More Data Structures (Part 2)

Queues

Queue

Queue

Queue Operations

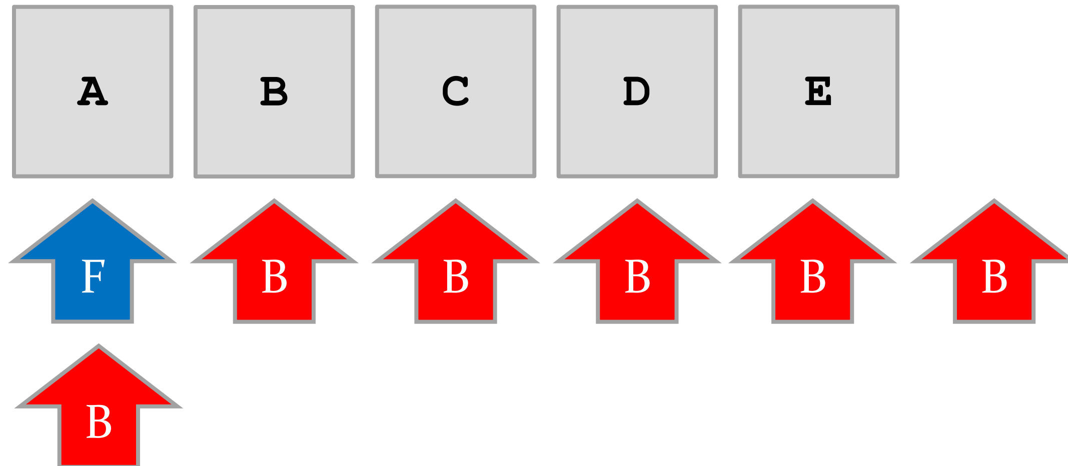
- classically, queues only support two operations
  1. enqueue
     - add to the back of the queue
  2. dequeue
     - remove from the front of the queue

Queue Optional Operations

- optional operations
  1. size
     - number of elements in the queue
  2. isEmpty
     - is the queue empty?
  3. peek
     - get the front element (without removing it)
  4. search
     - find the position of the element in the queue
  5. isFull
     - is the queue full? (for queues with finite capacity)
  6. capacity
     - total number of elements the queue can hold (for queues with finite capacity)

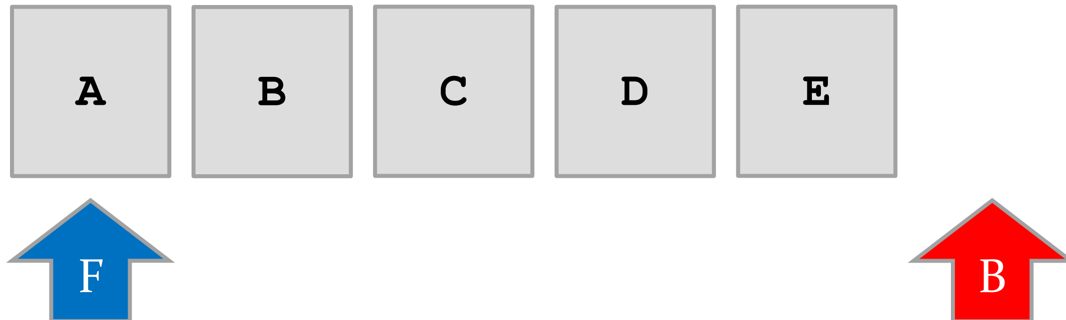
Enqueue

1. `q.enqueue("A")`
2. `q.enqueue("B")`
3. `q.enqueue("C")`
4. `q.enqueue("D")`
5. `q.enqueue("E")`



Dequeue

1. `String s = q.dequeue()`

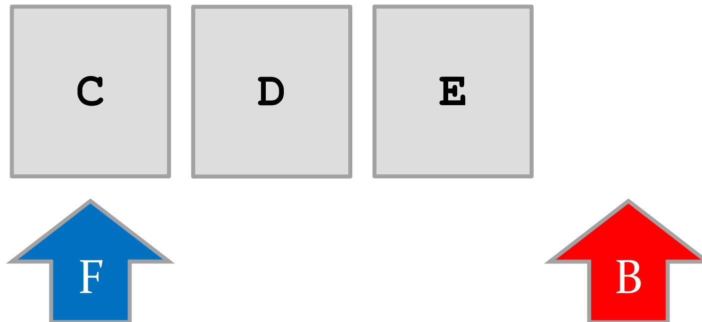


Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`

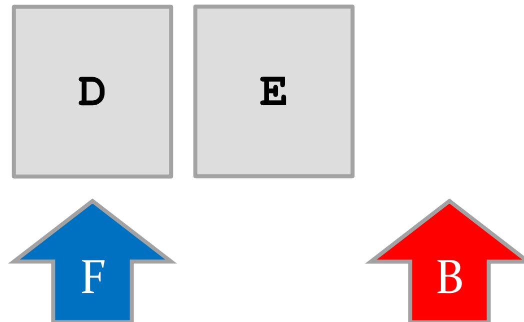
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`



