โครงสร้างข้อมูล

Data Structures using Java ฉบับภาษาจาวา

สมชาย ประสิทธิ์จูตระกูล
ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย

เนื้อหาบทเรียนวิชาโครงสร้างนี้ได้รับการสนับสนุนการผลิตจาก
สำนักนโยบายและแผนการอุดมศึกษา
สำนักงานคณะกรรมการการอุดมศึกษา
กระทรวงศึกษาธิการ

คำนำ

โครงสร้างข้อมูลเป็นหนึ่งในองค์ความรู้ขั้นพื้นฐานของการศึกษาทางวิทยาการคอมพิวเตอร์ วิศวกรรมคอมพิวเตอร์ และอื่น ๆ อีกหลากหลายสาขา ที่ว่าด้วยการจัดเก็บข้อมูลอย่างมีระเบียบ และการจัดการข้อมูลอย่างมีระบบ เพื่อให้ตรงตามความต้องการในการประมวลผลข้อมูลอย่างมีประสิทธิภาพ การศึกษาโครงสร้างข้อมูลจึงต้องอาศัยความรู้และความชำนาญในการเขียนโปรแกรม เพื่อพัฒนาโครงสร้างการจัดเก็บข้อมูลที่ออกแบบไว้ให้เห็นจริง และอาศัยความสามารถทางตรรก การนับ การวิเคราะห์ และการจำลอง เพื่อเป็นเครื่องมือในการออกแบบและวิเคราะห์วิธีการจัดการข้อมูลว่ามีประสิทธิภาพเพียงใด

วัตถุประสงค์ของการศึกษาเรื่องโครงสร้างข้อมูลคือเพื่อให้ “เลือกเป็น, ใช้เป็น, และสร้างเป็น” เนื่องจากไม่มีโครงสร้างข้อมูลใดที่สนองความต้องการได้ทุกงานอย่างรวดเร็ว ดังนั้นเราต้องรู้ว่าเมื่อไรควรเลือกใช้โครงสร้างข้อมูลแบบใด กับงานประเภทใด ด้วยประสิทธิภาพที่ต่างกันอย่างไร ทั้งทางทฤษฎีและทางปฏิบัติ ดังนั้นจึงต้องรู้ความแตกต่างของโครงสร้างข้อมูลที่มีหลากหลายแบบที่สนองความต้องการเดียวกัน เมื่อเลือกได้แล้วก็ต้องเรียกใช้บริการต่าง ๆ ที่ตัวข้อมูลมีให้ได้อย่างถูกต้อง รู้เงื่อนไขที่จำเป็นต้องมีก่อนใช้บริการ รู้ว่าอะไรคือผลที่ได้หลังการใช้บริการ และต้องจำชื่อหรือมีทักษะในการค้นหาชื่อของบริการต่าง ๆ ให้ได้ และสุดท้ายคือสามารถแปลงแนวคิดการจัดเก็บและจัดการข้อมูล ออกมาเป็นโปรแกรมที่ใช้งานจริงอย่างมีประสิทธิภาพ

ขอให้สนุกกับการเรียนวิชาโครงสร้างข้อมูลครับ

รศ. ดร. สมชาย ประสิทธิ์จูตระกูล
ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิยาลัย

เอกสารชุดนี้ใช้แบบอักษรที่มีให้ในระบบปฏิบัติการวินโดวส์ เช่น

แบบอักษร Arial Unicode MS
Courier New
แบบอักษร Tahoma
Times New Roman

นอกจากนี้ยังใช้แบบอักษรดังต่อไปนี้

แบบอักษร Layiji มหานิยม
แบบอักษร วรเศรษฐ

ผู้จัดทำขอขอบคุณนักออกแบบแบบอักษรมา ณ โอกาสนี้ด้วย

สารบัญ

บทนำ

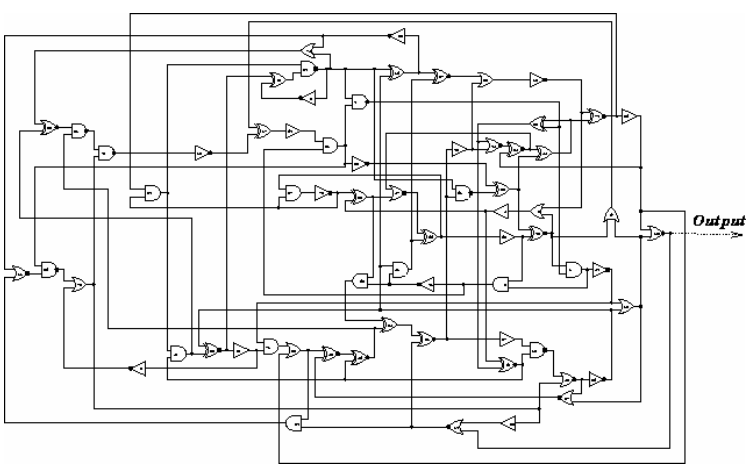
หัวข้อ

- นิยาม
- ตัวอย่างการใช้งาน
- วัตถุประสงค์ของการศึกษาโครงสร้างข้อมูล
- ตัวอย่างประสิทธิภาพเชิงเวลาการทำงาน

โครงสร้างข้อมูลคืออะไร

- วิธีการจัดเก็บและจัดการข้อมูลให้
  - ตรงตามความต้องการ
  - ทำงานรวดเร็ว
  - ประหยัดเนื้อที่
  - เข้าใจง่าย

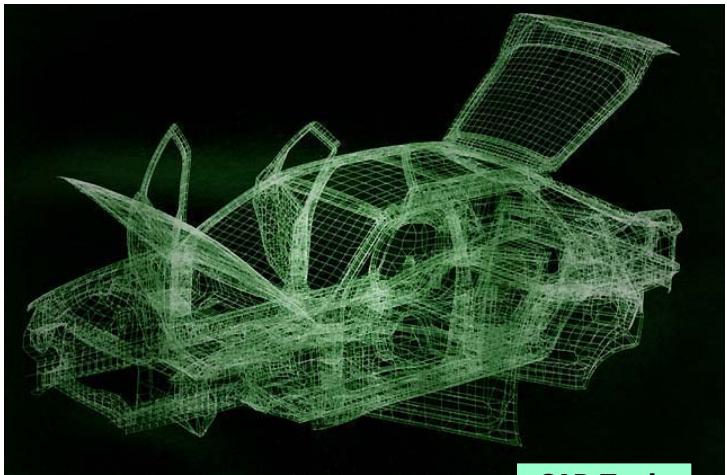
จะเขียนโปรแกรมจัดเก็บและจัดการอย่างไร ?



Logic Design Tools


intel inside
pentium 4
ทรานซิสเตอร์ 55 ล้านตัว
กับสายสัญญาณเชื่อมต่อ
อีกจำนวนมาก
VLSI Design


Games



CAD Tools

และอื่น ๆ

- Google เก็บเอกสารอย่างไร ทำให้ค้นได้รวดเร็ว
- jvm เก็บออปเจกต์ต่าง ๆ คลาสต่าง ๆ ตัวแปรต่าง ๆ thread ต่าง ๆ ไว้อย่างไร
- Word processor เก็บตัวอักษร คำ ข้อความ ย่อหน้า สูตร รูป และอื่น ๆ ในเอกสารภายในหน่วยความจำอย่างไร ขณะที่เรากำลังใช้งาน
- ...

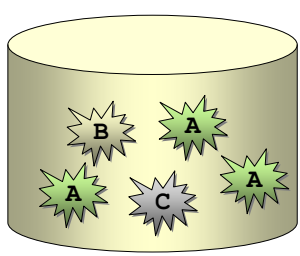
ล้วนเป็นปัญหาที่เกี่ยวกับโครงสร้างข้อมูล

ข้อมูลต่าง ๆ

- บางชนิด ตัวภาษามีให้แล้ว (primitive)
  - int, double, char, boolean, ...
- บางชนิดคลังคำสั่งของระบบมีให้ใช้ (class)
  - String, Color, BigDecimal, ArrayList, HashMap, ...
- และมีอีกมากมายที่ต้องออกแบบและสร้างเอง

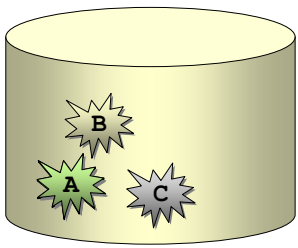
ที่เก็บข้อมูลแบบพื้นฐาน : Collection

- เก็บข้อมูลไม่มีอันดับ ซ้ำได้



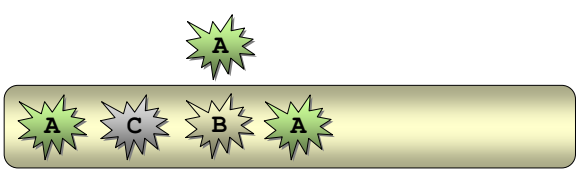
ที่เก็บข้อมูลแบบพื้นฐาน : Set

- เก็บข้อมูลไม่มีอันดับ ไม่ให้ซ้ำ



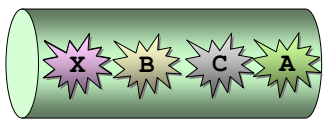
ที่เก็บข้อมูลแบบพื้นฐาน : List

- เก็บข้อมูลเรียงแบบมีอันดับ ซ้ำได้



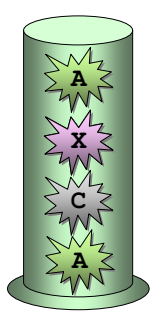
ที่เก็บข้อมูลแบบพื้นฐาน : Queue

- ข้อมูล เข้าก่อน ออกก่อน (First-In First-Out)



