Chapter 4 List-Based Collections

หน้า 1 ภาพรวมของบทเรียน

- แนะนำบทเรียนที่จะครอบคลุมเนื้อหาเกี่ยวกับโครงสร้างข้อมูลพื้นฐาน เช่น รายการเชื่อมโยง (Linked Lists), อาร์เรย์ (Array), สแต็ก (Stack), และคิว (Queue)พร้อมกับวัตถุประสงค์หลักของบทเรียน

หน้า 2-3 วัตถุประสงค์ของบทเรียน

- ระบุวัตถุประสงค์สำคัญของบทเรียน ได้แก่ การพัฒนาโซลูชันสำหรับปัญหาโดยใช้โครงสร้างข้อมูลต่างๆ และการใช้โครงสร้างเหล่านี้เพื่อการเข้าถึงข้อมูลอย่างมีประสิทธิภาพ

เมื่อเราศึกษาบทนี้จบ เราควรจะสามารถ

1. พัฒนาโซลูชันสำหรับปัญหาโดยใช้เทคนิคของรายการเชื่อมโยง (linked list), อาร์เรย์ (array), สแต็ก (stack) หรือคิว (queue).
2. พัฒนาโซลูชันสำหรับปัญหาโดยใช้รายการเชื่อมโยง, อาร์เรย์, สแต็ก และคิว เพื่อการเข้าถึงข้อมูล

หน้า 4-5 โครงสร้างข้อมูล List และ Linked List

- อธิบายความหมายและการทำงานของ List และ Linked List, รวมถึงคุณสมบัติพื้นฐานและการใช้งานในการเขียนโปรแกรม

คำจำกัดความของ List

- List คือโครงสร้างข้อมูลที่เก็บข้อมูลแบบลำดับ (Sequence) ซึ่งสามารถเข้าถึงข้อมูลแต่ละตัวได้โดยใช้ดัชนี (Index) ที่กำหนดตำแหน่งของข้อมูลใน List นั้นๆ

การดัชนี (Indexing)

- List มีความสามารถในการใช้ดัชนีทั้งในทิศทางข้างหน้าและทิศทางถอยหลัง
- **Forward direction indexing**: การใช้ดัชนีเริ่มจาก 0 ไปถึง n-1 สำหรับ List ที่มี n ข้อมูล (ตัวอย่างในภาพคือ MON ถึง FRI มีดัชนี 0 ถึง 4).
- **Backward direction indexing**: การใช้ดัชนีเริ่มจาก -1 ไปถึง -n เพื่ออ้างอิงถึงข้อมูลจากท้ายสุดของ List กลับไปหาต้น List (ตัวอย่างในภาพคือ FRI คือ -1 และ MON คือ -5).

ตัวอย่างข้อมูลใน List

- **people**: แสดง List ของชื่อผู้คน ["Alice", "Bob", "Carl"].
- **money**: แสดง List ของจำนวนเงิน [200, 20.75, 350.00].
- **id**: แสดง List ของรหัสประจำตัว ['"AC4X"', 2911, '"X703"']

คำจำกัดความของ Linked List

- ประกอบด้วยสองส่วนหลักคือ ข้อมูล (Data) และตัวชี้ (Pointer) ที่ชี้ไปยังโหนดถัดไปในรายการ โหนดสุดท้ายจะชี้ไปยังค่าว่าง (Null) ซึ่งบ่งบอกว่าเป็นจุดสิ้นสุดของ Linked List

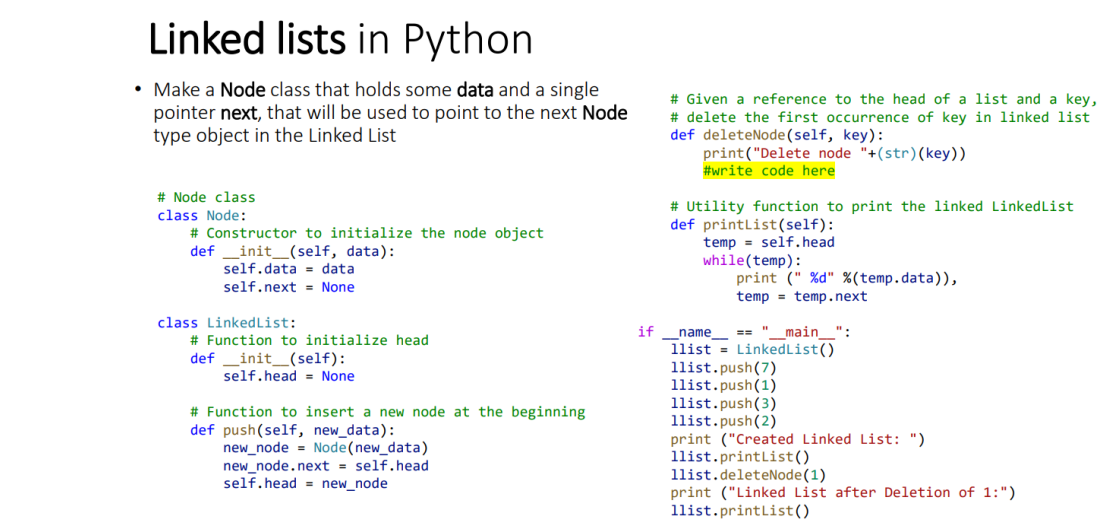
คุณสมบัติของ Linked List

- มีความยืดหยุ่นในการจัดการข้อมูล เนื่องจากสามารถเพิ่มหรือลบโหนดได้อย่างง่ายดายโดยไม่จำเป็นต้องจัดเรียงข้อมูลใหม่ทั้งหมดเหมือนในอาร์เรย์
- สามารถใช้สำหรับการสร้างโครงสร้างข้อมูลแบบต่างๆ เช่น สแต็กและคิว

ประเภทของ Linked List:

- **Singly Linked Lists**: โหนดแต่ละตัวมีตัวชี้เพียงตัวเดียวที่ชี้ไปยังโหนดถัดไป
- **Doubly Linked Lists**: โหนดแต่ละตัวมีสองตัวชี้ หนึ่งตัวชี้ไปยังโหนดถัดไปและอีกตัวหนึ่งชี้กลับไปยังโหนดก่อนหน้า
- **Circular Linked Lists**: โหนดสุดท้ายในรายการจะชี้กลับไปยังโหนดแรก ทำให้รายการมีลักษณะเป็นวงกลม
- **Circular Doubly Linked Lists**: เช่นเดียวกับ Circular Linked Lists แต่แต่ละโหนดมีการชี้ไปและกลับทั้งสองทิศทาง

หน้า 6-8 การใช้งาน Linked List ใน Python



Linked lists in Python

- Make a **Node** class that holds some **data** and a single pointer **next**, that will be used to point to the next **Node** type object in the Linked List

```
# Node class
class Node:
    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
```

```
    # Given a reference to the head of a list and a key,
    # delete the first occurrence of key in linked list
    def deleteNode(self, key):
        print("Delete node "+(str)(key))
        #write code here

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (" %d" %(temp.data)),
            temp = temp.next

if __name__ == "__main__":
    llist = LinkedList()
    llist.push(7)
    llist.push(1)
    llist.push(3)
    llist.push(2)
    print ("Created Linked List: ")
    llist.printList()
    llist.deleteNode(1)
    print ("Linked List after Deletion of 1:")
    llist.printList()
```

ตัวอย่างการใช้งานโครงสร้างข้อมูล Linked List ในภาษา Python และอธิบายวิธีการสร้างและจัดการกับ Linked List ผ่านตัวอย่างโค้ด

1. Node Class

    - Class Node ประกอบด้วย Constructor (__init__) ที่รับพารามิเตอร์ data ซึ่งเป็นข้อมูลที่โหนดจะเก็บ และ next เป็นตัวชี้ไปยังโหนดถัดไปใน Linked List โดยมีค่าเริ่มต้นเป็น None เพื่อบ่งบอกว่าโหนดนี้คือโหนดสุดท้าย หรือยังไม่ได้เชื่อมต่อกับโหนดอื่น.