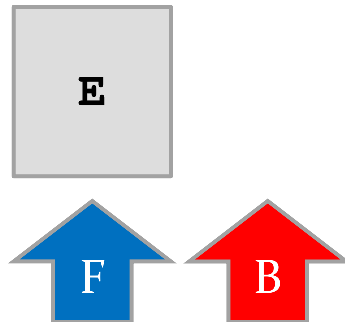
Dequeue

```
1. String s = q.dequeue()
2. s = q.dequeue()
3. s = q.dequeue()
4. s = q.dequeue()
```



Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`
5. `s = q.dequeue()`



FIFO

- queue is a First-In-First-Out (FIFO) data structure
  - the first element enqueued in the queue is the first element that can be accessed from the queue

Implementation with LinkedList

- a linked list can be used to efficiently implement a queue as long as the linked list keeps a reference to the last node in the list
  - required for enqueue
- the head of the list becomes the front of the queue
  - removing (dequeue) from the head of a linked list requires O(1) time
  - adding (enqueue) to the end of a linked list requires O(1) time if a reference to the last node is available

- java.util.LinkedList is a doubly linked list that holds a reference to the last node

```
public class Queue<E> {
  private LinkedList<E> q;

  public Queue() {
    this.q = new LinkedList<E>();
  }

  public enqueue(E element) {
    this.q.addLast(element);
  }

  public E dequeue() {
    return this.q.removeFirst();
  }
}
```

Implementation with LinkedList

- note that there is no need to implement your own queue as there is an existing interface
  - the interface does not use the names enqueue and dequeue however

java.util.Queue

```
public interface Queue<E>
  extends Collection<E>
```

`boolean`	**`add(E e)`** Inserts the specified element into this queue…
`E`	**`remove()`** Retrieves and removes the head of this queue…
`E`	**`peek()`** Retrieves, but does not remove, the head of this queue…

- plus other methods
  - http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

java.util.Queue

- **`LinkedList`** implements **`Queue`** so if you ever need a queue you can simply use:
  - e.g. for a queue of strings

```
Queue<String> q = new LinkedList<String>();
```