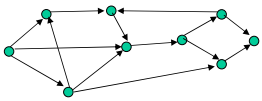
ที่เก็บข้อมูลแบบพื้นฐาน : Stack

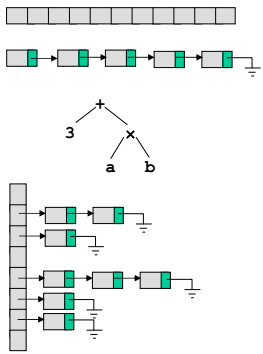
- ข้อมูล เข้าหลัง ออกก่อน (Last-In First-Out)



วิธีสร้างที่เก็บข้อมูล

- สร้างด้วยอาเรย์
- สร้างด้วยการโยง
- สร้างด้วยต้นไม้
- สร้างด้วยตาราง
- อื่น ๆ





วัตถุประสงค์

เลือกให้เป็น
ใช้ให้เป็น
สร้างให้เป็น

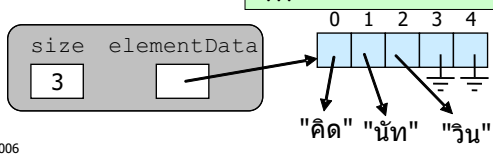
เลือกให้เป็น

ใช้ให้เป็น

```
public static PuzzleBoard solve(PuzzleBoard b) {
  Set set = new ArraySet();
  Queue queue = new ArrayQueue();
  queue.enqueue(b); set.add(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if ( b2.isAnswer() ) return b2;
        if (! set.contains(b2) ) {
          queue.enqueue(b2); set.add(b2);
        }
      }
    }
  }
  return null;
}
```

สร้างให้เป็น

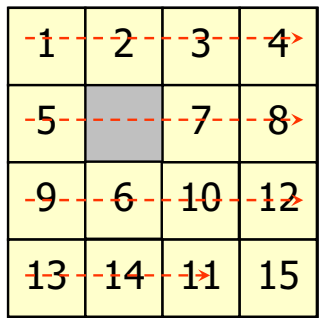

```
public class ArrayCollection
       implements Collection {
 private Object[] elementData;
 private int      size;

 public ArrayCollection(int c) {
   elementData = new Object[c];
   size = 0;
 }
 public void add(Object e) {
   elementData[size++] = e;
 }
 public boolean size() {
   return size;
 }
 public int isEmpty() {
   return size == 0;
 }
 ...
```



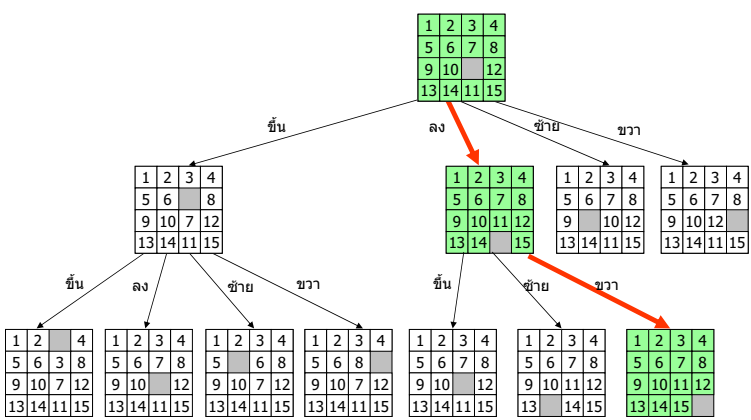
โครงสร้างข้อมูลที่ดี

- ให้บริการตามที่ต้องการ
- ใช้ง่าย
- กินเนื้อที่น้อย
- ทำงานได้อย่างรวดเร็ว

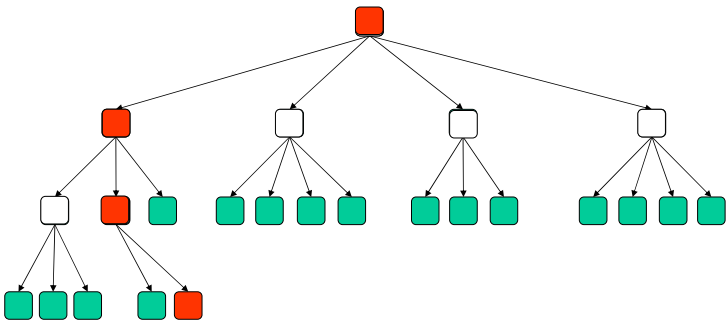
ตัวอย่าง : 15-puzzle



พบคำตอบ พบวิธีเลื่อน



ใช้ Queue เก็บตาราง

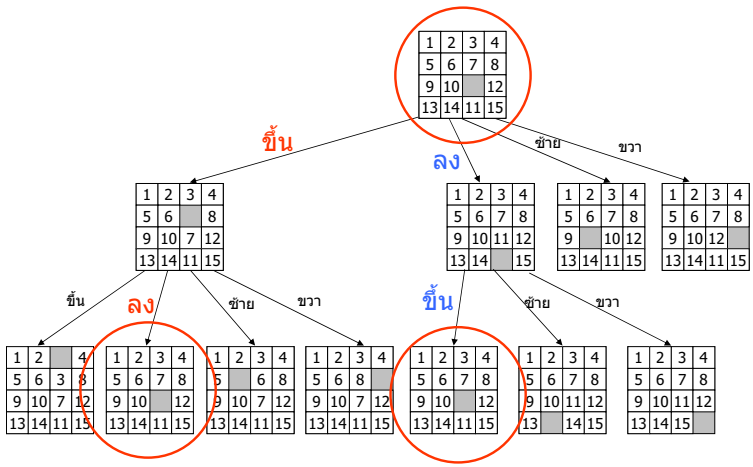


15-puzzle : ส่วนของโปรแกรม

```
public static PuzzleBoard solve(PuzzleBoard b) {

  Queue queue = new ArrayQueue();
  queue.enqueue(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if ( b2.isAnswer() ) return b2;

        queue.enqueue(b2);

      }
    }
  }
  return null;
}
```

ใช้ queue เก็บตารางที่ผลิตใหม่

15-puzzle : ตารางที่ผลิตอาจซ้ำกัน



15-puzzle : ส่วนของโปรแกรม

```
public static PuzzleBoard solve(PuzzleBoard b) {
  Set set = new ArraySet();
  Queue queue = new ArrayQueue();
  queue.enqueue(b);  set.add(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if ( b2.isAnswer() ) return b2;
        if ( ! set.contains(b2) )  {
          queue.enqueue(b2);  set.add(b);
        }
      }
    }
  }
  return null;
}
```

ใช้ queue เก็บตารางที่ผลิตใหม่

ใช้ set เพื่อตรวจสอบความซ้ำซ้อน

15-puzzle : ผลการทดลอง

ตารางเริ่มต้น	จำนวนตารางที่ผลิต	เวลาการทำงาน (วินาที)			
		ArraySet	BSTSet	AVLSet	HashSet
แบบที่ 1	552	0.03	0.02	0.04	0.05
แบบที่ 2	5242	1.94	0.22	0.18	0.12
แบบที่ 3	132049	1819.6	7.08	5.71	2.56

สรุป

- ศึกษาวิธีการจัดเก็บและการจัดการข้อมูล
  - ตรงตามความต้องการ
  - ทำงานรวดเร็ว
  - ประหยัดเนื้อที่
  - เข้าใจง่าย
- เราต้อง
  - เลือกให้เป็น
  - ใช้ให้เป็น
  - สร้างให้เป็น

การเก็บข้อมูลด้วยอาเรย์

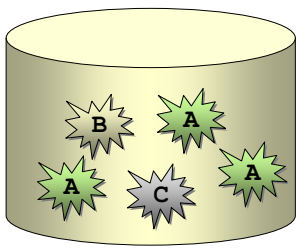
(ArrayCollection)

หัวข้อ

- คอลเล็กชัน
  - นิยาม
  - บริการ
  - การใช้งาน
- การสร้างคอลเล็กชันด้วยอาเรย์
- เรื่องจุกจิกของจาวา

คอลเล็กชัน (Collection)

- ที่เก็บกลุ่มของข้อมูลประเภทเดียวกัน
- ไม่มีอันดับ
- มีข้อมูลซ้ำได้
- บริการ : add, remove, contains, size, isEmpty



Collection Interface

```
public interface Collection {
  public void add(Object element);
  public void remove(Object element);
  public boolean isEmpty();
  public boolean contains(Object element);
  public int size();
}
```

```
public class ArrayCollection implements Collection {
  public ArrayCollection(int cap) { ... }
  public void add(Object element) { ... }
  public void remove(Object element) { ... }
  public boolean isEmpty() { ... }
  public boolean contains(Object element) { ... }
  public int size() { ... }
}
```

ตัวอย่างการใช้คอลเล็กชัน

```
public class TestCollection {
  public static void main(String[] args) {
    Collection c = new ArrayCollection(100);
    System.out.println(c.isEmpty());
    c.add("BANGKOK");
    c.add("PHUKET");
    c.add("BANGKOK");
    c.add("SONGKLA");
    System.out.println(c.size());
    c.remove("PHUKET");
    System.out.println(c.contains("PHUKET"));
    System.out.println(c.contains("BANGKOK"));
  }
}
```

true

"SONGKLA"
"PHUKET"
"BANGKOK"
"BANGKOK"

