สารบัญ

[บทที่ 1 Array และ Linked List 3](#_Toc208500173)

[1.1 ความหมายของ Array และ Linked List 3](#_Toc208500174)

[1.2 โครงสร้างและองค์ประกอบ 4](#_Toc208500175)

[1.3 การดำเนินการ (Operations) 6](#_Toc208500176)

[1.4 Time/Space Complexity (ตาราง Array vs Linked List) 8](#_Toc208500177)

[1.5 การใช้งานจริง 8](#_Toc208500178)

[1.6 ตัวอย่างโค้ด (Python) 9](#_Toc208500179)

[บทที่ 2 สแตก 15](#_Toc208500180)

[2.1 ความหมายของโครงสร้างข้อมูลแบบสแตก 15](#_Toc208500181)

[2.2 โครงสร้างและองค์ประกอบของสแตก 15](#_Toc208500182)

[2.3 การดำเนินการของสแตก 16](#_Toc208500183)

[2.4 ความซับซ้อนของเวลา (Time Complexity ของสแตก) 25](#_Toc208500184)

[2.5 การประยุกต์ใช้งานจริงของสแตก (Applications of Stack) 26](#_Toc208500185)

[2.6 ตัวอย่างโค้ด 28](#_Toc208500186)

[บทที่ 3 คิว 31](#_Toc208500187)

[3.1 ความหมาย (FIFO) 31](#_Toc208500188)

[3.2 โครงสร้างและองค์ประกอบ 31](#_Toc208500189)

[3.3 การดำเนินการ (Operations) 32](#_Toc208500190)

[3.4 ประเภทของ Queue 33](#_Toc208500191)

[3.5 Time Complexity 34](#_Toc208500192)

[3.6 การใช้งานจริง 34](#_Toc208500193)

[3.7 ตัวอย่างโค้ด 36](#_Toc208500194)

[บทที่ 4 ต้นไม้ 43](#_Toc208500195)

[4.1 ความหมายของโครงสร้างข้อมูลแบบต้นไม้ 43](#_Toc208500196)

[4.2 โครงสร้างและองค์ประกอบของต้นไม้ 43](#_Toc208500197)

[4.3 การดำเนินการของต้นไม้ 46](#_Toc208500198)

[4.4 Time Complexity 51](#_Toc208500199)

[4.5 การใช้งานจริงของต้นไม้ 51](#_Toc208500200)

[4.6 ตัวอย่างโค้ด 52](#_Toc208500201)

[บทที่ 5 กราฟ 62](#_Toc208500202)

[5.1 5.1 ความหมาย 62](#_Toc208500203)

[5.2 โครงสร้างและศัพท์สำคัญ 62](#_Toc208500204)

[5.3 การแทนกราฟ (Graph Representation) 63](#_Toc208500205)

[5.4 การดำเนินการ (Graph Operations) 64](#_Toc208500206)

[5.5 อัลกอริทึมบนกราฟ (Graph Algorithms) 65](#_Toc208500207)

[5.6 Time Complexity (ประสิทธิภาพเชิงเวลา) 68](#_Toc208500208)

[5.7 การใช้งานจริง (Applications) 71](#_Toc208500209)

[5.8 ตัวอย่างโค้ด (Code Examples) 73](#_Toc208500210)

# Array และ Linked List

## ความหมายของ Array และ Linked List

อาร์เรย์และลิงค์ลิสต์เป็นโครงสร้างข้อมูลเชิงเส้นที่มีการจัดเรียงข้อมูลแบบต่อเนื่องและเข้าถึง

ข้อมูลแบบลําดับ ซึ่งจัดเป็นโครงสร้างข้อมูลพื้นฐานสําหรับโครงสร้างข้อมูลแบบอื่น ๆ โดยโครงสร้าง ข้อมูลแบบอาร์เรย์และลิงค์ลิสต์นําไปใช้ในโครงสร้างข้อมูล เช่น คิว (Queues) ไบนารีทรี (BinaryTree) สแตก (Stack)

### Array คืออะไร

อาร์เรย์ (Array) คือโครงสร้างข้อมูลประเภทหนึ่งที่สามารถเก็บชุดข้อมูลชนิดเดียวกันหลายค่าภายใต้ชื่อตัวแปรเดียว โดยข้อมูลจะถูกจัดเก็บเป็นลำดับและสามารถเข้าถึงแต่ละข้อมูลได้ด้วยเลขดัชนี (Index) ซึ่งโดยทั่วไปจะเริ่มต้นที่ 0. อาร์เรย์มีการจองหน่วยความจำแบบต่อเนื่อง (contiguous memory) และมีขนาดที่แน่นอนซึ่งต้องกำหนดล่วงหน้า

### Linked List คืออะไร

ลิงค์ลิสต์ (Linked List) เป็นโครงสร้างข้อมูลเชิงเส้นที่เก็บข้อมูลในรูปแบบของโหนด (Node) โดยแต่ละโหนดจะประกอบด้วยสองส่วนหลักคือ ส่วนข้อมูล (Data Field) และส่วนตัวชี้ (Link Field หรือ Pointer/Reference) ซึ่งชี้ไปยังโหนดถัดไปในลำดับ. ลิงค์ลิสต์ไม่จำเป็นต้องจองหน่วยความจำแบบต่อเนื่องเหมือนอาร์เรย์ แต่จะจองหน่วยความจำให้กับแต่ละโหนดแบบไดนามิกเมื่อมีการเพิ่มข้อมูลเข้ามา และคืนพื้นที่เมื่อมีการลบข้อมูลออกไป

### ความแตกต่าง Array vs Linked List

ความแตกต่างหลักระหว่าง Array และ Linked List อยู่ที่โครงสร้างพื้นฐาน การจัดเก็บหน่วยความจำ และความยืดหยุ่นในการจัดการข้อมูล

#### **การจัดเก็บหน่วยความจำ** **Array**

#### เก็บข้อมูลในหน่วยความจำที่อยู่ติดกัน (contiguous memory) ในขณะที่ Linked List เก็บข้อมูลแบบกระจายในหน่วยความจำ โดยเชื่อมโยงกันด้วยพอยน์เตอร์

#### **ขนาด Array**

#### มีขนาดคงที่ที่ต้องกำหนดล่วงหน้า ทำให้ยากต่อการปรับเปลี่ยนขนาดเมื่อโปรแกรมทำงาน. ส่วน Linked List มีขนาดที่ยืดหยุ่น สามารถเพิ่มหรือลบโหนดได้ตามต้องการระหว่างการทำงาน

#### **การเข้าถึงข้อมูล**

#### การเข้าถึงข้อมูลใน Array ทำได้รวดเร็วโดยตรงผ่านเลขดัชนี (Random Access). สำหรับ Linked List การเข้าถึงข้อมูลต้องทำแบบเชิงเส้น (Sequential Access) โดยเริ่มจากโหนดแรก (head) แล้วไล่ไปตามพอยน์เตอร์จนกว่าจะเจอข้อมูลที่ต้องการ.

#### **การแทรกและลบข้อมูล**

#### การแทรกหรือลบข้อมูลตรงกลาง Array อาจไม่มีประสิทธิภาพ เนื่องจากต้องมีการเลื่อนตำแหน่งข้อมูลอื่นๆ. ในทางกลับกัน Linked List สามารถเพิ่มหรือลบข้อมูลได้สะดวกและยืดหยุ่นกว่ามาก เพียงแค่ปรับเปลี่ยนพอยน์เตอร์

#### **การใช้งานหน่วยความจำ**

#### Linked List ใช้พื้นที่หน่วยความจำมากกว่า Array เล็กน้อย เนื่องจากแต่ละโหนดต้องเก็บพอยน์เตอร์เพิ่มเติม

## โครงสร้างและองค์ประกอบ

### Array → index-based, contiguous memory

อาร์เรย์จัดเก็บข้อมูลตามดัชนี (index-based) โดยแต่ละตำแหน่งในอาร์เรย์จะมีเลขดัชนีที่ไม่ซ้ำกันเพื่อใช้ในการอ้างอิงถึงข้อมูลนั้นๆ. หน่วยความจำที่ใช้ในการจัดเก็บข้อมูลของอาร์เรย์จะถูกจัดสรรแบบต่อเนื่องกัน (contiguous memory) ซึ่งหมายความว่าข้อมูลทั้งหมดจะอยู่ติดกันในหน่วยความจำ ทำให้การเข้าถึงข้อมูลทำได้รวดเร็ว

### Linked List → node (data + pointer/next)

โครงสร้างข้อมูลแบบ Linked List ประกอบด้วยหน่วยย่อยที่เรียกว่า **โหนด (Node)** แต่ละโหนดโดยพื้นฐานแล้วจะประกอบด้วย 2 ส่วนหลักๆ คือ

#### **ฟิลด์ข้อมูล (Data Field)**

#### พื้นที่หรือช่องที่กำหนดไว้สำหรับเก็บ ข้อมูลหนึ่งชนิด/คุณสมบัติหนึ่งอย่าง ของวัตถุหรือสิ่งที่กำลังบันทึก

#### **ฟิลด์ตัวชี้ (Link Field หรือ Pointer/Next)**

ฟิลด์พิเศษในโครงสร้างข้อมูลแบบเชื่อมโยง (Linked Structure) ที่ทำหน้าที่เก็บ ตำแหน่งหรือที่อยู่ (Address/Reference) ของโหนดถัดไป (หรือก่อนหน้า) แทนที่จะเก็บค่าข้อมูลโดยตรง

#### โหนดสุดท้ายใน Linked List ทั่วไปจะชี้ไปยังค่าว่าง (NULL หรือ nullptr) เพื่อแสดงจุดสิ้นสุดของลิสต์

### ประเภท Linked List

Linked List มีหลายประเภท ขึ้นอยู่กับจำนวนพอยน์เตอร์ในแต่ละโหนดและวิธีการเชื่อมโยง

#### **Singly Linked List**

#### เป็นลิงค์ลิสต์ที่ง่ายที่สุด โดยแต่ละโหนดมีฟิลด์ตัวชี้เพียง 1 ฟิลด์สำหรับชี้ตำแหน่งของโหนดถัดไปเท่านั้น. ทำให้การเดินทางในลิสต์เป็นแบบทิศทางเดียว คือจากโหนดแรกไปยังโหนดสุดท้ายเท่านั้น ไม่สามารถย้อนกลับได้. โหนดสุดท้ายจะชี้ไปที่ NULL

#### **Doubly Linked List**

#### เป็นลิงค์ลิสต์ที่ได้รับการปรับปรุงจาก Singly Linked List โดยแต่ละโหนดประกอบด้วย 3 ฟิลด์ คือ ฟิลด์ข้อมูล และฟิลด์ตัวชี้จำนวน 2 ฟิลด์

- ตัวชี้ไปยังโหนดถัดไป (next pointer)

- ตัวชี้ไปยังโหนดก่อนหน้า (previous pointer หรือ prev pointer)  
การมีตัวชี้สองทิศทางทำให้สามารถเดินทางในลิสต์ได้ทั้งไปข้างหน้าและย้อนกลับ. โหนดแรกและโหนดสุดท้ายมักจะมีพอยน์เตอร์ prev และ next ชี้ไปที่ NULL ตามลำดับ

#### **Circular Linked List**

#### เป็น Linked List ชนิดพิเศษที่โหนดสุดท้ายมีตัวชี้กลับไปยังโหนดแรก ทำให้โครงสร้างของ Linked List เป็นวงกลมและไม่มีจุดสิ้นสุดที่ชัดเจน (ไม่มีโหนดใดชี้ไปที่ NULL) การวนลูปในลิสต์ประเภทนี้สามารถทำได้อย่างต่อเนื่อง Circular Linked List สามารถเป็นได้ทั้งแบบ Singly Circular Linked List (แต่ละโหนดมีตัวชี้เดียวไปยังโหนดถัดไป และโหนดสุดท้ายชี้กลับไปที่โหนดแรก) และ Doubly Circular Linked List (แต่ละโหนดมีตัวชี้สองตัวทั้งก่อนหน้าและถัดไป และมีการเชื่อมโยงเป็นวงกลม)

## การดำเนินการ (Operations)

### Array

การสร้างอาร์เรย์คือการประกาศตัวแปรอาร์เรย์และกำหนดขนาดหรือค่าเริ่มต้นให้กับมัน. ตัวอย่างเช่น ใน Java สามารถประกาศอาร์เรย์ของจำนวนเต็มขนาด 4 ได้ด้วย

 int[] number = new int[4];

การเข้าถึงข้อมูลในอาร์เรย์ทำได้โดยใช้เลขดัชนีของตำแหน่งที่ต้องการ โดยดัชนีเริ่มต้นที่ 0 ตัวอย่างเช่น number[0] จะเข้าถึงองค์ประกอบแรกของอาร์เรย์ชื่อ number

การแทรกข้อมูล ณ ตำแหน่งที่ระบุในอาร์เรย์ (ยกเว้นท้ายสุด) มักจะต้องเลื่อนข้อมูลที่อยู่ถัดจากตำแหน่งนั้นๆ ไปข้างหลังเพื่อสร้างพื้นที่ว่าง

การลบข้อมูล ณ ตำแหน่งที่ระบุในอาร์เรย์ มักจะต้องเลื่อนข้อมูลที่อยู่ถัดจากตำแหน่งนั้นๆ มาข้างหน้าเพื่อปิดช่องว่าง

การค้นหาข้อมูลในอาร์เรย์สามารถทำได้โดยการวนลูปตรวจสอบทีละองค์ประกอบ หรือหากอาร์เรย์ถูกจัดเรียงไว้แล้ว สามารถใช้อัลกอริทึมการค้นหาที่มีประสิทธิภาพ เช่น Binary Search

### Linked List

การสร้างโหนดใน Linked List คือการจองหน่วยความจำสำหรับโหนดใหม่และกำหนดค่าข้อมูลและพอยน์เตอร์เริ่มต้น. การสร้าง Linked List เริ่มต้นด้วยการสร้างโหนดแรก (head node)

#### Insert at head (แทรกที่หัวลิสต์)

#### สร้างโหนดใหม่ กำหนดให้พอยน์เตอร์ next ของโหนดใหม่ชี้ไปยังโหนด head ปัจจุบัน แล้วเปลี่ยน head ให้ชี้ไปยังโหนดใหม่

#### **Insert at tail (แทรกที่ท้ายลิสต์)**

#### สร้างโหนดใหม่ กำหนดให้พอยน์เตอร์ next ของโหนดสุดท้ายในลิสต์ปัจจุบันชี้ไปยังโหนดใหม่ แล้วเปลี่ยนให้โหนดใหม่เป็นโหนดสุดท้าย

#### **Insert at position (แทรกที่ตำแหน่งที่ระบุ)**

#### ค้นหาโหนดก่อนหน้าตำแหน่งที่ต้องการแทรก สร้างโหนดใหม่ กำหนดให้พอยน์เตอร์ next ของโหนดใหม่ชี้ไปยังโหนดถัดไปของโหนดก่อนหน้า แล้วเปลี่ยนพอยน์เตอร์ next ของโหนดก่อนหน้าให้ชี้ไปยังโหนดใหม่

#### **Delete head (ลบโหนดหัว)**

#### การลบโหนด สามารถลบทั้งโหนดและข้อมูลที่ไม่ต้องการออกไป ซึ่งมีหลักการง่าย ๆด้วยการเปลี่ยนลิ้งค์ของโหนดก่อนหน้าที่ต้องการลบให้ชี้ไปยังลิ้งค์ของโหนดที่ต้องการลบ

#### **Delete tail (ลบโหนดท้าย)**

#### วนลูปหาโหนดก่อนหน้าโหนดสุดท้าย ปรับเปลี่ยนพอยน์เตอร์ next ของโหนดก่อนหน้าให้เป็น NULL แล้วคืนหน่วยความจำของโหนดสุดท้าย

