SCIENCE
SILPAKORN UNIVERSITY

ค่ายโอลิมปิก สอวน. 2

Sorting and Its Applications

อ.ดร.ภิญโญ แท้ประสาทสิทธิ์

ภาควิชาคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร

taeprasartsit_p@silpakorn.edu

หัวข้อเนื้อหา

SCIENCE
SILPAKORN UNIVERSITY

- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - เรียงสตรัคหรือวัตถุจากคลาส
- วิธีที่มันเร็วขึ้นกว่าเดิม
  - Priority Queue and Heap sort
  - Merge sort
  - Quick sort
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

พื้นฐาน: จัดเรียงตัวเลขในอาเรย์

- สมมติว่า เรามีเลขในอาเรย์ทั้งหมด 8 ค่า ดังนี้ 2 8 4 7 6 1 3 5
  - แต่เราต้องการให้มันเรียงจากน้อยไปมากเป็น 1 2 3 4 5 6 7 8
  - และเราอยากจะเรียงโดยที่ไม่ต้องสร้างอาเรย์สำหรับเก็บผลลัพธ์เพิ่ม
  - คือจากข้อมูลเข้าถึงผลลัพธ์ เราจะไม่สร้างอาเรย์ใด ๆ เพิ่มขึ้นมาเลย
- เราจะขอเริ่มจากวิธีจัดเรียงที่ง่ายที่สุด ที่ทุกคนมักคิดออกด้วยตัวเอง
  - วิธีนั้นก็คือ Selection sort
  - อาศัย**การสลับค่า** โดยการวิ่งหาค่าที่มากที่สุด แล้วเอามันไปไว้ทางด้านปลาย
  - พูดง่าย ๆ ก็คือในตอนแรก เราจะสลับ 8 กับ 5 ให้ได้ผลลัพธ์เป็น 2 **5** 4 7 6 1 3 **8** นั่นคือเลขด้านท้ายอาเรย์จะอยู่ในตำแหน่งที่ต้อง
- **การสลับค่า**เป็นพื้นฐานที่ต้องใช้ในแทบทุกวิธีจัดเรียงตัวเลข

ทบทวนการสลับค่าตัวเลข

- ก่อนหน้านี้ หากเรามีค่าตัวเลขในตัวแปร x กับ y เช่น

```
int x = 7;
int y = 5;
```

- จากนั้น เราต้องการสลับให้ค่า x กลายเป็น y และค่า y กลายเป็น x
  - โดยสมมติว่าเราไม่ทราบล่วงหน้าว่า x และ y จะมีค่าเท่าใด
  - เพราะอาจจะเป็นค่าที่ผู้ใช้กรอกเข้ามา
  - วิธีที่นิยมใช้ก็คือการสร้างตัวแปรชั่วคราวขึ้นมาช่วยเก็บค่าก่อนการเขียนทับ

```
int temp = x; // เก็บค่า x ไว้ก่อน
x = y;        // ค่า x หายไปที่ตรงนี้ แต่เรามีสำรองไว้ใน temp
y = temp;
```

แล้วถ้าจะสลับค่าในอาเรย์ล่ะ

- ต้องใช้ temp เหมือนกัน แต่ x กับ y จะกลายเป็นช่องข้อมูลในอาเรย์แทน
- เราจะต้องตั้งสติว่า**ตำแหน่ง**ที่จะสลับคือที่ใด ซึ่งตำแหน่งที่ว่าก็คืออินเด็กซ์ในอาเรย์นั่นเอง
- ขอยกตัวอย่างจากข้อมูลอาเรย์อันเดิม

```
int A[] = {2, 8, 4, 7, 6, 1, 3, 5};
```

- ต่อมาเราจะสลับเลข 8 กับ 5 ซึ่งมีอินเด็กซ์เป็น 1 และ 7 ดังนั้นเราจึงเขียนว่า

```
int temp = A[1];
A[1] = A[7];
A[7] = temp;
```

เอาไปใช้จริงใน Selection Sort

- Selection sort คือการ sort ที่มุ่งเน้นการเลือกเลขที่มีค่ามากที่สุดในอาเรย์ และสลับย้ายไปตำแหน่งที่ถูกต้อง (ซึ่งในที่นี้ก็คือด้านท้ายอาเรย์)
- ดังนั้นถ้าอาเรย์มีอยู่ทั้งหมด N ตัว เราก็จะเริ่มจากการตามหาค่าสูงสุด และจำตำแหน่งที่มีค่าสูงสุดนั้นไว้ด้วย
  - หมายความว่าเราต้องจำทั้งค่าสูงสุดและตำแหน่งค่าสูงสุดไว้

โค้ดสำหรับการตามหาค่าสูงสุดพร้อมทั้งบันทึกค่าที่เกี่ยวข้อง

```
int max = A[0];
int index = 0;
for(int i = 1; i < 8; ++i) {
    if(A[i] > max) {
        max = A[i];
        index = i;
    }
}
```

สลับค่าในตำแหน่งเป้าหมาย

- จากหน้าที่แล้ว เราเก็บตำแหน่งที่จะสลับเปลี่ยนไว้ที่ index ถ้าหากเราจะใช้โค้ดการสลับค่าแบบเดิม ๆ เราก็เขียนได้ว่า
- อย่าลืมว่าจุดประสงค์ก็คือย้ายค่าสูงสุดไปไว้ด้านปลายอาเรย์

```
int temp = A[index];
A[index] = A[7];
A[7] = temp;
```

- แต่ดูดี ๆ เราจะพบว่าไอ้ค่า temp มันก็คือ max ดังนั้นโค้ดที่ดีกว่าก็คือ

```
A[index] = A[7];
A[7] = max;
```

สภาพโค้ด Selection Sort ณ เวลานี้


```
int A[] = {2, 8, 4, 7, 6, 1, 3, 5};

int max = A[0];
int index = 0;
for(int i = 1; i < 8; ++i) {
    if(A[i] > max) {
        max = A[i];
        index = i;
    }
}

A[index] = A[7];
A[7] = max;
```

ทำ Selection Sort ต่อไปเรื่อย ๆ

- ณ ตอนนี้ เราพบว่าตำแหน่งด้านท้ายของอาเรย์มีค่าที่ถูกต้องแล้ว ทว่าเราต้องการให้มันถูกต้องทุกตำแหน่ง ไม่ใช่เฉพาะด้านท้าย
- แบบนี้แสดงว่างานยังไม่เสร็จต้องทำต่อไปอีก แต่จะทำอย่างไรดี
- แนวคิดจำนวนมากในการเรียงข้อมูลก็คือ “อะไรที่ถูกต้องแล้ว อย่าไปแตะต้องมันอีก ให้จดจ่ออยู่กับส่วนที่ยังไม่ถูกต้องก็พอ”
- ถ้าเราไม่ไปแตะต้องอาเรย์ที่ด้านปลาย ก็จะดูประหนึ่งว่าอาเรย์ของเราลดขนาดลงมาอีก 1 ช่อง และเราจะตามหาค่าสูงสุดเฉพาะจาก 7 ช่องแรก
  - นั่นคือขนาดปัญหาลดลงไป 1 ช่องข้อมูล
  - ถ้าจะให้จัดเรียงให้เสร็จ ขนาดปัญหาก็จะต้องลดลงไปจนเหลือ 1 ช่องข้อมูล
  - ถ้ามีช่องเดียว ตัวเลขก็จะอยู่ในตำแหน่งที่ถูกต้องโดยปริยาย

ทำ Selection Sort ต่อไปเรื่อย ๆ

- ณ ตอนนี้ เราพบว่าตำแหน่งด้านท้ายของอาเรย์มีค่าที่ถูกต้องแล้ว ทว่าเราต้องการให้มันถูกต้องทุกตำแหน่ง ไม่ใช่เฉพาะด้านท้าย
- แบบนี้แสดงว่างานยังไม่เสร็จต้องทำต่อไปอีก แต่จะทำอย่างไรดี ?
- แนวคิดจำนวนมากในการเรียงข้อมูลก็คือ “**อะไรที่อยู่ถูกที่แล้ว อย่าไปแตะต้องมันอีก ให้ดูเฉพาะส่วนที่ยังอยู่ไม่ถูกที่ก็พอ**”
- ถ้าเราไม่ไปแตะต้องอาเรย์ที่ด้านปลาย ก็จะดูประหนึ่งว่าอาเรย์ของเราลดขนาดลงมาอีก 1 ช่อง และเราจะตามหาค่าสูงสุดเฉพาะจาก 7 ช่องแรก
  - นั่นคือขนาดปัญหาลดลงไป 1 ช่องข้อมูล
  - ถ้าจะให้จัดเรียงให้เสร็จ ขนาดปัญหาก็จะต้องลดลงไปจนเหลือ 1 ช่องข้อมูล
  - ถ้ามีช่องเดียว ตัวเลขก็จะอยู่ในตำแหน่งที่ถูกต้องโดยปริยาย

ดังนั้นเราจะต้องวนทำแบบเดิม 8-1 = 7 รอบ

- เอาลูปมาครอบด้านนอกให้มันวนจัดเลขไปด้านท้ายอาเรย์
  - ตำแหน่งด้านท้ายนี้ คือตำแหน่งที่ยังไม่แน่ว่าเลขจะถูกต้อง
  - ตามตัวอย่างเดิมในรอบแรกคือตำแหน่ง A[7]
  - ในรอบต่อ ๆ มาตำแหน่งที่ต้องเอาเลขใหญ่สุดไปวางก็คือ A[6], A[5], ...
- แสดงว่าเราต้องรู้ว่าเมื่อวนไปแต่ละรอบแล้วตำแหน่งท้ายสุดคือเท่าใด
  - สังเกตว่าในรอบแรก ท้ายสุดคือ 7, รอบสองคือ 7-1 = 6, รอบสามคือ 7-2 = 5
  - ดังนั้นยิ่งรอบเพิ่ม เลขสุดท้ายในอาเรย์ก็ต้องลด
- เมื่อคิดได้แบบนี้แล้ว ลูปทางด้านนอกที่เอามาครอบก็จะเป็นแบบหน้าถัดไป

