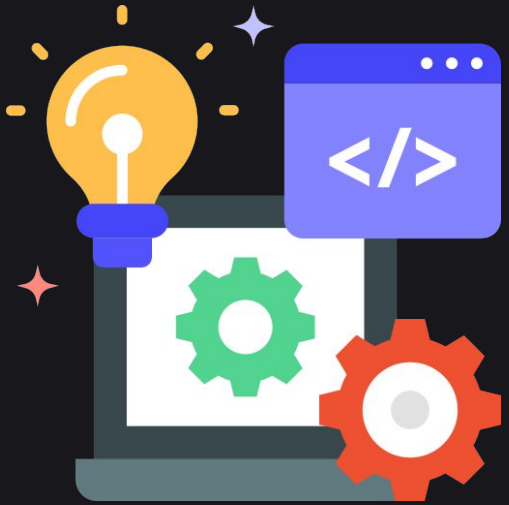
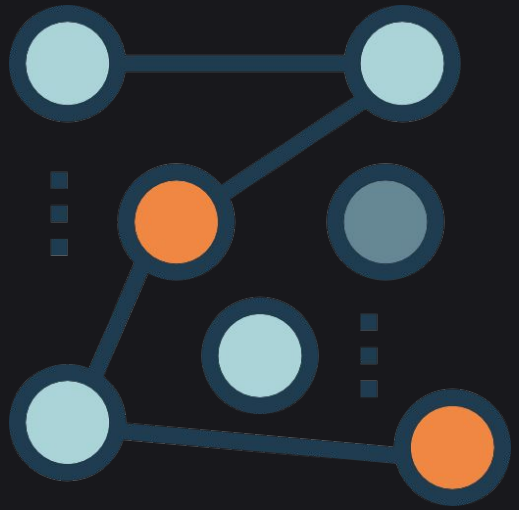
เอกสาร
โครงสร้างข้อมูลและอัลกอริทึม
ด้วย JavaScript

องค์ประกอบของโปรแกรมคอมพิวเตอร์



Program = Data Structure + Algorithm

(โครงสร้างข้อมูล + ขั้นตอนวิธี)



โครงสร้างข้อมูล (Data Structure)

โครงสร้างข้อมูล

คือ หน่วยข้อมูลย่อยหรือประเภทข้อมูลที่ถูกจัดวางในรูปแบบที่เหมาะสม โดยมีการนิยามความสัมพันธ์ภายในกลุ่มข้อมูลให้มีรูปแบบและข้อกำหนดที่ชัดเจน

ประโยชน์ของโครงสร้างข้อมูล

- ทำให้ข้อมูลมีระเบียบมากยิ่งขึ้น ง่ายต่อการนำไปใช้ตามวัตถุประสงค์ที่ต้องการ

- เมื่อจัดเก็บข้อมูลให้มีโครงสร้างก็จะส่งผลให้การทำงานของระบบเร็วขึ้น ถ้าจัดเก็บข้อมูลไม่ดี ไม่มีระเบียบก็จะส่งผลให้ระบบทำงานช้า

ประเภทของโครงสร้างข้อมูล

- โครงสร้างข้อมูลเชิงเส้น (Linear Data Structures)
- โครงสร้างข้อมูลไม่เชิงเส้น (Non-Linear Data Structures)

โครงสร้างข้อมูลเชิงเส้น

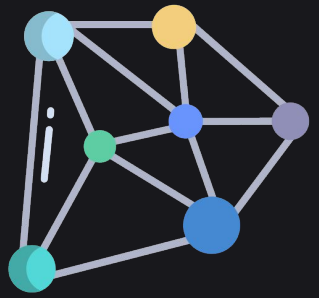
คือ โครงสร้างข้อมูลที่สมาชิกแต่ละตัวจะเชื่อมกับสมาชิกตัวถัดไปเพียงตัวเดียวและมีลำดับที่ต่อเนื่อง

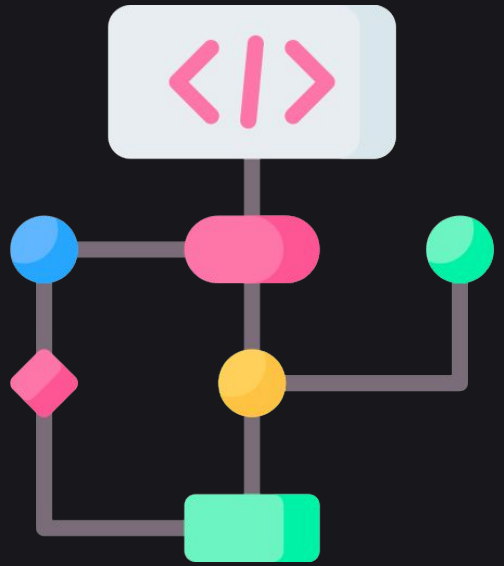
- อาร์เรย์ (Array)
- สแต็ก (Stack)
- ลิงค์ลิสต์ (Linked-List)
- คิว (Queue)

โครงสร้างข้อมูลไม่เชิงเส้น

คือ โครงสร้างที่ไม่มีคุณสมบัติของเชิงเส้น สามารถใช้แสดงความสัมพันธ์ของข้อมูลที่ซับซ้อนได้มากกว่าโครงสร้างข้อมูลแบบเชิงเส้น หมายถึงข้อมูลหนึ่งตัว มีความสัมพันธ์กับข้อมูลอื่นได้หลายตัว ตัวอย่าง เช่น

- ทรี (Tree)
- กราฟ (Graph)





อัลกอริทึม (Algorithm)

อัลกอริทึม (Algorithm)

อัลกอริทึม หรือ ขั้นตอนวิธี เป็นวิธีการแสดงลำดับขั้นตอนในการทำงาน หรือขั้นตอนในการแก้ปัญหา เช่น ขั้นตอนการชงกาแฟ

1. จัดเตรียมส่วนผสม (กาแฟ , น้ำตาล)
2. ต้มน้ำให้เดือด
3. นำส่วนผสมใส่ลงในแก้ว
4. เทน้ำร้อนใส่แก้ว
5. ผสมให้เข้ากัน

ขั้นตอนการทอดไข่เจียว

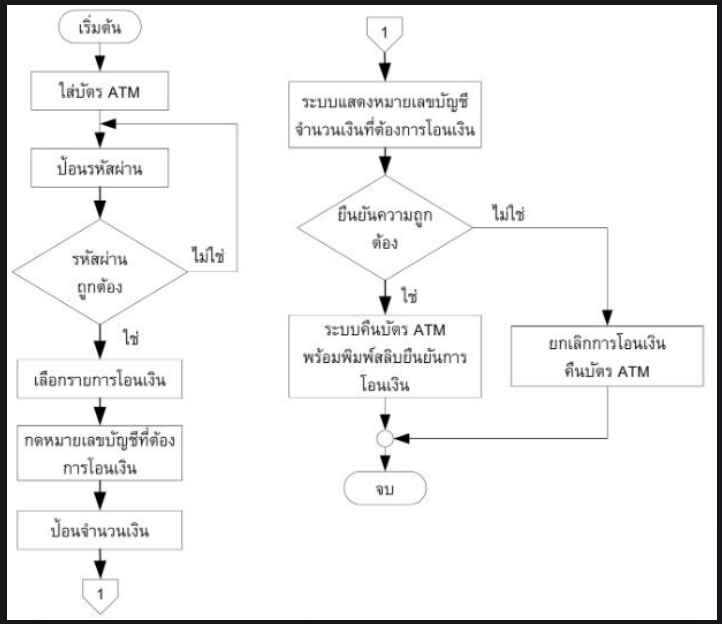
1. หยิบไข่ไก่
2. ตอกไข่ไก่ใส่ภาชนะ
3. ปรุงรส
4. ตีไข่
5. เปิดแก๊สและติดไฟ
6. ตั้งกระทะบนเตา
7. ใส่น้ำมันพืช
8. นำไข่ที่ปรุงรสแล้วใส่ในกระทะ
9. ทอดจนสุก
10. ตักใส่จาน

ขั้นตอนการโอนเงินที่ตู้ ATM

1. ใส่บัตร ATM
2. ป้อนรหัสบัตร ATM
3. เข้าสู่หน้าบริการ
4. เลือกเมนูโอนเงิน
5. กดหมายเลขบัญชีปลายทางที่ต้องการโอน
6. ป้อนจำนวนเงิน
7. ตรวจสอบข้อมูลบัญชีปลายทาง
8. กดตกลง
9. รับบัตรคืน
10. รับสลิปการโอนเงิน

ขั้นตอนการโอนเงินที่ตู้ ATM

1. ใส่บัตร ATM
2. ป้อนรหัสบัตร ATM
3. เข้าสู่หน้าบริการ
4. เลือกเมนูโอนเงิน
5. กดหมายเลขบัญชีปลายทางที่ต้องการโอน
6. ป้อนจำนวนเงิน
7. ตรวจสอบข้อมูลบัญชีปลายทาง
8. กดตกลง
9. รับบัตรคืน
10. รับสลิปการโอนเงิน



การวัดประสิทธิภาพอัลกอริทึม

การวัดประสิทธิภาพอัลกอริทึม

เมื่อมีการพัฒนาโปรแกรมขึ้นมาจะต้องมีการวัดผลการทำงานของโปรแกรมว่าทำงานถูกต้องและได้ผลลัพธ์ตามที่ต้องการหรือไม่

เมื่อทำงานถูกต้องแล้ว มีประสิทธิภาพดีหรือไม่ ตัวอย่าง เช่น ใช้ระยะเวลาในการประมวลผลนานเท่าใด ใช้หน่วยความจำมากเกินความจำเป็นหรือไม่

การวัดประสิทธิภาพอัลกอริทึม

การวัดประสิทธิภาพของอัลกอริทึมแบ่งออกเป็น 2 รูปแบบ

- การวิเคราะห์หน่วยความจำที่ใช้ประมวลผล (Space Complexity Analysis)
- การวิเคราะห์เวลาในการประมวลผล (Time Complexity Analysis)

การวัดประสิทธิภาพอัลกอริทึม

รูปแบบที่ 1 : การวิเคราะห์หน่วยความจำที่ใช้ประมวลผล (Space Complexity Analysis) คือ การวิเคราะห์หน่วยความจำทั้งหมดที่โปรแกรมใช้ประมวลผลอัลกอริทึม เพื่อให้ทราบถึงขนาดข้อมูลที่ต้องการป้อนหรือส่งข้อมูลเข้ามาให้อัลกอริทึมประมวลผลแล้วไม่เกิดข้อผิดพลาด

การวัดประสิทธิภาพอัลกอริทึม

การวิเคราะห์หน่วยความจำที่ใช้ประมวลผล มีองค์ประกอบอยู่ 3 ส่วน

1. Instruction Space คือ ขนาดหน่วยความจำที่จำเป็นต้องใช้ขณะคอมไพล์โปรแกรม
2. Data Space คือ ขนาดหน่วยความจำที่จำเป็นต้องใช้ในการเก็บข้อมูลประเภทตัวแปรและค่าคงที่ในการประมวลผลโปรแกรม
3. Environment Stack Space คือ หน่วยความจำที่ใช้สำหรับเก็บข้อมูลผลลัพธ์จากการประมวลผล

การวัดประสิทธิภาพอัลกอริทึม

รูปแบบที่ 2 : การวิเคราะห์เวลาในการประมวลผล (Time Complexity Analysis) เป็นการวิเคราะห์ขั้นตอนการประมวลผลของอัลกอริทึมในการประมาณการเวลาที่ใช้ประมวลผล

การวัดประสิทธิภาพอัลกอริทึม

หลักพิจารณาก่อนวิเคราะห์เวลาในการประมวลผล

- เมื่อประมวลผลโปรแกรมเดียวกัน คอมพิวเตอร์ที่สเปคดีกว่าจะประมวลผลได้เร็วกว่าคอมพิวเตอร์ที่สเปคต่ำหรือไม่
- เมื่อประมวลผลโปรแกรม จำนวนคำสั่งที่มีน้อยกว่าจะทำงานเร็วกว่าโปรแกรมที่มีคำสั่งมากกว่าหรือไม่
- ตัวแปรที่มีขนาดเล็กจะประมวลผลเร็วกว่าตัวแปรที่มีขนาดใหญ่หรือไม่

การวัดประสิทธิภาพอัลกอริทึม

Time Complexity Analysis

Input (ขนาดข้อมูล)	Time (ระยะเวลา)
10 ตัว	2 วินาที
100 ตัว	2.1 วินาที
1,000 ตัว	1 นาที
10,000 ตัว	15 นาที
100,000 ตัว	35 นาที

การวัดประสิทธิภาพอัลกอริทึม

ประเภทของเวลาที่ใช้ประมวลผลแบ่งออกเป็น 2 ประเภท ได้แก่

- Compile Time เวลาที่ใช้ตรวจไวยากรณ์ภาษา (Syntax) ของโค้ดโปรแกรมว่าเขียนถูกต้องตามโครงสร้างภาษาที่ใช้เขียนโปรแกรมหรือไม่
- Runtime เป็นเวลาที่เครื่องคอมพิวเตอร์ใช้ประมวลผลโปรแกรม ซึ่งขึ้นอยู่กับชนิดข้อมูล จำนวนตัวแปร และจำนวนลูปที่ใช้ภายในการพัฒนาโปรแกรมที่ประมวลผล

การวัดประสิทธิภาพอัลกอริทึม

อยากรู้ว่าโปรแกรมที่เขียนขึ้นมานั้นมีระยะเวลาทำงานเร็วหรือช้าจะต้องวัดประสิทธิภาพของอัลกอริทึม โดยวิเคราะห์จาก “ อัตราการเติบโตของฟังก์ชั่น ”

คือ การวิเคราะห์ประสิทธิภาพของอัลกอริทึมจะดูจากอัตราการเติบโตที่เกิดขึ้นภายในฟังก์ชั่น

การวัดประสิทธิภาพอัลกอริทึม

ถ้าไม่ต้องการวิเคราะห์อัตราการเติบโต
สามารถใช้การเขียนโปรแกรมเพื่อทดสอบ
ประสิทธิภาพของอัลกอริทึมได้

การวัดประสิทธิภาพอัลกอริทึม

มี 2 วิธีให้เลือกใช้

1.เขียนโปรแกรมทดสอบ

2.วิเคราะห์อัตราการเติบโต

ข้อเสียของการเขียนโปรแกรมทดสอบ

- ต้องเสียเวลามาเขียนโปรแกรมรวมถึงศึกษาไวยากรณ์ภาษาแล้วรันคำสั่งให้ผ่านจึงจะเห็นผลลัพธ์
- เวลาการทำงานโปรแกรมขึ้นอยู่กับปัจจัยภายนอก เช่น คอมพิวเตอร์ที่ใช้รัน , นักเขียนโปรแกรม , ภาษาคอมพิวเตอร์ที่ใช้เขียน

อัตราการเติบโตของฟังก์ชั่น

การวัดประสิทธิภาพอัลกอริทึม

อัลกอริทึม
คำนวณหาผลรวมของตัวเลข

การวัดประสิทธิภาพอัลกอริทึม

$$1+2+3+4+5+.....n = ?$$

ขั้นตอนการวิเคราะห์อัตราการเติบโต

- เขียนอัลกอริทึม
- เลือกคำสั่งตัวแทนที่ใช้
  (คำสั่งที่ถูกทำงานและแปรผันตามเวลาทำงาน)
- วิเคราะห์จำนวนครั้งที่คำสั่งทำงาน
- หาฟังก์ชั่นของจำนวนครั้งที่คำสั่งทำงานกับปริมาณข้อมูล

ขั้นตอนการวิเคราะห์อัตราการเติบโต

$$1+2+3+4+5+.....n = ?$$

ขั้นตอนการวิเคราะห์อัตราการเติบโต

$$1+2+3+4+5+.....n = ?$$

หาผลรวมของ 1+2+3 ต้องบวกทั้งหมด 2 ครั้ง

หาผลรวมของ 1+2+3+4 ต้องบวกทั้งหมด 3 ครั้ง

หาผลรวมของ 1+2+3+4+5 ต้องบวกทั้งหมด 4 ครั้ง

ขั้นตอนการวิเคราะห์อัตราการเติบโต

$$1+2+3+4+5+.....n = ?$$

หาผลรวมของ 1+2+3+....n ต้องบวกทั้งหมด n-1 ครั้ง

ขั้นตอนการวิเคราะห์อัตราการเติบโต

1+2+3+4+5+.....n = ?

หาผลรวมของ 1+2+3+....n ต้องบวกทั้งหมด n-1 ครั้ง

จำนวนครั้งที่ทำงาน คือ n-1

ขั้นตอนการวิเคราะห์อัตราการเติบโต

รูปแบบฟังก์ชั่น คือ f(n-1) หรือ O(n-1)

เขียนแบบลดรูป คือ O(n)

โดยเรียกอัลกอริทึมที่มีความเร็ว O(n) ว่า

“ Linear Time Complexity ”

วิเคราะห์การทำงาน

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum += j;
  }
}
```

1

```
for (i = 1; i < n; i++) {
  for (j = 3; j < n - 1; j++) {
    sum += j;
  }
}
```

2

คำสั่งใดทำงานได้เร็ว

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum += j;
  }
}
```

1

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      sum += j;
   }
}
```

1

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{j=0}^{n-1} n = n^2$$

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      sum += j;
   }
}
```

1

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{j=0}^{n-1} n = n^2$$

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      sum += j;
   }
}
```

1

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{j=0}^{n-1} n = n^2$$

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      sum += j;
   }
}
```

1

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{j=0}^{n-1} n = n^2$$

ตัวอย่างคำสั่งที่ 1

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      sum += j;
   }
}
```

1

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{j=0}^{n-1} n = n^2$$

ตัวอย่างคำสั่งที่ 2

```
for (i = 1; i < n; i++) {
  for (j = 3; j < n - 1; j++) {
    sum += j;
  }
}
```

$$\sum_{i=1}^{n-1}\sum_{j=3}^{n-2} 1 = \sum_{j=1}^{n-1} n-4$$

$$= (n-1)(n-4)$$

$$= n^2 - 5n + 4$$

ตารางแสดงอัตราการเติบโต

n	n^2	การเติบโต (เท่า)	$n^2 - 5n+4$	การเติบโต (เท่า)
10	100		54	
20	400	4	304	5.63
40	1600	4	1404	4.62
80	6400	4	6004	4.28
160	25600	4	24804	4.13
320	102400	4	100804	4.06
640	409600	4	406404	4.03
1280	1638400	4	1632004	4.02
2560	6553600	4	6540804	4.01

ตารางแสดงอัตราการเติบโต

n	n^2	การเติบโต (เท่า)	$n^2 - 5n+4$	การเติบโต (เท่า)
10	100		54	
20	400	4	304	5.63
40	1600	4	1404	4.62
80	6400	4	6004	4.28
160	25600	4	24804	4.13
320	102400	4	100804	4.06
640	409600	4	406404	4.03
1280	1638400	4	1632004	4.02
2560	6553600	4	6540804	4.01

ตารางแสดงอัตราการเติบโต

n	n^2	การเติบโต (เท่า)	$n^2 - 5n+4$	การเติบโต (เท่า)
10	100		54	
20	400	4	304	5.63
40	1600	4	1404	4.62
80	6400	4	6004	4.28
160	25600	4	24804	4.13
320	102400	4	100804	4.06
640	409600	4	406404	4.03
1280	1638400	4	1632004	4.02
2560	6553600	4	6540804	4.01

วิเคราะห์อัตราการเติบโต

รูปแบบฟังก์ชั่น คือ $f(n^2)$ หรือ $O(n^2)$

โดยเรียกอัลกอริทึมที่มีความเร็ว $O(n^2)$ ว่า

“ Quadratic Time Complexity”

วิเคราะห์อัตราการเติบโต

แล้วจะทราบได้อย่างไรว่า คำสั่งใดมีความ

เติบโตเร็ว หรือ เติบโตช้า ??

วิเคราะห์อัตราการเติบโต

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

- ถ้าได้ค่า 0 แสดงว่า f(n) โตช้ากว่า g(n)
- ถ้าได้ค่า ∞ แสดงว่า f(n) โตเร็วกว่า g(n)
- ถ้าได้ค่าคงที่ แสดงว่า f(n) เติบโตเท่ากับ g(n)

อัตราการเติบโต

$\log n , n , n \log n , n^2 , n^3 , 2^n$

อัตราการเติบโต

n^2 , $0.001n^2$, n^2-5n+4 , $5n^2+8$

อัตราการเติบโต

n^2, $0.001n^2$, n^2-5n+4 , $5n^2+8$

ตารางแสดงอัตราการเติบโต

n	n^2	การเติบโต (เท่า)	$n^2 - 5n+4$	การเติบโต (เท่า)
10	100		54	
20	400	4	304	5.63
40	1600	4	1404	4.62
80	6400	4	6004	4.28
160	25600	4	24804	4.13
320	102400	4	100804	4.06
640	409600	4	406404	4.03
1280	1638400	4	1632004	4.02
2560	6553600	4	6540804	4.01

การวิเคราะห์อัลกอริทึม

การวิเคราะห์อัลกอริทึม

การวิเคราะห์อัลกอริทึมเมื่อรับขนาดปัญหาเข้ามาทำงาน (n) สิ่งที่ต้องการคือระยะเวลาในการแก้ปัญหา

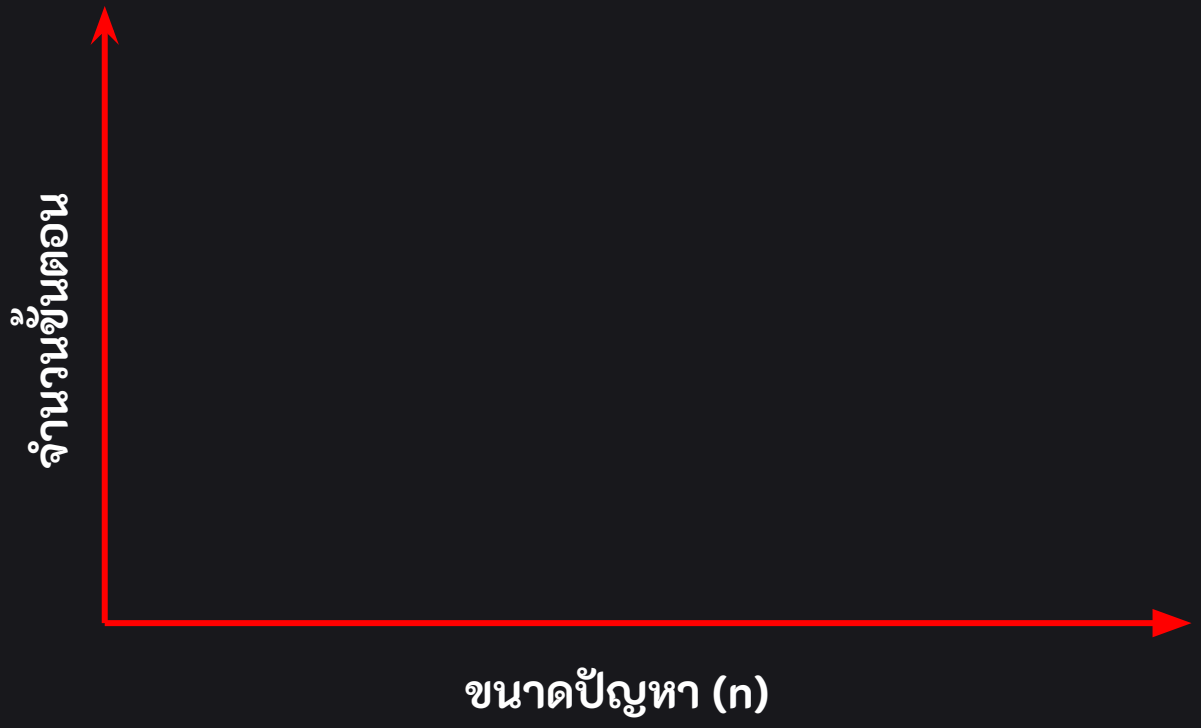
- ใช้เวลาน้อย -> ทำงานเร็ว
- ใช้เวลามาก -> ทำงานช้า

การวิเคราะห์อัลกอริทึม

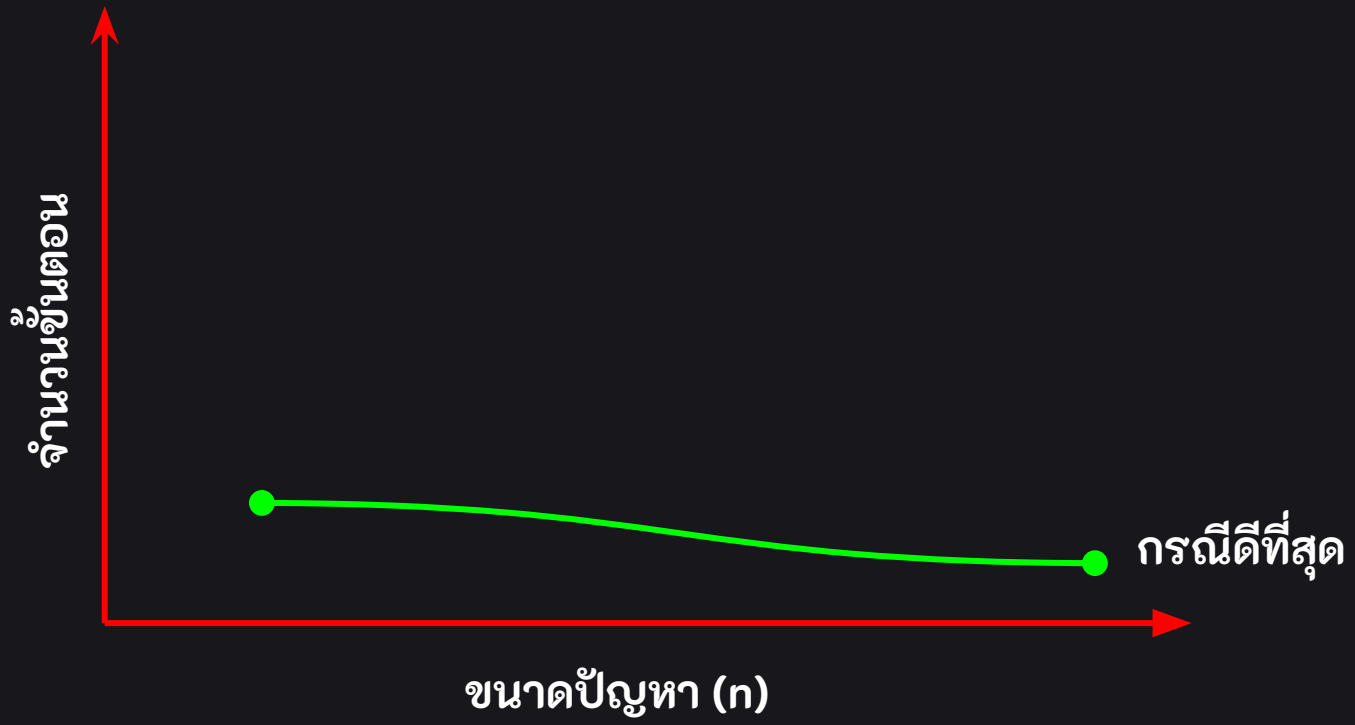
โดยอัลกอริทึมมีขั้นตอนการทำงานได้หลายรูปแบบเพื่อใช้แก้ปัญหาแบบเดียวกัน

- กรณีดีที่สุด (Best Case) คือ ใช้เวลาทำงานน้อย
- กรณีแย่ที่สุด (Worst Case) คือ ใช้เวลาทำงานมาก
- กรณีเฉลี่ย (Average Case) คือ ให้อัลกอริทึมทำงานหลายๆครั้ง โดยรับขนาดปัญหาที่แตกต่างกันมาหาเวลาเฉลี่ยในการทำงาน

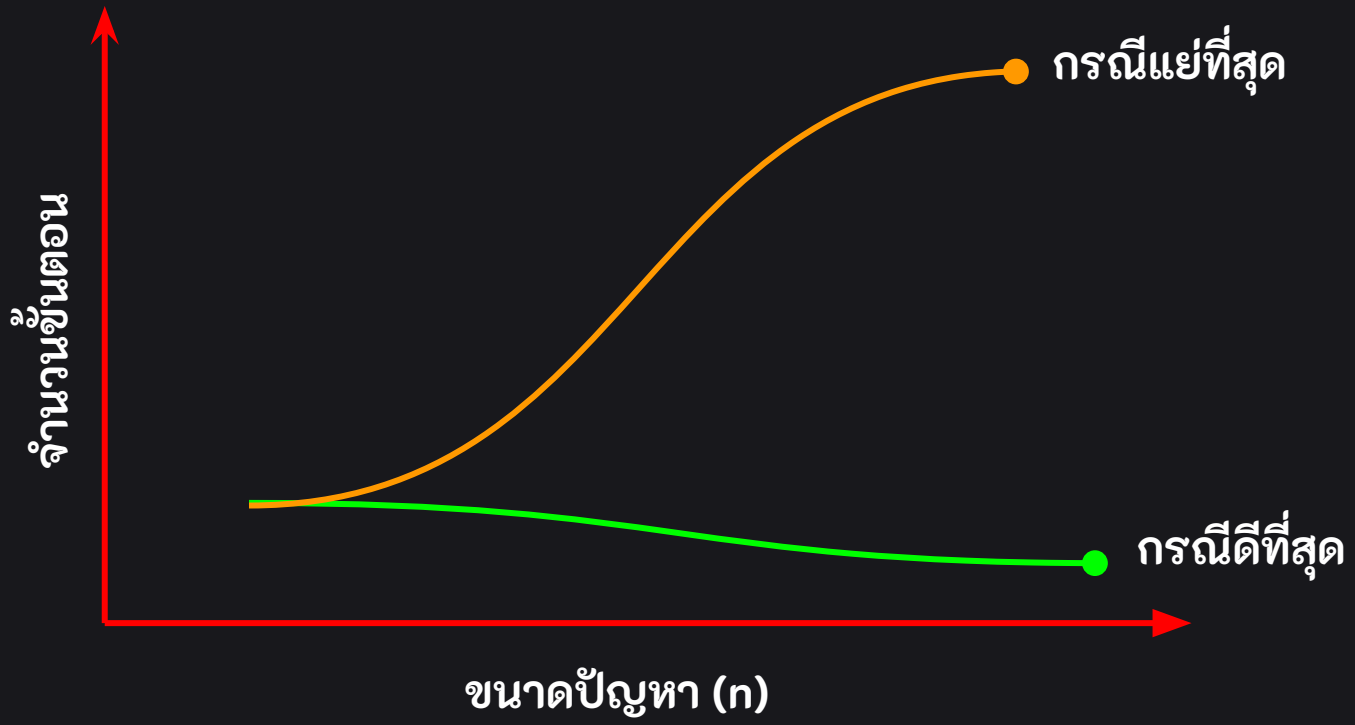
เส้นกราฟเวลาการเติบโตตามขนาดปัญหา (n)



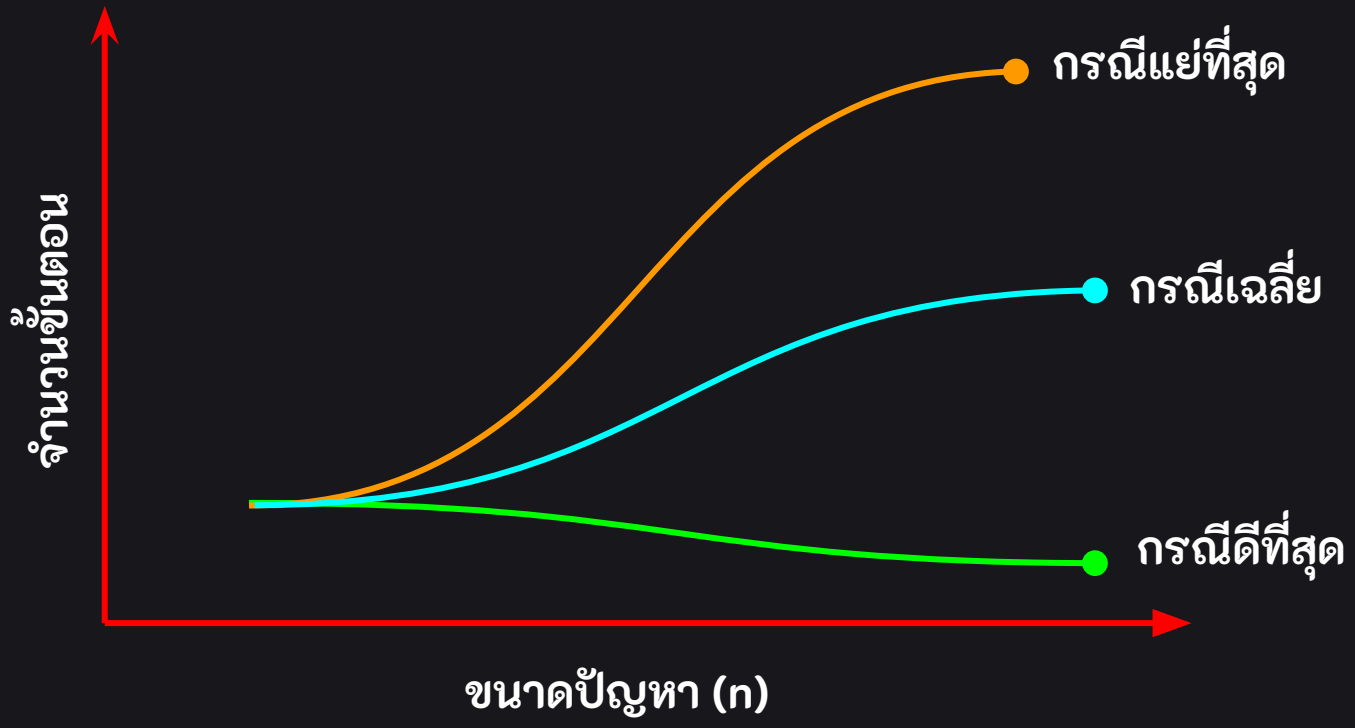
เส้นกราฟเวลาการเติบโตตามขนาดปัญหา (n)



เส้นกราฟเวลาการเติบโตตามขนาดปัญหา (n)



เส้นกราฟเวลาการเติบโตตามขนาดปัญหา (n)



การวิเคราะห์อัลกอริทึม

การวัดประสิทธิภาพด้วยอัตราการเติบโตนั้นเป็นการวิเคราะห์อัลกอริทึมโดยใช้เครื่องหมายมาเป็นเครื่องมือในการวัดประสิทธิภาพอัลกอริทึม เช่น Big-O , Big-Omega , Big-Teta เป็นต้น

- **กรณีที่แย่ที่สุด** จะใช้เครื่องหมาย Big-O
- **กรณีที่ดีที่สุด** จะใช้เครื่องหมาย Big-Ω (Omega)
- **กรณีเฉลี่ย** จะใช้เครื่องหมาย Big-Θ (Teta)

รู้จักกับ Big-O

รู้จักกับ Big-O

- วิธีที่เป็นมาตรฐานในการวิเคราะห์อัลกอริธึมในการระบุถึงเวลาที่อัลกอริธึมใช้เมื่อเทียบกับขนาดข้อมูลรับเข้า (Time Complexity)
- ระยะเวลาที่แย่ที่สุดที่คอมพิวเตอร์ต้องทำงานกับความซับซ้อนในการใช้อัลกอริทึม
- เป็นการวัดประสิทธิภาพเชิงเวลาในการวิเคราะห์การประมวลผลอัลกอริทึมในกรณีที่ใช้เวลานานที่สุด

รู้จักกับ Big-O

ทำไมต้องใช้ Big-O

ทำไมต้องใช้ Big-O

- ตัวเลข 100 จำนวน (0-99)

ลำดับที่ 1	ลำดับที่ 2	ลำดับที่ 3	ลำดับที่ 100
0	1	2	99

ทำไมต้องใช้ Big-O

- ตัวเลข 100 จำนวน (0-99)
- ต้องการค้นหาหมายเลข 99

ลำดับที่ 1	ลำดับที่ 2	ลำดับที่ 3	ลำดับที่ 100
0	1	2	99

ทำไมต้องใช้ Big-O

- ผลลัพธ์ที่ค้นต้องการถ้าไปตกที่ Worst Case คือ คอมพิวเตอร์ต้องวนลูป 100 ครั้งจึงจะค้นข้อมูลดังกล่าวเจอ

ลำดับที่ 1	ลำดับที่ 2	ลำดับที่ 3	ลำดับที่ 100
0	1	2	99

ทำไมต้องใช้ Big-O

- เพื่อลดโอกาสในการเกิดเหตุการณ์ดังกล่าว จึงต้องทำการ **วิเคราะห์ Big-O นั่นเอง!!!**

ลำดับที่ 1	ลำดับที่ 2	ลำดับที่ 3	ลำดับที่ 100
0	1	2	99

การนับตัวดำเนินการ (Operation Counts)

การนับตัวดำเนินการ (Operation Counts)

คือ การนับจำนวนครั้งการทำงานของตัวดำเนินการในอัลกอริทึม ซึ่งมี 4 รูปแบบ คือ

- แบบค่าคงที่ (Constant)
- แบบลูปลำดับ (Linear Loop)
- แบบลูปลอการิทึม (Logarithmic Loop)
- แบบลูปซ้อน (Nested Loop)

การนับตัวดำเนินการ (Operation Counts)

คือ การนับจำนวนครั้งการทำงานของตัวดำเนินการในอัลกอริทึม ซึ่งมี 4 รูปแบบ คือ

- แบบค่าคงที่ (Constant)
- แบบลูปลำดับ (Linear Loop)
- แบบลูปลอการิทึม (Logarithmic Loop)
- แบบลูปซ้อน (Nested Loop)

แบบค่าคงที่ (Constant)

พิจารณาขั้นตอนการดำเนินการของอัลกอริทึม ซึ่งแต่ละขั้นตอนจะนับการทำงานเป็น 1 ครั้ง ตัวอย่าง เช่น

count = 0 1

total = (1+n) * (n/2) 1

ผลรวมของฟังก์ชั่น f(n) = 1+1 = 2 หรือ O(f(n)) = O(2)

การนับตัวดำเนินการ (Operation Counts)

คือ การนับจำนวนครั้งการทำงานของตัวดำเนินการในอัลกอริทึม ซึ่งมี 4 รูปแบบ คือ

- แบบค่าคงที่ (Constant)
- แบบลูปลำดับ (Linear Loop)
- แบบลูปลอการิทึม (Logarithmic Loop)
- แบบลูปซ้อน (Nested Loop)

แบบลูปลำดับ (Linear Loop)

พิจารณาจากจำนวนการทำงานของตัวดำเนินการภายในลูปของอัลกอริทึม