4

false

true

โปรแกรมตรวจสอบข้อความซ้ำซ้อน

- แฟ้มข้อความที่ได้รับ
  มีบรรทัดที่มีข้อความซ้ำกันหรือไม่

Hello
สวัสดี
ลาก่อน
good bye
สวัสดี

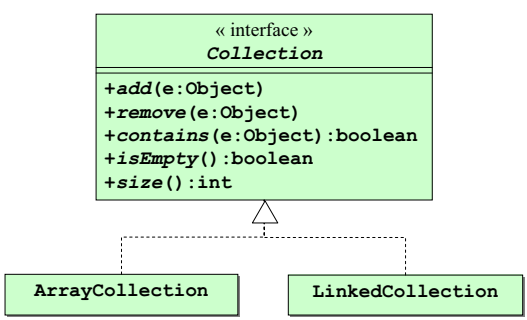
Hello
สวัสดี
ลาก่อน
good bye
สวัสดีครับ

โปรแกรมตรวจสอบข้อความซ้ำซ้อน

```
import java.io.*;

public class Test {
  public static void main(String[] args)
                    throws IOException {
    FileReader fr = new FileReader("data.txt");
    BufferedReader br = new BufferedReader(fr);
    Collection c = new ArrayCollection(100);
    String line;
    while ((line = br.readLine()) != null) {
      if (c.contains(line))
        System.out.println("ข้อความซ้ำ : " + line);
      else
        c.add(line);
    }
    System.out.println(c.size());
  }
}
```

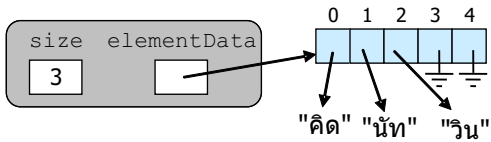

แล้วจะสร้างคอลเล็กชันได้อย่างไร ?

- ใช้อาเรย์ (ArrayCollection)
- ใช้การโยง (LinkedCollection)



ArrayCollection : ใช้อาเรย์เก็บข้อมูล

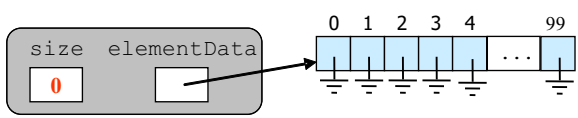
```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
```



ข้อมูลในคอลเล็กชันเก็บอยู่ใน elementData ช่องที่ 0 ถึง size – 1

ArrayCollection : constructor

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;

  public ArrayCollection(int c) {
    elementData = new Object[c];
    size = 0;
  }
```

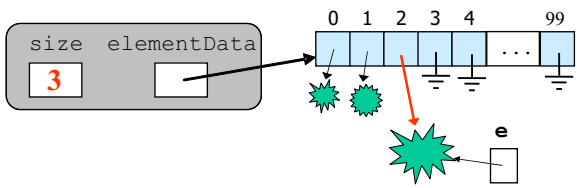



```
Collection c = new ArrayCollection(100);
```

ArrayCollection : add

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    elementData[size++] = e;
  }
```

นำตัวใหม่ต่อท้ายตัวสุดท้าย ใส่ใน elementData[size]



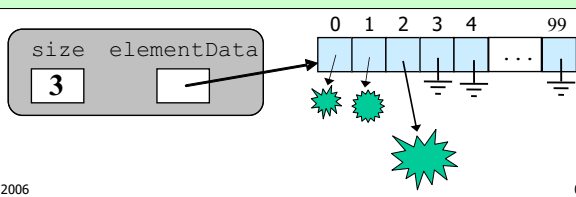
ArrayCollection : size กับ isEmpty

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
```

ใช้ size ให้เป็นประโยชน์

คืน true ถ้า size เป็น 0

คืน false ถ้ามีข้อมูล (size ≠ 0)



ArrayCollection : บริการต่าง ๆ

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;

  public ArrayCollection(int c) {
    elementData = new Object[c];
    size = 0;
  }
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    elementData[size++] = e;
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  ...
```

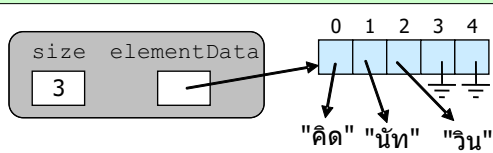
องค์ประกอบของโครงสร้างภายใน

บริการสาธารณะต่าง ๆ ให้กับผู้ใช้

ArrayCollection : contains

```
public class ArrayCollection implements Collection {
  ...
  private int indexOf(Object e) {
    for (int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
  public boolean contains(Object e) {
    return indexOf(e) != -1;
  }
```

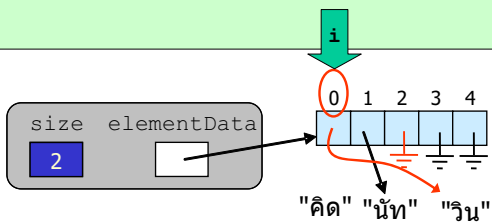
ไม่พบ e, คืน -1
พบ e, คืน index ที่พบ



ArrayCollection : remove

```
public class ArrayCollection implements Collection {
  ...
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      elementData[i] = elementData[--size];
      elementData[size] = null;
    }
  }
```

ย้ายตัวท้ายมาแทน
ตัวที่ต้องการลบออก



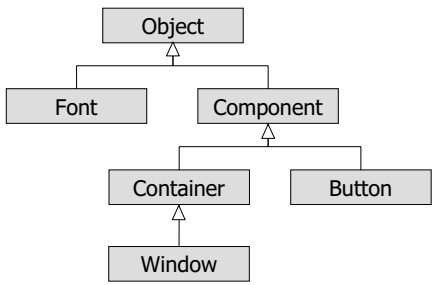
เรื่องจุกจิกของจาวา

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  private int indexOf(Object e) {
    for (int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      elementData[i] = elementData[--size];
      elementData[size] = null;
    }
  }
  ...
```

จาวา : คลาส Object เป็น root class

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
```

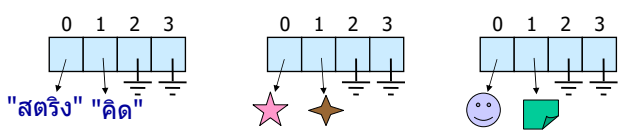
- ทุก ๆ คลาสในจาวาเป็นลูกหลานของคลาส Object



จาวา : ทุกอ็อบเจกต์เป็น Object

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
```

- ทุก ๆ คลาสในจาวาเป็นลูกหลานของคลาส Object
- ทุกอ็อบเจกต์ "เป็น" อ็อบเจกต์ของคลาส Object
- Object[] จึงเป็นอาเรย์เก็บอ็อบเจกต์ได้ทุกแบบ



Wrapper Class

- Object[] เก็บข้อมูลพื้นฐานไม่ได้
  - int, short, byte, long, char, float, double, boolean
- ให้ใช้ Wrapper classes
  - Integer, Short, Byte, Long, Character, Float, Double, Boolean

```
Collection c = new ArrayCollection(10);
c.add(new Integer(12));
c.add(new Float(17.2f));
c.add(new Double(13.2));
```

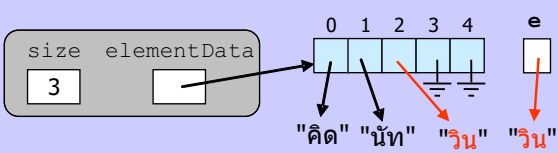
```
Collection c = new ArrayCollection(10);
c.add(12);
c.add(17.2f);      Java 5 ทำ autoboxing ให้
c.add(13.2);
```


จาวา : equals

```
public class ArrayCollection implements Collection {
  ...
  private int indexOf(Object e) {
    for (int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
```

```
if (elementData[i].equals(e)) return i;
```

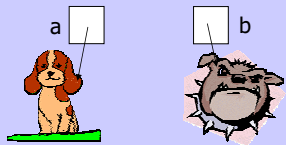
อ็อบเจกต์ `elementData[i]` มีค่าเท่ากับอ็อบเจกต์ `e` หรือไม่



จาวา : dereference

ในจาวา มีแต่ขอสร้างอ็อบเจกต์ ไม่มีการคืนอ็อบเจกต์
อ็อบเจกต์ใดที่ไม่มีตัวแปรอ้างอิงถือว่าเป็น "ขยะ"

```
Dog a = new Dog("ด่าง");
Dog b = new Dog("บูบู้");
a = b;
a = null;
b = null;
```




```
public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      elementData[i] = elementData[--size];
      elementData[size] = null;
    }
  }
```


อาเรย์อาจล้นได้

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    elementData[size++] = e;
  }
```

```
Collection c = new ArrayCollection(3);
c.add("nok");
c.add("kai");
c.add("bird");
c.add("poo");
```

3 ช่องเก็บ 3 ตัว

ArrayCollection : add แบบขยายอาเรย์ได้

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    ensureCapacity(size + 1);
    elementData[size++] = e;
  }
  private void ensureCapacity(int capacity) {
    if (capacity > elementData.length) {
      int s = Math.max(capacity, 2*elementData.length);
      Object[] arr = new Object[s];
      for(int i = 0; i < size; i++)
        arr[i] = elementData[i];
      elementData = arr;
    }
  }
```

ขยายอาเรย์ถ้ามีขนาดไม่พอ capacity

ขยายให้ใหญ่ขึ้นสองเท่า

บริการเสริม : toArray()

- มีคอลเล็กชันแล้วต้องการดึงข้อมูลออกมาใช้งาน
- toArray คืนอาเรย์
  - มีขนาดเท่ากับจำนวนข้อมูลของคอลเล็กชัน
  - เก็บข้อมูลทุกตัวในคอลเล็กชัน

```
ArrayCollection c = new ArrayCollection(4);
c.add("A");
c.add("B");
c.add("C");
c.add("D");
c.remove("B");
Object[] a = c.toArray();
System.out.println(a.length);
System.out.println(Arrays.toString(a));
```

[A,D,C]

ArrayCollection : toArray()

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
  public Object[] toArray() {
    Object[] a = new Object[size];
    for (int i=0; i<size; i++)
      a[i] = elementData[i];
    return a;
  }
```