#### **Delete at position (ลบข้อมูลที่ตำแหน่งที่ระบุ)**

#### ค้นหาโหนดก่อนหน้าตำแหน่งที่ต้องการลบ แล้วปรับเปลี่ยนพอยน์เตอร์ next ของโหนดก่อนหน้าให้ชี้ข้ามโหนดที่จะลบไป

#### **การค้นหาข้อมูลใน Linked List**

#### ทำได้โดยการเริ่มต้นจากโหนด head แล้วไล่ตามพอยน์เตอร์ next ไปทีละโหนด เปรียบเทียบข้อมูลในแต่ละโหนดจนกว่าจะเจอข้อมูลที่ต้องการ

#### **การ Traverse**

#### คือการเดินผ่านทุกโหนดใน Linked List ตั้งแต่โหนดแรกจนถึงโหนดสุดท้าย โดยทำตามพอยน์เตอร์ next ไปทีละขั้น

## Time/Space Complexity (ตาราง Array vs Linked List)

|  |  |  |  |
| --- | --- | --- | --- |
| การดำเนินการ | Array (เวลา) | Linked List (เวลา) | หมายเหตุ |
| การเข้าถึง (Access) | O(1) (เข้าถึงโดยตรง) | O(n) (ต้องไล่ค้นหา) | Array เร็วกว่าเพราะเข้าถึงด้วยดัชนีโดยตรง |
| การค้นหา (Search) | O(n) (เชิงเส้น) | O(n) (เชิงเส้น) | ในกรณีที่ข้อมูลไม่เรียง |
| การแทรก (Insertion) | O(n) (ต้องเลื่อนข้อมูล) | O(1) (ที่หัว/ท้าย), O(n) (ที่ตำแหน่ง) | Linked List ยืดหยุ่นกว่าเมื่อแทรกตรงกลาง |
| การลบ (Deletion) | O(n) (ต้องเลื่อนข้อมูล) | O(1) (ที่หัว/ท้าย), O(n) (ที่ตำแหน่ง) | Linked List ยืดหยุ่นกว่าเมื่อลบตรงกลาง |
| การใช้หน่วยความจำ (Space) | O(n) (สำหรับข้อมูล) | O(n) (สำหรับข้อมูล + พอยน์เตอร์) | Linked List ใช้หน่วยความจำเพิ่มสำหรับพอยน์เตอร์ |

## การใช้งานจริง

### Array

**- การจัดเก็บข้อมูลที่มีขนาดคงที่**

เหมาะสำหรับเก็บข้อมูลที่มีจำนวนแน่นอน เช่น วันในสัปดาห์, เดือนในปี

**- การจัดการข้อมูลในด้านการศึกษา**

ใช้ในการจัดเก็บคะแนนสอบของนักเรียน เช่น คะแนนสอบของนักเรียน 50 คนรู้จำนวนชัดเจนใช้ score[0] ... score[49] เข้าถึงได้รวดเร็ว (O(1))

**- ตาราง Lookup Table**

 สำหรับการเข้าถึงข้อมูลอย่างรวดเร็วโดยใช้ดัชนี เช่น

ตารางเดือน months = ["Jan", "Feb", ..., "Dec"]

**- Multimedia Processing** การจัดเก็บพิกเซลของรูปภาพ หรือ ตัวอย่างเสียง

เช่น ภาพขนาด 1920×1080 ใช้ Array 2 มิติ pixel[1920][1080]

### Linked List

**- ระบบจัดการหน่วยความจำ** เนื่องจากสามารถเพิ่มและลบโหนดได้แบบไดนามิก ทำให้เหมาะสำหรับระบบที่ต้องการการจัดสรรหน่วยความจำแบบยืดหยุ่น

**- การจัดการคิวและสแตก** เป็นโครงสร้างพื้นฐานในการสร้างคิว (Queue) และสแตก (Stack)

**- การจัดการเพลงเพลย์ลิสต์ หรือประวัติการเรียกดู (Browser History)** Circular Linked List เหมาะสำหรับการวนซ้ำ เช่น เพลย์ลิสต์เพลง หรือการจัดการ Round Robin Scheduling. Doubly Linked List ใช้ในฟังก์ชัน Undo/Redo หรือประวัติการเรียกดูที่ต้องเดินหน้าถอยหลังได้

**- การแสดงพหุนาม** แต่ละเทอมของพหุนามสามารถเป็นโหนดใน Linked List

### สรุปเปรียบเทียบ

|  |  |  |
| --- | --- | --- |
| **โครงสร้าง** | **เหมาะกับ** | **ตัวอย่างการใช้งานจริง** |
| Array | ข้อมูลมีจำนวนแน่นอน, เน้นการเข้าถึงด้วย Index | คะแนนสอบ, พิกเซลภาพ, ตารางเดือน, ตำแหน่งในเกม |
| Linked List | ข้อมูลเปลี่ยนแปลงบ่อย, ต้องเพิ่ม/ลบง่าย | Playlist เพลง, Undo/Redo, Print Queue, Memory Management |

## ตัวอย่างโค้ด (Python)

### ตัวอย่างโค้ด Array (Python)

# สร้าง Array (Initialization)

my\_array = [10, 20, 30, 40, 50]

print(f"Array: {my\_array}") # Array: [10, 20, 30, 40, 50]

# เข้าถึงข้อมูล (Access by index)

print(f"Element at index 2: {my\_array[2]}") # Element at index 2: 30

# แทรกข้อมูล (Insert at index)

my\_array.insert(2, 25) # แทรก 25 ที่ index 2

print(f"Array after insertion: {my\_array}") # Array after insertion: [10, 20, 25, 30, 40, 50]

# ลบข้อมูล (Delete at index)

del my\_array[3] # ลบข้อมูลที่ index 3

print(f"Array after deletion: {my\_array}") # Array after deletion: [10, 20, 25, 40, 50]

# ค้นหา (Search)

try:

index = my\_array.index(40)

print(f"40 found at index: {index}") # 40 found at index: 3

except ValueError:

print("40 not found in array")

### ตัวอย่างโค้ด Linked List (Python - Singly Linked List)

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

# แทรกข้อมูลที่หัว (Insert at head)

def insert\_at\_head(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

# แทรกข้อมูลที่ท้าย (Insert at tail)

def insert\_at\_tail(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

# แทรกข้อมูลที่ตำแหน่งที่ระบุ (Insert at position - ตัวอย่างแทรกหลัง node ที่มีค่าที่กำหนด)

def insert\_after\_node(self, prev\_data, new\_data):

current = self.head

while current and current.data != prev\_data:

current = current.next

if current:

new\_node = Node(new\_data)

new\_node.next = current.next

current.next = new\_node

else:

print(f"Node with data {prev\_data} not found.")

# ลบข้อมูลที่หัว (Delete head)

def delete\_head(self):

if self.head:

self.head = self.head.next

else:

print("List is empty, cannot delete head.")

# ลบข้อมูลที่ท้าย (Delete tail)

def delete\_tail(self):

if not self.head:

print("List is empty, cannot delete tail.")

return

if not self.head.next: # ถ้ามี node เดียว

self.head = None

return

current = self.head

while current.next.next:

current = current.next

current.next = None

# ลบข้อมูลที่ตำแหน่งที่ระบุ (Delete at position - ตัวอย่างลบ node ที่มีค่าที่กำหนด)

def delete\_node(self, key):

current = self.head

if current and current.data == key:

self.head = current.next

current = None

return

prev = None

while current and current.data != key:

prev = current

current = current.next

if not current:

print(f"Node with data {key} not found.")

return

prev.next = current.next

current = None

# ค้นหา (Search)

def search(self, key):

current = self.head

while current:

if current.data == key:

return True

current = current.next

return False

# เดินดูทุก node (Traverse)

def traverse(self):

current = self.head

elements = []

while current:

elements.append(current.data)

current = current.next

print(" -> ".join(map(str, elements)))

# การใช้งาน

my\_linked\_list = LinkedList()

my\_linked\_list.insert\_at\_head(5)

my\_linked\_list.insert\_at\_head(10)

my\_linked\_list.insert\_at\_tail(15)

my\_linked\_list.insert\_after\_node(5, 7) # แทรก 7 หลัง 5

print("Linked List after insertions:")

my\_linked\_list.traverse() # Output: 10 -> 5 -> 7 -> 15

print(f"Search for 7: {my\_linked\_list.search(7)}") # Search for 7: True

print(f"Search for 20: {my\_linked\_list.search(20)}") # Search for 20: False

my\_linked\_list.delete\_head()

print("Linked List after deleting head:")

my\_linked\_list.traverse() # Output: 5 -> 7 -> 15

my\_linked\_list.delete\_tail()

print("Linked List after deleting tail:")

my\_linked\_list.traverse() # Output: 5 -> 7

my\_linked\_list.delete\_node(5)

print("Linked List after deleting node 5:")

my\_linked\_list.traverse() # Output: 7

# สแตก

## ความหมายของโครงสร้างข้อมูลแบบสแตก

โครงสร้างข้อมูลแบบสแตก (Stack Structure) เป็นโครงสร้างข้อมูลที่มีการจัดเก็บข้อมูลแบบเรียงลำดับต่อเนื่องกัน การนำข้อมูลเข้าสู่สแตกหรือการนำข้อมูลออกจากสแตก จะกระทำที่ปลายด้านบนสุดของสแตกที่เรียกว่า Top เพียงจุดเดียวเท่านั้น การที่สแตกมีทางเข้าและออกของข้อมูลเพียงทางเดียว จึงทำให้โครงสร้างข้อมูลแบบสแตกเป็นโครงสร้างที่มีลักษณะการทำงานแบบ“เข้าทีหลังออกก่อน” (Last In, First Out) หรือเรียกสั้นๆ ว่า LIFO โครงสร้างข้อมูลแบบสแตกมีประโยชน์มากในการเขียนโปรแกรมคอมพิวเตอร์ ซึ่งใช้ในการจดจำลำดับการเรียกฟังก์ชั่นย่อยในโปรแกรมหลัก หรือโปรแกรมแบบรีเคอร์ซีฟ (recursive)

## โครงสร้างและองค์ประกอบของสแตก

สแตก (Stack) เป็น โครงสร้างข้อมูลเชิงเส้น (Linear Data Structure) ที่มีหลักการจัดเก็บและเข้าถึงข้อมูลแบบ เข้าทีหลัง–ออกก่อน (Last In, First Out: LIFO) กล่าวคือ ข้อมูลที่ถูกเก็บเข้ามาล่าสุดจะถูกนำออกก่อนเสมอ

### ตัวอย่างการทำงาน

นำข้อมูลเข้าเรียงลำดับ: A → B → C → D

เมื่อนำข้อมูลออก จะได้: D → C → B → A

ดังนั้น สแตกจึงเปรียบได้กับการวางของซ้อนกันในกล่อง ของที่วางบนสุดจะถูกหยิบออกก่อน

### ลักษณะสำคัญของโครงสร้างสแตก

#### **โครงสร้างเชิงเส้น (Linear Structure)**

ลักษณะโครงสร้างจะจัดเรียงต่อเนื่องกันไป ซึ่งถือเป็นโครงสร้าง ในรูปแบบของอาร์เรย์และเรคอร์ด

#### **โครงสร้างที่ปรับเปลี่ยนได้ (Dynamic Structure)**

สามารถปรับเปลี่ยนจำนวนสมาชิกในโครงสร้างได้ ขณะที่มีการทำงานอยู่ แต่ถ้าเป็นการใช้โครงสร้างของอาร์เรย์ จำเป็นต้องมีการกำหนดพื้นที่การจองบนหน่วยความจำที่เหมาะสม

## การดำเนินการของสแตก

### การสร้างสแตก (Initialization)

 คือ การเตรียมโครงสร้างข้อมูลสแตกให้อยู่ในสภาพ “พร้อมใช้งาน” ก่อนที่จะทำการเพิ่ม (Push) หรือลบข้อมูล (Pop) ได้

หลักการสร้างสแตก การสร้างสแตกจะแบ่งเป็น 2 วิธี ขึ้นกับโครงสร้างที่ใช้แทน (Implementation) ดังนี้

#### **แบบอาร์เรย์ (Array-based Stack)**

คือ การสร้างสแตกโดยใช้อาร์เรย์เป็นโครงสร้างพื้นฐานในการเก็บข้อมูล โดยข้อมูลจะถูกเก็บเรียงต่อกันในหน่วยความจำที่จัดสรรไว้ล่วงหน้า และการเข้าถึง/แก้ไขข้อมูลจะเกิดขึ้นที่ตำแหน่งบนสุด (Top) ของอาร์เรย์เท่านั้น

##### ลักษณะสำคัญ

**1.ใช้อาร์เรย์ (Array) เป็นที่เก็บข้อมูล**

###### ข้อมูลแต่ละตำแหน่งในอาร์เรย์เปรียบเสมือน “ชั้น” ของสแตก

###### ตำแหน่งแรกของอาร์เรย์คือ index 0 ขึ้นไปจนถึง index สูงสุด capacity - 1

**2.ต้องกำหนดขนาดสูงสุด (Capacity) ล่วงหน้า**

###### เมื่อสร้างอาร์เรย์ จะต้องระบุจำนวนช่องเก็บข้อมูล

###### ถ้าข้อมูลถูกเพิ่มเกินขนาดนี้จะเกิด Stack Overflow

**3.ใช้ตัวแปร top เก็บตำแหน่งของข้อมูลบนสุด**

###### top เป็นตัวชี้ (pointer) ว่าสมาชิกบนสุดอยู่ที่ตำแหน่งใด

###### เมื่อสแตกว่าง: top = -1

###### เมื่อ Push ข้อมูล: เพิ่มค่า top ขึ้น 1 แล้วบันทึกค่าที่ตำแหน่งนั้น

###### เมื่อ Pop ข้อมูล: คืนค่าที่ A[top] แล้วลดค่า top ลง 1

**4.การเริ่มต้น (Initialization)**

###### ตั้งค่า top = -1 เพื่อแสดงว่าสแตกยังว่าง

###### อาร์เรย์ถูกสร้างขึ้นมาแล้ว แต่ยังไม่มีข้อมูลที่ถูก Push เข้าไป

##### ตัวอย่างการทำงาน

สมมติสร้างสแตกด้วยขนาด capacity = 5

###### **เริ่มต้น**

###### A = [\_, \_, \_, \_, \_] // ช่องยังว่างทั้งหมด

###### top = -1

###### **Push(10)**

top = 0

A = [10, \_, \_, \_, \_]

###### **Push(20)**

top = 1

A = [10, 20, \_, \_, \_]

###### **Push(30)**

top = 2

A = [10, 20, 30, \_, \_]

###### **Pop()**

return 30

top = 1

A = [10, 20, \_, \_, \_]

##### ข้อดีของ Array-based Stack

###### เข้าถึงตำแหน่งบนสุดได้รวดเร็ว (เวลา O(1))

###### โครงสร้างชัดเจน อ่านง่าย เหมาะสำหรับผู้เริ่มต้น

###### ใช้หน่วยความจำต่อเนื่อง → ทำงานร่วมกับ Cache ได้ดี

##### ข้อเสียของ Array-based Stack

###### ขนาดตายตัว → หากข้อมูลเกินจะ Overflow

###### หากจองขนาดใหญ่เกินไป อาจเปลืองหน่วยความจำ

###### ถ้าขนาดเล็กเกินไป อาจไม่เพียงพอต่อการใช้งาน

#### **สแตกแบบลิงก์ลิสต์ (Linked List-based Stack)**

คือ การสร้างสแตกโดยใช้ โครงสร้างลิงก์ลิสต์ (Linked List) เป็นตัวเก็บข้อมูล แต่ละข้อมูลจะถูกเก็บไว้ใน โหนด (Node) และโหนดเหล่านี้จะเชื่อมโยงกันด้วยตัวชี้ (Pointer) โดย top จะชี้ไปยังโหนดบนสุดของสแตก