```
function sum(n=3) {
  total = 0;
  for (i = 1; i < n; i++) {
    total = total + 1;
  }
}
```

แบบลูปลำดับ (Linear Loop)

```
function sum(n=3) {
  total = 0;
  for (i = 1; i < n; i++) {
    total = total + 1;
  }
}
```

i	i<n	total=total+1
1	✔	✔
2	✔	✔
3	✔	X
4	X	X
จำนวนครั้ง	3	2

แบบลูปลำดับ (Linear Loop)

```
function sum(n=3) {
  total = 0;
  for (i = 1; i < n; i++)
    total = total + 1;
  }
}
```

1

n

n-1

i	i<n	total=total+1
1	✔	✔
2	✔	✔
3	✔	X
4	X	X
จำนวนครั้ง	3	2

แบบลูปลำดับ (Linear Loop)

```
function sum(n=3) {
  total = 0;          1
  for (i = 1; i < n; i++)      n
    total = total + 1;         n-1
  }
}
```

ผลรวมของฟังก์ชั่น f(n) = 1+n+n-1 = 2n

หรือ O(f(n)) = O(2n)

i	i<n	total=total+1
1	✔	✔
2	✔	✔
3	✔	X
4	X	X
จำนวนครั้ง	3	2

การนับตัวดำเนินการ (Operation Counts)

คือ การนับจำนวนครั้งการทำงานของตัวดำเนินการในอัลกอริทึม ซึ่งมี 4 รูปแบบ คือ

- แบบค่าคงที่ (Constant)
- แบบลูปลำดับ (Linear Loop)
- แบบลูปลอการิทึม (Logarithmic Loop)
- แบบลูปซ้อน (Nested Loop)

แบบลูปลอการิทึม (Logarithmic Loop)

ทำงานคล้ายกับลูปแบบลำดับ แต่จะใช้ค่าตัวแปรทำหน้าที่ในการเพิ่มหรือลดด้วยการคูณหรือหารเป็นอัตราเท่าตัว

1	2	3	4	5	6	7	8

แบบลูปลอการิทึม (Logarithmic Loop)

ทำงานคล้ายกับลูปแบบลำดับ แต่จะใช้ค่าตัวแปรทำหน้าที่ในการเพิ่มหรือลดด้วยการคูณหรือหารเป็นอัตราเท่าตัว

1	2	3	4	5	6	7	8

$$2^3 = 8$$

แบบลูปลอการิทึม (Logarithmic Loop)

ทำงานคล้ายกับลูปแบบลำดับ แต่จะใช้ค่าตัวแปรทำหน้าที่ในการเพิ่มหรือลดด้วยการคูณหรือหารเป็นอัตราเท่าตัว

1	2	3	4	5	6	7	8

$$\log_2 8 = 3$$

แบบลูปลอการิทึม (Logarithmic Loop)

```
function calculate() {
  total = 0;
  for (i = 1; i < 10; i=i*2) {
    //
    //
    }
}
```

1

$\log_2 n+1$

$\log_2 n$

$\log_2 n$

ผลรวมของฟังก์ชั่น f(n) = $1+\log_2 n+1 + \log_2 n + \log_2 n$

= $3 \log_2 n+2$ หรือ $O(f(n)) = O(3 \log_2 n+2)$

แบบลูปลอการิทึม

```
function calculate() {
  total = 0;
  for (i = 1; i < 10; i=i*2) {
    //statement
   }
}
```

เพิ่มด้วยการคูณ			
รอบที่	ค่า i	for loop	statement
1	1	✔	✔
2	2	✔	✔
3	4	✔	✔
4	8	✔	✔
5	16	✔	x
จำนวนครั้ง		$\log_2 n+1$	$\log_2 n$

แบบลูปลอการิทึม

```
function calculate() {
  total = 0;
  for (i = 1; i < 10; i=i*2) {
    //statement
   }
}
```

เงื่อนไขทั้งหมด 5 ครั้ง ($\log_2 n+1$)

มากกว่าการทำงานในลูป 1 ครั้ง ($\log_2 n$)

เนื่องจากลูปรอบสุดท้ายมีค่าเป็นเท็จ

เพิ่มด้วยการคูณ			
รอบที่	ค่า i	for loop	statement
1	1	✔	✔
2	2	✔	✔
3	4	✔	✔
4	8	✔	✔
5	16	✔	x
จำนวนครั้ง		5	4

การนับตัวดำเนินการ (Operation Counts)

คือ การนับจำนวนครั้งการทำงานของตัวดำเนินการในอัลกอริทึม ซึ่งมี 4 รูปแบบ คือ

- แบบค่าคงที่ (Constant)
- แบบลูปลำดับ (Linear Loop)
- แบบลูปลอการิทึม (Logarithmic Loop)
- แบบลูปซ้อน (Nested Loop)

แบบลูปซ้อนลูป (Nested Loop)

มีลักษณะเป็นลูปซ้อนลูป พิจารณาการทำงานจากจำนวนลูปนอกและลูปใน

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
  }
}
```

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
    }
}
```

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
    }
}
```

กำหนดให้ n = 2

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
  }
}
```

กำหนดให้ n = 2

i	j	i<n	j<n	statement

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
  }
}
```

กำหนดให้ n = 2

i	j	i<n	j<n	statement
0	0	✔	✔	✔
0	1	x	✔	✔
0	2	x	✔	x
1	0	✔	✔	✔
1	1	x	✔	✔
1	2	x	✔	x
2	0	✔	x	x

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
  }
}
```

กำหนดให้ n = 2

i	j	i<n	j<n	statement
0	0	✔	✔	✔
0	1	x	✔	✔
0	2	x	✔	x
1	0	✔	✔	✔
1	1	x	✔	✔
1	2	x	✔	x
2	0	✔	x	x
จำนวนที่ทำ		n+1 = 3	n*(n+1)=6	n*n = 4

แบบลูปซ้อนลูป (Nested Loop)

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
      //statement
  }
}
```

แบบลูปซ้อนลูป (Nested Loop)

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
      //statement
  }
}
```

1

n+1

n(n+1) = $n^2 + n$

n x n = n^2

n x n = n^2

แบบลูปซ้อนลูป (Nested Loop)

```
total = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
      //statement
      //statement
  }
}
```

1

n+1

n(n+1) = $n^2 + n$

n^2

n^2

ผลรวมของฟังก์ชั่น f(n) = 1 + n+1 + n(n+1) + n^2+ n^2

$f(n) = 3n^2+2n+2$

การเขียนลดรูปฟังก์ชั่น

การเขียนลดรูปฟังก์ชั่น

- $O(f(n)) = 500$
- $O(f(n)) = 3 \log_2 n+2$
- $O(f(n)) = 3n^2+2n+2$

การเขียนลดรูปฟังก์ชั่น

- $O(500)$ → $O(1)$
- $3\log_2 n+2$ → $O(\log_2 n)$
- $3n^2+2n+2$ → $O(n^2)$

Big-O Notation

Big-O Notation

สัญลักษณ์ (Notation)	ชื่อฟังก์ชั่น
O(1)	Constant
O(log n)	Logarithm
O(n)	Linear
O(n log n)	Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Big-O Notation

สัญลักษณ์ (Notation)	ชื่อฟังก์ชั่น
O(1)	ค่าคงที่ (Constant)
O(log n)	ฟังก์ชั่นลอการิทึม (Logarithm)
O(n)	ฟังก์ชั่นเชิงเส้น (Linear)
O(n log n)	ฟังก์ชั่นลอการิทึมเชิงเส้น (Linearithmic)
$O(n^2)$	ฟังก์ชั่นกำลังสอง (Quadratic)
$O(n^3)$	ฟังก์ชั่นกำลังสาม (Cubic)
$O(2^n)$	ฟังก์ชั่นเอ็กซ์โพแนนเชียล (Exponential)
O(n!)	ฟังก์ชั่นแฟกทอเรียล (Factorial)

ตัวอย่างฟังก์ชั่น

- n^2+n
- $4n^2+3n+12$  $O(n^2)$
- $2n^2+7$

Big-O Notation

สัญลักษณ์ (Notation)	ชื่อฟังก์ชั่น
O(1)	Constant
O(log n)	Logarithm
O(n)	Linear
O(n log n)	Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

ตัวอย่างฟังก์ชั่น

- n^2+n
- $4n^2+3n+12$
- $2n^2+7$

$O(n^2)$

Quadratic

ตัวอย่างฟังก์ชั่น

- n^3+n^2+n
- $4n^3+3n+12$  $O(n^3)$
- $2n^3+7$

Big-O Notation

สัญลักษณ์ (Notation)	ชื่อฟังก์ชั่น
O(1)	Constant
O(log n)	Logarithm
O(n)	Linear
O(n log n)	Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

ตัวอย่างฟังก์ชั่น

- n^3+n^2+n
- $4n^3+3n+12$
- $2n^3+7$

$O(n^3)$

Cubic

ตารางเปรียบเทียบประสิทธิภาพ

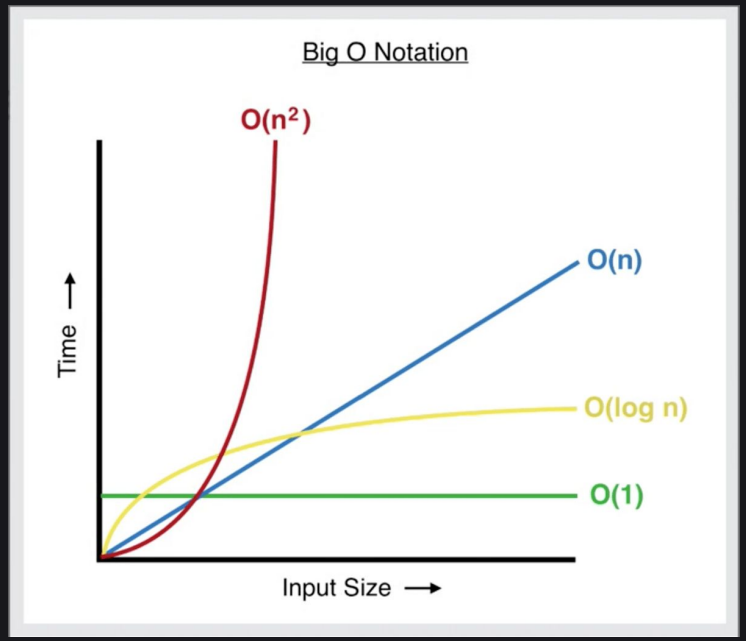
ชื่อฟังก์ชั่น	สัญลักษณ์	n = 1	n = 2	n = 4	n = 8	n = 16
Constant	O(1)	1	1	1	1	1
Logarithmic	O(log n)	1	1	2	3	4
Linear	O(n)	1	2	4	8	16
Linearithmic	O(n log n)	1	2	8	24	64
Quadratic	$O(n^2)$	1	4	16	64	256
Cubic	$O(n^3)$	1	8	64	512	4,096
Exponential	$O(2^n)$	2	4	16	256	65,536
Factorial	O(n!)	1	2	24	40,320	20,922,789,888,000

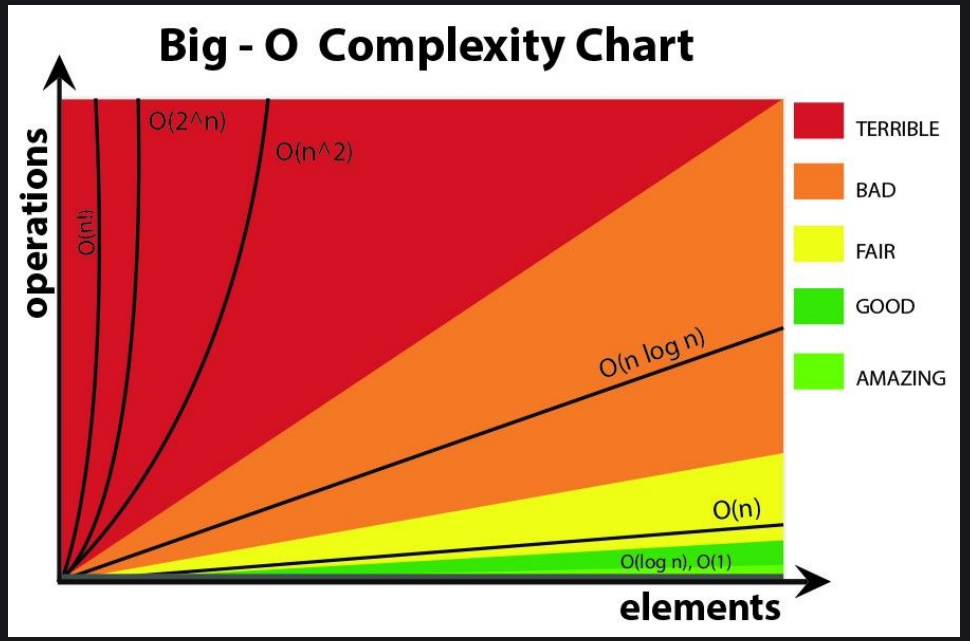
n = จำนวนข้อมูล

ตารางเปรียบเทียบประสิทธิภาพ

สัญลักษณ์	ชื่อฟังก์ชั่น	ประสิทธิภาพ
O(1)	Constant	ดีที่สุด
O(log n)	Logarithmic	
O(n)	Linear	
O(n log n)	Linearithmic	
$O(n^2)$	Quadratic	
$O(n^3)$	Cubic	
$O(2^n)$	Exponential	
O(n!)	Factorial	แย่ที่สุด

ตารางเปรียบเทียบประสิทธิภาพ





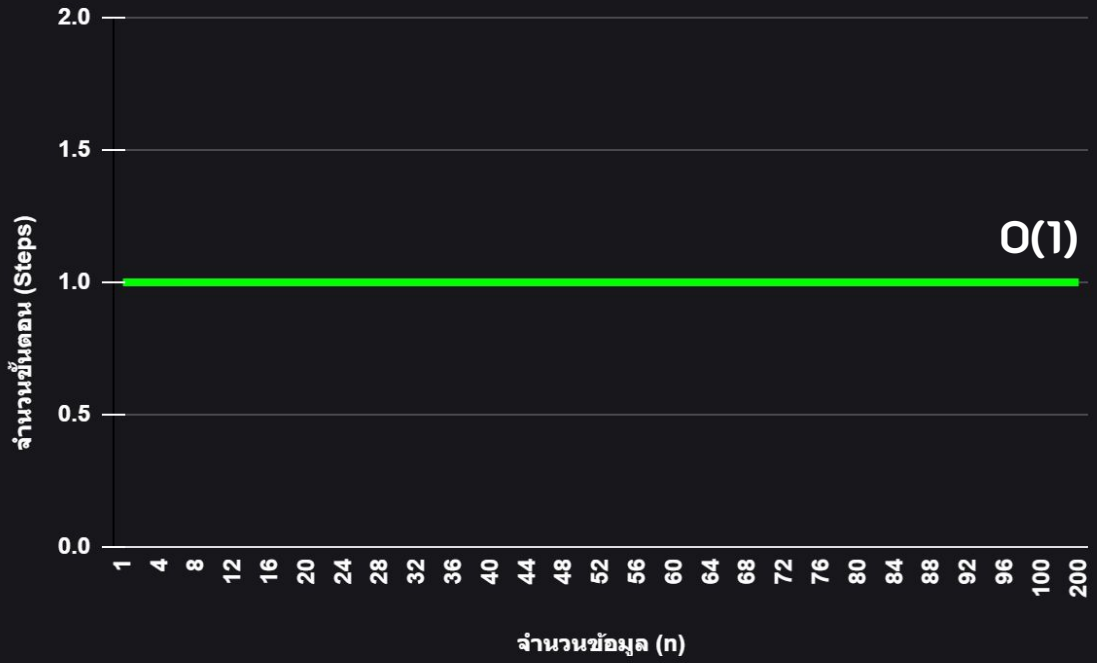
Constant: O(1)

Constant : O(1)

O(1) เป็น Big-O ที่ดีที่สุด ไม่ว่าปริมาณของข้อมูลจะมากเท่าใด ระยะเวลาในการประมวลผลจะไม่เปลี่ยนแปลง เช่น 1 จำนวน หรือ 1 ล้านจำนวน จะมีค่าเท่ากับ 1 เสมอ

Constant Time เวลาคงที่ไม่ขึ้นกับ input size (n)

Constant : O(1)



https://www.youtube.com/c/KongRuksiamOfficial/

https://www.facebook.com/KongRuksiamTutorial/

ตัวอย่างอัลกอริทึม

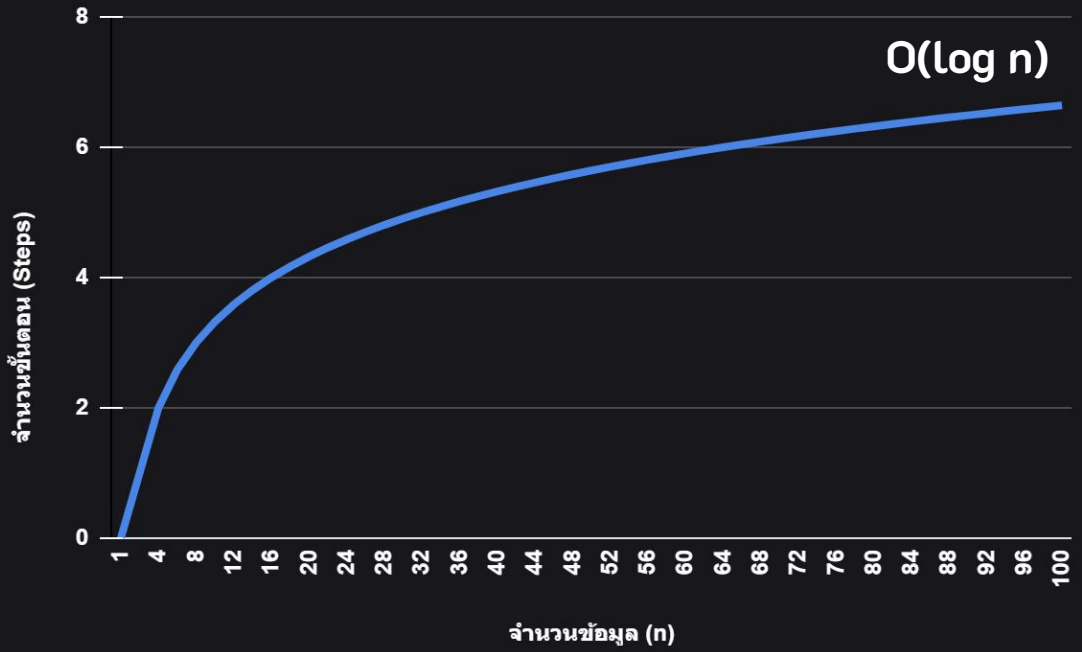
```
function firstElement(array){
    return array[0]// O(1)
}
let score = [70, 30, 15, 90, 50];
console.log(firstElement(score)); // 70
```

Logarithm Time : O(log n)

Logarithm Time : O(log n)

O (log n) เป็น Big-O ที่นำไปใช้ในการวนลูปและตัดจำนวนข้อมูลที่ไม่มีโอกาสเกิดขึ้นออกไปทีละครึ่งโดยแบ่งครึ่งข้อมูลไปเรื่อย ๆ ซึ่งสามารถพบการใช้งาน O(log n) ในการค้นหาข้อมูลที่เรียกว่า Binary Search ซึ่งเป็นการค้นหาแบบค่อย ๆ แบ่งครึ่งไปเรื่อย ๆ

Logarithm Time : O(log n)



ตัวอย่างอัลกอริทึม

```
function binarySearch(array, value){
  let firstIndex = 0;
  let lastIndex = array.length - 1;
  while (firstIndex <= lastIndex) {
    let middleIndex = Math.floor((firstIndex + lastIndex) / 2);
    if (array[middleIndex] === value) {
      return middleIndex;
    }
    if (array[middleIndex] > value) {
      lastIndex = middleIndex - 1;
    } else {
      firstIndex = middleIndex + 1;
    }
  }
  return -1;
};
```

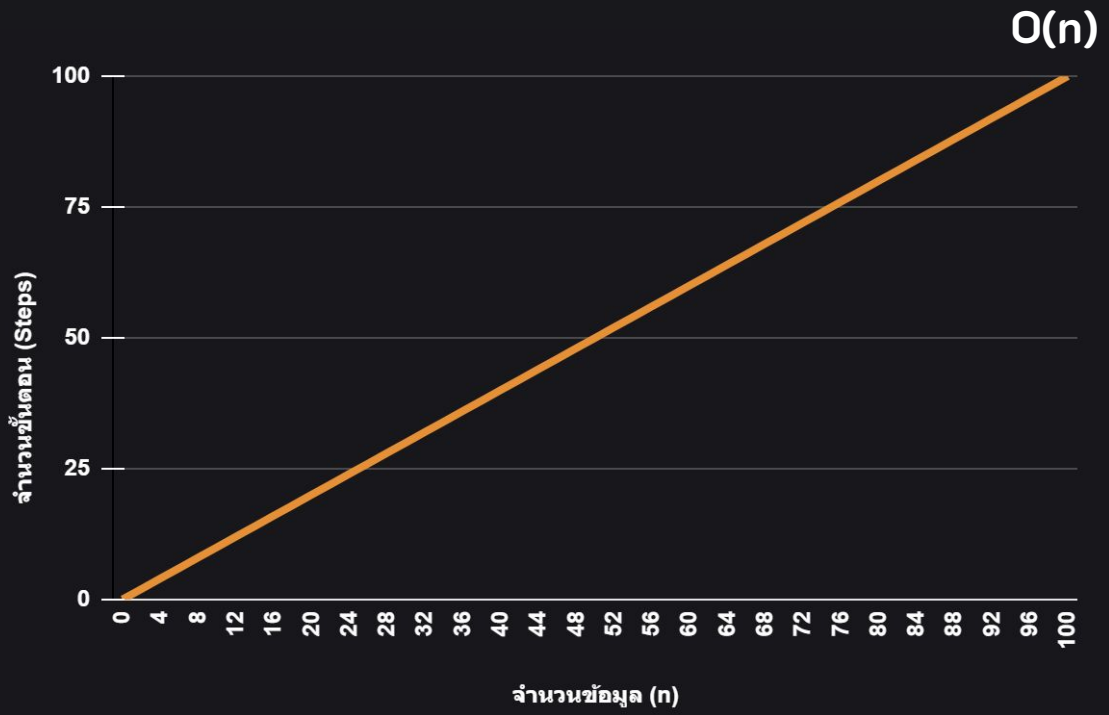
Binary Search

Linear Time: O(n)

Linear Time : O(n)

O(n) เป็น Big-O ที่ใช้ระยะเวลาทำงานอ้างอิงตามปริมาณข้อมูลที่มี ถ้ามีข้อมูลมากยิ่งใช้เวลามาก โดย Worst Case จะไม่เกินปริมาณข้อมูลที่ส่งมา

Linear Time : O(n)



https://www.youtube.com/c/KongRuksiamOfficial/

https://www.facebook.com/KongRuksiamTutorial/

ตัวอย่างอัลกอริทึม