2. LinkedList Class

    - Class LinkedList ประกอบด้วย Constructor (__init__) สำหรับสร้าง Linked List โดยเริ่มต้น head คือ None แสดงถึงรายการที่ว่างเปล่า.
    - Method push(self, new_data) ใช้สำหรับเพิ่มโหนดใหม่เข้าที่ตำแหน่งแรกของ Linked List. โหนดใหม่จะชี้ไปยังโหนดเดิมที่เคยเป็น head และโหนดใหม่นี้จะกลายเป็น head ใหม่ของ Linked List.

3. การทำงานของ Linked List

    - ตัวอย่างการทำงานของโปรแกรมในส่วน if __name__ == "__main__": แสดงถึงการสร้าง Linked List ใหม่และการเพิ่มข้อมูลตัวเลขลงไป หลังจากนั้นมีการแสดงข้อมูลก่อนและหลังจากการลบโหนดที่มีข้อมูลเป็น 1 โดยใช้ฟังก์ชัน deleteNode.

4. ฟังก์ชัน deleteNode(self, key)

    - ฟังก์ชันนี้ใช้สำหรับลบโหนดที่มีข้อมูลตรงกับค่า key ที่ระบุมา แสดงวิธีการค้นหาโหนดใน Linked List และลบโหนดนั้นออก.

5. การขยายโค้ด

    - ส่วน "Linked lists in Python - works" แนะนำให้เติมโค้ดในฟังก์ชัน deleteData และ printReverse โดยฟังก์ชันเหล่านี้จะใช้สำหรับลบข้อมูลและแสดงข้อมูลใน Linked List แบบย้อนกลับ.

หน้า 10-13 โครงสร้างข้อมูล Array และการใช้งานใน Python

คำจำกัดความของ Array

- Array เป็นโครงสร้างข้อมูลที่เก็บข้อมูลหลายๆ อย่างในรูปแบบที่ต่อเนื่องกันภายในหน่วยความจำ ซึ่งข้อมูลที่จัดเก็บใน Array ต้องเป็นชนิดเดียวกัน ทำให้การเข้าถึงข้อมูลเป็นไปได้ง่ายและรวดเร็วผ่านดัชนี (index).

การใช้งาน Array

- อาร์เรย์ถูกนำมาใช้ในกระบวนการแก้ไขปัญหาต่างๆ เช่น การจัดเก็บข้อมูลที่มีลักษณะเหมือนกัน ไม่ว่าจะเป็นอายุของบุคคล, อุณหภูมิ, ชื่อบุคคล หรือข้อมูลอื่นๆ ที่มีความสอดคล้องกัน โดยการใช้อาร์เรย์ช่วยให้การอ่านและการใช้งานข้อมูลเหล่านี้ง่ายขึ้น.

การแสดง Array

- ในภาพ, Array มีความยาว 10 และแสดงดัชนีตั้งแต่ 0 ถึง 9 โดยมี "Element at index 8" ถูกเน้น แสดงให้เห็นว่าแต่ละข้อมูลในอาร์เรย์สามารถเข้าถึงได้โดยใช้ดัชนี ซึ่งเป็นตัวเลขที่บ่งบอกตำแหน่งของข้อมูลนั้นๆ ในอาร์เรย์.

Array in Python

- Array in Python can be created by importing array module.

```
# importing "array" for array creations
import array as arr

# creating an array with integer type
a = arr.array('i', [1, 2, 3])
```

Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

แสดงถึงการใช้งานอาร์เรย์ในภาษา Python โดยใช้โมดูล array ซึ่งเป็นโมดูลที่ให้ความสามารถในการสร้างอาร์เรย์ที่มีข้อมูลชนิดเดียวกันแต่ใช้หน่วยความจำได้อย่างมีประสิทธิภาพ

1. **การนำเข้าโมดูล array**

   - เริ่มต้นด้วยการนำเข้าโมดูล array ซึ่งจำเป็นต้องทำก่อนที่จะสามารถสร้างอาร์เรย์ได้: import array as arr

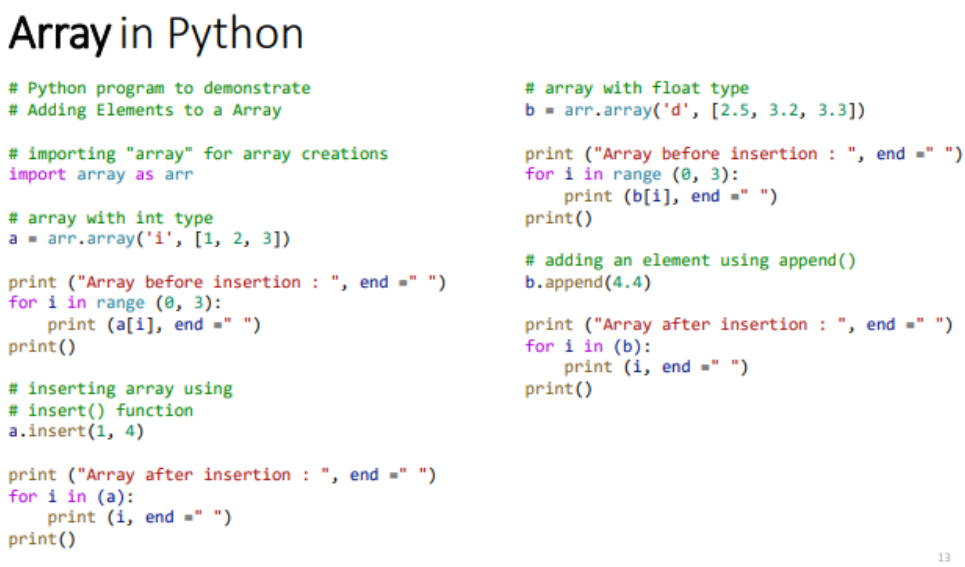
2. **การสร้างอาร์เรย์**

   - ตัวอย่างการสร้างอาร์เรย์ชนิดข้อมูลเต็ม (integer): a = arr.array('i', [1, 2, 3])
   - ที่นี่ 'i' คือรหัสประเภทข้อมูลที่ใช้ในการบอกว่าอาร์เรย์นี้จะเก็บข้อมูลชนิด integer ที่มีขนาด 2 ไบต์ต่อองค์ประกอบ

3. **รหัสประเภทข้อมูลและขนาด**

   - ภาพแสดงรหัสประเภทข้อมูล (Type Code) สำหรับอาร์เรย์ที่รองรับในโมดูล array พร้อมกับแมปปิ้งกับ C Type และ Python Type ที่สอดคล้องกัน และแสดงขนาดของข้อมูลแต่ละประเภทเป็นไบต์

- ตัวอย่างรวมถึงข้อมูลชนิด signed char ('b'), unsigned char ('B'), และชนิดข้อมูลอื่นๆ เช่น float ('f') และ double ('d')



ภาพนี้ให้ข้อมูลเกี่ยวกับวิธีการเพิ่มข้อมูลเข้าไปในอาร์เรย์ในภาษา Python โดยใช้โมดูล array และนำเสนอตัวอย่างโค้ดเกี่ยวกับการใช้งานอาร์เรย์ทั้งสำหรับข้อมูลชนิดเต็ม (integer) และข้อมูลชนิดทศนิยม (float):

1. **การนำเข้าโมดูลและสร้างอาร์เรย์**

   - โค้ดเริ่มต้นด้วยการนำเข้าโมดูล array และสร้างอาร์เรย์สองตัว
     - อาร์เรย์ a สำหรับข้อมูลชนิด integer โดยใช้รหัสประเภท 'i'.
     - อาร์เรย์ b สำหรับข้อมูลชนิด float โดยใช้รหัสประเภท 'd'.