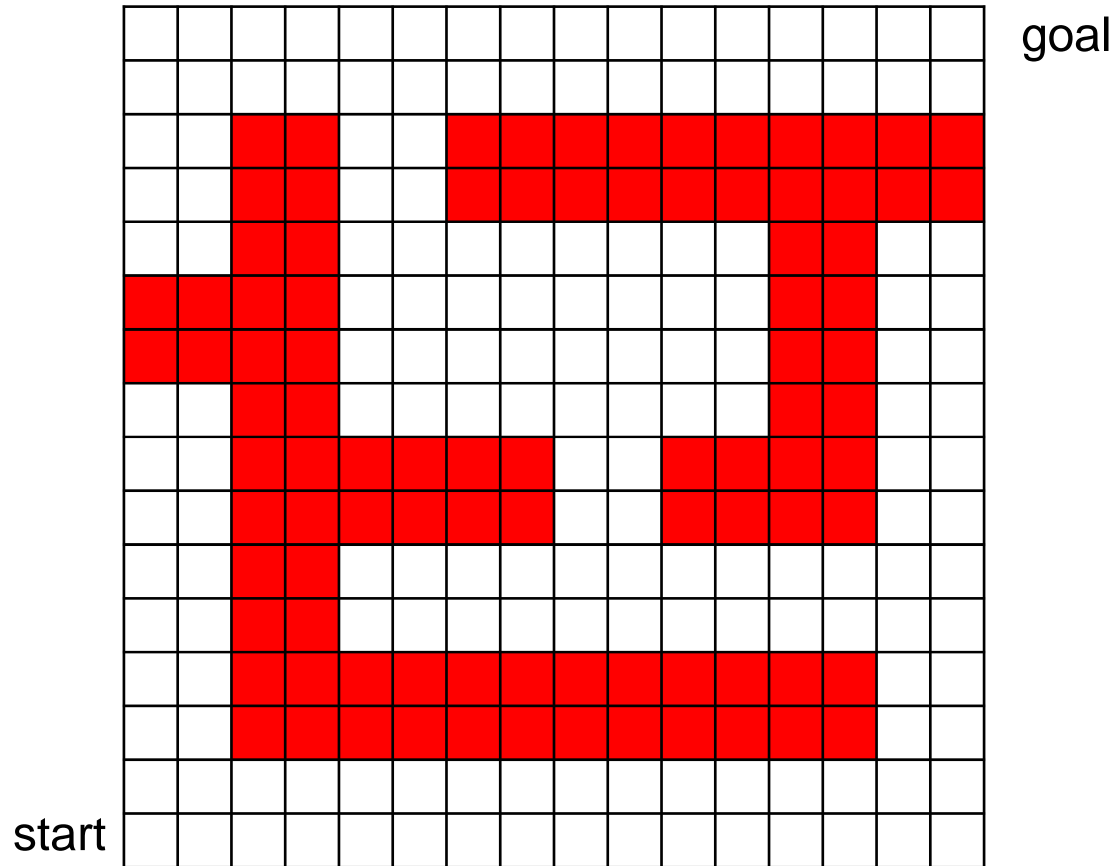
Queue applications

- queues are useful whenever you need to hold elements in their order of arrival
  - serving requests of a single resource
    - printer queue
    - disk queue
    - CPU queue
    - web server

Robotics example

- in robotics, the path planning problem is
  - given a map of the environment, find a path between the starting point of the robot and a goal location that does not pass through any obstacles
- one approach is to use a grid for the map
  - the robot can move from a square on to any adjacent square; i.e., up, down, left, right
  - in the following figures:
    - white square = free space; i.e., the robot can move on to the square
    - red square = obstacle; i.e., the robot cannot move on to the square

Grid-based map



Wave-front planner

- the wave-front planner finds a path between a start and goal point in spaces represented as a grid where
  - free space is labeled with a 0
  - obstacles are labeled with a 1
  - the goal is labeled with a 2
  - the start is known

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

start

Wave-front planner

- starting with the goal cell

```
label L = 2
while start cell is unlabelled {
  for each cell C with label L {
    for each cell Z connected to C with label 0 {
      label Z with L+1
    }
  }
  L = L + 1
}
```

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
goal	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	0	0	0	14	13	12	11	10	9	8	7	6	5	4	3	2
	0	0	0	0	14	13	12	11	10	9	8	7	6	5	4	3
	0	0	1	1	0	14	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	0	20	1	1	17	16	17	18	19	20	0	0	1	1	0	0
	1	1	1	1	18	17	18	19	20	0	0	0	1	1	0	0
	1	1	1	1	19	18	19	20	0	0	0	0	1	1	0	0
	0	0	1	1	20	19	20	0	0	0	0	0	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
start	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	21	20	1	1	17	16	17	18	19	20	21	22	1	1	37	38
	1	1	1	1	18	17	18	19	20	21	22	23	1	1	36	37
	1	1	1	1	19	18	19	20	21	22	23	24	1	1	35	36
	0	0	1	1	20	19	20	21	22	23	24	25	1	1	34	35
	0	0	1	1	1	1	1	1	23	24	1	1	1	1	33	34
	0	0	1	1	1	1	1	1	24	25	1	1	1	1	32	33
	0	0	1	1	29	28	27	26	25	26	27	28	29	30	31	32
	0	0	1	1	30	29	28	27	26	27	28	29	30	31	32	33
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	33	34
	0	49	1	1	1	1	1	1	1	1	1	1	1	1	34	35
	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	36
start	*	49	48	47	46	45	44	43	42	41	40	39	38	37	36	37

Wave-front planner

- to generate a path starting from the start point

```
L = start point label
while not at the goal {
  move to any connected cell with label L-1
  L = L-1
}
```

