สารบัญ

[บทที่ 1 อาร์เรย์และลิงค์ลิสต์ 3](#_Toc207458021)

[1.1 ความหมายของอาร์เรย์และลิงค์ลิสต์ (Array and Link list) 3](#_Toc207458022)

[1.2 ความหมายอาร์เรย์ 3](#_Toc207458023)

[1.3 โครงสร้างของอาร์เรย์ (Array Structure) 3](#_Toc207458024)

[1.4 ลักษณะโครงสร้างอาร์เรย์ 4](#_Toc207458025)

[1.5 ชนิดของอาร์เรย์ 4](#_Toc207458026)

[1.6 ความหมายลิงค์ลิสต์ 7](#_Toc207458027)

[1.7 โครงสร้างของลิงค์ลิสต์ 7](#_Toc207458028)

[1.8 โครงสร้างลิงค์ลิสต์แบบทิศทางเดียว (Single Linked List) 8](#_Toc207458029)

[1.9 การทํางานของลิงค์ลิสต์แบบทิศทางเดียว 8](#_Toc207458030)

[บทที่ 2 โครงสร้างคิว 16](#_Toc207458031)

[2.1 ความหมายของโครงสร้างคิว 16](#_Toc207458032)

[2.2 โครงสร้างคิว 16](#_Toc207458033)

[2.3 การดำเนินการของคิว 16](#_Toc207458034)

[2.4 การแทนคิวด้วยอาร์เรย์ 17](#_Toc207458035)

[2.5 การดำเนินงานของแถวคอย (Queue Operation) 18](#_Toc207458036)

[2.6 โครงสร้างข้อมูลประเภทของ Queue) 19](#_Toc207458037)

[2.7 การประยุกต์ใช้ Queue 19](#_Toc207458038)

[2.8 ประสิทธิภาพเชิงเวลา 21](#_Toc207458039)

[2.9 บทสรุปของคิว 21](#_Toc207458040)

[บทที่ 3 สแตก 32](#_Toc207458041)

[3.1 ความหมายของสแตก 32](#_Toc207458042)

[3.2 โครงสร้างของสแตก (Stack structure) 32](#_Toc207458043)

[3.3 การดำเนินการของสแตก 33](#_Toc207458044)

[3.4 การแทนสแตกด้วยอาร์เรย์ 34](#_Toc207458045)

[3.5 ส่วนประกอบของสแตก 34](#_Toc207458046)

[3.6 การสร้างสแตก 34](#_Toc207458047)

[3.7 ตัวอย่างโค้ดการทำงานของสแตก 35](#_Toc207458048)

[3.8 Operation พื้นฐานของ Stack 39](#_Toc207458049)

[3.9 เปรียบเทียบประสิทธิภาพของการสร้างสแตกด้วยอาร์เรย์และลิงค์ลิสต์ 39](#_Toc207458050)

[3.10 การประยุกต์ใช้งานสแตก 40](#_Toc207458051)

[3.11 การแปลงนิพจน์ infix ให้เป็น postfix 40](#_Toc207458052)

[3.12 Time Complexity ของ Stack 41](#_Toc207458053)

[บทที่ 4 กราฟ 43](#_Toc207458054)

[4.1 ความหมายและลักษณะของกราฟ 43](#_Toc207458055)

[4.1.1 ความหมายของกราฟ 43](#_Toc207458056)

[4.2 ประเภทของกราฟ 43](#_Toc207458057)

[4.3 การแทนข้อมูลกราฟ 48](#_Toc207458058)

[4.4 การแทนข้อมูลกราฟด้วยลิงค์ลิสต์ 49](#_Toc207458059)

[4.5 การคำนวณจำนวนเส้นทางระหว่างโหนด 50](#_Toc207458060)

[4.6 การท่องไปในกราฟ 50](#_Toc207458061)

[4.7 ประสิทธิภาพเชิงเวลา (Time Complexity) 52](#_Toc207458062)

[บทที่ 5 โครงสร้างต้นไม้ 55](#_Toc207458063)

[5.1 ความหมายของโครงสร้างต้นไม้ 55](#_Toc207458064)

[5.2 องค์ประกอบของต้นไม้ 55](#_Toc207458065)

[5.3 ประเภทของต้นไม้ 56](#_Toc207458066)

[5.4 Tree Traversal (การเดินต้นไม้) 57](#_Toc207458067)

[5.5 Binary Search Tree (BST) 61](#_Toc207458068)

[5.6 การวิเคราะห์ต้นไม้ด้วย Big-O 62](#_Toc207458069)

[5.7 โค้ดโครงสร้างต้นไม้ 63](#_Toc207458070)

# อาร์เรย์และลิงค์ลิสต์

## ความหมายของอาร์เรย์และลิงค์ลิสต์ (Array and Link list)

อาร์เรย์และลิงค์ลิสต์เป็นโครงสร้างข้อมูลเชิงเส้นที่มีการจัดเรียงข้อมูลแบบต่อเนื่องและเข้าถึง

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

## ความหมายอาร์เรย์

อาร์เรย์ หมายถึง การรวมกลุ่มของตัวแปรที่สามารถใช้ตัวแปรชื่อเดียวกันแทนข้อมูลสมาชิกได้หลาย ๆ ตัวในคราวเดียวกัน ด้วยการใช้เลขดรรชนี (Index) หรือซับสคริปต์ (Subscript) เป็นตัวอ้างอิงตําแหน่งสมาชิกบนแถว ลําดับนั้น ๆ

## โครงสร้างของอาร์เรย์ (Array Structure)

โครงสร้างอาร์เรย์ จัดในรูปแบบของแถวและคอลัมน์ ซึ่งมีทั้ง 1 มิติ 2 มิติ และ 3 มิติ โดยใช้ ตัวเลขเป็นตัวกํากับแถวและคอลัมน์

ลักษณะโครงสร้างอาร์เรย์

## ลักษณะโครงสร้างอาร์เรย์

### เป็นโครงสร้างเชิงเส้น (Linear Structure) คือ มีรูปแบบของการจัดเก็บข้อมูล เป็นแนวเส้นตรงที่ต่อเนื่องกัน

### จัดเก็บข้อมูลแบบเรียงลําดับ (Order) คือ นําข้อมูลที่ถูกนําเข้ามาจัดเก็บที่เป็น หมวดหมู่เดียวกัน หากบันทึกจะจัดเก็บเรียงแบบอันดับ เช่น เรียงลําดับของวันใน 1 สัปดาห์ Monday, Tuesday, Wednesday, ..., Sunday

### ข้อมูลภายในช่องจัดเก็บเป็นข้อมูลชนิดเดียวกัน (Homogenous Data Type) คือ หากใน 1 อาร์เรย์ มีช่องตาราง 5 ช่อง ทั้ง 5 ช่อง ก็จะต้องมีการบันทึกเป็นตัวเลขหรือ ตัวอักษรอย่างใดอย่างหนึ่งเท่านั้น

### มีการกําหนดช่องตารางที่มีโครงสร้างตายตัว (Static Structure) คือ ช่อง ตารางที่ใช้ในการจัดเก็บข้อมูลทุกช่องต้องมีขนาดที่เท่ากัน มีการจัดเรียงในรูปแบบเชิงเส้น และกําหนดจํานวนของช่องไว้อย่างตายตัว

### การเข้าถึงข้อมูล สามารถเข้าถึงข้อมูลได้แบบสุ่ม (Random Access) คือ สามารถที่จะเข้าถึงข้อมูลที่จัดเก็บในช่องตารางใดก็ได้

## ชนิดของอาร์เรย์

### อาร์เรย์ 1 มิติ อาร์เรย์ 1 มิติ เป็นอาร์เรย์ที่มีแถวเพียง 1 แถว แต่จะมีคอลัมน์หลายคอลัมน์ ซึ่งสามารถเขียน ตามลักษณะตารางได้ 2 แบบ คือ อาร์เรย์แนวนอนและอาร์เรย์แนวตั้ง

รูปแบบอาร์เรย์ 1 มิติ A[l:u] ; A คือ ชื่อของอาร์เรย์

{ คือ ค่าต่ำสุด

u คือ ค่าสูงสุด

โดยกําหนด ; ชื่อของอาร์เรย์ : เป็นการกําหนดชื่อให้กับอาร์เรย์ เพื่อความสะดวกการเรียกใช้งาน

ค่าต่ำสุด(Lower bound) : เป็นการระบุเป็นตําแหน่งเริ่มต้นของการจัดเก็บข้อมูล

ค่าสูงสุด (Upper bound) : เป็นการกําหนดตําแหน่งสูงสุดของการจัดเก็บข้อมูล

เช่น ; A [-5 : 3]

ชื่ออาร์เรย์ คือ A ค่าต่ำสุด คือ -5

ค่าสูงสุด คือ 3

ในการกําหนดค่าของข้อมูลอาร์เรย์จะต้องระบุจํานวนที่แน่นอน เพื่อจะกําหนดจํานวนช่อง

ภายในชุดอาร์เรย์ได้ และชนิดของข้อมูลจะต้องระบุเป็นชนิดเดียวกัน โดยรูปแบบของการกําหนดค่า จะต้องระบุตําแหน่งของอาร์เรย์ ดังนี้

รูปแบบ A [l:u] = N ; N คือ ค่าของข้อมูล

รูปแบบอาร์เรย์ที่เป็นการระบุรายละเอียด A[5] = 10

รูปแบบอาร์เรย์ที่เป็นช่องตาราง A 5 10 15 20 25

1 2 3 4 5

การกําหนดค่าตัวแปร A ให้กับอาร์เรย์ที่มีการจองพื้นที่ของจํานวนช่อง 5 ช่อง สําหรับข้อมูล ชนิด integer (เลขจํานวนเต็ม) จํานวน 5 ค่า ซึ่งจะทําให้เห็นความสัมพันธ์ของตําแหน่งของอาร์เรย์ และค่าข้อมูลว่ามีการจัดความสัมพันธ์แบบ 1 ต่อ 1 คือ ใน 1 ตําแหน่งมีข้อมูล 1 ค่า หรือ 1 ค่าข้อมูล ถูกจัดเก็บอยู่ใน 1 ตําแหน่ง

การหาจํานวนช่องของตารางของอาร์เรย์

หากต้องการที่จะกําหนดขนาดของอาร์เรย์ที่ต้องการใช้งาน เพื่อการเก็บข้อมูลมีสูตรดังนี้

จํานวนช่องตาราง = (u-l) + 1 ; เมื่อ u = ค่าสูงสุด

1 = ค่าต่ำสุด

ตำแหน่งบนพื้นที่หน่วยความจำ

แสดงตารางแทนค่าพื้นที่บนหน่วยความจํา (Address) เทียบกับ ตําแหน่ง อาร์เรย์ (index) ที่กําหนดค่าไว้คือ NEW[-3], NEW[-2], NEW[-1], NEW[0], NEW[1], NEW[2], NEW[3], NEW[4], NEW[5], NEW[6] และพื้นที่บนหน่วยความจําเริ่มจัดเก็บจากตําแหน่งที่ 200 ความกว้าง 10 ไบต์ ตําแหน่งต่อไปคือ 200, 210, 220, 230, 240, 250, 260, 270, 280, 290 ตามลําดับ ดังนั้นหากต้องการบันทึกข้อมูลลงในอาร์เรย์ช่องที่ 6 บนหน่วยความจําต้องบันทึกลงแอดเดรส 290

### อาร์เรย์ 2 มิติ เป็นอาร์เรย์ที่มีแถวหลายแถวและคอลัมน์หลายคอลัมน์ ซึ่งถูกกํากับด้วย ตําแหน่งของแถวและคอลัมน์ อาร์เรย์ A(M, N) สามารถที่จะกําหนดแสดงได้ในรูปแบบ

### ของตารางดังรูปแบบอาร์เรย์ 2 มิติ A[l1:u1 , l2 :u2 ] ; เมื่อ A คือ ชื่อของอาร์เรย์

l1 คือ ค่าตํ่าสุดของมิติที่ 1

u1 คือ ค่าสูงสุดของมิติที่ 1

l2 คือ ค่าต่ำสุดของมิติที่ 2

u2 คือ ค่าสูงสุดของมิติที่ 2

โดยกําหนด ; ชื่อของอาร์เรย์ : เป็นการกําหนดชื่อให้กับอาร์เรย์ เพื่อความสะดวกการเรียกใช้งาน

ค่าต่ำสุด (Lower bound) : เป็นการระบุเป็นตําแหน่งเริ่มต้นของการจัดเก็บข้อมูล มิติที่ 1 และมิติที่ 2

ค่าสูงสุด (Upper bound) : เป็นการกําหนดตําแหน่งสูงสุดของการจัดเก็บข้อมูล มิติที่ 1 และมิติที่ 2

มิติที่ 1 คือ อาร์เรย์ A ที่จัดเก็บตําแหน่งตั้งแต่ 1,..,M ค่าต่ำสุดคือ 1 ค่าสูงสุด คือ M

มิติที่ 2 คือ อาร์เรย์ A ที่จัดเก็บตําแหน่งตั้งแต่ 1,,N ค่าต่ำสุดคือ 1 ค่าสูงสุดคือ N

เช่น ; A[1:3,1:3] ชื่ออาร์เรย์ คือ A

มิติที่ 1 ค่าต่ำสุด คือ 1 ค่าสูงสุด คือ 3

มิติที่ 2 ค่าต่ำสุด คือ 1 ค่าสูงสุด คือ 3

การคํานวณหาจํานวนสมาชิก

การหาจํานวนสมาชิกในอาร์เรย์ 2 มิติ นั้น จะต้องการทราบจํานวนช่องของการ จัดเก็บข้อมูล จึงจะสามารถหาจํานวนสมาชิกของอาร์เรย์ จากการคูณกันของจํานวนของอาร์เรย์ในมิติที่ 1 และ 2

สูตรที่ 1 จํานวนช่องตาราง = (u-l) + 1 ; เมื่อ u = ค่าสูงสุด

l = ค่าต่ำสุด

สูตรที่ 2 จํานวนสมาชิก = M × N

การคํานวณหาตําแหน่ง (Address) ของ A(i,j)

อาร์เรย์ 2 มิติ จัดเรียงข้อมูลตามรูปแบบของแถวและคอลัมน์ โดยการจัดเรียงตาม

รูปแบบของแถว (Row major) คือ การเริ่มอ่านค่าข้อมูลของตําแหน่งแถวก่อนแล้วตามด้วยคอลัมน์ซึ่ง อ่านในแนวนอน ดังภาพที่ 2.8 ส่วนการจัดเรียงรูปแบบของคอลัมน์ (Column major) คือ การเริ่ม อ่านค่าของตําแหน่งคอลัมน์ก่อนแล้วตามด้วยแถวซึ่งอ่านค่าในแนวตั้ง

การคํานวณหาต่ำแหน่ง addr[i,j] = b+L(u2-l2+1)(i-j1)+L(j-l2) ;