Selection Sort ที่สมบูรณ์

SCIENCE
SILPAKORN UNIVERSITY

```
for(int rnd = 0; rnd < 7; ++rnd) {
  int max = A[0];
  int index = 0;
  for(int i = 1; i < 8 - rnd; ++i) {
    if(A[i] > max) {
      max = A[i];
      index = i;
    }
  }

  A[index] = A[7-rnd];
  A[7-rnd] = max;
}
```

Selection Sort สำหรับเลขทั้งหมด N ตัว


```
for(int rnd = 0; rnd < N-1; ++rnd) {
  int max = A[0];
  int index = 0;
  for(int i = 1; i < N-rnd; ++i) {
    if(A[i] > max) {
      max = A[i];
      index = i;
    }
  }

  A[index] = A[N-rnd-1];
  A[N-rnd-1] = max;
}
```

Insertion Sort

SCIENCE
SILPAKORN UNIVERSITY

- เป็นการเรียงข้อมูลแบบช้าที่นิยมมากวิธีหนึ่ง
  - นิยมใช้สำหรับการเรียงข้อมูลที่มีจำนวนไม่มากนัก โดยเรามักใช้มันผสมกับการเรียงที่มีความเร็วสูงกว่า
  - เช่นใช้ใน Quick sort เมื่อข้อมูลที่ต้องจัดเหลืออยู่เล็กน้อย
- ถ้าข้อมูลถูก sort มาแล้วก่อนหน้ามาก ๆ insertion sort จะเร็วเอาเรื่อง
  - ใช้เวลาเป็น O(Nk) เมื่อ k คือระยะทางที่มากที่สุดจากตำแหน่งที่ถูกต้องของค่าแต่ละค่าในอาเรย์
- มีลักษณะที่ Stable คือถ้าข้อมูลเข้ามามีเลขซ้ำ เลขที่มาก่อนในอาเรย์ข้อมูลเข้า จะอยู่ก่อนเลขซ้ำในอาเรย์ผลลัพธ์ด้วย
  - สำคัญสำหรับการเรียงสตรัคและวัตถุบางประเภท

แนวคิดของ Insertion Sort

- แทนที่จะวิ่งหาตัวที่มากที่สุด แล้วเปลี่ยนตำแหน่งแค่ครั้งเดียวต่อรอบ
  - Insertion sort จะวิ่งไปเรื่อย ๆ พร้อมกับเปลี่ยนตำแหน่งไปด้วย ถ้าพบว่ามีการผิดลำดับในระหว่างทางที่วิ่งนั้น
  - วิ่งแต่ละรอบ ไม่ได้รับประกันว่าจะมีข้อมูลถูกย้ายไปอยู่ในตำแหน่งที่ถูก แต่รับประกันว่าอย่างน้อยจะต้องขยับเข้าใกล้ตำแหน่งที่ถูกต้องมากขึ้น
- ในรอบแรกจะทำเหมือนกับว่าอาเรย์มีเฉพาะสองช่องทางซ้าย และจะเรียงสองช่องทางซ้ายให้ถูกต้อง (เมื่อนับเฉพาะอาเรย์สองช่องแรก)
  - ส่วนรอบต่อมาทำเหมือนกับว่าอาเรย์มีสามช่องทางซ้าย และจะเรียงอาเรย์สามช่องนี้ให้ถูกต้อง
  - รอบต่อมาจะทำเหมือนกับว่าอาเรย์มีสี่ช่องทางซ้าย และจะเรียงสี่ช่องทางซ้ายให้ถูกต้อง และทำเช่นนี้ไปเรื่อยจนครบทุกช่อง

ตัวอย่างแสดงแนวคิดของ Insertion Sort

- สมมติว่าในตอนแรกมีเลขเป็น 6 5 3 1 8 7 2 4
  - เนื่องจากในรอบแรก เราสมมติว่ามีแค่สองช่องซ้าย ดังนั้นเราจะเรียงเฉพาะสองช่องนี้ให้ถูกต้อง ทำให้ได้ผลลัพธ์เป็น 5 6 3 1 8 7 2 4
  - ต่อมาสมมติว่ามีสามช่อง เราจะเริ่มดูที่เลข 3 และเปรียบเทียบกับเลข 6 ซึ่งจะเห็นได้ว่า เลข 6 อยู่ผิดที่ ดังนั้นเราจะขยับเลข 6 ไปไว้ที่ช่องที่สาม
  - แต่เราจะยังไม่ย้ายเลข 3 เราจะเอามันไปลุยหาตำแหน่งที่ถูกต้องก่อน ซึ่งพอเทียบกับเลข 5 ก็จะพบว่าลำดับยังไม่ถูกต้อง จึงย้ายเลข 5 ไปไว้ช่องที่สอง
  - จุดนี้ เราพบตำแหน่งที่ถูกต้องของเลข 3 แล้ว ดังนั้นเราจะใส่มันไว้ในตำแหน่งแรก ทำให้ได้ผลลัพธ์เป็น 3 5 6 1 8 7 2 4
  - รอบต่อมาก็จะดูที่เลข 1 แล้วตรวจหาตำแหน่งที่จะใส่แทรก 1 เข้าไปได้อย่างถูกต้อง ซึ่งก็จะได้ผลลัพธ์ในรอบนี้เป็น 1 3 5 6 8 7 2 4
- ซึ่งการหาตำแหน่งในการแทรกเลขเข้าไปนี้แหละคือที่มาของชื่อ Insertion

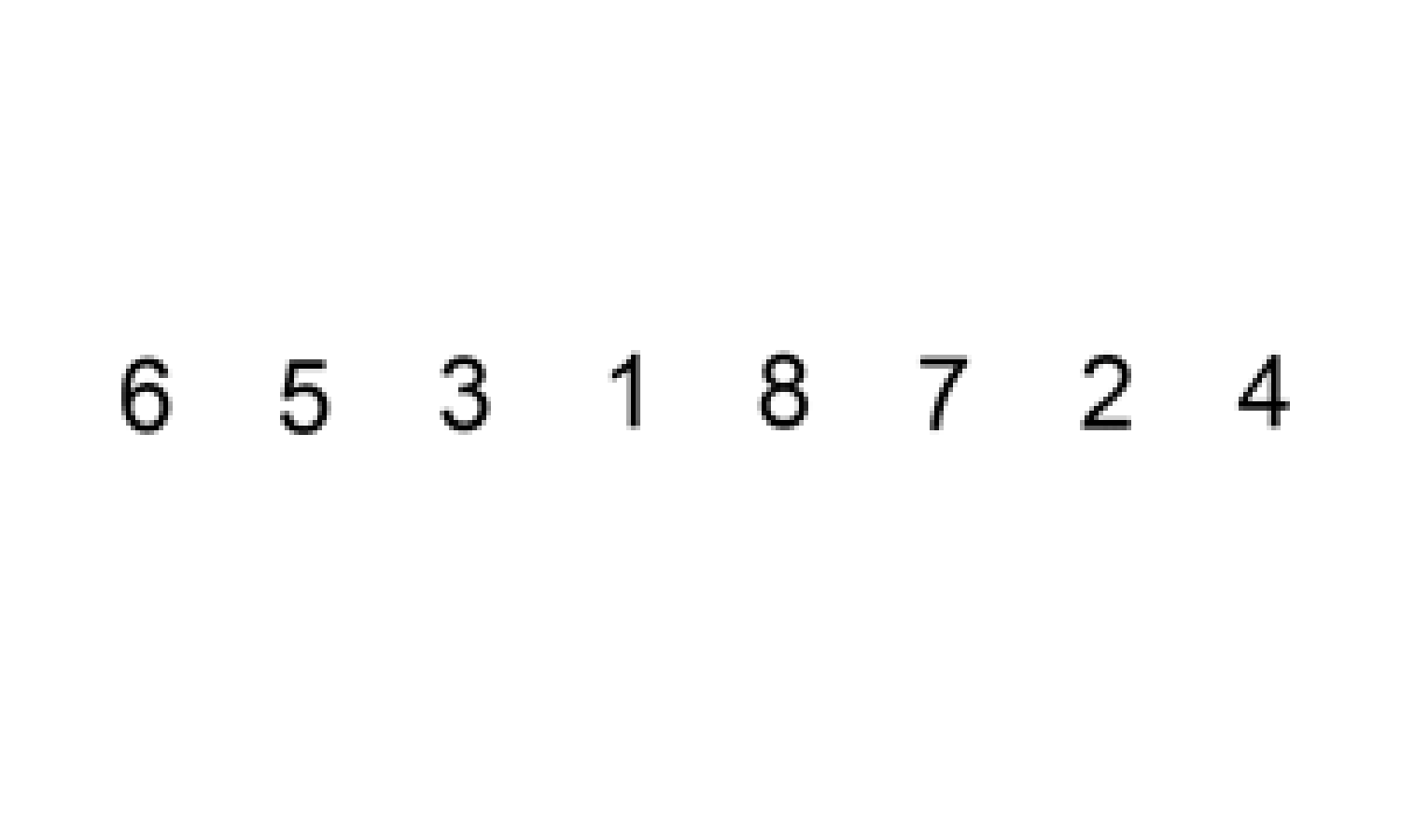
แล้วมันเร็วกว่า Selection sort ไหม

- ในตัวอย่างที่เห็น เราจะพบว่ามันต้องวิ่งเลื่อนตัวเลขขึ้นมาหลายรอบ
  - ต้องออกแรงเปลี่ยนค่าตัวเลขบ่อย ๆ ทำให้มันช้า เพราะ `selection sort` เปลี่ยนตำแหน่งเลขแค่ครั้งเดียวต่อรอบ
  - แต่โดยรวมแล้ว `insertion` เร็วกว่า `selection`
- พิจารณารอบการคำนวณถัดไป `1 3 5 6 8 7 2 4`
  - เรากำลังจะหาตำแหน่งที่ถูกต้องของเลข `8` แต่พอเทียบกับ `6` เราก็พบว่าเลขมันถูกลำดับอยู่แล้ว ดังนั้นรอบนี้เปรียบเทียบเลขครั้งเดียวก็รู้ผลแล้ว
  - ที่เรามั่นใจว่าไม่ต้องไปตรวจตำแหน่งอื่น ๆ อีกก็เพราะว่าเลขสี่ตัวทางซ้ายของอาเรย์เรียงตามลำดับมาก่อนหน้า จึงไม่ต้องเสียเวลาตรวจไปลึก ๆ
  - และการที่มันหาข้อสรุปในบางรอบได้เร็วมาก ทำให้โดยรวมมันเร็วกว่า `selection` เพราะ `selection` ต้องวนหาค่า `max` ยาวมาก

