ENG 346
Data Structures and Algorithms for Artificial Intelligence
Stacks and Queues

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **0uv7jlm**

Agenda

- Abstract Data Types
- Stacks
- Queues

Abstract Data Types

- High-level description of a collection of data and the operations that can be performed on that data.

- Benefits:
  - Data Structure Abstraction: Behavior of a data structure.
  - Operations: A set of operations that can be performed on the data.
  - Encapsulation: Encapsulate the data and operations into a single unit.
  - Reusability: Reuse in different applications.

ADTs – continued

- **List**: List of elements accessible by positions.
- **Dictionary**: Key-value pairs.
- **Set**: Collection of distinct elements.
- **Stack**: Follows Last-In-First-Out (LIFO) principle.
- **Queue**: Follows First-In-First-Out (FIFO) principle.
- **Graph**: Vertices and using edges.

Stacks

- Insertions and deletions from the same end of the list.
- Follow the last-in first-out scheme

Main Operations	Auxiliary Operations
S.push(item)	item = S.top()
item = S.pop()	len(S)
	S.is_empty()

Stacks – continued

- General Applications
  - Function's call stack
  - Internet Browser history
  - Editor undo/redo
  - …
- Algorithm design
  - Reverse polish notation
  - …

Stacks – Example

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

Array-Based Stack Implementation

```
class Stack:
    def __init__(self, size=10):
        pass
    def push(self, data):
        pass
    def pop(self):
        pass
    def is_empty(self):
        pass
    def top(self):
        pass
    def __len__(self):
        pass
    def display(self):
        pass
```

Example – Parenthesis Matching

Algorithm ParenthesisMatching(*X, n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in *X* match

Let *S* be an empty stack

for *i*=0 to *n*-1 **do**

    if *X*[*i*] is an opening grouping symbol **then**

        S.push(*X*[*i*])

    else if *X*[*i*] is a closing grouping symbol **then**

        if *S*.is_empty() **then**

            return false {nothing to match with}

        if *S*.pop() does not match the type of *X*[*i*] **then**

            return false {wrong type}

if *S*.isEmpty() **then**

    return true {every symbol matched}

else return false {some symbols were never matched}

Reverse Polish Notation

- Mathematical notation: Infix, operators are between operands.
  - E.g.: 2 + 3 * (4 + 7)
- Polish Notation: Prefix, operators before operands.
  - E.g.: + 2 * 3 + 4 7
- Reverse Polish Notation: Postfix, operators follow operands.
  - E.g.: 2 3 4 7 + * +

Example – Reverse Polish Notation Calculator

- Using Stack

Queues

- Insertions to the end, deletions from the front of the list.
- Follow the first-in first-out scheme

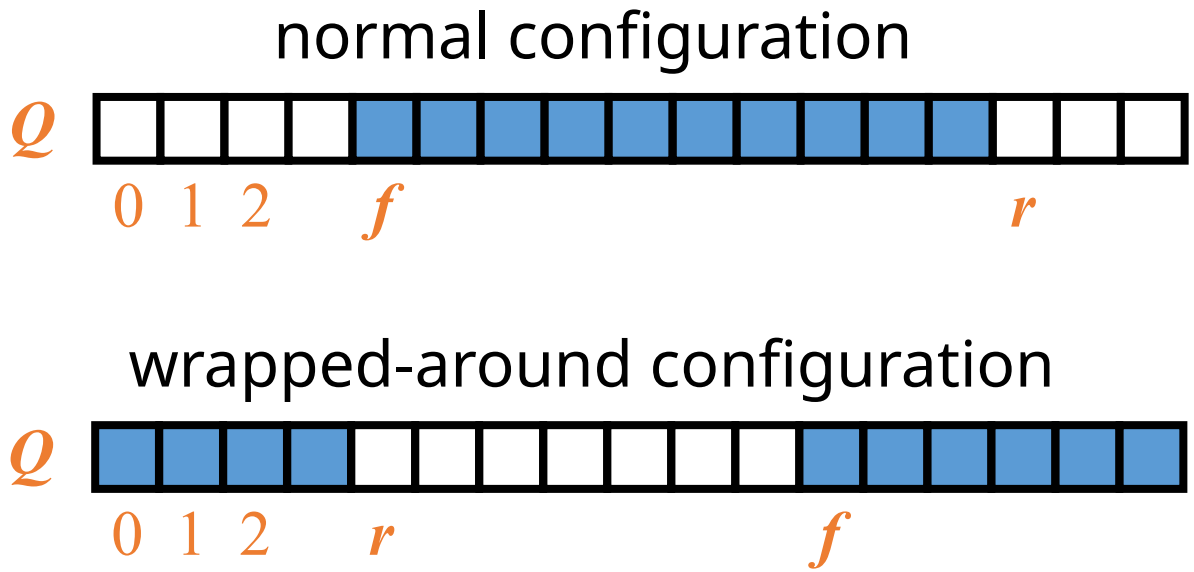
Main Operations	Auxiliary Operations
Q.enqueue(item)	item = Q.first()
item = Q.dequeue()	len(Q)
	Q.is_empty()