สรุป

- คอลเล็กชันคือที่เก็บข้อมูลประเภทหนึ่ง
  - ไม่สนใจอันดับของข้อมูล
  - เก็บข้อมูลซ้ำได้
- ArrayCollection : สร้างคอลเล็กชันด้วยอาเรย์
  - เก็บอ็อบเจกต์ได้ทุกประเภท (แต่ไม่เก็บ primitive)
  - มีอาเรย์และตัวแปรอีกตัวเก็บจำนวนข้อมูล
  - ทำให้อาเรย์ขยายขนาดได้ เมื่อเก็บข้อมูลจนเต็ม

การวิเคราะห์เวลาการทำงาน

หัวข้อ

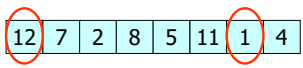
- การเปรียบเทียบเวลาการทำงาน
- การนับจำนวนครั้งที่คำสั่งทำงาน
- การวิเคราะห์ฟังก์ชันเวลาการทำงาน
- อัตราการเติบโตของฟังก์ชัน
- การวิเคราะห์เชิงเส้นกำกับ
- สัญกรณ์เชิงเส้นกำกับ
- การวิเคราะห์เวลาการทำงานของเมท็อดใน ArrayCollection

อยากรู้เวลาการทำงาน

- เขียนโปรแกรม
- สั่งทำงาน
- จับเวลา

ตัวอย่าง : มี int[] d

อยากทราบ ผลต่าง ของคู่ข้อมูลใน d ที่มีผลต่างมากสุด



คำตอบคือ 11

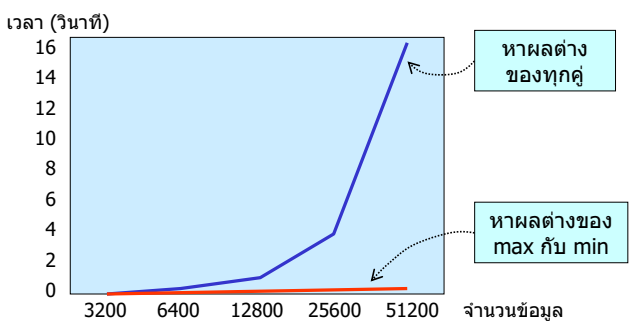
เขียนโปรแกรม : หาผลต่างของทุกคู่

```
public class Test1 {
  public static void main(String[] args) {
    for (int n = 3200; n < 100000; n *= 2) {
      int[] a = new int[n];
      long start = System.nanoTime();
      int diff = maxDiff(a);
      long d = (System.nanoTime() - start);
      System.out.println(n + " : " + d / 1E9);
    }
  }
  static int maxDiff(int[] d) {
    int maxDiff = 0;
    for (int i = 0; i < d.length; i++) {
      for (int j = i+1; j < d.length; j++) {
        int diff = Math.abs(d[i] - d[j]);
        if (diff > maxDiff) maxDiff = diff;
      }
    }
    return maxDiff;
  }
```

เขียนโปรแกรม : หาผลต่างของ max กับ min

```
public class Test2 {
  public static void main(String[] args) {
    for (int n = 3200; n < 100000; n *= 2) {
      int[] a = new int[n];
      long start = System.nanoTime();
      int diff = maxDiff(a);
      long d = (System.nanoTime() - start);
      System.out.println(n + " : " + d / 1E9);
    }
  }
  static int maxDiff(int[] d) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < d.length; i++) {
      if (d[i] > max) max = d[i];
      if (d[i] < min) min = d[i];
    }
    return max - min;
  }
```


เปรียบเทียบเวลาการทำงาน



ให้ n แทนจำนวนข้อมูล

$t(n)$ แทนเวลาในการหาผลต่างของคู่ข้อมูลที่มีผลต่างมากสุด

แบบที่ 1 : $t \propto n^2$

แบบที่ 2 : $t \propto n$

อยากรู้อัตราการเติบโตของเวลาการทำงาน

- เขียนอัลกอริทึม
- กำหนดตัวแปรที่แทนปริมาณข้อมูลขาเข้า
- หาคำสั่งตัวแทน
  - เวลาการทำงานแปรตามจำนวนครั้งที่คำสั่งตัวแทนทำงาน
- วิเคราะห์จำนวนครั้งที่คำสั่งตัวแทนทำงาน
- หาฟังก์ชันของจำนวนครั้งที่คำสั่งตัวแทนทำงานกับปริมาณข้อมูลขาเข้า

นับจำนวนครั้งที่คำสั่งทำงาน

```
public class Test1 {
  public static void main(String[] args) {
    int[] a = new int[5];
    int diff = maxDiff(a);
  }
  static int maxDiff(int[] d) {
    int maxDiff = 0;
    for (int i = 0; i < d.length; i++) {
      for (int j = i+1; j < d.length; j++) {
        int diff = Math.abs(d[i] - d[j]);
        if (diff > maxDiff) maxDiff = diff;
      }
    }
    return maxDiff;
  }
```

5, 10, 20

นับจำนวนครั้งที่คำสั่งทำงาน

```
public class Test2 {
  public static void main(String[] args) {
    int[] a = new int[5];
    int diff = maxDiff(a);
  }
  static int maxDiff(int[] d) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < d.length; i++) {
      if (d[i] > max) max = d[i];
      if (d[i] < min) min = d[i];
    }
    return max - min;
  }
```

5, 10, 20

ใช้เครื่องมือช่วยนับ

	n = 5	n = 10	n = 20	
แบบที่ 1	10	45	190	$n(n\text{-}1)/2$
แบบที่ 2	5	10	20	n

$(5\times4)/2$ $(10\times9)/2$ $(20\times19)/2$

วิเคราะห์ฟังก์ชันเวลาการทำงาน

```
public class Test1 {
  public static void main(String[] args) {
   int[] a = new int[5];
   int diff = maxDiff(a);
  }
  static int maxDiff(int[] d) {
    int maxDiff = 0;
    for (int i = 0; i < d.length; i++) {
      for (int j = i+1; j < d.length; j++) {
        int diff = Math.abs(d[i] - d[j]);
        if (diff > maxDiff) maxDiff = diff;
      }
    }
    return maxDiff;
  }
```

ให้ n คือขนาดของอาเรย์

เลือกบรรทัดนี้เป็นคำสั่งตัวแทน

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1-i) = n^2 - n - \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

วิเคราะห์ฟังก์ชันเวลาการทำงาน

```
public class Test2 {
  public static void main(String[] args) {
   int[] a = new int[5];
   int diff = maxDiff(a);
  }
  static int maxDiff(int[] d) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < d.length; i++) {
      if (d[i] > max) max = d[i];
      if (d[i] < min) min = d[i];
    }
    return max - min;
  }
```

ให้ n คือขนาดของอาเรย์

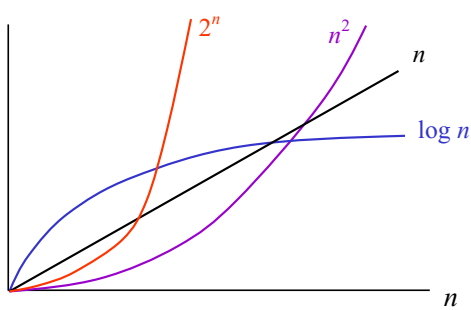
เลือกบรรทัดนี้เป็นคำสั่งตัวแทน

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

การวิเคราะห์ประสิทธิภาพเชิงเวลา

- เขียนโปรแกรมจริง แล้วจับเวลา
  - ต้องเขียนโปรแกรมให้สมบูรณ์
  - เวลาการทำงานขึ้นกับปัจจัยมากมาย
    ภาษา, เครื่อง, ผู้เขียน, ...
- วิเคราะห์อัตราการเติบโตของเวลาการทำงาน
  - เขียนอัลกอริทึม ไม่ต้องลงรายละเอียดมาก
  - เลือกนับเฉพาะคำสั่งตัวแทน
  - ผลที่ได้ใช้เปรียบเทียบได้ดี

อัตราการเติบโตแบบต่าง ๆ



$$\log n \prec n \prec n^2 \prec 2^n$$

ตัวอย่าง : อัตราการเติบโต

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

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

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

$$\sum_{i=1}^{n-1}\sum_{j=3}^{n-2} 1 = \sum_{j=1}^{n-1}(n-4)$$
$$= (n-1)(n-4)$$
$$= n^2 - 5n + 4$$

ตัวอย่าง : อัตราการเติบโต

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

เพิ่มขึ้นทีละ 2 เท่า

เพิ่มขึ้นทีละ 4 เท่า

เพิ่มขึ้นทีละ 4 เท่า

ทั้งคู่มีอัตราการเติบโตเท่ากัน
เป็นแบบ quadratic function

โตเร็ว โตช้า

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0 & f(n)\text{ โตช้ากว่า }g(n)\\ \infty & f(n)\text{ โตเร็วกว่า }g(n)\\ \text{ค่าคงตัว} & f(n)\text{ เติบโตในอัตราเท่ากับ }g(n)\end{cases}$$