##### ลักษณะสำคัญ

**1.ฟิลด์หลัก top ชี้ไปยังโหนดหัว (Head) ของลิสต์**

###### ตัวแปร top ทำหน้าที่บอกว่าสมาชิกบนสุด (Top) อยู่ตรงไหน

###### ถ้า Stack ว่าง → top = NULL

**2.โหนด (Node)**

###### โหนด 1 ตัวมี 2 ส่วนหลัก

###### 1.value → เก็บค่าของข้อมูล

2. next → ตัวชี้ไปยังโหนดถัดไป (หรือ NULL ถ้าเป็นโหนดสุดท้าย)

###### ทำให้สามารถเชื่อมโยงข้อมูลหลายตัวเป็นสายโซ่ (Chain) ได้

**3.สถานะพิเศษ**

###### Stack ว่าง top = NULL

###### Push สร้างโหนดใหม่ แล้วให้ newNode.next = top จากนั้นเลื่อน top ไปที่ newNode

###### Pop เก็บค่า top.value, ย้าย top ไปที่ top.next, และคืนหน่วยความจำของโหนดเดิม

**4.การเริ่มต้น (Initialization)**

###### ตั้งค่า top = NULL

###### แสดงว่า Stack ยังไม่มีโหนดใด ๆ อยู่

##### ตัวอย่างการทำงาน

###### **เริ่มต้น**

top → NULL

###### **Push(10)**

top → [10 | NULL]

###### **Push(20)**

top → [20 | \*] → [10 | NULL]

###### **Push(30)**

top → [30 | \*] → [20 | \*] → [10 | NULL]

###### **Pop()**

return 30

top → [20 | \*] → [10 | NULL]

##### ข้อดีของ Linked List-based Stack

###### ไม่ต้องกำหนดขนาดสูงสุดล่วงหน้า → ขนาดยืดหยุ่นตามหน่วยความจำ

###### ไม่มีปัญหา Stack Overflow (ยกเว้นกรณีหน่วยความจำหมด)

###### การเพิ่ม/ลบ (Push/Pop) ทำได้ง่ายที่ Top ในเวลา O(1)

##### ข้อเสียของ Linked List-based Stack

###### มี Overhead ของ pointer ในแต่ละโหนด → ใช้หน่วยความจำมากกว่า Array

###### การจัดเก็บข้อมูลไม่ต่อเนื่อง (Non-contiguous) → locality ของหน่วยความจำไม่ดี → อาจช้ากว่า Array ในบางงาน

###### ต้องจัดการหน่วยความจำเอง (malloc/new และ free/delete) → โค้ดซับซ้อนกว่า

### การเพิ่มข้อมูลเข้าสแตก Push()

คือ การนำข้อมูลออกจากสแตก โดยข้อมูลที่จะถูกนำออก ต้องเป็นข้อมูลบนสุด (Top) เท่านั้น ตามหลักการ เข้าทีหลัง–ออกก่อน (Last In, First Out: LIFO)

#### **หลักการทำงาน**

1.ตรวจสอบว่าสแตกอยู่ในสถานะที่สามารถเพิ่มข้อมูลได้หรือไม่

ถ้าสแตกเต็มแล้ว (เต็มตามข้อจำกัดของระบบ/หน่วยความจำ) จะไม่สามารถ Push ได้ และถือว่าเกิดข้อผิดพลาด Stack Overflow

2.ถ้าสามารถเพิ่มได้ → จัดเก็บข้อมูลใหม่ที่ตำแหน่งบนสุด

3.ปรับปรุงค่าตัวชี้หรือสถานะ Top ให้บ่งบอกตำแหน่งข้อมูลล่าสุดที่เพิ่งถูกเพิ่มเข้าไป

#### **เงื่อนไขที่เกี่ยวข้อง**

##### Stack Overflow

###### เกิดขึ้นเมื่อพยายาม Push ข้อมูลใหม่ แต่สแตกไม่สามารถรองรับได้อีกแล้ว

###### เช่น กองจานที่วางซ้อนกันจนเกินความสูงที่กำหนด → วางเพิ่มไม่ได้

##### Stack ไม่เต็ม

###### สามารถ Push ได้ตามปกติ ข้อมูลใหม่จะอยู่บนสุด

#### **ตัวอย่างการทำงาน**

###### **เริ่มต้น**

###### A = [\_] // ช่องยังว่างทั้งหมด

###### **Push(10)**

A = [10]

top = 10

###### **Push(20)**

A = [10,20]

top = 20

###### **Push(30)**

A = [10,20,30]

top = 30

### การลบข้อมูลเข้าสแตก Pop()

คือ กระบวนการนำข้อมูลใหม่เข้าสู่ สแตก (Stack) โดยข้อมูลทุกตัวที่ถูกเพิ่มจะถูกวางไว้ บนสุด (Top) ของสแตกเสมอ

#### **หลักการทำงาน**

1. ตรวจสอบว่าสแตกว่างหรือไม่

ถ้าว่าง → ไม่สามารถนำข้อมูลออกได้ → เกิดข้อผิดพลาด Stack Underflow

2. ถ้าไม่ว่างจะอ่านค่าข้อมูลที่ตำแหน่งบนสุด (Top)

3. ลบข้อมูลบนสุดออกจากสแตก

4. ปรับปรุงสถานะ Top ชี้ไปยังสมาชิกตัวก่อนหน้าเพิ่มเข้าไป

#### **เงื่อนไขที่เกี่ยวข้อง**

##### Stack Underflow

###### เกิดขึ้นเมื่อพยายาม Pop ข้อมูล แต่สแตกไม่มีข้อมูลเหลืออยู่แล้ว

###### เช่น กองจานที่ว่างเปล่า จะหยิบออกไม่ได้เช่น กองจานที่วางซ้อนกันจนเกินความสูงที่กำหนด วางเพิ่มไม่ได้

##### Stack ไม่ว่าง

###### สามารถ Pop ได้ตามปกติ ข้อมูลที่ถูกลบจะเป็นข้อมูลบนสุดเสมอ

#### **ตัวอย่างการทำงาน**

###### **เริ่มต้น**

###### A = [10, 20, 30]

###### **Pop()**

คืนค่า = 30

เหลือ [10, 20]

###### **Pop()**

คืนค่า = 20

เหลือ [10]

###### **Pop()**

คืนค่า = 10

เหลือ [ ]

###### **Pop()**

Stack Underflow

### การดูข้อมูลบนสุดของสแตก (Peek()/Top())

คือ การดำเนินการตรวจสอบหรืออ่านค่าข้อมูลที่อยู่ บนสุด (Top) ของสแตก โดย ไม่ลบข้อมูลออก

#### **หลักการทำงาน**

1. ตรวจสอบว่าสแตกว่างหรือไม่

ถ้าว่าง → ไม่มีข้อมูลให้ดู → ควรรายงานว่า “สแตกว่าง”

2. ถ้าไม่ว่าง → คืนค่าของข้อมูลที่อยู่บนสุด (Top)

3. โครงสร้างของสแตกจะยังคงเหมือนเดิม ไม่เปลี่ยนแปลง

#### **เงื่อนไขที่เกี่ยวข้อง**

##### Stack ว่าง

###### การ Peek จะไม่สามารถทำได้ → ต้องแจ้งผลว่าข้อมูลว่าง (เช่น คืนค่า null หรือข้อความแสดงข้อผิดพลาด)

##### Stack ไม่ว่าง

###### คืนค่าของข้อมูลบนสุดได้ตามปกติ

#### **ตัวอย่างการทำงาน**

###### **เริ่มต้น**

[10, 20, 30] // Top = 30

###### **Peek()**

คืนค่า = 30

สแตกยังคงเป็น [10, 20, 30]

###### **Push(40)**

[10, 20, 30, 40] // Top = 40

###### **Peek()**

คืนค่า = 40

สแตกยังคงเป็น [10, 20, 30, 40]

### การตรวจสอบว่าสแตกว่างหรือไม่ (isEmpty())

คือ การดำเนินการตรวจสอบว่าสแตกมีข้อมูลอยู่หรือไม่

- ถ้า ไม่มีข้อมูลเลย สแตกอยู่ในสถานะ ว่าง (Empty)

- ถ้ามีข้อมูลอย่างน้อยหนึ่งตัว สแตกอยู่ในสถานะ ไม่ว่าง (Not Empty)

#### **หลักการทำงาน**

##### ตรวจสอบค่าของตัวชี้ Top

###### ถ้า Top แสดงสถานะ “ไม่มีข้อมูล” → isEmpty() = True

###### ถ้า Top ชี้ไปยังข้อมูลใด ๆ → isEmpty() = False

##### ใช้เป็น ตัวช่วยสำคัญ ก่อนจะทำการ Pop() หรือ Peek()

###### เพื่อป้องกันไม่ให้เกิด Stack Underflow

#### **เงื่อนไขที่เกี่ยวข้อง**

##### สแตกว่าง (Empty Stack)

###### isEmpty() คืนค่า True

###### ไม่ควร Pop หรือ Peek เพราะจะเกิด Underflow

##### สแตกไม่ว่าง (Not Empty)

###### isEmpty() คืนค่า False

###### สามารถ Push, Pop หรือ Peek ได้ตามปกติ

#### **ตัวอย่างการทำงาน**

###### **เริ่มต้น**

###### [ ] // ไม่มีข้อมูล

###### isEmpty() → True

###### **Push(10)**

[10] // Top = 10

isEmpty() → False

###### **Pop()**

###### [ ] // ว่างอีกครั้ง

###### isEmpty() → True

### การตรวจสอบว่าสแตกเต็มหรือไม่ (isFull())

คือ การดำเนินการตรวจสอบว่าสแตกสามารถรับข้อมูลใหม่ได้หรือไม่

- ถ้าสแตกมีข้อมูลจนถึงขีดจำกัดที่กำหนดไว้ ถือว่า เต็ม (Full)

- ถ้ายังมีพื้นที่ว่างสำหรับเก็บข้อมูลเพิ่ม ถือว่า ไม่เต็ม (Not Full)

#### **หลักการทำงาน**

1. ตรวจสอบสถานะการจัดเก็บข้อมูลของสแตก

2. ถ้าสแตกมีข้อมูลจนถึง “ขนาดสูงสุด” คืนค่า True

3. ถ้ายังไม่ถึง คืนค่า False

#### **เงื่อนไขที่เกี่ยวข้อง**

##### ****สแตกเต็ม (Full Stack)****

###### isFull() คืนค่า **True**

###### ถ้า Push ข้อมูลเพิ่ม จะเกิด **Stack Overflow**

##### ****สแตกไม่เต็ม (Not Full)****

###### isFull() คืนค่า **False**

###### สามารถ Push ข้อมูลใหม่เข้าไปได้ตามปกติ

#### **ตัวอย่างการทำงาน**

สมมติสแตกมีขนาดสูงสุด capacity = 3

###### **เริ่มต้น**

###### [ ] // ไม่มีข้อมูล

###### isFull() → False

###### **Push(10), Push(20)**

[10,20]

###### isFull() → False

###### **Push(30)**

[10, 20, 30] // เต็มแล้ว

isFull() → True

### ความผิดพลาดที่อาจเกิดขึ้น: Stack Overflow และ Stack Underflow

#### **Stack Overflow**

###### เกิดขึ้นเมื่อมีการ Push ข้อมูลเข้าสแตกในขณะที่สแตก เต็มแล้ว

###### สแตกไม่สามารถรองรับข้อมูลใหม่ได้อีก

###### มักพบใน Array-based Stack ที่มีการกำหนดความจุ ตายตัว

#### **Stack Underflow**

###### เกิดขึ้นเมื่อมีการ Pop ข้อมูลออกจากสแตกในขณะที่สแตกว่างเปล่า

###### ไม่มีข้อมูลเหลือให้ลบออก

###### มักพบในทุกการใช้งานถ้าไม่ตรวจสอบก่อน Pop

## ความซับซ้อนของเวลา (Time Complexity ของสแตก)

คือ การวัดเวลาที่โครงสร้างข้อมูลสแตกใช้ในการทำงานของแต่ละคำสั่งหรือการดำเนินการ โดยใช้การวิเคราะห์เชิงคณิตศาสตร์แบบ **Big-O Notation** เพื่อดูว่า การทำงานจะใช้เวลามากน้อยเพียงใดเมื่อจำนวนข้อมูลในสแตกมีการเปลี่ยนแปลง

### การวิเคราะห์การดำเนินการของสแตก

#### **Initialization (การสร้างสแตก)**

###### การกำหนดค่าเริ่มต้น เช่น การสร้างอาร์เรย์และตั้งค่า top = -1 หรือการตั้ง top = NULL ในแบบลิงก์ลิสต์

###### ใช้เวลาเพียงครั้งเดียว ไม่ขึ้นกับจำนวนข้อมูล

###### มีความซับซ้อนเวลาเป็น O(1)

#### **Push (การเพิ่มข้อมูล)**

การนำข้อมูลใหม่เข้ามาที่ตำแหน่งบนสุดของสแตก

###### ไม่ต้องค้นหาหรือเลื่อนข้อมูลใด ๆ

###### เพียงวางค่าที่ตำแหน่ง Top และเลื่อน Top ไปข้างหน้า

###### ความซับซ้อนเวลา O(1)

#### **Pop (การลบข้อมูลออก)**

การนำข้อมูลที่อยู่บนสุดของสแตกออก

###### แค่คืนค่าข้อมูลที่ Top และเลื่อน Top ไปยังตำแหน่งก่อนหน้า

###### ไม่มีการวนลูปหรือค้นหาข้อมูล

###### ความซับซ้อนเวลา O(1)

#### **Peek/Top (การดูข้อมูลบนสุด)**

ใช้ตรวจสอบว่าข้อมูลบนสุดคืออะไร โดยไม่ต้องลบออก

###### แค่เข้าถึงค่าที่ตำแหน่ง Top

###### ความซับซ้อนเวลา O(1)

#### **isEmpty (ตรวจสอบว่าสแตกว่างหรือไม่)**

ใช้ตรวจสอบว่ามีข้อมูลเหลืออยู่หรือไม่

###### ตรวจสอบด้วยเงื่อนไขง่าย ๆ เช่น top == -1 หรือ top == NULL

###### ความซับซ้อนเวลา O(1)

#### **isFull (ตรวจสอบว่าสแตกเต็มหรือไม่)**

ใช้กับสแตกแบบอาร์เรย์ที่มีการกำหนดขนาดไว้ล่วงหน้า

###### ตรวจสอบด้วยเงื่อนไข top == capacity - 1

###### ความซับซ้อนเวลา O(1)

### เหตุผลที่ทุกการดำเนินการเป็น O(1)

สแตกถูกออกแบบให้ทำงานกับข้อมูล เฉพาะที่ตำแหน่งบนสุด (Top) เท่านั้น จึงไม่ต้องทำการค้นหา เลื่อน หรือวนลูปใด ๆ การเข้าถึงตำแหน่ง Top สามารถทำได้โดยตรงเสมอ ทำให้ทุกการดำเนินการใช้เวลาคงที่ ไม่ขึ้นกับจำนวนข้อมูลในสแตก

## การประยุกต์ใช้งานจริงของสแตก (Applications of Stack)

คือ การนำแนวคิดการทำงานของสแตกไปใช้แก้ปัญหาหรือสนับสนุนการทำงานในระบบคอมพิวเตอร์และสถานการณ์ต่าง ๆ ในชีวิตจริง