Queues – continued

- General Applications
  - Waiting lists
  - Access to shared resources (e.g., printer)
  - Round Robin Scheduler
  - …
- Algorithm design
  - …

Array-based Queue Implementation

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty



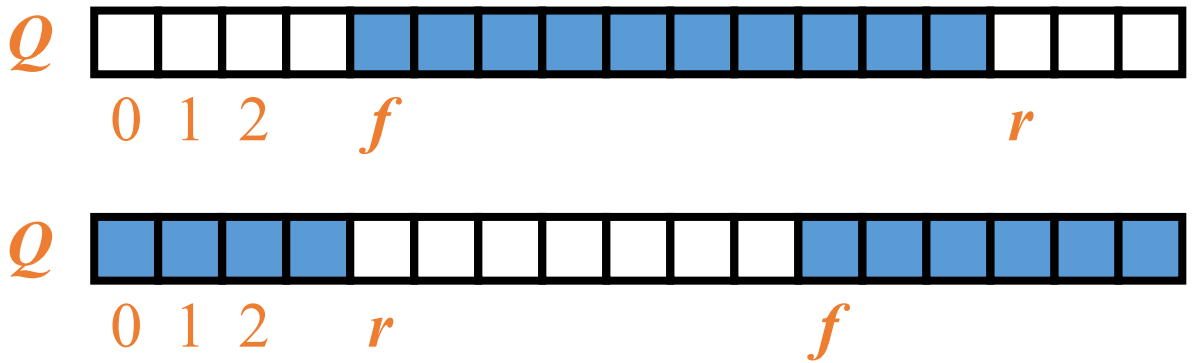
© 2013 Goodrich, Tamassia, Goldwasser

Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

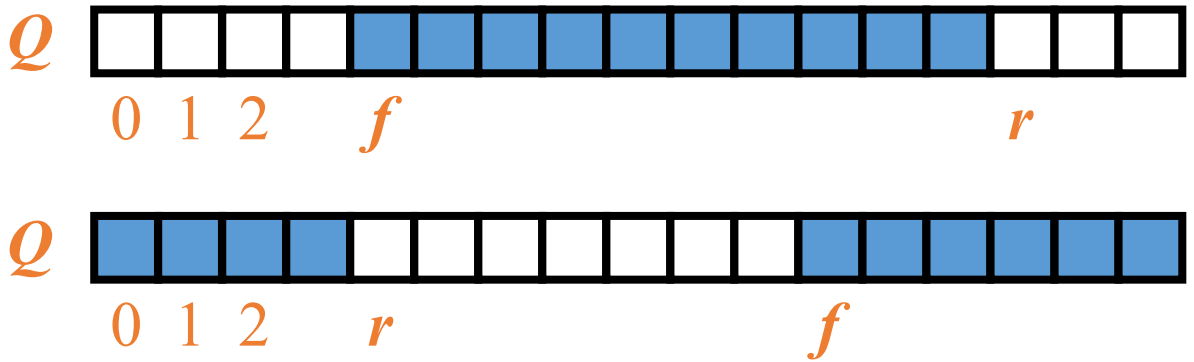
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

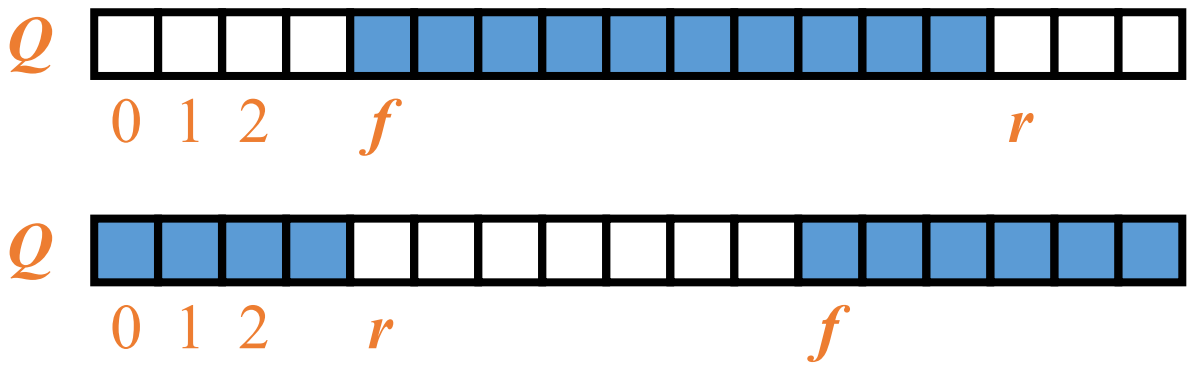
Algorithm *enqueue*(o)
  if *size*$() = \tilde{N}\ 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \sqcup o$
    $r \sqcup (r + 1) \bmod N$



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm *dequeue*()
  if *isEmpty*() **then**
    throw *EmptyQueueException*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return o



Queues – Example

Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

Array Based Queue Implementation

```
class Queue:
    def __init__(self, c=10):
        pass
    def is_empty(self):
        pass
    def enqueue(self, data):
        pass
    def dequeue(self):
        pass
    def __len__(self):
        pass
    def display(self):
        pass
    def first(self):
        pass
```