```
function search(array, value) {
  for (let i = 0; i < array.length; i++) {
    if (array[i] === value) {
      return i;
    }
  }
  return -1
}
```

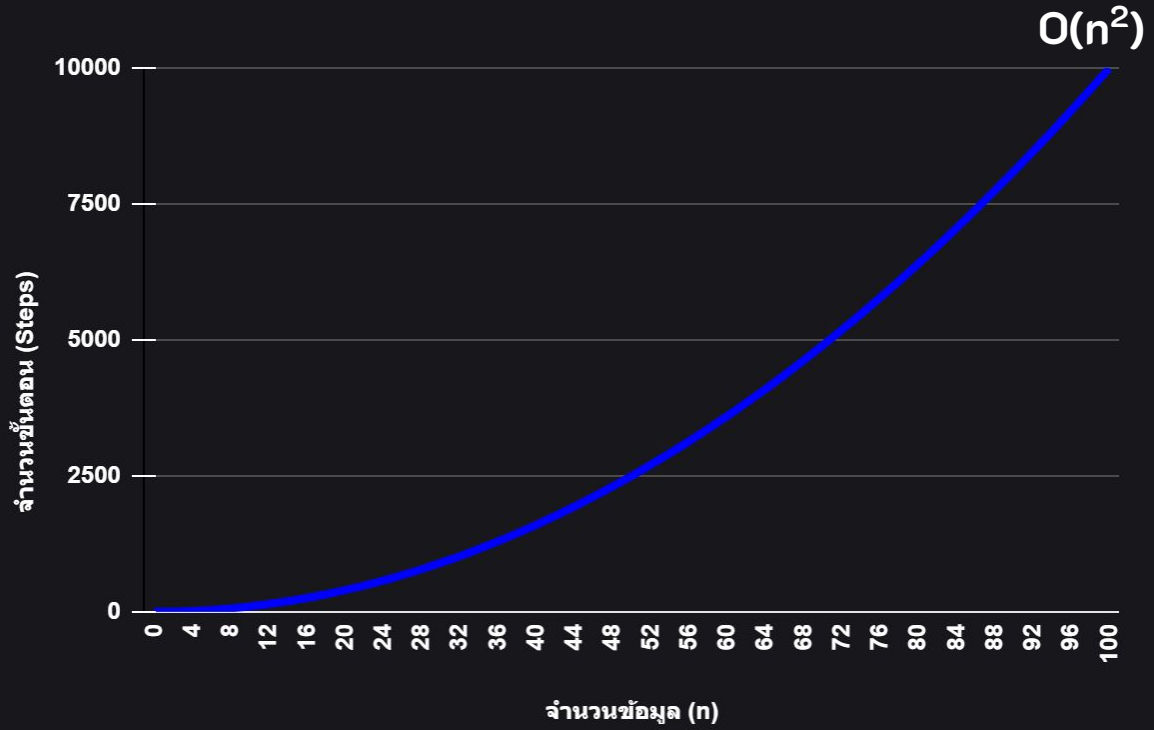
Sequential Search

Quadratic Time : $O(n^2)$

Quadratic Time : $O(n^2)$

$O(n^2)$ เป็น Big-O ที่อ้างอิงการทำงานตามการเพิ่มขนาดของ input (n) ที่ส่งเข้ามา 2 เท่า ส่งผลให้ระยะเวลาทำงานเพิ่มขึ้น 4 เท่า เช่น การใช้งานลูปซ้ำกัน 2 ชั้น เป็นต้น

Quadratic Time : $O(n^2)$



ตัวอย่างอัลกอริทึม

```
function matchElement(array){
    for (let i = 0; i < array.length; i++) {
      for (let j = 0; j < array.length; j++) {
        if (i !== j && array[i] === array[j]) {
          return `ตำแหน่งซ้ำกัน คือ ${i} และ ${j}`;
        }
      }
    }
    return "ไม่มีค่าซ้ำกันเลย";
};
```

ค้นหาข้อมูลที่มีค่าซ้ำกันในอาร์เรย์

ตัวอย่างอัลกอริทึม

```
const fruit = ["ส้ม","มะละกอ","มังคุด","ทุเรียน","ส้ม","แตงกวา"];
console.log(matchElement(fruit)); // "ตำแหน่งซ้ำกัน คือ 0 และ 4"
```

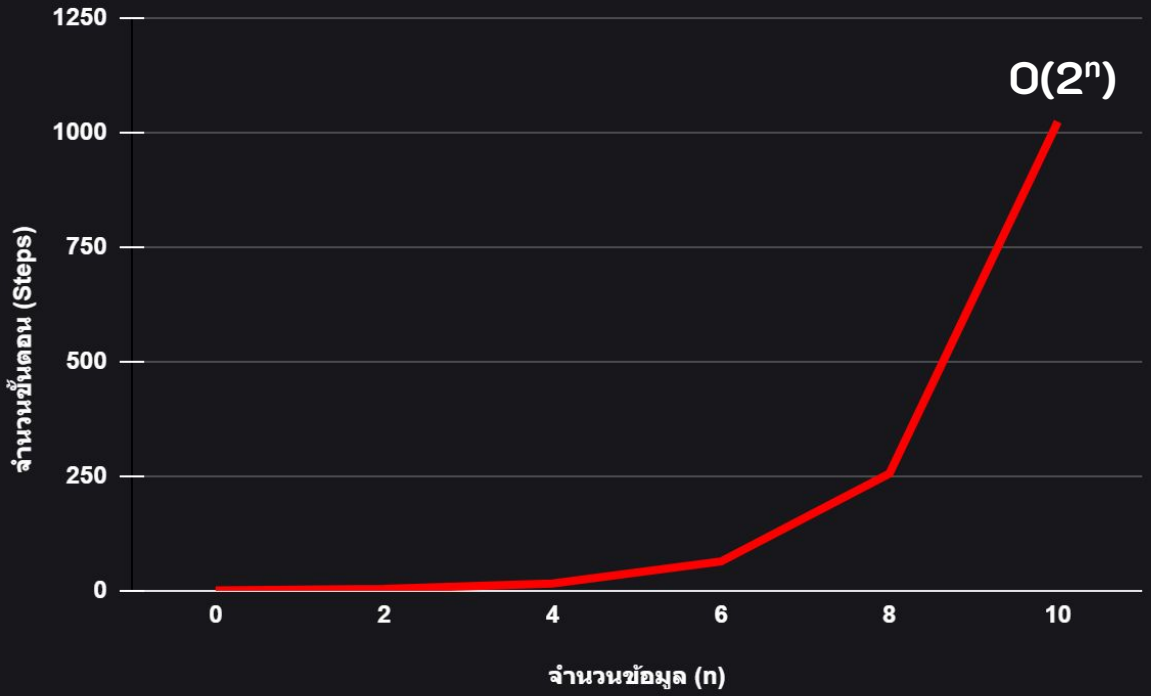
ค้นหาข้อมูลที่มีค่าซ้ำกันในอาร์เรย์

Exponential Time : $O(2^n)$

Exponential Time : O(2^n)

O(2^n) เป็น Big-O ที่ใช้จำนวนข้อมูลแค่นิดเดียว แต่ใช้เวลาประมวลผลนานและเพิ่มอัตราการเติบโตเป็นเท่าตัว ยกตัวอย่างถ้า n = 4 ก็ทำงาน 16 รอบ ถ้า n = 8 ใช้ไป 256 รอบ เช่น ฟังก์ชั่นฟีโบนัชชี เป็นต้น

Exponential Time : O(2ⁿ)



ตัวอย่างอัลกอริทึม

```
function fibonacci(n) {
  if (n <= 1) {
    return n;
  } else {
    return fibonacci(n - 2) + fibonacci(n - 1);
  }
}
console.log(fibonacci(6)); // 8
```

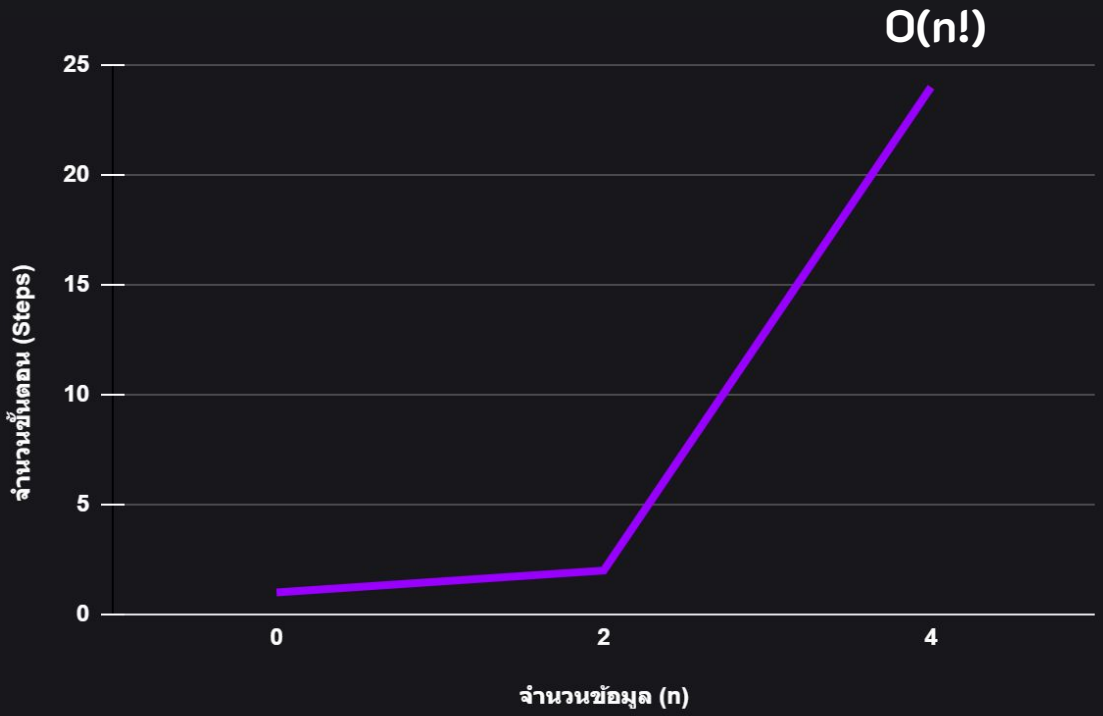
ฟังก์ชั่นฟีโบนัชชี

Factorial Time : O(n!)

Factorial Time : O(n!)

O(n!) เป็น Big-O เคสที่แย่ที่สุดใช้เวลาประมวลผลนาน
และเพิ่มอัตราการเติบโตเป็นเท่าตัว เช่น ถ้า n = 5
ก็มีค่าเท่ากับ 5! หรือ 5x4x3x2x1 ทำงาน 120 รอบ

Factorial Time : O(n!)



ตัวอย่างอัลกอริทึม

```
function factorial(n) {
  if (n == 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
console.log(factorial(5)); // 120
```

ฟังก์ชั่นหาค่าแฟกทอเรียล

ตารางเปรียบเทียบประสิทธิภาพ

สัญลักษณ์	ชื่อฟังก์ชั่น	ประสิทธิภาพ
O(1)	Constant	ดีที่สุด
O(log n)	Logarithmic	
O(n)	Linear	
O(n log n)	Linearithmic	
$O(n^2)$	Quadratic	
$O(n^3)$	Cubic	
$O(2^n)$	Exponential	
O(n!)	Factorial	แย่ที่สุด

การคำนวณ Big-O

ขั้นตอนการคำนวณ Big-O

1. หา Worst Case และ Constant
2. ลบส่วนที่เป็น Constant
3. แยกเคส Big - O ออกเป็นส่วนๆและลดรูปฟังก์ชั่น

**Constant ค่าคงที่ เช่น ตัวเลข

ลิงค์ลิสต์
(Linked-List)

โครงสร้างข้อมูลเชิงเส้น

คือ โครงสร้างข้อมูลที่สมาชิกแต่ละตัวจะเชื่อมกับสมาชิกตัวถัดไปเพียงตัวเดียวและมีลำดับที่ต่อเนื่อง

- อาร์เรย์ (Array)
- สแต็ก (Stack)
- ลิงค์ลิสต์ (Linked-List)
- คิว (Queue)

ลิงค์ลิสต์ (Linked-List)

เป็นโครงสร้างข้อมูลขั้นพื้นฐานในการเก็บข้อมูล โดยมีโครงสร้างแบบเชิงเส้นสามารถเพิ่ม-ลดขนาดการเก็บข้อมูลได้ตามความต้องการ

โดยการเก็บข้อมูลนั้นจะเก็บลงในโหนด (Node) และนำข้อมูลแต่ละโหนดมาเรียงต่อกันเป็นลิสต์โดยใช้ ลิงค์ (Link) หรือ พอยเตอร์ (Pointer) เป็นตัวเชื่อมโยงแต่ละโหนดเข้าด้วยกัน

องค์ประกอบของโหนด (Node)

องค์ประกอบของโหนด (Node)

- Data คือ ส่วนที่เก็บข้อมูล

องค์ประกอบของโหนด (Node)

- Data คือ ส่วนที่เก็บข้อมูล
- Next / Pointer คือ ส่วนที่เชื่อมโยงแต่ละโหนดเข้าด้วยกัน

ประเภทของลิงค์ลิสต์ (Linked-List)

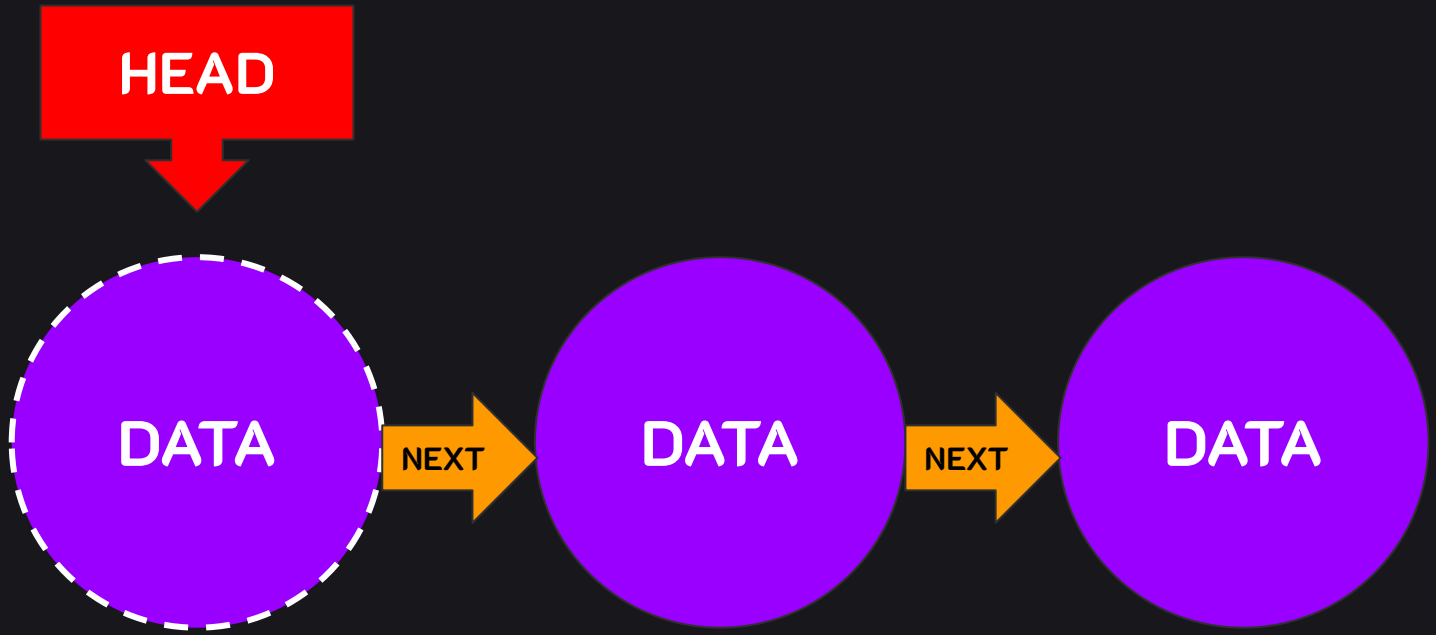
- แบบทิศทางเดียว (Singly Linked-List)
- แบบสองทิศทาง (Doubly Linked-List)

ลิงค์ลิสต์แบบทิศทางเดียว

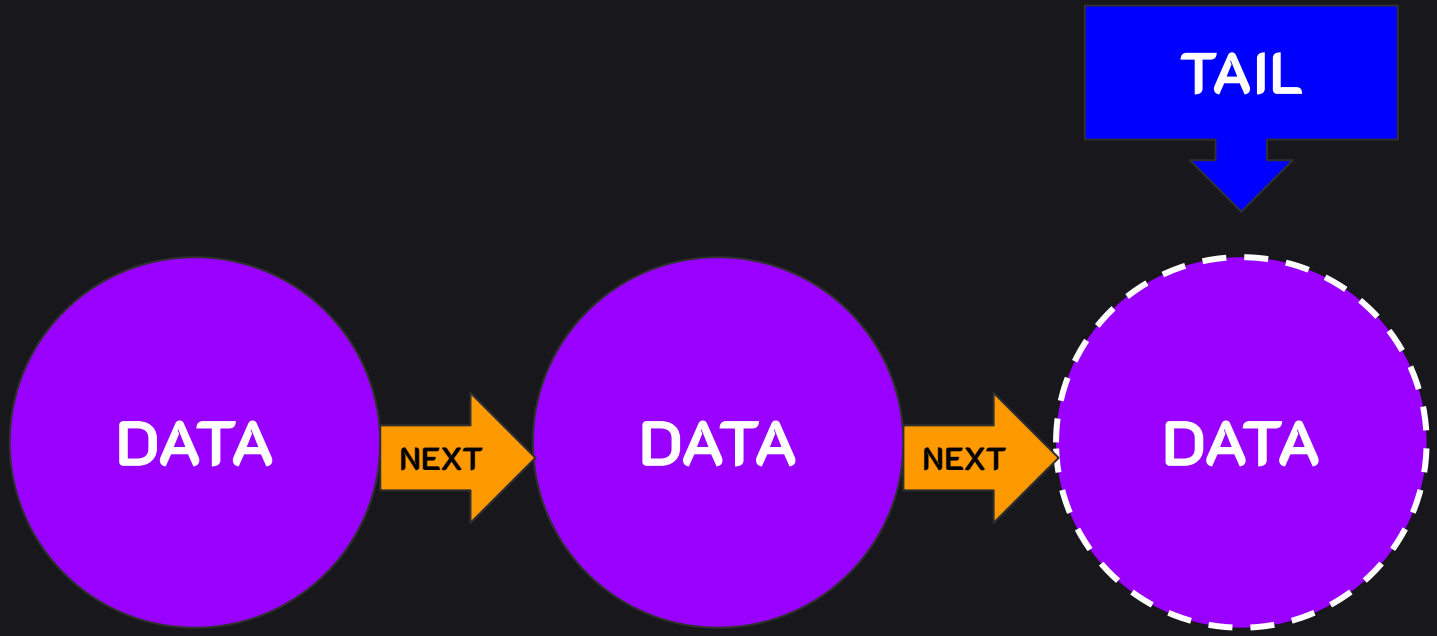
ลิงค์ลิสต์แบบทิศทางเดียว (Singly Linked-List) จะทำงานในทิศทางเดียวโดยมีจุดเริ่มต้นที่โหนดหัว (Head) ไปยังโหนดสุดท้าย (Tail) โดยไม่สามารถทำงานแบบย้อนกลับได้

ลิงค์ลิสต์แบบทิศทางเดียว

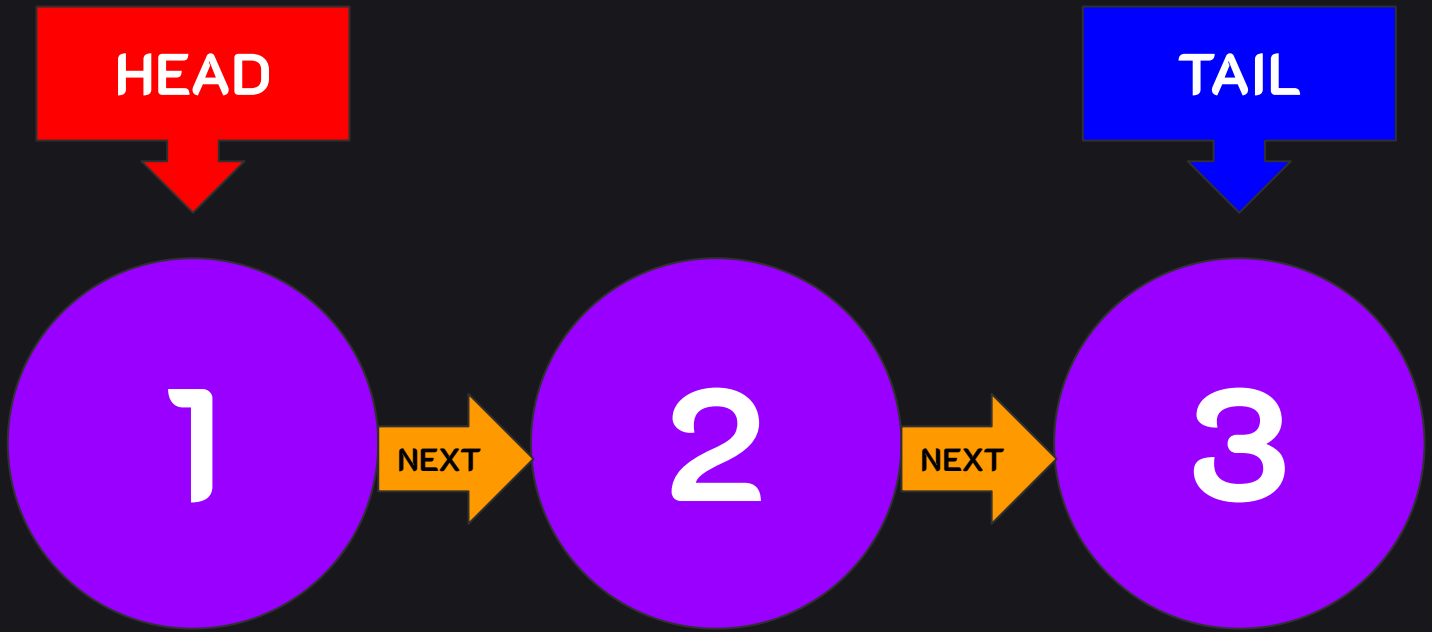
ลิงค์ลิสต์แบบทิศทางเดียว



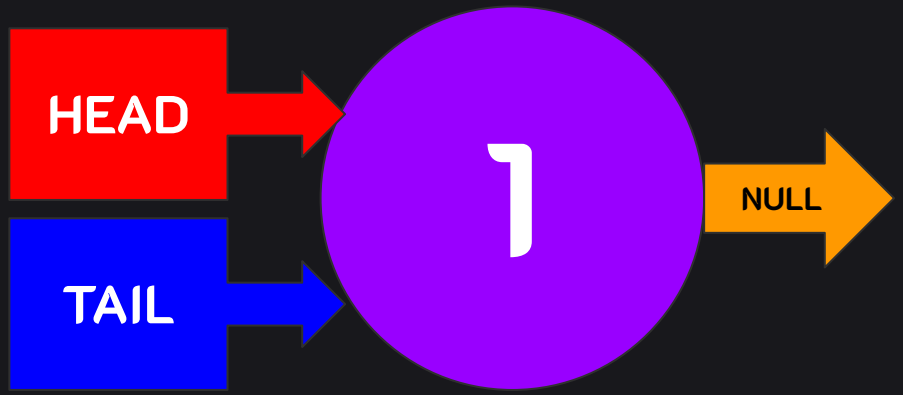
ลิงค์ลิสต์แบบทิศทางเดียว



ลิงค์ลิสต์แบบทิศทางเดียว

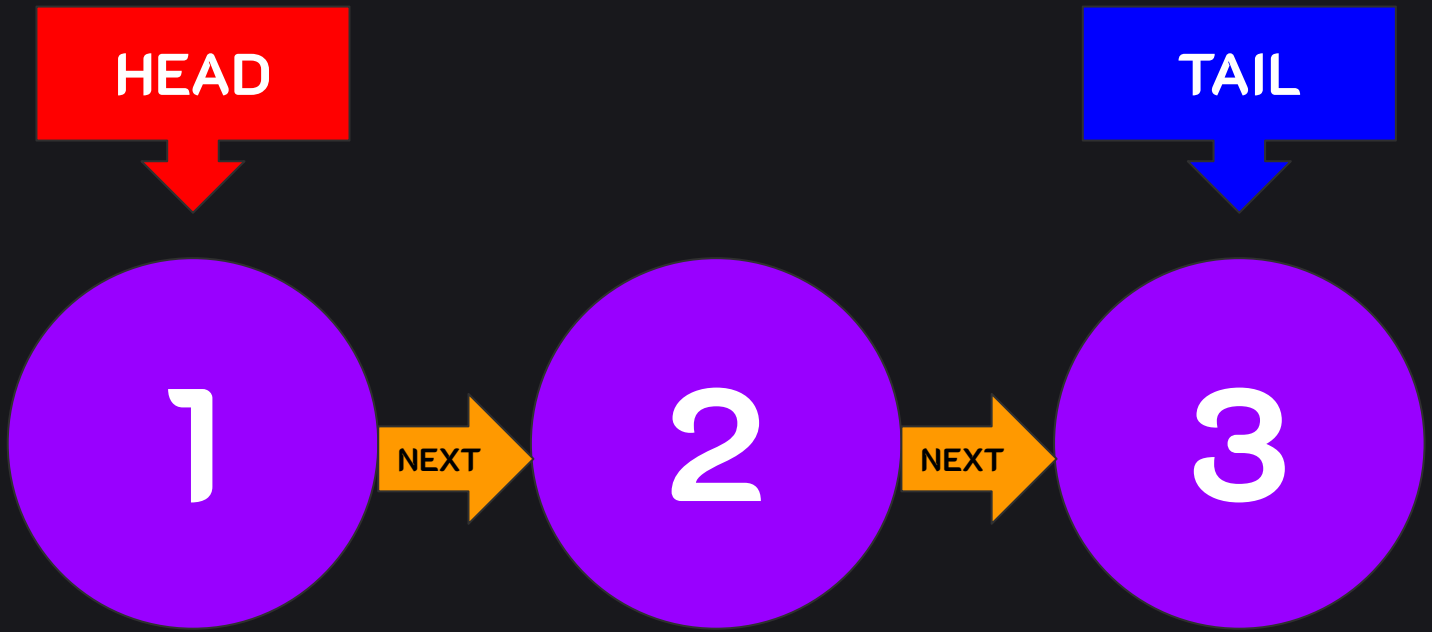


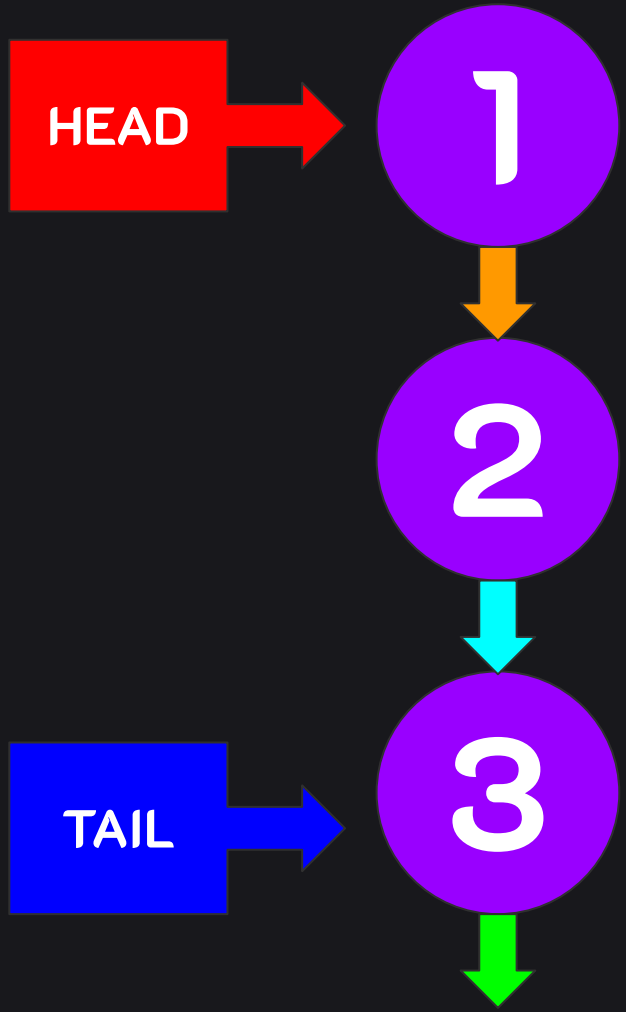
ลิงค์ลิสต์แบบทิศทางเดียว