### การใช้งานในระบบคอมพิวเตอร์

#### **การย้อนกลับ (Undo/Redo) ในโปรแกรมแก้ไขข้อความหรือรูปภาพ**

###### ทุกครั้งที่มีการแก้ไข จะ Push สถานะเก่าเข้าไปในสแตก

###### เมื่อกด Undo จะ Pop สถานะล่าสุดออกมาเพื่อย้อนกลับ

###### หากต้องการ Redo ใช้สแตกอีกกองหนึ่งเก็บสถานะที่ถูก Undo

#### **ประวัติการเข้าชมเว็บไซต์ (Browser History)**

###### เมื่อเข้าเว็บใหม่ → Push URL ลงสแตก

###### เมื่อกด Back → Pop เว็บล่าสุดออก และกลับไปยังเว็บก่อนหน้า

###### ใช้ร่วมกับสแตกอีกกองหนึ่งเพื่อเก็บ Forward

#### **การประมวลผลนิพจน์ทางคณิตศาสตร์**

###### ใช้สแตกช่วยในการแปลง Infix → Postfix/Prefix

###### ใช้สแตกช่วยในการคำนวณค่าของ Postfix Expression

###### เช่น นิพจน์ (3 + 4) \* 5 จะถูกแปลงเป็น 3 4 + 5 \* ด้วยสแตก

#### **การตรวจสอบวงเล็บ (Parentheses Matching)**

###### ใช้สแตกตรวจสอบว่าวงเล็บเปิดและปิดตรงกันหรือไม่

###### ตัวอย่าง: ((a+b) \* (c-d)) → ถูกต้องเพราะทุกวงเล็บปิดครบ

###### ถ้าวงเล็บไม่ครบ เช่น (a+b))( → สแตกจะตรวจพบข้อผิดพลาด

#### **การทำงานของ Compiler และ Interpreter**

###### ใช้สแตกเก็บ Activation Record ของฟังก์ชัน (Call Stack)

###### เวลาเรียกฟังก์ชันใหม่ → Push ข้อมูลการทำงาน แตก

###### เมื่อฟังก์ชันทำงานเสร็จ → Pop ออกจากสแตก

### การใช้งานในโครงสร้างข้อมูลและอัลกอริทึม

#### **การค้นหาเชิงลึก (Depth-First Search: DFS)**

###### ใช้สแตกเก็บเส้นทางที่ต้องไปต่อ

###### ทำให้สามารถเดินหน้าไปให้ลึกที่สุด แล้วค่อยย้อนกลับ

#### **การทำงานแบบ Recursion**

###### ระบบปฏิบัติการใช้ Call Stack จัดการการเรียกซ้อนของฟังก์ชัน

###### ทุกครั้งที่เรียกฟังก์ชัน → Push ข้อมูลใหม่

###### ทุกครั้งที่ฟังก์ชันจบ → Pop ข้อมูลออก

#### การจัดการปัญหาทางคณิตศาสตร์

###### เช่น การแปลงเลขฐาน (Decimal → Binary)

###### ใช้สแตกเก็บเศษจากการหาร แล้ว Pop ออกมาเพื่อได้เลขฐานสองตามลำดับที่ถูกต้อง

### การใช้งานในชีวิตจริง

#### **การซ้อนจานในโรงอาหาร**

###### วางจานใหม่ = Push

###### หยิบจานออก = Pop

#### **กองหนังสือ/กองกล่อง**

###### หนังสือเล่มที่วางล่าสุดอยู่บนสุด และจะถูกหยิบออกก่อน

## ตัวอย่างโค้ด

### โค้ดตัวอย่าง Python (Array-based Stack)

class Stack:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.items = [None] \* capacity

self.top = -1

def isFull(self):

return self.top == self.capacity - 1

def isEmpty(self):

return self.top == -1

def push(self, value):

if self.isFull():

print("Stack Overflow")

return

self.top += 1

self.items[self.top] = value

def pop(self):

if self.isEmpty():

print("Stack Underflow")

return None

value = self.items[self.top]

self.top -= 1

return value

### โค้ดตัวอย่าง Python (Linked List-based Stack)

# โครงสร้างโหนด

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

# โครงสร้าง Stack

class Stack:

def \_\_init\_\_(self):

self.top = None # เริ่มต้นว่าง

def isEmpty(self):

return self.top is None

def push(self, value):

newNode = Node(value)

newNode.next = self.top # เชื่อมโหนดใหม่เข้ากับโหนดเดิม

self.top = newNode # อัปเดต top

print(f"Pushed {value}")

def pop(self):

if self.isEmpty():

print("Stack Underflow ❌")

return None

value = self.top.data

self.top = self.top.next # เลื่อน top ไปที่โหนดถัดไป

print(f"Popped {value}")

return value

def peek(self):

if self.isEmpty():

print("Stack is empty")

return None

return self.top.data

# ทดลองใช้งาน

s = Stack()

s.push(10)

s.push(20)

s.push(30)

print("Top element =", s.peek()) # ได้ 30

s.pop() # นำ 30 ออก

# คิว

## ความหมาย (FIFO)

คิว (Queue) คือโครงสร้างข้อมูลที่รับประกัน “ความยุติธรรมตามเวลา”—สิ่งที่เข้ามาก่อนจะถูกบริการก่อน (First-In First-Out: FIFO) จินตนาการถึงแถวซื้อข้าวเที่ยง: คนแรกที่มายืนรอคือคนแรกที่ได้จ่ายและรับอาหาร การรักษาสัญญา FIFO ทำให้หลายอัลกอริทึมและระบบมีพฤติกรรมคาดเดาได้ เช่น BFS ที่ต้องเดินกราฟตาม “ชั้นของระยะทาง” จากจุดเริ่มต้น หรือระบบพิมพ์งาน/จัดคิวงานที่ต้องไม่ให้คนหลังแซงหน้า

ในมุมมอง ADT (ชนิดข้อมูลนามธรรม) คิวเปิดเพียงไม่กี่การกระทำ—ใส่ของ “ท้ายคิว”, เอาของ “หัวคิว”, แอบดูหัวคิว, และถามว่าคิวว่างไหม—และทั้งหมดต้องมีเวลาเป็นค่าคงที่ (O(1)) เมื่อใช้โครงสร้างพื้นหลังที่ถูกต้อง สิ่งที่ทำให้คิว “เป็นคิวจริงๆ” คืออินเวเรียนต์: ลำดับของสมาชิกในคิวต้องตรงกับลำดับเวลาที่ถูกใส่เข้า และการนำออกต้องเกิดที่หัวเท่านั้น หากวันหนึ่งเราเปลี่ยนใจไปหยิบท้ายหรือแทรกกลาง อินเวเรียนต์นี้พัง ระบบอาจยัง “รันได้” แต่ผลลัพธ์จะไม่ใช่สิ่งที่คิวสัญญาไว้

## โครงสร้างและองค์ประกอบ

การทำให้คิวเร็วและเสถียรมีสองสายหลัก—วงแหวนบนอาร์เรย์ (circular array/ring buffer) และลิงก์ลิสต์แบบมีหัว-หาง—ทั้งสองให้ O(1) สำหรับการใส่และเอาออก หากออกแบบสถานะให้ชัดเจน

### Array-based Queue

เมื่อใช้อาร์เรย์เป็นบัฟเฟอร์ เรากำหนดช่องเก็บข้อมูลขนาดคงที่แล้วถือดัชนีสองตัว: front สำหรับหัวคิว และ rear สำหรับช่องว่างถัดไปที่จะใส่ของใหม่ พร้อมตัวนับ size ที่บอกจำนวนสมาชิกจริง การเดินของดัชนีต้อง “วน” ด้วยโมดูลัส (เช่น (idx+1) % capacity) เพื่อให้เมื่อ rear วิ่งถึงสุดขวาแล้วสามารถวนกลับไป “ต้นอาร์เรย์” โดยไม่สูญเสียคุณสมบัติ FIFO การวนเช่นนี้กำจัดปัญหา wasted space ของคิวอาร์เรย์แบบตรงๆ ที่เคยปล่อยให้ช่องซ้ายที่ถูกเอาออกไปแล้วกลับมาใช้ไม่ได้อีก

รายละเอียดเล็กๆ ที่มักทำคนน็อคคือการบอกสถานะ “ว่าง/เต็ม” อย่างชัดเจน วิธีที่ทนและอ่านง่ายคือถือ size แล้วให้ “ว่างเมื่อ size==0” และ “เต็มเมื่อ size==capacity” ผลพลอยได้คือเราสามารถใช้ความจุได้ครบทุกช่อง ต่างจากเทคนิคใช้ front==rear แทน “ว่าง” ซึ่งต้องเสียพื้นที่ 1 ช่องเป็นตัวคั่น นอกจากนี้ การแมปตำแหน่งตรรกะ (ลำดับที่ k จากหัว) ไปสู่ดัชนีจริงทำได้ด้วยสมการ (front + k) mod capacity จึงยังเข้าถึงหัวคิวเป็น O(1) เสมอ

บนเครื่องจริง อาร์เรย์มอบ locality สูง—ข้อมูลอยู่ติดกันในหน่วยความจำ ซีพียูแคชจึงชอบ ทำให้ throughput ดีกว่าลิงก์ลิสต์อย่างสม่ำเสมอในเวิร์คโหลดที่ใส่-เอาออกถี่ๆ

### Linked List-based Queue

ลิงก์ลิสต์ทำให้คิว “ยืดหยุ่น” โดยไม่ต้องล็อกความจุล่วงหน้า โหนดแต่ละตัวเก็บค่าและตัวชี้ไปโหนดถัดไป เราถือ head และ tail ไว้คนละหน้าที่: หัวสำหรับการนำออก หางสำหรับการใส่เข้า เมื่อคิวว่าง (ทั้งหัว-หางเป็น null) การใส่ตัวแรกต้องเซ็ตทั้งสองชี้ไปโหนดใหม่ทันที เมื่อเอาสมาชิกสุดท้ายออกต้องคืน tail=null เพื่อให้สถานะ “ว่าง” สมบูรณ์ไม่เหลือปลายใดลอยอยู่

ลิงก์ลิสต์ตัดปัญหา overflow จากความจุคงที่ แต่แลกมาด้วย overhead ของ pointer ต่อโหนดและ locality ที่ด้อยกว่า ในภาษาแบบมี GC โค้ดสั้นและปลอดภัยขึ้น แต่ค่าคงที่จากการกระโดดหน่วยความจำก็ยังมีผลกับประสิทธิภาพ

## การดำเนินการ (Operations)

### สร้าง Queue (Initialization)

วงแหวนบนอาร์เรย์เริ่มจากจองบัฟเฟอร์ความจุที่ต้องการ ตั้ง front=0, rear=0, size=0 หากอยากให้ “ขยายได้” เมื่อเต็ม เราต้องยอมมีขั้นตอน copy-up: สร้างอาร์เรย์ใหม่ที่ใหญ่ขึ้น คัดลอกสมาชิกตามลำดับตรรกะเริ่มจากหัวไปทีละชิ้นลงช่อง 0..size-1 แล้วตั้ง front=0 และ rear=size วิธีนี้ทำให้ enqueue ยังคง O(1) แบบแอมอร์ไทซ์เหมือน dynamic array ทั่วไป ส่วนลิงก์ลิสต์ก็ตรงไปตรงมาด้วย head=tail=null, size=0

### enqueue(x)

บนวงแหวน การใส่ของต้องป้องกันกรณีเต็มก่อน ถ้าเต็มและเราไม่รองรับการขยาย ให้ “ล้มเหลวชัดเจน” (คืนสถานะหรือโยนข้อยกเว้น) แทนการเขียนทับค่าที่ยังไม่ได้เอาออก จากนั้นวาง x ลงที่ rear, หมุน rear = (rear+1)%capacity, และเพิ่ม size หนึ่ง บนลิงก์ลิสต์ ให้สร้างโหนดใหม่แล้วเชื่อมต่อหางเดิมไปยังโหนดนี้ ย้าย tail มาที่โหนดใหม่ และเพิ่ม size เช่นกัน

### dequeue()

บนวงแหวน หาก size==0 คือ underflow—อย่าพยายามอ่านค่าที่ไม่อยู่จริง เมื่อมีของ ให้ดึงจาก arr[front] แล้วหมุน front=(front+1)%capacity ลด size หนึ่ง บนลิงก์ลิสต์ให้ดึงจาก head แล้วเลื่อนไป head.next ถ้าเลื่อนจน head เป็น null ต้องตามด้วย tail=null เพื่อไม่ให้คิวอยู่ในสถานะ “ว่างแต่หางยังชี้” ซึ่งทำให้ enqueue ถัดไปต่อปลายผิด

### peek() / front()

คือการ “ชำเลืองหัวคิว” โดยไม่เปลี่ยนสถานะใดๆ หากคิวว่างควรส่งสัญญาณที่ตรวจจับได้ (เช่น null หรือ exception) แทนการคืนค่าหลอก เพื่อบังคับผู้ใช้โครงสร้างให้จัดการกรณีว่างอย่างตั้งใจ

### isEmpty()

วิธีตรวจที่น่าเชื่อถือสุดคือพึ่ง size ซึ่งอัปเดตในทุก enqueue/dequeue และมีค่าอยู่ในช่วง 0..capacity เสมอ การผูกความหมาย “ว่าง” กับเงื่อนไข front==rear นั้นเปราะกว่าและบังคับให้เสียหนึ่งช่องจุ

## ประเภทของ Queue

### Circular Queue (แก้ปัญหา wasted space)

คิววงแหวนคือหัวใจของคิวแบบอาร์เรย์ยุคโปรดักชัน ความคิดหลักคือ “หมุนดัชนีด้วยโมดูลัส” เพื่อให้บัฟเฟอร์นำช่องว่างกลับมาใช้ซ้ำได้โดยไม่ละเมิด FIFO เราสามารถมองว่ามีเข็มสองอันวิ่งบนหน้าปัด: เข็มหัว (front) และเข็มท้าย (rear) ที่วิ่งทับรอยเดิมได้แต่ไม่เคย “ข้ามกัน” เพราะ size กั้นไว้ การวิเคราะห์ถูกต้องทำได้ด้วยฟังก์ชันแมประหว่างลำดับตรรกะ 0..size-1 ไปสู่อินเด็กซ์จริง (front+i) mod capacity และอินเวเรียนต์ 0 ≤ size ≤ capacity

### Double-ended Queue (Deque)

ดีคิวให้ความคล่องตัวสูง—รับเข้า/ส่งออกได้ทั้งหัวและท้าย—จึงเป็นเครื่องมือยุทธศาสตร์ในหลายอัลกอริทึม เช่น “คิวโมโนโทน” สำหรับหาค่าสูงสุดในทุกหน้าต่างยาว k แบบ O(n) หลักการคือรักษาดีคิวให้ลดหลั่นตามค่า เมื่อค่ามาใหม่ที่ดีกว่าเข้าท้าย ให้ปัดค่าที่ด้อยกว่าท้ายทิ้งจนหมด แล้วเก็บแต่ “ผู้สมัครชนะ” ไว้หัว วิธีนี้ทำให้หัวของดีคิวเสมือน “คำตอบของหน้าต่างปัจจุบัน” โดยไม่ต้องคำนวณซ้ำ ในแง่โครงสร้าง ดีคิวทำได้ทั้งบนอาร์เรย์วงแหวน (ให้ locality) หรือ doubly-linked list (ให้ลบ/แทรกกลางง่ายขึ้น) โดยปกติภาษาระดับสูงอย่าง Python มี collections.deque ที่จูนมาอย่างดีให้ O(1) สำหรับการดัน-ดึงทั้งสองด้าน

### Priority Queue

เมื่อ “ความสำคัญ” ชนะ “เวลา” เราปรับจาก FIFO เป็นคิวตามลำดับความสำคัญ (priority) โครงสร้างมาตรฐานคือ binary heap ซึ่งรับประกันว่า “ดูค่าสูงสุด/ต่ำสุด” ทำได้ O(1) และ “แทรกหรือนำตัวท็อปออก” ทำได้ O(log n) โดยใช้พื้นที่ O(n) ต้นไม้ของ heap มีโครงแบบ complete ทำให้การจัดเก็บบนอาร์เรย์เป็นธรรมชาติ (ลูกซ้าย-ขวาคือ index 2i และ 2i+1) อย่างไรก็ดี heap “ไม่ stable” โดยธรรมชาติ—หาก priority เท่ากัน ลำดับเวลาอาจสลับได้—วิธีรักษาเสถียรภาพคือแปะ “เลขลำดับมาถึง” เป็นคีย์รอง เช่นจัดเรียงตาม (priority, seqno) แล้วเลือก min/ max ด้วย comparator ที่คำนึงถึงคีย์ทั้งคู่

