CS 171: Introduction to Computer Science II

Stacks and Queues

Li Xiong

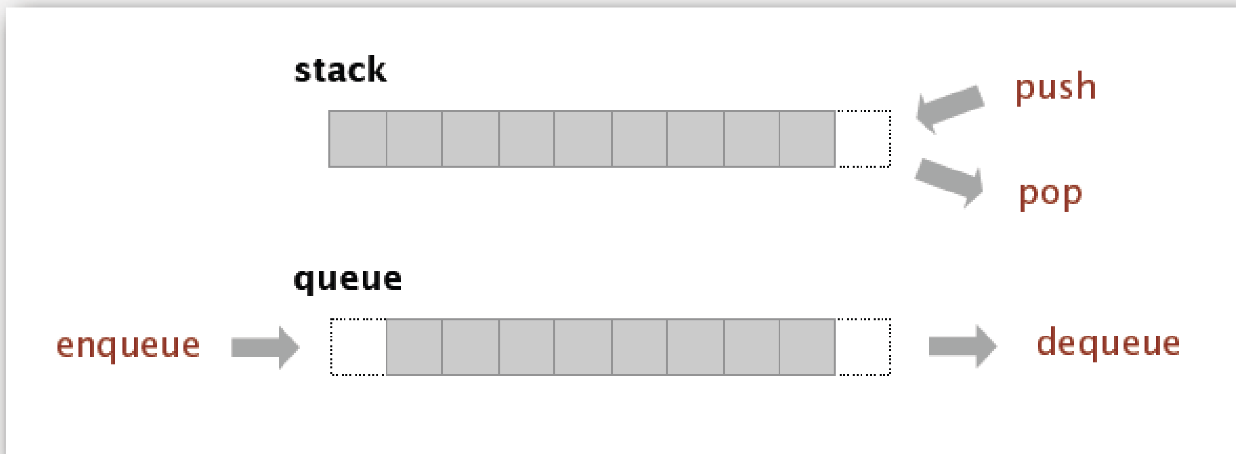
Today

- Stacks: implementations and applications
- Queues: implementations
- Applications using queues
- Deque
- Iterators
- Java collections library – List, Stack, Queue
- Maze application (Hw3)

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

Queue: applications

- Josephus problem

  N people arrange themselves in a circle (at positions numbered from 0 to N-1) and proceed around the circle, eliminating every Mth person until only one person is left.

  Print out the order in which people are eliminated.

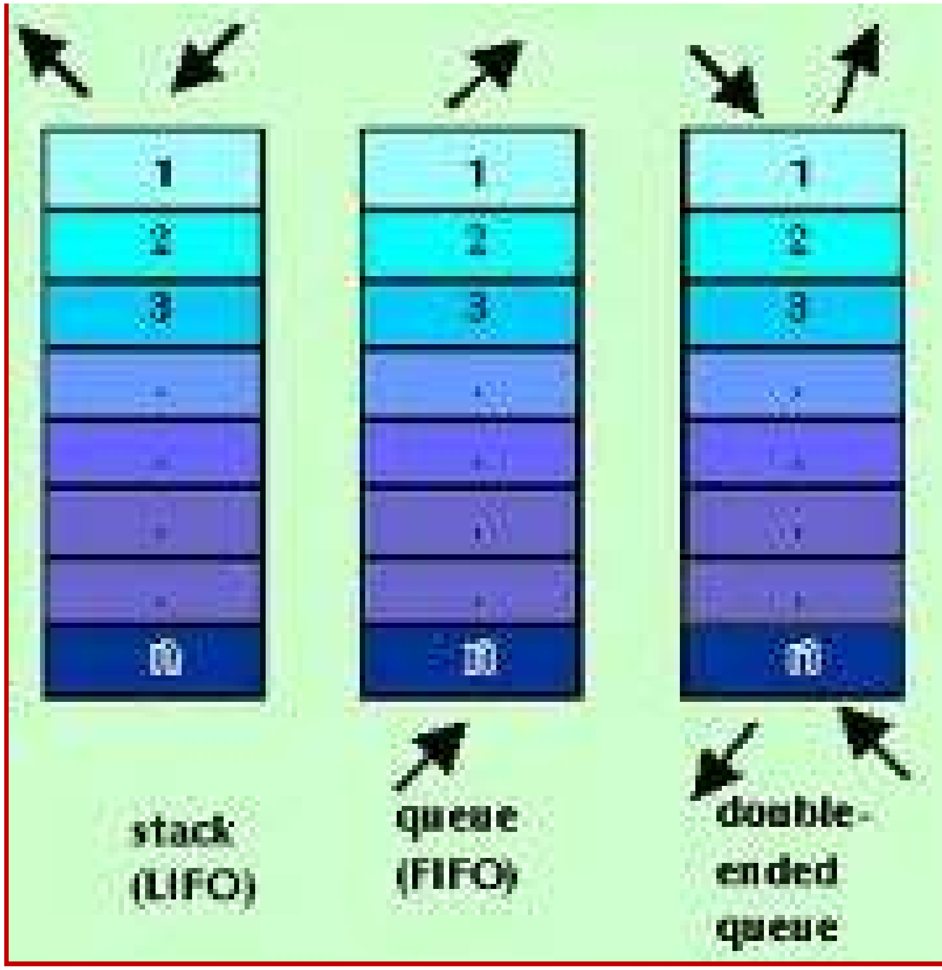
Queue: applications

```
public class Josephus {
	public static void main(String[] args) {
		int M = Integer.parseInt(args[0]);
		int N = Integer.parseInt(args[1]);

		// initialize the queue
		Queue<Integer> q = new Queue<Integer>();
		for (int i = 0; i < N; i++)
		q.enqueue(i);

		// eliminating every Mth element
		while (!q.isEmpty()) {
			for (int i = 0; i < M-1; i++)
				q.enqueue(q.dequeue());
				StdOut.print(q.dequeue() + " ");
		}
		StdOut.println();
	}
}
```