```
{
    value : 1,
    next : null
}
```

ลิงค์ลิสต์แบบทิศทางเดียว






```
{
    value : 1,
        next : {
            value: 2,
            next : {
                value:3,
                next : null
            }
        }
}
```

จัดการลิงค์ลิสต์

จัดการลิงค์ลิสต์ (Linked-List)

เป็นโครงสร้างข้อมูลขั้นพื้นฐานในการเก็บข้อมูล โดยมีโครงสร้างแบบเชิงเส้นสามารถเพิ่ม-ลดขนาดการเก็บข้อมูลได้ตามความต้องการ

โดยการเก็บข้อมูลนั้นจะเก็บลงในโหนด (Node) และนำข้อมูลแต่ละโหนดมาเรียงต่อกันเป็นลิสต์โดยใช้ ลิงค์ (Link) หรือ พอยเตอร์ (Pointer) เป็นตัวเชื่อมโยงแต่ละโหนดเข้าด้วยกัน

จัดการลิงค์ลิสต์ (Linked-List)

เมธอด (Method)	คำอธิบาย
push(value)	เพิ่มข้อมูลเข้าไปที่ลำดับสุดท้ายของลิงค์ลิสต์
pop()	ดึงข้อมูลลำดับสุดท้ายออกจากลิงค์ลิสต์
unshift(value)	เพิ่มข้อมูลเข้าไปที่ลำดับแรกของลิงค์ลิสต์
shift()	ดึงข้อมูลลำดับแรกออกจากลิงค์ลิสต์

จัดการลิงค์ลิสต์ (Linked-List)

เมธอด (Method)	คำอธิบาย
get (index)	ดึงข้อมูลจาก Index ที่อ้างอิง
set(index,value)	กำหนดข้อมูลลงไปใน index ที่อ้างอิง
insert(index,value)	แทรกข้อมูลลงไปใน index ที่อ้างอิง
remove(index)	ลบข้อมูลออกจากลิงค์ลิตส์ตาม index ที่อ้างอิง
reverse()	สลับข้อมูลในลิงค์ลิสต์จากหน้าไปหลัง จากหลังไปหน้า

สแต็ก
(Stack)

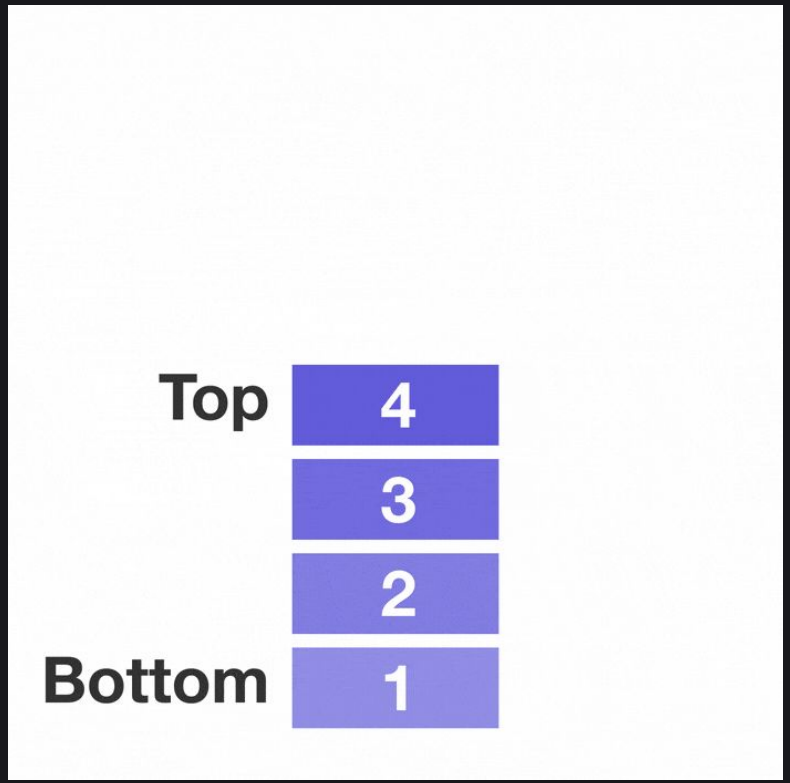
สแต็ก (Stack)

เป็นโครงสร้างข้อมูลที่มีการเก็บข้อมูลแบบลำดับรูปแบบการจัดเก็บข้อมูลใน Stack เป็นรูปแบบ **เข้าก่อนออกทีหลัง (Last In , First Out : LIFO)** หมายถึง ข้อมูลที่เข้าไปก่อนจะอยู่ด้านล่าง ข้อมูลที่เข้าหลังสุดจะอยู่ด้านบน

แนวคิดของสแต็ก (Stack)

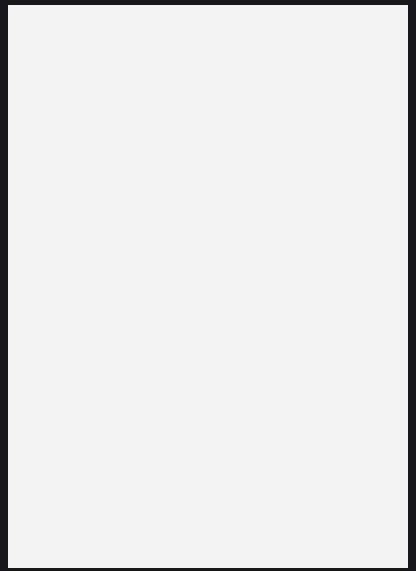
คือ การนำเอาข้อมูลมาเรียงซ้อนกันเป็นชั้นๆไปเรื่อยๆในพื้นที่ของสแต็ก (Stack) โดยเรียงจากล่างขึ้นบน สมาชิกหรือข้อมูลที่อยู่ด้านบนสุดจะเรียกว่า **“Top Stack”**

การทำงานของสแต็ก (Stack)



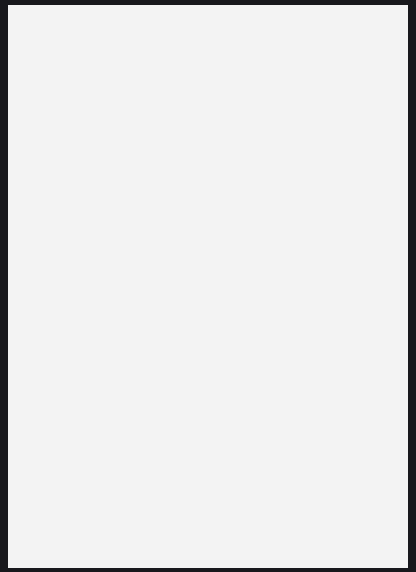
- Push การนำสมาชิกมาใส่ไว้บนสุดของสแต็ก (Top Stack)
- Pop การนำสมาชิกบนสุดออกไปจากสแต็ก

การทำงานของสแต็ก (Stack)



Empty Stack

การทำงานของสแต็ก (Stack)

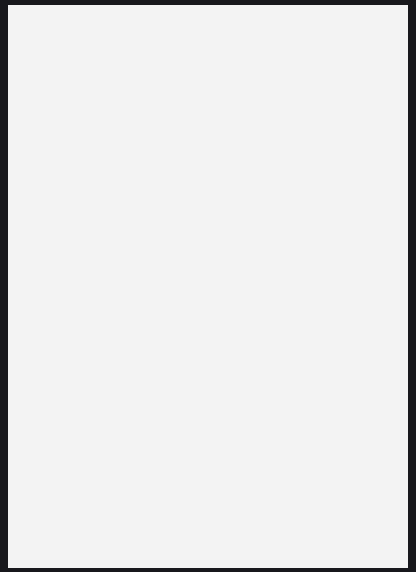


Empty Stack

Node

ข้อมูลที่ต้องการ
จัดเก็บลงใน Stack

การทำงานของสแต็ก (Stack)

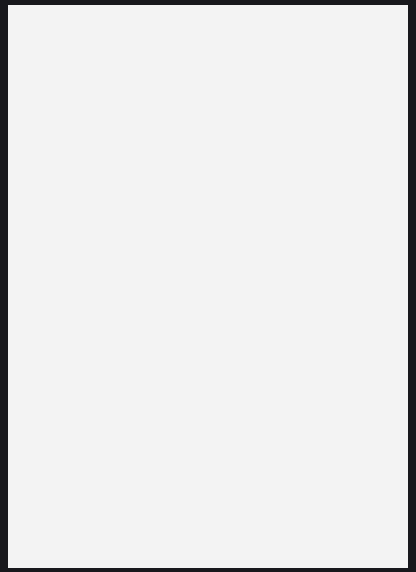


Empty Stack

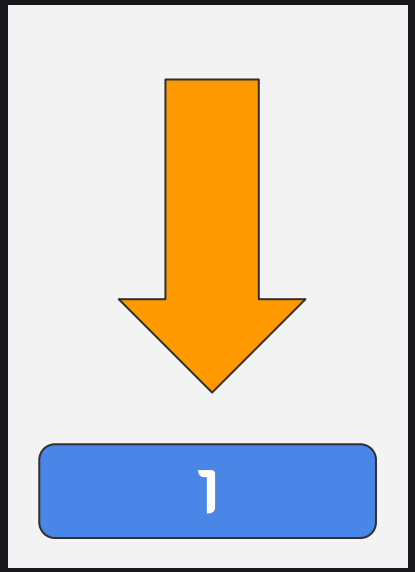
1

ข้อมูลที่ต้องการ
จัดเก็บลงใน Stack

การทำงานของสแต็ก (Stack)

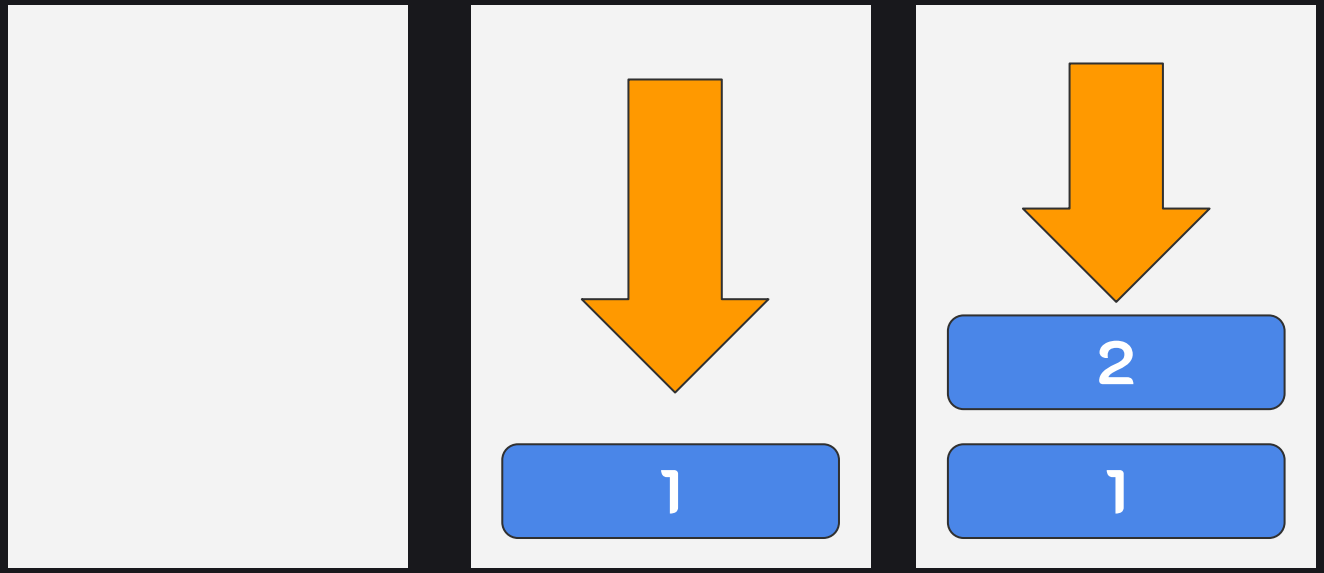


Empty Stack



Push

การทำงานของสแต็ก (Stack)

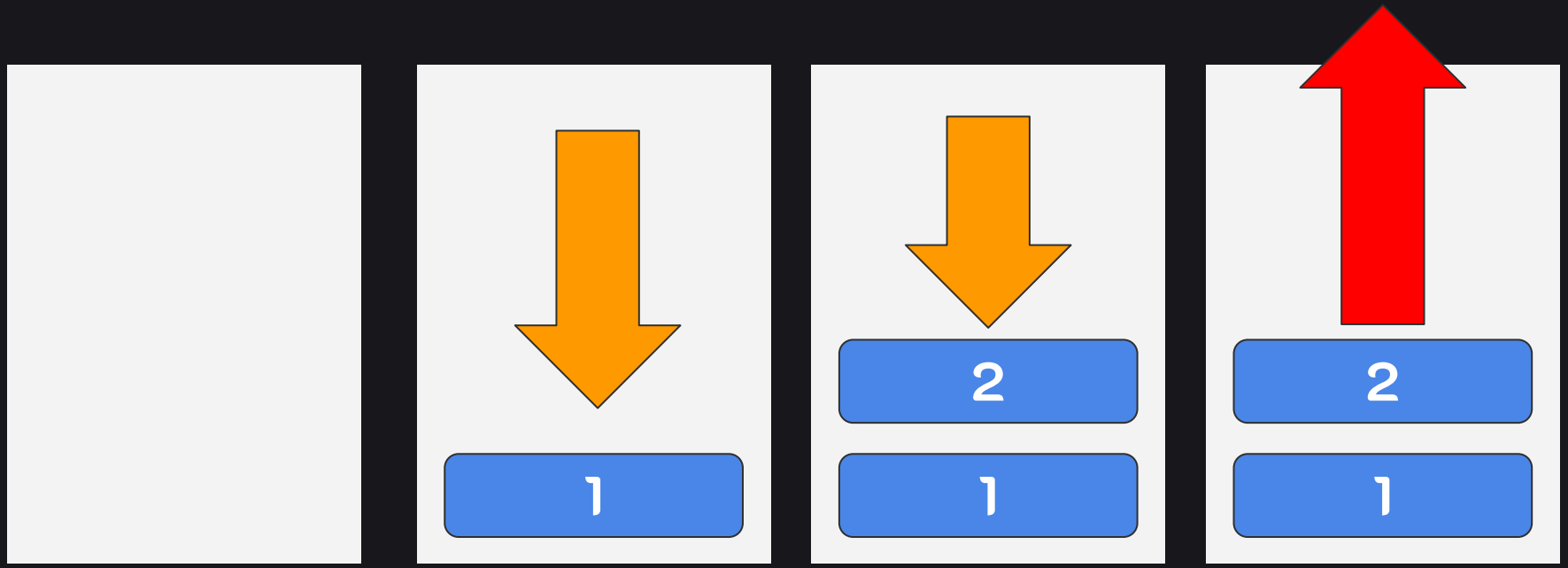


Empty Stack

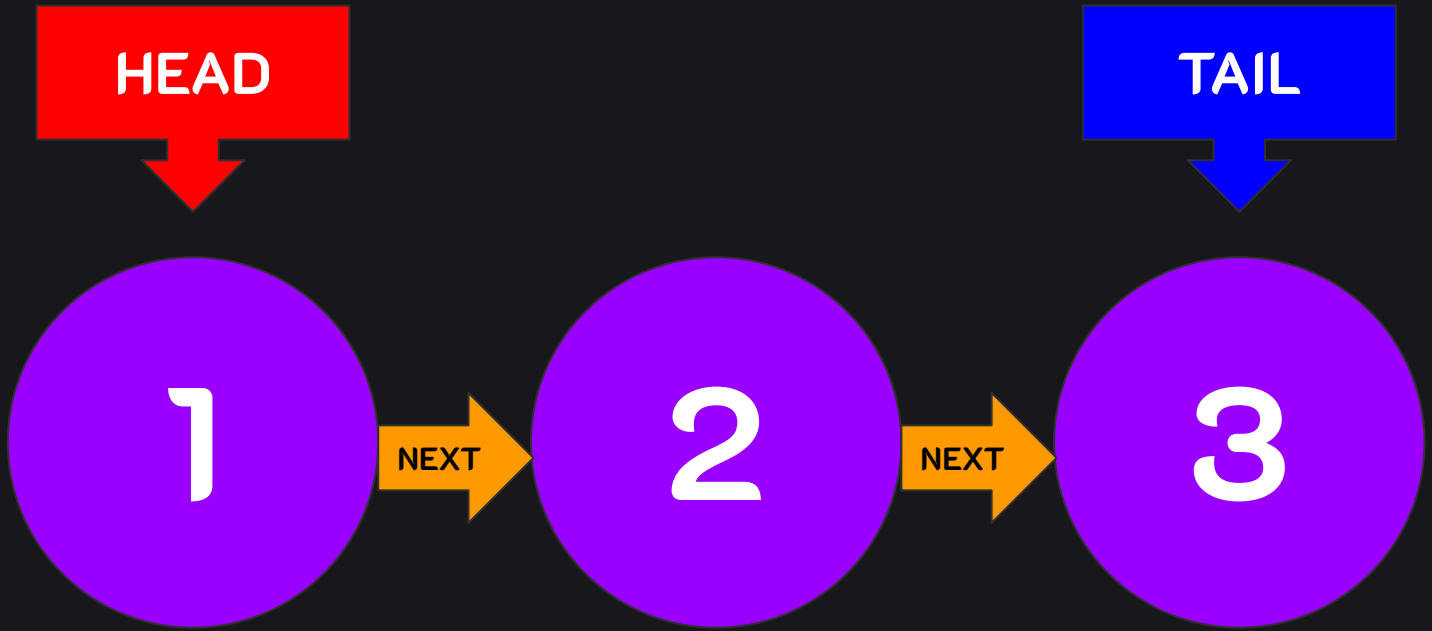
Push

Push

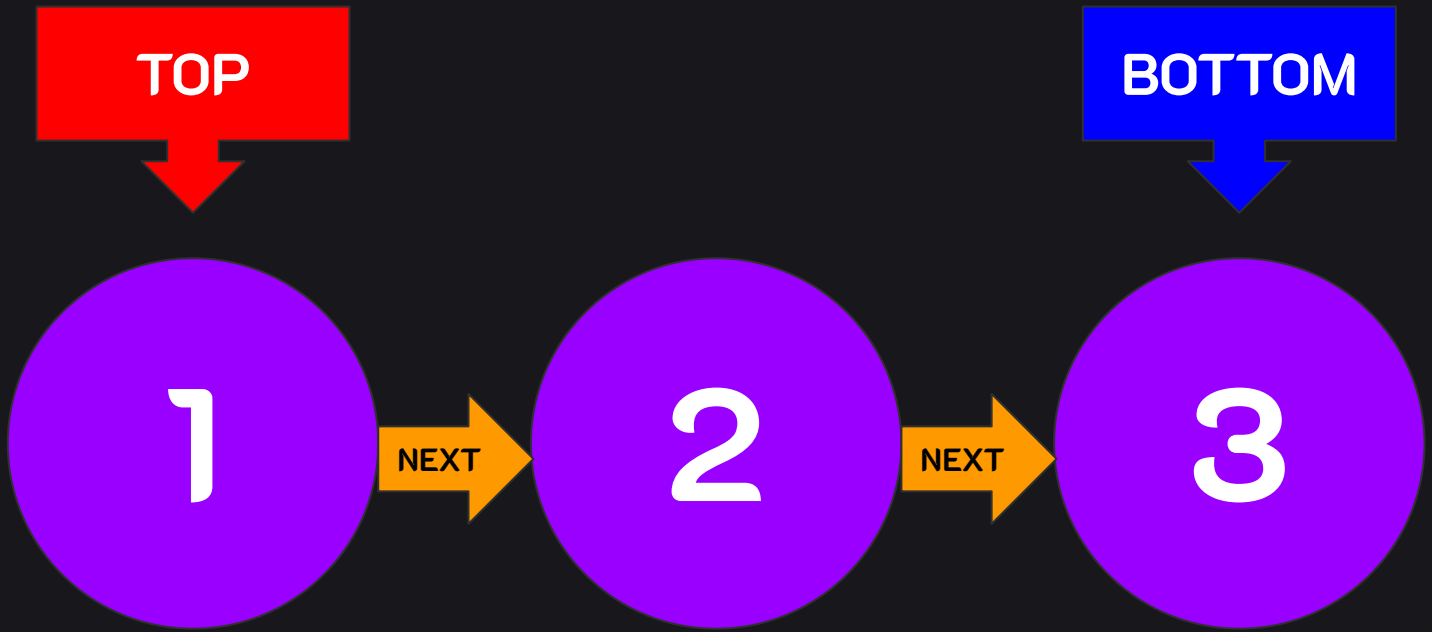
การทำงานของสแต็ก (Stack)

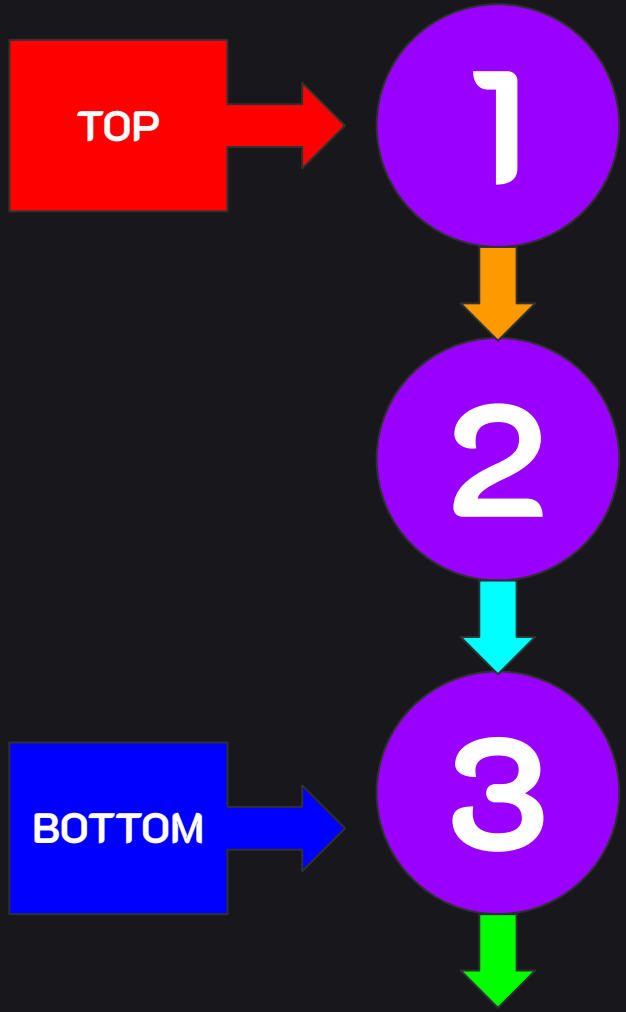


การทำงานของลิงค์ลิสต์

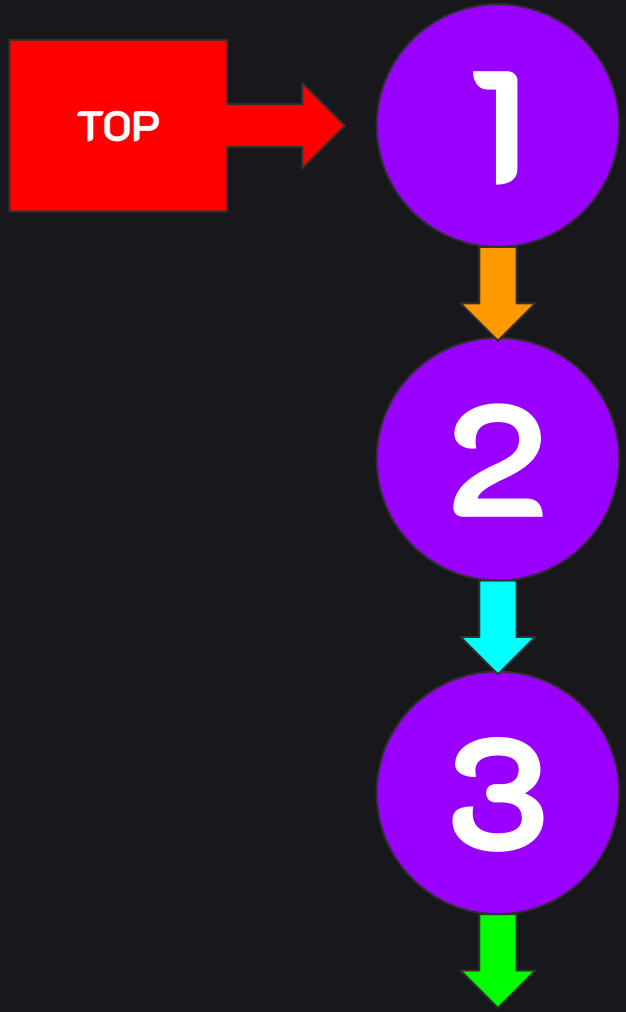


การทำงานของสแต็ก





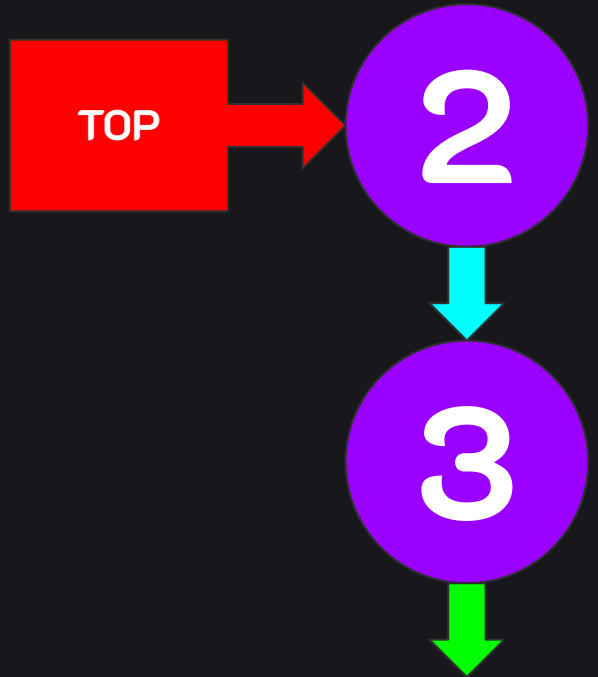

```
{
    value : 1,
        next : {
            value: 2,
            next : {
                value:3,
                next : null
            }
        }
}
```



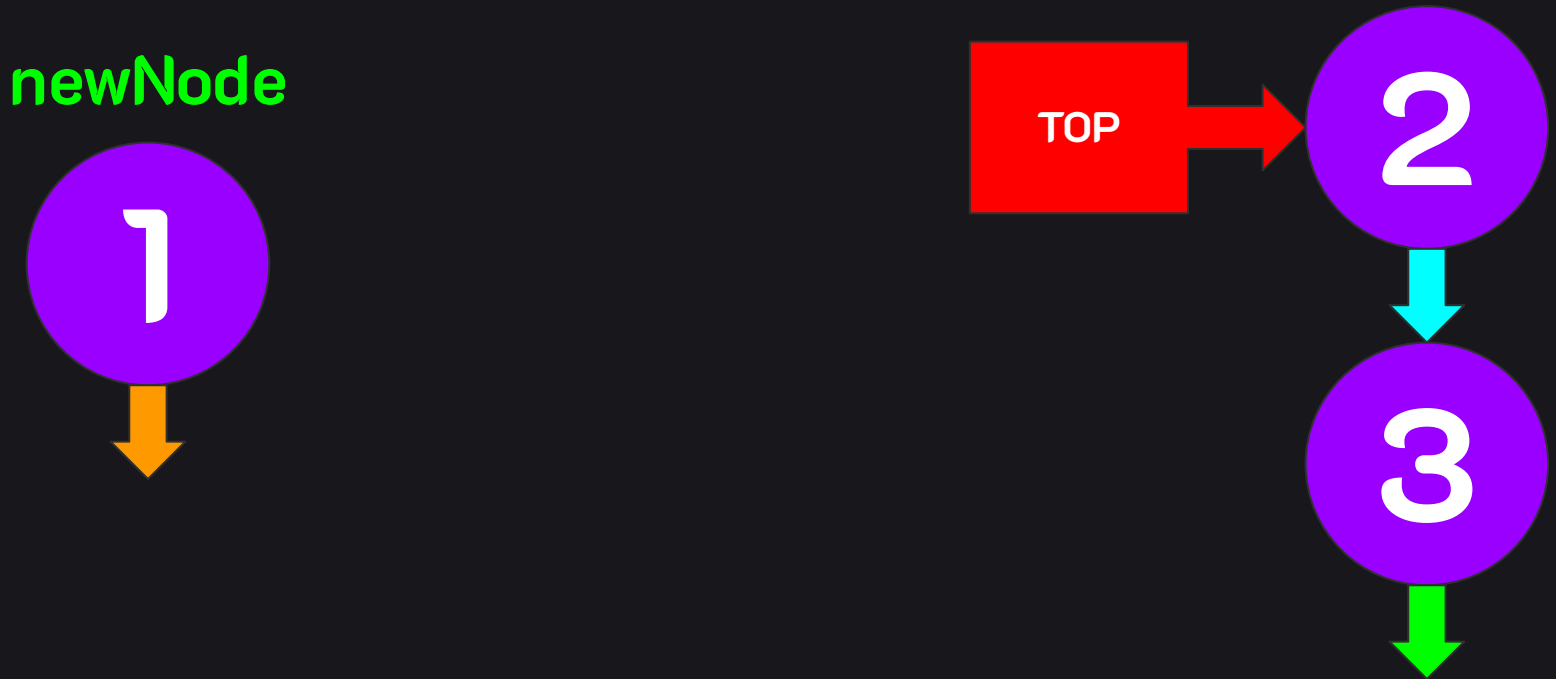