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

โตช้า ⟶ โตเร็ว

$n^2,\quad 0.01n^2,\quad 2n^2-10n,\quad 5n^2+8$

⟵ มีอัตราการเติบโตเท่ากัน ⟶

polylogarithm โตช้ากว่า sublinear

$f(n)=\log n \quad g(n)=\sqrt{n}$

$$\lim_{n\to\infty}\frac{\log n}{\sqrt{n}}=\lim_{n\to\infty}\frac{\ln n}{\ln 10\sqrt{n}}$$
$$=\frac{1}{\ln 10}\lim_{n\to\infty}\frac{\ln n}{\sqrt{n}}$$
$$=\frac{1}{\ln 10}\lim_{n\to\infty}\frac{1/n}{1/\left(2\sqrt{n}\right)}$$
$$=\frac{1}{\ln 10}\lim_{n\to\infty}\frac{2}{\sqrt{n}}$$
$$=0$$

$\log n$ โตช้ากว่า $n^{0.5}$

สามารถแสดงได้ว่า
$(\log n)^c$ โตช้ากว่า n^k
$c, k > 0$

$(\log n)^{1000}$ โตช้ากว่า $n^{0.0001}$

การวิเคราะห์เชิงเส้นกำกับ

- ใช้วิเคราะห์พฤติกรรมการทำงานของอัลกอริทึม เมื่อข้อมูลขาเข้ามีขนาดใหญ่
- ช่วยให้วิเคราะห์ได้ง่ายขึ้น
- นำผลมาเปรียบเทียบได้ง่าย

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n \qquad \sum_{i=1}^{n-1}\sum_{j=3}^{n-2} 1 = \sum_{i=1}^{n-1}(n-4)$$

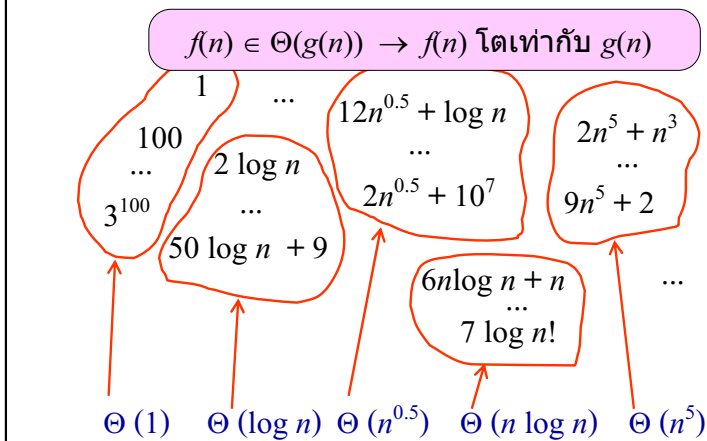
$$= n^2 \qquad = \sum_{i=1}^{n-1}\Theta(n) = \Theta\left(\sum_{j=1}^{n-1} n\right)$$

$$= \Theta(n^2) \quad \text{โตเร็วเท่ากัน} \quad = \Theta(n^2)$$

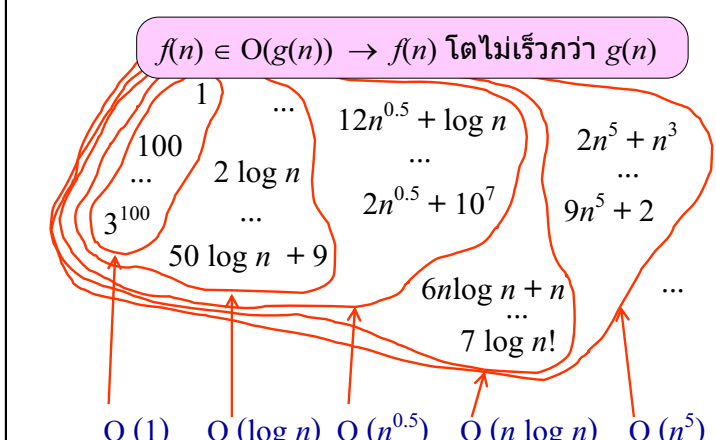
สัญกรณ์เชิงเส้นกำกับ

- $o(g(n))$ คือ เซตของฟังก์ชันที่โตช้ากว่า $g(n)$
- $\omega(g(n))$ คือ " " โตเร็วกว่า $g(n)$
- $\Theta(g(n))$ คือ " " โตเท่ากับ $g(n)$
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- $O(g(n))$ คือ " " โตไม่เร็วกว่า $g(n)$
  - $O(g(n)) = o(g(n)) \cup \Theta(g(n))$
- $\Omega(g(n))$ คือ " " โตไม่ช้ากว่า $g(n)$
  - $\Omega(g(n)) = \omega(g(n)) \cup \Theta(g(n))$

$\Theta(g(n))$ คือเซตของฟังก์ชันที่โตเท่ากับ $g(n)$

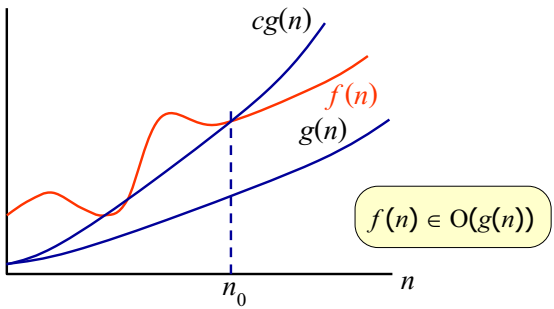


$O(g(n))$ คือเซตของฟังก์ชันที่โตไม่เร็วกว่า $g(n)$



นิยาม $O(g(n))$ อีกแบบ

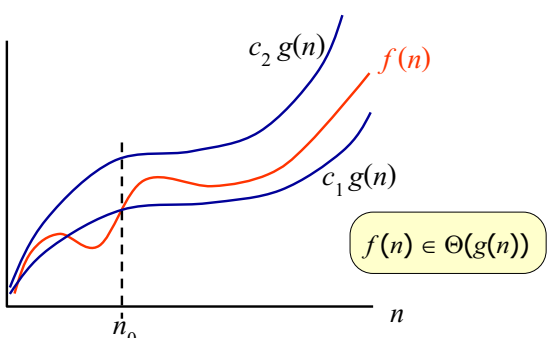
$O(g(n)) = \{ f(n) \mid$ มีจำนวน $c > 0$ และ $n_0 \geq 0$
ที่ทำให้ $f(n) \leq cg(n)$ เมื่อ $n \geq n_0 \}$



$f(n) \in O(g(n))$

นิยาม $\Theta(g(n))$ อีกแบบ

$\Theta(g(n)) = \{ f(n) \mid$ มีจำนวน $c_1, c_2 > 0$ และ $n_0 \geq 0$
ที่ทำให้ $c_1 g(n) \leq f(n) \leq c_2 g(n)$ เมื่อ $n \geq n_0 \}$



$f(n) \in \Theta(g(n))$

ตัวอย่างที่ 1

$O(g(n)) = \{ f(n) \mid$ มีจำนวน $c > 0$ และ $n_0 \geq 0$
ที่ทำให้ $f(n) \leq cg(n)$ เมื่อ $n \geq n_0 \}$

จงแสดงว่า $15n^2 + 10n \in O(n^2)$

หา c และ n_0 ที่ทำให้ $15n^2 + 10n \leq cn^2$ เมื่อ $n \geq n_0$

หารด้วย n^2 ได้ $15 + 10/n \leq c$

ให้ $c = 16$ จะได้ $15 + 10/n \leq 16$ เมื่อ $n \geq 10$

สรุป $15n^2 + 10n \leq 16n^2$ เมื่อ $n \geq 10$

ดังนั้น $15n^2 + 10n \in O(n^2)$

ตัวอย่างที่ 2

$\Theta(g(n)) = \{ f(n) \mid$ มีจำนวน $c_1, c_2 > 0$ และ $n_0 \geq 0$
ที่ทำให้ $c_1 g(n) \leq f(n) \leq c_2 g(n)$ เมื่อ $n \geq n_0 \}$

จงแสดงว่า $15n^2 + 10n \in \Theta(n^2)$

ได้แสดงแล้วว่า $15n^2 + 10n \leq 16n^2$ เมื่อ $n \geq 10$

หา c_1 และ n_0 ที่ทำให้ $c_1 n^2 \leq 15n^2 + 10n$ เมื่อ $n \geq n_0$

ให้ $c_1 = 1$ จะได้ $n^2 \leq 15n^2 + 10n$ เมื่อ $n \geq 0$

สรุป $n^2 \leq 15n^2 + 10n \leq 16n^2$ เมื่อ $n \geq 10$

ดังนั้น $15n^2 + 10n \in \Theta(n^2)$

ตัวอย่างที่ 3

จงแสดงว่า $\sum_{i=1}^{n}\left(\frac{i}{2}\right)^k \in \mathrm{O}\left(n^{k+1}\right)$

$$\sum_{i=1}^{n}\left(\frac{i}{2}\right)^k < \sum_{i=1}^{n}\left(\frac{n}{2}\right)^k$$

$$< \sum_{i=1}^{n} n^k$$

$$= n^{k+1}$$

$$\in \mathrm{O}\left(n^{k+1}\right)$$

ตัวอย่างที่ 4

จงแสดงว่า $\sum_{h=0}^{\lfloor \log_2 n \rfloor}\left(\frac{n}{2^h}h\right) = \mathrm{O}(n)$

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor}\left(\frac{n}{2^h}h\right) = n\sum_{h=0}^{\lfloor \log_2 n \rfloor}\left(\frac{h}{2^h}\right)$$

$$< n\sum_{h=0}^{\infty}\left(\frac{h}{2^h}\right)$$

$$= 2n \quad \in \mathrm{O}(n)$$

ตัวอย่างที่ 5 : $\log n! \in \Theta(n \log n)$

ขอบเขตบน

$$\begin{aligned} n! &= n \times (n\text{-}1) \times \ldots \times 2 \times 1 \\ &\leq n \times n \times \ldots \times n \times n = n^n \\ \log n! &\leq \log n^n = n \log n \quad \text{เมื่อ } n \geq 1 \end{aligned}$$

