

# Data Structure (again)

Shutchon Premchaisawatt

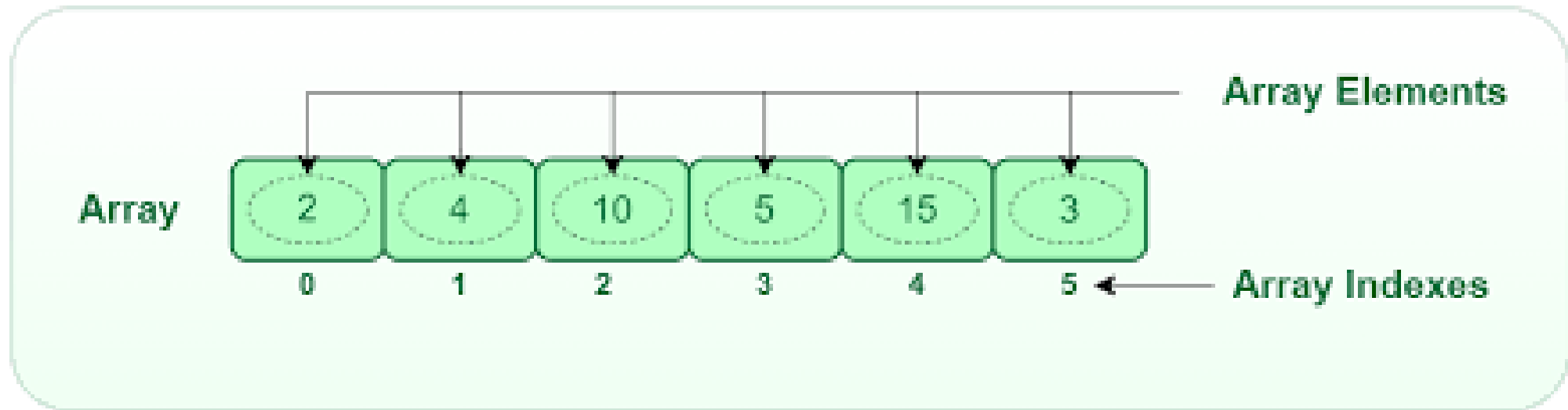
# โครงสร้างข้อมูลคืออะไร?

โครงสร้างข้อมูล (Data Structure) คือวิธีการจัดเก็บและจัดการข้อมูลในคอมพิวเตอร์อย่างมีประสิทธิภาพ เพื่อให้สามารถเข้าถึงและปรับปรุงข้อมูลได้อย่างรวดเร็วและง่ายดาย ตัวอย่างของโครงสร้างข้อมูลที่พบบ่อยได้แก่:

- **Array:** การจัดเก็บข้อมูลในรูปแบบของลำดับที่มีขนาดคงที่
- **Linked List:** การจัดเก็บข้อมูลในรูปแบบของโหนดที่เชื่อมต่อกัน
- **Stack:** โครงสร้างข้อมูลที่ทำงานในลักษณะ LIFO (Last In, First Out)
- **Queue:** โครงสร้างข้อมูลที่ทำงานในลักษณะ FIFO (First In, First Out)
- **Tree:** โครงสร้างข้อมูลที่มีลักษณะเป็นลำดับชั้น
- **Graph:** โครงสร้างข้อมูลที่ประกอบด้วยโหนดและเส้นเชื่อมระหว่างโหนด

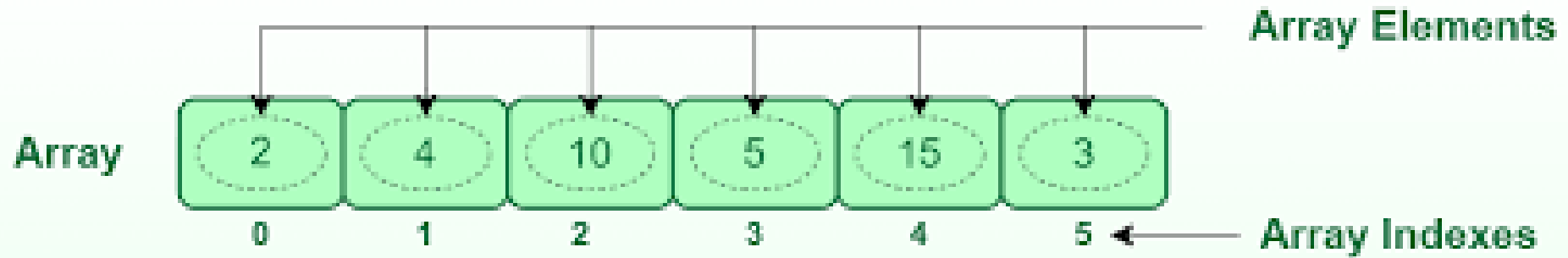
# ทำไมต้องเรียนรู้เกี่ยวกับ โครงสร้างข้อมูล?