Deque



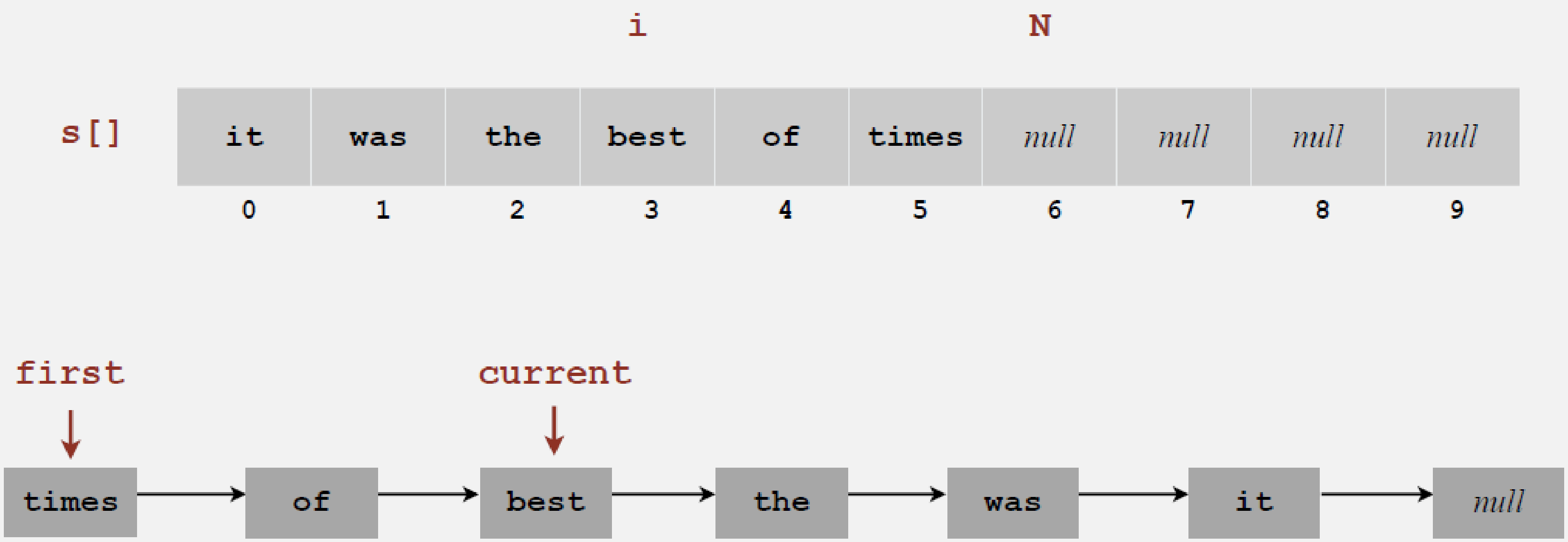
- **Double-ended** queue
- Can insert and delete items at **either end**
- Can be a Stack OR a Queue!
  - addFirst, addLast, removeFirst, removeLast
- **Stack**: if only **addLast** and **removeLast**
- **Queue**: if only **addLast** and **removeFirst**

Today

- Applications using queues
- Deque
- Iterators
- Java collections library – List, Stack, Queue
- Maze application (Hw3)

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



Java solution. Make stack implement the `Iterable` interface.

Iterators

Q. What is an `Iterable` ?

A. Has a method that returns an `Iterator`.

Iterable interface

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?

A. Has methods `hasNext()` and `next()`.

Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();   ←  optional; use
                       at your own risk
}
```

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement

```
for (String s : stack)
   StdOut.println(s);
```

equivalent code

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
     ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;            }
        public void remove()     {  /* not supported */  }
        public Item next()       {  return s[--i];          }
    }
}
```

				i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

Today

- Stacks: implementations and applications
- Queues: implementations
- Applications using queues
- Deque
- Iterators
- Java collections library – List, Stack, Queue
- Maze application (Hw3)

Java collections library

List interface. `java.util.List` is API for ordered collection of items.