## Time Complexity

เมื่ออินเวเรียนต์ได้รับการรักษา คิวบนวงแหวนและลิงก์ลิสต์ต่างให้เวลาคงที่สำหรับการใส่ การเอาออก การชำเลืองหัว และการตรวจว่าง ความแตกต่างจริงๆ อยู่ที่ค่าคงที่และหน่วยความจำ: วงแหวนให้อัตราการกดแคชที่ดีกว่าและ footprint คงที่ตาม capacity ในขณะที่ลิงก์ลิสต์ยืดหยุ่นแต่มี overhead ของ pointer ต่อโหนด หากอนุญาตให้วงแหวน “โตอัตโนมัติ” เวลาของ enqueue ยังถือเป็น O(1) แบบแอมอร์ไทซ์ เพราะการคัดลอกเกิดเป็นครั้งคราวและต้นทุนเฉลี่ยต่อการใส่ถูกเฉลี่ยต่ำลง ส่วน priority queue บน heap ยอมใช้ O(log n) สำหรับการแทรกและการดึงค่าสูงสุด/ต่ำสุดแลกกับความสามารถจัดคิวตามความสำคัญที่คาดเดาได้

## การใช้งานจริง

### Job Scheduling

ในโลก OS/บริการแบ็กเอนด์ งาน (job/process/thread/task) ไหลเข้ามาอย่างต่อเนื่องแล้วถูก “กัก” ไว้ใน ready queue รอทรัพยากร (CPU/worker) ตามนโยบายจัดคิว ตัวอย่างยอดฮิตคือ FCFS/FIFO และ Round-Robin (RR)

#### FCFS (First-Come, First-Served) ไม่พรีเอ็มป์ ใครมาก่อนเริ่มก่อนจนจบ เหมาะกับงาน burst สั้นพอๆ กัน จุดเสียคือ *convoy effect*—ถ้ามีงานยาวหนึ่งตัว จะดันให้ทุกงานหลังรอนานขึ้นมาก

#### Round-Robin (RR) แบ่งเวลาตัวประมวลผลเป็นช่วงสั้นๆ ชื่อ quantum (เช่น 50 ms) แล้ว “หมุน” งานในคิวทีละช่วง ถ้างานยังไม่เสร็จจะถูกต่อท้ายคิว เป็นการทำให้ระบบ interactive ตอบสนองดีขึ้น (response time ลดลง) และยุติธรรมกว่า FCFS

ในงานจริงคิวของ RR มักทำเป็น circular queue เพราะการใส่ท้าย/เอาหน้า/วนหัวท้ายเป็น O(1) เสถียร ส่วนเมตริกที่ใช้ดูภาพรวมคือ waiting time, turnaround, response time, throughput, และ CPU utilization (กฎ Little: L=λWL=\lambda WL=λW ใช้กะภาพรวมได้ดีถ้าระบบเสถียร)

#### จำลองสถานการณ์ (ทำให้เห็นภาพจริง)

สร้างตัวจำลอง สองนโยบาย ให้แล้ว: FCFS (ไม่พรีเอ็มป์) และ RR (พรีเอ็มป์ด้วย quantum 0.05 วินาที) สร้างงานแบบ Poisson (อัตรามาถึง λ\lambdaλ) และสุ่ม burst time แบบเอ็กซ์โปเนนเชียล (ดูคล้ายงานจริงที่เบอร์สท์คละสั้น-ยาว)

### Print Queue

กรณีคลาสสิกของ คิวเดียว-เครื่องเดียว (single-server queue): เอกสารถูกส่งเข้า spooler แล้วเครื่องพิมพ์หยิบ “หัวคิว” มาพิมพ์ตามลำดับ—นี่คือ FIFO ตรงๆ อย่างไรก็ตาม ในโลกจริงเวลาให้บริการไม่ได้เท่ากันเลย: จำนวนหน้าแตกต่างกันมาก (heavy-tail), โหมดสี/duplex ช้ากว่า และรุ่นเครื่องต่างกันจึงมี “pages-per-minute (PPM)” ไม่เท่ากัน

เพราะ service time แตกต่างมาก เราจึงอยากเห็นผลของนโยบายพิเศษ เช่น SRPT (Shortest Remaining Processing Time) แบบ *preemptive*—ใครเหลือพิมพ์น้อยสุดก็ได้ทำก่อน (ถ้ามีงานหน้าเดียวเข้ามาระหว่างที่กำลังพิมพ์งาน 50 หน้า SRPT จะยอม “แทรก” เพื่อให้ลูกค้าที่งานเล็กออกจากระบบเร็วขึ้น) ผลตามทฤษฎีคือ SRPT ลดเวลารอเฉลี่ย ได้ดีที่สุด แต่ต้องแลกกับ fairness (งานใหญ่ถูกผลักไปเรื่อยๆ) จึงมักต้องปรับใช้อย่างระวังในองค์กร

#### จำลองสถานการณ์ (พร้อมข้อมูลหน้า/สี/duplex)

จำลองสองนโยบายให้เทียบกัน

FIFO (ไม่พรีเอ็มป์)

SRPT (พรีเอ็มป์จาก “เวลาที่เหลือ” ซึ่งคำนวณจาก pages/pps + ปัจจัยสี/duplex)

### Simulation (ร้านค้า, สายการบิน)

นี่คือสนามจริงของทฤษฎีคิวแบบ M/M/c: ลูกค้ามาถึงแบบ Poisson (λ\lambdaλ), เวลาให้บริการ Exponential (μ\muμ), และมีเซิร์ฟเวอร์ ccc ช่อง (แคชเชียร์/เคาน์เตอร์เช็คอิน) ระบบ “เสถียร” ถ้า ρ=λ/(cμ)<1\rho = \lambda / (c\mu) < 1ρ=λ/(cμ)<1 มิฉะนั้นคิวจะโตไม่หยุด

ในความเป็นจริงยังมี “ความอดทน” (patience) ของลูกค้า—ถ้ารอนานเกินก็ abandon (ทิ้งคิว) สมจริงมากขึ้นสำหรับร้าน/สนามบิน จำลองแบบ Discrete-Event (ใกล้ของจริง)

#### จำลอง M/M/c ที่รองรับ abandonment (patience ~ Exponential)

เหตุการณ์: arrive, service\_start, service\_end, abandon

คิวเป็น deque: เซิร์ฟเวอร์ว่างเก็บในลิสต์ เซิร์ฟเวอร์ยุ่งเก็บใน min-heap ตามเวลาเสร็จ

สร้างตารางเมตริกต่อคน: arrival\_ts, start\_ts, end\_ts, wait, system\_time, served?, abandoned?

ทำสรุป: served\_count, abandon\_count, avg\_wait, p95\_wait, avg\_system

## ตัวอย่างโค้ด

### Circular Queue (Array-based → enqueue/dequeue) C/Java

#### Circular Queue (Array-based; enqueue/dequeue) — C

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <assert.h>

typedef struct {

int \*buf;

int capacity;

int front;

int rear;

int size;

} CQueue;

CQueue\* cq\_new(int capacity) {

assert(capacity > 0);

CQueue\* q = (CQueue\*)malloc(sizeof(CQueue));

q->buf = (int\*)malloc(sizeof(int) \* capacity);

q->capacity = capacity;

q->front = 0;

q->rear = 0;

q->size = 0;

return q;

}

bool cq\_empty(CQueue\* q) { return q->size == 0; }

bool cq\_full (CQueue\* q) { return q->size == q->capacity; }

bool cq\_enqueue(CQueue\* q, int x) {

if (cq\_full(q)) return false; // overflow

q->buf[q->rear] = x;

q->rear = (q->rear + 1) % q->capacity; // หมุน index

q->size++;

return true;

}

bool cq\_dequeue(CQueue\* q, int \*out) {

if (cq\_empty(q)) return false; // underflow

\*out = q->buf[q->front];

q->front = (q->front + 1) % q->capacity;

q->size--;

return true;

}

bool cq\_peek(CQueue\* q, int \*out) {

if (cq\_empty(q)) return false;

\*out = q->buf[q->front];

return true;

}

void cq\_free(CQueue\* q) {

free(q->buf);

free(q);

}

// mini test

int main(void) {

CQueue\* q = cq\_new(3);

assert(cq\_enqueue(q, 10));

assert(cq\_enqueue(q, 20));

assert(cq\_enqueue(q, 30));

assert(!cq\_enqueue(q, 40)); // เต็มแล้ว

int v;

assert(cq\_dequeue(q, &v) && v == 10);

assert(cq\_enqueue(q, 40)); // ช่องเดิมกลับมาใช้ใหม่ (ring)

assert(cq\_peek(q, &v) && v == 20);

while (cq\_dequeue(q, &v)) printf("%d ", v); // 20 30 40

printf("\n");

assert(!cq\_dequeue(q, &v)); // ว่างแล้ว -> underflow

cq\_free(q);

**}**

#### Circular Queue แบบโตอัตโนมัติ — Java

import java.util.NoSuchElementException;

public class RingQueue<T> {

private Object[] a;

private int front = 0;

private int rear = 0;

private int size = 0;

public RingQueue(int capacity) {

if (capacity <= 0) throw new IllegalArgumentException("capacity>0");

this.a = new Object[capacity];

}

public boolean isEmpty() { return size == 0; }

public int size() { return size; }

public void enqueue(T x) {

if (size == a.length) grow();

a[rear] = x;

rear = (rear + 1) % a.length;

size++;

}

@SuppressWarnings("unchecked")

public T dequeue() {

if (isEmpty()) throw new NoSuchElementException("underflow");

T v = (T) a[front];

a[front] = null; // help GC

front = (front + 1) % a.length;

size--;

return v;

}

@SuppressWarnings("unchecked")

public T peek() {

if (isEmpty()) throw new NoSuchElementException("empty");

return (T) a[front];

}

private void grow() {

int n = a.length, m = n << 1;

Object[] b = new Object[m];

for (int i = 0; i < size; i++) {

b[i] = a[(front + i) % n];

}

a = b;

front = 0;

rear = size;

}

public static void main(String[] args) {

RingQueue<String> q = new RingQueue<>(2);

q.enqueue("A"); q.enqueue("B"); q.enqueue("C"); // โตอัตโนมัติ

System.out.println(q.dequeue()); // A

q.enqueue("D");

System.out.println(q.peek()); // B

while (!q.isEmpty()) System.out.println(q.dequeue()); // B C D

}

}

#### Linked List Queue — Java

import java.util.NoSuchElementException;

public class LinkedQueue<T> {

private static final class Node<E> {

E val; Node<E> next;

Node(E v) { val = v; }

}

private Node<T> head = null;

private Node<T> tail = null;

private int size = 0;

public boolean isEmpty() { return size == 0; }

public int size() { return size; }

public void enqueue(T x) {

Node<T> n = new Node<>(x);

if (tail == null) head = tail = n;

else { tail.next = n; tail = n; }

size++;

}

public T dequeue() {

if (isEmpty()) throw new NoSuchElementException("underflow");

T v = head.val;

head = head.next;

if (head == null) tail = null;

size--;

return v;

}

public T peek() {

if (isEmpty()) throw new NoSuchElementException("empty");

return head.val;

}

}

### Python → ใช้ collections.deque

from collections import deque

# คิวมาตรฐาน

q = deque()

q.append(10) # enqueue

q.append(20)

q.append(30)

print(q[0]) # peek -> 10

print(q.popleft()) # dequeue -> 10

print(len(q) == 0) # isEmpty?

# ดีคิวในงานอัลกอริทึม: sliding window maximum แบบ O(n)

def sliding\_max(nums, k):

dq = deque() # จะเก็บ "ดัชนี" ของผู้สมัครที่ยังชนะ

out = []

for i, x in enumerate(nums):

while dq and nums[dq[-1]] <= x:

dq.pop() # ปัดผู้แพ้ท้ายออก

dq.append(i)

if dq[0] <= i - k: # หลุดหน้าต่าง

dq.popleft()

if i >= k - 1: # เริ่มมีหน้าต่างครบ k

out.append(nums[dq[0]])

return out

print(sliding\_max([1,3,-1,-3,5,3,6,7], 3)) # [3,3,5,5,6,7]

# ต้นไม้

## ความหมายของโครงสร้างข้อมูลแบบต้นไม้

Tree (ต้นไม้) เป็นโครงสร้างข้อมูลแบบลำดับชั้น (Hierarchical Data Structure) ประกอบด้วยโหนด (Node) ที่เชื่อมต่อกันด้วยเส้นเชื่อม (Edge) ใช้แทนความสัมพันธ์ลักษณะ “หนึ่ง-ต่อ-หลาย” (One-to-Many) เช่น โครงสร้างแฟ้มข้อมูล, ระบบไฟล์, ผังองค์กร โดยมีลักษณะสำคัญคือ:

* มีโหนดราก (root) เพียงหนึ่งโหนด
* เป็นกราฟไม่มีวงจร (acyclic)
* ทุกโหนด (ยกเว้น root) มี parent เดียว (connected acyclic graph where each node except root has exactly one parent)
* สามารถมองเป็นชุดของโหนดที่เชื่อมกันเป็นลำดับชั้น (hierarchy)

## โครงสร้างและองค์ประกอบของต้นไม้

### Tree (ทรี) คือโครงสร้างข้อมูลแบบลำดับชั้น (Hierarchical Data Structure) ที่ใช้แทนความสัมพันธ์ระหว่างข้อมูลที่มีลักษณะเป็น หนึ่งต่อหลาย (One-to-Many relationship) โดยพื้นฐานแล้ว Tree ประกอบด้วย โหนด (Node) ที่เชื่อมโยงกันด้วย เส้นเชื่อม (Edge)

###### Node (โหนด) → แทนข้อมูลแต่ละตัว

###### Edge (เส้นเชื่อม) → แทนความสัมพันธ์ระหว่างโหนด เช่น ความเป็นพ่อ-ลูก

###### โครงสร้างจะมีโหนดพิเศษที่เรียกว่า Root (ราก) ซึ่งเป็นจุดเริ่มต้น และโหนดอื่น ๆ จะสืบทอดความสัมพันธ์ต่อ ๆ กันลงมา

### องค์ประกอบสำคัญของต้นไม้

#### Node (โหนด)

หน่วยข้อมูลพื้นฐานของ Tree 1 Node จะประกอบด้วย:

###### Key/Value → ค่าที่เก็บ เช่น ตัวเลข, ตัวอักษร, ข้อมูลอื่น ๆ

###### Pointer/Reference → ตัวชี้ไปยัง Child (ลูก) ของโหนดนั้น

ตัวอย่าง:

###### -Node A (key=10) → child: Node B, Node C

###### -เปรียบเหมือน“กล่อง” ที่เก็บข้อมูล และยังบอกว่าเชื่อมต่อกับใครบ้าง

#### Edge (ขอบ)

###### เส้นเชื่อมความสัมพันธ์ระหว่างโหนด มักจะเป็น Parent → Child (เชื่อมทางเดียว)

###### ตัวอย่าง: A → B แปลว่า A เป็นพ่อแม่ของ B

#### Root (ราก)

โหนดบนสุดของ Tree ไม่มี Parent Tree 1 ต้นมี Root ได้เพียงโหนดเดียว เช่น ใน Family Tree, "ปู่/ย่า" คือ Root

#### Parent (พ่อแม่)

โหนดที่มีลูก (Child) อย่างน้อยหนึ่งโหนด

###### ตัวอย่าง: A มีลูก B, C → A = Parent

#### Child (ลูก)

#### โหนดที่อยู่ภายใต้ Parent แต่ละ Child จะมี Parent เพียงหนึ่งโหนดเท่านั้น

###### ตัวอย่าง: A → B, C → B และCเป็นChild ของ A

#### Sibling (พี่น้อง)

โหนดที่มี Parent เดียวกัน

###### ตัวอย่าง: B และ C เป็น Sibling เพราะมี Parent คือ A

#### Leaf (ใบ)

โหนดที่ ไม่มีลูกเป็นจุดสิ้นสุดของ Tree โหนดที่ไม่มีลูกต่ออีก (Child = 0)เปรียบเสมือนจุดสิ้นสุดของ Tree