ดูต่อจนจบ

- ถ้าจะให้คิดต่อ ก็จะพบว่าตอนนี้เราต้องพิจารณาหาตำแหน่งที่ถูกต้องของเลข 7 จากข้อมูลปัจจุบันซึ่งก็คือ 1 3 5 6 8 **7** 2 4
  - ซึ่งตอนตรวจจังหวะแรก จะพบว่าเลข 8 ต้องถูกเลื่อนขึ้นมา
  - ส่วนจังหวะที่สองเป็นการเทียบ 6 กับ 7 ซึ่งพบว่าอยู่ถูกตำแหน่ง แบบนี้แสดงว่าตรวจสองครั้งก็พอ และมีการย้ายเลขแค่ครั้งเดียว
  - รอบนี้จึงทำงานเสร็จเร็ว และได้ผลลัพธ์เป็น 1 3 5 6 **7** 8 2 4
- แต่พอจะหาตำแหน่งแทรกเลข 2 เราจะต้องออกแรงมากหน่อย
  - ต้องตรวจไปจนถึงตัวแรก และได้ผลลัพธ์เป็น 1 **2** 3 5 6 7 8 4
- ส่วนตัวสุดท้าย ออกแรงมาจนถึงตรงกลาง และได้ผลลัพธ์เป็น 1 2 3 **4** 5 6 7 8

ภาพประกอบการทำงานของ Insertion sort



[ขอบคุณภาพจาก Wikipedia.org]

Code ของ Insertion Sort

ดูแนวคิดแล้วอาจจะคิดว่าโค้ดจะยาว แต่แท้จริงแล้วมีเพียงไม่กี่บรรทัด

```
for(int i = 0; i < N-1; ++i) {
    int j = i + 1;
    int val = A[j];
    while(j > 0 && A[j-1] > val) {
        A[j] = A[j-1];
        j--;
    }
    A[j] = val;
}
```

หัวข้อเนื้อหา

- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - **เรียงสตรัคหรือวัตถุจากคลาส**
- วิธีที่มันเร็วขึ้นกว่าเดิม
  - Priority Queue and Heap sort
  - Merge sort
  - Quick sort
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

เรียงสตรัคหรือวัตถุจากคลาส

- การเข้าใจวิธีเรียงตัวเลขเป็นสิ่งที่สำคัญก็จริง แต่การประยุกต์ใช้งานจริงนั้น
  - เรามักจะต้องเรียงข้อมูลในสตรัคหรือวัตถุ
  - เช่น ต้องการประกาศรายชื่อนักเรียนตามลำดับคะแนน
  - นั่นคือเราเรียงตัวเลขคะแนน แต่การสลับค่าต่าง ๆ ก็จะต้องทำให้ชื่อนักเรียนที่เป็นเจ้าของคะแนนเปลี่ยนลำดับตามคะแนนที่ถูกย้ายตำแหน่งในอาเรย์ด้วย
- ดังนั้น เราจะจัดข้อมูลคะแนนและชื่อลงในสตรัคด้วยกัน
  - ทำการเปรียบเทียบค่าตัวเลขในสตรัคตามปรกติ
  - เวลาจะย้ายข้อมูล เราจะย้ายไปทั้งสตรัค แทนที่จะย้ายแค่ตัวเลข

หมายเหตุ เพื่อความกระชับ เราจะใช้คำว่าสตรัคแทนพวกวัตถุและคลาสไปด้วยกัน เนื่องจากผู้เรียนมีแนวโน้มจะเข้าใจตัวภาษาซีมาก่อนจากค่าย 1

สตรัคสำหรับเรียงคะแนนนินจา

สมมติว่านักเรียนของเราคือผู้เข้าศึกษาในโรงเรียนนินจา และเราต้องการเก็บเฉพาะชื่อ (string) และคะแนนสอบ เราจะได้สตรัคเก็บข้อมูลดังนี้

```
#include <string>
using namespace std;

typedef struct ninja {
    string name;
    int score;
} Ninja;
```

[โค้ดสีเทาคือสิ่งที่ควรใส่มาด้วย แต่เนื่องจากเราต้องการเน้นที่สตรัคจึงขอใส่สีจาง ๆ อย่างสีเทาเอาไว้แทน]

เรียงคะแนนนินจา

เตรียมข้อมูลกันก่อน

```
Ninja A[] = {{"Naruto", 38}, {"Itachi", 100},
             {"Sasuke", 92}, {"Sakura", 84},
             {"Kakashi", 93}};
const int N = sizeof(A)/sizeof(Ninja);
```

เปิดฉากลุย

```
for(int i = 0; i < N-1; ++i) {
        int j = i + 1;
        Ninja val = A[j];
        while(j > 0 && A[j-1].score > val.score) {
            A[j] = A[j-1];
            j--;
        }
        A[j] = val;
}
```

ความเร็วของ Selection/Insertion Sort

- สังเกตจากลูปสองชั้น ซึ่งลูปด้านในนั้น ในกรณีที่แย่ที่สุดจะต้องทำการเปรียบเทียบนับจากรอบแรกไปรอบสุดท้ายเป็น N-1, N-2, N-3, ..., 1 ครั้ง
  - ซึ่ง 1+2+3+...+N-1 $\Rightarrow \frac{(N-1)N}{2}$
  - หรือกล่าวได้ว่ามันมีบิ๊กโอเป็น $O(N^2)$
- ส่วนในกรณีที่ดีที่สุดนั้น เราจะพบว่า `Insertion sort` จะได้เปรียบ
  - ถ้าข้อมูลถูกเรียงไว้ตั้งแต่แรก เราจะทำการเปรียบเทียบเพียงรอบละ 1 ครั้ง
  - ดังนั้นประสิทธิภาพในกรณีที่ดีที่สุดคือ $\Omega(N)$
- ส่วน `selection sort` นั้น ไม่ว่าอย่างไรก็ต้องทำการเปรียบเทียบเป็นจำนวนเท่าเดิม
  - หรือไม่ก็ตั้งแก้โค้ดให้มันคอยตรวจการสลับลำดับเพิ่ม ซึ่งทำให้ภาระการคำนวณในกรณีทั่วไปเพิ่มขึ้นจากเดิมมาก

หัวข้อเนื้อหา

SCIENCE
SILPAKORN UNIVERSITY

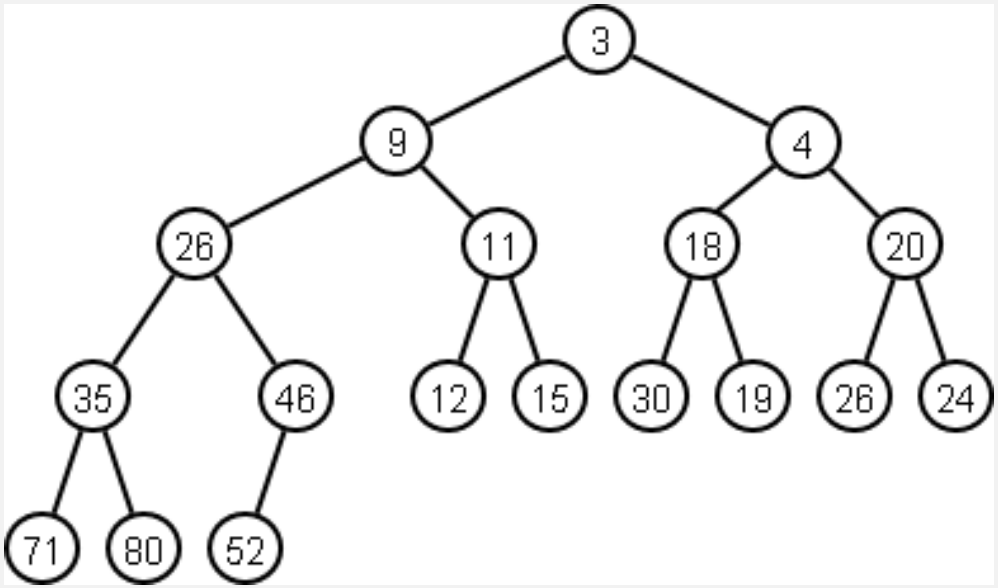
- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - เรียงสตรัคหรือวัตถุจากคลาส
- **วิธีที่มันเร็วขึ้นกว่าเดิม**
  - **Priority Queue and Heap sort**
  - Merge sort
  - Quick sort
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