2. **การแสดงอาร์เรย์ก่อนการเพิ่มข้อมูล**

   - ใช้ loop for เพื่อพิมพ์ข้อมูลในอาร์เรย์ a และ b ก่อนทำการเพิ่มข้อมูลใหม่.

3. **การเพิ่มข้อมูลในอาร์เรย์**

   - สำหรับอาร์เรย์ a ใช้ฟังก์ชัน insert() สำหรับการเพิ่มข้อมูลใหม่ที่ตำแหน่งที่ระบุ ในตัวอย่างนี้เพิ่มข้อมูล 4 ที่ตำแหน่งแรกของอาร์เรย์.
   - สำหรับอาร์เรย์ b ใช้ฟังก์ชัน append() เพื่อเพิ่มข้อมูล 4.4 ไปที่ตำแหน่งสุดท้ายของอาร์เรย์.

4. **การแสดงอาร์เรย์หลังจากการเพิ่มข้อมูล**

- พิมพ์ข้อมูลในอาร์เรย์ a และ b อีกครั้งหลังจากเพิ่มข้อมูลใหม่เพื่อแสดงผลลัพธ์ที่เปลี่ยนแปลงไป.

5. **ความสามารถของ Array ใน Python**

- อธิบายว่าอาร์เรย์ใน Python สามารถจัดการข้อมูลในรูปแบบที่มีประสิทธิภาพ เนื่องจากใช้หน่วยความจำได้แบบมีโครงสร้างและจัดเรียงได้อย่างเป็นระเบียบเรียบร้อย ทำให้ง่ายต่อการเข้าถึงและแก้ไขข้อมูล.

หน้า 15-17 โครงสร้างข้อมูล Stack และการใช้งานใน Python

คำจำกัดความของ Stack

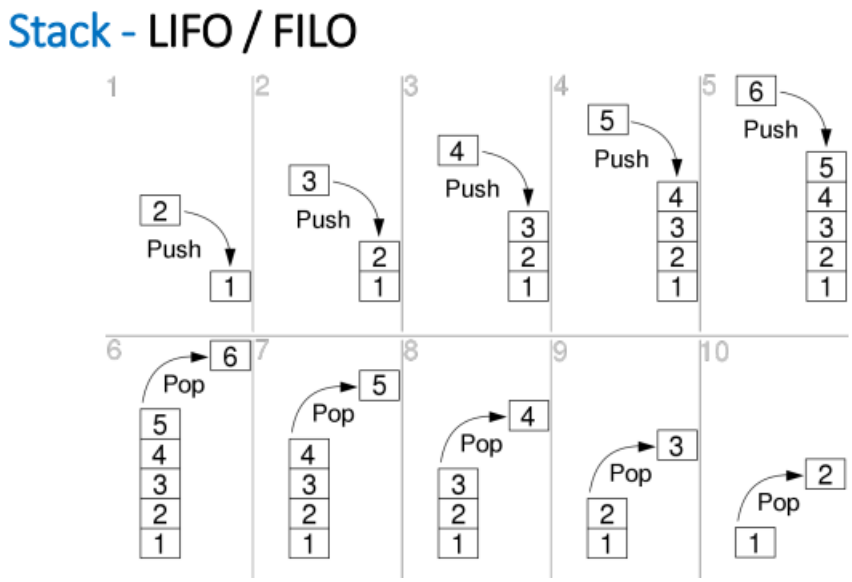
- Stack เป็นโครงสร้างข้อมูลเชิงเส้นที่ทำงานตามลำดับการดำเนินการเฉพาะ ข้อมูลที่เพิ่มเข้ามาล่าสุดจะถูกนำออกก่อน (LIFO) หรือข้อมูลที่เพิ่มเข้ามาก่อนจะถูกนำออกทีหลัง (FILO) ขึ้นอยู่กับการมองแบบต่างๆ ของการดำเนินการ.

การทำงานของ Stack

- **Push**: เพิ่ม (push) ข้อมูลเข้าสู่ stack โดยข้อมูลจะถูกเพิ่มเข้าไปที่ด้านบนสุดของ stack. ในภาพ, ข้อมูล C ถูก push เข้าไปบนข้อมูล B และ A ที่มีอยู่แล้วใน stack.
- **Pop**: ลบ (pop) ข้อมูลจาก stack โดยการลบข้อมูลที่ด้านบนสุดของ stack ซึ่งในตัวอย่างคือการนำ C ออกจาก stack.

ประเภทของ Stack:

- ภาพนี้แสดงการทำงานของ stack โดยมีการรักษาตัวชี้ไปที่ top ของ stack เสมอ เพื่อทราบว่าข้อมูลใดอยู่ด้านบนสุด และสามารถเข้าถึงข้อมูลนั้นได้ทันทีเมื่อต้องการทำการ pop



ภาพนี้แสดงซีรีส์ของกระบวนการในการใช้งาน Stack ตามหลักการ LIFO (Last In First Out) หรือ FILO (First In Last Out) โดยเป็นตัวอย่างการทำงานทีละขั้นตอนจากการ push (เพิ่ม) ข้อมูลเข้าสู่ stack ไปจนถึงการ pop (ลบ) ข้อมูลออกจาก stack

1. **Push Operations**

   - ในขั้นตอน 1 ถึง 5, แสดงกระบวนการเพิ่มข้อมูลเข้าสู่ stack:

     - ขั้นตอนที่ 1: เริ่มด้วย stack ที่มีข้อมูล '1', จากนั้น push '2' เข้าไปบน stack.
     - ขั้นตอนที่ 2: ตามด้วยการ push '3' ไปบน stack.
     - ขั้นตอนที่ 3: push '4' เข้าไปต่อ.
     - ขั้นตอนที่ 4: ดำเนินการ push '5' เข้าไปใน stack.
     - ขั้นตอนที่ 5: push '6' เข้าไปบนสุด.