เมื่อ b คือ ตําแหน่งแรกที่บันทึกข้อมูล

L คือ ความกว้างของชนิดข้อมูล

i คือ ตําแหน่งที่เป็นตัวชี้ของมิติที่ 1

j คือ ตําแหน่งที่เป็นตัวชี้ของมิติที่ 2

u2 คือ ค่าสูงสุดของมิติที่ 2

l2 คือ ค่าต่ำสุดของมิติที่ 2

l1 คือ ค่าต่ำสุดของมิติที่ 1

## ความหมายลิงค์ลิสต์

นักวิชาการได้ให้ความหมายของคําว่า “ลิงค์ลิสต์” ไว้หลายความหมาย ดังนี้

ณัฐพงษ์ วารีประเสริฐ และสุธี พงศาสกุลชัย (2552 : 56) ได้ให้ความหมายของลิงค์ลิสต์ไว้ว่า ลิงค์ลิสต์ หมายถึง โครงสร้างที่จัดเก็บข้อมูลด้วยโหนดต่อเนื่องกัน โดยแต่ละโหนดจะถูกระบุตําแหน่ง บนหน่วยความจํา เพื่อใช้อ้างอิงหรือเข้าถึงได้ ภายในโหนดประกอบด้วย 2 ส่วน คือ ส่วนที่จัดเก็บ ข้อมูล และส่วนที่จัดเก็บพอยน์เตอร์

กล่าวโดยสรุปได้ว่า ลิงค์ลิสต์ หมายถึง ข้อมูลที่ถูกนํามาจัดเก็บเรียงกันในลักษณะเส้นตรง โดยข้อมูลจะถูกจัดเรียงเป็นชุด ๆ เรียกว่า โหนด (Node) ซึ่งแต่ละโหนดจะประกอบด้วยข้อมูล 2 ส่วน คือ ส่วนของข้อมูล (data) และส่วนลิงค์ข้อมูล (Link)

## โครงสร้างของลิงค์ลิสต์

การเชื่อมโยงไปยังโหนดอื่น ๆ ของลิงค์ลิสต์จะใช้รูปแบบของพอยน์เตอร์เป็นตัวชี้ไปยังโหนด ต่อ ๆ ไป ดังนั้นลิงค์ลิสต์จึงมีรูปแบบของการจัดเก็บข้อมูลที่ประกอบไปด้วยโหนดต่าง ๆ ที่จัดเก็บ ข้อมูลอยู่และทําการเชื่อมโยงโหนดเหล่านี้ด้วยพอยน์เตอร์ ทําให้มีลักษณะโครงสร้าง

โหนดของลิงค์ลิสต์มีส่วนที่เป็นข้อมูลและการเชื่อมโยง โดยส่วนของการ เชื่อมโยงที่พอยน์เตอร์ในการกําหนดชี้ไปยังโหนดเริ่มต้น ปกติจะเรียกพอยน์เตอร์นี้ว่าHead และในตําแหน่งสุดท้ายของโหนดที่ไม่มีการเชื่อมโยงไปยังโหนดอื่นก็จะมีการประกาศค่าเป็นลิสต์ว่าง (Emptylink list)

ลิงค์ลิสต์ถือเป็นโครงสร้างที่มีการทํางานทั้งการเก็บข้อมูลและการเชื่อมโยงไปยังตําแหน่ง ต่อไปมีลักษณะสําคัญ ดังนี้

### โครงสร้างข้อมูลเป็นรูปแบบเชิงเส้น (Linear Structure) มีลักษณะของการ เริ่มต้นค่าข้อมูลหน่วยแรกเป็นหน่วยเริ่มต้น จากนั้นมีหน่วยต่อไปจะเห็นได้ว่าลิงค์ลิสต์เริ่มชี้ตําแหน่งไปยังโหนดแรกและมีการชี้ไปยังตําแหน่งต่อไปจนถึงตําแหน่งโหนดสุดท้าย

### โครงสร้างข้อมูลไม่ตายตัว (Dynamic Structure) ลักษณะของลิงค์ลิสต์เมื่อ ทําการประมวลผลสามารถที่จะปรับเปลี่ยนโครงสร้างได้โดยไม่ต้องกําหนดจองพื้นที่หน่วยความจําไว้ล่วงหน้า สามารถที่จะทําการเพิ่มโหนดหรือลบโหนดได้ทันที

### โครงสร้างข้อมูลไม่เป็นลําดับ (Non-Order Structure) โดยลักษณะของ โครงสร้างนั้นไม่จําเป็นต้องลําดับการจัดเก็บว่าจาก 1 ต้องเป็น 2 หรือเพิ่มค่าตามกําหนด แต่สามารถ ที่จะจัดโครงสร้างในลําดับใดก่อนก็ได้ตามที่หน่วยความจําจัดให้

## โครงสร้างลิงค์ลิสต์แบบทิศทางเดียว (Single Linked List)

โครงสร้างลิงค์ลิสต์แบบทิศทางเดียวลิงค์ลิสต์แบบทิศทางเดียวเป็นลิงค์ลิสต์ที่มีการเชื่อมโยงด้วยพอยน์เตอร์ไปในทิศทางเดียวกัน

คือ ชี้ไปยังโหนดที่อยู่ถัดไปในโครงสร้าง ซึ่งลิงค์ลิสต์ในลักษณะนี้ถือเป็นรูปแบบพื้นฐานของลิงค์ลิสต์ แบบอื่น ๆ คือประกอบไปด้วยโหนดที่จัดเก็บข้อมูลและมีการเชื่อมโยง เนื่องจากลิงค์ลิสต์เป็นโครงสร้างแบบไดนามิกคือไม่ตายตัวสามารถที่จะปรับโครงสร้างได้ ตลอดการประมวลผล ดังนั้นจึงมีการกําหนดโครงสร้างของลิงค์ลิสต์ให้มีส่วนประกอบสําคัญ 3 ประการคือ

### โครงสร้างโหนดต้นลิสต์ (Head Node Structure) การที่โครงสร้างของลิงค์ ลิสต์มีโครงสร้างที่เป็นแบบไดนามิก ทําให้เกิดความไม่แน่นอนเรื่องของจํานวนโหนดว่าขณะปัจจุบัน เมื่อทําการประมวลผลจํานวนโหนดถูกลบหรือเพิ่ม ซึ่งเป็นลักษณะที่เรียกว่า Meta data ถึงแม้ว่าสามารถตรวจสอบได้ด้วยการใช้พอยน์เตอร์เข้าไปนับจํานวนแต่ละโหนดได้แต่จะเป็นการเสียเวลาเพราะจะต้องเริ่มตั้งแต่โหนดแรก ดังนั้นจึงมีการกําหนดโครงสร้างของโหนดแรกของโครงสร้างซึ่งจะ เรียกว่า โหนดต้นลิสต์ (Head Node) มาใช้ในการเก็บข้อมูลเกี่ยวกับจํานวนของโหนดในโครงสร้าง หรือข้อมูลสําคัญอื่น ๆ Count: การนับจํานวนโหนดในลิสต์ เป็นส่วนประกอบของโหนดต้นลิสต์ที่ทําการจัดเก็บจํานวนโหนดภายในลิสต์ไม่รวมโหนดต้นลิสต์ทําให้สะดวกและรวดเร็วเมื่อต้องการทราบจํานวนของโหนดก็อ่านค่าได้จาก Count และจากภาพที่ แสดงให้ทราบว่ามีจํานวนโหนดอยู่ในลิสต์ 4 โหนดด้วยกัน

### โครงสร้างโหนดข้อมูล (Data Node Structure) สําหรับโครงสร้างโหนด ข้อมูลของลิงค์ลิสต์นั้นโดยรูปแบบทั่วไปจะประกอบไปด้วย 2 ส่วน หลักหรือ 2 ฟิลด์หลักคือ ฟิลด์แรกเก็บในส่วนของข้อมูล และฟิลด์ที่ 2 เก็บค่าของการเชื่อมโยงในส่วนของการเชื่อมโยงก็จะใช้พอยน์เตอร์เป็นตัวเชื่อมโยงไปยังโหนดอื่น ๆ

### ทิศทางของพอยน์เตอร์ (Pointer Link) สําหรับทิศทางพอยน์เตอร์ของลิงค์ลิสต์แบบทิศทางเดียว จะทําการชี้ไปยังทิศทางเดียวกันเสมอคือไปยังโหนดต่อไปในสมาชิก เมื่อไม่มี สมาชิกใดหรือโหนดใดอีก พอยน์เตอร์ก็จะประกาศเป็นลิสต์ว่าง (Empty List)

## การทํางานของลิงค์ลิสต์แบบทิศทางเดียว

### การสร้างลิสต์ (Create List)ก่อนที่จะมีการสร้างลิสต์จะยังไม่มีการชี้ไปที่โหนดใด แต่เมื่อมีการสร้างลิสต์ขึ้นมาเพื่อที่จะนําไปใช้งาน จะต้องมีการประกาศค่าให้ชี้ไปยังโหนดเริ่มต้น ทําให้ลิสต์ที่เป็นลิสต์ของพอยน์ เตอร์เริ่มต้นที่ไม่มีการชี้ค่า null และมีจํานวนสมาชิกเป็นศูนย์

### การเพิ่มโหนด (Add Node)การเพิ่มโหนด สามารถเพิ่มโหนดต่อ ๆ กันไปได้ โดยใช้ลิ้งค์เป็นตัวเชื่อมระหว่างโหนด และกําหนดค่าสุดท้ายที่ไม่มีการชี้เป็นค่า nil ซึ่งจํานวนสมาชิกจะมีค่าเท่ากับจํานวนโหนด ทั้งหมด

### การลบโหนด (Delete node)การลบโหนด สามารถลบทั้งโหนดและข้อมูลที่ไม่ต้องการออกไป ซึ่งมีหลักการง่าย ๆด้วยการเปลี่ยนลิ้งค์ของโหนดก่อนหน้าที่ต้องการลบให้ชี้ไปยังลิ้งค์ของโหนดที่ต้องการลบ

### การแทรกโหนด (Insert Node)เมื่อมีโหนดมาเพิ่มอีกก็จะต้องเพิ่มในลักษณะที่เป็นการแทรกเข้าไประหว่างโหนดเดิมและโหนดต้นลิสต์ การแทรกโหนดมี 4 รูปแบบดังนี้

#### การแทรกโหนดเมื่อลิสต์ว่าง (Insert Into Empty List) ก่อนที่จะมีการแทรกโหนดจะมีการสร้างลิสต์ขึ้นมากําหนดพอยน์เตอร์ให้ชี้ไปยังโหนดต้นลิสต์ หากไม่มีการชี้ต่อไปที่โหนดใด แสดงว่าเป็นลิสต์ที่ว่าเป็นลิสต์ที่ว่างและเมื่อมีการ เพิ่มโหนดขึ้นมาใหม่ ก็จะมีการจองพื้นที่ให้กับโหนดพร้อมกับค่าข้อมูล เมื่อต้องการลิงค์ของโหนดก่อนหน้า มายังโหนดที่ทําการแทรก

#### การแทรกโหนดตําแหน่งก่อนหน้า (Insert at Beginning) สําหรับการแทรกโหนดรูปแบบนี้กระทําได้ก็เมื่อมีโหนดอื่น ๆ อยู่ในโครงสร้างแล้ว หากมีการเพิ่มโหนดขึ้นมาใหม่ก็จะแทรกไว้ก่อนหน้าโหนดอื่น ๆ เสมอ

#### การแทรกโหนดตําแหน่งกลาง (Insert in Middle) เมื่อมีการแทรกโหนดเข้าไปหลาย ๆ ตัว หากต้องการที่จะแทรกเข้าไปยัง ตําาแหน่งกลางก็สามารถทําได้ตามหลักการคือ ชี้ลิงค์ของโหนดใหม่ไปยังลิงค์ที่โหนดตําแหน่งก่อนหน้าชี้ไป จากนั้นเซตลิงค์ของตําแหน่งโหนดก่อนหน้านั้นชี้มาที่โหนดใหม่

#### การแทรกโหนดตําแหน่งท้าย (Insert at End) สําหรับการแทรกโหนดไปยังตําาแหน่งท้ายจะมีการเซตลิ้งค์คล้ายกับการแทรกโหนดที่ตําาแหน่งกลางเพียงแต่จะทําการตรวจสอบว่าตําแหน่งก่อนการแทรกมีค่าของลิ้งค์เป็น nil หรือไม่ หากใช่ก็ทําการแทรกไปยังตําแหน่งนั้นและกําหนดการลิ้งค์ไปยังโหนดใหม่

ตัวอย่างโค้ด **LINKED LIST (สร้างเองแบบคลาส)**

Node + Singly Linked List

from IPython.display import display, Markdown

class Node:

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

self.data = data

self.next = None

class SinglyLinkedList:

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

self.head = None

def append(self, data):

new\_node = Node(data)

if not self.head: # ถ้า list ว่าง

self.head = new\_node

else:

current = self.head

while current.next: # ไปหาตัวสุดท้าย

current = current.next

current.next = new\_node

def \_\_str\_\_(self):

"""ทำให้ print(list) แสดงผลเป็นลูกศร"""

current = self.head

result = ""

while current:

result += str(current.data) + " → "

current = current.next

result += "None"

return result

def display(self):

display(Markdown("\*\*Linked List:\*\* " + str(self)))

def delete(self, key):

current = self.head

if current and current.data == key:

self.head = current.next

return

prev = None

while current and current.data != key:

prev = current

current = current.next

if current is None:

return

prev.next = current.next

**# ทดลองใช้งาน**

linked\_list = SinglyLinkedList()

linked\_list.append(10)

linked\_list.append(20)

linked\_list.append(30)

linked\_list.display()

linked\_list.delete(20)

linked\_list.display()

**ตัวอย่างโค้ด Doubly Linked List**

from IPython.display import display, Markdown

class DNode:

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

self.data = data

self.prev = None

self.next = None

class DoublyLinkedList:

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

self.head = None

def append(self, data):

new\_node = DNode(data)

if not self.head:

self.head = new\_node

return

current = self.head

while current.next:

current = current.next

current.next = new\_node

new\_node.prev = current

def \_\_str\_\_(self):

"""เวลาพิมพ์ object จะเห็นเป็นลูกศร"""

current = self.head

result = ""

while current:

result += f"{current.data} "

current = current.next

result += "None"

return result

def display\_forward(self):

display(Markdown("\*\*Forward:\*\* " + str(self)))

def display\_backward(self):

current = self.head

if not current:

display(Markdown("\*\*Backward:\*\* None"))

return

# ไปหาตัวสุดท้ายก่อน

while current.next:

current = current.next

result = ""

while current:

result += f"{current.data} "

current = current.prev

result += "None"

display(Markdown("\*\*Backward:\*\* " + result))

# ------------------------

# ทดลองใช้งาน

dll = DoublyLinkedList()

dll.append(10)

dll.append(20)

dll.append(30)

dll.display\_forward()

dll.display\_backward()

**ตัวอย่างโค้ด การสร้าง Array**

arr = [10, 20, 30, 40, 50]

print("เริ่มต้น:", arr)

# [10, 20, 30, 40, 50]

**ตัวอย่างโค้ด เพิ่มสมาชิก**

arr.append(60)

print("เพิ่ม 60 (append):", arr)

arr.insert(2, 25) # แทรกตำแหน่งที่ 2

print("แทรก 25 ที่ index=2:", arr)

**ตัวอย่างโค้ด ลบสมาชิก**

arr.remove(30) # ลบค่าที่เป็น 30

print("ลบค่า 30:", arr)

arr.pop() # ลบตัวสุดท้าย

print("ลบตัวท้าย (pop):", arr)

arr.pop(1) # ลบ index 1

print("ลบ index=1:", arr)