```
{
    value : 1,
        next : {
            value: 2,
            next : {
                value:3,
                next : null
            }
        }
}
```

Stack (Push)

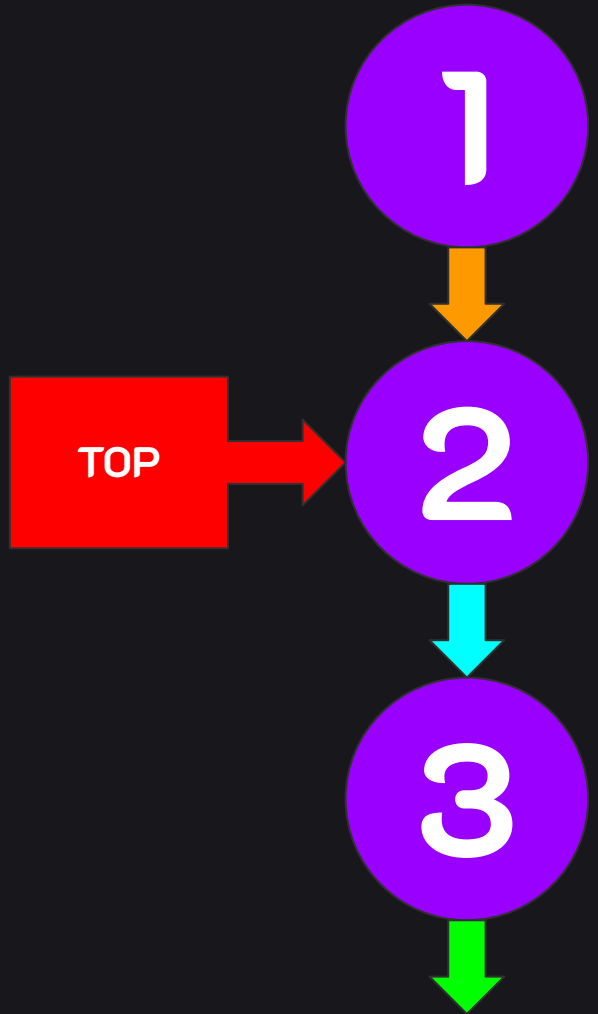
Stack (Push)



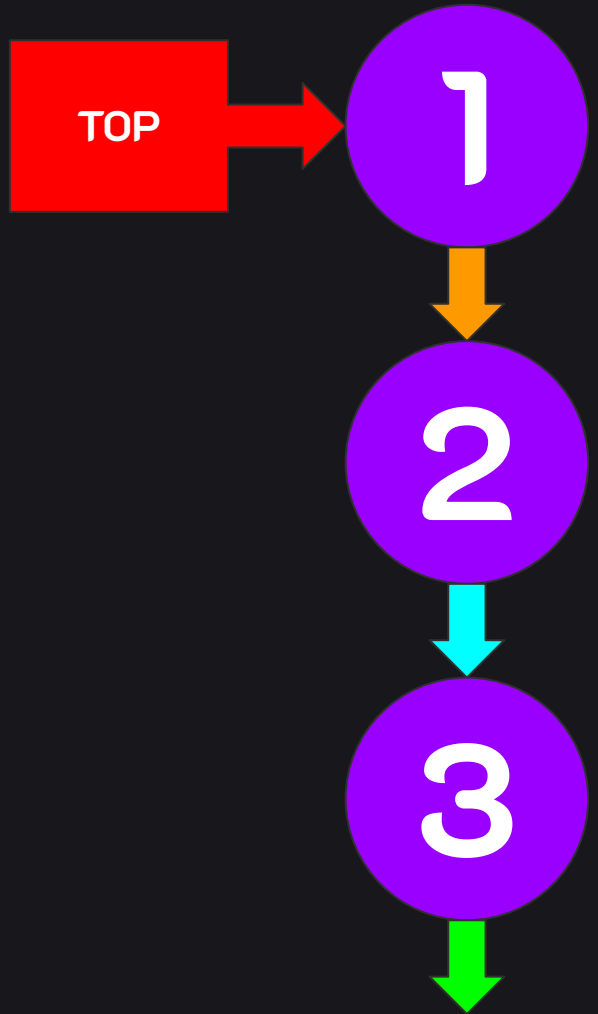
Stack (Push)



Stack (Push)

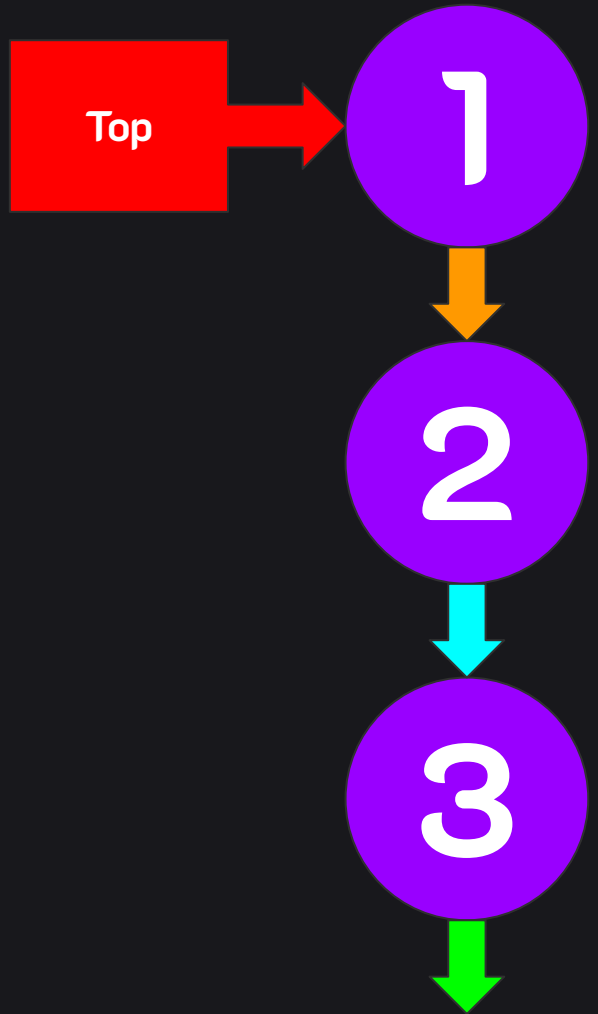


Stack (Push)

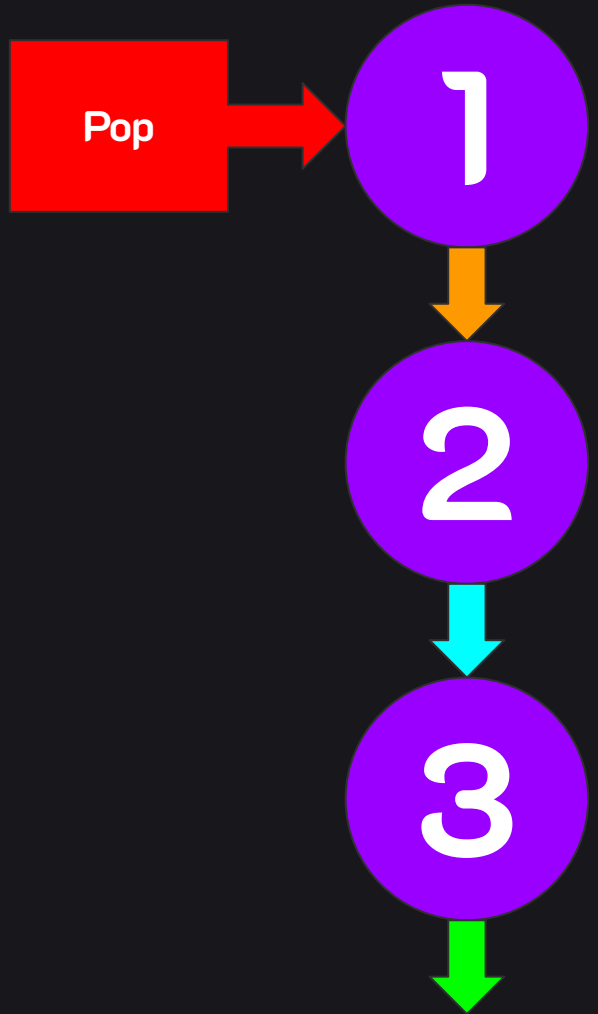


Stack (Pop)

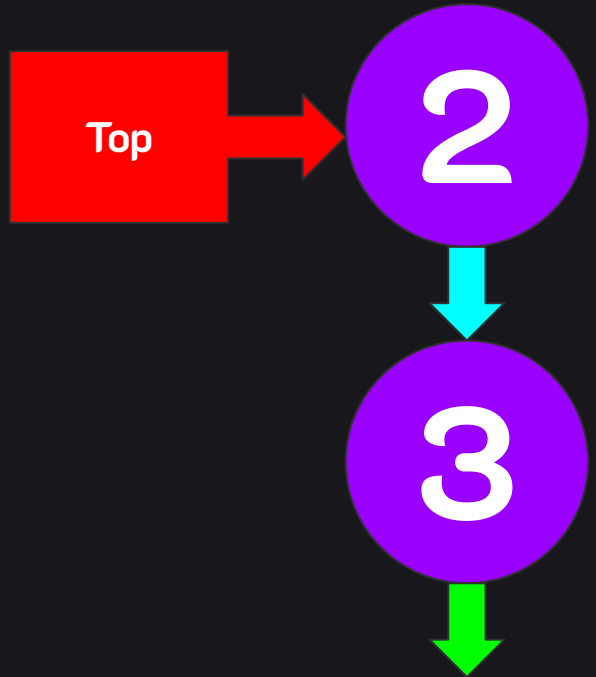
Stack (Pop)



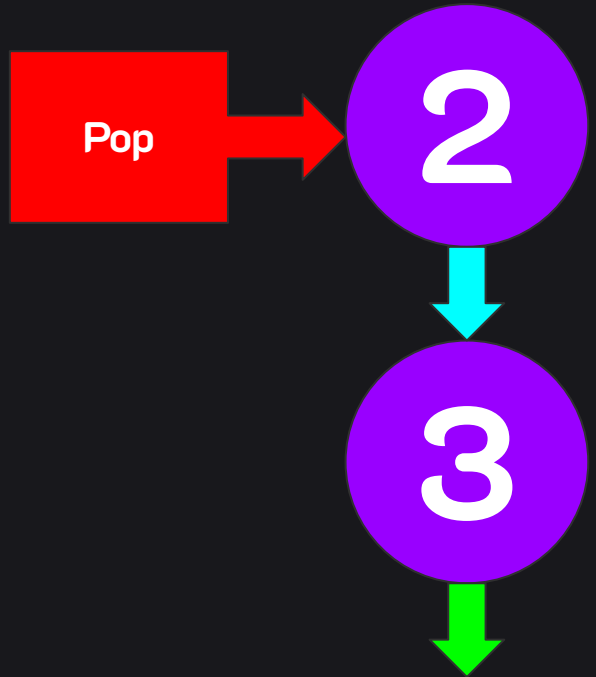
Stack (Pop)



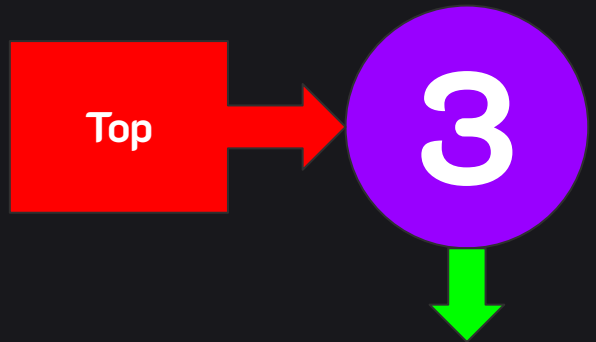
Stack (Pop)



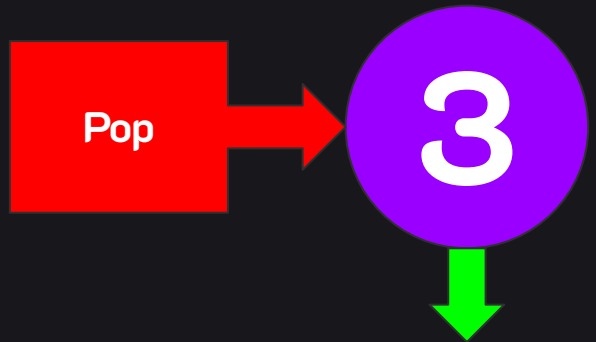
Stack (Pop)



Stack (Pop)



Stack (Pop)



Stack (Pop)

Empty Stack

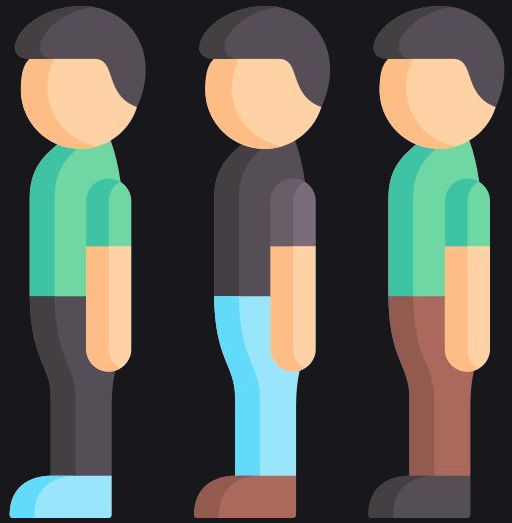
คิว
(Queue)

คิว (Queue)

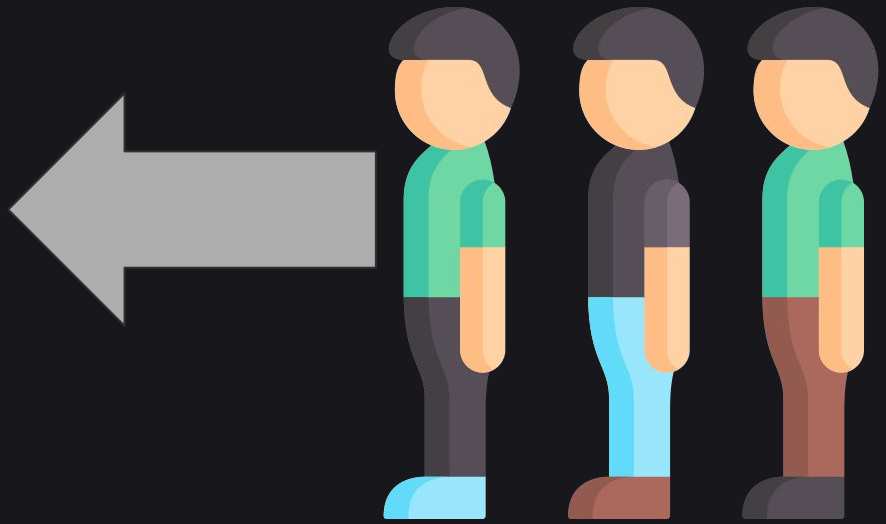
เป็นโครงสร้างข้อมูลที่มีการเก็บข้อมูลแบบลำดับ รูปแบบการจัดเก็บข้อมูลใน Queue เป็นรูปแบบ **เข้าก่อนออกก่อน (First In , First Out : FIFO)**

เปรียบเสมือนกับการเข้าแถวรอคิว ข้อมูลที่เข้าไปก่อนจะถูกใช้งานก่อน ข้อมูลที่เข้าหลังสุดจะถูกใช้งานทีหลัง

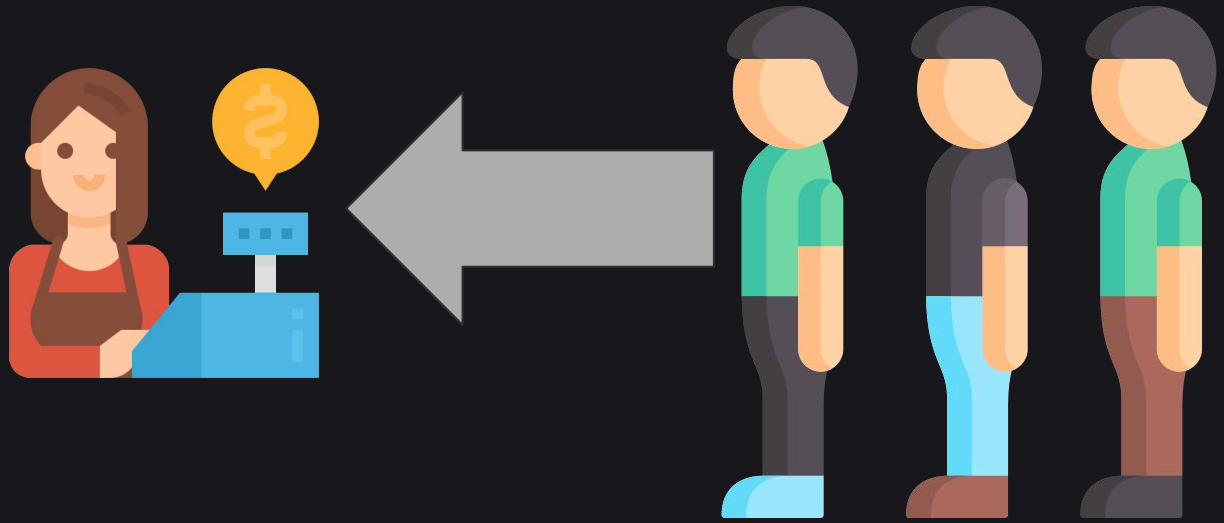
การทำงานของคิว (Queue)



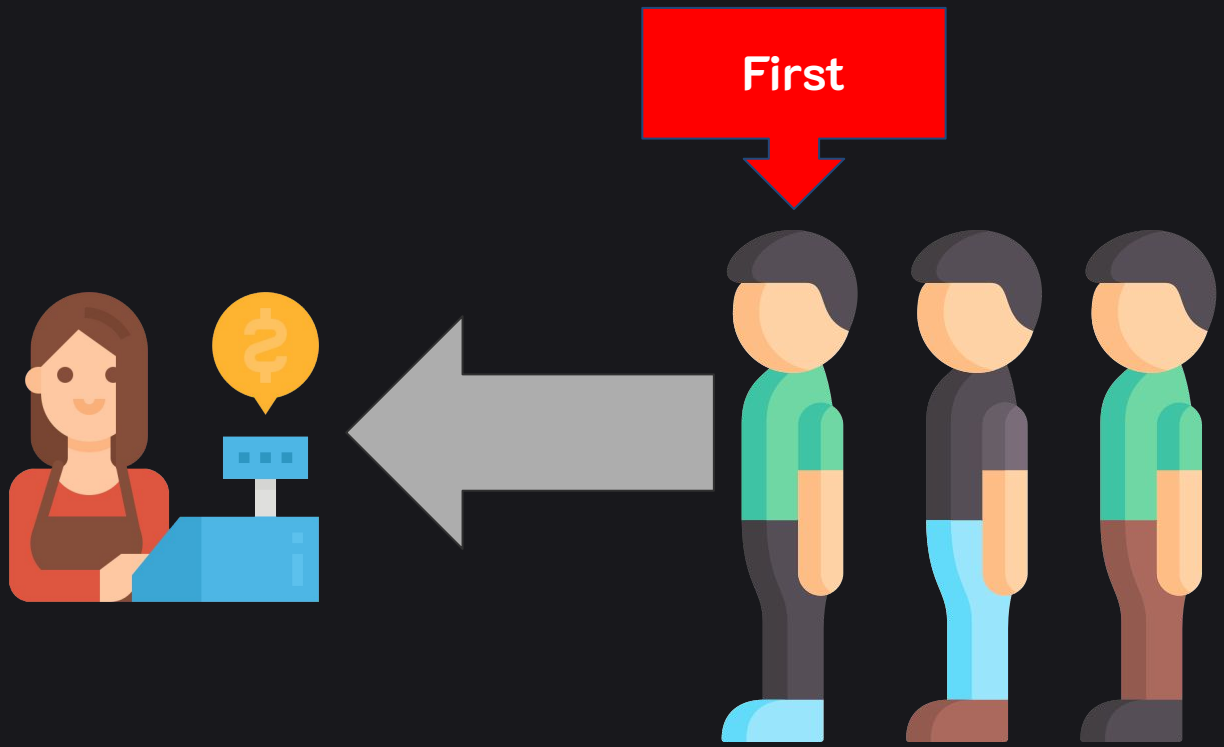
การทำงานของคิว (Queue)



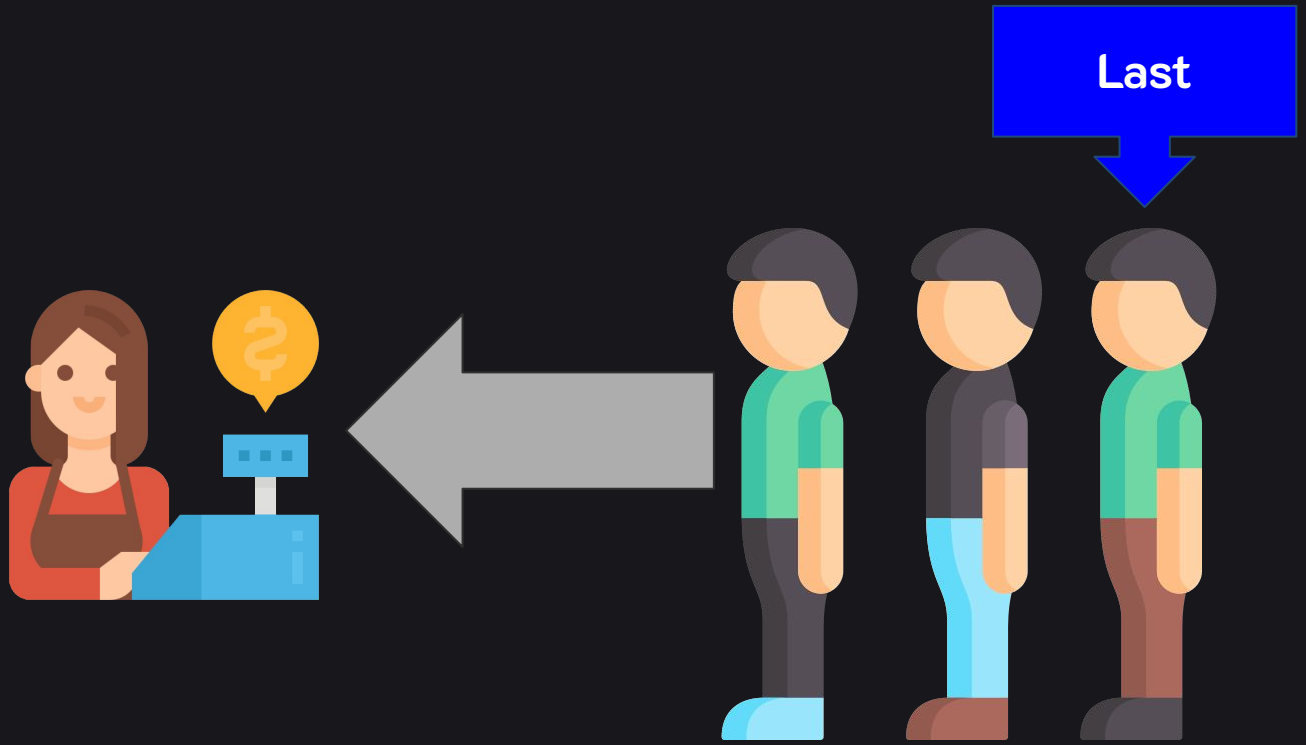
การทำงานของคิว (Queue)



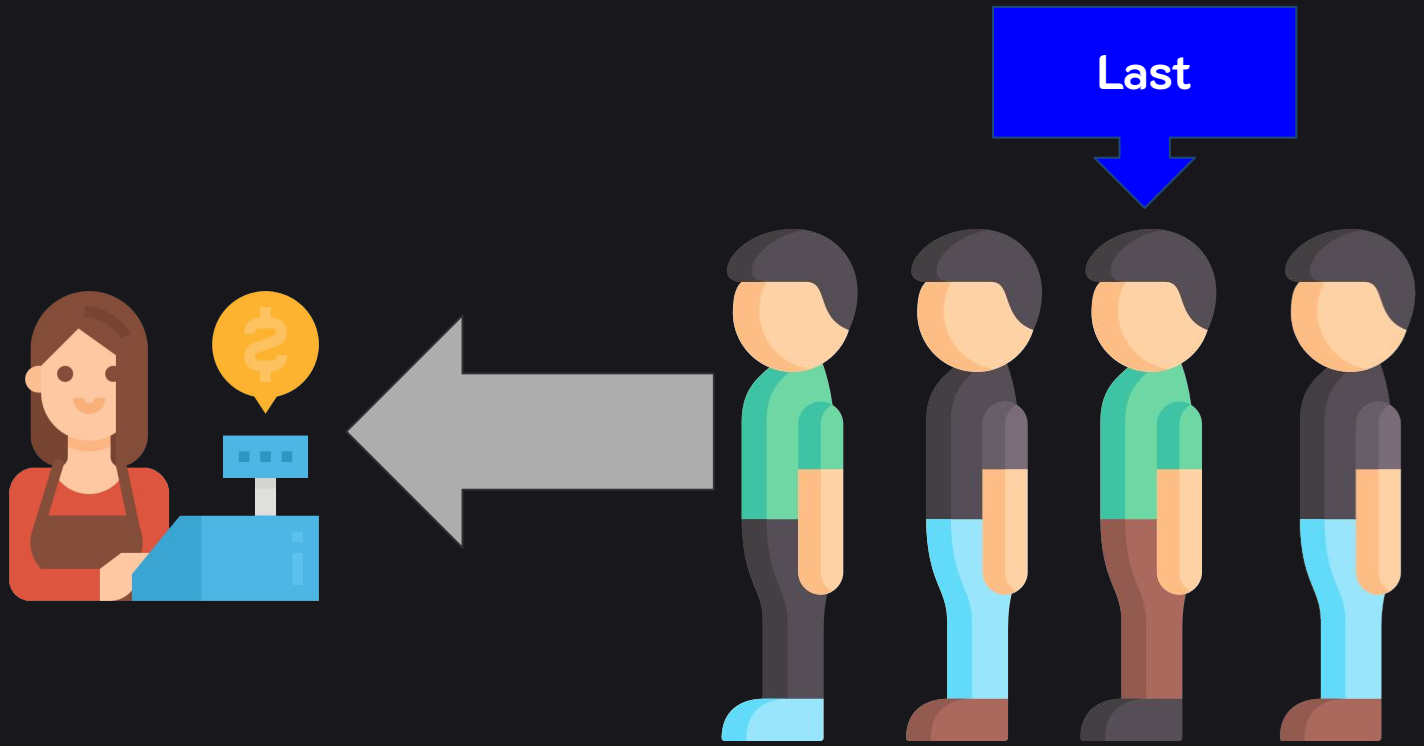
การทำงานของคิว (Queue)



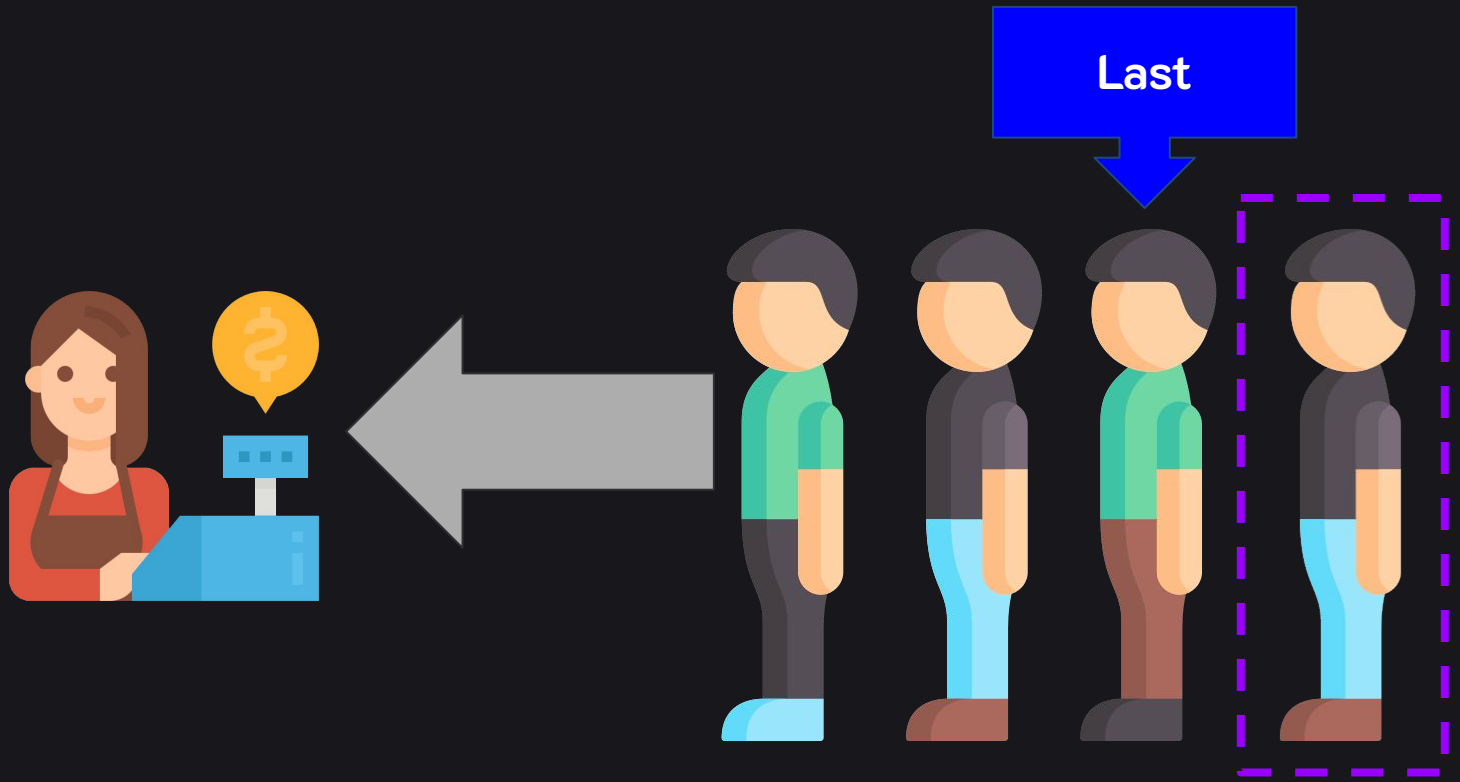
การทำงานของคิว (Queue)



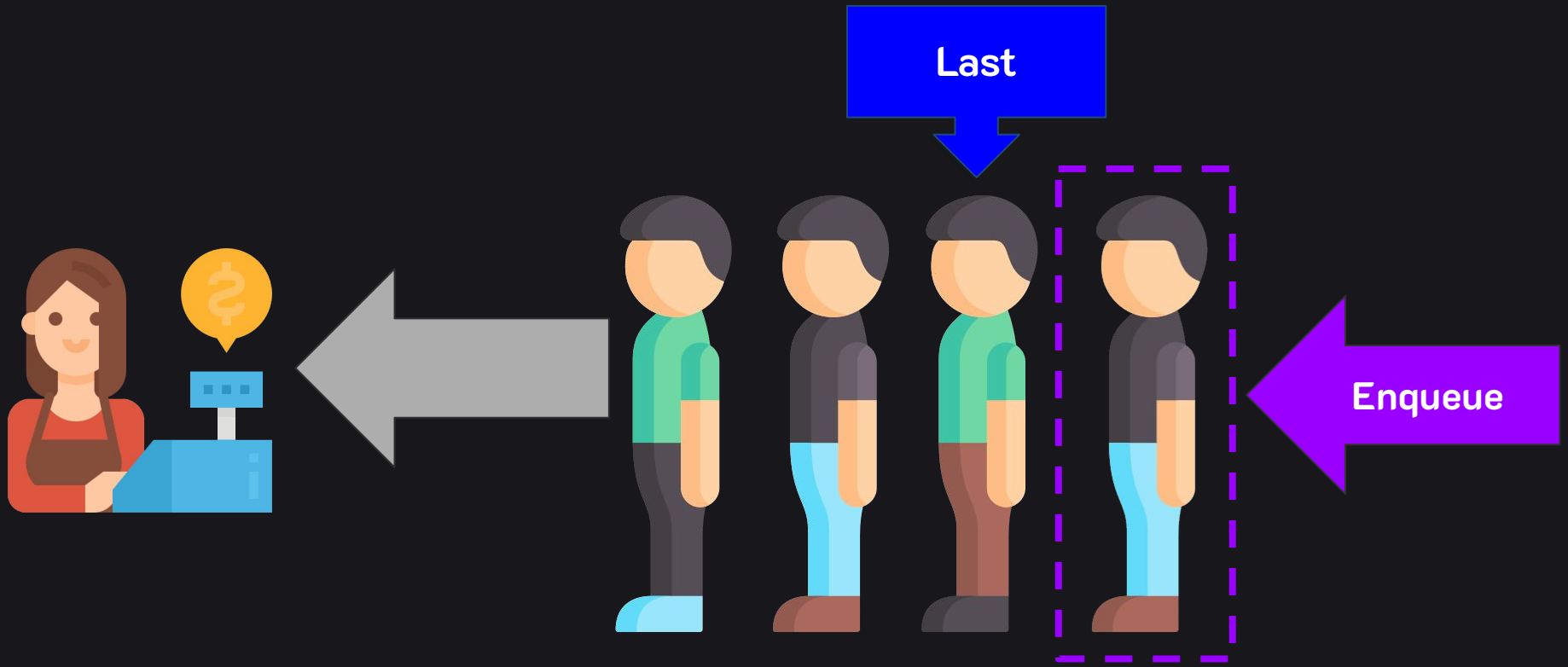
การทำงานของคิว (Queue)



การทำงานของคิว (Queue)

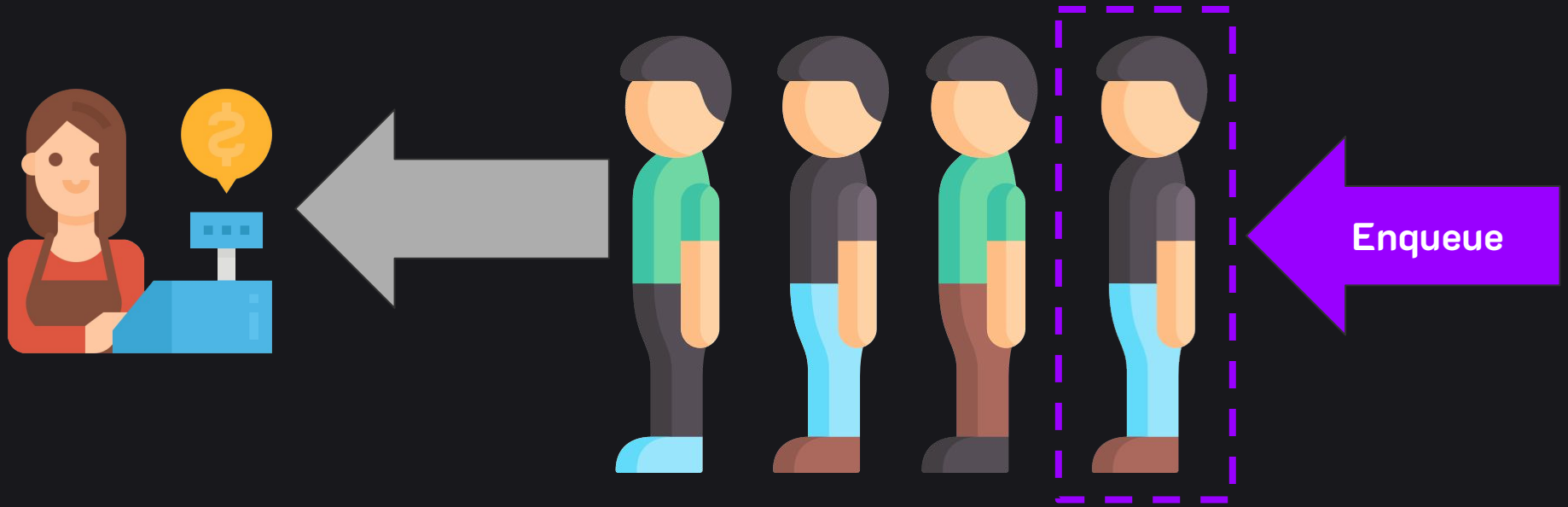


การทำงานของคิว (Queue)

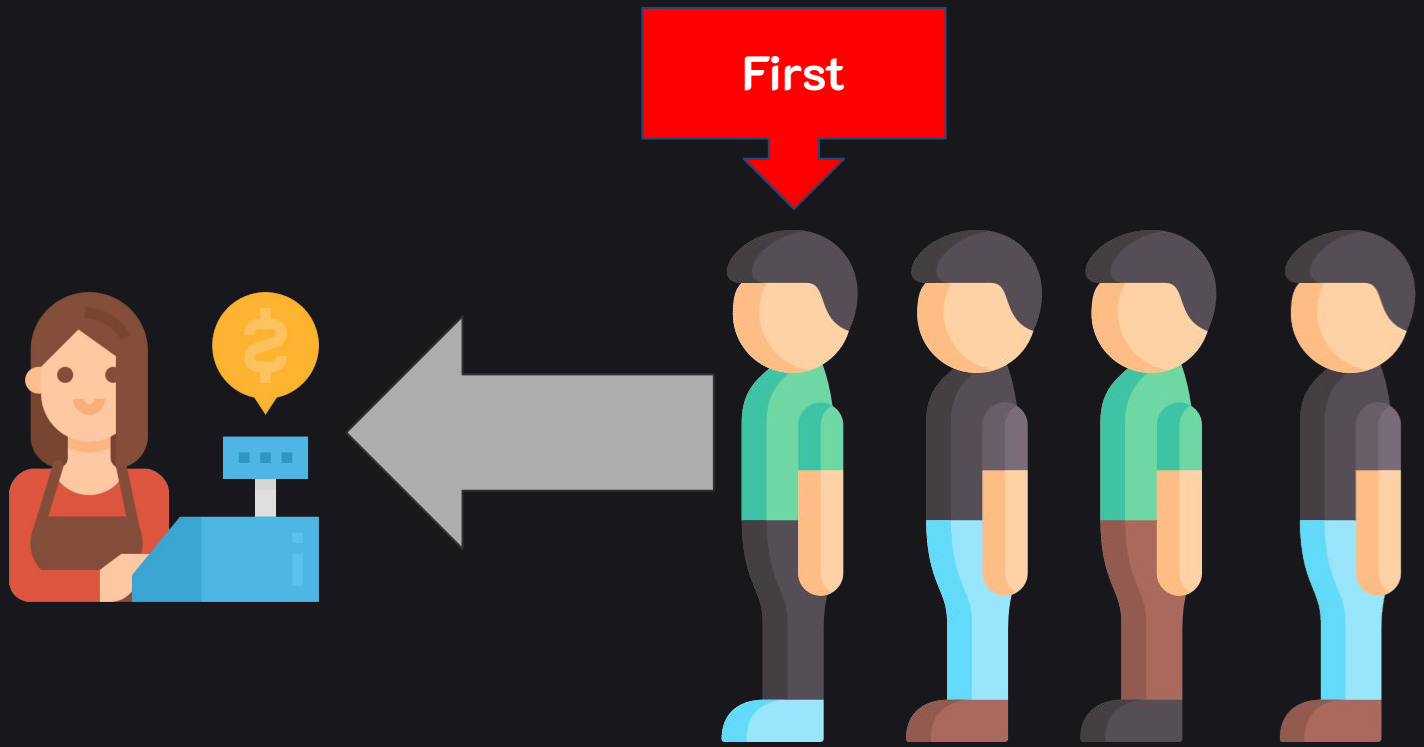


การทำงานของคิว (Queue)

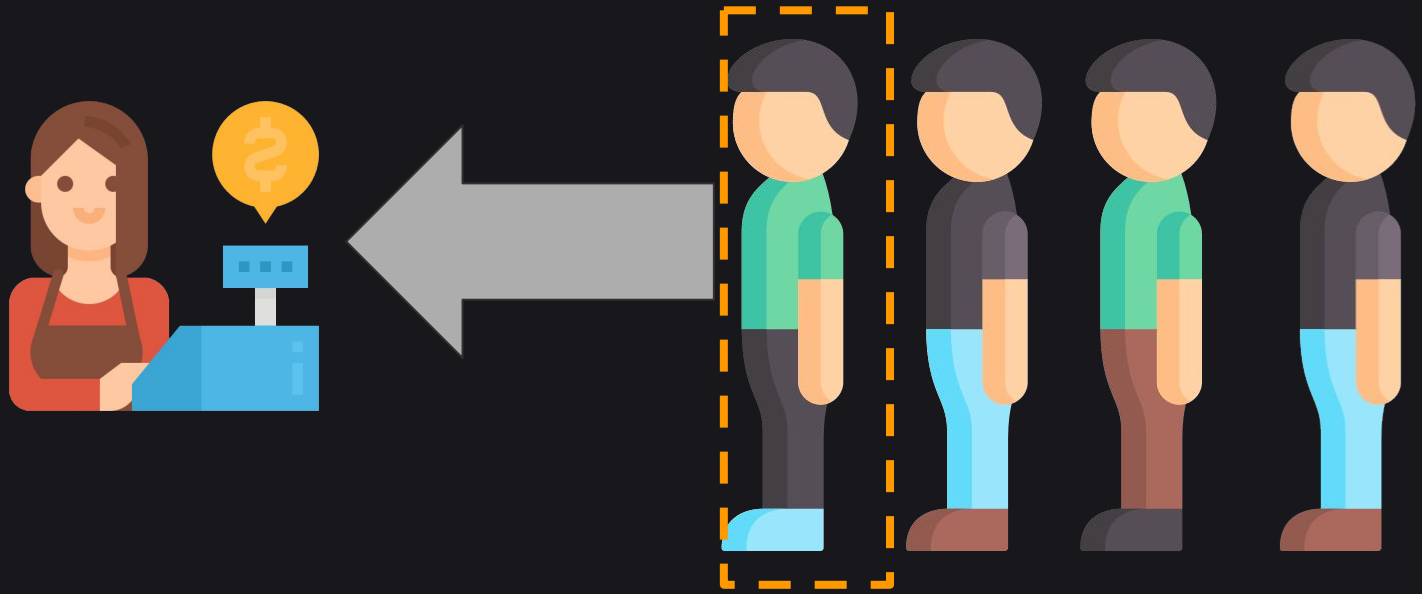
Enqueue คือการนำสมาชิก
มาต่อตำแหน่งท้ายสุดของคิว



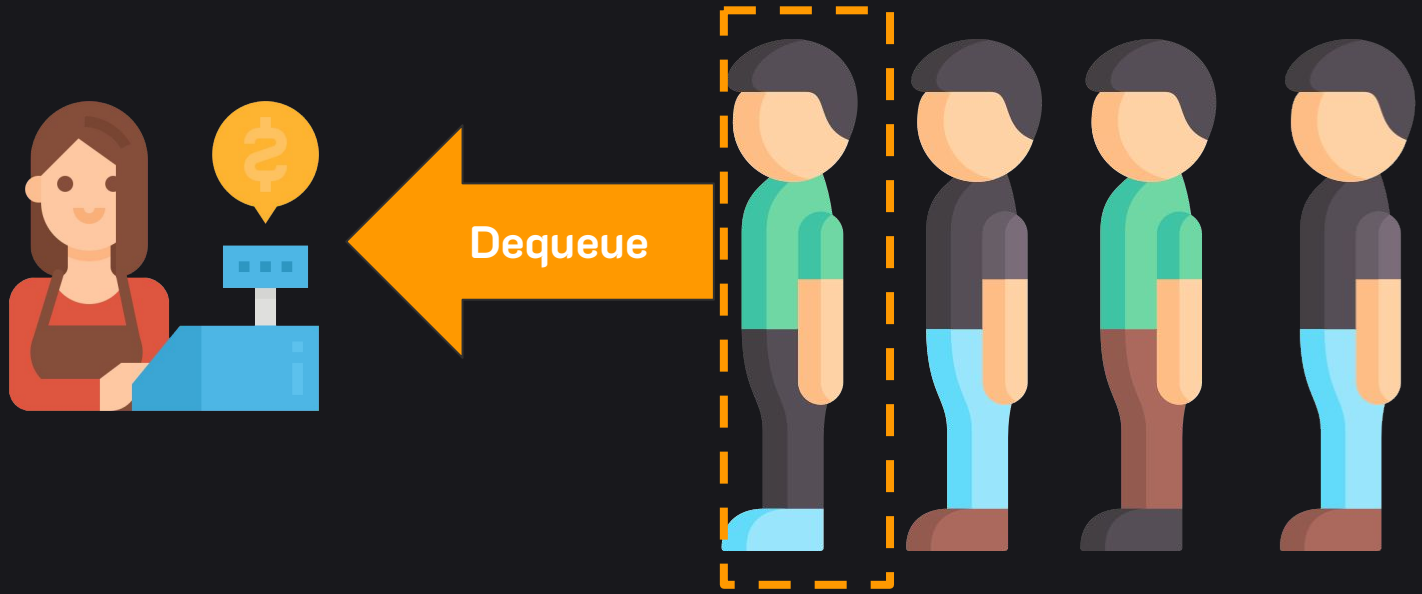
การทำงานของคิว (Queue)



การทำงานของคิว (Queue)

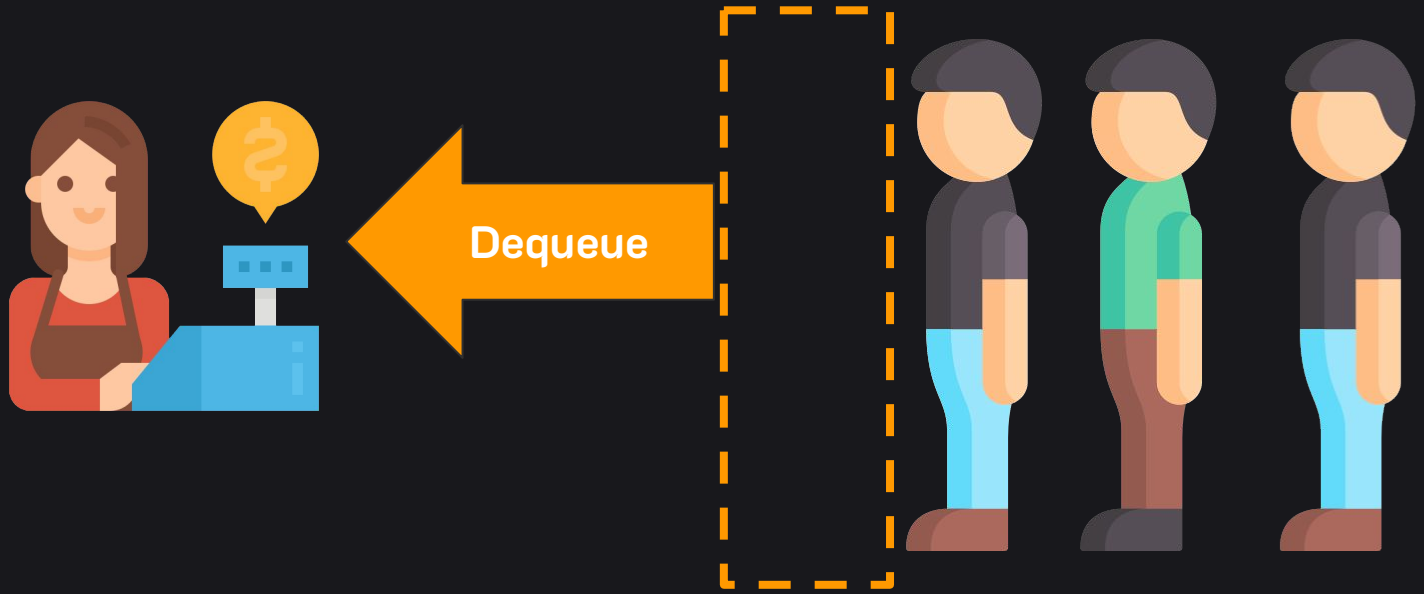


การทำงานของคิว (Queue)

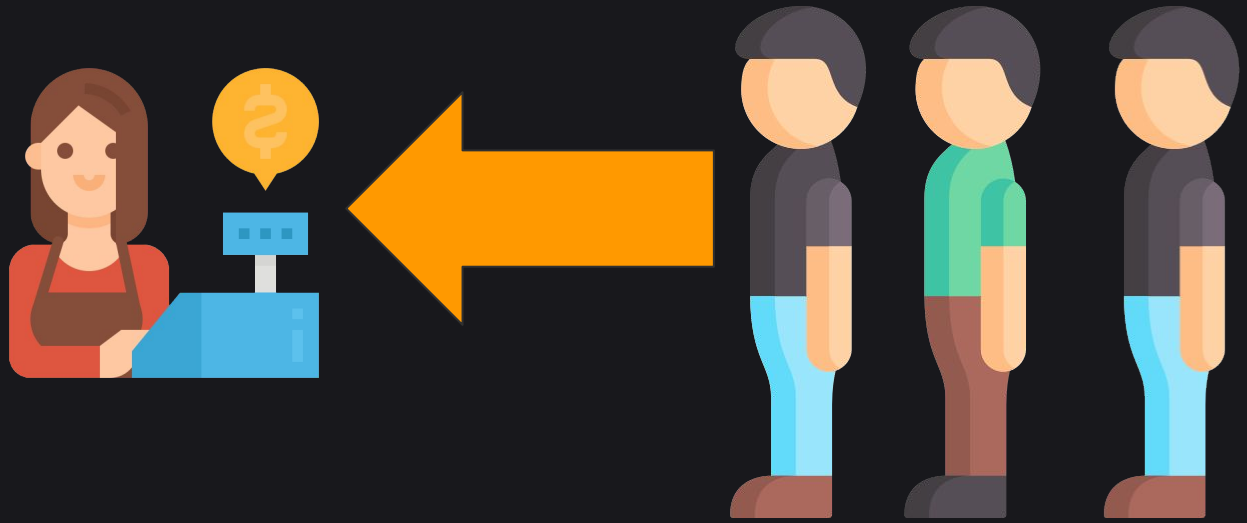


การทำงานของคิว (Queue)

Dequeue คือการนำสมาชิก
ตำแหน่งหน้าสุดออกจากคิว



การทำงานของคิว (Queue)



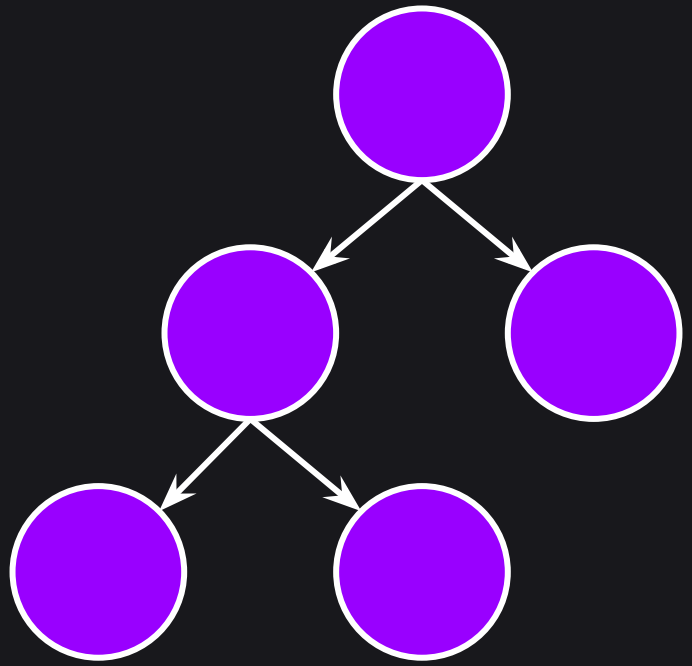
ทรี (Tree)

ทรี (Tree)

เป็นโครงสร้างที่ไม่เป็นเชิงเส้น (Non-Linear) รูปแบบของทรีเหมือนกับแผนผังโครงสร้าง ที่มีความสัมพันธ์ด้านในเป็นลำดับชั้น โดยจะทำงานตั้งแต่ลำดับสูงสุดมาที่ลำดับล่างสุด

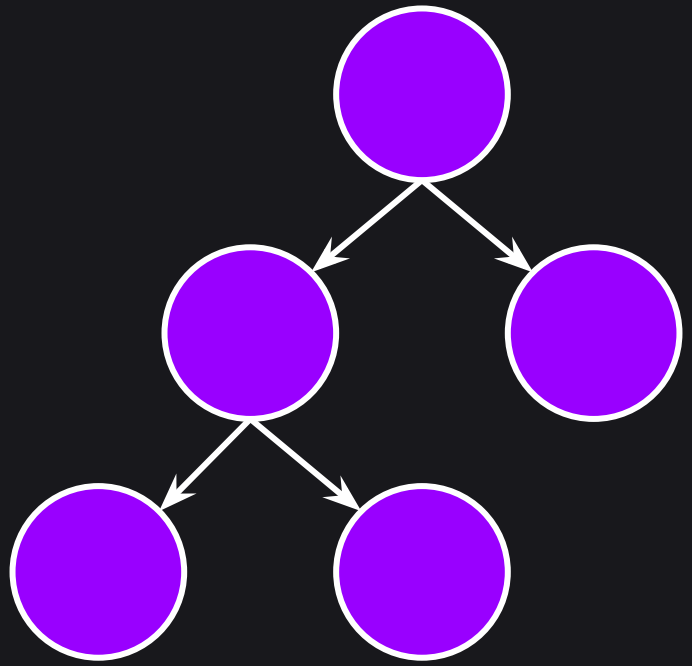
ทรี (Tree)

เป็นโครงสร้างที่ไม่เป็นเชิงเส้น (Non-Linear) รูปแบบของทรีเหมือนกับแผนผังโครงสร้าง ที่มีความสัมพันธ์ด้านในเป็นลำดับชั้น โดยจะทำงานตั้งแต่ลำดับสูงสุดมาที่ลำดับล่างสุด



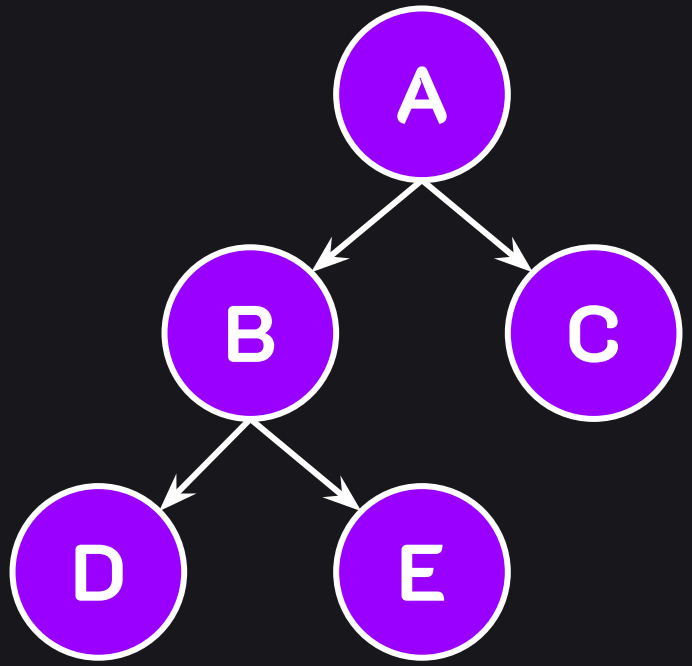
คุณสมบัติของทรี

ทรีมีโครงสร้างแบบลำดับชั้นแต่ละลำดับชั้นมีความสัมพันธ์ระหว่างข้อมูลเป็นแบบแม่กับลูก (Parent-Child) ภายในความสัมพันธ์ระหว่างโหนดแม่และโหนดลูกนี้จะถูกเชื่อมด้วยเส้นความสัมพันธ์ (Edge)