Wave-front planner

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
	19	18	1	1	15	14	1	1	1	1	1	1	1	1	1	1
	20	19	1	1	16	15	1	1	1	1	1	1	1	1	1	1
	21	20	1	1	17	16	17	18	19	20	21	22	1	1	37	38
	1	1	1	1	18	17	18	19	20	21	22	23	1	1	36	37
	1	1	1	1	19	18	19	20	21	22	23	24	1	1	35	36
	0	0	1	1	20	19	20	21	22	23	24	25	1	1	34	35
	0	0	1	1	1	1	1	1	23	24	1	1	1	1	33	34
	0	0	1	1	1	1	1	1	24	25	1	1	1	1	32	33
	0	0	1	1	29	28	27	26	25	26	27	28	29	30	31	32
	0	0	1	1	30	29	28	27	26	27	28	29	30	31	32	33
	0	50	1	1	1	1	1	1	1	1	1	1	1	1	33	34
	50	49	1	1	1	1	1	1	1	1	1	1	1	1	34	35
	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	36
start	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	37

Breadth-first search

- the wave-front planner is actually a classic computer science algorithm called breadth-first search
  - for grid-based maps, the grid can be viewed as a *graph*
  - you learn about graphs in CSE2011
    - the trees we have been studying are a kind of graph called a *directed acyclic graph*
- breadth-first search can also be used to traverse a tree
  - visiting every node of a tree using breadth-first search results in visiting nodes in order of their level in the tree

50

27 73

8 44 83

73 93

BFS: 50

BFS: 50, 27, 73

50

27

73

8

44

83

73

93

BFS: 50, 27, 73, 8, 44, 83

BFS: 50, 27, 73, 8, 44, 83, 73, 93

Breadth-first search algorithm

```
Q.enqueue(root node)
while Q is not empty {
  n = Q.dequeue()
  if n.left != null {
    Q.enqueue(n.left)
  }
  if n.right != null {
    Q.enqueue(n.right)
  }
}
```

50

27

73

8

44

83

73

93

BFS:

50

27

73

8

44

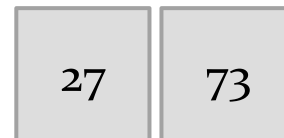
83

73

93

BFS: 50

dequeue 50,
enqueue left and right



50

27

73

8

44

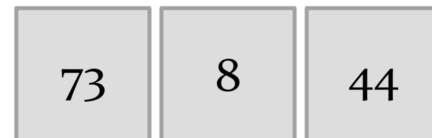
83

73

93

BFS: 50, 27

dequeue 27,
enqueue left and right



50

27

73

8

44

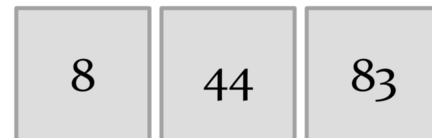
83

73

93

BFS: 50, 27, 73

dequeue 73,
enqueue right



50

27

73

8

44

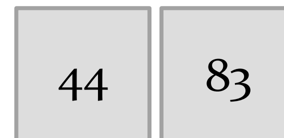
83

73

93

BFS: 50, 27, 73, 8

dequeue 8



50

27

73

8

44

83

73

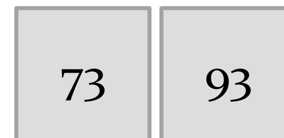
93

BFS: 50, 27, 73, 8, 44

dequeue 44

BFS: 50, 27, 73, 8, 44, 83

dequeue 83,
enqueue left and right



50

27

73

8

44

83

73

93

BFS: 50, 27, 73, 8, 44, 83, 73

dequeue 73

50

27 73

8 44 83

73 93

BFS: 50, 27, 73, 8, 44, 83, 73, 93

dequeue 93

50

27 73

8 44 83

73 93

BFS: 50, 27, 73, 8, 44, 83, 73, 93

queue empty

Previous written exam question

Suppose that you have a **`Stack`** class that has only the following features:

- the elements are of type **`int`**
- a default constructor that creates an empty stack
- a method **`isEmpty`** that returns true if the stack is empty
- **`push`** and **`pop`** methods

Describe how you would write a (static) method that makes a copy of a stack. A postcondition of your method must be that the state of the stack when the method finishes is the same as when the method started. Try to avoid using additional data structures (such as lists and arrays) if possible. Functional Java code is not required.

Previous written exam question

Suppose that you have a **`Queue`** class that has only the following features:

- the elements are of type **`int`**
- a default constructor that creates an empty queue
- a method **`size`** that returns the number of elements in the queue
- **`enqueue`** and **`dequeue`** methods

Describe how you would write a (static) method that makes a copy of a queue. A postcondition of your method must be that the state of the queue when the method finishes is the same as when the method started. Try to avoid using additional data structures (such as lists and arrays) if possible. Functional Java code is not required.