- 1.เพิ่มประสิทธิภาพ:** โครงสร้างข้อมูลที่เหมาะสมช่วยให้การจัดเก็บและเข้าถึงข้อมูลมีประสิทธิภาพมากขึ้น ซึ่งสำคัญมากในการพัฒนาโปรแกรมที่ทำงานได้รวดเร็วและใช้ทรัพยากรน้อยลง
- 2.การแก้ปัญหา:** การเข้าใจโครงสร้างข้อมูลช่วยให้สามารถเลือกวิธีการแก้ปัญหาที่เหมาะสมกับปัญหาที่พบได้ เช่น การใช้ **Stack** ในการทำงานที่ต้องการย้อนกลับไปยังสถานะก่อนหน้า หรือการใช้ **Tree** ในการจัดการข้อมูลที่มีลำดับชั้น
- 3.การพัฒนาโปรแกรม:** โครงสร้างข้อมูลเป็นพื้นฐานสำคัญในการพัฒนาโปรแกรมและอัลกอริทึมที่มีประสิทธิภาพ การรู้จักและเข้าใจโครงสร้างข้อมูลต่าง ๆ จะช่วยให้คุณเขียนโค้ดที่มีประสิทธิภาพและง่ายต่อการบำรุงรักษา
- 4.การเตรียมตัวสำหรับการสัมภาษณ์งาน:** หลายบริษัทใช้คำถามเกี่ยวกับโครงสร้างข้อมูลในการสัมภาษณ์งาน ด้านวิศวกรรมซอฟต์แวร์ การมีความรู้ในด้านนี้จะช่วยให้คุณมีความพร้อมและมั่นใจมากขึ้นในการสัมภาษณ์
- 5.การพัฒนาทักษะการคิดเชิงตรรกะ:** การเรียนรู้โครงสร้างข้อมูลช่วยพัฒนาทักษะการคิดเชิงตรรกะและการแก้ปัญหา ซึ่งเป็นทักษะที่มีประโยชน์ในหลายด้านของชีวิตและการทำงาน



# Arrays

- **นิยาม: Array** คือโครงสร้างข้อมูลที่ใช้ในการจัดเก็บข้อมูลหลายๆ ค่าในตัวแปรเดียว โดยค่าทั้งหมดจะถูกจัดเก็บในลำดับที่ต่อเนื่องกันในหน่วยความจำ
- **ลักษณะ:**
  - **ขนาดคงที่:** ขนาดของ **Array** จะถูกกำหนดตั้งแต่ตอนสร้างและไม่สามารถเปลี่ยนแปลงได้
  - **การเข้าถึงแบบสุ่ม:** สามารถเข้าถึงข้อมูลในตำแหน่งใดๆ ได้โดยตรงผ่านดัชนี (index)
  - **ประเภทข้อมูลเดียวกัน:** ข้อมูลทั้งหมดใน **Array** จะต้องเป็นประเภทเดียวกัน เช่น ทั้งหมดเป็นตัวเลขหรือทั้งหมดเป็นตัวอักษร



- # การสร้าง **Array** ใน **Python**  
`array = [2, 4, 10, 5, 15, 3]`  
`print(array[2])` # ผลลัพธ์: 10

