CBS Department

Stack, Queue and Deque

Soma Soleiman Zadeh

Data Structures & Algorithms (CBS 216)

Spring 2025 - 2026

Week 3

February 15, 2026

Outline

- **Abstract Data Types (ADT)**
- **Linear Data Structures**
  - **Lists**
  - **Stacks**
  - **Queues**
  - **Deques**

Linear Data Structures

- **Lists**, **stacks**, **queues** and **deques** are examples of **Linear Data Structures**.
- Their elements are arranged based on the order in which they are added or removed.
- **Linear data structures have two ends**. What makes one linear structure different from another is the way their elements are added and removed.

Advantages & Disadvantages of Some Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick insertion, Fast access if index known.	Slow search, slow deletion, fixed size.
Ordered Array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in first-out (LIFO) access.	Slow access to other items.
Queue	Provides first-in first-out (FIFO) access.	Slow access to other items.

Abstract Data Types

- An **Abstract Data Type (ADT)** is a theoretical concept that
  - **Defines a data type** only based on its behavior and operations,
  - Without specifying its implementation details.

Review on Lists

Lists

- A type of sequence (ordered collection), can contain elements of any type.

  product = ['laptop' , 1.7 , 'Silver' , 1100]

- Lists are mutable.
  - Elements can be changed, added or removed from a list.
  - You can sort a list.

product[2] = 'Black'

→ product = ['laptop' , 1.7 , 'Black' , 1100]

Indexing and Slicing Lists

- We can get at any single element or range of elements in a list using index specified in square brackets.

  subjects = ['C#' , 'OS' , 'Network']

  subjects[-2]

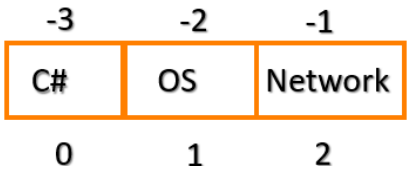
  → OS

  subjects[:2]

  → ['C#' , 'OS']

  len(subjects)

  → 3



range( ) Function

- The **range()** function returns a list of numbers that range from zero (by default) to one less than the parameter.
- We can construct an index loop using **for** and an integer **iterator**.

```
range(4)          → [0, 1, 2, 3]
range(2,7)        → [2, 3, 4, 5, 6]
range(2,10,2)
                  → [2, 4, 6, 8]

for i in range(4):
    print(i)
```

List Methods

- A **list** has its methods for manipulating the list:
  - **Adding an element** to a list
  - **Removing an element** from a list
  - **Sorting** the List

append( ) Method

- append(value)
  - **append( )** adds a value to the end of a list.
- **Example:**

myList = [5, 8, 4, 8]

myList.**append**(**6**)

print(myList)

→ **Output:** [5, 8, 4, 8, **6**]

insert( ) Method

- insert(*i* , *value*)
  - Inserts ***value*** at position ***i***.

- **Example:**

myList = [“apple” , “banana” , “cherry” , “orange”]

myList.**insert**(**2, “mango”**)

print(myList)

→ **Output:** [“apple” , “banana”, “mango”, “cherry”, “orange”]

pop( ) Method

- pop(*i*)
  - Returns ***value*** at position ***i*** and removes ***value*** from the list.
  - Providing the position number ***i*** is *optional*. Without it, the last element in the list is removed and returned.
- **Example:**

```
L = [9, 6, 0, 3]
L.pop(0)
print(L)
```

→ **Output:** [6,0,3]

```
L = [9, 6, 0, 3]
A = L.pop( )
print(L)
```

→ **Output:** [9, 6, 0]

```
print(A)
```

→ **Output:** 3

remove( ) Method

- remove(*value*)
  - Removes the first occurrence of the ***value*** from the list.
- **Example:**

```
L = [5, 3, 1, 9, 1]
L.remove(3)
print(L)
```

→ **Output:** [5, 1, 9, 1]

```
L = [5, 3, 1, 9, 1]
L.remove(1)
print(L)
```

→ **Output:** [5, 3, 9, 1]

List Methods

List Method	Description
append( )	Adds a single element at the end of list.
insert( )	Inserts a single element at the defined index.
pop( )	Removes an element with given index from the list.
remove( )	Removes an element with given value from the list.