วิธีที่เร็วขึ้นกว่าเดิม

- วิธีที่เร็วขึ้นกว่าเดิมนั้น อาจจะอาศัยโครงสร้างข้อมูลที่ดีขึ้น หรือมีอัลกอริทึมที่มีแนวคิดที่แหวกแนวออกไป
- อย่างไรก็ตาม ขีดจำกัดของวิธีการเรียงข้อมูลทั่วไปคือ $O(N \log N)$
  - “ทั่วไป” คือไม่ไปจำกัดว่าต้องเป็นจำนวนเต็มหรือมีเงื่อนไขพิเศษของข้อมูล
  - จะไม่มีทางทำให้บิ๊กโอมันดีกว่านี้ได้ มีการพิสูจน์ทางคณิตศาสตร์มาเรียบร้อย
  - ดูจากบิ๊กโอแล้ว เราจะนึกถึงพวกอัลกอริทึมแบบ **divide and conquer** หรือไม่ก็พวกโครงสร้างข้อมูลแบบ **balanced tree**
- ซึ่งก็เป็นเช่นนั้นจริง เพราะหากขาดการใช้อัลกอริทึมและโครงสร้างข้อมูลจำพวกนั้นแล้ว การเรียงข้อมูลทั่วไปจะยากที่จะได้บิ๊กโอ $O(N \log N)$

รู้จักกับ Priority Queue (Heap)

- Priority Queue หรือที่เรานิยมเรียกแบบง่าย ๆ ว่า Heap นั้น ...
  - มีโครงสร้างเป็นต้นไม้ที่สมดุลแบบเต็มที่ คือไม่มีกิ่งก้านแหว่งภายในจะมีเฉพาะชั้นล่างสุดที่ใบไม่เต็ม



  - ตอบคำถามได้เร็วมาก ว่าเลขที่มีค่าน้อยที่สุดเป็นเท่าใด ทำได้ใน $O(1)$ เพราะมันเก็บเลขที่น้อยสุดไว้ด้านบนตลอดเวลา

ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

คุณสมบัติอื่น ๆ ในฮีป

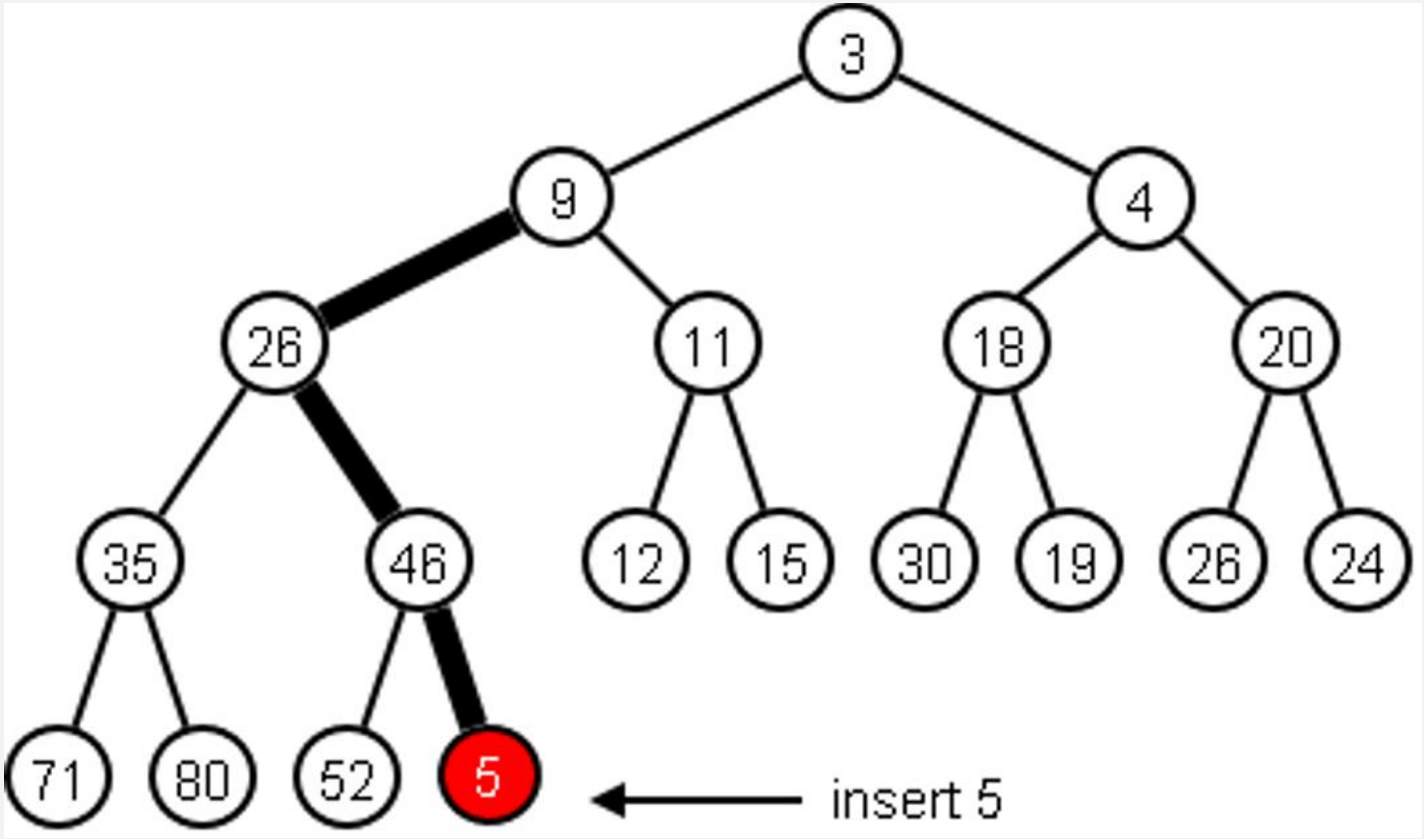
- การหาเลขที่น้อยที่สุดนั้นเร็วก็จริง แต่จะหาเลขทั่ว ๆ ไปจะใช้เวลา $O(N)$
- สามารถเติมเลขใหม่เข้าไปได้ใน $O(\log N)$
  - คำว่าเติมเลขใหม่เข้าไปนี้ จะต้องรักษาคุณสมบัติของฮีปไว้ได้
  - นั่นคือจะต้องจัดให้เลขน้อยที่สุดอยู่ด้านบนสุดของฮีปให้ได้
- ถ้าจะลบโหนดออกไป (ก็ลบเลขนั่นแหละ) หากรู้ตำแหน่งโหนดแล้ว จะใช้เวลาในการลบและจัดระเบียบเพื่อให้เลขน้อยสุดอยู่ด้านบนเป็น $O(\log N)$
  - แต่ถ้าไม่รู้ว่าโหนดอยู่ไหน ก็ต้องวิ่งหากันทั้งฮีป ใช้เวลาหาเป็น $O(N)$
- โดยทั่วไป เลขที่เรารู้ตำแหน่งแน่ ๆ ก็คือเลขที่น้อยที่สุด เพราะอยู่บนสุด
  - ดังนั้นสิ่งที่ฮีปถนัดก็คือ 1. Find Min , 2. Insert Value, 3. Extract Min
  - รวมถึงการเปลี่ยนตัวเลขข้างในให้น้อยลงด้วย 4. Decrease Key

รักษาคุณสมบัติฮีป

- เรื่องท้าทายของโครงสร้างข้อมูลก็คือ การรักษาคุณสมบัติของโครงสร้างข้อมูลนั้น อย่างเช่นในฮีป เราต้องรับประกันว่าด้านบนน้อยสุด
- ต้องรับประกันเวลาในการทำงานของแต่ละการดำเนินการด้วย
- เพื่อที่จะให้ตอบเรื่องเลขน้อยได้เร็ว ๆ ฮีปจึงอยู่ภายใต้แนวคิดที่ว่า
  - จะรักษาเรื่อง **partial order** ไว้ คือจะทำให้โหนดที่อยู่ด้านบนมีค่าน้อยกว่า
  - นั่นคือรับประกันว่าโหนดพ่อ จะมีค่าน้อยกว่าโหนดลูกทั้งด้านซ้ายขวา
  - ถ้ารับประกันแบบนี้ไปเรื่อย ๆ สุดยอดของโหนดพ่อซึ่งก็คือรากของฮีปจะต้องเป็นเลขที่น้อยที่สุดแน่นอน
- ดังนั้นถ้าหากเจอเหตุการณ์ที่โหนดลูกมีค่าน้อยกว่าโหนดพ่อ แสดงว่ามันอยู่ผิดตำแหน่ง ถ้าจะรักษาคุณสมบัติฮีป ก็ต้องมีการสลับโหนดพ่อลูกกัน