**ตัวอย่างโค้ด ค้นหา / นับ / ความยาว**

print("ตำแหน่งของ 40:", arr.index(40)) # หาตำแหน่งของ 40

print("จำนวนครั้งที่มี 10:", arr.count(10)) # นับจำนวน 10

print("ความยาว array:", len(arr)) # ความยาว array

**ตัวอย่างโค้ด ลูป array**

print("ลูปแสดงสมาชิก:")

for item in arr:

print(item)

**ตัวอย่างโค้ด เรียงลำดับ**

arr.sort() # เรียงน้อยไปมาก

print("เรียงจากน้อยไปมาก:", arr)

arr.sort(reverse=True) # มากไปน้อย

print("เรียงจากมากไปน้อย:", arr)

# โครงสร้างคิว

## ความหมายของโครงสร้างคิว

คิวเป็นโครงสร้างข้อมูลแบบเชิงเส้น ที่มีลักษณะเฉพาะคือการนําเข้าข้อมูล (Enqueue) ทางด้านส่วนท้ายของคิว และการนําข้อมูลออก (Dequeue) ทางด้านส่วนหัวของคิว โดยข้อมูลที่เข้า มาก่อนจะต้องออกก่อน ส่วนข้อมูลที่เข้ามาทีหลังต้องออกหลัง ซึ่งเป็นรูปแบบคล้ายกับการเข้าคิว เรียกว่า เข้าก่อนออกก่อน (First in First out)

## โครงสร้างคิว

คิวเป็นโครงสร้างข้อมูลที่มีลักษณะของการเข้าและออกของข้อมูลอย่างเป็นลําดับ ข้อมูลใด เข้ามาก่อนก็จะดําเนินการก่อน หากข้อมูลใดเข้ามาทีหลังก็จะดําเนินการทีหลัง เรียกว่า เข้าก่อนออก ก่อน หรือ First in first out (FIFO) คือ เมื่อต้องการนําเข้าข้อมูลจะต้องนําข้อมูลไปต่อส่วนท้ายของ คิว เรียกว่า rear และถ้าต้องการนําข้อมูลออก จะต้องนําออกทางส่วนหัวของคิว ซึ่งเรียกว่า front

โครงสร้างคิวมีลักษณะดังนี้

### โครงสร้างข้อมูลเป็นรูปแบบเชิงเส้น (Linear structure) คิวจะมีการจัดข้อมูล ที่นําเข้ามาต่อเนื่องกันไปโดยข้อมูลใดเข้ามาก่อนก็จะอยู่อันดับแรก และข้อมูลที่ตามเข้ามาทีหลังก็จะเรียงกันไปChild Node: โหนดลูก คือโหนดที่อยู่ถัดจากโหนดแม่

### โครงสร้างเป็นแบบไม่ตายตัว (Dynamic structure) เนื่องจากคิวสามารถที่ จะดําเนินการทั้งการเพิ่มและลดข้อมูลได้ทําให้ขนาดของข้อมูลนั้นมีการเปลี่ยนแปลงตลอดเวลา Leaf Node: โหนดปลายทาง (ไม่มีลูก)

### สามารถนําเข้าและดึงข้อมูลออกได้ (Enqueue and Dequeue) โดยการ นําเข้าข้อมูลและการนําข้อมูลออกของคิวนั้นสามารถเพิ่มข้อมูลได้ในส่วนของ rear และดึงข้อมูลออกจากโครงสร้างในด้าน front Subtree: ต้นไม้ย่อยของต้นไม้ใหญ่

### ทํางานแบบเข้าก่อนออกก่อน (First in First out) โดยการดําเนินการกับ ข้อมูลนั้น หากการนําเข้าข้อมูลใดก่อนก็จะดําเนินการกับข้อมูลนั้นก่อนเรียงต่อเนื่องกันไปจนกว่าจะหมดรายการLevel: ระดับความลึกของโหนด (Root อยู่ระดับ 0)

## การดำเนินการของคิว

การดําเนินการพื้นฐานโครงสร้างข้อมูลแบบคิว คือ มีการนําเข้าข้อมูลด้วยการต่อท้ายและดึง ข้อมูลออกในส่วนหัว สามารถแบ่งออกเป็น 2 กรณี คือ

### การนําเข้าข้อมูล (Enqueue)

หากมีการนําเข้าข้อมูลแรกเข้าสู่คิวแล้วข้อมูลนี้จะเป็นข้อมูลอันดับแรก และเมื่อมีการเพิ่ม ข้อมูลเข้ามาอีกข้อมูลใหม่เข้ามา จะต้องมีการตรวจสอบก่อนว่าคิวเต็ม (Overflow) หรือไม่ ถ้าไม่เต็ม จึงจะสามารถเพิ่มข้อมูลลงในคิวได้ โดยจะต่อท้ายในส่วนของ rear ซึ่งก็คือ rear =rear + 1

### การนําข้อมูลออก (Dequeue)

ก่อนการนําข้อมูลที่ตําแหน่งส่วนหัวออกจากคิว จะต้องมีการตรวจสอบว่า คิวว่าง (Underflow) หรือไม่ ถ้าไม่ว่างจึงสามารถดําเนินการนําข้อมูลออกจากคิวได้ โดยกระทําเฉพาะส่วน หัวหรือ front ของโครงสร้างเท่านั้น ซึ่งก็คือ front = front +1

## การแทนคิวด้วยอาร์เรย์

การแทนโครงสร้างคิวด้วยอาร์เรย์ จะต้องกําหนดจํานวนของการจองพื้นที่บนหน่วยความจําและใช้อาร์เรย์ในการนําข้อมูลเข้าด้านท้ายและนําข้อมูลออกในส่วนหัว

### โครงสร้างของการแทนคิวด้วยอาร์เรย์

การแทนคิ้วด้วยอาร์เรย์จะต้องมีการระบุจํานวนสูงสุดของพื้นที่เก็บข้อมูล (Maxsize) พร้อม กับกําหนดพอยน์เตอร์ขึ้นมาอีก 2 ตัว คือ front กับ rear เพื่อใช้ในการชี้ค่าข้อมูลที่อยู่ส่วนหัวและส่วนท้าย

### การดำเนินการแทนคิวด้วยอาร์เรย์

#### เพิ่มข้อมูลลงในคิว (Enqueue) โดยข้อมูลจะเพิ่มเข้ามาทางด้าน rear ของคิว- insert (q,A); เขียนแทนความหมายของการเพิ่มข้อมูล A ผ่านทาง rear เข้ามายังคิวชื่อ q ตัวอย่างการเดิน Inorder:

#### ดึงข้อมูลออกจากคิว (Dequeue) ข้อมูลจะถูกดึงออกจากคิวผ่านทาง front ถ้าข้อมูลที่อยู่ภายในคิวมีเพียงตัวเดียว ค่า front จะมีค่าเท่ากับ rear (อยู่ในตำแหน่งเดียวกัน) - remove (q); เขียนแทนความหมายของการดึงข้อมูล A ออกจากคิว 3 ทางด้าน front

#### ตรวจสอบว่ามีข้อมูลอยู่ภายในคิวหรือไม่ (Empty หรือ Underflow) จะไม่สามารถดึงข้อมูลออกจากคิวว่าง หรือคิวที่ไม่มีข้อมูลใดได้ ดังนั้นต้องมีการตรวจสอบคิวว่าง คือempty (q); ถ้าคิวชื่อ q ว่างไม่มีข้อมูลอยู่ภายในนั้นจะได้ค่าออกมาเป็น True แต่ถ้าภายในคิวมีข้อมูลอยู่จะได้ค่าออกมาเป็น False

#### ตรวจสอบว่ามีข้อมูลอยู่ภายในคิวเต็มหรือไม่ (Full หรือ Overflow) การดำเนินการนี้มักตรวจสอบเมื่อต้องการเพิ่มข้อมูลลงไปในคิว และต้องมีการตรวจสอบก่อนว่าคิวเต็มหรือไม่ มีเนื้อที่ที่จะใส่ข้อมูลลงไปได้หรือไม่ เสร็จแล้วจึงใส่ข้อมูลลงไปได้

### การดำเนินการแทนคิวด้วยวงกลม(Circular Queue)

การใช้คิววงกลมสามารถแก้ปัญหาการมีพื้นที่ว่างในส่วนหัวของคิว ซึ่งทำให้สามารถ Enqueue ข้อมูลเข้ามายังคิวได้จนกว่าคิวเต็มจริง ๆ

## การดำเนินงานของแถวคอย (Queue Operation)

ประกอบด้วยการทำงาน 2 แบบคือ การนำข้อมูล เข้าทางด้านท้ายแถว เรียกว่า Enqueue และการนำข้อมูลออกจากหัวแถว เรียกว่า Dequeue ซึ่งจะต้องใช้ Pointer จำนวน 2 ตัวคือ Front ทำหน้าที่กำหนดตำแหน่งของหัวแถว และ Rear ทำหน้าที่กำหนดตำแหน่ง ของท้ายแถว [2-5]

### ฟังก์ชันการดำเนินงานพื้นฐานของแถวคอย (Queue)

#### Enqueue เป็นการนำข้อมูลเพิ่มเข้าไปในคิว โดยสมาชิกใหม่จะเพิ่มต่อจากสมาชิกที่อยู่ท้ายสุดคือตำแหน่ง Rear และมีการขยับเพิ่มตำแหน่งของ

#### Dequeue เป็นการลบข้อมูลออกจากคิว โดยข้อมูลที่จะถูกนำออกคือข้อมูลที่อยู่ตำแหน่งหน้าสุดคือข้อมูล ณ ตำแหน่งของ Front หากในคิวไม่มีข้อมูลจะเรียกว่า Queue Underflow

## โครงสร้างข้อมูลประเภทของ Queue)

### ประเภทของ Queue แบ่งตามลักษณะของการจัดเก็บข้อมูลใต้ 3 ประเภทคือ

#### คิวเชิงเส้น (Linear Queue) หมายถึง Queue ทำการจัดเก็บข้อมูลจากด้านหน้าสุดถึงหลังสุด เรียงเป็นเส้นตรงเท่านั้น

#### คิววงกลม (Circular Queue) หมายถึง Queue ทำการจัดเก็บข้อมูลเป็นเส้นวงกลม เมื่อข้อมูลด้านหลังสุดเต็ม จะสามารถจัดเก็บข้อมูลย้อนเข้าไปด้านหน้าได้

#### อภิสิทธิ์คิว (Priority Queue) หมายถึง Queue ทำการจัดลำดับการให้บริการข้อมูลแบบตามความสำคัญของข้อมูล

##### **Linear Queue**ในการพัฒนาQueueตามโปรแกรมด้านบนจะเรียกว่า Linear Queueซึ่งจะมีข้อเสียที่ความไม่เต็มประสิทธิภาพ

##### **Circular Queue**จากปัญหาของความสิ้นเปลืองของ Linear Queue จึงได้มีการพัฒนาศิวประเภทวงกลมขึ้นเพื่อให้สามารถย้อนกลับไปใช้ในของว่างด้านหน้า

##### **Priority Queue**นอกจากนี้ยังมีคิวอีกประเภทหนึ่งที่มีความสำคัญและปรากฏใช้ในที่ต่าง ๆ คือ Priority Queue หรือคิวอภิสิทธิ์ โดยคิวประเภทนี้จะต้องมีระดับความสำคัญของคิวมาด้วย เช่นคิวที่ใช้ในโรงพยาบาล เมื่อคนไข้มาถึงโรงพยาบาลจะต้องมีการจัดระดับความสำคัญตามความรุนแรงของอาการ หากมีอาการที่มีความร้ายแรงจะต้องได้ลำดับความสำคัญและได้รับการให้บริการก่อน

## การประยุกต์ใช้ Queue

ในการประยุกต์ใช้คิวสามารถพบได้ในทั้งในชีวิตประจำวันได้แก่ การไปเข้าคิวรับบริการต่าง ๆ

นอกจากนี้ในการทำงานของคอมพิวเตอร์ยังมีการนำคิวไปให้บริการในหลาย ๆ ที่ เช่น การให้บริการของเครื่องพิมพ์ การใช้ CircularQ ในการทำงานแบบ Round Robin Scheduting ของการทำงานของ CPU

### การประยุกต์ใช้คิวจำลองแบบ

#### **การจำลองแบบ (Simulation)** หมายถึง การใช้ระบบหนึ่งเพื่อเลียนแบบพฤติกรรมของอีกระบบหนึ่ง ใช้งานเมื่อการทดลองด้วยระบบจริงๆ มีค่าใช้จ่ายสูง หรือเสี่ยงต่ออันตรายการจ่าลองแบบของคอมพิวเตอร์ จะใช้ขั้นตอนการทำงานของโปรแกรม เพื่อการเลียนแบบพฤติกรรมของระบบที่เราต้องการศึกษาการจำลองแบบของระบบแบ่งกันใช้เวลาระบบคอมพิวเตอร์ที่มีการทํางานแบบแบ่งกันใช้เวลาShardเป็นระบบที่มีผู้ใช้เครื่องคอมพิวเตอร์พร้อมกันในเวลาเดียวกันโดยระบบมีหน่วยผลกลาง (ซีพียู) และหน่วยความจำหลักเพียงอย่างละ 1 เท่านั้น ผู้ใช้หลาย ๆ คนนี้ จะต้องมีการ ใช้หน่วยความจําหลักและหน่วยประมวลผลกลางร่วมกัน ซึ่ง อนุญาติให้ผู้ใช้แต่ละคนประมวลผล โปรแกรม (ใช้ทรัพยากร ของระบบ) ในเวลาหนึ่งๆ แล้วก็จะให้ผู้ใช้คนต่อไปใช้จนกว่า จะวนกลับมายังผู้ใช้คนแรกอีก วิธีการประมวลผลร่วมกัน ระหว่างผู้ใช้หลายๆ คน เราเรียกว่า ระบบแบ่งกันใช้เวลา (Time sharing) ซึ่งลักษณะการใช้ซีพียูจะเป็นไปตามลําดับ คือ “มาก่อนได้ก่อน” (first – come – first – serve) และมี ล่าดับการทํางานดังนี้เมื่อโปรแกรมขอใช้เวลาซีพียู โปรแกรมนั้นจะถูกนําไปต่อท้ายคิวประมวลผลโปรแกรมที่อยู่ต้นคิวจะถูกส่งไปทํางาน และยังคงอยู่ที่ตันคิวจนกว่าจะใช้ซีพียูเสร็จเมื่อรันโปรแกรมทํางานเสร็จตามเวลาการขอใช้ซีพียู ก็ จะถูกนําออกจากคิว และจะไม่ถูกนํากลับมาอีก จนกว่าจะมี การขอใช้ซีพียูครั้งใหม่ (จึงจะกลับไปที่ข้อ 1. อีก)