2. **Pop Operations**

   - ในขั้นตอน 6 ถึง 10, แสดงการลบข้อมูลออกจาก stack ตามหลักการ LIFO:

     - ขั้นตอนที่ 6: เริ่มการ pop ข้อมูล '6' ออกจาก stack.
     - ขั้นตอนที่ 7: จากนั้น pop '5'.
     - ขั้นตอนที่ 8: pop '4' ออก.
     - ขั้นตอนที่ 9: ต่อด้วยการ pop '3'.
     - ขั้นตอนที่ 10: ปิดท้ายด้วยการ pop '2', ทิ้งเพียง '1' อยู่ใน stack

โครงสร้างข้อมูล Stack ในภาษา Python รวมทั้งการใช้งานและวิธีการทำงานของฟังก์ชันต่างๆ ที่เกี่ยวข้องกับ stack

1. **การทำงานของ Stack**

   - Stack เป็นโครงสร้างข้อมูลเชิงเส้นที่ทำงานตามหลักการ "Last In, First Out" (LIFO) หรือ "First In, Last Out" (FILO) โดยวัตถุใดที่ถูกเพิ่มเข้าไปล่าสุดจะถูกนำออกก่อนตามหลักการนี้.
   - ใน Stack, การเพิ่ม (push) และการลบ (pop) วัตถุจะทำที่ด้านเดียวของ Stack เท่านั้น ซึ่งก็คือที่ปลาย Stack.

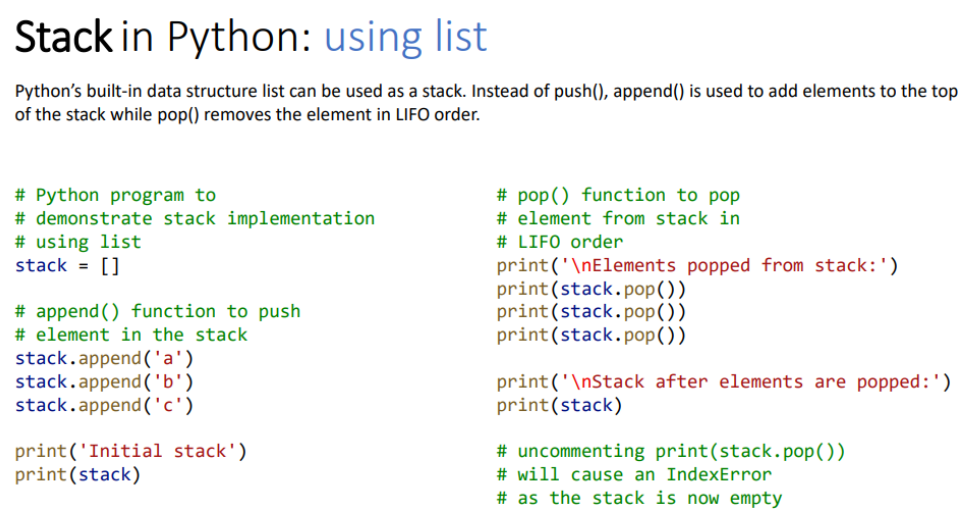
2. **ฟังก์ชันที่ใช้ใน Stack**

   - empty() ตรวจสอบว่า stack ว่างหรือไม่ มีเวลาประมวลผล O(1)
   - size() คืนค่าขนาดของ stack มีเวลาประมวลผล O(1)
   - top() หรือ peek() คืนค่าอ้างอิงไปยังองค์ประกอบที่อยู่บนสุดของ stack มีเวลาประมวลผล O(1)
   - push(a) เพิ่มวัตถุ a ไปที่ด้านบนของ stack มีเวลาประมวลผล O(1)
   - pop() ลบวัตถุที่อยู่บนสุดของ stack มีเวลาประมวลผล O(1)

3. **การนำไปใช้งาน Stack ใน Python**

   - Stack สามารถถูกสร้างขึ้นใน Python ด้วยวิธีต่างๆ เช่น

     - การใช้งาน list ซึ่งเป็นวิธีที่ง่ายและสะดวก
     - การใช้ collections.deque ที่เหมาะสำหรับการทำงานที่ต้องการการเข้าถึงข้อมูลแบบเร็วขึ้น
     - การใช้ queue.LifoQueue ที่ให้ฟีเจอร์เพิ่มเติมสำหรับการจัดการข้อมูลแบบ LIFO



Stack in Python: using list

Python's built-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of the stack while pop() removes the element in LIFO order.

```
# Python program to
# demonstrate stack implementation
# using list
stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)
```

```
# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

โค้ดสำหรับการใช้งานโครงสร้างข้อมูล Stack โดยใช้ list ในภาษา Python

1. **การสร้าง Stack**

- ใช้ list ธรรมดาใน Python เพื่อเป็นตัวแทนของ stack (stack = []).

2. **การเพิ่มข้อมูล (Push)**

   - ใช้ฟังก์ชัน append() เพื่อเพิ่มข้อมูลเข้าไปใน stack. ในตัวอย่าง, ข้อมูล 'a', 'b', และ 'c' ถูกเพิ่มเข้าไปใน stack ตามลำดับ ซึ่ง 'c' จะอยู่ที่ด้านบนสุดของ stack หลังจากการเพิ่มเสร็จสิ้น.

3. **การแสดง Stack ก่อนการลบข้อมูล**

   - พิมพ์ stack เพื่อแสดงข้อมูลภายในก่อนทำการลบข้อมูล (print('Initial stack') และ print(stack)).

4. **การลบข้อมูล (Pop)**

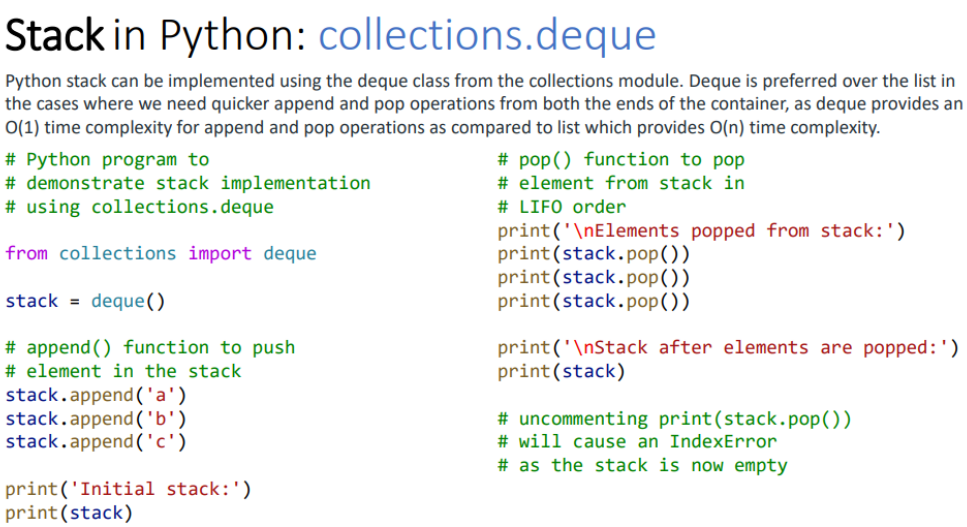
   - ใช้ฟังก์ชัน pop() เพื่อลบข้อมูลจากด้านบนสุดของ stack. ในตัวอย่าง, ลบ 'c', 'b', และ 'a' ออกจาก stack ตามลำดับ. ทุกครั้งที่ pop() ถูกเรียก, ข้อมูลที่อยู่ด้านบนสุดจะถูกลบออกและคืนค่านั้นกลับมา.

5. **การแสดง Stack หลังจากการลบข้อมูล**

   - พิมพ์ stack อีกครั้งเพื่อแสดงข้อมูลภายในหลังจากทำการลบข้อมูลทั้งหมด (print('\nStack after elements are popped:') และ print(stack)).

6. **คำเตือนเกี่ยวกับการเรียก pop() บน Stack ที่ว่าง**

   - หมายเหตุในโค้ดบอกว่าหากคุณไม่ได้ใส่ความคิดเห็น (uncomment) บรรทัดที่เรียก pop() บน stack ที่ว่างเปล่า จะเกิดข้อผิดพลาด IndexError เนื่องจากไม่มีองค์ประกอบที่จะลบออก.



Stack in Python: collections.deque

Python stack can be implemented using the deque class from the collections module. Deque is preferred over the list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.