ขอบเขตล่าง

$$\begin{aligned} n! &= n \times (n\text{-}1) \times \ldots \times (n/2) \times (n/2\text{-}1) \times \ldots \times 2 \times 1 \\ &\geq n/2 \times n/2 \times \ldots \times n/2 \times 1 \times \ldots \times 1 \times 1 \\ &\geq (n/2)^{n/2} \\ \log n! &\geq (n/2) \log (n/2) \\ &= (n/2) \log n - (n/2) \\ &\geq 0.4n \log n \quad \text{เมื่อ } n \geq 10^5 \end{aligned}$$

$$0.4n \log n \leq \log n! \leq n \log n \quad \text{เมื่อ } n \geq 10^5$$

$$\log n! \in \Theta(n \log n)$$

อัตราการเติบโตที่พบบ่อย

- constant : $\Theta(1)$
- logarithmic : $\Theta(\log n)$
- polylogarithmic: $\Theta(\log^c n)$, $c \geq 1$
- sublinear : $\Theta(n^a)$, $0 < a < 1$
- linear : $\Theta(n)$
- quadratic : $\Theta(n^2)$
- polynomial : $\Theta(n^c)$, $c \geq 1$
- exponential : $\Theta(c^n)$, $c > 1$

การเขียนฟังก์ชันในรูปของ O, Θ แบบง่าย ๆ

- ผลบวกของพจน์หลายพจน์ เลือกพจน์ที่โตเร็วสุด
- ข้อสังเกต
  - $cg(n) = \Theta(g(n))$ เมื่อ c เป็นค่าคงตัว
  - $\log_a n = \Theta(\log_b n)$ เพราะ $\log_a n = (\log_a b) \log_b n$
  - . $\sum \Theta(t(n)) = \Theta\left(\sum t(n)\right)$
- เช่น
  - $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 = \Theta(n^k)$
  - $0.001n^3 + 7000n^2 - 11 = \Theta(n^3)$
  - $\log_2 n^{10} = 10(\log_2 n) = \Theta(\log n)$
  - $\cdot \sum_{i=1}^{n} \Theta(i) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(n(n+1)/2\right) = \Theta(n^2)$

กลับมาวิเคราะห์ ArrayCollection กัน

- constructor
  - เป็นการจองอาเรย์จำนวนช่องตามที่ผู้ใช้กำหนด จึงใช้เวลาแปรตามขนาด
  - ใช้เวลา Θ(c)
- isEmpty และ size
  - ทำแค่ 1 คำสั่ง : ใช้เวลา Θ(1)

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int size;
  public ArrayCollection(int c) {
    elementData = new Object[c];
  }
  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }
  ...
```

ArrayCollecton.add

- ถ้าไม่ต้องขยายอาเรย์ ก็ใช้เวลาแค่ $\Theta(1)$
- ให้ n แทนจำนวนข้อมูลในคอลเลกชัน
- ถ้าข้อมูลเต็มอาเรย์ เกิดการขยาย จึงใช้เวลา $\Theta(n)$

```
public void add(Object e) {
  if(e == null) throw new IllegalArgumentException();
  ensureCapacity(size+1);
  elementData[size++] = e;
}
private void ensureCapacity(int capacity) {
  if (capacity > elementData.length) {
    int s = Math.max(capacity, 2*elementData.length);
    Object[] arr = new Object[s];
    for(int i=0; i<elementData.length; i
      arr[i] = elementData[i];
    elementData = newA;
  }
}
```

$\Theta(n)$

ตอนที่เต็ม n = ขนาดของอาเรย์

ArrayCollection.contains

- contains เรียก indexOf เพื่อค้นหาข้อมูลแบบลำดับ
- ทำการค้นด้วยวงวน for หมุนไม่เกิน n รอบ
- ถ้าโชคดีก็พบที่ช่องที่ 0 โชคร้ายก็พบช่องท้าย ๆ
- จึงใช้เวลาการค้นเป็น $O(n)$

```
public boolean contains(Object e) {
  return indexOf(e) != -1;
}
private int indexOf(Object e) {
  for(int i=0; i<size; i++)
    if (elementData[i].equals(e)) return i;
  return -1;
}
```

$O(n)$

ArrayCollection.remove

- remove เรียก indexOf เพื่อค้นหาข้อมูลแบบลำดับ
- ทำการค้นด้วยวงวน for ใช้เวลา $O(n)$
- ทำการลบใช้เวลา $\Theta(1)$
- remove ใช้เวลาเป็น $O(n)$

```
public void remove(Object e) {
  int i = indexOf(e);
  if (i != -1) {
    elementData[i] = elementData[--size];
    elementData[size] = null;
  }
}
```

$O(n)$

$\Theta(1)$

สรุป

- ใช้จำนวนครั้งที่คำสั่งตัวแทนทำงานแทนเวลา
- หาความสัมพันธ์ของจำนวนครั้งที่คำสั่งตัวแทนทำงานกับจำนวนข้อมูล
- เขียนฟังก์ชันในรูปของสัญกรณ์เชิงเส้นกำกับ
- ใช้ O เพื่อแสดงขอบเขตบนของเวลา
- ใช้ Θ เมื่อการทำงานมีขอบเขตบนและล่างเท่ากัน

การเก็บข้อมูลด้วยการโยง

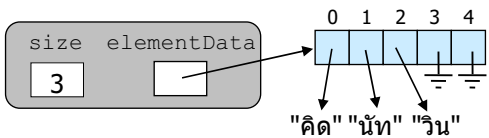
(LinkedCollection)

หัวข้อ

- การสร้างคอลเล็กชันด้วยการโยงข้อมูล
- การโยงแบบไม่มีและมีปมหัว
- การสร้างเซตด้วยคอลเล็กชัน

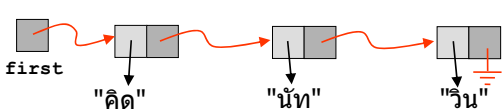
การเก็บข้อมูลด้วยอาเรย์

- แต่ละช่องเก็บ reference ไปยังออปเจกต์
- ช่องใดไม่เก็บข้อมูล ก็เก็บ null
- ข้อดี : เข้าถึง elementData[k] ใดๆ ได้ง่ายและรวดเร็ว
- ข้อเสีย : ต้องจองอาเรย์ไว้ก่อน ไม่พอต้องขยาย



การเก็บข้อมูลด้วยการโยง

- เก็บข้อมูลไว้ตาม "ปม" ข้อมูล
- เชื่อมโยงปมต่าง ๆ ให้ถึงกัน
- แต่ละปมเก็บ
  - reference ไปยังข้อมูล
  - reference ไปยังปมถัดไป (ไม่มีปมถัดไปเก็บ null)
- มีตัวแปรเก็บปมแรก
- ข้อดี : จองปมข้อมูลเท่าที่จำเป็น
- ข้อเสีย : เปลืองเนื้อที่สำหรับเก็บการโยง



การสร้างคอลเลกชันด้วยการโยง

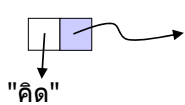
```
public interface Collection {
  public void add(Object element);
  public void remove(Object element);
  public boolean isEmpty();
  public boolean contains(Object element);
  public int size();
}
```

```
public class LinkedCollection implements Collection {
  public LinkedCollection() { ... }
  public void add(Object element) { ... }
  public void remove(Object element) { ... }
  public boolean isEmpty() { ... }
  public boolean contains(Object element) { ... }
  public int size() { ... }
}
```


LinkedNode : โครงสร้างของปม

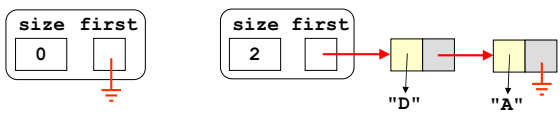
```
public class LinkedCollection implements Collection {
  private static class LinkedNode {
    Object element;
    LinkedNode next;
    LinkedNode(Object e, LinkedNode next) {
      this.element = e;
      this.next = next;
    }
  }
  ...
```

static inner class



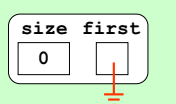
LinkedCollection : โครงสร้างการเก็บข้อมูล

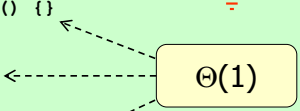
```
public class LinkedCollection implements Collection {
  private static class LinkedNode { ... }

  private LinkedNode first;
  private int size;

  public LinkedCollection() {
    first = null;
    size = 0;
  }
  ...
```



isEmpty, size

```
public class LinkedCollection implements Collection {
  private static class LinkedNode { ... }

  private LinkedNode first;
  private int size;

  public LinkedCollection() {}

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

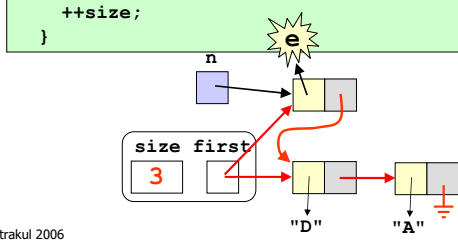




add : การเพิ่มข้อมูล

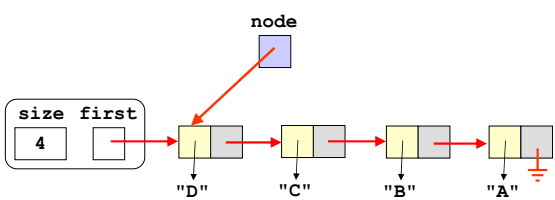
- สร้างปมข้อมูลใหม่ แล้วแทรกไว้ด้านหน้า

```
public class LinkedCollection implements Collection {
  private LinkedNode first;
  private int size;
  ...
  public void add(Object e) {

    first = new LinkedNode(e, first);

    ++size;
  }
```