# List

- นิยาม: **List** คือโครงสร้างข้อมูลที่ใช้ในการจัดเก็บข้อมูลหลายๆ ค่าในตัวแปรเดียว โดยค่าทั้งหมดสามารถมีประเภทข้อมูลที่แตกต่างกันได้และสามารถเปลี่ยนแปลงขนาดได้
- ลักษณะ:
  - ขนาดยืดหยุ่น: ขนาดของ **List** สามารถเปลี่ยนแปลงได้ตามต้องการ สามารถเพิ่มหรือลบข้อมูลได้
  - การเข้าถึงแบบสุ่ม: สามารถเข้าถึงข้อมูลในตำแหน่งใดๆ ได้โดยตรงผ่านดัชนี (**index**)
  - ประเภทข้อมูลหลากหลาย: ข้อมูลใน **List** สามารถเป็นประเภทใดก็ได้ เช่น ตัวเลข ตัวอักษร หรือแม้กระทั่ง **List** อื่นๆ

## List in Python

L = [ 20, 'Jessa', 35.75, [30, 60, 90] ]

↑      ↑      ↑      ↑

L[0]   L[1]   L[2]   L[3]

- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Changeable**: List is mutable and we can modify items.
- ✓ **Heterogeneous**: List can contain data of different types
- ✓ **Contains duplicate**: Allows duplicates data

# List

PYnative.com

## List in Python

`L = [ 20, 'Jessa', 35.75, [30, 60, 90] ]`

`L[0]`   `L[1]`   `L[2]`   `L[3]`

- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Changeable**: List is mutable and we can modify items.
- ✓ **Heterogeneous**: List can contain data of different types