```
# Python program to
# demonstrate stack implementation
# using collections.deque

from collections import deque

stack = deque()

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack:')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

โค้ดในภาษา Python สำหรับการใช้งานโครงสร้างข้อมูล Stack โดยใช้ collections.deque เพื่อการทำงานที่มีประสิทธิภาพสูงกว่าการใช้ list ปกติ

1. **นำเข้าและการสร้าง Stack**

   - เริ่มต้นด้วยการนำเข้า deque จากโมดูล collections ซึ่งเป็นโครงสร้างข้อมูลที่ใช้สำหรับจัดการข้อมูลที่ต้องการการเข้าถึงทั้งทางด้านหน้าและด้านหลังของโครงสร้างอย่างรวดเร็ว.
   - สร้าง object deque ซึ่งจะใช้เป็น stack ในโค้ด (stack = deque()).

2. **การเพิ่มข้อมูล (Push)**

   - ใช้ฟังก์ชัน append() เพื่อเพิ่มข้อมูลเข้าไปใน stack. ข้อมูล 'a', 'b', และ 'c' ถูกเพิ่มเข้าไปใน stack ตามลำดับ.

3. **แสดง Stack ก่อนการลบข้อมูล**

   - พิมพ์ข้อมูลใน stack ก่อนทำการลบเพื่อแสดงสถานะเริ่มต้นของ stack (print('Initial stack:') และ print(stack)).

4. **การลบข้อมูล (Pop)**

   - ใช้ฟังก์ชัน pop() สำหรับลบข้อมูลจากด้านบนของ stack. การลบข้อมูลจาก stack ตามหลักการ LIFO โดยลบ 'c', 'b', และ 'a' ออกจาก stack ตามลำดับ.

5. **แสดง Stack หลังจากการลบข้อมูล**

   - พิมพ์ข้อมูลใน stack หลังจากทำการลบข้อมูลทั้งหมดเพื่อแสดงสถานะที่ว่างเปล่า (print('\nStack after elements are popped:') และ print(stack)).

6. **การจัดการกับสถานะ Stack ที่ว่าง**

   - มีคำเตือนเกี่ยวกับการเรียก pop() บน stack ที่ว่างเปล่า ซึ่งจะทำให้เกิด IndexError ซึ่งแสดงถึงการเรียกข้อมูลจาก stack ที่ไม่มีข้อมูลอยู่.

Stack in Python: queue module

Queue module also has a LIFO Queue, which is basically a Stack. Data is inserted into Queue using the put() function and get() takes data out from the Queue.

```
# Python program to
# demonstrate stack implementation
# using queue module
from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize=3)

# qsize() show the number of elements
# in the stack
print(stack.qsize())

# put() function to push
# element in the stack
stack.put('a')
stack.put('b')
stack.put('c')

print("Full: ", stack.full())
print("Size: ", stack.qsize())

# get() function to pop
# element from stack in
# LIFO order
print('\nElements popped from the stack')
print(stack.get())
print(stack.get())
print(stack.get())

print("\nEmpty: ", stack.empty())
```

23

การใช้งานโครงสร้างข้อมูล Stack โดยใช้โมดูล queue ใน Python, โดยเฉพาะการใช้คลาส LifoQueue ซึ่งจัดการข้อมูลตามหลักการ Last In First Out (LIFO) ที่เหมาะสมสำหรับการทำงานแบบ Stack:

1. **การนำเข้าและสร้าง Stack**

   - เริ่มต้นด้วยการนำเข้าคลาส LifoQueue จากโมดูล queue.
   - สร้าง Stack โดยใช้ LifoQueue(maxsize=3) ที่กำหนดขนาดสูงสุดเป็น 3.

2. **ฟังก์ชันต่างๆ ใน LifoQueue**

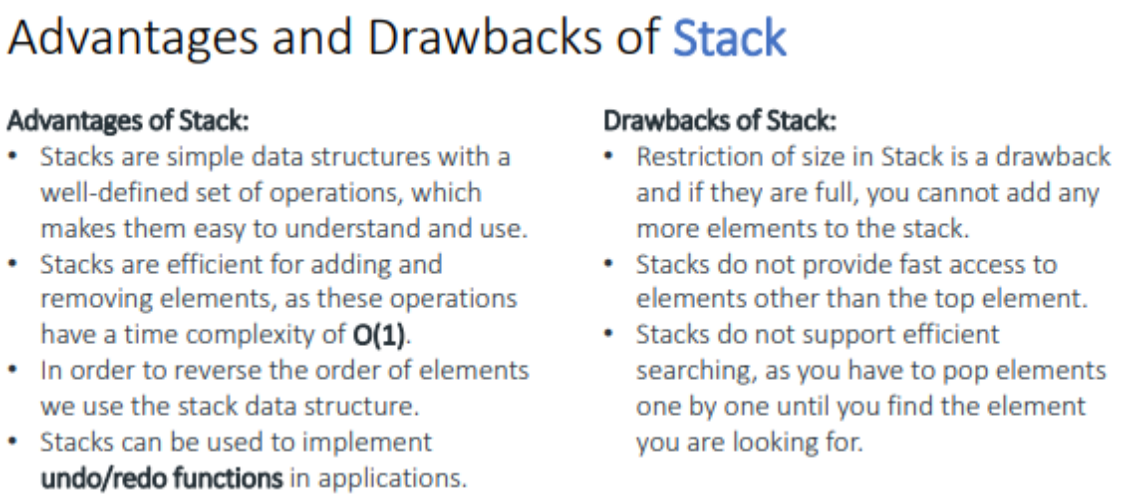
   - empty(): ตรวจสอบว่า queue ว่างหรือไม่.
   - full(): ตรวจสอบว่า queue ได้เต็มตามความจุที่กำหนดหรือยัง.
   - qsize(): คืนค่าจำนวนขององค์ประกอบที่มีใน queue.
   - put(item): เพิ่มองค์ประกอบเข้า queue.
   - get(): ลบและคืนค่าองค์ประกอบจากด้านบนสุดของ queue.
   - put_nowait(item): เพิ่มองค์ประกอบโดยไม่รอถ้า queue ยังไม่เต็ม.
   - get_nowait(): ลบและคืนค่าองค์ประกอบโดยไม่รอถ้า queue ไม่ว่าง.

3. **ตัวอย่างการใช้งาน Stack**

   - ใช้ put() เพื่อเพิ่ม 'a', 'b', 'c' เข้าสู่ stack และแสดงสถานะ stack หลังจากการเพิ่มข้อมูลเหล่านี้.
   - ใช้ get() เพื่อลบข้อมูลจาก stack และแสดงสถานะหลังจากการลบข้อมูล.
   - แสดงจำนวนข้อมูลใน stack และสถานะของ stack (ว่างหรือเต็ม) ทั้งก่อนและหลังจากการดำเนินการต่างๆ.