###### ตัวอย่าง: Node D, E ถ้าไม่มี Child → Leaf

#### Degree ของ Node

จำนวน Child ของโหนดนั้น ๆ

###### เช่น Node A มีลูก 2 ตัว (B, C) → Degree(A) = 2

#### Degree ของ Tree

ค่าสูงสุดของ Degree ที่เกิดขึ้นใน Tree ทั้งหมด

###### เช่น Binary Tree → Degree ≤ 2 เสมอ

#### Subtree (ทรีย่อย)

ต้นไม้ที่เกิดขึ้น Parent บางโหนดและลูกของมัน Subtree แต่ละต้นก็ยังคงมีโครงสร้าง Tree ที่สมบูรณ์

###### ตัวอย่าง: ถ้า A มี Subtree ที่เริ่มจาก B → Subtree ของ B ก็เป็น Tree ที่สมบูรณ์ในตัวเอง

#### Path (เส้นทาง)

ลำดับของ Node ที่เชื่อมต่อกันด้วย Edge

###### เช่น Path จาก A → E คือ [A, C, E]

#### Length of Path

จำนวน Edge ใน Path

###### เช่น Path [A, C, E] → มี Edge 2 (A→C, C→E)

#### Depth(v)

ระยะทาง (จำนวน Edge) จาก Root ไปถึง Node v Root มี Depth = 0

###### ตัวอย่าง: Root = A → B อยู่Depth = 1 → D อยู่ Depth = 2

#### Height(v)

ความยาวของ Path ที่ยาวที่สุดจาก Node v ลงไปยัง Leaf

Leaf → Height = 0

###### ตัวอย่าง: A → B → D → D เป็น Leaf → Height(D)=0, Height(B)=1, Height(A)=2

#### Height(tree)

ความสูงของ Tree = Height ของ Root ใช้บอกว่าต้นไม้สูงกี่ชั้น

#### Level

บางครั้งเท่ากับ Depth Root มักอยู่ที่ Level 0 (บางตำราใช้ Level 1) Child ของ Root อยู่ Level 1 (หรือ 2 ตามคอนเวนชัน)

## การดำเนินการของต้นไม้

### Binary Search Tree (BST)เป็น Binary Tree ที่มีลักษณะเฉพาะตัวในการจัดเรียงข้อมูล ทำให้สามารถค้นหา/เพิ่ม/ลบ ได้เร็วโดยเฉลี่ย O(log n)หากข้อมูลเรียงก่อนจะสร้าง ทำให้ BST กลายเป็นแบบไม่สมดุล (Worst Case = O(n))

**ข้อดีของ BST**:เข้าถึงข้อมูลได้รวดเร็ว สนับสนุนการค้นหาที่มีประสิทธิภาพ

**ข้อเสียของ BST**:ถ้าไม่ Balance จะช้ากว่า Linked List จำเป็นต้องใช้อัลกอริทึม Balance เช่น AVL หรือ Red-Black Tree

**การประยุกต์ใช้ต้นไม้ในชีวิตจริง** :

###### ระบบแฟ้ม (File System): เช่น Root → Folder → File

###### โครงสร้างองค์กร (Organization Chart)

###### AI: Decision Tree, Minimax Tree

###### การเข้ารหัสข้อมูล (Huffman Tree)

###### การคำนวณทางคณิตศาสตร์ (Expression Tree)

###### ระบบค้นหาอัตโนมัติ (Auto Suggestion using Trie)

###### ระบบฐานข้อมูล (B+ Tree สำหรับ Index)

**ปัญหาที่ใช้แผนภาพต้นไม้ :**

###### แสดงลำดับชั้น เช่น ผังองค์กร

###### การหาค่าที่เหมาะสม เช่น Max-Min

###### การคำนวณแบบแบ่งซ้ายขวา เช่น การจัดการนิพจน์

###### การตัดสินใจแบบ IF-ELSE หลายระดับ เช่น Decision Tree

###### การเรียนรู้ของเครื่อง (Machine Learning) เช่น Random Forest

ตัวอย่างแบบฝึกหัดที่เกี่ยวข้อง :

###### เขียนฟังก์ชัน BST insert, delete และ search

###### เขียนโค้ด Inorder, Preorder, Postorder traversal

###### วาดภาพ Tree ตามข้อมูลที่ให้ และเรียง Inorder

###### แปลง Expression infix เป็น Expression Tree และคำนวณ

###### ออกแบบระบบแฟ้มโดยใช้ Tree Diagram

#### **Insert (ใน BST)**

#### เดินลงจาก root เปรียบเทียบค่า → ถ้า < ไปซ้าย, ถ้า > ไปขวา → จนถึงตำแหน่งว่าง แล้วแทรกเป็น leaf

##### Complexity:

Time: O(h) (h = height of tree)

Space: O(h) recursion stack (หรือ O(1) if iterative)

Note: ใน worst-case (skewed) h = n → O(n). ใน balanced tree h = O(log n) → O(log n).

#### **Search (BST)**

#### เดินเทียบจาก root ลงไปตามกฎ < / > Recursive/Iterative ทั้งคู่

Time: O(h). Space iterative O(1), recursive O(h).

#### **Delete (BST)**

ต้องพิจารณา 3 กรณีหลัก:

Case 1 — Node is leaf: ลบได้ตรง ๆ (set parent.child = null)

Case 2 — Node has one child: ย้าย child ขึ้นมาแทน (parent.child = node.child)

Case 3 — Node has two children: หา Inorder successor (เล็กสุดใน right subtree) หรือ Inorder predecessor (ใหญ่สุดใน left subtree) มาแทนค่า node แล้วลบ successor (หรือ predecessor) ซึ่งจะเป็นกรณี leaf หรือ one-child เสมอ

การแทนด้วย inorder successor คงรักษา property ของ BST

Complexity: O(h) time, O(h) recursion.

Exampleสเต็ป: ลบ node 50 ใน tree ที่มี children ทั้งสอง:

หา successor (เช่น 52)

copy key 52 → node

delete original node 52 (ซึ่งน่าจะเป็น leaf หรือ one-child)

#### **ความซับซ้อนของเวลา (Time Complexity ของต้นไม้)**

###### -ทุกการ traversal เยี่ยม ทุกโหนด 1 ครั้ง → เวลา O(n)

-หน่วยความจำ: โดยทั่วไป O(h) (h = height) สำหรับ recursion หรือ explicit stack

-สำหรับ tree skewed (เหมือน linked list): h = n → O(n) space

-สำหรับ balanced tree: h ≈ O(log n) → O(log n) space

-Morris (inorder) → O(n) เวลา แต่ O(1) พื้นที่ (แลกกับการแก้ pointer ชั่วคราว)

### การดำเนินการของต้นไม้

### Tree Traversal (การเดินต้นไม้) คือการเดินไปตามโหนดของต้นไม้ เพื่ออ่านหรือจัดเรียงข้อมูล

#### **Preorder** คือ การเรียงลำดับไปทางซ้ายให้สุดก่อน แล้วแสดงราก หลังจากนั้นให้ไปทางขวา แล้วแสดงราก

**ลักษณะเด่นและเหตุผลการใช้งาน:**

* เยี่ยมชม parent ก่อนลูก ดังนั้น โครงสร้างของต้นไม้ (root-first structure) ถูกเก็บก่อน → เหมาะกับการ *serialize* หรือ *copy* ต้นไม้ (เพราะเมื่ออ่าน root ก่อน เราสามารถสร้าง node ก่อนแล้วค่อยเชื่อมลูก)
* ในการแปลงเป็น notation แบบ prefix (Polish notation) ของ expression tree → Preorder ให้รูปแบบ prefix
* ในเชิงเวลา/หน่วยความจำ: เยี่ยมทุกโหนดครั้งเดียว → O(n) เวลา, stack/recursion depth = ความสูงของต้นไม้ (O(h) space)

**ตัวอย่างการคิดเป็นสเต็ป (walk-through) :**

##### เริ่มที่ A → เยี่ยมชม A (ออก A)ไปซ้าย B → เยี่ยมชม B (ออก B)และไปซ้ายของ B → D → เยี่ยมชม D (ออก D) → กลับขึ้นไป B → ไปขวา E → เยี่ยมชม E (ออก E)และกลับ root → ไปขวา C → เยี่ยมชม C (ออก C) → แล้ว F, G ตามลำดับ ผลลัพธ์: A B D E C F

A

/ \

B C

/ \ \

D E F

Preorder: A → B → D → E → C → F

เก็บโหนดที่ยังต้องไปเยี่ยมภายหลังไว้ใน stack — เมื่อ pop มาเยี่ยม ให้ผลเหมือนเข้า node ก่อนลงลูก

#### **Inorder** คือ การทำซับทรีซ้ายให้เสร็จก่อนเยี่ยมชม rootแล้วทำซับทรีด้านขวา **ลักษณะเด่นและเหตุผลการใช้งาน:**

* เมื่อใช้กับ Binary Search Tree (BST) → Inorder จะคืนค่าคีย์ใน ลำดับเรียงจากน้อยไปมาก (sorted ascending) เพราะ property ของ BST: ทุกค่าใน left < root < ทุกค่าใน right
* เหมาะกับการดึงข้อมูลเรียงลำดับ, ตรวจสอบว่าโครงสร้างเป็น BST หรือไม่ (เช็กว่าผล inorder เป็นลำดับไม่ลด)
* เวลา/หน่วยความจำ: O(n) เวลา, O(h) stack/recursion

การคิดเป็นสเต็ป (walk-through) แสดง snapshot ของการเคลื่อนที่แบบแนวคิด

**ตัวอย่างการเดินทางแบบ Inorder** :

เริ่ม A → ยังไม่เยี่ยม A → ย้ายซ้ายไป B → ย้ายซ้ายไป D

D ไม่มีซ้าย → เยี่ยม D → กลับขึ้นมาที่ B → เยี่ยม B → ไปขวา E → เยี่ยม E → กลับขึ้นมาที่ A → เยี่ยม A → ไปขวา C → process F,C,G → สรุป D B E A F C G

หมายเหตุเชิงปฏิบัติ:Inorder กับ tree ที่ไม่ใช่ BST ไม่มีความหมายว่า "เรียง" ใด ๆ โดยปริยาย ต้องมีสมบัติของ BST จึงได้ลำดับตัวเลขที่มีประโยชน์

A

/ \

B C

/ \ \

D E F

Inorder: D → B → E → A → C → F

#### **Postorder** คือ การทำซับทรีซ้ายให้เสร็จ แล้วทำซับทรีขวาให้เสร็จ แล้วค่อยเยี่ยม root **ลักษณะเด่นและเหตุผลการใช้งาน:**

* เหมาะกับการทำงานที่ต้องให้ ลูกถูกประมวลผลก่อน parent เช่น:
  + การลบหน่วยความจำ (free nodes): ถ้าลบ parent ก่อน อาจเกิด dangling reference — postorder ลบลูกก่อนจึงปลอดภัย
  + การประเมิน expression tree: ค่า operand (leaf) ถูกคำนวณก่อน operator (internal node) → Postorder ให้รูปแบบ postfix (Reverse Polish Notation) ที่สามารถประเมินด้วย stack ได้ง่าย
* เวลา/หน่วยความจำ: O(n) เวลา, O(h) space

**ตัวอย่างการเดินทางแบบ Postorder** :

เริ่ม A → ลงซ้าย B → ลงซ้าย D → D ไม่มีลูก → เยี่ยม D → กลับ B → ไปขวา E → เยี่ยม E → ทั้งซ้าย/ขวาของ B เสร็จ → เยี่ยม B → ไปขวาของ A (C) → process F,G → เยี่ยม C → สุดท้ายเยี่ยม A → ผล D E B F G C A

A

/ \

B C

/ \ \

D E F

Postorder: D → E → B → F → C → A

**การเปรียบเทียบเชิงพฤติกรรม**

###### -Preorder — เก็บโครงสร้างก่อนลูก → serialize / copy / produce prefix notation

-Inorder — เหมาะกับการได้ลำดับเรียงของ BST → *sorted output / check BST*

-Postorder — ประมวล/ลบลูกก่อน parent → *evaluate expressions / free memory safely / produce postfix notation*

#### **Level-order Traversal (BFS)**

ลำดับการเยี่ยมชมแต่ละชั้นของต้นไม้จากบนลงล่างใช้ Queue ในการเก็บโหนดที่จะเยี่ยมชมต่อไปเหมาะกับการ ค้นหาแบบกว้าง (Breadth-first search) หรือแสดงโครงสร้างต้นไม้ทีละชั้น

**ตัวอย่างการเดินทางแบบ Level-order Traversal** :

A

/ \

B C

Level-order: A B C

## Time Complexity

Time Complexity เป็นการ วัดจำนวนขั้นตอนการทำงานของอัลกอริทึม ตามจำนวนข้อมูลที่ต้องจัดการ สำหรับต้นไม้ (Tree) โดยเฉพาะ Binary Search Tree (BST) หรือ Balanced Tree เรามักสนใจ เวลาที่ใช้ในการค้นหา แทรก หรือ ลบโหนด

### BST ปกติ (ไม่สมดุล) BST มีโครงสร้างแบบ ซ้าย < root < ขวา

การทำงานเช่น search, insert, delete มักเริ่มจากโหนด root แล้วเปรียบเทียบค่าไปเรื่อย ๆ

ถ้า ต้นไม้สมดุลดีแต่ละ level จะมีจำนวนโหนดเพิ่มขึ้น เป็นกำลังสอง ดังนั้นความสูงของต้นไม้ (Height) จะประมาณ log₂(n)เวลาเฉลี่ย ของ search/insert/delete จะเท่ากับ O(log n) เพราะเราต้องเดินผ่านความสูงของต้นไม้เท่านั้นแต่ถ้า ต้นไม้ไม่สมดุล เช่นเราเพิ่มเลขเรียง 1, 2, 3, 4, 5 โหนดจะเรียงเป็นเส้นตรงเหมือน linked list เราต้องเดินผ่านทุกโหนดเพื่อค้นหาหรือลบ → เวลาแย่สุด O(n) นั่นคือถ้ามี 1000 โหนด เราอาจต้องเช็คทุกโหนด 1000 ครั้ง

### Balanced Tree (เช่น AVL Tree, Red-Black Tree) Balanced Tree มี กลไกปรับสมดุลอัตโนมัติ AVL Tree: จะหมุนต้นไม้ (rotation) เมื่อความสูงของซ้าย-ขวาไม่สมดุล

Red-Black Tree: มีเงื่อนไขสีเพื่อรักษาสมดุลต้นไม้ เพราะ Balanced Tree รักษาความสูงให้ใกล้ log₂(n) เสมอดังนั้น เวลาเฉลี่ยและเวลาแย่สุด ของทุก operation เช่น search, insert, delete → O(log n) ข้อดีคือ ไม่ว่าเราจะเพิ่มหรือลบโหนดแบบเรียงหรือตามลำดับใด ๆ

Performance จะยังคง เร็วและคงที่ไม่เกิดกรณีเหมือน BST ปกติที่กลายเป็น linked list

## การใช้งานจริงของต้นไม้

### Database index

- ฐานข้อมูลขนาดใหญ่ เช่น MySQL, PostgreSQL จะมีข้อมูลเป็น ตาราง (Table) ซึ่งถ้าเราอยากหาข้อมูลบางแถว (Row) โดยตรง หากไม่มีโครงสร้างพิเศษต้อง สแกนทั้งตาราง → ช้ามาก

- เพื่อแก้ปัญหานี้ Database จะสร้าง Index ซึ่งส่วนมากใช้ Binary Search Tree (BST) หรือ B-Tree

- B-Tree เป็นต้นไม้แบบ Balanced ที่ออกแบบมาสำหรับจัดการข้อมูลขนาดใหญ่บนดิสก์

- การค้นหาข้อมูลด้วย Index → เร็วขึ้นมาก เพราะไม่ต้องตรวจทุกแถว ใช้เวลาประมาณ log(n) ของจำนวนแถว