#### **การจำลองแบบสนามบิน** การเขียนโปรแกรมเพื่อจ่าลองแบบของสนามบิน จะใช้โครงสร้างข้อมูลแบบคิว แทนคิวขอเครื่องบินที่จะรอขึ้นหรือรอลง แต่คอนข้างเป็นโปรแกรมที่ซับซ้อน ดังสภาพความเป็นจริงที่ว่า สนามบินมีขนาดเล็กแต่มีเครื่องบินขึ้นลงจำนวนมากมีทางวิ่ง (runway) เพียงทางเดียว ดังนั้น ณ เวลาใด ๆเครื่องบิน จะต้องขึ้นหรือลงอย่างใดอย่างหนึ่งเท่านั้น และเพียงเครื่องเดียวด้วยในเวลาที่เครื่องบินซึ่งพร้อมจะขึ้นหรือลงมาถึงสนามบิน สนามบินนั้นอาจจะว่างหรือมีเครื่องบินอื่นกำลังขึ้นหรือลงอยู่ก็ได้ และอาจจะมีเครื่องบินหลายลำที่รอขึ้นและรอลง จึงมีคิว 2 คิวเกิดขึ้น คือ คิวขึ้น (takeoff) และคิวลง ( landing) ในการรอนั้นบนพื้นจะดีกว่าบนอากาศ ดังนั้นจึงให้เครื่องบินขึ้นได้ก็ต่อเมื่อไม่มีเครื่องบินลง หลักจากได้รับสัญญาณร้องขอจากเครื่องบินลำใหม่เพื่อจะลงหรือขึ้นโปรแกรมการจำลองแบบจะให้บริการเครื่องบินที่อยู่ในตำแหน่งหัวคิวของคิวลงก่อน และถ้าคิวลงว่าง จึงอนุญาตให้เครื่องบินในคิวขึ้นขึ้นได้ โปรแกรมจำลองแบบนี้สามารถจะทำงานได้ตลอดเวลา

## ประสิทธิภาพเชิงเวลา

### enqueue() – การเพิ่มข้อมูลไปท้ายคิว

Queue แบบ Array: ใช้เวลา O(1) ถ้ามีที่ว่าง, แต่จะกลายเป็น O(n) ถ้าต้องเลื่อนข้อมูลหรือขยาย arrayQueue แบบ Linked List: ใช้เวลา O(1) เพราะแค่ชี้ node ไปท้ายได้เลย

### dequeue() – การนำข้อมูลออกจากหน้าคิว

Queue แบบ Array: ใช้เวลา O(n) เพราะต้องเลื่อนของทุกตัวให้มาชิดซ้ายQueue แบบ Linked List: ใช้เวลา O(1) เพราะแค่ขยับหัว node ไปยังตัวถัดไป

### peek() – ดูว่าใครอยู่หน้าสุด

ทั้ง Array และ Linked List ใช้เวลา O(1)

### isEmpty() – เช็คว่าคิวว่างไหม

ทั้ง Array และ Linked List ใช้เวลา O(1)

## บทสรุปของคิว

คิวเป็นโครงสร้างข้อมูลแบบเชิงเส้นที่มีลักษณะเฉพาะตัวที่เป็นการนําข้อมูลเข้าก่อนจะออกก่อน (First-in, First-Out) ซึ่งจะต้องใช้ตัวแปรพอยต์เตอร์ จํานวน 2 ตัวคือ Front (ใช้ในการระบุต้นแถวของคิว) Rear (ใช้ในการระบุท้ายแถวของคิว) ซึ่งจะแตกต่างจะ Stack ที่ใช้พอยเตอร์เพียงตัวเดียวคือ Top ในส่วนของ การพัฒนาสามารถพัฒนาได้ทั้งบนโครงสร้าง Array และ โครงสร้าง Linked list นอกจากคิวทีมีลักษณะเป็น Linear แล้ว ยังมีคิวแบบวงกลม (Circular queue) ที่มีลักษณะพิเศษคือมีลักษณะเป็นวงกลม และคิวอภิสิทธิ์ (Priority queue) ที่จะมีการสลับการให้บริการได้ตามลําดับความสําคัญ คิวพบเห็นได้ในชีวิตประจําวันเช่น การเข้าแถวรอรับบริการต่าง ๆ

**ตัวอย่างโค้ดคิวพื้นฐานด้วย list**

# คิวด้วย list ธรรมดา

queue = []

# enqueue - เพิ่มเข้าไปท้ายคิว

queue.append('A')

queue.append('B')

queue.append('C')

# dequeue - เอาคนหน้าออก

print(queue.pop(0)) # A

print(queue) # ['B', 'C']

**ตัวอย่างโค้ดคิวด้วย collections.deque**

from collections import deque

queue = deque()

# enqueue

queue.append('A')

queue.append('B')

queue.append('C')

# dequeue

print(queue.popleft()) # A

print(queue) # deque(['B', 'C'])

# peek

print(queue[0]) # B

# เช็คว่างไหม

print(not queue) # False

**ตัวอย่างโค้ดสร้าง Queue แบบ Circular Queue (วงกลม)**

class CircularQueue:

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

self.queue = [None] \* size

self.size = size

self.front = self.rear = -1

def enqueue(self, item):

if (self.rear + 1) % self.size == self.front:

print("Queue is full")

return

if self.front == -1:

self.front = 0

self.rear = (self.rear + 1) % self.size

self.queue[self.rear] = item

def dequeue(self):

if self.front == -1:

print("Queue is empty")

return

data = self.queue[self.front]

if self.front == self.rear:

self.front = self.rear = -1

else:

self.front = (self.front + 1) % self.size

return data

def display(self):

if self.front == -1:

print("Queue is empty")

return

i = self.front

while True:

print(self.queue[i], end=" ")

if i == self.rear:

break

i = (i + 1) % self.size

print()

# ทดลอง

cq = CircularQueue(5)

cq.enqueue("A")

cq.enqueue("B")

cq.enqueue("C")

cq.display()

cq.dequeue()

cq.display()

**ตัวอย่างโค้ดใช้ queue.Queue (เหมาะกับงาน multi-threading)**

import queue

q = queue.Queue()

q.put("A")

q.put("B")

q.put("C")

print(q.get()) # A

print(q.get\_nowait()) # B

print(q.empty()) # False

**ตัวอย่างโค้ด Priority Queue (คิวตามลำดับความสำคัญ)**

import queue

pq = queue.PriorityQueue()

pq.put((2, "ชานมไข่มุก")) # ยิ่งเลขน้อย ยิ่งได้ก่อน

pq.put((1, "บิงซู"))

pq.put((3, "เครป"))

print(pq.get()) # (1, 'บิงซู')

print(pq.get()) # (2, 'ชานมไข่มุก')

**ตัวอย่างโค้ด Queue เองด้วย Linked List**

class Node:

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

self.data = data

self.next = None

class LinkedQueue:

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

self.front = self.rear = None

def enqueue(self, data):

new\_node = Node(data)

if self.rear is None:

self.front = self.rear = new\_node

return

self.rear.next = new\_node

self.rear = new\_node

def dequeue(self):

if self.front is None:

return "Queue is empty"

data = self.front.data

self.front = self.front.next

if self.front is None:

self.rear = None

return data

def peek(self):

return self.front.data if self.front else "Queue is empty"

def is\_empty(self):

return self.front is None

**# ใช้งาน (ทดสอบ)**

q = LinkedQueue()

q.enqueue("A")

q.enqueue("B")

print(q.dequeue()) # A

print(q.peek()) # B

**ตัวอย่างโค้ดคิวแบบ Double-ended Queue (Deque ใช้ได้ทั้งสองฝั่ง)**

from collections import deque

dq = deque()

# เพิ่มหน้าหลัง

dq.append("ขวา")

dq.appendleft("ซ้าย")

# เอาออกจากทั้งสองฝั่ง

print(dq.pop()) # ขวา

print(dq.popleft()) # ซ้าย

**ตัวอย่างโค้ด Priority Queue แบบ Heaps**

import heapq

pq = []

heapq.heappush(pq, (3, "เรียน"))

heapq.heappush(pq, (1, "กินข้าว"))

heapq.heappush(pq, (2, "นอน"))

print(heapq.heappop(pq)) # (1, 'กินข้าว')

**ตัวอย่างโค้ด Asynchronous Queue (คิวแบบ async)**

import asyncio

async def producer(queue):

for i in range(3):

item = f"ข้อมูล-{i}"

await queue.put(item)

print("Producer: put", item)

await asyncio.sleep(0.5) # จำลองดีเลย์

await queue.put(None) # ส่ง sentinel ให้ consumer หยุด

async def consumer(queue):

while True:

item = await queue.get()

if item is None: # เจอ sentinel

print("Consumer: หยุดทำงาน")

break

print("Consumer: ทำงานกับ", item)

await asyncio.sleep(0.2) # จำลองการประมวลผล

queue.task\_done()

async def main():

queue = asyncio.Queue()

prod = asyncio.create\_task(producer(queue))

cons = asyncio.create\_task(consumer(queue))

await prod

await cons

await main()

import asyncio

async def producer(queue):

for i in range(3):

item = f"ข้อมูล-{i}"

await queue.put(item)

print("Producer: put", item)

await asyncio.sleep(0.5) # จำลองดีเลย์

await queue.put(None) # ส่ง sentinel ให้ consumer หยุด

async def consumer(queue):

while True:

item = await queue.get()

if item is None: # เจอ sentinel

print("Consumer: หยุดทำงาน")

break

print("Consumer: ทำงานกับ", item)

await asyncio.sleep(0.2) # จำลองการประมวลผล

queue.task\_done()

async def main():

queue = asyncio.Queue()

prod = asyncio.create\_task(producer(queue))

cons = asyncio.create\_task(consumer(queue))

await prod

await cons

**ตัวอย่างโค้ด Double Priority Queue (Min/Max Queue) – คิวที่จัดได้ทั้งมากและน้อยพร้อมกัน**

import heapq

class DualPriorityQueue:

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

self.min\_heap = []

self.max\_heap = []

def insert(self, val):

heapq.heappush(self.min\_heap, val)

heapq.heappush(self.max\_heap, -val)

def get\_min(self):

return self.min\_heap[0]

def get\_max(self):

return -self.max\_heap[0]

dpq = DualPriorityQueue()

dpq.insert(5)

dpq.insert(2)

dpq.insert(9)

print("Min:", dpq.get\_min()) # 2

print("Max:", dpq.get\_max()) # 9

**ตัวอย่างโค้ด Simulation Queue (จำลองคนต่อคิว)**

from collections import deque

import random

queue = deque()

for i in range(5):

queue.append(f"คนที่ {i+1}")

while queue:

current = queue.popleft()

print(f"{current} กำลังทำธุรกรรม...")

if random.random() < 0.2:

print(f"{current} เอกสารไม่ครบ! ต่อคิวใหม่")

queue.append(current)ฃ

**ตัวอย่างโค้ด Implement Stack Using Queue (เอาคิวมาสร้างสแตก)**

from collections import deque

class StackUsingQueue:

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

self.q = deque()

def push(self, x):

self.q.append(x)

for \_ in range(len(self.q) - 1):

self.q.append(self.q.popleft())

def pop(self):

return self.q.popleft()

def top(self):

return self.q[0]

s = StackUsingQueue()

s.push(1)

s.push(2)

print(s.top()) # 2

s.pop()

print(s.top()) # 1

# สแตก

## ความหมายของสแตก

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

## โครงสร้างของสแตก (Stack structure)

โครงสร้างข้อมูลแบบสแตก (Stack Structure) เป็นโครงสร้างข้อมูลประเภทเชิงเส้นที่มีการจัดเก็บข้อมูลแบบเรียงลำดับต่อเนื่องกัน ในนำข้อมูลเข้าสู่สแตกหรือนำข้อมูลออกจากสแตกสามารถทำได้เพียงทางเดียวเท่านั้น คือที่ส่วนปลายสแตกหรือส่วนบนสุดของสแตก (Top of the Stack) ที่เรียกว่า Top ตัวอย่างเช่น นำข้อมูลเข้าสู่ สแตกตามลำดับดังนี้คือ A, B, C และ D เมื่อมี การนำข้อมูลออกจากสแตก จะเรียงลำดับดังนี้คือ D, C, B และ A จะเห็นได้ว่าข้อมูลที่ออกมาเป็นข้อมูลย้อนกลับจากตอนนำเข้า จากลักษณะการทำงานของสแตกดังกล่าว ทำให้โครงสร้างข้อมูลแบบสแตกถูกเรียกว่าเป็นโครงสร้างข้อมูลแบบ เข้าทีหลังออกก่อน หรือ Last In, First Out (LIFO) หมายถึง สมาชิกที่เข้าไปในสแตกหลังสุด จะถูกนำออกจากสแตกก่อนสมาชิกตัวอื่น

โครงสร้างของสแตกมีลักษณะ ดังนี้

### โค้งสร้างข้อมูลเป็นแบบเชิงเส้น (Linear structure) ลักษณะโครงสร้างจะจัดเรียงต่อเนื่องกันไป ซึ่งถือเป็นโครงสร้าง ในรูปแบบของอาร์เรย์และเรคอร์ด

### มีลักษณะโครงสร้างที่ไม่ตายตัว (Dynamic structure) สามารถปรับเปลี่ยนจำนวนสมาชิกในโครงสร้างได้ ขณะที่มีการทำงานอยู่ แต่ถ้าเป็นการใช้โครงสร้างของอาร์เรย์ จำเป็นต้องมีการกำหนดพื้นที่การจองบนหน่วยความจำที่เหมาะสม

นำข้อมูลและดึงออกข้อมูลได้ (push and pop) สามารถที่จะทำการ push และ pop สลับกันไปมาได้ ไม่จำเป็นต้องนำข้อมูลเข้าอย่างเดียว หรือดึงข้อมูลออกทั้งหมดพร้อมกัน เพราะถือว่าแต่ละข้อมูลแยกกันตามโครงสร้างที่ออกแบบมาเพื่อจัดเก็บ แต่สำหรับการนำข้อมูลเข้าหรือนำข้อมูลออกนั้น อาจเกิดข้อผิดพลาดขึ้นได้กรณีที่สแตกนั้นเต็มแล้ว นำข้อมูลเข้าไปอีกจะเกิดการผิดพลาดที่ เรียกว่า stack overflow หรือหากเป็นการนำข้อมูลออก โดยที่ภายในสแตกไม่มีข้อมูลอยู่เลยเรียกความผิดพลาดนี้ว่า stack underflow

ข้อมูลเข้าและดึงข้อมูลออกแบบลำดับ (LIFO mechanism) เมื่อต้องการนำข้อมูลเข้าหรือออกจากสแตกจะต้องนำออกตามการเรียงที่เป็นลำดับ ไม่ข้าม หรือกระโดดไปเอาข้อมูลใดข้อมูลหนึ่งมาก่อน

มีการจัดการนำเข้าและดึงข้อมูลในตำแหน่งบนสุด (push and pop at top) การจัดการต่าง ๆ อันได้แก่ การนำข้อมูลเข้าและการดึงข้อมูลออกจะต้องกระทำ ณ ตำแหน่งบนสุดเท่านั้น

## การดำเนินการของสแตก

ลักษณะการดำเนินการในโครงสร้างแบบสแตก ประกอบด้วย 3 กรณี คือ

### การนำข้อมูลเข้า (push) คือ การเพิ่มข้อมูลลงในสแตก กรณีที่ไม่มีข้อมูลได้อยู่ก็จะต้อง push เข้าไปในตำแหน่งแรก ซึ่งถือว่าข้อมูลนี้เป็นตำแหน่ง top แต่ถ้าหากนำข้อมูล push เข้ามาอีก ก็จะดำเนินการ จัดลงในตำแหน่งต่อจาก top และปรับค่า top มาอยู่ที่ข้อมูลที่ push เข้าไปใหม่ แต่ต้องมีการตรวจสอบก่อนว่าสแตกเต็ม stack overflow หรือไม่