- ✓ **Contains duplicate**: Allows duplicates data

# การสร้าง List ใน Python

```
L = [2, "Jessa", 35.75, [30, 60, 90]]  
print(list[1])   # ผลลัพธ์: Jessa
```

# ความเหมือนและแตกต่างของ List และ Array

**List** และ **Array** มีความแตกต่างกันในหลายด้าน

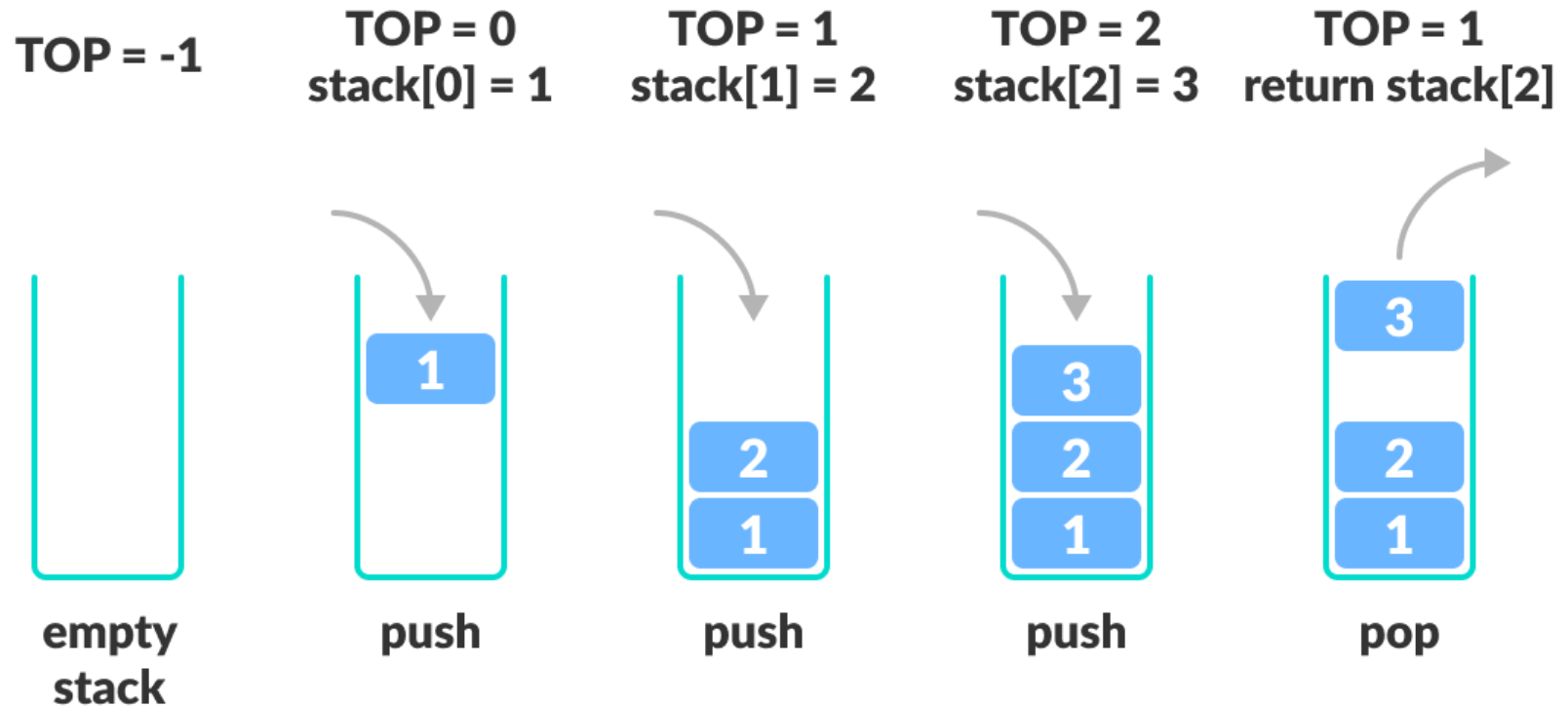
- **ขนาดและการปรับขนาด**
  - **Array:** ขนาดของ **Array** ถูกกำหนดตั้งแต่ตอนสร้างและไม่สามารถเปลี่ยนแปลงได้
  - **List:** ขนาดของ **List** สามารถเปลี่ยนแปลงได้ตามต้องการ สามารถเพิ่มหรือลบข้อมูลได้
- **ประเภทข้อมูล**
  - **Array:** ข้อมูลทั้งหมดใน **Array** จะต้องเป็นประเภทเดียวกัน เช่น ทั้งหมดเป็นตัวเลขหรือทั้งหมดเป็นตัวอักษร
  - **List:** ข้อมูลใน **List** สามารถเป็นประเภทใดก็ได้ เช่น ตัวเลข ตัวอักษร หรือแม้กระทั่ง **List** อื่นๆ
- **การเข้าถึงข้อมูล**
  - **Array:** สามารถเข้าถึงข้อมูลในตำแหน่งใดๆ ได้โดยตรงผ่านดัชนี (index)
  - **List:** สามารถเข้าถึงข้อมูลในตำแหน่งใดๆ ได้โดยตรงผ่านดัชนี (index)
- **การใช้งานในภาษาโปรแกรม**
  - **Array:** ในบางภาษาการเขียนโปรแกรม เช่น C หรือ Java, **Array** เป็นโครงสร้างข้อมูลพื้นฐานที่มีประสิทธิภาพสูง
  - **List:** ในภาษาโปรแกรมบางภาษา เช่น Python, **List** เป็นโครงสร้างข้อมูลที่มีความยืดหยุ่นและใช้งานง่าย