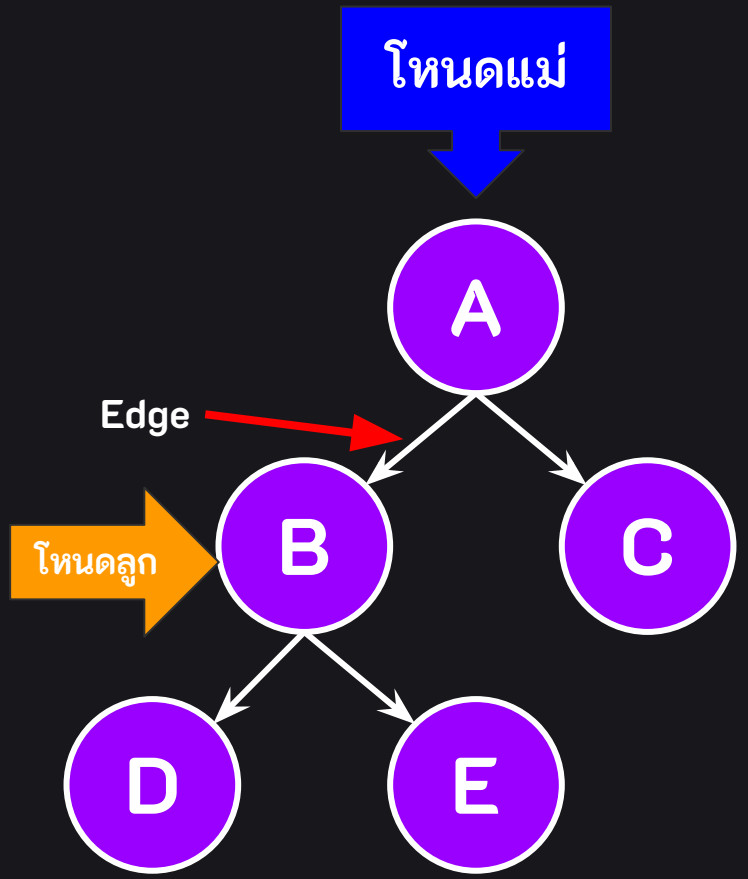
คุณสมบัติของทรี

ทรีมีโครงสร้างแบบลำดับชั้นแต่ละลำดับชั้นมีความสัมพันธ์ระหว่างข้อมูลเป็นแบบแม่กับลูก (Parent-Child) ภายในความสัมพันธ์ระหว่างโหนดแม่และโหนดลูกนี้จะถูกเชื่อมด้วยเส้นความสัมพันธ์ (Edge)



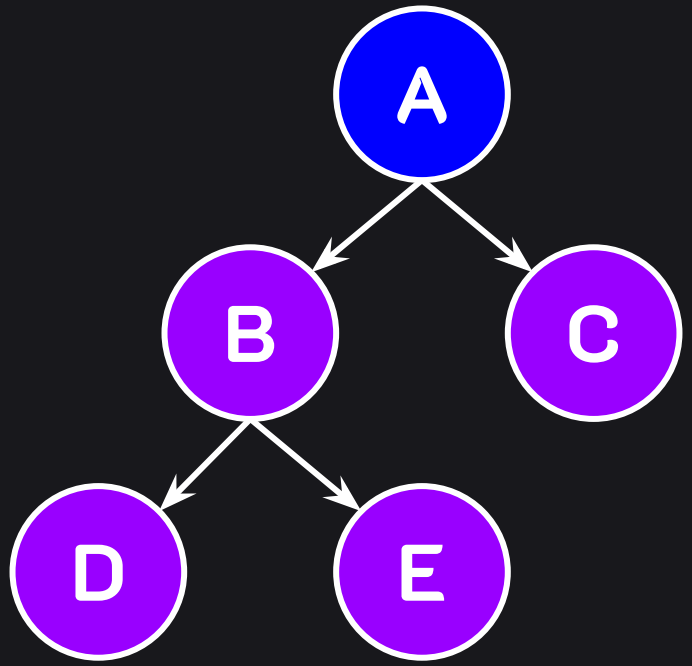
คุณสมบัติของทรี

ทรีมีโครงสร้างแบบลำดับชั้นแต่ละลำดับชั้นมีความสัมพันธ์ระหว่างข้อมูลเป็นแบบแม่กับลูก (Parent-Child) ภายในความสัมพันธ์ระหว่างโหนดแม่และโหนดลูกนี้จะถูกเชื่อมด้วยเส้นความสัมพันธ์ (Edge)



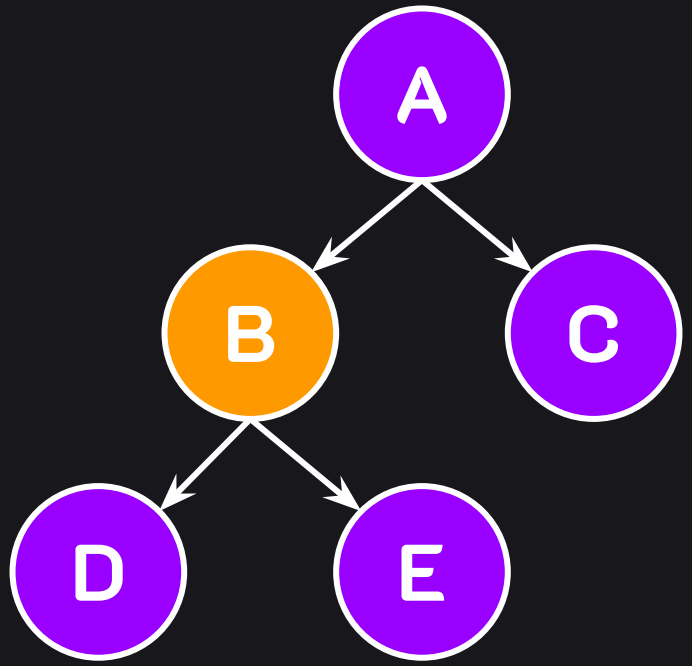
อธิบายโครงสร้างทรี

โหนดแม่ (Parent) คือ โหนดที่อยู่ลำดับบนของอีกโหนด จากภาพตำแหน่งโหนด A อยู่ตำแหน่งบนของโหนด B ดังนั้น จะเรียกโหนด A เป็นโหนดแม่ของโหนด B



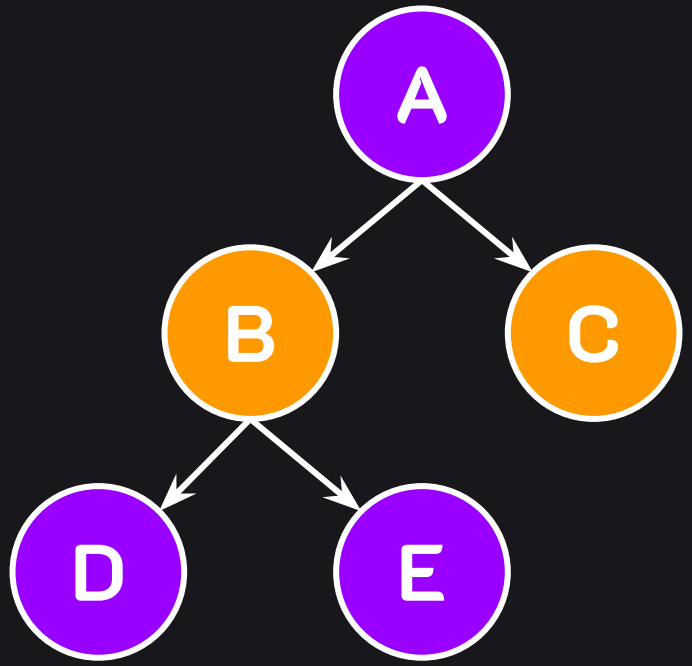
อธิบายโครงสร้างทรี

โหนดลูก (Child) คือ โหนดที่อยู่ลำดับล่างของอีกโหนด จากภาพตำแหน่งโหนด B อยู่ตำแหน่งล่างของโหนด A ดังนั้น จะเรียกโหนด B เป็นโหนดลูกของโหนด A



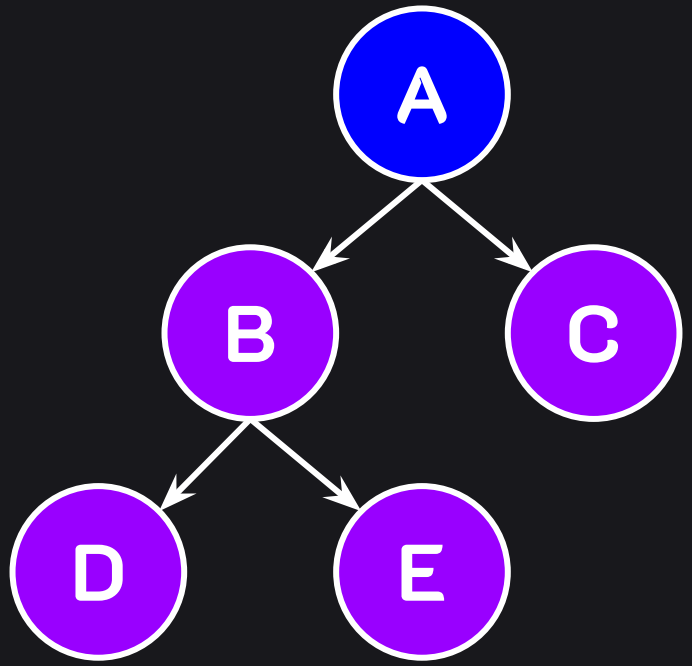
อธิบายโครงสร้างทรี

โหนดพี่น้อง (Sibling) คือ โหนดที่อยู่ระดับเดียวกัน จากภาพตำแหน่งโหนด B และโหนด C อยู่ในระดับเดียวกัน จะเรียกโหนด B และโหนด C ว่าเป็นโหนดพี่น้อง



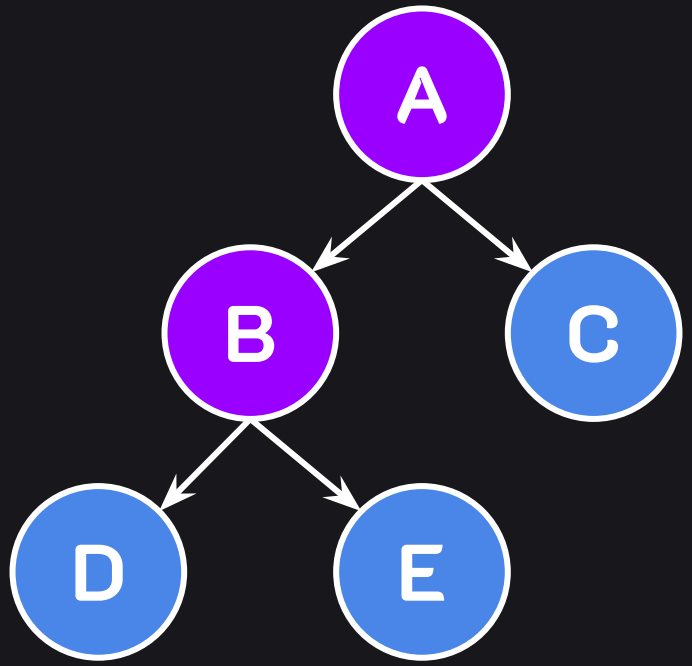
อธิบายโครงสร้างทรี

โหนดราก (Root) คือ โหนดที่มีคุณสมบัติพิเศษ เป็นโหนดที่ไม่มีโหนดแม่ และทุกโหนดในทรีจะมีโหนดนี้เป็นโหนดแม่ จากภาพ โหนด A ไม่มีโหนดแม่ จะเรียกโหนด A ว่าเป็นโหนดราก



อธิบายโครงสร้างทรี

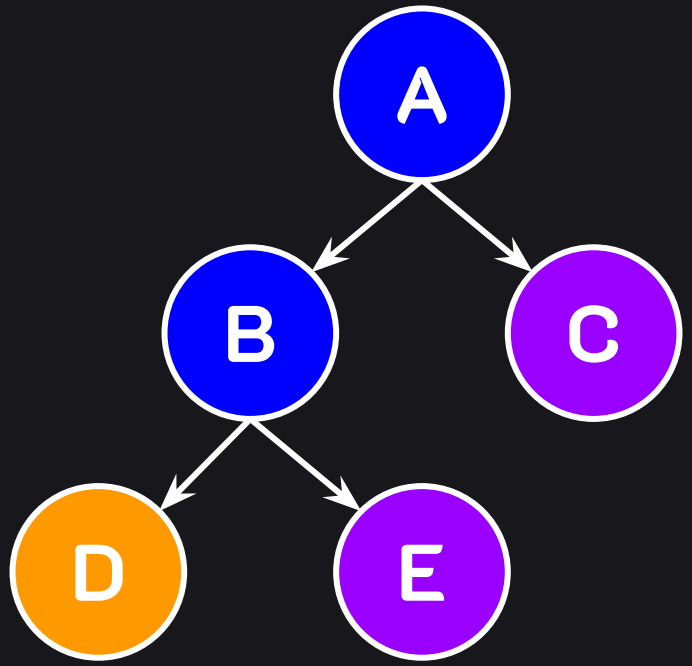
โหนดใบ (Leaf) คือ โหนดที่อยู่ตำแหน่งล่างสุดของทรี จากภาพตำแหน่งโหนด C , D , E อยู่ตำแหน่งล่างสุดของทรี ดังนั้นจะเรียกโหนดดังกล่าวว่า โหนดใบ



อธิบายโครงสร้างทรี

บรรพบุรุษ (Ancestor) คือ ความสัมพันธ์ระหว่างโหนด

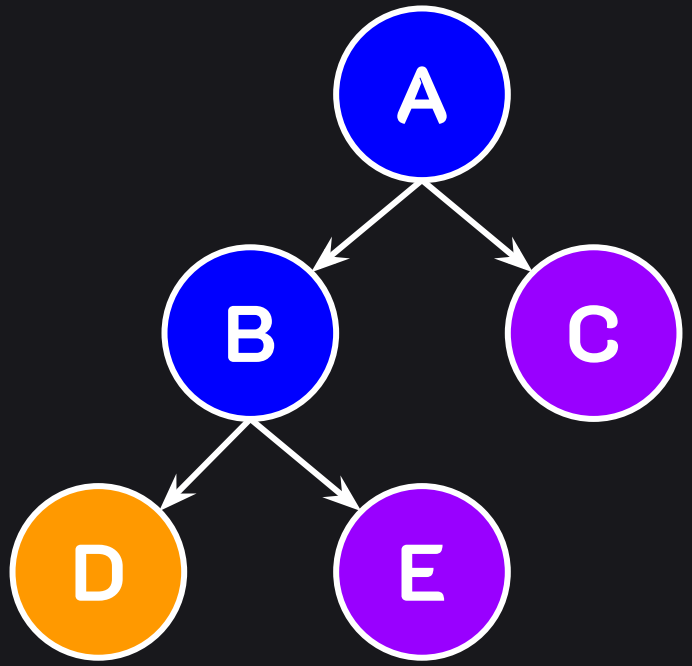
จากภาพอธิบายความสัมพันธ์ระหว่างโหนด A โหนด B , D โดยที่โหนด A , B นั้นเป็นบรรพบุรุษของโหนด D



อธิบายโครงสร้างทรี

ลูกหลาน (Descendant) คือ ความสัมพันธ์ที่มีคุณสมบัติการสืบทอดระหว่างโหนดแม่กับลูกหรือหลาน

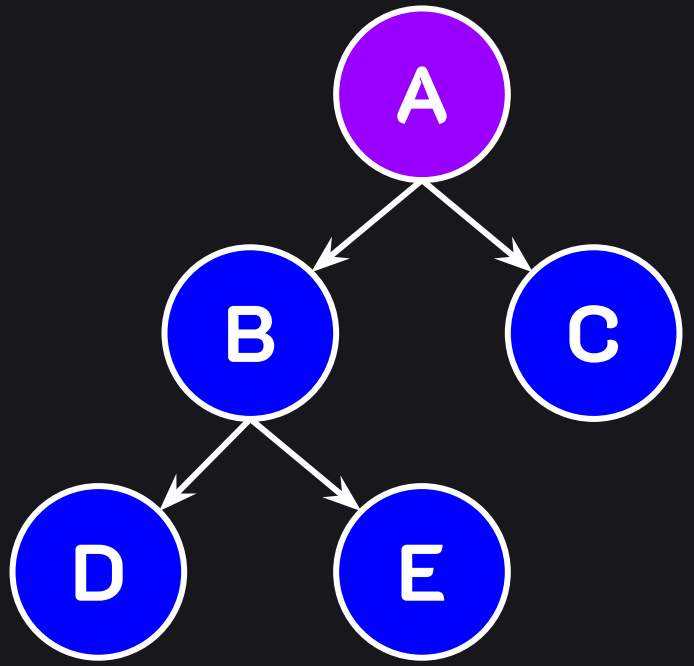
จากภาพโหนด D ได้สืบทอดมาจากโหนด A และ โหนด B



อธิบายโครงสร้างทรี

ทรีย่อย (Subtree) คือ โหนดที่อยู่ตำแหน่งสืบทอดของโหนดที่เป็นราก

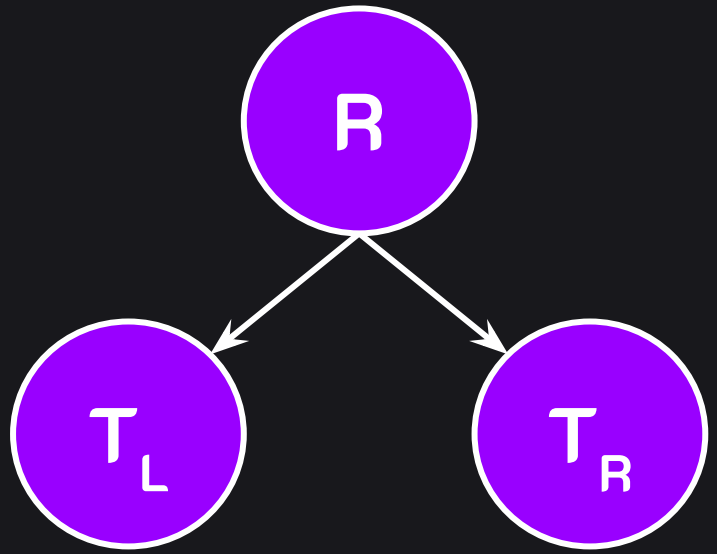
จากภาพโหนด A มีทรีย่อย คือ โหนด B , C , D และ E



ไบนารีทรี
(Binary Tree)

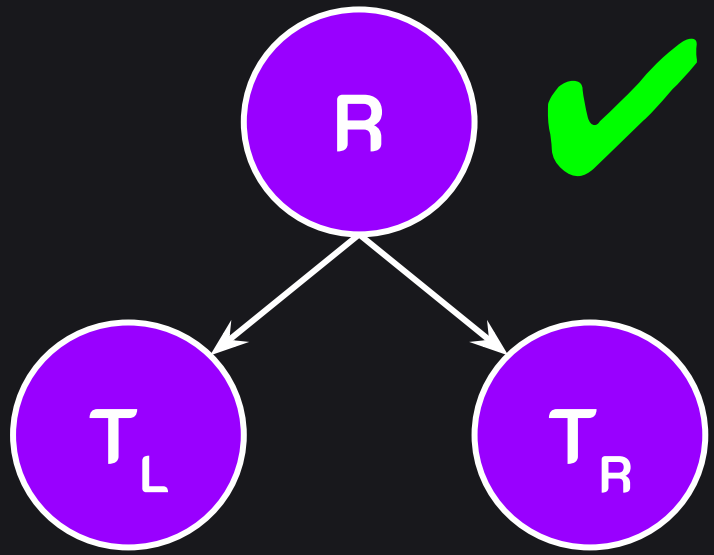
ไบนารีทรี (Binary Tree)

โครงสร้างของไบนารีทรี มีข้อกำหนดเกี่ยวกับโหนดแม่และโหนดลูก คือ โหนดแม่หนึ่งโหนด (R) มีลูกได้ไม่เกิน 2 โหนด และเรียกโหนดลูกเหล่านี้ว่า ทรีย่อย (Subtrees)



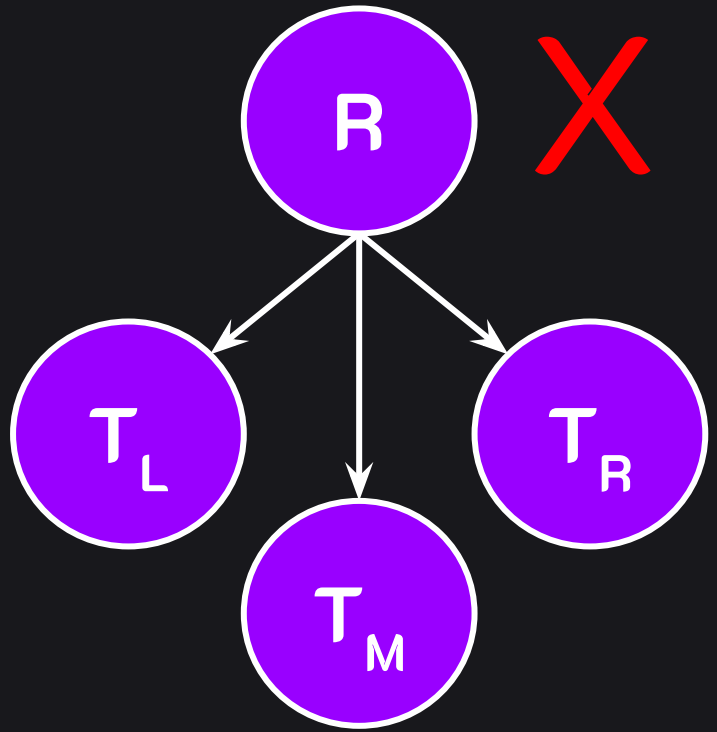
ไบนารีทรี (Binary Tree)

โครงสร้างของไบนารีทรี มีข้อกำหนดเกี่ยวกับโหนดแม่และโหนดลูก คือ โหนดแม่หนึ่งโหนด (R) มีลูกได้ไม่เกิน 2 โหนด และเรียกโหนดลูกเหล่านี้ว่า ทรีย่อย (Subtrees)



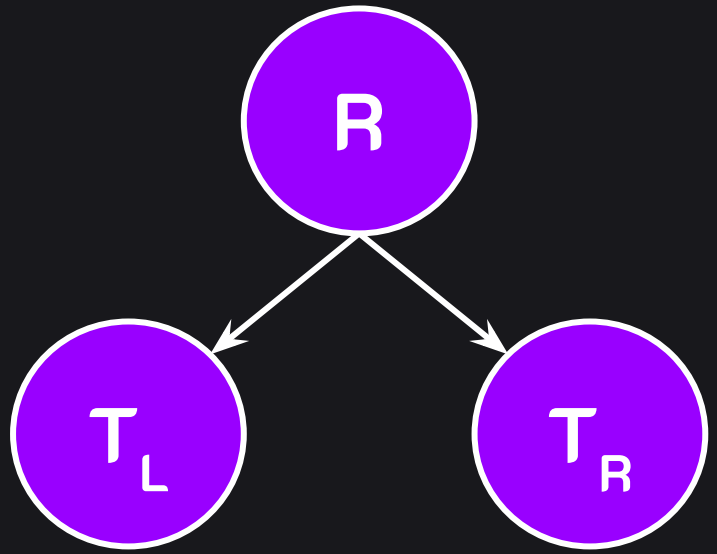
ไบนารีทรี (Binary Tree)