ตัวอย่าง: เรามีตารางนักศึกษา 1 ล้านแถว ต้องการค้นหา "รหัสนักศึกษา = 12345" ถ้าไม่มี index → scan 1 ล้านแถว ถ้ามี B-Tree index → เดินโหนดเพียงประมาณ 20-30 ครั้งก็เจอ

### Expression tree

Expression Tree เป็น ต้นไม้ที่เก็บนิพจน์ทางคณิตศาสตร์ โหนดใบ (Leaf Node) จะเก็บตัวเลขหรือตัวแปร โหนดภายใน (Internal Node) จะเก็บตัวดำเนินการ เช่น +, -, \*, / การคำนวณค่าของนิพจน์ นิยมใช้ Postorder Traversal (Left → Right → Root) เพราะเราต้องคำนวณค่าของลูกซ้ายและลูกขวาก่อน แล้วจึงเอาผลลัพธ์มาประมวลผลที่โหนดแม่

ตัวอย่าง: นิพจน์ 3 + 4 \* 5

Expression Tree จะเป็น:

+

/ \

3 \*

/ \

4 5

Postorder Traversal → 3 4 5 \* + → คำนวณ 4\*5=20 → 3+20=23

### File system

คอมพิวเตอร์จัดการ โฟลเดอร์และไฟล์เป็นต้นไม้ Root Directory → โฟลเดอร์ย่อย → ไฟล์ การแสดงผลหรือค้นหาไฟล์บางครั้งใช้ Level-order Traversal (BFS) เพราะเราต้องการ ดูไฟล์ทีละชั้น จากบนลงล่าง เช่น โปรแกรม Explorer/ Finder จะแสดงโฟลเดอร์หลักก่อน แล้วค่อยแสดงโฟลเดอร์ย่อยและไฟล์

ข้อดี: เห็นโครงสร้างชั้นชัดเจน,ค้นหาไฟล์เร็วขึ้นแบบเป็นระบบ

## ตัวอย่างโค้ด

### โครงสร้างพื้นฐานของ Binary Tree

#### โค้ดตัวอย่าง Python class Node:

# โครงสร้างของ Node

# คลาส Node

class Node:

def \_\_init\_\_(self, data):

self.data = data # ค่าของโหนด

self.left = None # ลูกซ้าย

self.right = None # ลูกขวา

# คลาส Binary Tree

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

# ฟังก์ชันเพิ่มค่า (แบบง่าย ใส่ซ้ายก่อนขวา)

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, node, data):

# ใส่ข้อมูลด้านซ้ายถ้ายังว่าง

if node.left is None:

node.left = Node(data)

# ถ้าว่างด้านซ้ายไม่แล้ว → ไปใส่ด้านขวา

elif node.right is None:

node.right = Node(data)

else:

# ถ้าเต็มแล้ว → ลองแทรกฝั่งซ้ายก่อน

self.\_insert(node.left, data)

# ฟังก์ชันแสดงผลแบบ Preorder

def preorder(self, node):

if node:

print(node.data, end=" ")

self.preorder(node.left)

self.preorder(node.right)

# ตัวอย่างการใช้งาน

bt = BinaryTree()

for value in [1, 2, 3, 4, 5]:

bt.insert(value)

print("Preorder Traversal:")

bt.preorder(bt.root)

### การเดินต้นไม้ (Tree Traversal)

#### Inorder (Left, Root, Right)

Python : # ฟังก์ชัน Inorder Traversal

def inorder(self, node):

if node:

self.inorder(node.left) # เยี่ยมชมซ้าย

print(node.data, end=" ") # แสดงค่าโหนด

self.inorder(node.right) # เยี่ยมชมขวา

โดยจะใช้ร่วมกับ Binary Tree

# คลาส Node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

# คลาส BinaryTree

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, node, data):

if node.left is None:

node.left = Node(data)

elif node.right is None:

node.right = Node(data)

else:

self.\_insert(node.left, data)

# Inorder Traversal

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

self.inorder(node.right)

# ตัวอย่างใช้งาน

bt = BinaryTree()

for value in [1, 2, 3, 4, 5]:

bt.insert(value)

print("Inorder Traversal:")

bt.inorder(bt.root)

#### Preorder (Root, Left, Right)

##### Python :

# ฟังก์ชัน Preorder Traversal

def preorder(self, node):

if node:

print(node.data, end=" ") # แสดงค่าโหนดก่อน

self.preorder(node.left) # เยี่ยมชมลูกซ้าย

self.preorder(node.right) # เยี่ยมชมลูกขวา

โดยจะใช้ร่วมกับ Binary Tree

# คลาส Node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

# คลาส BinaryTree

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

# ฟังก์ชัน insert แบบง่าย (ใส่ซ้ายก่อนขวา)

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, node, data):

if node.left is None:

node.left = Node(data)

elif node.right is None:

node.right = Node(data)

else:

self.\_insert(node.left, data)

# Preorder Traversal

def preorder(self, node):

if node:

print(node.data, end=" ")

self.preorder(node.left)

self.preorder(node.right)

# ตัวอย่างใช้งาน

bt = BinaryTree()

for value in [1, 2, 3, 4, 5]:

bt.insert(value)

print("Preorder Traversal:")

bt.preorder(bt.root)

#### Postorder (Left, Right, Root)

##### Python :

# ฟังก์ชัน Postorder Traversal

def postorder(self, node):

if node:

self.postorder(node.left) # เยี่ยมชมลูกซ้ายก่อน

self.postorder(node.right) # เยี่ยมชมลูกขวา

print(node.data, end=" ") # แสดงค่าโหนดสุดท้าย

โดยจะใช้ร่วมกับ Binary Tree

# คลาส Node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

# คลาส BinaryTree

class BinaryTree:

def \_\_init\_\_(self):

self.root = None

# ฟังก์ชัน insert แบบง่าย (ใส่ซ้ายก่อนขวา)

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, node, data):

if node.left is None:

node.left = Node(data)

elif node.right is None:

node.right = Node(data)

else:

self.\_insert(node.left, data)

# Postorder Traversal

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.data, end=" ")

# ตัวอย่างใช้งาน

bt = BinaryTree()

for value in [1, 2, 3, 4, 5]:

bt.insert(value)

print("Postorder Traversal:")

bt.postorder(bt.root)

#### การค้นหาค่าใน Binary Search Tree (BST)

##### Python class Node:

# คลาส Node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

# คลาส BST

class BST:

def \_\_init\_\_(self):

self.root = None

# ฟังก์ชันเพิ่มข้อมูล

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, node, data):

if data < node.data:

if node.left is None:

node.left = Node(data)

else:

self.\_insert(node.left, data)

elif data > node.data:

if node.right is None:

node.right = Node(data)

else:

self.\_insert(node.right, data)

# ถ้า data == node.data จะไม่แทรกซ้ำ

# ฟังก์ชันค้นหา

def search(self, data):

return self.\_search(self.root, data)

def \_search(self, node, data):

if node is None:

return False

if data == node.data:

return True

elif data < node.data:

return self.\_search(node.left, data)

else:

return self.\_search(node.right, data)

# ฟังก์ชัน Traversal แบบ Inorder

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

self.inorder(node.right)

# ตัวอย่างการใช้งาน

if \_\_name\_\_ == "\_\_main\_\_":

tree = BST()

values = [50, 30, 70, 20, 40, 60, 80]

# ใส่ค่าเข้า BST

for v in values:

tree.insert(v)

print("Inorder Traversal (เรียงจากน้อยไปมาก):")

tree.inorder(tree.root) # ผลลัพธ์: 20 30 40 50 60 70 80

# กราฟ

## 5.1 ความหมาย

กราฟ (Graph) หมายถึง กลุ่มของโหนด (Vertices) และกลุ่มของเส้นเชื่อม (Edges) ที่ใช้เชื่อมโยงโหนดแต่ละคู่เข้าด้วยกัน อาจมีทิศทางหรือไม่ก็ได้ กราฟสามารถใช้แทนความสัมพันธ์ของข้อมูลได้หลากหลาย เช่น ความสัมพันธ์ของบุคคลในสังคม เส้นทางคมนาคม หรือการเชื่อมต่อในระบบเครือข่ายคอมพิวเตอร์

นิยามทางคณิตศาสตร์

กราฟ คือ คู่อันดับ G = (V, E)

V คือเซตของจุดยอด (Vertices หรือ Nodes)

E คือเซตของเส้นเชื่อม (Edges หรือ Arcs)

## โครงสร้างและศัพท์สำคัญ

### Vertex และ Edge

#### **Vertex (จุดยอด):** จุดหรือโหนดที่เป็นองค์ประกอบของกราฟ ใช้แทนวัตถุ เช่น เมือง, บุคคล, คอมพิวเตอร์

#### **Edge (เส้นเชื่อม):** เส้นที่เชื่อมระหว่างจุดยอดสองจุด เพื่อบอกความสัมพันธ์ระหว่างวัตถุ

### Directed และ Undirected

#### **Directed Graph (กราฟมีทิศทาง):** เส้นเชื่อมมีทิศทาง กำหนดการเคลื่อนจากจุดยอดหนึ่งไปอีกจุดยอดหนึ่ง

##### นิยามทางคณิตศาสตร์: G = (V, A) โดย A เป็นเซตของคู่อันดับ (Ordered Pair)

#### **Undirected Graph (กราฟไม่มีทิศทาง):** เส้นเชื่อมไม่มีทิศทาง ความสัมพันธ์ระหว่างจุดยอดเป็นแบบสองทาง (สมมาตร)

##### นิยามทางคณิตศาสตร์: G = (V, E) โดย E เป็นเซตของคู่ไม่เรียงลำดับ (Unordered Pair)

### Weighted และ Unweighted

#### Weighted Graph (กราฟมีน้ำหนัก): แต่ละเส้นเชื่อมมีค่าตัวเลขกำกับ (Weight) เช่น ระยะทาง, ค่าใช้จ่าย, เวลา

##### นิยามทางคณิตศาสตร์: G = (V, E, w) โดย w: E → R กำหนดน้ำหนักของแต่ละเส้นเชื่อม

#### Unweighted Graph (กราฟไม่มีน้ำหนัก): ทุกเส้นเชื่อมมีค่าเท่ากัน หรือไม่มีการกำหนดค่าน้ำหนัก

## การแทนกราฟ (Graph Representation)

การแทนข้อมูลของกราฟสามารถทำได้หลายวิธี ที่นิยมคือ Adjacency Matrix และ Adjacency List

### Adjacency Matrix

เป็นการแทนกราฟโดยใช้เมทริกซ์ขนาด n x n เมื่อ n คือจำนวนโหนดในกราฟ:

#### ถ้าโหนด i เชื่อมกับโหนด j ให้ค่า a(i,j) = 1 (หรือค่า weight ถ้าเป็น weighted graph)

#### ถ้าไม่เชื่อมให้ a(i,j) = 0

### Adjacency List

เป็นการเก็บข้อมูลโดยใช้ลิงค์ลิสต์หรือรายการ สำหรับแต่ละโหนดจะเก็บรายชื่อโหนดที่เชื่อมถึง

#### ตัวอย่าง (ไม่มีทิศทาง):

A → B, C

B → A, D, E

C → A, D

D → B, C, E

E → B, D

#### ตัวอย่าง (มีทิศทาง):

A → C

B → A, E

C → D

D → B, E

E → D

## การดำเนินการ (Graph Operations)

การดำเนินการบนกราฟ (Graph Operations) คือกระบวนการที่เราสามารถ “แก้ไขโครงสร้าง” ของกราฟได้ เช่น การเพิ่ม/ลบ Vertex (จุดยอด) หรือ Edge (เส้นเชื่อม) การทำงานเหล่านี้สำคัญเพราะกราฟมักใช้แทนระบบที่ มีการเปลี่ยนแปลงตลอดเวลา เช่น เครือข่ายสังคม (Social Network), เครือข่ายถนน, หรือระบบคอมพิวเตอร์

### เพิ่ม Vertex (Add Vertex)

การเพิ่ม Vertex หมายถึงการเพิ่มโหนดใหม่เข้าไปในกราฟ พร้อมทั้งปรับโครงสร้างการแทนกราฟให้รองรับโหนดใหม่นั้น

#### Adjacency Matrix: ต้องขยายเมทริกซ์เพิ่มทั้งแถวและคอลัมน์ (ค่าเริ่มต้นเป็น 0 หรือ ∞ ถ้าเป็น weighted)

#### Adjacency List: เพิ่มรายการใหม่ใน list เพื่อแทนโหนดที่เพิ่มเข้ามา

### เพิ่ม Edge (Add Edge)

การเพิ่ม Edge หมายถึงการสร้างเส้นเชื่อมระหว่างโหนดสองโหนด

#### Directed Graph: ต้องระบุทิศทางจากโหนดต้นทาง → โหนดปลายทาง

#### Undirected Graph: เพิ่ม edge ทั้งสองทิศ (A→B และ B→A)

#### Weighted Graph: ต้องกำหนดค่าน้ำหนัก (weight) ของ edge ด้วย

#### ใน adjacency matrix จะเปลี่ยนค่าของ cell ที่ตรงกับ edge เป็น 1 หรือค่าน้ำหนัก ส่วน adjacency list จะเพิ่มปลายทางเข้าไปใน list ของต้นทาง

### ลบ Vertex (Remove Vertex)

การลบ Vertex จะทำให้ต้อง:

#### ลบรายการของ vertex นั้นออกจาก adjacency list และลบ edges ทั้งหมดที่เกี่ยวข้อง (ทั้ง incoming และ outgoing edges)

#### ใน adjacency matrix ต้องลบทั้งแถวและคอลัมน์ที่เกี่ยวข้องกับ vertex นั้น

#### ต้องระวังการรีอินเด็กซ์ (reindex) ของโหนดที่เหลือในกราฟหลังลบ

### ลบ Edge (Remove Edge)

การลบ Edge หมายถึงการเอาเส้นเชื่อมออก โดยไม่กระทบตัว vertex:

#### Adjacency Matrix: เปลี่ยนค่าของ cell ที่ตรงกับ edge เป็น 0 หรือ ∞ (ไม่มีการเชื่อมต่อ)

#### Adjacency List: ลบรายการของปลายทางออกจาก list ของต้นทาง

#### สำหรับกราฟที่มีน้ำหนัก อาจเลือกตั้งค่าน้ำหนักให้เป็น ∞ หรือ null เพื่อแสดงว่าไม่มีการเชื่อม

## อัลกอริทึมบนกราฟ (Graph Algorithms)

กราฟเป็นโครงสร้างที่ยืดหยุ่นและซับซ้อน การประมวลผลหรือวิเคราะห์กราฟต้องอาศัย **อัลกอริทึม** ซึ่งสามารถแบ่งออกเป็น 3 กลุ่มหลัก:

Traversal (การท่องกราฟ)

Shortest Path (เส้นทางสั้นที่สุด)

Minimum Spanning Tree (ต้นไม้ครอบคลุมขั้นต่ำ)

### Traversal (การท่องกราฟ)

การท่องกราฟคือ **กระบวนการเยี่ยมชมทุกโหนดในกราฟ** ตามรูปแบบที่กำหนด มีประโยชน์หลายด้าน เช่น

ตรวจสอบว่าโหนดทั้งหมดเชื่อมต่อกันไหม

หาค่าทางสั้นในกราฟ unweighted

ใช้เป็นพื้นฐานในการแก้ปัญหาอื่น ๆ เช่น Cycle Detection, Topological Sorting

#### BFS (Breadth First Search)

##### แนวคิด: BFS (Breadth-First Search): เดินเป็นชั้น ๆ — เหมาะกับการหา *shortest path* เมื่อทุกขอบมีค่าเท่ากัน (unweighted), หา level ของแต่ละ node, หรือหา connected components ในกราฟที่ไม่ชี้ทิศ