# ตัวอย่างการใช้งาน List

- # สร้าง List  
my\_list = [1, 2, 3, 4, 5]  
  
# เข้าถึงสมาชิก  
print(my\_list[0]) # Output: 1  
  
# เพิ่มสมาชิก  
my\_list.append(6)  
  
# ลบสมาชิก  
my\_list.remove(3)  
  
# วนลูปผ่าน List  
for item in my\_list:  
 print(item)

# Stacks



- **Stack** เป็นโครงสร้างข้อมูลที่ใช้ในการจัดเก็บข้อมูลในลำดับที่สามารถเข้าถึงได้เฉพาะจากด้านบนสุด (top) เท่านั้น การดำเนินการหลักที่สามารถทำได้กับ **Stack** มีดังนี้:
  - 1.Push:** การเพิ่มข้อมูลใหม่เข้าไปที่ด้านบนสุดของ **Stack**
  - 2.Pop:** การนำข้อมูลที่อยู่ด้านบนสุดของ **Stack** ออก
  - 3.Peek (หรือ Top):** การดูข้อมูลที่อยู่ด้านบนสุดของ **Stack** โดยไม่ลบออก
  - 4.isEmpty:** การตรวจสอบว่า **Stack** ว่างหรือไม่

# การเขียน Stack:

```
# Stack สร้างจาก List
# สร้าง List
my_list = [1, 2, 3, 4, 5]

# เข้าถึงสมาชิก
print(my_list[0]) # Output: 1

# เพิ่มสมาชิก
my_list.append(6) # เอาเข้าตัวสุดท้าย

# ลบสมาชิก # เอาตัวสุดท้ายออก
n = my_list[-1]
my_list.remove(my_list[-1])

# วนลูปผ่าน List
for item in my_list:
    print(item)

# ดูค่า n
print(n)
```

```
# ใช้คำสั่งของ Stack
# สร้าง List
my_list = [1, 2, 3, 4, 5]

# เข้าถึงสมาชิก
print(my_list[0]) # Output: 1

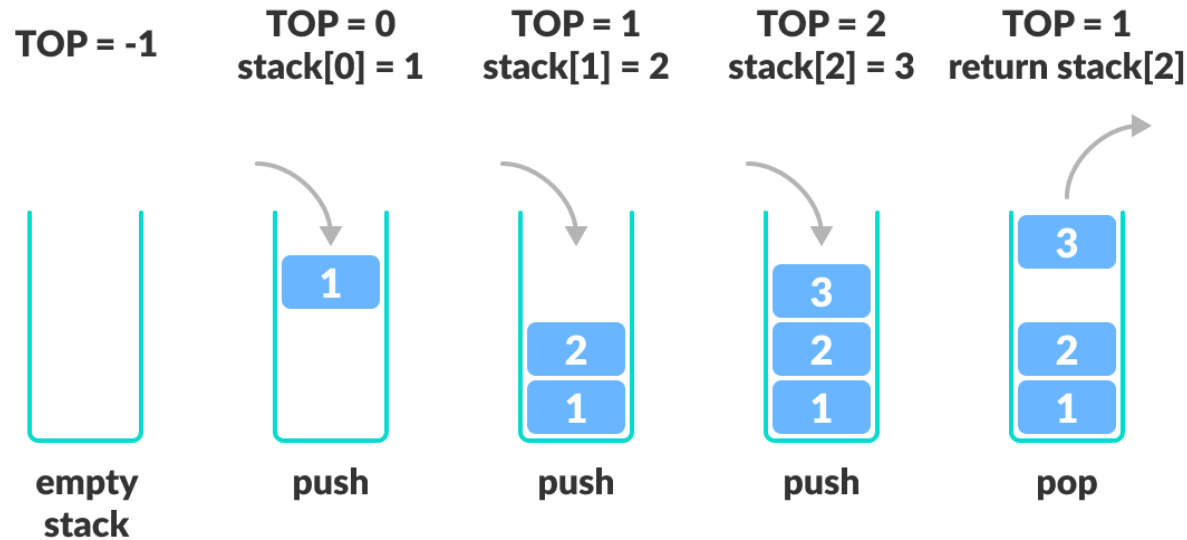
# เพิ่มสมาชิก
my_list.append(6) # เอาเข้า

# เอาสมาชิกตัวสุดท้ายออกออก มาเก็บไว้ที่ n
n = my_list.pop() # เอาออก

# วนลูปผ่าน List
for item in my_list:
    print(item)

# ดูค่า n
print(n)
```