Θ(1)

การเลื่อนไปทีละปมตามการโยง




```
LinkedNode node = first;
node = first.next;
node = node.next;
node = node.next;
node = node.next;
```

การเลื่อนไปยังปมถัดไป

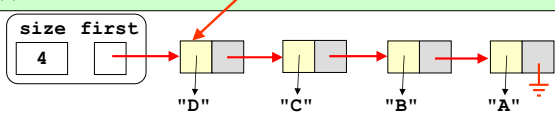
contains : การค้นข้อมูล

- ค่อย ๆ วิ่งไล่เปรียบเทียบจาก first

```
public class LinkedCollection implements Collection {
  private LinkedNode first;
  private int size;
  ...
  public boolean contains(Object e) {
    LinkedNode node = first;
    while( node != null ) {
      if (node.element.equals(e)) return true;
      node = node.next;
    }
    return false;
  }
  ...
```

node

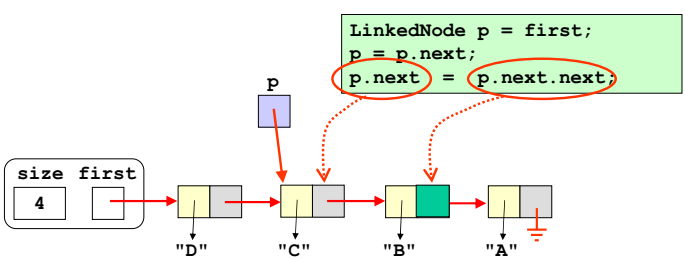
$O(n)$



remove : การลบข้อมูล

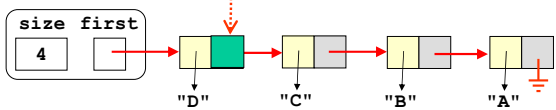
- ต้องการลบปมใดออก ต้องรู้ตำแหน่งของปมก่อนหน้า
  - ต้องการลบปม "B", หาปมที่ปมถัดไปเก็บ "B"
- เปลี่ยนตัวโยงให้ข้ามตัวที่ต้องการลบ

```
LinkedNode p = first;
p = p.next;
p.next = p.next.next;
```



remove : การลบข้อมูลที่ปมแรก

- การลบ node แรก จะเป็นกรณีพิเศษ
- ตัวอย่าง : remove("D")
  - ต้องทำ first = first.next;



remove : การลบข้อมูล

```
public class LinkedCollection implements Collection {
  private LinkedNode first;
  private int size;
  ...
  public void remove(Object e) {
    if (first == null) return;
    if (first.element.equals(e)) {
      first = first.next; --size;
    } else {
      LinkedNode p = first;
      while( p.next != null &&
             ! p.next.element.equals(e)) {
        p = p.next;
      }
      if (p.next != null) {
        p.next = p.next.next; --size;
      }
    }
  }
```

คอลเลกชันว่าง

ลบปมแรก

ค้นให้พบ

ค้นพบ ก็ลบ

$O(n)$

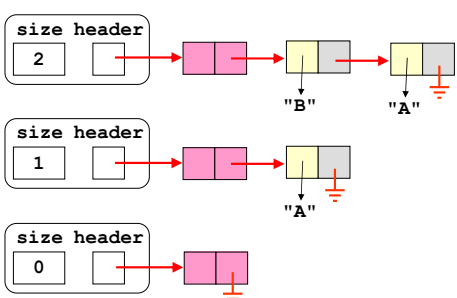
การลบมีหลายกรณี

- remove น่ารำคาญที่ต้องมีกรณีพิเศษ

```
public void remove(Object e) {
  if (first == null) return;
  if (first.element.equals(e)) {
    first = first.next; --size;
  } else {
    LinkedNode p = first;
    while( p.next != null &&
           ! p.next.element.equals(e)) {
      p = p.next;
    }
    if (p.next != null) {
      p.next = p.next.next; --size;
    }
  }
}
```


การโยงแบบมีปมหัว (header)

- ต้องมี "ปมหัว" เป็นปมแรก
- ห้ามลบปมหัว
- ปมหัวนี้ไม่เก็บข้อมูล



การโยงแบบมีปมหัว

```
public class LinkedCollection implements Collection {
  private LinkedNode first;
  private int size;

  public LinkedCollection() { }
  public void add(Object e) {
    first = new LinkedNode(e, first);
    ++size;
  }
```

ไม่มีปมหัว

```
public class LinkedCollection implements Collection {
  private LinkedNode header
                    = new LinkedNode(null, null);
  private int size;

  public LinkedCollection() { }
  public void add(Object e) {
    header.next = new LinkedNode(e, header.next);
    ++size;
  }
```

มีปมหัว

การโยงแบบมีปมหัว : contains

```
public class LinkedCollection implements Collection {
  ...
  public boolean contains(Object e) {
    LinkedNode node = first;
    while( node != null ) {
      if (node.element.equals(e)) return true;
      node = node.next;
    }
    return false;
  }
```

ไม่มีปมหัว

```
public class LinkedCollection implements Collection {
  ...
  public boolean contains(Object e) {
    LinkedNode node = header.next;
    while( node != null ) {
      if (node.element.equals(e)) return true;
      node = node.next;
    }
    return false;
  }
```

มีปมหัว

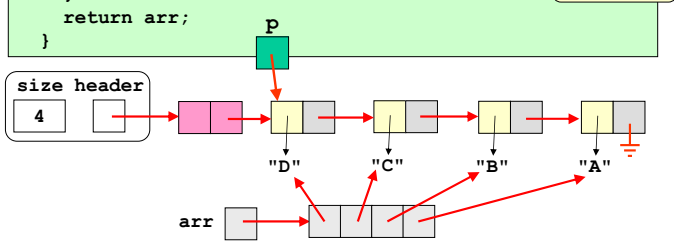
การโยงแบบมีปมหัว : remove

```
public class LinkedCollection implements Collection {
  private LinkedNode header = new LinkedNode(null,null);
  private int size;
  ...
  public void remove(Object e) {
    if (first == null) return;
    if (first.element.equals(e)) {
      first = first.next; --size;
    } else {
       LinkedNode p = header;
       while( p.next != null &&
              ! p.next.element.equals(e)) {
         p = p.next;
       }
       if (p.next != null) {
         p.next = p.next.next; --size;
       }
    }
  }
```


บริการเสริม : toArray

```
  public Object[] toArray() {
    Object[] arr = new Object[size];
    LinkedNode p = header.next;
    int k = 0;
    while (p != null) {
      arr[k++] = p.element;
      p = p.next;
    }
    return arr;
  }
```

$\Theta(n)$



Set คือ Collection ที่ไม่เก็บตัวซ้ำ

- Set ก็เป็น collection อย่างหนึ่ง ที่ห้ามเก็บตัวซ้ำ

```
public interface Set extends Collection {
  /**
   * add a new element without duplication.
   */
  public void add(Object element);
}
```


การสร้างเซตด้วยคอลเลกชัน : inheritance

```
public class ArraySet extends ArrayCollection
                      implements Set {
  public void add(Object element) {
    if (!contains(element)) {
      add(element);
    }
  }
}
```

```
public class LinkedSet extends LinkedCollection
                       implements Set {
  public void add(Object element) {
    if (!contains(element)) {
      add(element);
    }
  }
}
```

การสร้างเซตด้วยคอลเลกชัน : composition

```
public class LinkedSet implements Set {
  private Collection c;
  public LinkedSet() {
    c = new LinkedCollection();
  }
  public boolean isEmpty() {
    return c.isEmpty();
  }
  public int size() {
    return c.size();
  }
  public boolean contains(Object e) {
    return c.contains(e);
  }
  public void remove(Object e) {
    return c.remove(e);
  }
  public void add(Object e) {
    if (!c.contains(e)) c.add(e);
  }
}
```


สรุป

- ข้อมูลเก็บในปมข้อมูล
- นำปมข้อมูลมาโยง
- โยงแบบไม่มีปมหัว มักมีกรณีพิเศษที่ต้องจัดการ
- โยงแบบมีปมหัว
  - เพิ่มความซับซ้อนในการจัดเก็บ
  - แต่ลดความซับซ้อนในการจัดการ

รายการ

(ArrayList & LinkedList)

หัวข้อ

- นิยามรายการ และอินเตอร์เฟส List
- การสร้างรายการด้วยอาเรย์
- การสร้างรายการด้วยการโยง
  - โยงเดี่ยวแบบไม่วนที่มีปมหัว
  - โยงคู่แบบวนที่มีปมหัว
- การสร้างเวกเตอร์มากเลขศูนย์ด้วยรายการ

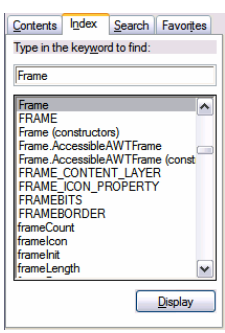
รายการ (List)

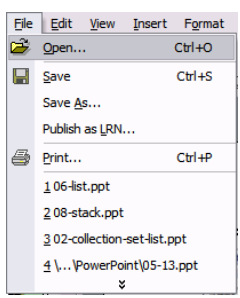
- List ก็เป็น Collection อย่างหนึ่ง
- เก็บแบบมีอันดับ ข้อมูลแต่ละตัวมีหมายเลขกำกับ

$$< a_0, a_1, a_2, ..., a_{n-1} >$$

```
public interface List extends Collection {
  public void add(int index, Object e);
  public void remove(int index);
  public Object get(int index);
  public void set(int index, Object e);
  public int indexOf(Object e);
}
```