### การดึงข้อมูลออก (pop**)** คือ การนำข้อมูลออกจากสแตก ซึ่งการดำเนินการก็จะต้องดำเนินการในตำแหน่ง top กรณีของการ pop ต้องตรวจสอบก่อนว่า สแตกว่าง stack underflow หรือไม่ จึงจะสามารถนำข้อมูลออกจากสแตกได้

1.3.3 ตำแหน่งบนสุด (top)คือ ตำแหน่งบนสุดของสแตกหากต้องการ pop หรือ push ข้อมูลจากต้องทำ ณ ตำแหน่งนี้ โดยลักษณะของการดำเนินการของ top เป็นเพียงสิ่งที่บอกตำแหน่งของข้อมูลที่อยู่บนสุดเท่านั้น หากมีการ push ข้อมูลตำแหน่งของ top ก็จะชี้ค่าไปที่ตำแหน่งสูงสุดใหม่ คือ top = top+1 และหากมีการ pop ข้อมูลเข้าไป top ก็ไม่ใช่ตัวดำเนินการในการลบค่า แต่จะเป็นการคืนค่าและลดตำแหน่งลงมา คือ top = top-1 ซึ่ง top จะเกิดความผิดพลาดกรณีเดียวกันกับ pop ก็คือ stack underflow เมื่อสแตกนั้นเกิดการวาง

## การแทนสแตกด้วยอาร์เรย์

การนำอาร์เรย์เข้ามาใช้งานในการกำหนดโครงสร้างหรือแทนที่ข้อมูลในสแตก ซึ่งลักษณะเฉพาะตัวของโครงสร้างอาร์เรย์เป็นโครงสร้างที่ สามารถกำหนดจองพื้นที่บนหน่วยความจำได้แน่นอนและ สามารถเก็บข้อมูลที่เป็นชนิดเดียวกัน และมีตัวแปรในการเก็บตำแหน่งบนสุดของสแตกคือ top โดยข้อดี คือ ง่ายและสะดวกต่อการใช้งานและการสร้าง แต่ข้อจำกัดคือ ประกาศหน่วยความจำแล้วไม่สามารถเพิ่มขนาดหน่วยความจำได้อีก

การดำเนินการแทนสแตกด้วย อาร์เรย์มีการทำงานพื้นฐานประกอบด้วย 3 ประการ คือ

### การเพิ่มข้อมูลในสแตก (Push) การเพิ่มข้อมูลลงในส่วน top ของสแตก

### การลบข้อมูลลงในสแตก (Pop) การดึงข้อมูลจาก top ของสแตก

### ตรวจสอบว่ามีข้อมูลอยู่ภายในสแตกหรือไม่ (Empty) ส่วนใหญ่แล้วมักใช้ร่วมกับ (pop) เนื่องจากจะไม่สามารถดึงข้อมูลออกมาจากสแตกว่างหรือสแตกที่ไม่มีข้อมูลใดๆ อยู่ได้ ดังนั้นจึงต้องตรวจสอบดูก่อนว่ามีข้อมูลอยู่ในสแตกหรือไม่ ก่อนที่ดึงข้อมูลออกมาจากสแตก

## ส่วนประกอบของสแตก

### ตัวชี้สแตก หรือ Stack Pointer ซึ่งเป็นตัวควบคุมการนำสมาชิกเข้า หรือออกจากสแตก เป็นตัวใช้บอกกว่าสแตกนั้นเต็มหรือยัง

### ส่วนสมาชิกของสแตก หรือจะเรียกอีกอย่างว่า Stack Element สมาชิกของสแตกนี้จะเป็นข้อมูลชนิดเดียวกันทั้งหมด

## การสร้างสแตก

การสร้างสแตกเป็นการจองพื้นที่ในหน่วยความจำให้กับสแตกเพื่อจัดเก็บข้อมูล การสร้างสแตกสามารถทำได้โดยใช้โครงสร้างข้อมูลแบบอาร์เรย์หรือโครงสร้างข้อมูลแบบลิงค์ลิสต์ ก็ได้เมื่อสร้างสแตกเสร็จแล้ว สแตกที่ถูกสร้างขึ้นมาใหม่จะเป็นสแตกว่างยังไม่มีข้อมูลใดๆ แต่พร้อมที่จะดำเนินการในการรับข้อมูล

การสร้างสแตกด้วยโครงสร้างข้อมูลแบบอาร์เรย์ การจัดสรรพื้นที่หน่วยความจำจะเป็นแบบสถิตย์ (Static) กล่าวคือ จัดสรรพื้นที่แบบคงที่ตายตัวไม่สามารถเปลี่ยนแปลงได้ ดังนั้น จึงต้องทำการกำหนดขนาดของสแตกไว้ล่วงหน้าว่าจะมีขนาดเท่าใด โดยจะต้องกำหนดให้เพียงพอกับ จำนวนข้อมูล และข้อมูลที่จัดเก็บในสแตกจะเป็นชนิดใดก็ได้ แต่ต้องเป็นชนิดเดียวกันทั้งหมด

การสร้างสแตกด้วยโครงสร้างข้อมูลแบบลิงค์ลิสต์ เป็นการจัดสรรพื้นที่หน่วยความจำ แบบไดนามิก (Dynamic) กล่าวคือ เป็นแบบยืดหยุ่น จึงไม่จำเป็นต้องกำหนดขนาดพื้นที่ของ สแตก ไว้ล่วงหน้า เนื่องจากหน่วยความจำจะถูกจัดสรรเมื่อมีการใช้งานจริงเท่านั้น จึงทำให้สแตกที่สร้าง ด้วยลิงค์ลิสต์ไม่มีวันเต็ม ตราบที่ยังมีพื้นที่เพียงพอต่อการจัดสรรได้อยู่

ในที่นี้จะขอยกตัวอย่างการสร้างสแตกด้วยโครงสร้างข้อมูลแบบอาร์เรย์เท่านั้นเนื่องจาก โครงสร้างข้อมูลแบบสแตกมีการเก็บข้อมูลแบบเรียงลำดับเหมือนกับโครงสร้างข้อมูลแบบอาร์เรย์แต่ จะแตกต่างกันในส่วนของการประมวลผล นั่นคือ การเข้าถึงสมาชิกในอาร์เรย์จะใช้เลขดัชนีระบุ ตำแหน่งที่ต้องการได้โดยตรง แต่โครงสร้างข้อมูลแบบสแตกจะมีการนำข้อมูลเข้าและออกได้ทางเดียว คือที่ปลายสแตกหรือส่วนบนสุดของสแตกเท่านั้น

## ตัวอย่างโค้ดการทำงานของสแตก

### ตัวอย่าง Code การ Push และ Pop

class Stack:

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

self.items = []

def is\_empty(self):

return len(self.items) == 0

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

return None

def peek(self):

if not self.is\_empty():

return self.items[-1]

return None

def size(self):

return len(self.items)

def \_\_str\_\_(self):

return f"Stack: {self.items}"

# เรียกใช้งานจริง

my\_stack = Stack()

print("Stack ว่างอยู่ไหม:", my\_stack.is\_empty())

print("ขนาดเริ่มต้นของ Stack:", my\_stack.size())

print("สถานะปัจจุบัน:", my\_stack)

# ทดสอบการ push

my\_stack.push("A")

my\_stack.push("B")

my\_stack.push("C")

print("\nหลังจาก push:")

print("Stack ว่างอยู่ไหม:", my\_stack.is\_empty())

print("ดูค่าบนสุด:", my\_stack.peek())

print("ขนาด Stack:", my\_stack.size())

print("สถานะปัจจุบัน:", my\_stack)

# ทดสอบการ pop

print("\nกำลัง pop:", my\_stack.pop())

print("สถานะหลัง pop:", my\_stack)

**ตัวอย่างผลลัพธ์**

Stack ว่างอยู่ไหม: True

ขนาดเริ่มต้นของ Stack: 0

สถานะปัจจุบัน: Stack: []

หลังจาก push:

Stack ว่างอยู่ไหม: False

ดูค่าบนสุด: C

ขนาด Stack: 3

สถานะปัจจุบัน: Stack: ['A', '1', 'C']

กำลัง pop: C

สถานะหลัง pop: Stack: ['A', '1']

### ตัวอย่างการสร้าง Stack

class Stack:

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

self.\_data = []

def push(self, item):

self.\_data.append(item)

def pop(self):

if self.is\_empty():

print("สแตกว่างอยู่ ไม่มีข้อมูลให้ดึง")

return None

return self.\_data.pop()

def peek(self):

if self.is\_empty():

print("สแตกว่างอยู่ ไม่มีข้อมูลบนสุด")

return None

return self.\_data[-1]

def is\_empty(self):

return len(self.\_data) == 0

def show(self):

if self.is\_empty():

print("สแตกว่าง")

else:

print("ข้อมูลในสแตก (บนสุดอยู่ซ้าย):", list(reversed(self.\_data)))

# โปรแกรมหลัก

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

s = Stack()

while True:

print("\n=== เมนูจัดการสแตก ===")

print("1. Push (ใส่ข้อมูล)")

print("2. Pop (ดึงข้อมูลบนสุดออก)")

print("3. Peek (ดูข้อมูลบนสุด)")

print("4. แสดงสแตกทั้งหมด")

print("5. ออก")

choice = input("เลือกเมนู (1-5): ")

if choice == "1":

item = input("ป้อนค่าที่จะใส่: ")

s.push(item)

print(f" ใส่ {item} ลงสแตกแล้ว")

elif choice == "2":

item = s.pop()

if item is not None:

print(f" ดึงออก: {item}")

elif choice == "3":

item = s.peek()

if item is not None:

print(f" ข้อมูลบนสุดคือ: {item}")

elif choice == "4":

s.show()

else:

print(" เลือกเมนูไม่ถูกต้อง ลองใหม่อีกที")

## Operation พื้นฐานของ Stack

### push() ใช้เพื่อเพิ่มข้อมูลใน Stack

### pop() ใช้เพื่อลบข้อมูลออกจาก Stack

### isEmpty() ใช้เพื่อตรวจสอบว่า Stack ว่างหรือไม่

### isFull() ใช้เพื่อตรวจสอบว่า Stack เต็มหรือไม่

### peek() return ข้อมูลที่ตำแหน่งที่ต้องการ

### count() return จำนวนรายการทั้งหมดที่มีใน Stack

### change() ใช้เพื่อเปลี่ยนข้อมูลที่ตำแหน่งต้องการ

### display() ใช้เพื่อแสดงผลข้อมูลทั้งหมดที่มีใน Stack

## เปรียบเทียบประสิทธิภาพของการสร้างสแตกด้วยอาร์เรย์และลิงค์ลิสต์

การเลือกการแทนที่ข้อมูลสแตกด้วยอาร์เรย์ มีข้อจำกัดสำหรับขนาดของสแตกและจะต้องจองเนื้อที่เท่ากับขนาดที่ใหญ่ที่สุดของสแตกไว้เลย เพราะเป็นการจัดสรรเนื้อที่ในหน่วยความจำแบบสแตก

ส่วนการเลือกการแทนที่ข้อมูลสแตกด้วยลิงค์ลิสต์ ไม่มีข้อจำกัดของขนาดของสแตกและหน่วยความจำจะถูกใช้ก็ต่อเมื่อมีข้อมูลจริงๆ แล้วเท่านั้น เพราะเป็นการจัดสรรเนื้อที่หน่วยความจำแบบไดนามิก ซึ่งทำให้ประหยัดเนื้อที่ในหน่วยความจำมากกว่า แต่การเขียนโค้ดสำหรับการแทนที่ข้อมูลสแตกด้วยอาร์เรย์ง่ายกว่าการแทนที่ข้อมูลด้วยลิงค์ลิสต์

## การประยุกต์ใช้งานสแตก

### การจัดการหน่วยความจำ

### การใช้สแตกในกระบวนการเรียกใช้โพรซีเจอร์หรือฟังก์ชัน สแตกเป็นโครงสร้างข้อมูลที่มีประโยชน์มาก ถูกนำไปใช้ทั้งในซอฟท์แวร์ระบบ (System Software) และในการประยุกต์โดยยูสเซอร์ (user) เช่น ช่วยคอมไพเลอร์ (Com-piler) สร้างกฏเกณฑ์ของโปรแกรมมิ่งเกี่ยวกับการเรียกโปรแกรมย่อย (Subprogram call) ที่ว่าโปรแกรมย่อยใดที่ถูกเรียกทำงานที่หลังสุด ต้องทำงานเสร็จก่อน

### การตรวจสอบอักขระสมดุล (Balancing Symbol)

ผู้ที่มีประสบการณ์ในการเขียนโปรแกรมมาแล้ว จะพบว่าสิ่งที่เรามักจะหลงลืมเมื่อเขียนโปรแกรม และทำให้เกิดข้อผิดพลาดอยู่บ่อย ๆ คือ การหลงลืมอักขระสมดุล เช่น { คู่กับ }, [คู่กับ ], ( คู่กับ ) เป็นต้น ซึ่งในการตรวจสอบอักขระสมดุลนั้นคอมไพเลอร์นำชนิดข้อมูลแบบสแตกมาประยุกต์ใช้ได้ โดยมีวิธีการดังนี้ให้อ่านอักขระทีละตัว

ถ้า อักขระเป็นอักขระเปิด เช่น {, [, (, เป็นต้น ให้ Push ลงสแตก

### ถ้า อักขระเป็นอักขระปิด เช่น }, ], ), เป็นต้น ให้ตรวจสอบว่าอักขระบน Top ของสแตกเป็นอักขระเปิดที่คู่กันหรือไม่ ถ้าใช้ให้ Pop อักขระนั้นออกจากสแตก แต่ถ้าไม่ใช่ แสดงผล error เมื่ออ่านอักขระ หมดแล้ว แต่สแตกไม่เป็นสแตกว่างให้แสดงผล error

## การแปลงนิพจน์ infix ให้เป็น postfix

โดยปกติเวลาเขียนโปรแกรมสั่งให้เครื่องคำนวณ ต้องเขียนนิพจน์ที่ต้องการไปในตัวโปรแกรม ซึ่งนิพจน์เหล่านี้ เรียกว่า นิพจน์ infix คือนิพจน์ที่มี ตัวดำเนินการ (Operator) อยู่ระหว่างตัวถูกกระทำ (Operand) เช่น A + B เครื่องหมาย + เป็นโอเปอร์เตอร์ระหว่างโอเปอร์แรนด์ A และ B ซึ่งเห็นว่าเป็นนิพจน์ที่มนุษย์คุ้นเคย

ตัวดำเนินการ ก็คือ เครื่องหมายทางคณิตศาสตร์สำหรับการคำนวณต่างๆ เรียงตามลำดับการดำเนินการก่อน-หลัง (precedence) ได้แก่

ยกกำลัง ^

คูณ หาร \* , /

บวก ลบ + , -

ถ้าเครื่องหมายมีลำดับการดำเนินการเดียวกัน จะเลือกดำเนินงานของเครื่องหมายจากซ้ายไปขวา (ยกเว้น ยกกำลัง) และถ้ามีวงเล็บจะดำเนินงานสิ่งที่อยู่ในวงเล็บก่อน