# การสร้าง Stack ตามโครงสร้าง



#สร้าง

```
stack stack = Stack()
```

#ใส่ค่า

```
stack.push(1)
```

```
stack.push(2)
```

```
stack.push(3)
```

#เอาค่าสุดท้ายออก

```
stack.pop()
```

```
class Stack:
    def __init__(self):
        # ใช้ list เป็น container สำหรับเก็บข้อมูลใน stack
        self.items = []

    def push(self, item):
        # เพิ่มข้อมูลไปที่ stack
        self.items.append(item)

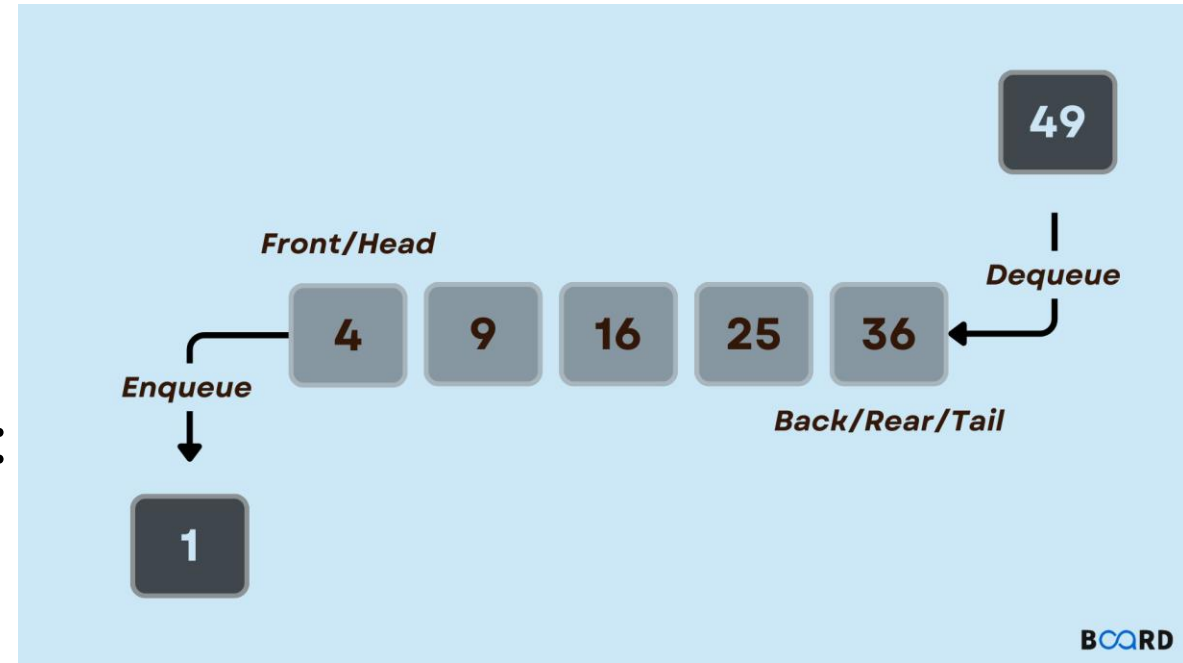
    def pop(self):
        # ลบข้อมูลตัวบนสุดออกจาก stack และคืนค่าข้อมูลนั้น
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Stack is empty"

    def peek(self):
        # ดูข้อมูลตัวบนสุดของ stack โดยไม่ลบออก
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"

    def is_empty(self):
        # เช็คว่างหรือไม่
        return len(self.items) == 0
```

# Queue

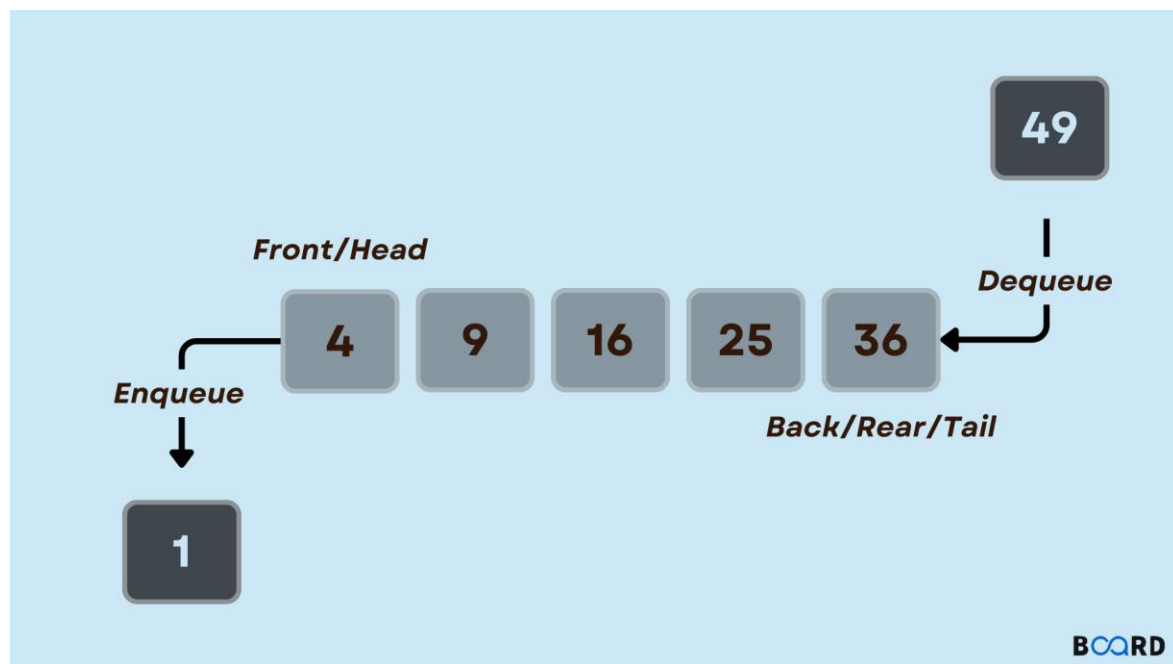
**Queue** หรือคิว คือโครงสร้างข้อมูลที่ใช้ในการจัดเก็บและจัดการข้อมูลในลำดับที่เข้ามาก่อนออกก่อน (**FIFO: First-In-First-Out**) ซึ่งหมายความว่าข้อมูลที่ถูกเพิ่มเข้ามาในคิวก่อน จะถูกนำออกไปใช้งานก่อนเสมอ