##### ใช้ Queue ในการเก็บโหนดที่ต้องเยี่ยมชม

##### เหมาะสำหรับหาค่า shortest path ในกราฟ unweighted

##### ขั้นตอน

1.ใส่โหนดเริ่มต้นในคิว

2.ดึงออกจากคิว → ทำเครื่องหมายว่าเยี่ยมชมแล้ว

3.ใส่เพื่อนบ้าน (neighbors) ที่ยังไม่ถูกเยี่ยมชมเข้าคิว

4.ทำซ้ำจนคิวว่าง

#### DFS (Depth First Search)

##### แนวคิด: DFS (Depth-First Search): ดำดิ่งให้สุดแล้วค่อยย้อน — เหมาะกับการตรวจหา cycle, สร้าง topological order บน DAG, หาจุดเชื่อม (bridges/articulation points)

##### ใช้ Recursion หรือ Stack ในการเก็บโหนด

##### เหมาะสำหรับตรวจสอบการเชื่อมโยง (Connectivity) และหาเส้นทางทั้งหมด

##### ขั้นตอน

1.เยี่ยมชมโหนดปัจจุบันและทำเครื่องหมาย

2.เรียก DFS กับเพื่อนบ้านที่ยังไม่ถูกเยี่ยมชม

3.ทำซ้ำจนเยี่ยมชมครบ

### Shortest Path (เส้นทางสั้นที่สุด)

เส้นทางสั้นที่สุดคือการหาค่า ระยะทางรวมต่ำสุด จากโหนดต้นทางไปยังโหนดปลายทางในกราฟ โดยพิจารณาน้ำหนักของ edge

มีสองอัลกอริทึมหลักที่ใช้กันมาก

#### Dijkstra Algorithm

##### **แนวคิดใช้**: หาเส้นทางสั้นที่สุดในกราฟที่ น้ำหนักเป็นบวกเท่านั้นใช้ Greedy Approach: เลือกโหนดที่มีระยะทางจากต้นทางน้อยที่สุดในแต่ละขั้นอัปเดตระยะทางของเพื่อนบ้าน (neighbors) ของโหนดนั้นทำซ้ำจนกว่าทุกโหนดจะถูกเยี่ยมชม

##### ขั้นตอนการทำงาน

1.กำหนดระยะทางของโหนดต้นทางเป็น 0 และโหนดอื่นเป็น ∞

2.ใช้ Priority Queue (Min Heap) เก็บโหนดพร้อมระยะทาง

3.ดึงโหนดที่มีระยะทางน้อยที่สุดออกจาก heap

4.สำหรับเพื่อนบ้านของโหนดนั้น:

4.1.คำนวณระยะทางใหม่ = ระยะทางของโหนดปัจจุบัน + น้ำหนักของ edge

4.2.ถ้าระยะทางใหม่ < ระยะทางเก่า → อัปเดตและใส่ลง heap

5.ทำซ้ำจน heap ว่าง → ได้ระยะทางสั้นที่สุดจากต้นทางไปทุกโหนด

### Minimum Spanning Tree (MST)

MST คือ ต้นไม้ย่อยที่เชื่อมทุกโหนดของกราฟโดยใช้เส้นเชื่อมน้อยที่สุด (Minimum Total Weight)

#### **เงื่อนไข:**

ครอบคลุมทุกโหนด (Spanning Tree)

ไม่มี cycle (เป็นต้นไม้)

ผลรวมของน้ำหนัก edge น้อยที่สุด

#### **Prim’s Algorithm**

##### แนวคิด

เป็น Greedy Algorithmเริ่มจาก โหนดใดก็ได้เลือก edge น้ำหนักต่ำสุด ที่เชื่อมจากโหนดใน MST ปัจจุบันไปยังโหนดที่ยังไม่อยู่ใน MSTทำซ้ำจนเชื่อมทุกโหนด

##### โครงสร้างข้อมูลที่ใช้

1.Priority Queue (Min-Heap): เลือก edge ที่น้ำหนักต่ำสุดในแต่ละขั้น

2.Set หรือ List เก็บโหนดที่ถูกเลือกแล้ว

##### ขั้นตอน

1.เลือกโหนดเริ่มต้น (Start Node)

2. เพิ่มทุก edge ของโหนดนั้นลงใน Priority Queue

3.ดึง edge ที่น้ำหนักต่ำสุดจาก PQ

3.1ถ้าโหนดปลายทางยังไม่อยู่ใน MST → เพิ่มโหนดนั้นและ edge ลง MST

3.2เพิ่ม edge ของโหนดใหม่ลง PQ

4.ทำซ้ำจน MST มี V-1 edge

#### Kruskal’s Algorithm

##### แนวคิดหลัก: เป็นอีกหนึ่ง Greedy Algorithmเลือก edge จาก น้ำหนักต่ำสุดไปสูงสุด ใช้ Union-Find (Disjoint Set) ป้องกันเกิด cycle หยุดเมื่อมี edge ใน MST ครบ V-1 เส้น

##### ขั้นตอน

1.เรียง edge ทั้งหมดจากน้ำหนักน้อย → มาก

2.สำหรับแต่ละ edge:

2.1ถ้าโหนดต้นทางและปลายทางยังไม่เชื่อมกันใน MST → เพิ่ม edge

2.2ใช้ Union-Find เช็กและรวม Set ของโหนด

3.ทำซ้ำจน MST ครบ V-1 edge

## Time Complexity (ประสิทธิภาพเชิงเวลา)

Time Complexity (ประสิทธิภาพเชิงเวลา) คือแนวคิดในวิทยาการคอมพิวเตอร์ที่ใช้วัดว่า อัลกอริทึมใช้เวลามากน้อยเพียงใด เมื่อขนาดของข้อมูล (input size) เพิ่มขึ้น โดยไม่สนใจเวลาจริงที่ใช้ในการประมวลผล แต่เน้นที่ การเติบโตของจำนวนขั้นตอน ที่อัลกอริทึมต้องทำงาน

### BFS (Breadth-First Search)

#### **แนวคิด**

เยี่ยมชมโหนด **level by level** ใช้ **Queue** เพื่อจัดลำดับโหนดที่ต้องไปต่อเหมาะกับกราฟ **unweighted**

#### **ขั้นตอน**

1.เริ่มจากโหนดต้นทาง → mark visited

2.ใส่เพื่อนบ้านทั้งหมดที่ยังไม่เยี่ยมชมใน queue

3.ดึงโหนดออกจาก queue → mark visited

4.ทำซ้ำจน queue ว่าง

### DFS (Depth-First Search)

#### **แนวคิด:**

เยี่ยมชมโหนด ลึกที่สุดก่อน แล้วย้อนกลับใช้ Stack หรือ Recursion

#### **ขั้นตอน**

1.เลือก **โหนดเริ่มต้น** → mark visited

2.สำหรับเพื่อนบ้านของ node ปัจจุบัน:

2.1ถ้ายังไม่ visited → **เรียก DFS ซ้ำ (recursion)**

3.ทำซ้ำจนทุก node เยี่ยมชม

### Dijkstra Algorithm

#### **แนวคิด:**

หา shortest path สำหรับกราฟ น้ำหนักบวก เลือก node ที่ระยะทางน้อยที่สุดจาก nodes ที่ยังไม่เยี่ยมชม (ใช้ Priority Queue)

#### **ขั้นตอน**

1.กำหนด distance ของทุก node = ∞

2.กำหนด distance ของ node เริ่มต้น = 0

3.ใส่ node เริ่มต้นลง Priority Queue (min-heap)

4.ทำซ้ำจน Priority Queue ว่าง:

4.1ดึง node ปัจจุบันที่มี distance น้อยที่สุด

4.2ตรวจสอบเพื่อนบ้าน

4.3ถ้า distance ใหม่ < distance เก่า → update distance → ใส่ลง queue

5.จบ → distance ของทุก node = shortest path จาก start

### Bellman-Ford Algorithm

#### **แนวคิด**:

หา shortest path สำหรับกราฟ มีnegative edge Relax ทุก edge V-1 รอบ → ตรวจสอบ negative cycle

#### **ขั้นตอน**

1.กำหนด distance ของทุก node = ∞

2.กำหนด distance ของ node เริ่มต้น = 0

3.ทำ V-1 รอบ:

3.1ตรวจสอบทุก edge u → v กับ weight w

3.2ถ้า distance[u] + w < distance[v] → update distance[v]

4.ตรวจสอบ negative cycle รอบที่ V:

4.1ถ้า distance[v] ลดลง → มี negative cycle

### Prim’s Algorithm (MST)

#### **แนวคิด:**

เริ่มจากโหนดใดก็ได้ → ขยาย MST โดยเลือก edge เบาที่สุดจาก node ที่อยู่ใน MST ไป node ที่ยังไม่อยู่ ทำซ้ำจนครบทุก node

#### **ขั้นตอน**

1.เลือก node เริ่มต้น → mark in MST

2.ใส่ edges ของ node เริ่มต้นลง Priority Queue (เรียงจาก weight น้อยสุด)

3.ทำซ้ำจน MST ครบทุก node:

3.1ดึง edge ที่ weight น้อยสุด

3.2ถ้า node ปลายทางยังไม่อยู่ใน MST → mark node → เพิ่ม edges ของ node ปลายทางลง queue

4.MST เสร็จ → ครอบคลุมทุก node, น้ำหนักรวมต่ำสุด

### Kruskal’s Algorithm

#### **แนวคิด:**

เลือก edge จากน้อยไปมาก → เอามาเป็น MST ถ้าไม่สร้าง cycleใช้ Union-Find เพื่อตรวจ cycle

#### **ขั้นตอน**

1.จัด edges ทั้งหมดจากน้อยไปมาก

2.เริ่มต้น MST = empty

3.ทำซ้ำจน MST มี V-1 edges:

3.1เลือก edge ที่ weight น้อยสุด

3.2ตรวจสอบ cycle ด้วย Union-Find

3.3ถ้าไม่เกิด cycle → เพิ่ม edge ลง MST

4.MST เสร็จ → ครอบคลุมทุก node

## การใช้งานจริง (Applications)

กราฟถูกนำไปใช้ในหลายด้านของชีวิตและเทคโนโลยี เพราะสามารถแทนความสัมพันธ์ระหว่างสิ่งต่าง ๆ ได้อย่างยืดหยุ่น ตั้งแต่การสื่อสารระหว่างผู้คน การเคลื่อนที่ในโลกจริง ไปจนถึงการวางแผนเส้นทางในเกมและระบบ AI

### Social Network: แสดงความสัมพันธ์ระหว่างผู้ใช้ (Friendship Graph)

ในเครือข่ายสังคมออนไลน์ เช่น Facebook, Instagram หรือ LinkedIn ผู้ใช้แต่ละคนสามารถแทนด้วย **Node (Vertex)** ส่วนความสัมพันธ์ เช่น การเป็นเพื่อนหรือการติดตาม จะถูกแทนด้วย **Edge (เส้นเชื่อม)** กราฟนี้สามารถใช้ได้ทั้งแบบมีทิศทาง (Directed Graph) เช่น ผู้ใช้ A ติดตาม B แต่ B ไม่จำเป็นต้องติดตาม A หรือแบบไม่มีทิศทาง (Undirected Graph) เช่น ความสัมพันธ์แบบเพื่อนกัน การใช้กราฟใน Social Network ทำให้สามารถวิเคราะห์ความสัมพันธ์ระหว่างผู้ใช้ได้ เช่น การค้นหากลุ่มเพื่อนที่ใกล้ชิดที่สุด การแนะนำเพื่อนใหม่ หรือการระบุผู้มีอิทธิพลสูงในเครือข่าย โดยใช้ **อัลกอริทึม DFS/BFS** เพื่อตรวจสอบการเชื่อมโยงหรือหาค่า shortest path ระหว่างผู้ใช้

### GPS / Shortest Path: หาทางสั้นที่สุดระหว่างสองตำแหน่ง

ในระบบ GPS หรือการนำทาง กราฟถูกใช้เพื่อแทนเส้นทางและสถานที่ จุดสำคัญ เช่น เมืองหรือแยกถนน เป็น Node ส่วนถนนหรือเส้นทางระหว่างจุดต่าง ๆ เป็น Edge โดย edge เหล่านี้อาจมีน้ำหนัก (Weight) เช่น ระยะทาง เวลาเดินทาง หรือค่าใช้จ่าย

กราฟนี้ช่วยให้ระบบ GPS หาทางสั้นที่สุดหรือเร็วที่สุดจากจุดหนึ่งไปยังอีกจุดหนึ่งได้ โดยใช้ Dijkstra Algorithm สำหรับกราฟที่มีน้ำหนักบวก หรือ Bellman-Ford Algorithm สำหรับกราฟที่อาจมีน้ำหนักติดลบ

### Network Routing: วางเส้นทางส่งข้อมูลในเครือข่าย

ในระบบเครือข่ายคอมพิวเตอร์ เช่น Internet, LAN หรือระบบโทรคมนาคม กราฟถูกใช้แทนโครงสร้างเครือข่าย

Router หรือคอมพิวเตอร์แต่ละเครื่อง = Node

การเชื่อมต่อระหว่างอุปกรณ์ = Edge

น้ำหนักของ Edge = bandwidth, latency หรือค่าใช้จ่ายในการส่งข้อมูล

การวิเคราะห์กราฟในเครือข่ายช่วยให้สามารถหาเส้นทางที่ดีที่สุดสำหรับส่ง packet ข้อมูล ลดเวลาและค่าใช้จ่าย นอกจากนี้ยังใช้ Minimum Spanning Tree (Prim/Kruskal) ในการออกแบบเครือข่ายให้เชื่อมทุก node แบบมีต้นทุนต่ำที่สุด และใช้ BFS/DFS ตรวจสอบความเชื่อมโยงและความเสถียรของระบบ

### AI Search: ใช้ในเส้นทางการตัดสินใจของเกม

ในเกมหรือ AI ระบบกราฟใช้แทน สถานะหรือจุดบนแผนที่

Node = ตำแหน่งบน map หรือ state ของเกม

Edge = การเคลื่อนที่จากตำแหน่งหนึ่งไปยังอีกตำแหน่งหนึ่ง

Weight = ค่า cost ของการเคลื่อนที่ เช่น เวลา, ความพยายาม, หรือพลังงาน

อัลกอริทึม BFS/DFS ถูกใช้เพื่อตรวจสอบทุกเส้นทางหรือค้นหา path ที่สั้นที่สุดในกรณีที่ไม่ซับซ้อน ส่วน Dijkstra หรือ A\* ใช้สำหรับหาเส้นทาง optimal เมื่อ edge มีน้ำหนักแตกต่างกัน นอกจากนี้ยังสามารถใช้กราฟแทน state space ของ AI ในการตัดสินใจ เช่น การวางแผนเส้นทางการเดินของตัวละครเกมหรือหาทางออกใน puzzle

## ตัวอย่างโค้ด (Code Examples)

### BFS (Breadth First Search) – Adjacency List

Python

from collections import deque

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['G'],

'F': ['H'],

'G': [],

'H': []

}

def bfs(graph, start):

visited = []

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

visited.append(node)

queue.extend([neighbor for neighbor in graph[node] if neighbor not in visited])

return visited

print(bfs(graph, 'A'))

## DFS (Depth First Search) – Adjacency List

### Python

def dfs(graph, node, visited=None):

if visited is None:

visited = []

visited.append(node)

for neighbor in graph[node]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

return visited

graph = {

'A': ['B','C'],

'B': ['D','E'],

'C': ['F'],

'D': [],

'E': ['G'],

'F': ['H'],

'G': [],

'H': []

}

print(dfs(graph, 'A'))

## Dijkstra Algorithm – Python (Weighted Graph)

### Python

import heapq

def dijkstra(graph, start):

distances = {node: float('inf') for node in graph}

distances[start] = 0

pq = [(0, start)]

while pq:

current\_distance, current\_node = heapq.heappop(pq)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(pq, (distance, neighbor))

return distances

weighted\_graph = {

'A': [('B',6), ('C',7)],

'B': [('D',5), ('E',4)],

'C': [('D',2)],

'D': [('E',2)],

'E': []

}

print(dijkstra(weighted\_graph, 'A'))