### ข้อดีของนิพจน์ postfix คือ เป็นนิพจน์ที่มีการคำนวณตามโอเปอร์เรเตอร์ที่มาก่อนหลัง เช่น นิพจน์ ABC\*+ หมายถึง ทำการคูณแล้วจึงทำการบวก ซึ่งคือต้องคำนวณ B\*C ก่อนแล้วจึงนำผลลัพธ์นั้นไปบวกกับ A ต่อไป

### ข้อเสียของนิพจน์ infix ที่ทำให้คอมไพเลอร์ยุ่งยาก คือลำดับความสำคัญของโอเปอร์เรเตอร์(Precedence) ที่ต่างกัน เช่น เครื่องหมายยกกำลัง (^) มีความสำคัญมากกว่าเครื่องหมายคูณ (\*) และหาร (/) และเครื่องหมายคูณและหารมีความสำคัญมากกว่าเครื่องหมายบวก (+) และลบ (-) เครื่องหมายใดมีลำดับความสำคัญมากกว่าจะถูกคำนวณก่อน (ถ้าไม่มีวงเล็บกำกับ) เช่น A + B \* C เครื่องจะคำนวณ B \* C ก่อนแล้วนำผลลัพธ์นั้นไปบวกกับค่า A ซึ่งจะทำงานเหมือนกับ A + (B \* C) ส่วนนิพจน์ใดที่มีโอเปอร์เรเตอร์ที่มีลำดับความสำคัญเท่ากัน การคำนวณจะกระทำจากซ้ายไปขวา เช่น A – B + C จะทำ A – B ก่อน แล้วจึงนำผลลัพธ์นั้นไปบวกกับ ค่า C เมื่อการประมวลผลนิพจน์ infix เป็นไปด้วยความยากที่การคำนวณไม่เป็นไปตามลำดับของเครื่องหมายโอเปอร์เรเตอร์ที่มีก่อนหลัง คอมไพเลอร์จึงแปลงนิพจน์ infix ให้เป็นนิพจน์ postfix เสียก่อน ซึ่งนิพจน์ postfix คือนิพจน์ที่มีโอเปอร์เรเตอร์อยู่หลังโอเปอร์แรนด์ทั้งสองของมัน

### การหาค่าผลลัพธ์จากนิพจน์ Postfix

การหาค่าทางคณิตศาสตร์จากนิพจน์ Postfix มีขั้นตอนใหญ่ ๆ 2 ขั้นตอน คือ

1. ถ้าเป็น Operand ให้ PUSH ลงสู่สแตก

2. ถ้าเป็น Operator ให้ POP ค่า 2 ค่า จากสแตก แล้วดำเนินการโดยใช้ Operator ตัวนั้น ในการนี้ให้ใช้ค่าแรกที่ได้จากสแตกเป็น Operand ตัวที่ 2 จากนั้นก็เก็บค่าผลลัพธ์ในสแตก

### สรุปขั้นตอนในการหาผลลัพธ์ของนิพจน์ Postfix

ค้นหาเครื่องหมายดำเนินการทางซ้ายสุด ของนิพจน์เลือกตัวถูกดำเนินการ 2 ตัว ที่อยู่ติดกับเครื่องหมายดำเนินการทางซ้าย ประมวลผลตามเครื่องหมายดำเนินการนั้นแทนที่เครื่องหมายดำเนินการและตัวถูกดำเนินการ ด้วยผลลัพธ์ที่ได้

## Time Complexity ของ Stack

Big O notation สำหรับ stack โดยทั่วไปแล้วจะมีค่าคงที่ O(1) สำหรับการดำเนินการพื้นฐาน เช่น push, pop, peek, และ isEmpty หาก stack ถูกนำไปใช้ในโครงสร้างข้อมูลอื่นๆ ที่มีการดำเนินการที่ซับซ้อนกว่านั้น ความซับซ้อนของ Big O อาจเปลี่ยนไป

### การดำเนินการพื้นฐานของ Stack และ Big O ของแต่ละการดำเนินการ

#### push (การเพิ่ม Element) O(1)

#### pop (การลบ Element) O(1)

#### peek (การดู Element บนสุด) O(1)

#### isEmpty (การตรวจสอบ stack ว่าว่างหรือไม่) O(1)

### เหตุผลที่ Big O เป็น O(1)

การดำเนินการ push, pop, peek, และ isEmpty ใน stack มักจะดำเนินการโดยการแก้ไข pointer หรืออ้างอิงไปยังตำแหน่งหน่วยความจำที่แน่นอน โดยไม่จำเป็นต้องวนลูปหรือประมวลผลข้อมูลจำนวนมาก ดังนั้นจึงใช้เวลาคงที่ในการดำเนินการ ไม่ว่า stack จะมีขนาดเท่าไหร่ก็ตาม

### O(1) : constant คืออะไร

O(1) เป็น BigO ที่ดีที่สุด ไม่ว่า input จะเป็นอะไรก็ใช้เวลาทำงานเพียง 1 ครั้งเท่าเดิมเสมอ เช่น การเช็คว่าตัวเลข input เป็นเลขคู่หรือเลขคี่ จากโค้ดตัวอย่างด้านล่างนี้ ไม่ว่าใส่ค่า 0 หรือ 9999999 ก็ทำงานด้วยการ mod ด้วย 2 ครั้งเดียวเท่ากัน

### Big O ในสถานการณ์อื่นๆ

หาก stack ถูกนำไปใช้ในอัลกอริทึมที่ซับซ้อนกว่า เช่น การค้นหาแบบ backtracking หรือการประมวลผลข้อมูลที่มีลำดับความสำคัญ อาจมี Big O ที่ซับซ้อนกว่า O(1)

ตัวอย่าง เช่น การค้นหาแบบ backtracking อาจมี Big O ที่เป็น O(n) หรือ O(n^2) ในกรณีที่แย่ที่สุด ซึ่ง n คือจำนวนข้อมูลใน stack หรือโครงสร้างข้อมูลที่เกี่ยวข้อง

### สรุป

โดยทั่วไปแล้ว Big O ของ Stack สำหรับการดำเนินการพื้นฐาน (push, pop, peek, isEmpty) คือ O(1) แต่สำหรับสถานการณ์ที่ซับซ้อนกว่านั้น Big O อาจมีค่าที่สูงกว่านี้ได้

# กราฟ

## **ความหมายและลักษณะของกราฟ**

### ความหมายของกราฟ

กราฟ หมายถึง กลุ่มของโหนดหลาย ๆ โหนดที่อยู่กันอย่างมีระเบียบซึ่งถูกเชื่อมกันเป็นคู่ โดยเชื่อมกันผ่านโหนดที่เรียกว่า Vertices ส่วนของเส้นเชื่อมแต่ละคู่ที่อยู่ระหว่างคู่โหนดจะเรียกว่า Arcs หรือ Edgesสรุปจากข้อความข้างต้น กราฟ หมายถึง กลุ่มของโหนด (Vertices) และกลุ่มของเส้นเชื่อม (Edges) ที่ใช้เชื่อมโยงโหนดแต่ละคู่เข้าด้วยกันอาจมีทิศทางหรือไม่ก็ได้

### ลักษณะของกราฟ

การเก็บโครงสร้างข้อมูลแบบกราฟ

#### เป็นโครงสร้างข้อมูลเชิงเส้น (Linear Structure) และแบบไม่เชิงเส้น (Non Linear Structure) คือมีทั้งโครงสร้างข้อมูลที่เชื่อมโยงกันโดยเส้นตรง และสามารถเชื่อมโยงกันแบบไม่เป็นลำดับ

#### เป็นโครงสร้างข้อมูลเชิงเครือข่าย (Network Structure) เนื่องจากเส้นเชื่อมในกราฟจะแสดงความสัมพันธ์ของข้อมูลที่สามารถเชื่อมโยงถึงกันได้

#### เป็นโครงสร้างแบบไม่คงที่ (Dynamic Structure) เนื่องจากสามารถเปลี่ยนแปลงได้ง่าย เช่น การเพิ่มหรือลดจุดยอดหรือเส้นเชื่อม ซึ่งทำให้สามารถปรับเปลี่ยนโครงสร้างของกราฟได้อย่างอิสระและยืดหยุ่น

#### การเข้าถึงแบบลำดับ (Ordered Traversal) การเข้าถึงเวอร์เท็กซ์ต่าง ๆ โดย จะเริ่มต้นจากเวอร์เท็กซ์แรกและเรียงตามลำดับไปยังเวอร์เท็กซ์ต่อ ๆ ไป ตามเส้นการเชื่อมต่อสมมติฐานการวิจัย

## ประเภทของกราฟ

ประเภทของโครงสร้างข้อมูลแบบกราฟ แบ่งเป็น 4 ประเภท ดังนี้ 6.2.1 กราฟมีทิศทาง (Directed Graph หรือ Digraph) เป็นกราฟที่ระบุทิศทางการ เชื่อมต่อ ซึ่งเป็นการบอกทิศทางของการเดินทางจากโหนดหนึ่งไปยังโหนดต่อไป

โดยกำหนดสมการ G = (V,E) กำหนดให้ G แทน กราฟ V แทน โหนดบนกราฟ E แทน เส้นเชื่อมโยงบนกราฟ ดังนั้น ในการจัดเก็บส่วนความสัมพันธ์เส้นเชื่องโยง V และ E แล้ว ประสิทธิภาพใน การทำงานของขั้นตอนวิธีได้เป็น O (V+|E) เป็นเวลาที่ใช้ในการเข้าถึงโหนดและเส้นเชื่อมโยงของกราฟ

### กราฟทิศทาง

กราฟทิศทาง หรือที่เรียกว่า กราฟระบุทิศทาง (Directed Graph หรือ Digraph) คือโครงสร้างทางคณิตศาสตร์ที่ใช้แสดงความสัมพันธ์ระหว่างวัตถุ โดยมีลักษณะเฉพาะคือ เส้นเชื่อม (edges) ระหว่างจุดยอด (vertices) จะมี “ทิศทาง” กำกับอยู่ด้วย

#### นิยามทางคณิตศาสตร์

กราฟทิศทางคือคู่อันดับ G = (V, A)

V คือเซตของจุดยอด (Vertices หรือ Nodes)

A คือเซตของ คู่อันดับ (Ordered Pairs) ของจุดยอด ซึ่งเรียกว่า เส้นเชื่อมมีทิศทาง (Directed Edges หรือ Arcs)

#### โค้ด Python สำหรับกราฟมีทิศทาง

import networkx as nx

import matplotlib.pyplot as plt

# สร้างกราฟแบบมีทิศทาง

G = nx.DiGraph()

# เพิ่มโหนด (Vertices)

nodes = ['A', 'B', 'C', 'D', 'E']

G.add\_nodes\_from(nodes)

# เพิ่มเส้นเชื่อม (Edges)

edges = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'D'), ('D', 'E')]

G.add\_edges\_from(edges)

# วาดกราฟ

pos = nx.spring\_layout(G) # หรือใช้ nx.shell\_layout(G) เพื่อให้ดูเป็นวง

nx.draw(G,pos,with\_labels=True,node\_size=1500, node\_color='lightblue', arrows=True, arrowstyle='-|>')

nx.draw\_networkx\_edges(G, pos, arrows=True)

plt.title("Directed Graph: ")

plt.show()

### กราฟไม่มีทิศทาง

กราฟไม่มีทิศทาง (Undirected Graph) คือโครงสร้างทางคณิตศาสตร์ที่ใช้แสดงความสัมพันธ์ระหว่างวัตถุ โดย เส้นเชื่อม (edges) ระหว่างจุดยอด (vertices) ไม่มีทิศทาง — หมายความว่า ความสัมพันธ์ระหว่างจุดยอดทั้งสองเป็นแบบ “สองทาง” หรือ “สมมาตร”

#### นิยามทางคณิตศาสตร์

กราฟไม่มีทิศทางคือคู่อันดับ G = (V, E)

V คือเซตของจุดยอด (Vertices หรือ Nodes)

E คือเซตของ คู่ไม่อันดับ (Unordered Pairs) ของจุดยอด ซึ่งเรียกว่า เส้นเชื่อม (Edges)

#### โค้ด Python สำหรับกราฟไม่มีทิศทาง

import networkx as nx

import matplotlib.pyplot as plt

# สร้างกราฟแบบไม่มีทิศทาง

G = nx.Graph()

# เพิ่มโหนด (Vertices)

nodes = ['A', 'B', 'C', 'D', 'E']

G.add\_nodes\_from(nodes)

# เพิ่มเส้นเชื่อม (Edges) ตามที่ไม่ระบุทิศทาง

edges = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'D'), ('D', 'E')]

G.add\_edges\_from(edges)

# วาดกราฟ

pos = nx.spring\_layout(G) # หรือใช้ nx.circular\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=1500, node\_color='lightgreen', edge\_color='gray')

plt.title("Undirected Graph: ภาพที่ 6.2")

plt.show()

### กราฟมีน้ำหนัก

กราฟมีน้ำหนัก (Weighted Graph) คือกราฟที่ เส้นเชื่อม (edges) ระหว่างจุดยอด (vertices) มีค่าตัวเลขกำกับอยู่ ซึ่งเรียกว่า “น้ำหนัก” (weight) โดยน้ำหนักนี้สามารถแทนค่าได้หลากหลาย เช่น ระยะทาง, เวลา, ค่าใช้จ่าย, ความสำคัญ

#### นิยามทางคณิตศาสตร์

กราฟมีน้ำหนักคือคู่อันดับ G = (V, E, w)

V คือเซตของจุดยอด

E คือเซตของเส้นเชื่อม (อาจมีหรือไม่มีทิศทาง)

w คือฟังก์ชันที่กำหนดน้ำหนักให้กับแต่ละเส้นเชื่อม: $$w: E \rightarrow \mathbb{R}$$ หมายความว่าแต่ละเส้นเชื่อม e จะมีค่าน้ำหนัก w(e) เป็นจำนวนจริง

#### โค้ด Python สำหรับกราฟมีน้ำหนัก

import networkx as nx

import matplotlib.pyplot as plt

# สร้างกราฟมีน้ำหนัก

G\_weighted = nx.Graph()

# เพิ่มเส้นเชื่อมพร้อมน้ำหนัก

edges\_with\_weights = [

('A', 'B', 6),

('A', 'C', 7),

('B', 'D', 5),

('B', 'E', 4),

('C', 'D', 2),

('D', 'E', 2)

]

G\_weighted.add\_weighted\_edges\_from(edges\_with\_weights)

# วาดกราฟ

pos = nx.spring\_layout(G\_weighted)

nx.draw(G\_weighted, pos, with\_labels=True, node\_color='lightblue', node\_size=800, font\_size=12)

nx.draw\_networkx\_edge\_labels(G\_weighted, pos, edge\_labels={(u, v): d['weight'] for u, v, d in G\_weighted.edges(data=True)})

plt.title("กราฟมีน้ำหนัก")

plt.show()

### กราฟไม่มีน้ำหนัก

กราฟไม่มีน้ำหนัก (Unweighted Graph) คือกราฟที่ เส้นเชื่อม (edges) ระหว่างจุดยอด (vertices) ไม่มีค่ากำกับ เช่น ระยะทาง, ค่าใช้จ่าย หรือเวลา กล่าวคือ ทุกเส้นเชื่อมมี “น้ำหนักเท่ากัน” หรือ “ไม่มีน้ำหนัก” โดยปริยาย