หลักการทำงานของ **Queue (FIFO)**

1. การเพิ่มข้อมูล (**Enqueue**): ข้อมูลใหม่จะถูกเพิ่มเข้ามาที่ท้ายคิว.
2. การนำข้อมูลออก (**Dequeue**): ข้อมูลที่อยู่หน้าคิวจะถูกนำออกไปใช้งานก่อน.
3. การตรวจสอบข้อมูล (**Peek/Front**): ตรวจสอบข้อมูลที่อยู่หน้าคิวโดยไม่ลบออก.
4. การตรวจสอบความว่างเปล่า (**IsEmpty**): ตรวจสอบว่าคิวว่างหรือไม่.

# การใช้งาน Queue อย่างง่าย



```
from collections import deque
```

```
queue = deque()
```

```
# Enqueue
```

```
queue.append(1)  
queue.append(4)  
queue.append(9)  
queue.append(16)  
queue.append(25)  
queue.append(36)  
queue.append(49)
```

```
# Dequeue
```

```
print(queue.popleft()) # Output: 1
```

```
# Peek
```

```
print(queue[0]) # Output: 2
```

```
# Check if empty
```

```
print(len(queue) == 0) # Output: False
```

# การสร้าง Queue ตามโครงสร้าง

```
# ทดสอบการทำงานของ queue
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print(queue.front())      # ควรแสดงผล 10
print(queue.dequeue())    # ควรแสดงผล 10
print(queue.dequeue())    # ควรแสดงผล 20
print(queue.is_empty())   # ควรแสดงผล False
print(queue.dequeue())    # ควรแสดงผล 30
print(queue.is_empty())   # ควรแสดงผล True
```

```
class Queue:
    def __init__(self):
        # ใช้ list เป็น container สำหรับเก็บข้อมูลใน queue
        self.items = []

    def enqueue(self, item):
        # เพิ่มข้อมูลไปที่ queue (ที่ทำของ list)
        self.items.append(item)

    def dequeue(self):
        # ลบข้อมูลตัวแรกออกจาก queue และคืนค่าข้อมูลนั้น
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Queue is empty"

    def front(self):
        # ดูข้อมูลตัวแรกของ queue โดยไม่ลบออก
        if not self.is_empty():
            return self.items[0]
        else:
            return "Queue is empty"

    def is_empty(self):
        # เช็คว่างหรือไม่
        return len(self.items) == 0
```

# ตัวอย่างการใช้งาน Queue

Queue (หรือ Queue Data Structure) เป็นโครงสร้างข้อมูลทำงานตามหลักการ FIFO (First In, First Out) ซึ่งหมายความว่า ข้อมูลที่ถูกเพิ่มเข้ามาก่อนจะถูกนำออกก่อน ตัวอย่างการใช้งาน Queue มีดังนี้:

- 1.การจัดการงานในเครื่องพิมพ์ (Print Queue):** เมื่อคุณส่งพิมพ์เอกสารหลาย ๆ ฉบับ เอกสารจะถูกจัดเก็บใน Queue และพิมพ์ตามลำดับที่ส่งพิมพ์
- 2.การจัดการคิวในระบบปฏิบัติการ (Process Scheduling):** ระบบปฏิบัติการใช้ Queue ในการจัดการคิวของกระบวนการ (process) ที่ต้องการใช้ CPU เพื่อให้แน่ใจว่ากระบวนการทั้งหมดได้รับการประมวลผลตามลำดับ
- 3.การจัดการคิวในเครือข่าย (Network Queue):** ในการส่งข้อมูลผ่านเครือข่าย ข้อมูลจะถูกจัดเก็บใน Queue เพื่อรอการส่งไปยังปลายทางตามลำดับ
- 4.การบริการลูกค้า (Customer Service):** ในการให้บริการลูกค้า เช่น ในธนาคารหรือร้านอาหาร ลูกค้าจะถูกจัดเก็บใน Queue และให้บริการตามลำดับที่มาถึง
- 5.การจัดการเหตุการณ์ (Event Handling):** ในการพัฒนาโปรแกรมที่ต้องจัดการกับเหตุการณ์ต่าง ๆ เช่น การคลิกเมาส์หรือการกดแป้นพิมพ์ เหตุการณ์จะถูกจัดเก็บใน Queue และประมวลผลตามลำดับที่เกิดขึ้น