`public interface List<Item> implements Iterable<Item>`		
	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*append item to the end*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
`Iteartor<Item>`	`iterator()`	*iterator over all items in the list*
`...`		

Implementations. `java.util.ArrayList` uses resizing array; `java.util.LinkedList` uses linked list.

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
- Also implements `java.util.List` interface from previous slide, including, `get()`, `remove()`, and `contains()`.

Java Queues and Deques

- **java.util.Queue** is an interface and has multiple implementing classes
  - insert() and remove()
- **java.util.Deque** is an interface and has multiple implementing classes
  - Supports insertion and removal at both ends
  - addFirst(), removeFirst(), addLast(), removeLast()

Java ArrayDeque class

- **java.util.ArrayDeque** implements Deque interface and supports both stack and queue operations
- Queue methods
  - add(), addLast()
  - remove(), removeFirst()
  - peek(), peekFirst()
- Stack methods
  - push(), addFirst()
  - pop(), removeFirst()
  - peek(), peekFirst()

Java ArrayDeque Example

```
import java.util.ArrayDeque;
import java.util.Iterator;

public class DequeTest {
  public static void main(String[] args) {
      ArrayDeque<Integer> s = new ArrayDeque<Integer> ();

      s.push(2);
      s.push(4);
      s.push(6);

      System.out.println(s);
      System.out.println(s.pop());

      // use iterator to access inner elements
      Iterator<Integer> iter = s.iterator();
      while (iter.hasNext())
          System.out.println(iter.next());
  }
}
```

Today

- Stacks: implementations and applications
- Queues: implementations
- Applications using queues
- Deque
- Iterators
- Java collections library – List, Stack, Queue
- Hw3: Maze application using stacks and queues

HW3: Maze Traversal

Maze Traversal

- A maze is a square space represented using two-dimensional array
  - Each cell has value 0 (passage) or 1 (**internal wall)**.
  - Entrance at **upper left corner**, an **exit at lower right corner**
- F**ind a path** through from entrance to exit

```
ENTER --> X   1   1   1   0   0   X---X---X   0
          |                       |       |
          X---X---X   1   0   0   X   1   X   0
                  |               |       |
          0   1   X   1   1   X---X   1   X   0
                  |           |           |
          0   1   X---X   1   X   1   1   X   0
                      |       |           |
          0   1   0   X   1   X   1   1   X   0
                      |       |           |
          1   1   1   X   1   X   1   X---X   0
                      |       |       |
          0   0   1   X---X---X   1   X   1   1
                                      |
          0   0   1   0   0   0   1   X   1   1
                                      |
          0   1   1   0   1   0   1   X-- X-- X
                                              |
          0   0   0   0   1   0   1   1   0   X --> EXIT
```

Example Output

```
Path: ( [0][0], [1][0], [1][1], [1][2], [2][2],
        [3][2], [3][3], [4][3], [5][3], [6][3],
        [6][4], [6][5], [5][5], [4][5], [3][5],
        [2][5], [2][6], [1][6], [0][6], [0][7],
        [0][8], [1][8], [2][8], [3][8], [4][8],
        [5][8], [5][7], [6][7], [7][7], [8][7],
        [8][8], [8][9], [9][9])
```

```
ENTER --> x   1   1   1   0   0   x---x---x   0
          |                       |       |
          x---x---x   1   0   0   x   1   x   0
                  |               |       |
          0   1   x   1   1   x---x   1   x   0
                  |           |           |
          0   1   x---x   1   x   1   1   x   0
                      |       |           |
          0   1   0   x   1   x   1   1   x   0
                      |       |           |
          1   1   1   x   1   x   1   x---x   0
                      |       |       |
          0   0   1   x---x---x   1   x   1   1
                                      |
          0   0   1   0   0   0   1   x   1   1
                                      |
          0   1   1   0   1   0   1   x-- x-- x
                                            |
          0   0   0   0   1   0   1   1   0   x --> EXIT
```

Maze search

- Depth-first search
  - At each choice point, follow one path until there is no further choice or exit reached
  - Back trace to previous choice point
- Breadth-first search
  - Split at every choice point

Maze Search Using Stack

- Create a search stack of positions, push the entrance position, (0,0), to the search stack
- While the search stack is not empty
  – Pop the current position from the search stack
  – If it is the exit position, [n-1, n-1], then a path is found, print out the path.
  – else, mark the position as visited, push all **valid** up, down, left, or right neighbor positions (with the current position as its parent) to the stack
- If the stack is empty and a path is not found, there is no path

Maze Search Using Queue

- Create a search queue of positions, push the entrance position, (0,0), to the search queue
- While the search queue is not empty
  - remove a position from the search queue
  - If it is the exit position, [n-1, n-1], then a path is found, print out the path.
  - else, mark the position as visited, insert all **valid** up, down, left, or right neighbor positions (with the current position as its parent) to the queue
- If the queue is empty and a path is not found, there is no path

Implementation Hints/details

- Use a simple object (e.g., Cell) to store the (i, j) position in the maze
- Use built-in Java Deque to manage your Cells
  - uses a **Stack** or a Queue to manage the search list