โครงสร้างของไบนารีทรี มีข้อกำหนดเกี่ยวกับโหนดแม่และโหนดลูก คือ โหนดแม่หนึ่งโหนด (R) มีลูกได้ไม่เกิน 2 โหนด และเรียกโหนดลูกเหล่านี้ว่า ทรีย่อย (Subtrees)



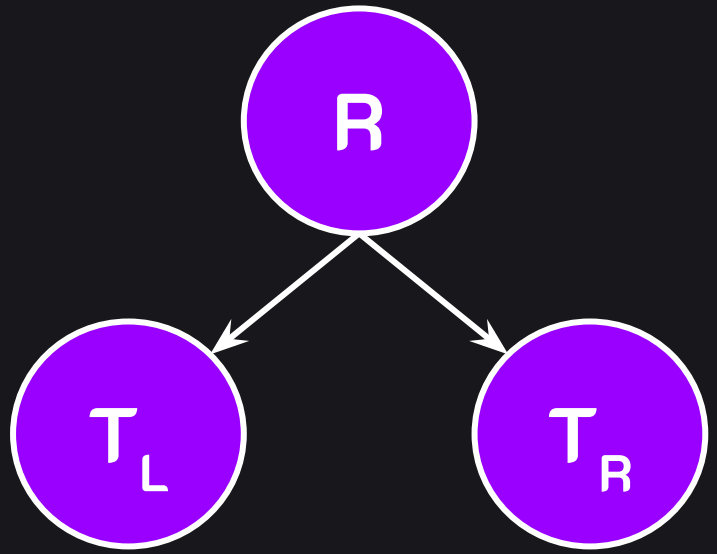
ไบนารีทรี (Binary Tree)

- โหนดลูกที่อยู่ด้านซ้ายของโหนดแม่จะเรียกว่า **ทรีย่อยซ้าย (T_L)**
- โหนดลูกที่อยู่ด้านขวาของโหนดแม่จะเรียกว่า **ทรีย่อยขวา (T_R)**



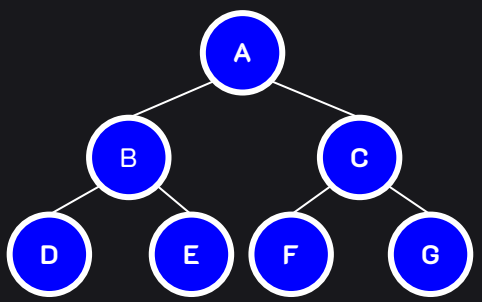
ไบนารีทรี (Binary Tree)

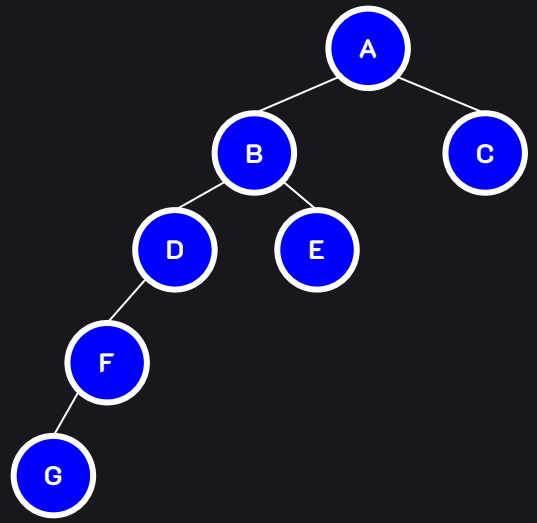
ความสูงของทรี หมายถึง จำนวนโหนดที่มีความสูงมาจากโหนดรากถึงโหนดตำแหน่งใบ โดยแทนความสูงด้วยสัญลักษณ์ h

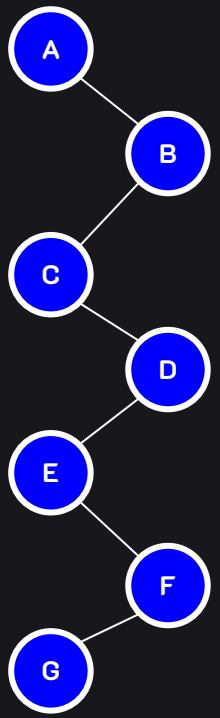


ไบนารีทรี (Binary Tree)

1
2
3
4
5
6
7

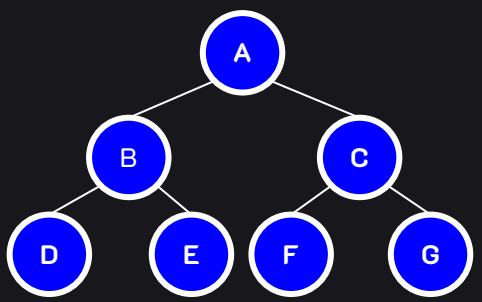




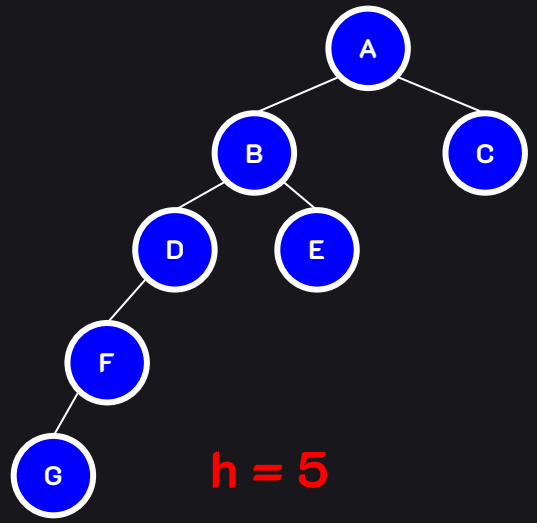


ไบนารีทรี (Binary Tree)

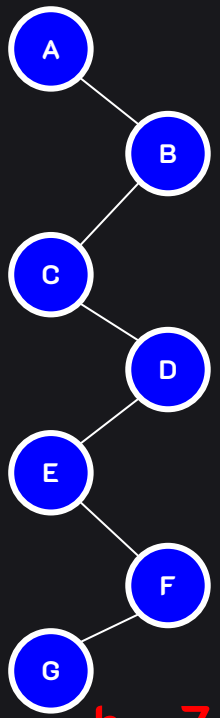
1
2
3
4
5
6
7



h = 3



h = 5



h = 7

รูปแบบของไบนารีทรี

- Full Binary Tree
- Perfect Binary Tree
- Complete Binary Tree

Full Binary Tree

- ภายในไบนารีทรีนั้นสามารถมีโหนดลูกทั้งสองด้านหรือไม่มีโหนดลูกเลย
- จำนวนโหนดลูกสามารถเป็น 2 หรือ 0

Perfect Binary Tree

- ภายในไบนารีทรีนั้นโหนดที่อยู่ในระดับเดียวกันต้องมีจำนวนโหนดเท่ากัน

Complete Binary Tree

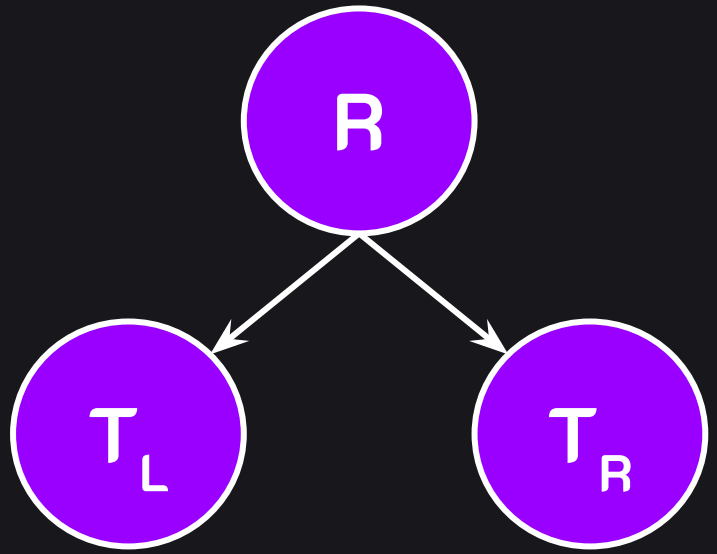
- เหมือน Full Binary Tree แต่โหนดลำดับท้ายสุดหรือโหนดใบสามารถมีลูกได้ 1 โหนด

Binary Search Tree (BST)

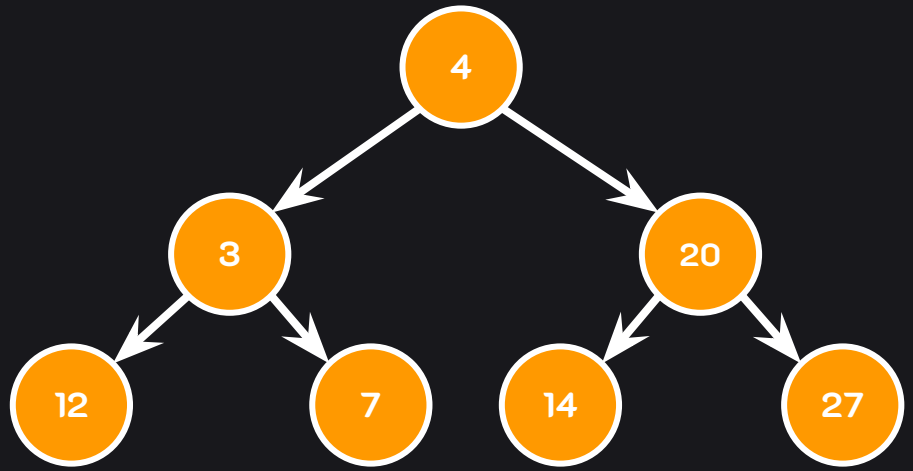
BST (Binary Search Tree)

- โหนดลูกที่อยู่ด้านซ้ายของโหนดแม่จะเรียกว่า **ทรีย่อยซ้าย (T_L)** เป็นกลุ่มข้อมูลที่มีค่าน้อยกว่าโหนดแม่

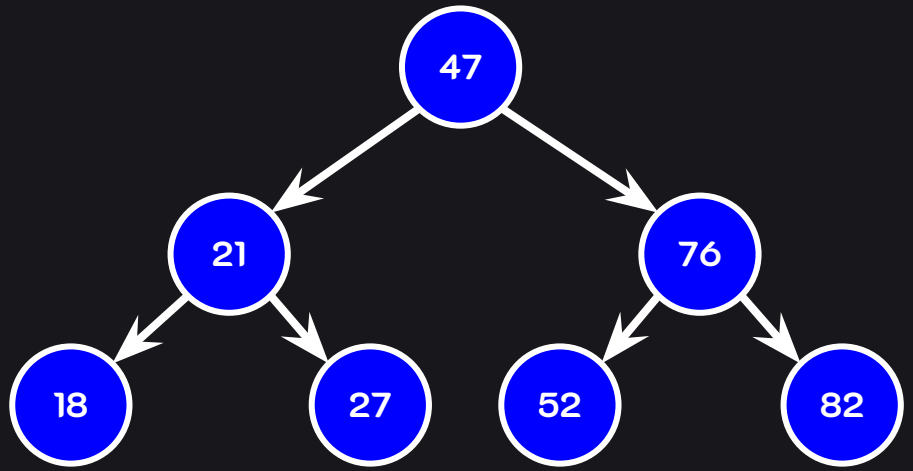
- โหนดลูกที่อยู่ด้านขวาของโหนดแม่จะเรียกว่า **ทรีย่อยขวา (T_R)** เป็นกลุ่มข้อมูลที่มีค่ามากกว่าโหนดแม่



BST (Binary Search Tree)

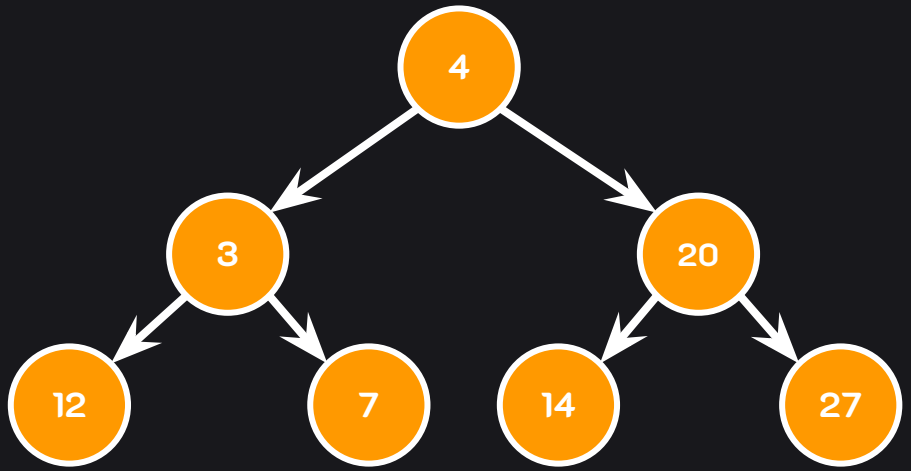


Binary Tree

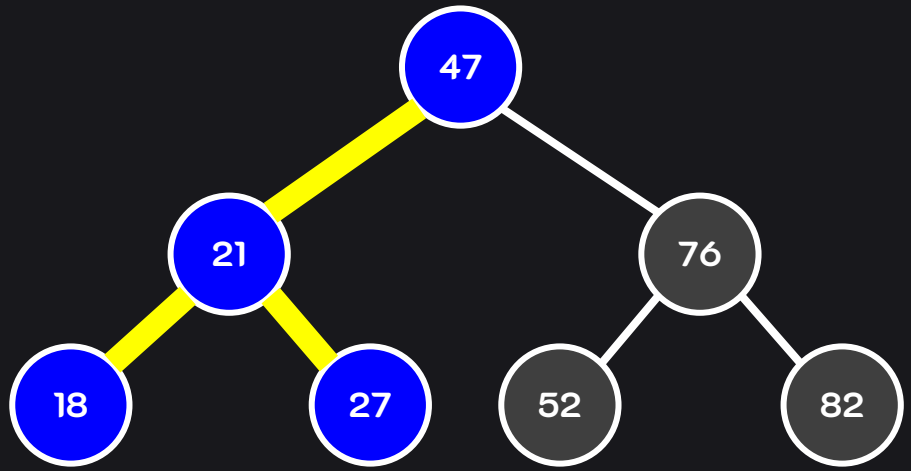


Binary Search Tree

BST (Binary Search Tree)

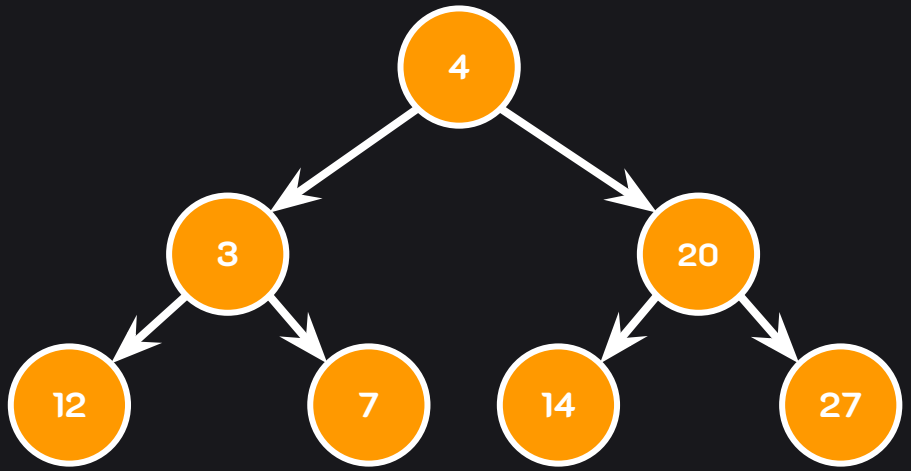


Binary Tree

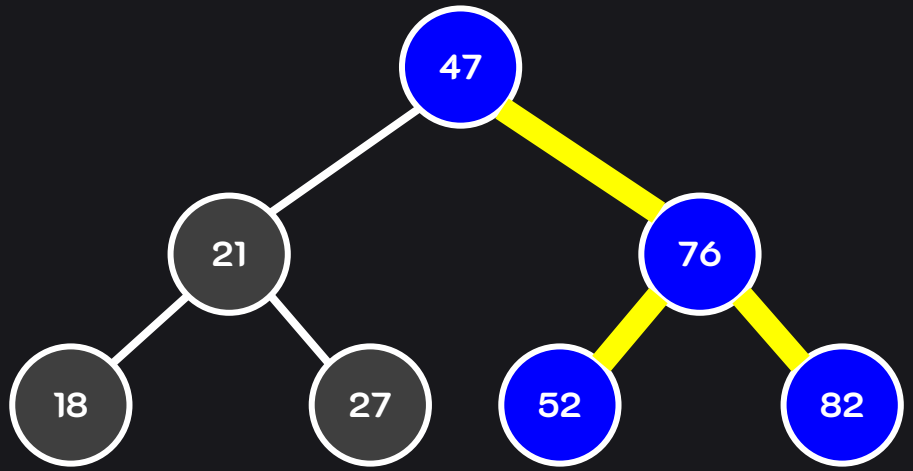


Binary Search Tree

BST (Binary Search Tree)



Binary Tree



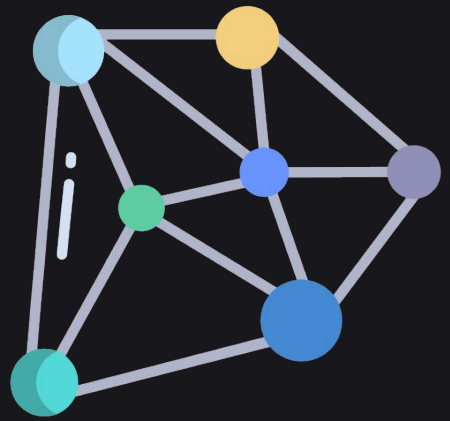
Binary Search Tree

กราฟ
(Graphs)

กราฟ (Graphs)

เป็นโครงสร้างข้อมูลที่ไม่เป็นเชิงเส้น (Non-Linear) ที่กล่าวถึงความสัมพันธ์ระหว่างโหนดที่เชื่อมความสัมพันธ์ด้วยเส้นเชื่อมโยง (Edge)

โครงสร้างข้อมูลกราฟถูกนำไปใช้ในระบบงานที่เป็นประเภทเครือข่าย การวิเคราะห์เส้นทางในการเดินทางและการเชื่อมโยงหากันในระยะทางที่สั้นที่สุด



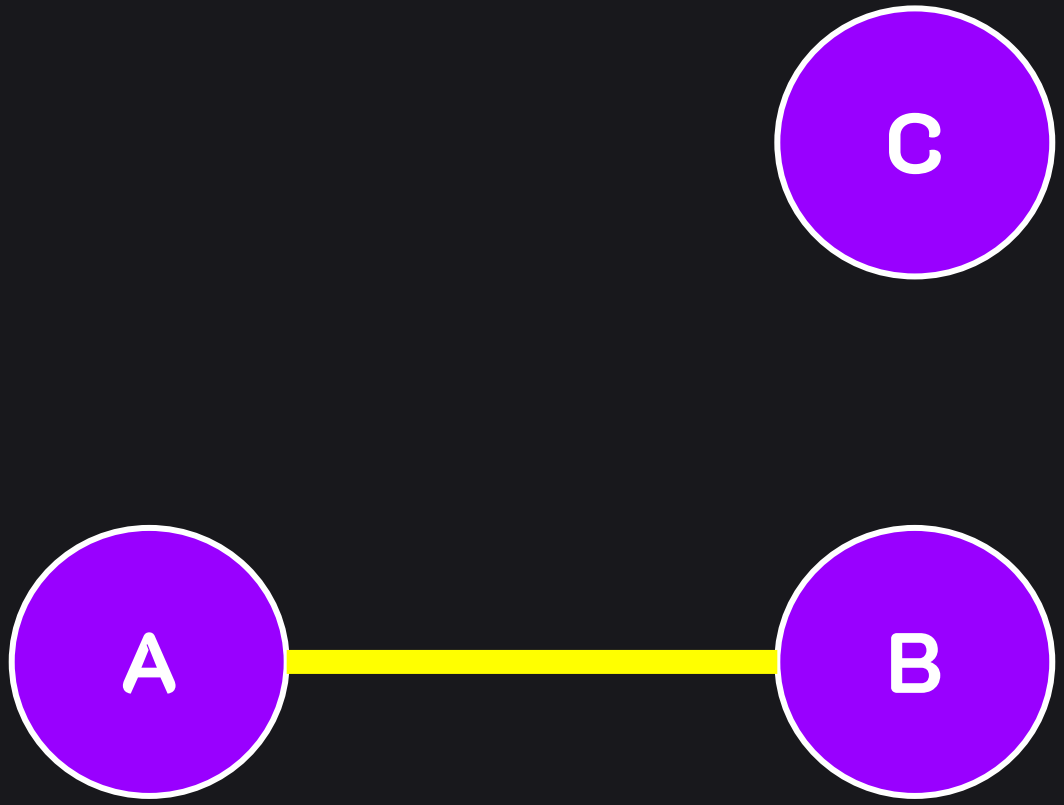
โหนด
(Node)

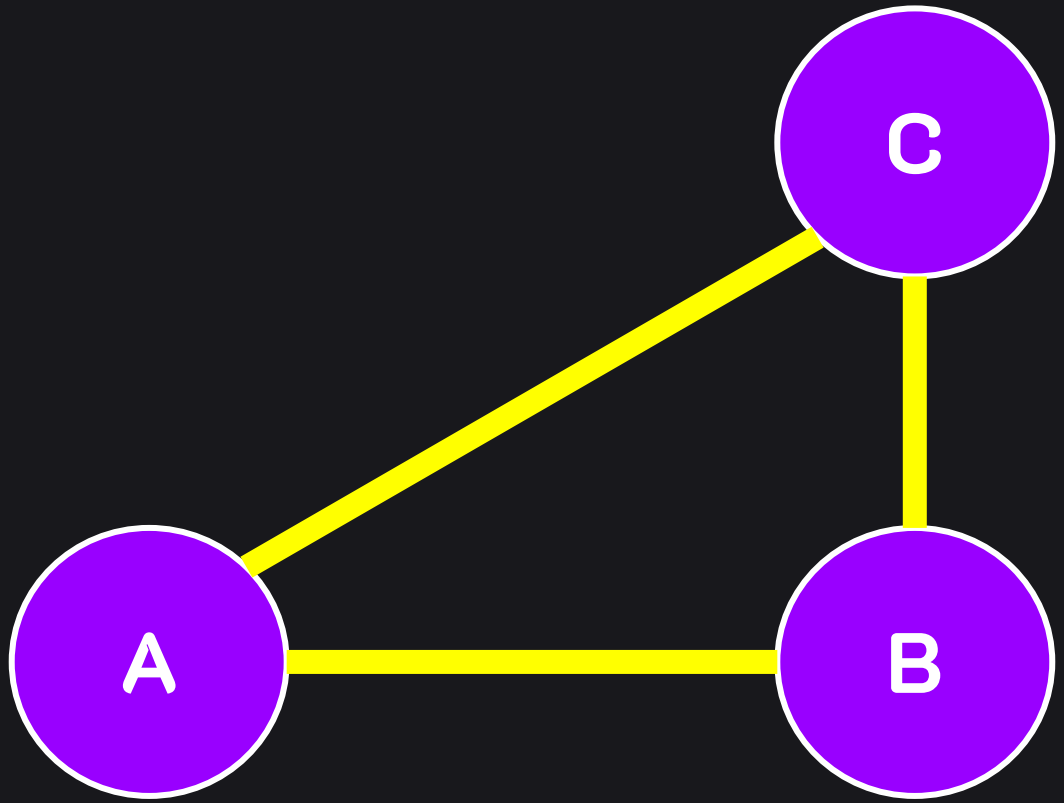
เวอร์เท็กซ์
(Vertex)

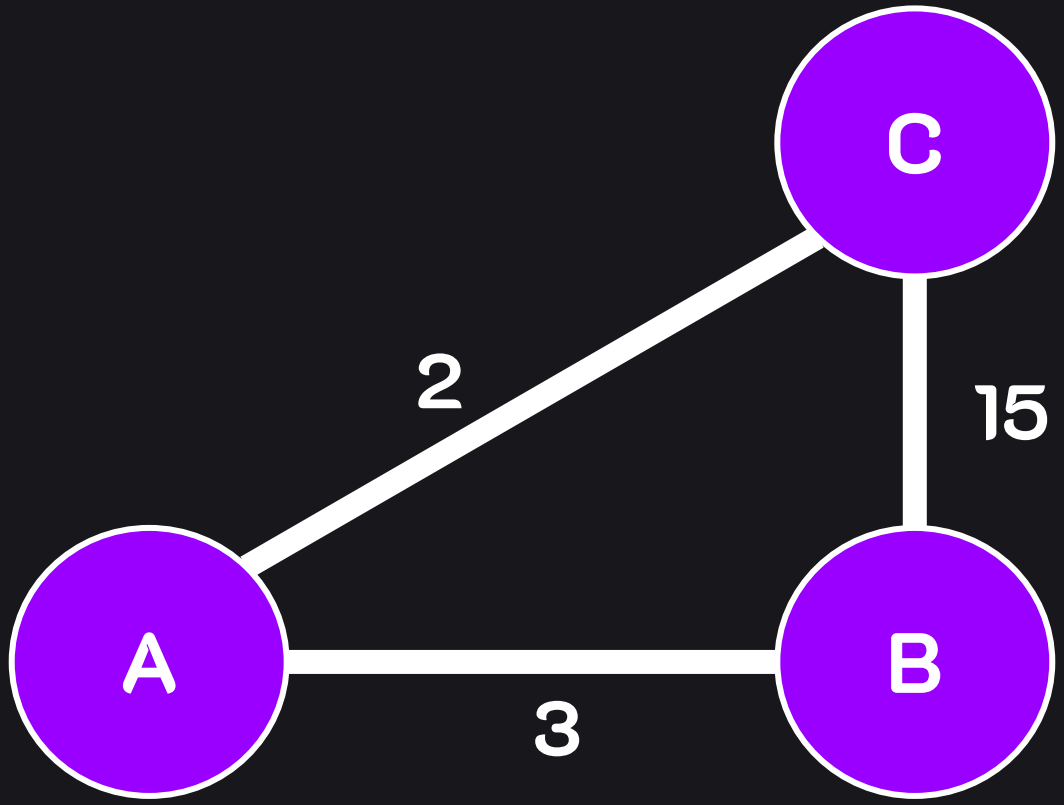
เวอร์เท็กซ์
(Vertex)

เวอร์เท็กซ์
(Vertex)


เอดจ์ (Edge)
A
B
เวอร์เท็กซ์
(Vertex)
เวอร์เท็กซ์
(Vertex)


C
A
B


C
A
B



ระยะห่างระหว่างเวอร์เท็กซ์จะเรียกว่า**น้ำหนักของกราฟ**

ประเภทของกราฟ (Graphs)

1. กราฟแบบไม่มีทิศทาง (Undirected Graph)
2. กราฟแบบมีทิศทาง (Directed Graph)