# ตัวอย่างการใช้งาน Stack

**Stack** เป็นโครงสร้างข้อมูลทำงานแบบ **Last-In-First-Out (LIFO)** ซึ่งหมายความว่าข้อมูลที่ถูกเพิ่มเข้ามาล่าสุดจะถูกนำออกก่อน ตัวอย่างการใช้งาน **Stack** มีหลายแบบ เช่น การย้อนกลับในเว็บเบราว์เซอร์ การจัดการกับการเรียกใช้ฟังก์ชันในโปรแกรมมิ่ง หรือการจัดการกับข้อมูลในเกม

## 1.การย้อนกลับในเว็บเบราว์เซอร์:

- เมื่อคุณเปิดหน้าเว็บใหม่หน้านั้นจะถูกเพิ่มลงใน **Stack**
- เมื่อคุณกดปุ่มย้อนกลับ (**Back**)หน้านั้นจะถูกนำออกจาก **Stack** และคุณจะไปยังหน้าก่อนหน้า

## 2.การจัดการกับการเรียกใช้ฟังก์ชันในโปรแกรมมิ่ง:

- เมื่อฟังก์ชันหนึ่งเรียกใช้งานฟังก์ชันอื่น ฟังก์ชันที่ถูกเรียกจะถูกเพิ่มลงใน **Stack**
- เมื่อฟังก์ชันนั้นทำงานเสร็จ จะถูกนำออกจาก **Stack** และโปรแกรมจะกลับไปทำงานต่อในฟังก์ชันที่เรียกใช้งาน

## 3.การตรวจสอบวงเล็บในนิพจน์ทางคณิตศาสตร์:

- เมื่อพบวงเล็บเปิด ( จะถูกเพิ่มลงใน **Stack**
- เมื่อพบวงเล็บปิด ) จะตรวจสอบว่ามีวงเล็บเปิดใน **Stack** หรือไม่ ถ้ามีจะนำออกจาก **Stack**

## 4.การจัดการกับการเลิกทำ (**Undo**) ในโปรแกรมแก้ไขข้อความ:

- เมื่อคุณทำการแก้ไขข้อความ การกระทำเหล่านั้นจะถูกเพิ่มลงใน **Stack**
- เมื่อคุณกดปุ่มเลิกทำ (**Undo**) การกระทำล่าสุดจะถูกนำออกจาก **Stack** และยกเลิกการกระทำนั้น

## 5.การจัดการกับข้อมูลในเกม:

- ในเกมที่มีการย้อนกลับไปยังสถานะก่อนหน้า เช่น เกมหมากรุก การเคลื่อนไหวแต่ละครั้งจะถูกเพิ่มลงใน **Stack**
- เมื่อคุณต้องการย้อนกลับไปยังการเคลื่อนไหวก่อนหน้า การเคลื่อนไหวล่าสุดจะถูกนำออกจาก **Stack**

# ตัวอย่างการใช้งาน Queue

**Queue** เป็นโครงสร้างข้อมูลที่มีประโยชน์มากในหลายๆ สถานการณ์ที่ต้องการจัดการข้อมูลตามลำดับเวลา

- การจัดการงานในระบบคอมพิวเตอร์: งานที่เข้ามาก่อนจะถูกประมวลผลก่อน.
- การจัดการคิวในร้านค้า: ลูกค้าที่มาถึงก่อนจะได้รับบริการก่อน.
- การจัดการข้อมูลในเครือข่าย: แพ็กเก็ตข้อมูลที่ส่งมาก่อนจะถูกประมวลผลก่อน.