#### นิยามทางคณิตศาสตร์

กราฟไม่มีน้ำหนักคือคู่อันดับ G = (V, E)

V คือเซตของจุดยอด (Vertices)

E คือเซตของเส้นเชื่อม (Edges) ซึ่งเป็นคู่ของจุดยอด

ไม่มีฟังก์ชันน้ำหนักกำกับเส้นเชื่อม (ไม่มี w(e))

ตัวอย่างเช่น ถ้า E = {{A, B}, {B, C}, {C, D}} หมายความว่า A เชื่อมกับ B, B เชื่อมกับ C, C เชื่อมกับ D โดยไม่มีการระบุค่าใด ๆ บนเส้นเชื่อม

#### โค้ด Python สำหรับกราฟไม่มีน้ำหนัก

# สร้างกราฟไม่มีน้ำหนัก

G\_unweighted = nx.Graph()

# เพิ่มเส้นเชื่อมโดยไม่ระบุน้ำหนัก

edges = [

('A', 'B'),

('A', 'C'),

('B', 'D'),

('B', 'E'),

('C', 'D'),

('D', 'E')

]

G\_unweighted.add\_edges\_from(edges)

# วาดกราฟ

pos = nx.spring\_layout(G\_unweighted)

nx.draw(G\_unweighted, pos, with\_labels=True, node\_color='lightgreen', node\_size=800, font\_size=12)

plt.title("กราฟไม่มีน้ำหนัก")

plt.show()

## การแทนข้อมูลกราฟ

การแทนข้อมูลกราฟด้วยอาร์เรย์และเมทริกซ์ การแทนข้อมูลกราฟด้วยเมทริกซ์และอาร์เรย์ แทนความสัมพันธ์นั้นด้วย nxn ซึ่ง ก คือ A, B, C, D, E เป็นจำนวน 5 โหนด ทำให้ได้ค่าเมทริกซ์ จำนวน 5x5 = 25 ช่อง และหากพิจารณาตามทิศ ทางการเชื่อมต่อ สามารถพิจารณาได้ตามทิศทางต่อไปนี้

### การแทนข้อมูลกราฟไม่มีทิศทาง การแทนข้อมูลกราฟแบบไม่มีทิศทาง กำหนดให้สมาชิกของ A เป็นอาร์เรย์ 2 มิติ ที่ มีขนาด non เมื่อ ก เป็นจำนวนโหนดบนกราฟ สามารถเขียนเป็นสมการได้ดังนี้ Aj = a(i,j) 1 เมื่อ (ij) และ (j,i) EE 0 เมื่อ (i.j) และ (j,i) Eการแทนข้อมูลกราฟไม่มีทิศทางด้วยอาร์เรย์และเมทริกซ์ แทนที่อาร์เรย์และเมทริกซ์ด้วยจำนวนโหนด โดยกำหนดโหนด แนวนอนเป็นโหนดต้นทางและแนวตั้งเป็นโหนดปลายทาง สามารถเขียนลำดับโดยระบุค่าได้ดังนี้ โหนด A มีเส้นเชื่อมต่อระบุไป B,C ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด B มีเส้นเชื่อมต่อระบุไป A,D,E ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด C มีเส้นเชื่อมต่อระบุไป A,D ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด D มีเส้นเชื่อมต่อระบุไป B,C,E ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด E มีเส้นเชื่อมต่อระบุไป BD ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0

### การแทนข้อมูลกราฟมีทิศทาง การแทนข้อมูลกราฟแบบมีทิศทาง กำหนดให้สมาชิกของ A เป็นอาร์เรย์ 2 มิติ ที่มี ขนาด nxn เมื่อ ก เป็นจำนวนโหนดบนกราฟ สามารถเขียนเป็นสมการได้ดังนี้A = a(i,j); 11 เมื่อ (ij) EE 0 เมื่อ (ij) Eการแทนข้อมูลกราฟมีทิศทางด้วยอาร์เรย์และเมท แทนที่อาร์เรย์และเมทริกซ์ด้วยจำนวนโหนด โดยกำหนดโหนด แนวนอนเป็นโหนดต้นทางและแนวตั้งเป็นโหนดปลายทาง สามารถเขียนลำดับโดยระบุค่าได้ดังนี้ โหนด A มีเส้นเชื่อมต่อระบุไป C ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด B มีเส้นเชื่อมต่อระบุไป A,E ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด C มีเส้นเชื่อมต่อระบุไป D ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด D มีเส้นเชื่อมต่อระบุไป B,E ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0 โหนด E มีเส้นเชื่อมต่อระบุไป D ระบุหมายเลข 1 นอกนั้นระบุค่าเป็น 0

### 3.การแทนข้อมูลกราฟมีน้ำหนักและมีทิศทาง การแทนข้อมูลกราฟมีน้ำหนักและมีทิศทาง กำหนดให้สมาชิกของ A เป็นอาร์เรย์ 2 มิติ ที่มีขนาด non เมื่อ n เป็นจำนวนโหนดบนกราฟ สามารถเขียนเป็นสมการได้ดังนี้ A = a(ij);W เมื่อ (ij) EE W เมื่อ (ij) E 0 กรณีอื่น ๆการแทนข้อมูลกราฟมีน้ำหนักและมีทิศทางด้วยอาร์เรย์และเมทริกซ์ สามารถกำหนดโหนด โดยระบุค่าได้ดังนี้ โหนด A มีเส้นเชื่อมต่อระบุไป B,C ระบุหมายเลข 3,2 นอกนั้นระบุค่าเป็น 0 โหนด B มีเส้นเชื่อมต่อระบุไป D ระบุหมายเลข 5 นอกนั้นระบุค่าเป็น 0 โหนด C มีเส้นเชื่อมต่อระบุไป D ระบุหมายเลข 6 นอกนั้นระบุค่าเป็น 0 โหนด D มีเส้นเชื่อมต่อระบุไป E ระบุหมายเลข 4 นอกนั้นระบุค่าเป็น 0 โหนด E มีเส้นเชื่อมต่อระบุไป B ระบุหมายเลข 7 นอกนั้นระบุค่าเป็น 0

## การแทนข้อมูลกราฟด้วยลิงค์ลิสต์

การแทนข้อมูลกราฟด้วยลิงค์ลิสต์เพื่อจัดเก็บค่าของเส้นเชื่อมโยง จะเก็บเฉพาะโหนดที่มี ความสัมพันธ์กันเท่านั้น

### การแทนข้อมูลกราฟไม่มีทิศทาง การแทนข้อมูลกราฟไม่มีทิศทางโดยใช้พอยน์เตอร์ (Pointer) ทางซ้าย ตามแนวดิ่ง เพื่อเชื่อมโยงแต่ละโหนด และพอยน์เตอร์ทางขวาชี้ตามแนวนอนเพื่อเชื่อมโยงไปยังเส้นเชื่อมโยง

### การแทนข้อมูลกราฟมีทิศทาง การแทนข้อมูลกราฟมีทิศทางโดยใช้พอยน์เตอร์ (Pointer)ทางซ้ายชี้ตามแนวดิ่งเพื่อ เชื่อมโยงแต่ละโหนด และพอยน์เตอร์ทางขวาตามแนวนอนเพื่อเชื่อมโยงไปยังเส้นเชื่อมโยงแต่ละโหนด โดยสามารถแทนด้วยลิงค์ลิสต์ได้จากโหนดเริ่มต้นไปตามเส้นแนวลูกศรชี้เชื่อมโยงจากโหนดหนึ่งไปยัง อีกโหนดหนึ่ง

## การคำนวณจำนวนเส้นทางระหว่างโหนด

การคำนวณจำนวนเส้นทางระหว่างโหนดในกราฟเป็นหัวข้อสำคัญในทฤษฎีกราฟ โดยขึ้นอยู่กับประเภทของกราฟ (มีทิศทาง/ไม่มีทิศทาง, มีน้ำหนัก/ไม่มีน้ำหนัก) และเงื่อนไขของเส้นทางที่ต้องการ เช่น ต้องการเส้นทางทั้งหมด, เส้นทางที่ไม่ซ้ำ, หรือเส้นทางที่สั้นที่สุด

### โค้ด Python

import numpy as np

# Adjacency matrix ของกราฟที่กำหนด

A = np.array([

[0, 1, 1, 0, 0], # A

[0, 0, 0, 1, 0], # B

[0, 0, 0, 0, 1], # C

[0, 0, 1, 0, 0], # D

[0, 0, 0, 0, 0] # E

])

# คูณเมทริกซ์เพื่อหาจำนวนเส้นทางความยาว 2

Z = np.dot(A, A)

# แสดงผลลัพธ์

nodes = ['A', 'B', 'C', 'D', 'E']

print("จำนวนเส้นทางความยาว 2 ระหว่างจุดต่าง ๆ:\n")

for i in range(len(nodes)):

for j in range(len(nodes)):

print(f"{nodes[i]} → {nodes[j]}: {Z[i][j]}"

## การท่องไปในกราฟ

การท่องไปในกราฟ (Graph Traversal) คือกระบวนการในการเยี่ยมชมจุดยอด (vertices) และเส้นเชื่อม (edges) ของกราฟตามลำดับที่กำหนด โดยมีเป้าหมายเพื่อค้นหาข้อมูล, ตรวจสอบโครงสร้าง, หรือแก้ปัญหาต่าง ๆ เช่น การหาทางเดิน, การตรวจสอบการเชื่อมต่อ, หรือการค้นหาเส้นทางที่สั้นที่สุด

### การท่องแบบแนวกว้าง (Breadth First Search : BFS) การท่องเข้าไปในกราฟแบบแนวกว้าง เป็นการท่องเข้าไปในกราฟโดยเข้าเยี่ยมโหนดตัวแรก และดำเนินการ หากมีโหนดใกล้เคียงจะดำเนินการกับโหนดที่อยู่ด้านซ้ายก่อน

#### **โค้ด Python สำหรับ BFS**

from collections import deque

# กำหนดโครงสร้างกราฟแบบ adjacency list

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['G'],

'F': ['H'],

'G': [],

'H': []

}

# ฟังก์ชัน BFS

def bfs(graph, start):

visited = []

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

visited.append(node)

queue.extend(graph[node])

return visited

# เรียกใช้ BFS จากจุด A

order = bfs(graph, 'A')

# แสดงลำดับการเข้าถึง

print("ลำดับการเข้าถึงแบบ BFS:")

for i, node in enumerate(order, start=1):

print(f"{i}. {node}")

### การท่องแบบแนวลึก (Depth First Search : DFS) การท่องเข้าไปในกราฟแบบแนวลึก เป็นลักษณะการท่องไปยังโหนดเริ่มต้น แล้วให้โหนด ใกล้เคียงเป็นโหนดเริ่มต้น เข้าเยี่ยมโหนด ทำต่อไปจนไม่มีโหนดใกล้เคียงจึงย้อนกลับมายังโหนดก่อน หน้า และเข้าเยี่ยมโหนดอีกด้านด้วยรูปแบบเดียวกันจนครบทุกโหนด

#### **โค้ด Python สำหรับ DFS**

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

if visited is None:

visited = []

if node not in visited:

visited.append(node)

for neighbor in graph[node]:

dfs(graph, neighbor, visited)

return visited

# เรียกใช้ DFS จากจุด A

order = dfs(graph, 'A')

#แสดงลำดับการเข้าถึง

print("ลำดับการเข้าถึงแบบ DFS:")

for i, node in enumerate(order, start=1):

print(f"{i}. {node}")

## ประสิทธิภาพเชิงเวลา (Time Complexity)

ประสิทธิภาพเชิงเวลา (Time Complexity หรือ ประสิทธิภาพด้านเวลา) คือการวัดว่า อัลกอริธึมหรือกระบวนการใด ๆ ใช้เวลาในการทำงานมากน้อยเพียงใด เมื่อขนาดของข้อมูลเปลี่ยนไป โดยไม่จำเป็นต้องวัดเป็นวินาที แต่ใช้การวิเคราะห์เชิงคณิตศาสตร์เพื่อประเมิน “แนวโน้ม” ของเวลาในการทำงานตามขนาดของอินพุ

### ตัวอย่าง โค้ด Python

import networkx as nx

import time

import random

# สร้างจำนวนโหนดและเส้นเชื่อม

V = 1000 # จำนวนโหนด

E = 5000 # จำนวนเส้นเชื่อม

# สร้างรายการเส้นเชื่อมแบบสุ่ม

edges = [(random.randint(0, V-1), random.randint(0, V-1)) for \_ in range(E)]

# ==== Adjacency List ====

start\_time = time.time()

G\_list = nx.DiGraph()

G\_list.add\_edges\_from(edges)

list\_build\_time = time.time() - start\_time

# ตรวจสอบว่า node A เชื่อมกับ B หรือไม่ (List)

A, B = edges[0]

start\_check = time.time()

has\_edge\_list = G\_list.has\_edge(A, B)

list\_check\_time = time.time() - start\_check

# หาคู่เพื่อนบ้าน (List)

start\_neighbors = time.time()

neighbors\_list = list(G\_list.neighbors(A))

list\_neighbors\_time = time.time() - start\_neighbors

# ==== Adjacency Matrix ====

start\_time = time.time()

G\_matrix = nx.to\_numpy\_array(G\_list) # แปลงเป็นเมทริกซ์

matrix\_build\_time = time.time() - start\_time

# ตรวจสอบว่า node A เชื่อมกับ B หรือไม่ (Matrix)

start\_check = time.time()

has\_edge\_matrix = G\_matrix[A][B] > 0

matrix\_check\_time = time.time() - start\_check

# หาคู่เพื่อนบ้าน (Matrix)

start\_neighbors = time.time()

neighbors\_matrix = [i for i in range(V) if G\_matrix[A][i] > 0]

matrix\_neighbors\_time = time.time() - start\_neighbors

# ==== แสดงผล ====

print(" เวลาสร้างกราฟ")

print(f"Adjacency List: {list\_build\_time:.6f} sec")

print(f"Adjacency Matrix: {matrix\_build\_time:.6f} sec\n")

print(" เวลาตรวจสอบว่า A → B หรือไม่")

print(f"Adjacency List: {list\_check\_time:.8f} sec")

print(f"Adjacency Matrix: {matrix\_check\_time:.8f} sec\n")

print(" เวลาหาเพื่อนบ้านของ A")

print(f"Adjacency List: {list\_neighbors\_time:.8f} sec")

print(f"Adjacency Matrix: {matrix\_neighbors\_time:.8f} sec")

# โครงสร้างต้นไม้

## ความหมายของโครงสร้างต้นไม้

แผนภาพต้นไม้ หรือ Tree Diagram เป็นหนึ่งในโครงสร้างข้อมูลไม่เชิงเส้น (Non-linear Data Structure) ที่สำคัญมากในวิชาวิทยาการคอมพิวเตอร์ โดยเฉพาะในสาขาโครงสร้างข้อมูลและอัลกอริทึม เพราะช่วยให้สามารถจัดเก็บข้อมูลในรูปแบบลำดับชั้น (Hierarchical)ประมวลผลข้อมูลได้อย่างมีประสิทธิภาพใช้ในการคำนวณ, แสดงความสัมพันธ์ หรือการตัดสินใจการเข้าใจต้นไม้ในแง่ทั้งเชิงโครงสร้างและเชิงตรรกะจะช่วยให้นักเรียนสามารถประยุกต์ใช้กับโครงสร้างที่ซับซ้อนยิ่งขึ้นได้ เช่น Graph, Trie, Heap และ B-Trees

