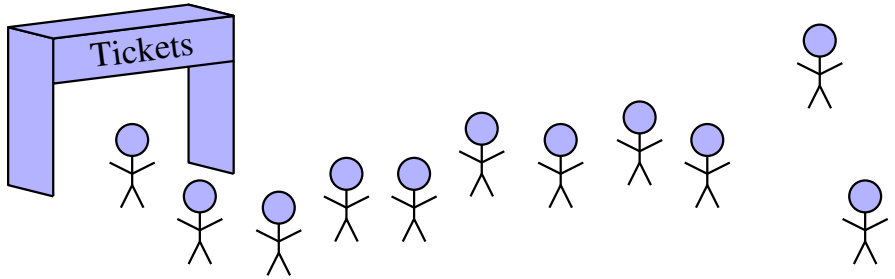
Comparison on space consumption: Array vs Doubly LinkedList

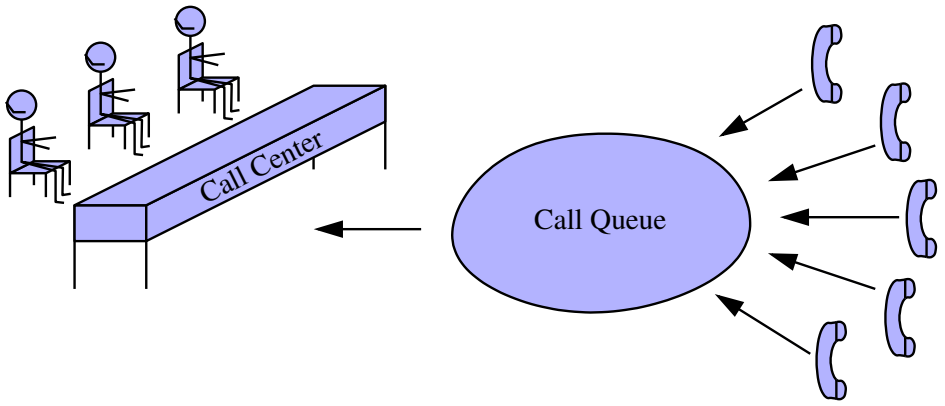
.

- Each list element requires a Node instance.
  - 32bit reference to data element,
  - 32bit reference to next Node,
  - 32bit reference to previous Node.
  - 64bit object header
- At least five 32-bit words

Depends on JVM. In 64-bit JVM, object header is 128 bits.

Examples of queues





Queue

- A queue is another linear data structure.
- Unlike stacks, queues follow a FIFO protocol which stands for "first in, first out".
- The element that has been in the list the longest will be the first to leave the list.

Queue ADT

Queue ADT

- Main queue operations:

  enqueue(object) : inserts an element at the end of the queue
  object dequeue() : removes and returns the element at the front of the queue

- Auxiliary queue operations:

  object front() : returns the element at the front without removing it
  integer size() : returns the number of elements stored
  boolean isEmpty() : indicates whether no elements are stored

Compare with the Stack ADT:

Stack ADT

E push(E item)	Pushes an item onto the top of this stack.
E pop()	Removes the object at the top of this stack and returns it.
boolean empty()	Tests if this stack is empty.
E top()	Looks at the object at the top of this stack without removing it.

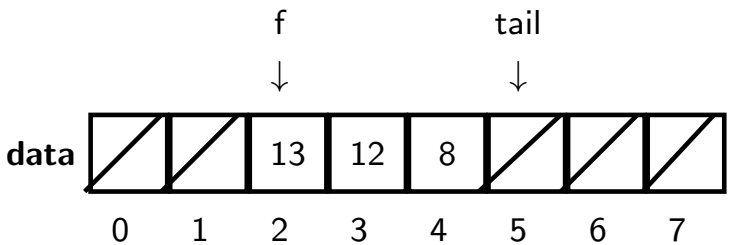
Queue operations

Operation	Output	Q
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	5	(3)
enqueue(7)	–	(3, 7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	“error”	()
isEmpty()	true	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	2	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

Implementation of Queue using Array

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - front: index of the front element
  - size: size of the queue
- The tail position can be calculated by

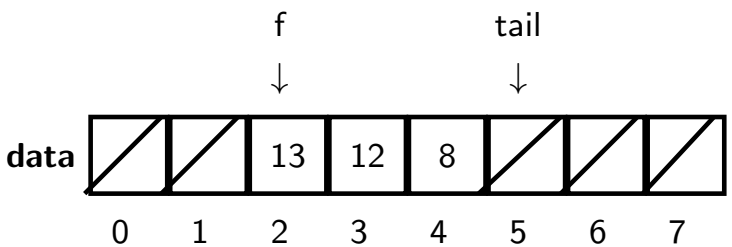
$$tail = \begin{cases} f + sz, & if\, f + sz < N \\ f + sz - N, & if\, f + sz \geq N \end{cases} \quad (1)$$




```
public void enqueue(E e) {
   if (sz == data.length) error();
   int avail = (f + sz) % data.length;
   data[avail] = e;
   sz++;
 }
```

Queue: remove the front element

```
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```



Implement using LinkedList

```
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public LinkedQueue( ) { }
  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( ); }
  }
```

Efficient analysis

- Each method is $O(1)$
- No re-sizing of array
- Each call has several primitive operations

Takeaways

- What is ADT
- What is a Stack (FILO)
- What is a Queue (FIFO)
- The ADTs for Stack and Queue
- Implementation of Stack and Queue
  - use Array
  - use LinkedList
- Example applications of Stack
  - Reverse an Array, a string ...
  - Matching brackets, HTML tags, ...
- Readings: Goodrich et al. Chapter 6.