รักษาคุณสมบัติฮีป เมื่อใส่ข้อมูลใหม่เข้าไป

ทดลองใส่เลข 5 เข้าไป

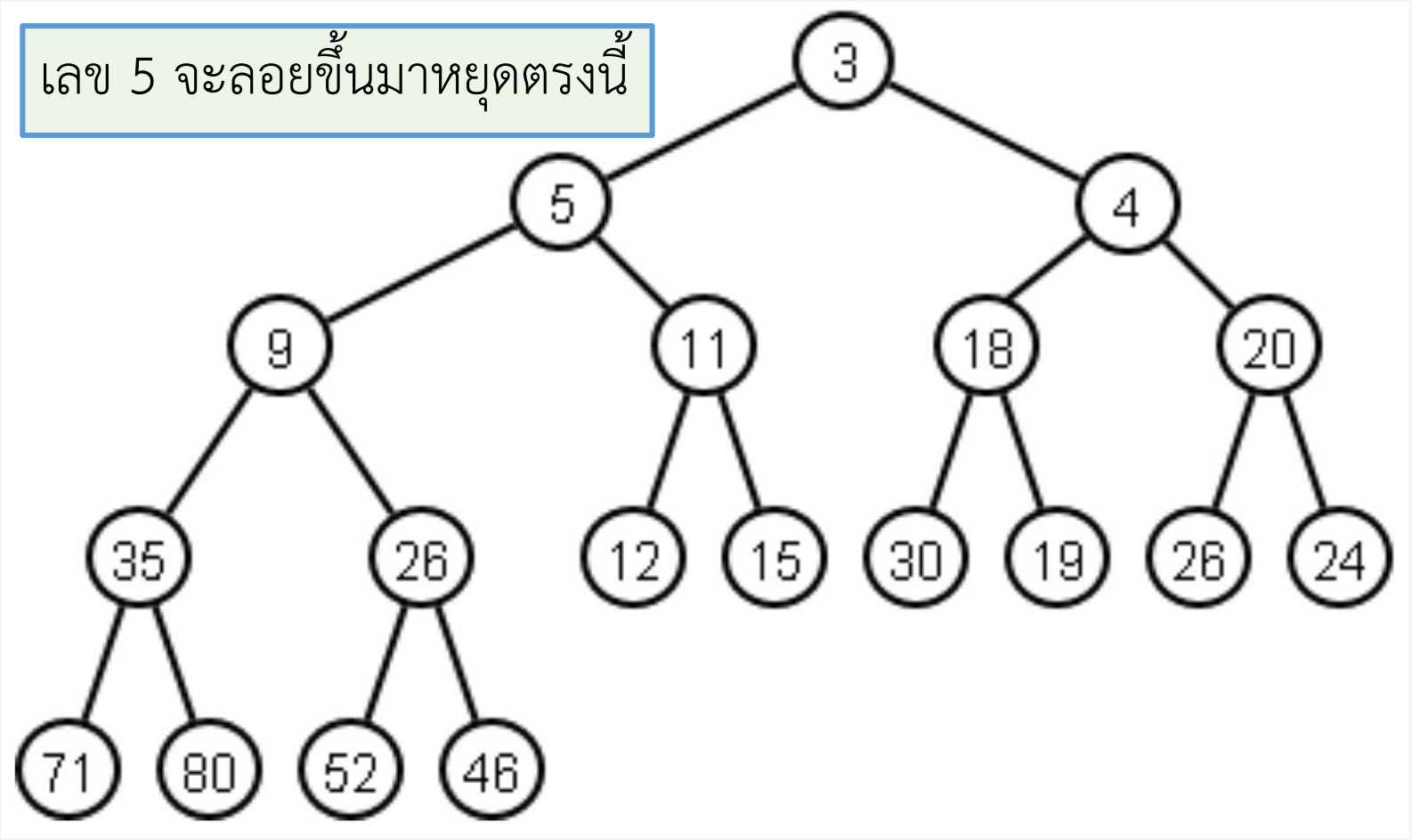


ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

จะเห็นได้ว่า 5 < 46 แสดงว่าผิดตำแหน่ง จึงต้องสลับที่กับโหนดพ่อ

สลับแล้วก็ยังพบว่า 5 < 26 และ 5 < 9 จึงต้องสลับอีกสองครั้ง

ผลลัพธ์เมื่อใส่ข้อมูลใหม่เข้าไป

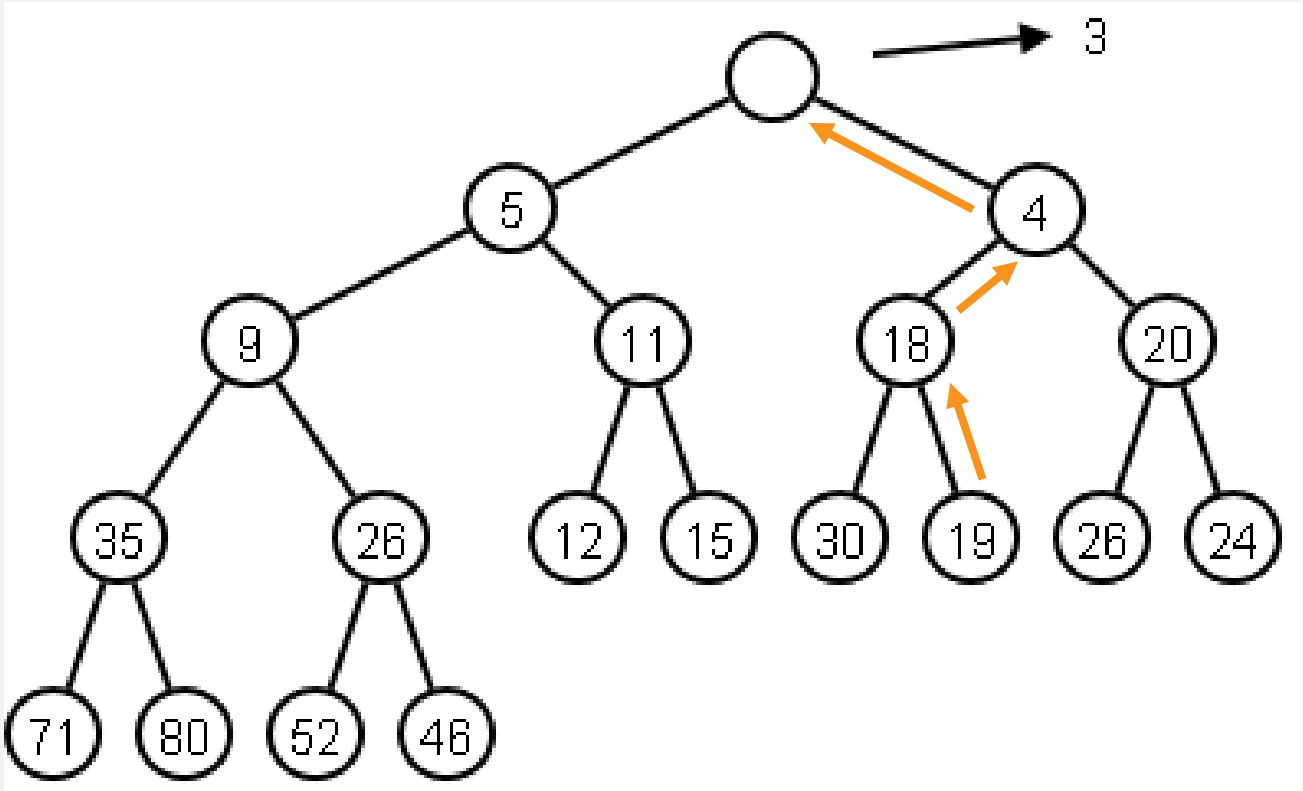


ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

ส่วนโหนดพ่อเดิม ๆ ของโหนด 5 จะไหลลงไป ให้สังเกตว่าความเป็น partial order ของฮีปจะยังมีอยู่ เพราะโหนดพ่อแต่ละตำแหน่งเปลี่ยนค่าไปในทิศทางที่น้อยลง

ลองลบโหนดที่รากออกไป (Extract Min)

- เป็นแบบนี้เราต้องหาโหนดมาแทนโหนดพ่อ
  - เราอาจจะคิดว่าถ้าเลือกโหนดลูกที่น้อยที่สุดจากซ้ายและขวามาแทนที่
  - จากนั้นก็เลือกโหนดที่น้อยกว่ามาแทนที่ขึ้นไปเรื่อย ๆ

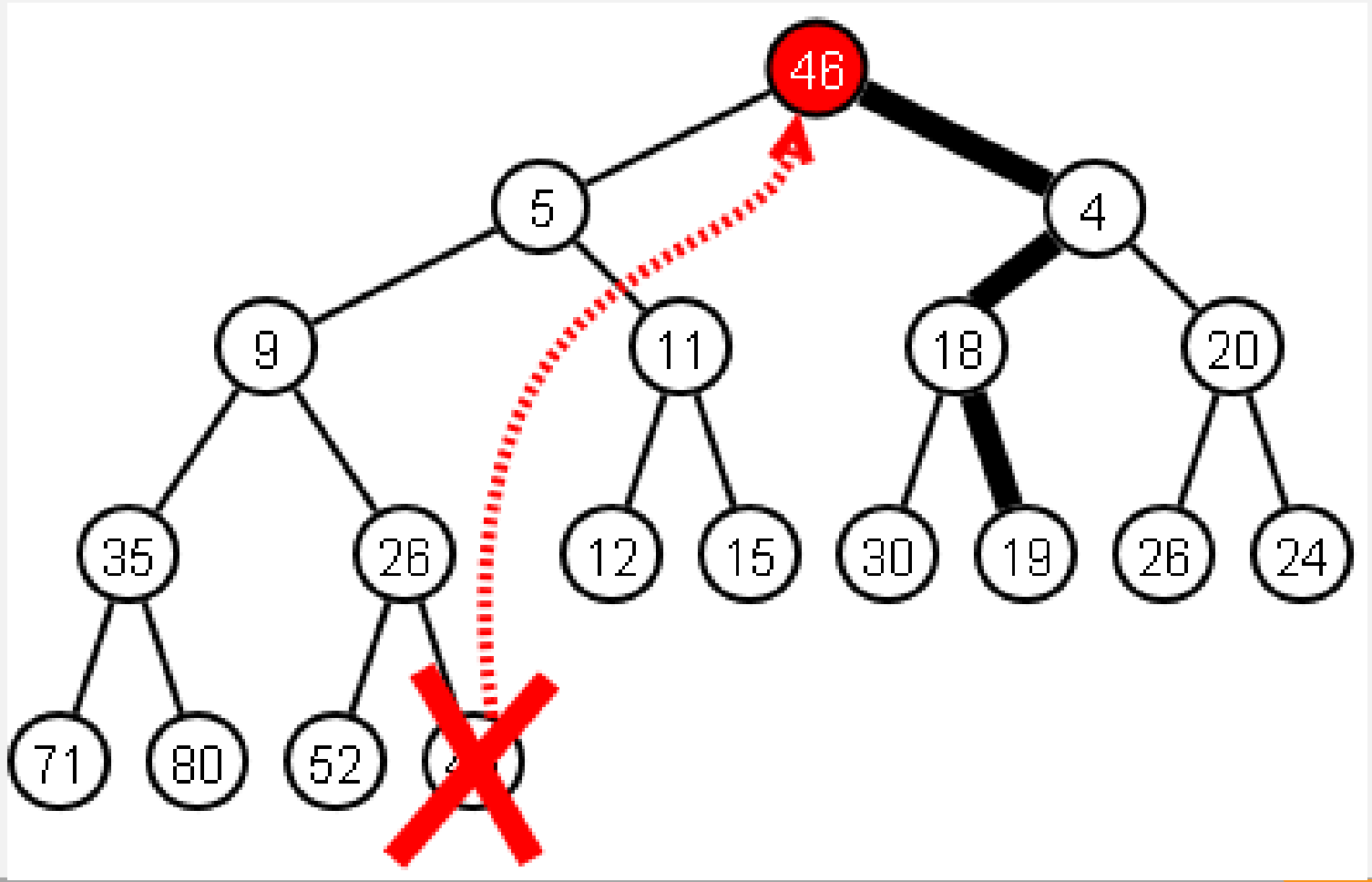


เรารักษาแนวคิดเรื่อง partial order ได้จริง แต่ฮีปจะแหว่งข้างในทำให้เรารักษาความสูงให้น้อย ๆ ไว้ไม่ได้ เมื่อรักษาความสูงไม่ได้ ความเร็วก็จะลดลง

ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

แล้วต้องทำอย่างไรถึงจะรักษาความสูงฮีปได้

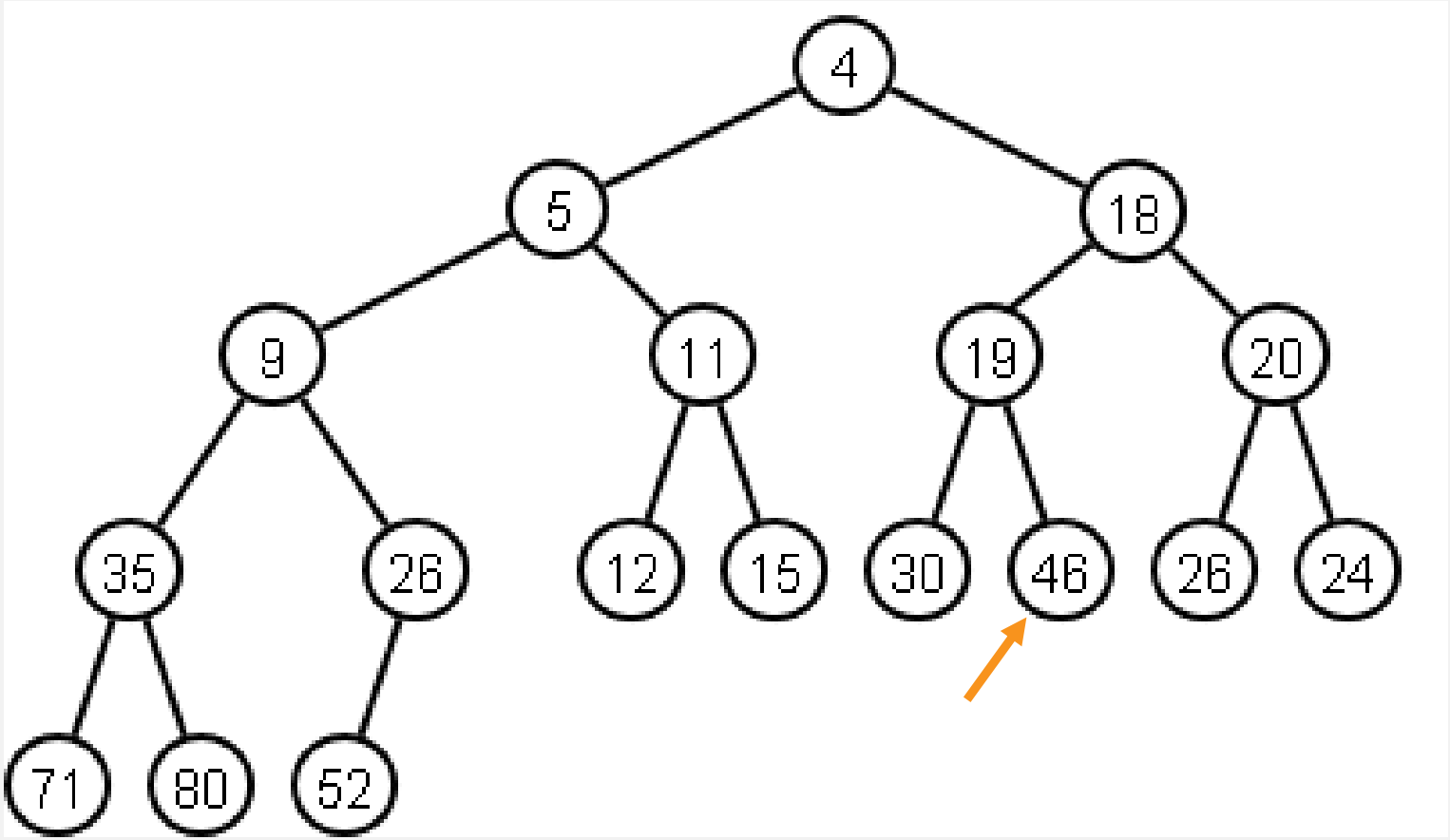
- ควรจะย้ายโหนดปลายขึ้นไปทางด้านบน
  - ย้ายไปตอนแรกมันจะอยู่ผิดตำแหน่ง แต่เราค่อย ๆ ไล่สลับมันลงมาก็ได้



ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

ผลลัพธ์จากการลบโหนดที่รากออกไป

เนื่องจากต้นไม้ฮีปยังเต็มเปี่ยมไม่มีแหว่ง แบบนี้ความสูงยังไงก็ต้องน้อยที่สุด



ดังนั้นแนวคิดการโยนโหนดปลายสุดขึ้นไปแล้วหาบ้านใหม่ให้จึงถือว่าใช้ได้

ภาพจาก http://cs.lmu.edu/~ray/notes/pqueues/

แล้วการ Decrease Key ล่ะ

- การดำเนินการลดค่าตัวเลขนั้นง่ายมาก
  - เราเพียงเทียบค่ากับโหนดพ่อ ถ้าลำดับผิดก็สลับกัน
  - ยังไงกฎเรื่อง **partial order** ก็ยังได้รับการรักษาไว้ไม่ขาดหาย
  - เนื่องจากเป็นการสลับค่า ยังไงต้นไม้ฮีปก็จะยังเต็มเปี่ยม ไม่มีขาดแหว่ง

แล้วคุณคิดว่า เราจะ Increase Key ด้วยหลักการเดียวกันได้ไหม

ฮีปเกิดขึ้นมาได้อย่างไร

- ฮีปเกิดขึ้นมาจากอาเรย์ธรรมดา ๆ
  - ใน C++ จะใช้ vector อยู่ข้างหลัง
  - ไม่จำเป็นต้องอาศัย **pointer** (ตัวชี้) อย่างที่ต้นไม้ทั่วไปต้องทำ
  - เพราะมันเป็นต้นไม้แบบเต็ม ทำให้เรารู้ตำแหน่งของลูกและพ่อจากการคำนวณทางตัวเลข การใช้ตัวชี้จึงเป็นสิ่งที่ไม่จำเป็น
  - ดังนั้นหน่วยความจำที่ต้องใช้ในฮีปจึงค่อนข้างน้อย เราตัดตัวชี้ได้ออกไปโหนดละสองตัว
- ลองพิจารณาตัวเลขในอาเรย์ทั่วไปที่ยังไม่เป็นฮีป แล้วทำให้เป็นฮีป

7 8 3 5 2 1 4 6

แล้วถ้าจะ sort ข้อมูลล่ะ

- ง่ายมาก สร้างฮีปแล้วสกัดเอาค่าน้อยที่สุดออกมาเรื่อย ๆ ได้เลย
- แนวคิดมีแค่นี้จริง ๆ นะ ดังนั้นมาเขียนโปรแกรมกันเลยดีกว่า
  - แต่ตอนจะเขียนโปรแกรม เราจะต้องเตรียมคลาส/สตรัคให้เรียบร้อย
  - มันยากตรงนี้แหละ

```
#include <iostream>
#include <vector>
#include <string>

/// This header declares prirority_queue
#include <queue>

using namespace std;
```

สร้างคลาสนินจาพร้อมตัวเปรียบเทียบ

SCIENCE
SILPAKORN UNIVERSITY

```
class Ninja {
public:
    string name;
    int score;

    Ninja(string name, int score) {
        this->name = name;
        this->score = score;
    };
};

struct NinjaComparator {
    bool operator()(const Ninja& n0, const Ninja& n1) {
        return n0.score > n1.score;
    };
};
```

หน้าตาของ Priority Queue ใน C++

Data type ของมันดูน่าเกลียดมาก

```
priority_queue<Ninja, vector<Ninja>, NinjaComparator> pq;
```

แต่ตอนใส่ข้อมูลนี่ง่ายหน่อย push วัตถุเข้าไปใหม่เรื่อย ๆ ก็พอแล้ว

```
void prepareData(auto& pq) {
    ///Ninja A[] = {{"Naruto", 38}, {"Itachi", 100},
    ///              {"Sasuke", 92}, {"Sakura", 84},
    ///              {"Kakashi", 93}};
    pq.push(Ninja("Naruto", 38));
    pq.push(Ninja("Itachi", 100));
    pq.push(Ninja("Sasuke", 92));
    pq.push(Ninja("Kakashi", 93));
    pq.push(Ninja("Sakura", 84));
}
```

หลังจากลำบากมานาน ตอน sort นั้นง่ายนัก


```
priority_queue<Ninja, vector<Ninja>, NinjaComparator> pq;
prepareData(pq);

while(pq.size() > 0) {
    Ninja nin = pq.top();
    cout << nin.name << " ";
    pq.pop();
}
```

หัวข้อเนื้อหา

SCIENCE
SILPAKORN UNIVERSITY

- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - เรียงสตรัคหรือวัตถุจากคลาส
- วิธีที่มันเร็วขึ้นกว่าเดิม
  - Priority Queue and Heap sort
  - **Merge sort**
  - Quick sort
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