## องค์ประกอบของต้นไม้

โครงสร้างต้นไม้ประกอบด้วย ดังนี้

### Root Node: โหนดราก คือโหนดเริ่มต้นของต้นไม้

### Child Node: โหนดลูก คือโหนดที่อยู่ถัดจากโหนดแม่

### Parent Node: โหนดแม่ ของโหนดลูก

### Leaf Node: โหนดปลายทาง (ไม่มีลูก)

### Edge: เส้นเชื่อมระหว่างโหนด

### Subtree: ต้นไม้ย่อยของต้นไม้ใหญ่

### Height: ระยะทางจากโหนดไปถึงโหนดปลายลึกที่สุด

### Level: ระดับความลึกของโหนด (Root อยู่ระดับ 0)

## ประเภทของต้นไม้

### General Tree

#### โหนดใดๆ สามารถมีลูกกี่โหนดก็ได้

### Binary Tree

#### แต่ละโหนดมีลูกได้สูงสุด 2 โหนด (ซ้ายและขวา)

#### ใช้งานได้หลากหลาย เช่น Expression Tree, Parse Tree

### Full Binary Tree

#### ทุกโหนดมีลูกครบ 0 หรือ 2 คน

### Complete Binary Tree

#### เติมโหนดจากซ้ายไปขวาให้เต็มแต่ละระดับก่อน

### Perfect Binary Tree

#### โหนดทั้งหมดเต็มทุกระดับ และมีลูกครบทุกโหนดที่ไม่ใช่ใบ

### Binary Search Tree (BST)

#### โหนดซ้ายมีค่าน้อยกว่าแม่, ขวามากกว่าแม่

#### ใช้เพื่อเพิ่มประสิทธิภาพการค้นหา, แทรก และลบข้อมูล

### AVL Tree (Balanced Tree)

#### BST ที่สมดุลอยู่เสมอ

#### ความสูงของต้นไม้ซ้ายและขวาแตกต่างไม่เกิน 1

#### มีการ Rotate โหนดเมื่อไม่สมดุล

### Heap Tree

#### ใช้ใน Priority Queue

#### Max-Heap: โหนดแม่มีค่ามากกว่าหรือเท่าลูก

#### Min-Heap: โหนดแม่มีค่าน้อยกว่าหรือเท่าลูก

### Trie (Prefix Tree)

#### ใช้เก็บชุดข้อความ เช่น คำศัพท์, การค้นหาอัตโนมัติ

### B-Tree, B+ Tree

#### ใช้ในระบบฐานข้อมูลและระบบไฟล์

#### เก็บข้อมูลจำนวนมากในรูปแบบมีลำดับและ Balance

## Tree Traversal (การเดินต้นไม้)

### การเดินแบบ Traversal

Traversal คือการเดินไปตามโหนดของต้นไม้ เพื่ออ่านหรือจัดเรียงข้อมูล

#### Inorder Traversal (ซ้าย – ราก – ขวา)

##### ลำดับ:

###### ไปทางซ้ายให้สุด

###### แสดงราก

###### แล้วไปขวา

##### ตัวอย่างการเดิน Inorder:

###### Tree:

A

/ \

B C

/ \ \

D E F

Inorder: D → B → E → A → C → F

##### โค้ด (Python):

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def inorder(node):

"""การเดิน Inorder: ซ้าย -> ราก -> ขวา"""

if node:

inorder(node.left)        # ไปซ้ายก่อน

print(node.value, end=' ')  # แสดงราก

inorder(node.right)       # ไปขวา

root = TreeNode('A')

root.left = TreeNode('B')

root.right = TreeNode('C')

root.left.left = TreeNode('D')

root.left.right = TreeNode('E')

root.right.right = TreeNode('F')

print("Inorder Traversal:")

inorder(root)

print()

#### Preorder Traversal (ราก – ซ้าย – ขวา)

##### ลำดับ:

###### แสดงรากก่อน

###### ไปทางซ้าย

###### แล้วไปขวา

##### ตัวอย่างการเดิน Preorder:

###### Tree:

A

/ \

B C

/ \ \

D E F

Preorder: A → B → D → E → C → F

##### โค้ด (Python):

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def inorder(node):

if node:

inorder(node.left)          # ไปซ้ายก่อน

print(node.value, end=' ')  # แสดงค่าราก

inorder(node.right)         # ไปขวา

print("สร้างต้นไม้...")

root = TreeNode('A')

root.left = TreeNode('B')

root.right = TreeNode('C')

root.left.left = TreeNode('D')

root.left.right = TreeNode('E')

root.right.right = TreeNode('F')

print("โครงสร้างต้นไม้:")

print("       A")

print("      / \\")

print("     B   C")

print("    / \\   \\")

print("   D   E   F")

print()

print("ผลลัพธ์ Inorder Traversal (ซ้าย → ราก → ขวา):")

inorder(root)

print()

print("\nคำตอบที่ถูกต้อง: D B E A C F")

#### Postorder Traversal (ซ้าย – ขวา – ราก)

##### ลำดับ:

###### ไปทางซ้าย

###### แล้วไปขวา

###### สุดท้ายจึงแสดงราก

##### ตัวอย่างการเดิน Postorder:

###### Tree:

A

/ \

B C

/ \ \

D E F

Postorder: D → E → B → F → C → A

##### โค้ด (Python):

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def postorder(node):

if node:

postorder(node.left)        # ไปซ้ายก่อน

postorder(node.right)       # ไปขวา

print(node.value, end=' ')

print("สร้างต้นไม้...")

root = TreeNode('A')

root.left = TreeNode('B')

root.right = TreeNode('C')

root.left.left = TreeNode('D')

root.left.right = TreeNode('E')

root.right.right = TreeNode('F')

print("โครงสร้างต้นไม้:")

print("       A")

print("      / \\")

print("     B   C")

print("    / \\   \\")

print("   D   E   F")

print()

print("ผลลัพธ์ Postorder Traversal (ซ้าย → ขวา → ราก):")

postorder(root)

print()

print("\nคำตอบที่ถูกต้อง: D E B F C A")

## Binary Search Tree (BST)

### เป็น Binary Tree ที่มีลักษณะเฉพาะตัวในการจัดเรียงข้อมูล

### ทำให้สามารถค้นหา/เพิ่ม/ลบ ได้เร็วโดยเฉลี่ย O(log n)

### หากข้อมูลเรียงก่อนจะสร้าง ทำให้ BST กลายเป็นแบบไม่สมดุล (Worst Case = O(n))

### โค้ดพื้นฐาน BST Node:

class Node:

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

self.left = None

self.right = None

self.value = value

### ข้อดีของ BST:

#### เข้าถึงข้อมูลได้รวดเร็ว

#### สนับสนุนการค้นหาที่มีประสิทธิภาพ

### ข้อเสีย:

#### 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

## การวิเคราะห์ต้นไม้ด้วย Big-O

### การวิเคราะห์ใช้ Big-O เพื่อดู worst-case ของต้นไม้

### ถ้าต้นไม้ไม่สมดุล: O(n)

### ถ้าสมดุล (AVL, Red-Black): O(log n)

### การ Traversal ทั้งต้นไม้: O(n)

## โค้ดโครงสร้างต้นไม้

### โครงสร้างพื้นฐานของ Binary Tree

#### Python class Node:

# โครงสร้างของ 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 ลองแทรกฝั่งซ้ายลึกลงไปก่อน

self.\_insert(node.left, data)

### การเดินต้นไม้ (Tree Traversal)

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

##### Python :

def inorder(root):

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

self.inorder(node.right)

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

##### Python :

def preorder(root):

def preorder(self, node):

if node:

print(node.data, end=" ")

self.preorder(node.left)

self.preorder(node.right)

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

##### Python :

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.data, end=" ")

### การค้นหาค่าใน Binary Search Tree (BST)

##### Python class Node:

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

self.data = data

self.left = None

self.right = None

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)

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) # ควรได้: 20 30 40 50 60 70 80

##### ทดสอบการค้นหา

print("\n\nค้นหา 60:", tree.search(60)) # True

print("ค้นหา 25:", tree.search(25)) # False

### การแทรกโหนดใน BST

##### Python class Node:

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

self.data = data

self.left = None # โหนดฝั่งซ้าย

self.right = None # โหนดฝั่งขวา

class BST:

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

self.root = None

# ฟังก์ชันเพิ่มค่าเข้า BST

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 จะไม่เพิ่ม (ป้องกันค่าซ้ำ)

###### Traversal แบบ Inorder (เรียงค่าจากน้อยไปมาก)

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

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

tree = BST()

###### แทรกค่าเข้า BST

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

for v in values:

tree.insert(v

print("Inorder Traversal หลังจากแทรกโหนดทั้งหมด:")

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

### การลบโหนดใน BST

##### Python class Node:

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

self.data = data

self.left = None

self.right = None

class BST:

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 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)

def delete(self, data):

self.root = self.\_delete(self.root, data)

def \_delete(self, node, data):

if node is None:

return node

ถ้าค่าที่ต้องการลบอยู่ทางซ้าย

if data < node.data:

node.left = self.\_delete(node.left, data)

ถ้าค่าที่ต้องการลบอยู่ทางขวา

elif data > node.data:

node.right = self.\_delete(node.right, data)

else:

###### กรณี 1: ไม่มีลูก

if node.left is None and node.right is None:

return

###### กรณี 2: มีลูกข้างเดียว

elif node.left is None:

return node.right

elif node.right is None:

return node.left

###### กรณี 3: มีลูก 2 ข้าง

else:

min\_node = self.\_find\_min(node.right)

node.data = min\_node.data

node.right = self.\_delete(node.right, min\_node.data)

return node

def \_find\_min(self, node):

while node.left:

node = node.left

return node

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

self.inorder(node.right)

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

tree = BST()

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

for v in values:

tree.insert(v)

print("Inorder Traversal ก่อนลบ:")

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

###### ลองลบโหนด

print("\n\nลบ 20 (ไม่มีลูก):")

tree.delete(20)

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

print("\n\nลบ 30 (มีลูก 1 ข้าง):")

tree.delete(30)

tree.inorder(tree.root) # 40 50 60 70 80

print("\n\nลบ 50 (มีลูก 2 ข้าง):")

tree.delete(50)

tree.inorder(tree.root) # 40 60 70 80

### การนับจำนวนโหนดในต้นไม้

##### Python

def count\_nodes(root):

if root is None:

return 0

return 1 + count\_nodes(root.left) + count\_nodes(root.right)

### การหาความสูงของต้นไม้

##### Python class Node:

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

self.data = data

self.left = None

self.right = None

class BST:

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 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)

def height(self, node):

if node is None:

return 0

else:

left\_height = self.height(node.left)

right\_height = self.height(node.right)

return 1 + max(left\_height, right\_height)

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

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

tree = BST()

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

for v in values:

tree.insert(v)

print("Inorder Traversal:")

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

print("\n\nความสูงของต้นไม้:", tree.height(tree.root))

### ตัวอย่าง Expression Tree (ต้นไม้คำนวณนิพจน์)

##### Python class Node:

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

self.value = value

self.left = None

self.right = None

class ExpressionTree:

# สร้าง Expression Tree จาก postfix (Reverse Polish Notation)

def build\_tree(self, postfix):

stack = []

operators = set(['+', '-', '\*', '/'])

for token in postfix:

if token not in operators:

# ถ้าเป็นตัวเลข -> สร้างโหนดใบ

node = Node(token)

stack.append(node)

else:

# ถ้าเป็น operator -> pop 2 โหนดแล้วสร้างเป็นต้นไม้ย่อย

right = stack.pop()

left = stack.pop()

node = Node(token)

node.left = left

node.right = right

stack.append(node)

return stack.pop()

# ประเมินค่า Expression Tree

def evaluate(self, node):

if node is None:

return 0

# ถ้าเป็นใบ (ตัวเลข)

if node.left is None and node.right is None:

return int(node.value)

# คำนวณตาม operator

left\_val = self.evaluate(node.left)

right\_val = self.evaluate(node.right)

if node.value == '+':

return left\_val + right\_val

elif node.value == '-':

return left\_val - right\_val

elif node.value == '\*':

return left\_val \* right\_val

elif node.value == '/':

return left\_val / right\_val

# Traversal (infix expression)

def inorder(self, node):

if node:

if node.left or node.right:

print("(", end="")

self.inorder(node.left)

print(node.value, end=" ")

self.inorder(node.right)

if node.left or node.right:

print(")", end="")

ทดสอบการทำงานของโค้ด

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

postfix expression: 3 4 + 2 \* 7 /

infix: ((3 + 4) \* 2) / 7

postfix = ["3", "4", "+", "2", "\*", "7", "/"]

et = ExpressionTree()

root = et.build\_tree(postfix)

print("Infix Expression: ", end="")

et.inorder(root) # แสดงผล: ((3 + 4) \* 2) / 7

print("\nผลลัพธ์ของนิพจน์:", et.evaluate(root)) # 2.0

### แสดงต้นไม้ระดับต่อระดับ (Level-order Traversal)

##### Python from collections import deque

##### class Node:

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

self.data = data

self.left = None

self.right = None

class BST:

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

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)

def level\_order(self):

if self.root is None:

return

queue = deque([self.root])

while queue:

node = queue.popleft()

print(node.data, end=" ")

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

##### ทดสอบการทำงานของโค้ด

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

tree = BST()

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

for v in values:

tree.insert(v)

print("Level-order Traversal:")

tree.level\_order()

### โค้ดสร้างต้นไม้จากรายการ Inorder + Preorder

##### Python class Node:

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

self.data = data

self.left = None

self.right = None

class BinaryTree:

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

self.pre\_idx = 0 # เก็บตำแหน่งปัจจุบันของ preorder

def buildTree(self, inorder, preorder):

# เก็บ index ของค่าใน inorder เพื่อค้นหาได้เร็วขึ้น

in\_index = {value: i for i, value in enumerate(inorder)}

def arrayToTree(left, right):

if left > right:

return None

###### เลือกค่าปัจจุบันจาก preorder เป็น root

root\_val = preorder[self.pre\_idx]

self.pre\_idx += 1

root = Node(root\_val)

###### ถ้าไม่มีลูกแล้ว

if left == right:

return root

# สร้างซ้ายและขวาจาก inorder

root.left = arrayToTree(left, in\_index[root\_val] - 1)

root.right = arrayToTree(in\_index[root\_val] + 1, right)

return root

return arrayToTree(0, len(inorder) - 1)

# Traversal แบบต่าง ๆ เพื่อทดสอบผลลัพธ์

def inorder(self, node):

if node:

self.inorder(node.left)

print(node.data, end=" ")

self.inorder(node.right)

def preorder(self, node):

if node:

print(node.data, end=" ")

self.preorder(node.left)

self.preorder(node.right)

def postorder(self, node):

if node:

self.postorder(node.left)

self.postorder(node.right)

print(node.data, end=" ")