Stacks & Queues

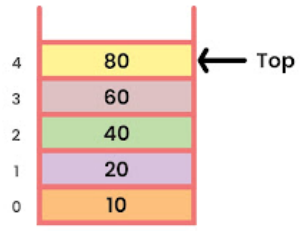
- **Stacks** and **Queues** are data structures for handling **temporary data**.
- Some Applications of **stacks** and **queues**:
  - In **Operating System Architecture**
  - **Printing jobs**
  - **Traversing data**.
- **Stacks** and **queues** allow you to handle data in order, and then get rid of it once you don't need it anymore.

Stacks

- A **stack** is a linear data structure that has two ends; bottom and top.

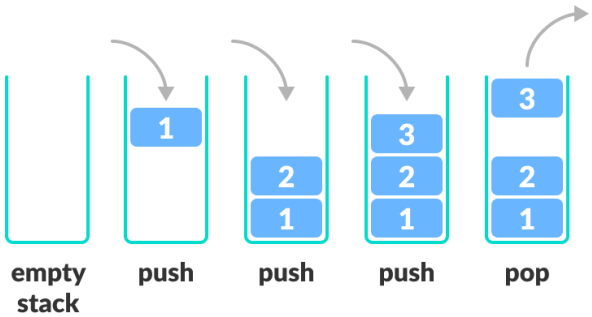
  All operations (adding and removing) are made at one end (**Top**).

- **Stacks** follow **Last-In-First-Out (LIFO)** approach.



Stack Methods (Push and Pop)

- **Pushing** an element → adding a new element on top of the stack.
- **Popping** an element → removing an element from the top of the stack.



Stack Methods

- **push(item)** → Inserts element **item** onto top of stack.
- **pop( )** → Removes the top element of stack and returns it; if stack is empty an error is shown.
- **size( )** → Returns the number of elements in stack.
- **isEmpty( )** → Returns a Boolean indicating if stack is empty.
- **peek( )** → Returns the top element of the stack, without removing it; if the stack is empty an error is shown.

Example of Stack Operations

Stack Operation	Stack items	Return Value
s.isEmpty( )	[ ]	True
s.push(4)	[4]	
s.push('a')	[4, 'a']	
s.peek( )	[4, 'a']	'a'
s.push(5.3)	[4, 'a', 5.3]	
s.size( )	[4, 'a', 5.3]	3
s.pop( )	[4, 'a']	5.3
s.pop( )	[4]	'a'
s.size( )	[4]	1

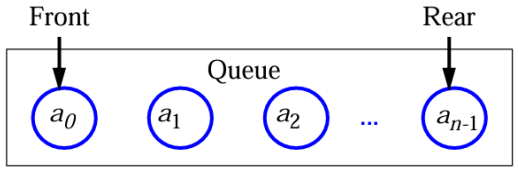
Implementation of Stack in Python

- You can define a class for **Stack** data structure.
- To define the stack methods, you can use list methods such as append (for **push** operation), pop (for **pop** operation) and len (for **size** operation).

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, newItem):
        self.items.append(newItem)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

Queues

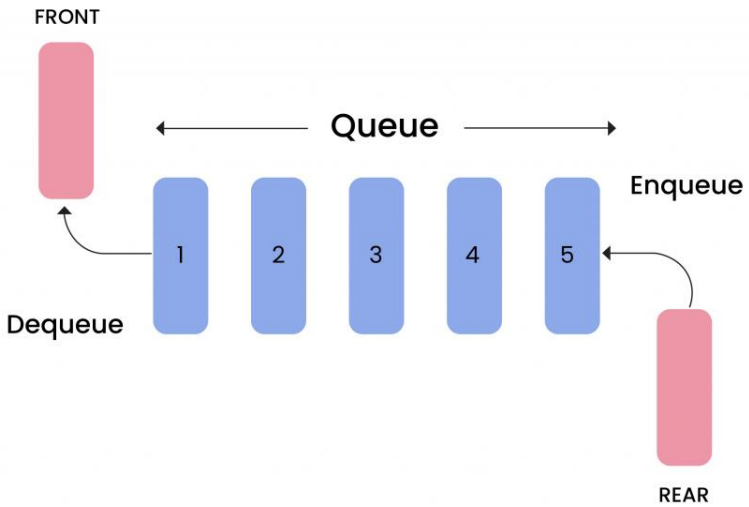
- A Queue is a linear data structure that elements are inserted at the rear (enqueued) and removed from the front (dequeued).
- **Queues** follow **First-In-First-Out (FIFO)** approach.
- **Enqueue** → Insertion at the **rear** of the queue.
- **Dequeue** → Deletion from the **front** of the queue.