ฮีปเกิดขึ้นมาได้อย่างไร

- ฮีปเกิดขึ้นมาจากอาเรย์ธรรมดา ๆ
  - ใน C++ จะใช้ vector อยู่ข้างหลัง
  - ไม่จำเป็นต้องอาศัย **pointer** (ตัวชี้) อย่างที่ต้นไม้ทั่วไปต้องทำ
  - เพราะมันเป็นต้นไม้แบบเต็ม ทำให้เรารู้ตำแหน่งของลูกและพ่อจากการคำนวณทางตัวเลข การใช้ตัวชี้จึงเป็นสิ่งที่ไม่จำเป็น
  - ดังนั้นหน่วยความจำที่ต้องใช้ในฮีปจึงค่อนข้างน้อย เราตัดตัวชี้ได้ออกไปโหนดละสองตัว
- ลองพิจารณาตัวเลขในอาเรย์ทั่วไปที่ยังไม่เป็นฮีป แล้วทำให้เป็นฮีป

  7 8 3 5 2 1 4 6

- วิธีการก็คือให้คิดเหมือนกับว่าตอนแรกฮีปว่างเปล่าแล้วเราค่อย ๆ เติมค่าเข้าไปทีละตัวจนครบ

หัวข้อเนื้อหา

SCIENCE
SILPAKORN UNIVERSITY

- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - เรียงสตรัคหรือวัตถุจากคลาส
- วิธีที่มันเร็วขึ้นกว่าเดิม
  - Priority Queue and Heap sort
  - **Merge sort**
  - Quick sort
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

Merge Sort

- เป็นวิธีการจัดเรียงข้อมูลที่นิยมใช้กับข้อมูลขนาดใหญ่ที่อยู่ในดิสก์หรือต้องกระจายไปหลาย ๆ เครื่อง
- เป็นอัลกอริทึมที่ใช้วิธี **divide and conquer**
  - เพราะการแบ่งงานนี่แหละที่ทำให้มันเหมาะกับการกระจายไปหลาย ๆ เครื่อง
  - นอกจากนี้มันยังแบ่งงานแบบเท่า ๆ กันระหว่างเครื่องด้วย ทำให้มีความสมดุลในการกระจายงาน
- ในเวอร์ชันพื้นฐาน การทำงานเริ่มจากการจับคู่ตัวเลขเป็นกลุ่มย่อย แล้วสลับลำดับให้ถูก
  - 6 5 3 1 8 7 2 4 → 6 5 3 1 8 7 2 4 → 5 6 1 3 7 8 2 4 (ลำดับในกลุ่มย่อยจะถูกต้อง)
- ต่อจากนั้นจะค่อย ๆ รวมผลลัพธ์เข้าด้วยกัน ด้วยการจับคู่กลุ่มย่อย

การรวมผลลัพธ์จากกลุ่มย่อย

- Merge sort รวมผลลัพธ์จากกลุ่มย่อยที่ลำดับภายในกลุ่มย่อยถูกต้องอยู่แล้ว
  - การเปรียบเทียบค่าสามารถเริ่มจากการเทียบตัวที่น้อยที่สุดของแต่ละกลุ่ม
  - ตัวไหนน้อยกว่าก็จะถูกคัดลอกไปไว้ในพื้นที่ผลลัพธ์ก่อน
  - จากนั้นตัวถัดไปจากกลุ่มย่อยที่ถูกคัดลอกไปจะถูกนำมาพิจารณาต่อ
  - แต่ในกลุ่มย่อยของตัวที่มากกว่าจะยังไม่เกิดการเปลี่ยนแปลง เพราะพฤติกรรมมันเหมือนแพ้คัดออก คนชนะอยู่สู้ต่อไปเรื่อย ๆ จนกว่าจะแพ้
- 5 6 1 3 → เริ่มเทียบ 5 กับ 1 ก่อน ซึ่ง 1 จะแพ้และถูกคัดลอกไปเก็บไว้ก่อน จากนั้น 5 จะเจอกับ 3 และ 3 ก็แพ้อีก และถูกคัดลอกไป
  - ต่อมาพบว่า 5 หมดคู่ต่อสู้ เราจะคัดลอกเลขที่เหลือในกลุ่มย่อยนั้นไปเก็บไว้
  - ทำให้ได้ผลลัพธ์เป็น 1 3 5 6

รวมผลลัพธ์จากคู่กลุ่มย่อยอีกอัน

- 7 8 2 4 → เริ่มเทียบ 7 กับ 2 ก่อน ซึ่ง 2 จะแพ้และถูกคัดลอกไปเก็บไว้ก่อน จากนั้น 7 จะเจอกับ 4 และ 4 ก็แพ้อีก และถูกคัดลอกไป
  - ต่อมาพบว่า 7 หมดคู่ต่อสู้ เราจะคัดลอกเลขที่เหลือในกลุ่มย่อยนั้นไปเก็บไว้
  - ทำให้ได้ผลลัพธ์เป็น 2 4 7 8
- ณ เวลานี้กลุ่มย่อยของเลขที่ต้องนำมารวมกันเป็นดังนี้
  1 3 5 6 2 4 7 8 (ในแต่ละกลุ่มย่อยเรียงจากน้อยไปมาก)
  - 1 เจอกับ 2 ซึ่ง 1 แพ้และถูกนำไปเก็บ ต่อมา 3 เจอกับ 2 และ 2 โดนเก็บ
  - 3 เจอกับ 4 เลข 3 โดนเก็บ ได้ 5 ขึ้นมาสู้ต่อ และคราวนี้ 4 โดนเก็บ
- ณ เวลานี้ เลขที่โดนเก็บไปเป็นผลลัพธ์ เรียงตามลำดับได้เป็น 1 2 3 4 ส่วนเลขที่ยังเหลืออยู่และต้องนำมาเปรียบเทียบกันต่อคือ 5 6 7 8

รวมผลลัพธ์ให้เสร็จ

- ณ เวลานี้ เลขที่โดนเก็บไปเป็นผลลัพธ์ เรียงตามลำดับได้เป็น 1 2 3 4 ส่วนเลขที่ยังเหลืออยู่และต้องนำมาเปรียบเทียบกันต่อคือ 5 6 7 8
- ในทำนองเดิม เรานำ 5 มาเทียบกับ 7 และ 5 ถูกนำไปเก็บและได้ 6 มาสู้ต่อ เมื่อ 6 ถูกเก็บ กลุ่มย่อยทางซ้ายจะหมดสมาชิก
  - เราก็จะนำเลขที่เหลือทั้งหมดในกลุ่มย่อยทางขวาไปเก็บไว้เป็นผลลัพธ์
- พิจารณาตามลำดับการนำไปเก็บเป็นผลลัพธ์ทั้งจากหน้าที่แล้วและหน้านี้ เราจะได้ผลลัพธ์เรียงตามลำดับที่ถูกต้อง