4. **การจัดการข้อผิดพลาด**

    - แสดงคำเตือนเกี่ยวกับการเรียกใช้ get() บน stack ที่ว่างเปล่าซึ่งอาจทำให้เกิดข้อผิดพลาด IndexError เนื่องจากการพยายามลบข้อมูลจาก stack ที่ไม่มีข้อมูลใดๆ อยู่.



ข้อดีและข้อเสียของโครงสร้างข้อมูล Stack

ข้อดีของ Stack

1. **ความเรียบง่ายและชัดเจน**

    - Stack เป็นโครงสร้างข้อมูลที่มีการดำเนินการที่ชัดเจนและเฉพาะเจาะจง ทำให้ง่ายต่อการเรียนรู้และใช้งานในการเพิ่มหรือลบข้อมูล.

2. **ประสิทธิภาพในการดำเนินการ**

    - การเพิ่ม (push) และการลบ (pop) ข้อมูลใน stack มีความซับซ้อนในการดำเนินการเพียง O(1) หมายความว่าการดำเนินการเหล่านี้ใช้เวลาคงที่และเป็นไปอย่างรวดเร็ว.

3. **การประยุกต์ใช้งาน**

    - Stack สามารถใช้ในการพลิกกลับลำดับขององค์ประกอบได้ และยังใช้ในการพัฒนาฟังก์ชันการทำงานแบบย้อนกลับ/ทำซ้ำ (undo/redo) ในแอปพลิเคชันต่างๆ.

ข้อเสียของ Stack

1. **จำกัดขนาด**

    - ในบางการใช้งาน Stack อาจมีข้อจำกัดในแง่ขนาด ซึ่งหาก stack เต็มแล้วจะไม่สามารถเพิ่มข้อมูลเข้าไปได้อีก ทำให้ต้องมีการจัดการทรัพยากรอย่างรอบคอบ.

2. **การเข้าถึงข้อมูลที่จำกัด**

    - Stack ไม่สามารถเข้าถึงข้อมูลที่ไม่ได้อยู่ที่ด้านบนสุดได้โดยตรง หากต้องการข้อมูลที่อยู่ลึกลงไปจำเป็นต้องลบข้อมูลที่อยู่ด้านบนออกก่อน ซึ่งทำให้การเข้าถึงข้อมูลไม่รวดเร็วและอาจส่งผลต่อประสิทธิภาพในบางสถานการณ์.

3. **ความยากในการค้นหา**

    - Stack ไม่สนับสนุนการค้นหาข้อมูลอย่างมีประสิทธิภาพ เพราะต้องทำการ pop องค์ประกอบออกหนึ่งๆ เพื่อเข้าถึงข้อมูลที่ต้องการ ทำให้เสียเวลาและทรัพยากรในกระบวนการนี้.

หน้า 25-34 โครงสร้างข้อมูล Queue และการใช้งานใน Python

โครงสร้างข้อมูลที่เรียกว่า "Queue" หรือ "คิว" ซึ่งเป็นโครงสร้างข้อมูลแบบเชิงเส้นที่ทำงานตามหลักการ First In First Out (FIFO)

1. **คำจำกัดความของ Queue**

    - Queue เป็นโครงสร้างข้อมูลที่จัดการข้อมูลตามหลักการ FIFO หรือ First In First Out ซึ่งหมายความว่าข้อมูลที่ถูกเพิ่มเข้ามาใน queue ก่อนจะเป็นข้อมูลแรกที่ถูกนำออกจาก queue เมื่อมีการลบข้อมูล.

2. **การทำงานของ Queue**

    - ในภาพแสดงถึงการใช้งาน queue พร้อมทั้งจุดที่เรียกว่า Front/Head และ Back/Tail/Rear:

        - Enqueue: การเพิ่มข้อมูลใหม่เข้าไปใน queue ทำที่ด้านท้ายของ queue (ตัวอย่างคือการเพิ่มข้อมูล '9' ในภาพ).
        - Dequeue: การนำข้อมูลออกจาก queue ทำที่ด้านหน้าของ queue (ตัวอย่างคือการนำข้อมูล '2' ออกจาก queue ในภาพ).

3. **การใช้งาน Queue**

    - Queue มักใช้ในสถานการณ์ที่ต้องการบริการหรือการจัดการข้อมูลตามลำดับที่เข้ามา เช่น การจัดคิวในระบบธนาคารหรือการจัดการกระบวนการในระบบปฏิบัติการ โดยผู้ใช้หรือข้อมูลที่เข้ามาก่อนจะได้รับการบริการก่อน.

ประเภทต่างๆ ของ Queue ซึ่งเป็นโครงสร้างข้อมูลที่ใช้ในสถานการณ์ต่างๆ ตามลักษณะการใช้งานที่ต้องการ:

1. **Circular Queue**:

    - Queue ประเภทนี้จะมีตำแหน่งท้ายสุดของ queue เชื่อมกลับไปยังตำแหน่งแรกทำให้เป็นรูปวงกลม ช่วยให้การจัดการคิวมีประสิทธิภาพมากขึ้นโดยไม่ต้องเสียเวลาในการจัดการพื้นที่หน่วยความจำเมื่อคิวว่างเปล่า

2. **Input Restricted Queue**

    - นี่เป็นรูปแบบพิเศษของ queue ที่อนุญาตให้เพิ่มข้อมูลเข้า queue ที่ปลายเดียวเท่านั้น ขณะที่การนำข้อมูลออกสามารถทำได้ทั้งสองปลาย ช่วยลดความซับซ้อนในการใช้งานในบางสถานการณ์

3. **Output Restricted Queue**

    - คล้ายกับ Input Restricted Queue แต่กลับกัน คือ สามารถเพิ่มข้อมูลเข้าได้ทั้งสองปลาย แต่จะนำข้อมูลออกได้ที่ปลายเดียวเท่านั้น โดยปกติจะเป็นปลายทางด้านหน้าของ queue

4. **Double Ended Queue (Deque)**

    - Deque อนุญาตให้เพิ่มหรือลบข้อมูลที่ปลายใดปลายหนึ่งของ queue ช่วยให้มีความยืดหยุ่นมากขึ้นในการใช้งานทั้งในลักษณะ stack และ queue

5. **Priority Queue**

    - ไม่เหมือนกับ queue ปกติที่ทำงานตามหลักการ FIFO, priority queue จะเรียงลำดับข้อมูลตามความสำคัญหรือค่าความสำคัญที่กำหนดไว้
    - **Ascending Priority Queue**: ข้อมูลที่มีค่าน้อยที่สุดจะอยู่หน้าสุดและจะถูกลบออกก่อน
    - **Descending Priority Queue**: ข้อมูลที่มีค่ามากที่สุดจะอยู่หน้าสุดและจะถูกลบออกก่อน

Queue: Circular Queue

ข้อมูลเกี่ยวกับ "Circular Queue" ซึ่งเป็นรูปแบบพิเศษของ queue ที่ใช้โครงสร้างข้อมูลแบบวงกลมหรือที่เรียกว่า "Ring Buffer":

1. **คำจำกัดความของ Circular Queue**

    - Circular Queue คือ queue ที่ตำแหน่งสุดท้ายของ queue จะเชื่อมต่อกลับไปยังตำแหน่งแรกทำให้สามารถนำโครงสร้างนี้มาใช้แทนการเพิ่มหรือลบข้อมูลได้อย่างต่อเนื่องโดยไม่ต้องเปลี่ยนตำแหน่งข้อมูลภายในหน่วยความจำ
    - ทำงานตามหลักการ FIFO (First In First Out) ซึ่งหมายความว่าข้อมูลที่เข้ามาก่อนจะถูกนำออกก่อน