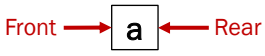
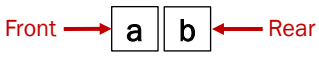
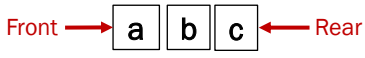
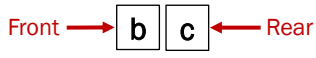
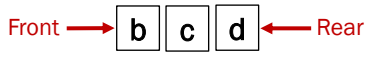
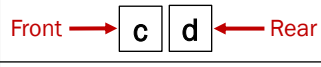
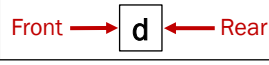
Queue Methods

- **enqueue(item)** → Inserts element **item** at the rear of the queue.
- **dequeue( )** → Removes the element from the front of the queue and returns it; if queue is empty an error is shown.
- **size( )** → Returns the number of elements in queue.
- **isEmpty( )** → Returns a Boolean indicating if queue is empty.
- **front( )** → Returns the front element of the queue, without removing it; if the queue is empty an error is shown.

Enqueue vs Dequeue



Enqueue vs Dequeue

Operation	Queue After Operation
enqueue(a)	
enqueue(b)	
enqueue(c)	
dequeue()	
enqueue(d)	
dequeue()	
dequeue()	

Example of Queue Operations

Queue Operation	Queue items	Return Value
q.isEmpty( )	[ ]	True
q.enqueue(4)	[4]	
q.enqueue('pen')	[4, 'pen']	
q.enqueue(False)	[4, 'pen', False]	
q.size( )	[4, 'pen', False]	3
q.enqueue(7.5)	[4, 'pen', False, 7.5]	
q.dequeue( )	['pen', False, 7.5]	4
q.dequeue( )	[False, 7.5]	'pen'
q.size( )	[False, 7.5]	2

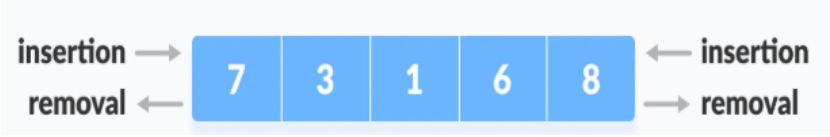
Implementation of Queue in Python

- You can define a class for **Queue** data structure.
- To define the queue methods, you can use list methods such as append (for **enqueue** operation), pop (for **dequeue** operation) and len (for **size** operation).

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, newItem):
        self.items.append(newItem)

    def dequeue(self):
        return self.items.pop(0)

    def front(self):
         return self.items[0]

    def size(self):
        return len(self.items)
```

Deques

- **Deque** or **Double Ended Queue** is a generalized version of Queue data structure that allows insertion and removal elements at both ends.
- **Deque** generalizes both a Stack and a Queue.
- **Deques** are **First-In-First-Out (FIFO)** and **Last-In-First-Out (LIFO).**



collections Module in Python

- The **collections** module is a module in Python, containing additional data structures and operations which are more efficient than the built-in data structures in certain situations.
- **deque** from **collection** module is used to create deque object. The deque object is a list-like object that provides fast insertion and deletion from both ends.
-

deque Methods

- deque from collections module has special methods such as:
  - **append(*item*)** → Adds element **item** to the end of the deque.
  - **appendleft(*item*)** → Adds element **item** to the beginning (left side) of the deque.
  - **pop( )** → Removes and returns an element from the end of the deque.
  - **popleft( )** → Removes and returns an element from the beginning (left side) of the deque.

Implementation of deque Using collections Module

```
from collections import deque

my_deque = deque()
my_deque.append(10)
my_deque.append(15)
my_deque.append(20)
my_deque.appendleft(25)

print(my_deque)
print("Deque size is:", len(my_deque))
```

```
deque([25, 10, 15, 20])
Deque size is: 4

Deque after removing from the end: deque([25, 10, 15])
Deque after removing from the beginning: deque([10, 15])
```

```
my_deque.pop()
print("Deque after removing from the end:", my_deque)

my_deque.popleft()
print("Deque after removing from the beginning:",my_deque)
```

Thank You!