1 2 3 4 5 6 7 8

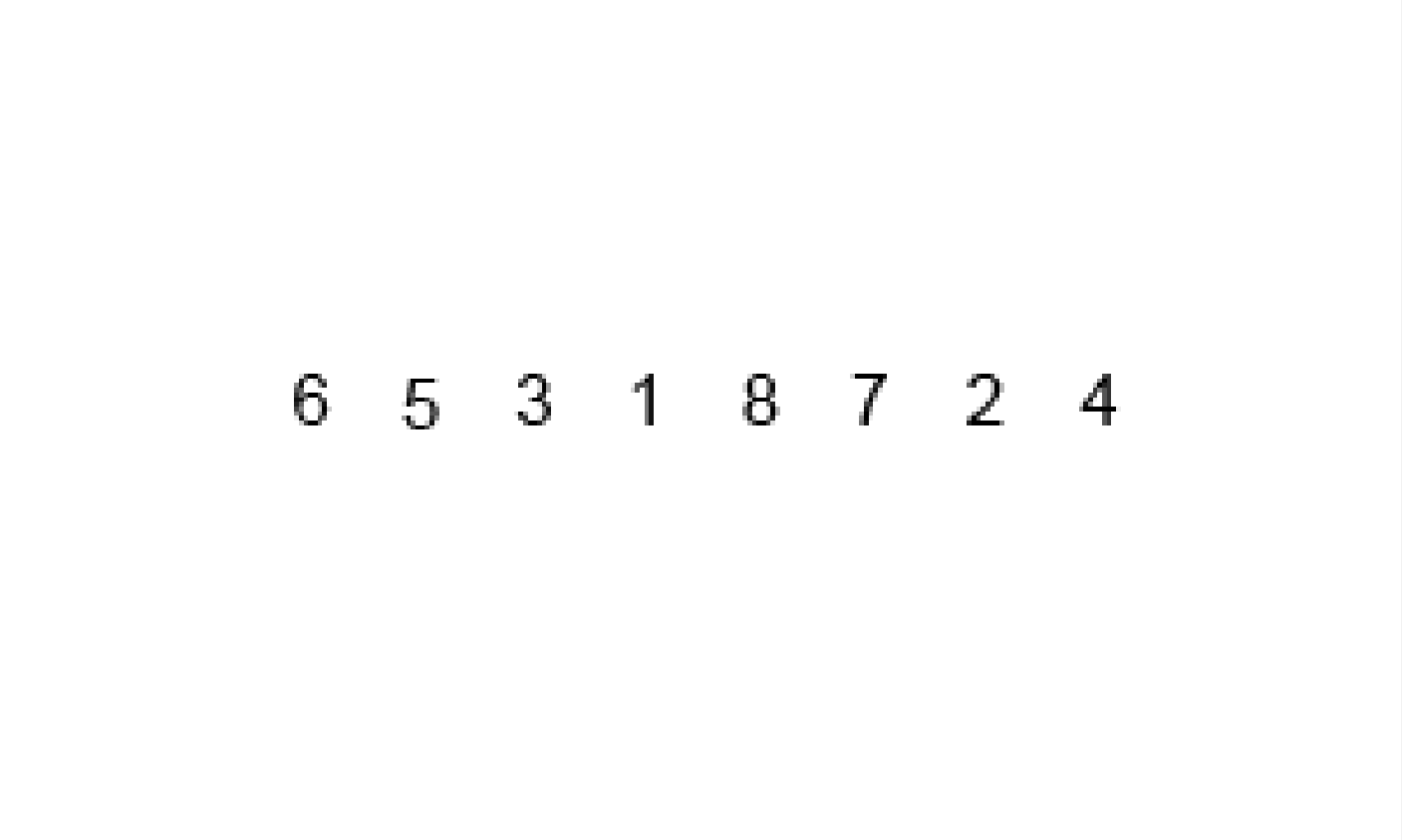
วิเคราะห์การทำงาน

- ตอนแรกเราจะจับคู่เปรียบเทียบ ขนาดของกลุ่มย่อยที่เรียงแล้วคือ 2
  - เมื่อนำกลุ่มย่อยขนาด 2 มารวมกัน ขนาดของกลุ่มย่อยที่เรียงแล้วคือ 4
  - สุดท้าย นำกลุ่มย่อยขนาด 4 มารวมกัน ขนาดของกลุ่มย่อยที่เรียงแล้วคือ 8
- จะเห็นได้ว่าขนาดของกลุ่มย่อยจะโตขึ้นทีละเท่าตัว
  - ในตัวอย่างมีเลข 8 ตัวจึงต้องใช้การสร้างกลุ่มย่อย 3 รอบ (2^3) และหากพิจารณาว่า N คือจำนวนตัวเลขทั้งหมด จำนวนรอบก็คือ $\log_2 N$
  - แต่ละรอบเราเทียบเลขมากสุด N ครั้ง และคัดลอกค่า N ครั้ง เราจึงได้ Big Oh เป็น $O(N \log_2 N)$

ทำไมจึงใช้คำว่าคัดลอก แทนที่จะเป็นสลับค่า

- เพราะโดยธรรมดา เราจะต้องมีที่เก็บข้อมูล 2 ชุดใน merge sort
- การสลับค่าในกลุ่มย่อยสองกลุ่มไม่ได้ทำให้เราได้ลำดับเลขที่ถูกต้อง
  - เช่นตอนที่เราจะรวมกลุ่มย่อยสองกลุ่มนี้ 5 6 1 3 ถ้าเราสลับลำดับ 5 กับ 1 ตอนแรกผลลัพธ์เหมือนจะถูกเพราะได้เป็น 1 6 5 3
  - แต่จังหวะต่อมา เราต้องเทียบ 3 กับ 5 ซึ่งสลับลำดับแล้วจะได้เป็น 1 6 3 5 และเลข 6 จะไม่มีที่อยู่ที่ถูกต้อง เว้นเสียเราจะย้ายข้อมูลครั้งใหญ่
- วิธีที่ถูกต้องก็คือสร้างที่เก็บผลลัพธ์มาไว้อีกชุด เทียบแต่ละครั้งเสร็จแล้วก็ย้ายไปไว้ในที่เก็บผลลัพธ์ที่สร้างมาโดยเฉพาะ
  - จากนั้นที่เก็บผลลัพธ์ก็จะมีบทเป็นกลุ่มย่อยที่จะเปรียบเทียบ ส่วนที่เก็บข้อมูลเข้าในตอนแรกจะทำหน้าที่เป็นที่เก็บผลลัพธ์แทน
  - บทบาทหน้าที่จะสลับอย่างนี้ไปเรื่อย ๆ จนจบการทำงาน

ภาพแสดงการทำงาน



หมายเหตุ ตอนที่เลขถูกย้ายไปอีกแถวในภาพ
แท้จริงก็คือการคัดลอกไปไว้ในที่เก็บข้อมูลอีกชุดหนึ่ง

ภายในของ merge sort

- มักจะมี insertion sort ซ่อนอยู่ข้างใน เพราะการเรียงข้อมูลกลุ่มเล็ก ๆ เช่น น้อยกว่า 7 ตัว การจะ merge ข้อมูลไปมาจะใช้เวลาค่อนข้างมาก
  - การทำงานในตอนแรกจึงมักเริ่มด้วยการสร้างกลุ่มย่อยขนาดประมาณ 6 ตัวที่จัดเรียงแล้วด้วย insertion sort ก่อน
- การรวมกลุ่มย่อยแต่ละคู่สามารถทำได้เป็นอิสระ
  - เช่นการทำงานของคู่แรกและคู่ที่สองเกิดขึ้นบนข้อมูลคนละช่องในอาเรย์ ไม่คาบเกี่ยวกัน
  - ต่างกับ Heap sort ซึ่งเราจะ pop ข้อมูลที่น้อยที่สุดอันดับถัดไปได้ก็ต่อเมื่อเรา pop ตัวแรกให้เสร็จก่อน
  - คุณสมบัตินี้ทำให้เราสามารถทำ merge sort แบบขนานด้วย CPU หลายคอร์หรือเครื่องหลายเครื่องได้

หัวข้อเนื้อหา

SCIENCE
SILPAKORN UNIVERSITY

- พื้นฐานการจัดลำดับข้อมูล
  - เรียงตัวเลขกันดื้อ ๆ ด้วยวิธีที่เข้าใจง่ายแต่ช้า
  - Selection sort, Insertion sort
  - เรียงสตรัคหรือวัตถุจากคลาส
- วิธีที่มันเร็วขึ้นกว่าเดิม
  - Priority Queue and Heap sort
  - Merge sort
  - **Quick sort**
- ตัวอย่างการประยุกต์ใช้งานการจัดลำดับข้อมูล

Quick sort

- เป็นการ sort ในหน่วยความจำที่มีเวลาในการรันเฉลี่ยเป็น $O(N \log N)$
- แต่เวลาที่เลวร้ายที่สุดจะเป็น $O(N^2)$
  - ทำให้ Quick sort ดูเลวร้ายและไม่น่าใช้
  - แต่วิธีป้องกันเหตุการณ์นี้ก็มีอยู่ ซึ่งทำให้ความน่าจะเป็นที่มันจะช้านั้นต่ำมาก
  - หรือถ้าหากสถานการณ์เริ่มดูเลวร้าย บางส่วนของข้อมูลสามารถจัดการด้วยฮีปซอร์ตแทนได้ ทำให้เวลาการทำงานดูดีได้เสมอไม่มีพลาด
- ความสวยงามของ Quick sort ก็คือสัมประสิทธิ์ที่ซ่อนอยู่ใน $O(N \log N)$ นั้นมีค่าน้อยมาก
  - ในขณะที่ merge sort มีสัมประสิทธิ์ที่ซ่อนอยู่ค่อนข้างมาก
- ควิกซอร์ตเป็น divide and conquer เช่นกัน แบ่งงานข้ามคอร์ของ CPU ได้
  - แต่การแบ่งไม่ได้รับประกันว่างานจะมีขนาดเท่า ๆ กันเหมือน merge sort

แล้วมันทำงานยังไง

https://visualgo.net/bn/sorting

เรียกใช้งาน Quick sort ผ่าน std::sort

- ถึงแม้ว่า std::sort จะไม่ได้ระบุว่าจะต้องใช้ quick sort ไว้ด้านใน
  - แต่โดยปรกติ C++ library จะนิยมใช้เวอร์ชันอันใดอันหนึ่งของ quick sort
  - ภายในจะมีวิธีป้องกันไม่ให้ quick sort ช้า คือไม่มีทางโชคร้ายแน่ ๆ
  - เช่นการบรรจุ heap sort ไว้ด้านในเวลาที่สถานการณ์ไม่ดี หรือมีการ random pivot ที่ดี
- ควรใช้คอนเทนเนอร์แบบเวคเตอร์ในการเก็บข้อมูล
- จะเรียงจากน้อยไปมาก ถ้าตัวเปรียบเทียบคืน true เมื่อพารามิเตอร์ตัวแรกน้อยกว่าตัวที่สอง (ตรงกันข้ามกับพฤติกรรมของการใช้ฮีปในการเรียงข้อมูล)

หน้าตาตัวเปรียบเทียบ

- สมมติว่าเราจะเรียงนินจา ตัวเปรียบเทียบเราจะมีหน้าตาคล้ายเดิม แบบนี้

```
struct NinjaComparator2 {
    bool operator()(const Ninja& n0, const Ninja& n1) {
        return n0.score < n1.score;
    };
};
```

- ถ้าจะให้เรียงจากน้อยไปมาก ก็คืน true เมื่อพารามิเตอร์ตัวแรกน้อยกว่า ซึ่งก็เหมือนการทดสอบว่าพารามิเตอร์เรียงจากน้อยไปมากหรือเปล่า

เตรียมเวคเตอร์ข้อมูลให้เรียบร้อย

- พอใช้เวคเตอร์ เราจะใช้ฟังก์ชัน push_back ในการใส่ข้อมูล

```
void prepareData(auto& vec) {
    vec.push_back(Ninja("Naruto", 38));
    vec.push_back(Ninja("Itachi", 100));
    vec.push_back(Ninja("Sasuke", 92));
    vec.push_back(Ninja("Kakashi", 93));
    vec.push_back(Ninja("Sakura", 84));
}
```

- พิมพ์ออกมาดูว่าข้างในเวคเตอร์เรียงข้อมูลอย่างไร

```
void printData(auto& vec) {
    for(Ninja& nin: vec)
        cout << nin.name << " ";
    cout << endl;
}
```

วิธีวนไปในคอนเทนเนอร์จาก begin ไปจนถึง end

เรียงข้อมูลจริง ๆ เสียที

- สบายยิ่งกว่าฮีปซอร์ต สั่งเรียงแล้วเวคเตอร์แปรสภาพเป็นที่เก็บผลลัพธ์ทันที

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
  vector<Ninja> vec;
  prepareData(vec);
  printData(vec);
  std::sort(vec.begin(), vec.end(), NinjaComparator2());
  printData(vec);

  return 0;
}
```