2. **ประโยชน์ของ Circular Queue**

   - **การจัดการหน่วยความจำ**: ช่วยให้สามารถใช้หน่วยความจำที่ไม่ได้ใช้งานใน queue ปกติได้อย่างมีประสิทธิภาพ
   - **ระบบจราจร**: ในระบบจราจรที่ควบคุมด้วยคอมพิวเตอร์ ใช้ circular queues เพื่อควบคุมการเปลี่ยนแปลงสัญญาณไฟจราจรอย่างต่อเนื่องตามเวลาที่กำหนด
   - **การวางแผนการประมวลผลของ CPU (CPU Scheduling)**: ระบบปฏิบัติการมักใช้ circular queue เพื่อจัดการกับกระบวนการที่พร้อมทำงานหรือรอการเกิดเหตุการณ์บางอย่าง

3. **การดำเนินการใน Circular Queue**

   - ตัวดำเนินการหลักใน circular queue ประกอบไปด้วยการเพิ่มข้อมูล (enqueue) ที่ด้านหลังและการลบข้อมูล (dequeue) ที่ด้านหน้า โดยรักษาการเชื่อมต่อแบบวงกลมระหว่างหัวและท้ายของ queue เพื่อให้การดำเนินการเหล่านี้มีประสิทธิภาพและไม่ต้องการการเคลื่อนย้ายข้อมูลภายในโครงสร้าง

Queue: Input Restricted Queue และ Queue: Output Restricted Queue

สองประเภทของ queue ที่มีการจำกัดการดำเนินการที่หนึ่งปลายของ queue ได้แก่ Input Restricted Queue และ Output Restricted Queue

1. **Input Restricted Queue (คิวที่จำกัดการใส่ข้อมูล):**

   - **การใช้งาน**: ใน Input Restricted Queue นี้, การใส่ข้อมูลสามารถทำได้ที่ปลายเดียวของ queue เท่านั้น แต่สามารถลบข้อมูลได้จากทั้งสองปลายของ queue นั่นคือทั้งจาก front และ rear.
   - **ความพิเศษ**: ประเภทนี้ไม่ปฏิบัติตามหลัก FIFO โดยสมบูรณ์ เนื่องจากสามารถลบข้อมูลที่เพิ่งถูกเพิ่มเข้าไปได้ (จากด้านหลัง) โดยทันทีถ้าจำเป็น
   - **ตัวอย่างการใช้งาน**: ใช้ในสถานการณ์ที่ต้องการประมวลผลข้อมูลล่าสุดก่อนในบางครั้ง แต่ยังคงต้องการการเข้าถึงข้อมูลตามลำดับที่เข้ามาเพื่อประมวลผลต่อไป

2. **Output Restricted Queue (คิวที่จำกัดการนำข้อมูลออก)**

   - **การใช้งาน**: ใน Output Restricted Queue, ข้อมูลสามารถเพิ่มเข้าได้จากทั้งสองปลายของ queue (front และ rear) แต่การนำข้อมูลออกสามารถทำได้เฉพาะที่ front เท่านั้น.
   - **ความพิเศษ**: ประเภทนี้ช่วยให้สามารถจัดลำดับความสำคัญของข้อมูลที่จะประมวลผลได้โดยการเพิ่มข้อมูลใหม่ไปที่ตำแหน่งต้น queue หากจำเป็น
   - **ตัวอย่างการใช้งาน**: เหมาะสำหรับสถานการณ์ที่ข้อมูลบางอย่างต้องได้รับการประมวลผลทันทีและได้รับความสำคัญสูงสุด เช่น ในระบบที่มีการจัดการกับข้อมูลระดับสูง

Double Ended Queue (Deque)

ซึ่งเป็นโครงสร้างข้อมูล Queue ที่สามารถดำเนินการได้ทั้งสองปลาย (ทั้งด้านหน้าและด้านหลัง)

คำจำกัดความของ Double Ended Queue (Deque)

- Double Ended Queue เป็นโครงสร้างข้อมูลที่อนุญาตให้สามารถ **เพิ่มข้อมูล (enqueue)** และ **ลบข้อมูล (dequeue)** ได้ทั้งที่ **ปลายหน้า (front)** และ **ปลายหลัง (rear)** ของ queue
- มีความยืดหยุ่นในการจัดการข้อมูลมากกว่า queue แบบปกติที่สามารถเพิ่มและลบข้อมูลได้เพียงด้านเดียว

คุณสมบัติของ Deque

1. **การทำงานที่ปลายทั้งสอง**

   - **Enqueue:** สามารถเพิ่มข้อมูลได้ทั้งที่ด้านหน้าและด้านหลัง
   - **Dequeue:** สามารถลบข้อมูลได้ทั้งที่ด้านหน้าและด้านหลัง

2. **ประสิทธิภาพ:** การดำเนินการทุกอย่างใน Deque มีความซับซ้อนของเวลาเป็น O(1) ทำให้การเพิ่มหรือลบข้อมูลรวดเร็ว

การประยุกต์ใช้งาน (Practical Applications)

1. **จัดเก็บประวัติการใช้งานเว็บเบราว์เซอร์**

   - ใช้สำหรับการเก็บหน้าเว็บที่เปิดก่อนหน้าและสามารถย้อนกลับไปดูหรือไปยังหน้าถัดไปได้

2. **ฟังก์ชัน Undo/Redo ในแอปพลิเคชัน**

   - Deque สามารถใช้จัดการรายการการเปลี่ยนแปลงในแอปพลิเคชัน เช่น การเลิกทำ (undo) และการทำซ้ำ (redo)

3. **อัลกอริทึมจัดการงาน (Job Scheduling)**

   - ใช้สำหรับจัดลำดับการทำงานในระบบที่มีการจัดการกระบวนการที่ซับซ้อน

การใช้งานในวงล้อการหมุน (Clockwise/Anticlockwise Rotations)

- Deque รองรับการหมุน (rotation) ทั้งแบบตามเข็มนาฬิกา (clockwise) และทวนเข็มนาฬิกา (anticlockwise) ในเวลา O(1) ซึ่งมีประโยชน์ในบางแอปพลิเคชัน เช่น การจัดการแถวงานแบบวงกลม

Priority Queue ซึ่งเป็นโครงสร้างข้อมูลแบบ Queue ที่จัดลำดับการเข้าถึงข้อมูลตามค่าความสำคัญ (Priority) ของข้อมูล ไม่ใช่ตามลำดับที่ข้อมูลถูกเพิ่มเข้ามา

คำจำกัดความของ Priority Queue

- Priority Queue เป็น Queue ที่จัดลำดับองค์ประกอบตาม **ค่าความสำคัญ** (Priority Value) แทนที่จะเป็นลำดับเวลา (FIFO).
- ข้อมูลที่มีค่าความสำคัญสูงกว่าจะถูกนำออกก่อนข้อมูลที่มีค่าความสำคัญต่ำกว่า.
- ความซับซ้อนของการดำเนินการใน Priority Queue คือ **O(log n)**.