กราฟแบบไม่มีทิศทาง

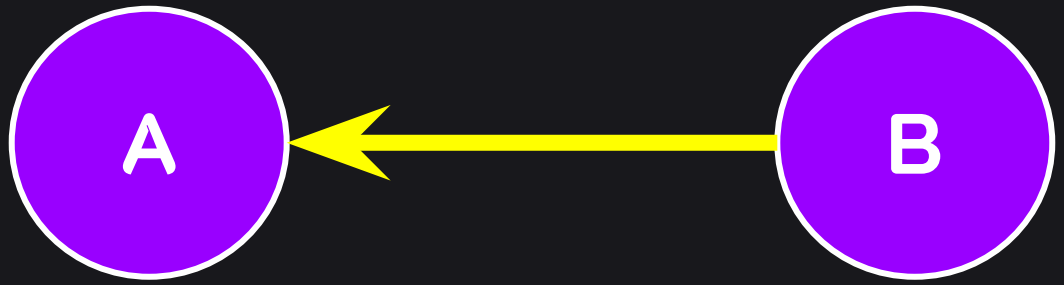
คือ กราฟที่ไม่มีการระบุทิศทาง

กราฟแบบไม่มีทิศทาง

คือ กราฟที่ไม่มีการระบุทิศทาง

กราฟแบบไม่มีทิศทาง

คือ กราฟที่ไม่มีการระบุทิศทาง



กราฟแบบไม่มีทิศทาง

คือ กราฟที่ไม่มีการระบุทิศทาง

กราฟแบบไม่มีทิศทาง

คือ กราฟที่ไม่มีการระบุทิศทาง (มีความสัมพันธ์แบบ 2 ทิศทาง)

กราฟแบบมีทิศทาง

คือ กราฟที่มีการระบุทิศทางแต่ละเส้นจะมีลูกศรกำกับเพื่อบอกทิศทาง ซึ่งจะสามารถเดินทางไปตามที่หัวลูกศรอยู่เท่านั้น

Adjacency Matrix

การแทนกราฟ

คือ การจัดการกราฟ เพื่อให้เห็นภาพรวมของกราฟและสามารถทำงานกับกราฟได้ง่ายมากยิ่งขึ้น ซึ่งมี 2 วิธี คือ

- Adjacency Matrix
- Adjacency List

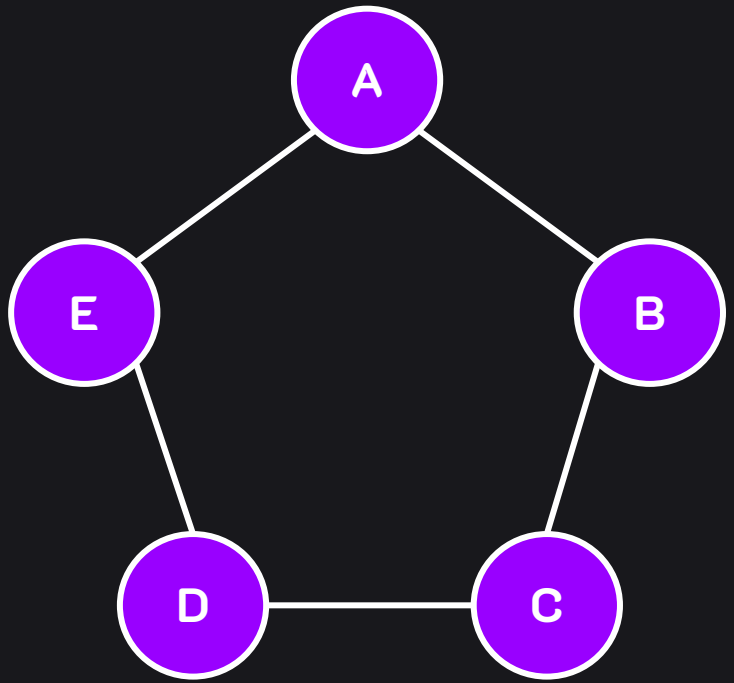
Adjacency Matrix

คือ การเก็บข้อมูลหรือแจกแจงความสัมพันธ์ภายในกราฟในลักษณะตาราง (Matrix)

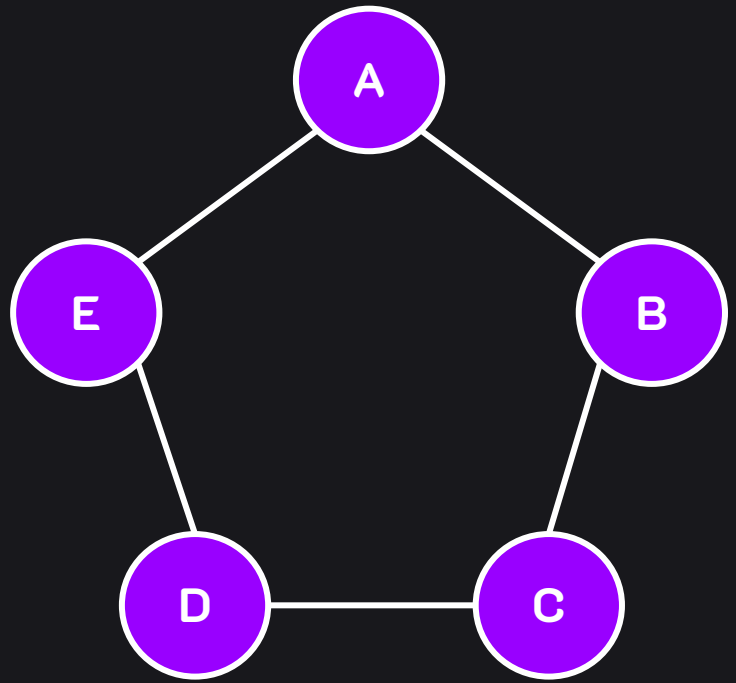
ข้อกำหนด

- ถ้ามีเส้นเชื่อมโยงระหว่าง Vertex ให้เก็บค่า 1
- ถ้าไม่มีเส้นเชื่อมโยงระหว่าง Vertex จะเก็บค่า 0

Adjacency Matrix

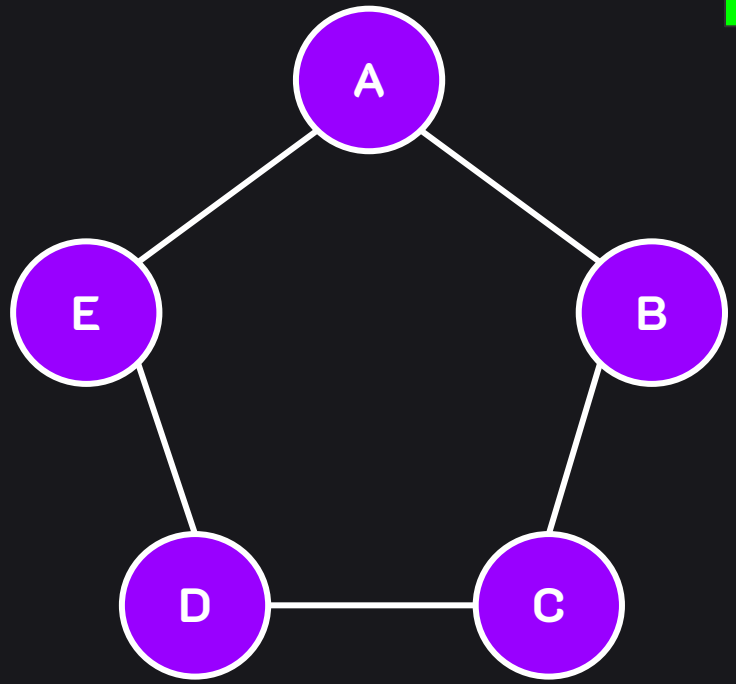


Adjacency Matrix



	A	B	C	D	E
A					
B					
C					
D					
E					

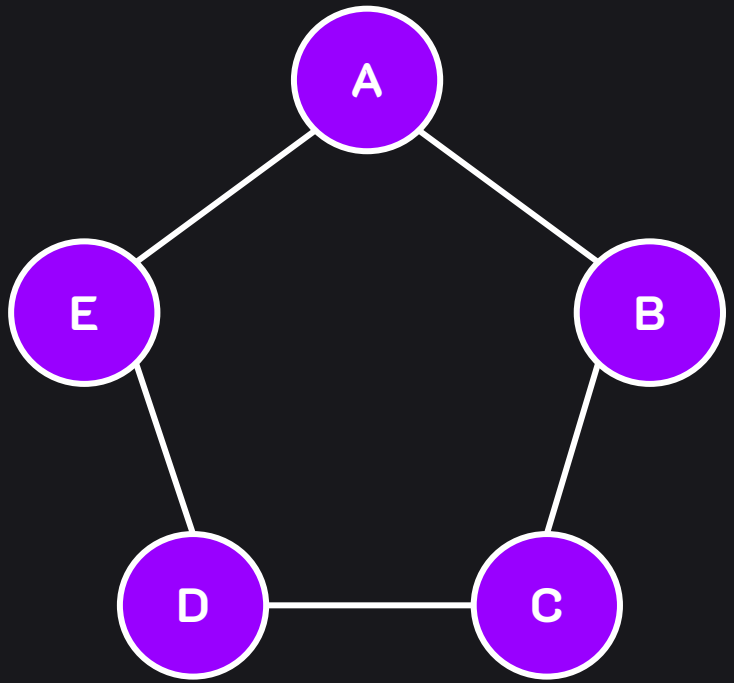
Adjacency Matrix



Vertex

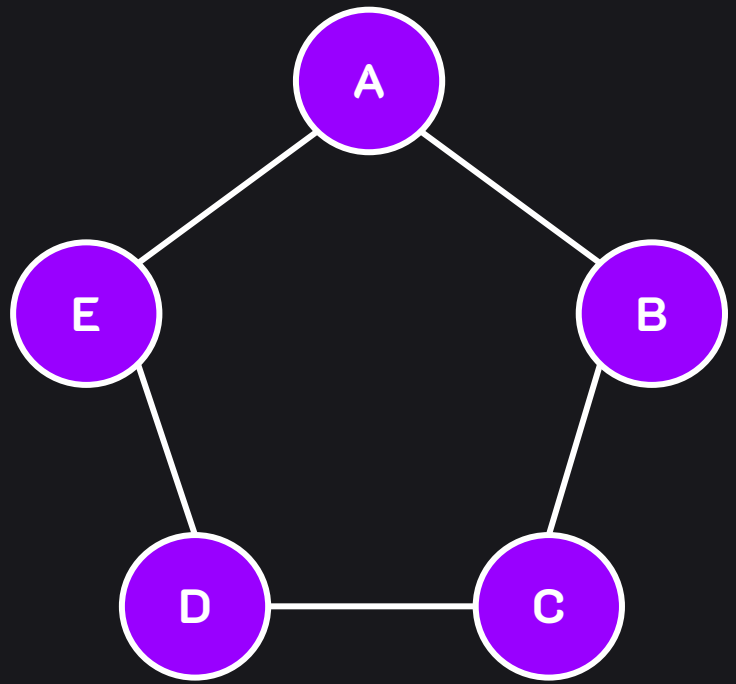
	A	B	C	D	E
A					
B					
C					
D					
E					

Adjacency Matrix



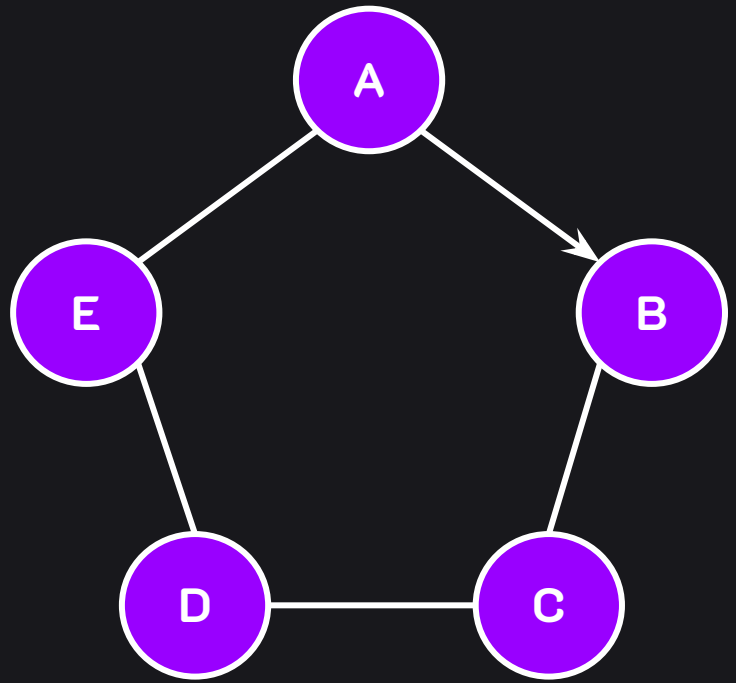
	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency Matrix (แบบมีทิศทาง)



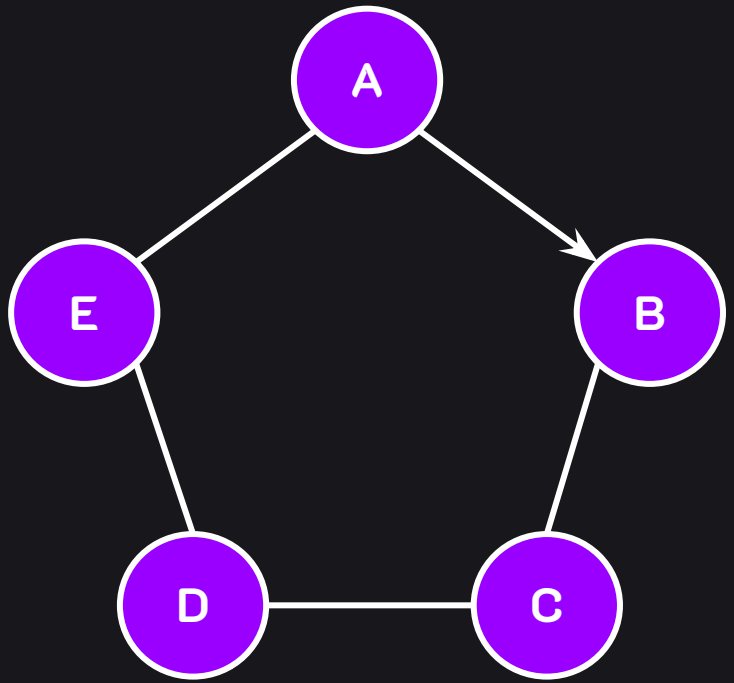
	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency Matrix (แบบมีทิศทาง)



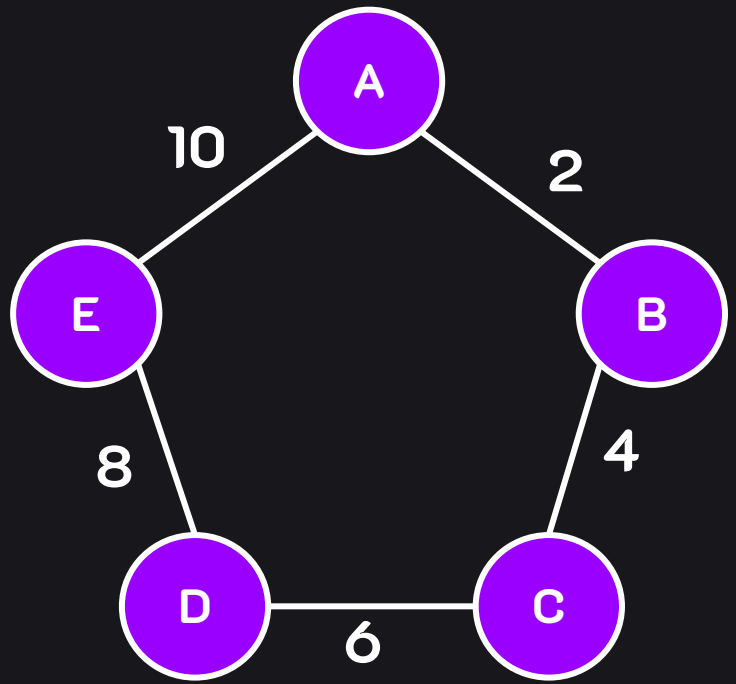
	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency Matrix (แบบมีทิศทาง)



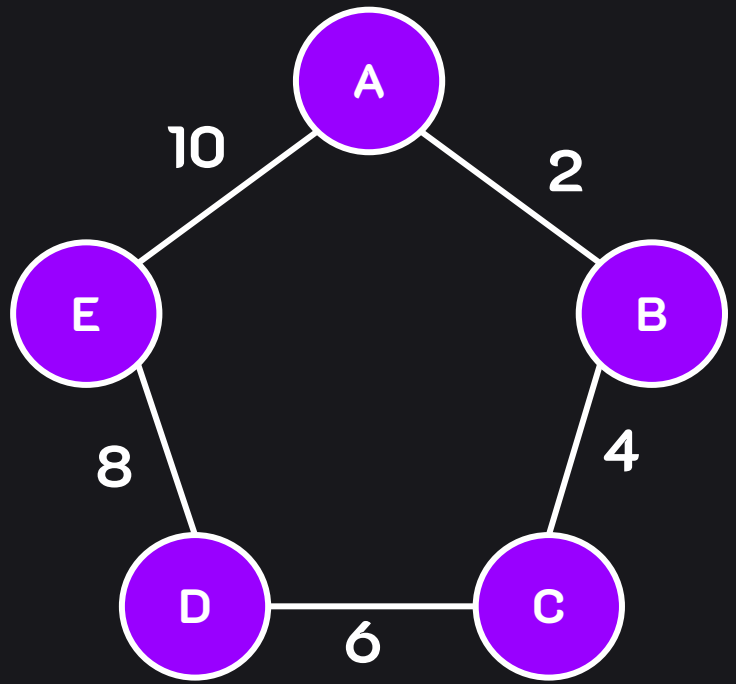
	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency Matrix (แบบมีน้ำหนัก)



	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

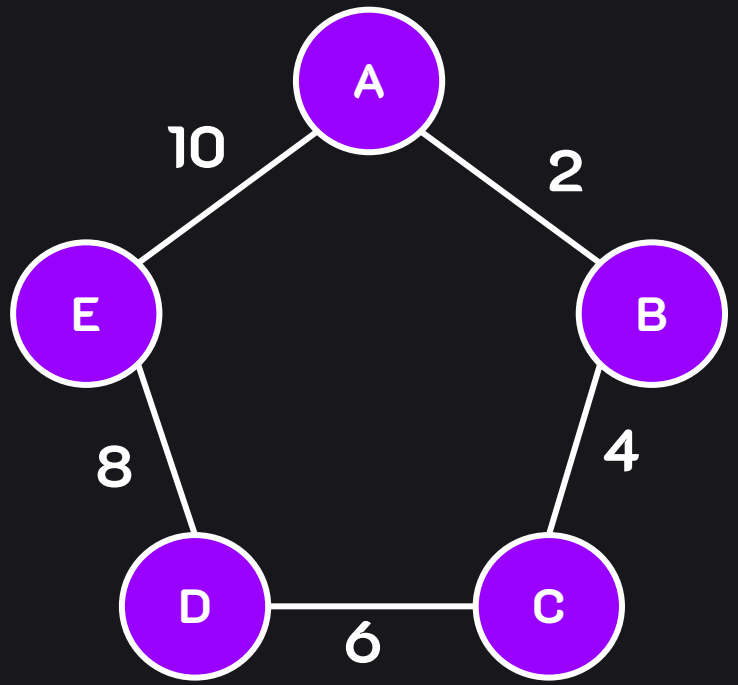
Adjacency Matrix (แบบมีน้ำหนัก)



แทนเลขน้ำหนักแต่ละเส้นลงไปในตาราง

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

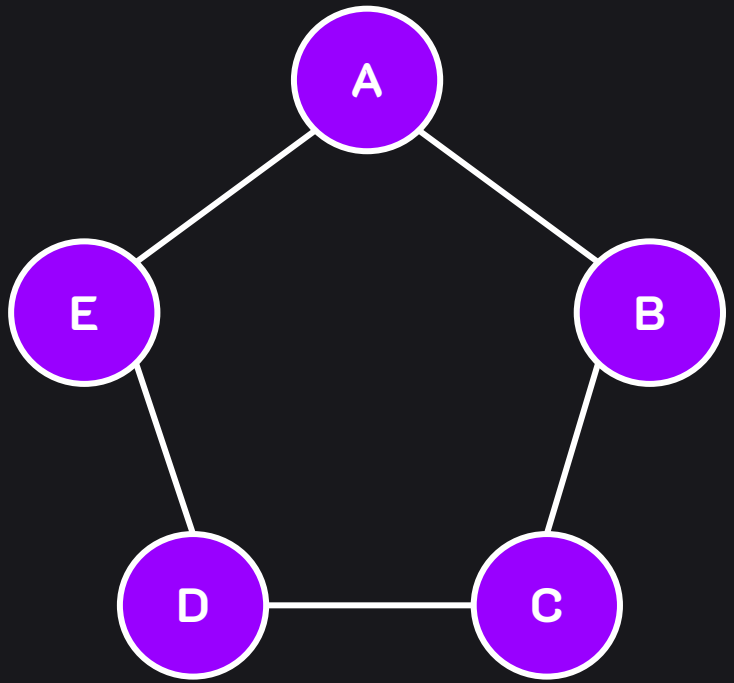
Adjacency Matrix (แบบมีน้ำหนัก)



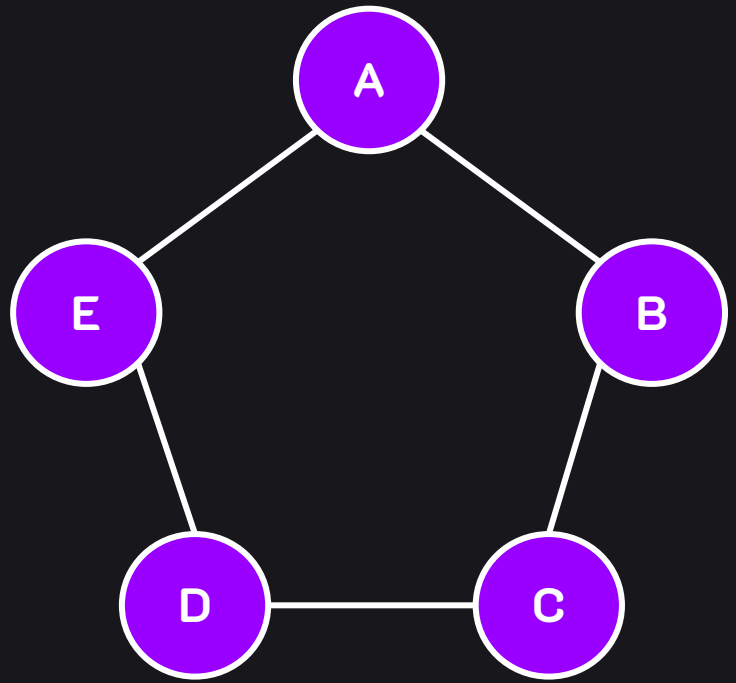
	A	B	C	D	E
A	0	2	0	0	10
B	2	0	4	0	0
C	0	4	0	6	0
D	0	0	6	0	8
E	10	0	0	8	0

Adjacency List

Adjacency List



Adjacency List




```
{
    A : ["B" , "E"],
    B : ["A" , "C"],
    C : ["B" , "D"],
    D : ["C" , "E"],
    E : ["A" , "D"]
}
```

Adjacency Matrix

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency List

```
{
    A : [“B” , “E”],
    B : [“A” , “C”],
    C : [“B” , “D”],
    D : [“C” , “E”],
    E : [“A” , “D”]
}
```

ช่องทางการสนับสนุน

🕵️ ช่องยูทูป :https://www.youtube.com/c/KongRuksiamOfficial

🎓 คอร์สเรียน : https://www.udemy.com/user/kong-ruksiam/

🎯 แฟนเพจ : https://www.facebook.com/KongRuksiamTutorial/