ประเภทของ Priority Queue

1. **Ascending Priority Queue (คิวที่เรียงลำดับเพิ่มขึ้น):**

   - สามารถเพิ่มข้อมูลได้ตามลำดับที่ต้องการ (arbitrarily) แต่ **จะลบได้เฉพาะข้อมูลที่มีค่าต่ำสุด**.
   - **ตัวอย่าง**: หากมีข้อมูล [4, 2, 8] ถูกเพิ่มเข้าไป:
     - ข้อมูลจะถูกลบออกในลำดับ [2, 4, 8] (ค่าที่เล็กที่สุดออกก่อน).

   - เหมาะสำหรับการจัดการข้อมูลที่ต้องลำดับการเข้าถึงจากค่าที่เล็กที่สุด.

2. **Descending Priority Queue (คิวที่เรียงลำดับลดลง)**

   - สามารถเพิ่มข้อมูลได้ตามลำดับที่ต้องการ (arbitrarily) แต่ **จะลบได้เฉพาะข้อมูลที่มีค่าสูงสุด**.
   - **ตัวอย่าง**: หากมีข้อมูล [4, 2, 8] ถูกเพิ่มเข้าไป:
     - ข้อมูลจะถูกลบออกในลำดับ [8, 4, 2] (ค่าที่ใหญ่ที่สุดออกก่อน).

   - เหมาะสำหรับการจัดการข้อมูลที่ต้องลำดับการเข้าถึงจากค่าที่ใหญ่ที่สุด.

ลักษณะการทำงาน

- **เพิ่มข้อมูล (Insertion):** การเพิ่มข้อมูลสามารถทำได้อย่างอิสระ โดยไม่ต้องเรียงลำดับขณะเพิ่ม.
- **ลบข้อมูล (Deletion):** ข้อมูลจะถูกจัดลำดับขณะลบ โดยเรียงตามค่าความสำคัญ (สูงสุดหรือต่ำสุด ขึ้นอยู่กับประเภทของ Priority Queue).

การใช้งาน Priority Queue

- **งานวางแผนกระบวนการ (Process Scheduling):** ใช้ในระบบปฏิบัติการเพื่อจัดลำดับกระบวนการตามค่าความสำคัญ.

- **งานคำนวณเส้นทางสั้นสุด (Shortest Path):** ใช้ในอัลกอริทึม Dijkstra สำหรับการค้นหาเส้นทางที่มีค่าต้นทุนต่ำสุด.
- **ระบบการจองตั๋ว:** ใช้เพื่อจัดลำดับผู้ใช้ตามความสำคัญ เช่น สมาชิกพิเศษได้รับการบริการก่อน.

การดำเนินการพื้นฐานของ Queue (Basic Operations) พร้อมตัวอย่างโค้ดสำหรับการใช้งานใน Python โดยใช้การสร้างคลาส Queue แบบกำหนดเอง

การดำเนินการพื้นฐานของ Queue

1. **enqueue()** ใช้สำหรับ **เพิ่มข้อมูล** เข้าไปใน queue ที่ตำแหน่งท้าย (rear).
2. **dequeue()** ใช้สำหรับ **ลบข้อมูล** ออกจาก queue ที่ตำแหน่งหน้า (front).
3. **peek() หรือ front()** ใช้สำหรับดูข้อมูลที่ตำแหน่งหน้าสุด (front) โดย **ไม่ลบข้อมูล** ออก.
4. **rear()** ใช้สำหรับดูข้อมูลที่ตำแหน่งท้ายสุด (rear) โดย **ไม่ลบข้อมูล** ออก.
5. **isFull()** ตรวจสอบว่า queue เต็มหรือไม่.
6. **isNull() หรือ isEmpty()** ตรวจสอบว่า queue ว่างเปล่าหรือไม่.

Queue: Basic Operations

```python
# Python3 program for array implementation of queue
# Class Queue to represent a queue
class Queue:
    # __init__ function
    def __init__(self, capacity):
        self.front = self.size = 0
        self.rear = capacity - 1
        self.Q = [None]*capacity
        self.capacity = capacity

    # Queue is full when size becomes
    # equal to the capacity
    def isFull(self):
        return self.size == self.capacity

    # Queue is empty when size is 0
    def isEmpty(self):
        return self.size == 0

    # Function to add an item to the queue.
    # It changes rear and size
    def EnQueue(self, item):
        if self.isFull():
            print("Full")
            return
        self.rear = (self.rear + 1) % (self.capacity)
        self.Q[self.rear] = item
        self.size = self.size + 1
        print("% s enqueued to queue" % str(item))
```

```python
    # Function to remove an item from queue.
    # It changes front and size
    def DeQueue(self):
        if self.isEmpty():
            print("Empty")
            return

        print("% s dequeued from queue" % str(self.Q[self.front]))
        self.front = (self.front + 1) % (self.capacity)
        self.size = self.size - 1

    # Function to get front of queue
    def que_front(self):
        if self.isEmpty():
            print("Queue is empty")

        print("Front item is", self.Q[self.front])

    # Function to get rear of queue
    def que_rear(self):
        if self.isEmpty():
            print("Queue is empty")
        print("Rear item is", self.Q[self.rear])

# Driver Code
if __name__ == '__main__':
    queue = Queue(30)
    queue.EnQueue(10)
    queue.EnQueue(20)
    queue.EnQueue(30)
    queue.EnQueue(40)
    queue.DeQueue()
    queue.que_front()
    queue.que_rear()
```

ตัวอย่างโค้ดสำหรับการใช้งาน Queue ใน Python

1. **การกำหนดคลาส Queue**

    - คลาส Queue ถูกสร้างขึ้นเพื่อแสดงการทำงานของ Queue

- ตัวแปร front, rear, size, และ capacity ใช้จัดการตำแหน่งและขนาดของ queue.
- ใช้ลิสต์ self.Q เพื่อเก็บข้อมูลใน queue.

2. **การทำงานของฟังก์ชันต่างๆ**

- **isFull()** ตรวจสอบว่า queue เต็มหรือไม่ โดยเปรียบเทียบ size กับ capacity.
- **isEmpty()** ตรวจสอบว่า queue ว่างหรือไม่ โดยตรวจสอบว่า size เท่ากับ 0 หรือไม่.
- **Enqueue(item)** เพิ่มข้อมูลใหม่ที่ตำแหน่งท้าย (rear) ของ queue และปรับค่า rear และ size.
- **Dequeue()** ลบข้อมูลที่ตำแหน่งหน้าสุด (front) ของ queue และปรับค่า front และ size.
- **get_front()** คืนค่าข้อมูลที่ตำแหน่ง front โดยไม่ลบข้อมูล.
- **get_rear()** คืนค่าข้อมูลที่ตำแหน่ง rear โดยไม่ลบข้อมูล.

3. **โค้ดตัวอย่างการใช้งาน**

- สร้าง object ของคลาส Queue (queue = Queue(3)).
- ใช้ฟังก์ชัน Enqueue() เพิ่มข้อมูล เช่น 10, 20, 30.
- ใช้ฟังก์ชัน Dequeue() เพื่อลบข้อมูลออก.
- แสดงข้อมูลที่ตำแหน่ง front และ rear ด้วย get_front() และ get_rear().

ประโยชน์ของตัวอย่างนี้

- แสดงให้เห็นการทำงานของ Queue อย่างชัดเจนตั้งแต่การเพิ่มและลบข้อมูลไปจนถึงการตรวจสอบสถานะของ Queue.
- การสร้างคลาส Queue แบบกำหนดเองช่วยให้เข้าใจโครงสร้างข้อมูล Queue และการจัดการข้อมูลในหน่วยความจำได้อย่างชัดเจน.