ตัวอย่างรายการ






```
<"SU", "MO", "TU", "WE", "TH", "FR", "SA">
```

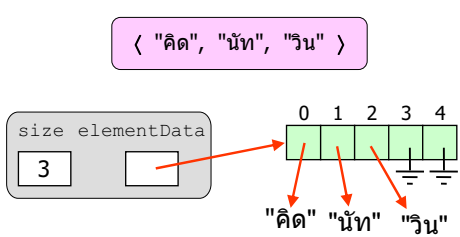
$f(x) = 2x^5 + 4x^3 - 6x + 9$ $\quad < (2,5), (4,3), (-6,1), (9,0) >$

ตัวอย่างการใช้งาน

```
List x = new ArrayList(10);     ⟨ ⟩
x.add(0,"A");                   ⟨ "A" ⟩
x.add("B");                     ⟨ "A", "B" ⟩
x.add(0,"C");                   ⟨ "C", "A", "B" ⟩
x.set(2,"D");                   ⟨ "C", "A", "D" ⟩
int i = x.indexOf("A");
x.add(i,"Z");                   ⟨ "C", "Z", "A", "D" ⟩
x.remove(2);                    ⟨ "C", "Z", "D" ⟩
for(int i=0; i<x.size(); i++)
  System.out.println(x.get(i));
```


การสร้างรายการด้วยอาเรย์

- สร้างอาเรย์เก็บข้อมูลเริ่มตั้งแต่ช่องที่ 0
- มีตัวแปร size เก็บจำนวนข้อมูล
- $< a_0, a_1, a_2, ..., a_{n-1} >$ เก็บข้อมูลไว้ที่ elementData[0], ..., elementData[$n-1$] ตามลำดับ



ArrayList : เมท็อดของ Collection

```
public class ArrayList implements List {
  private Object[] elementData;
  private int size;
  public ArrayList(int cap) {
    elementData = new Object[cap];
    size = 0;
  }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public boolean contains(Object e) {
    return indexOf(e) != -1;
  }
  public void add(Object e) {
    add(size, e);
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (i >= 0) remove(i);
  }
```


indexOf, get, set

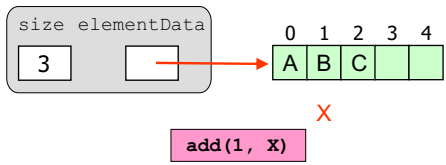
```
public class ArrayList implements List {
  ...
  public int indexOf(Object e) {
    for(int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
  public Object get(int index) {
    return elementData[index];
  }
  public void set(int index, Object e) {
    elementData[index] = e;
  }
```

$O(n)$

$\Theta(1)$

$\Theta(1)$

add(index, e)

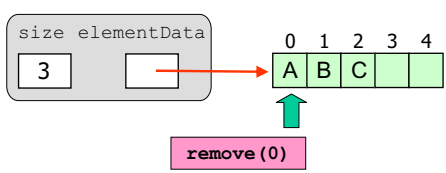



```
public void add(int index, Object e) {
  ensureCapacity(size+1);
  for(int i=size; i>index; i--) {
    elementData[i] = elementData[i-1];
  }
  elementData[index] = e;
  size++;
}
```

$O(n)$

add(0,e) ช้า add(size,e) เร็ว

remove(index)

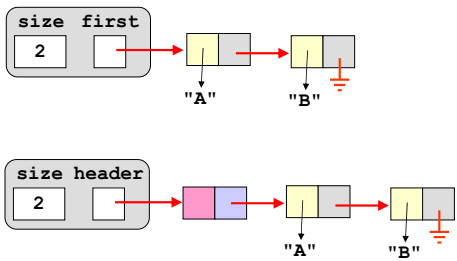



```
public void remove(int index) {
  for(int i=index+1; i<size; i++) {
    elementData[i-1] = elementData[i];
  }
  size--;
  elementData[size] = null;
}
```

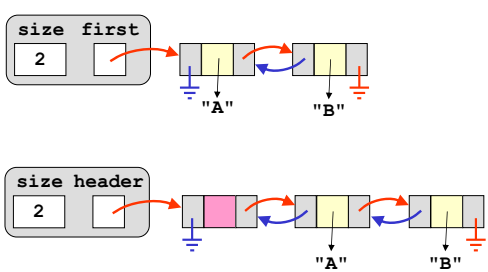
$O(n)$

remove(0) ช้า remove(size-1) เร็ว

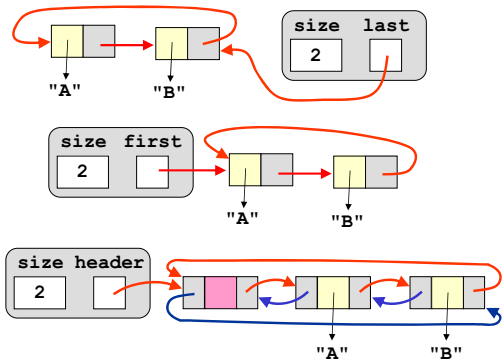
การสร้างแบบโยงเดี่ยว (singly linked)



การสร้างแบบโยงคู่ (doubly linked)



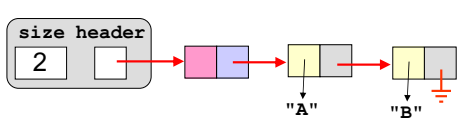
การสร้างแบบโยงวน (circular)



รายการโยงเดี่ยวแบบไม่วนที่มีปมหัว

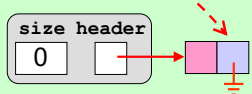
```
public class SinglyLinkedList implements List {
  private static class LinkedNode {
    Object element;
    LinkedNode next;
    LinkedNode(Object e, LinkedNode n) {
      this.element = e;
      this.next = n;
    }
  }
  private LinkedNode header;
  private int size;
  ...
```

Singly linked list with header



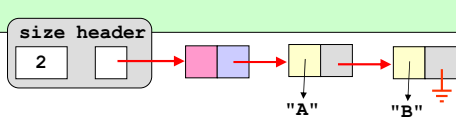
SinglyLinkedList

```
public class SinglyLinkedList implements List {
  ...
  private LinkedNode header = new LinkedNode(null,null);
  private int size = 0;

  public SinglyLinkedList() { }
  public boolean isEmpty() {
    return header.next == null;  // size == 0
  }
  public int size() {
    return size;
  }
  ...
```



indexOf(e), contains(e)

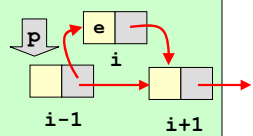
```
public class SinglyLinkedList implements List {
  ...
  public int indexOf(Object e) {
    LinkedNode q = header.next;
    for (int i=0; i<size; i++) {
      if (q.element.equals(e)) return i;
      q = q.next;
    }
    return -1;
  }
  public boolean contains(Object e) {
    return indexOf(e) >= 0;
  }
  ...
```



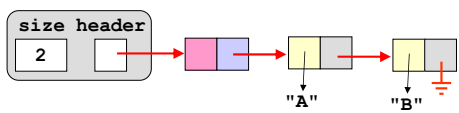
add(e), add(i, e)

```
public void add(Object e) {
  add(size, e);
}
public void add(int i, Object e) {
  LinkedNode p = nodeAt(i-1);
  p.next = new LinkedNode(e, p.next);
  ++size;
}
private LinkedNode nodeAt(int i) {
  LinkedNode p = header;
  for (int j = -1; j < i; j++) p = p.next;
  return p;
}
```

ขอปมที่อยู่ตำแหน่ง i – 1

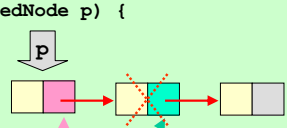


i เป็น –1 จะคืน header



remove(e), remove(i)

```
private void removeAfter(LinkedNode p) {
  if (p.next != null) {
    p.next = p.next.next;
    --size;
  }
}
public void remove(Object e) {
  LinkedNode p = header;
  while (p.next != null && !p.next.element.equals(e))
    p = p.next;
  removeAfter(p);
}
public void remove(int i) {
  LinkedNode p = nodeAt(i-1);
  removeAfter(p);
}
```



วิ่งหาปมก่อน
หน้าปมที่เก็บ e

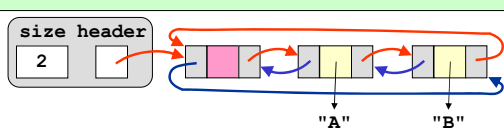
get(i), set(i, e)

```
public Object get(int i) {
  return nodeAt(i).element;
}
public void set(int i, Object e) {
  nodeAt(i).element = e;
}
```


รายการโยงคู่แบบวนที่มีปมหัว

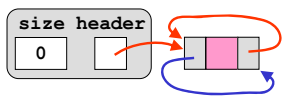
```
public class LinkedList implements List {
  private static class LinkedNode {
    Object element;
    LinkedNode prev, next;
    LinkedNode(Object e, LinkedNode p, LinkedNode n) {
      this.element = e;
      this.prev = p;
      this.next = n;
    }
  }
  private LinkedNode header;
  private int size;
  ...
```

Circular doubly linked list with header



constructor

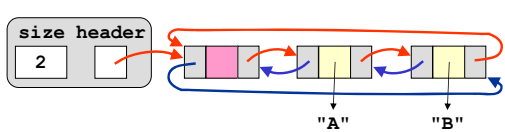
```
public class LinkedList implements List {
  ...
  private LinkedNode header;
  private int size;

  public LinkedList() {
    header = new LinkedNode(null, null, null);
    header.prev = header.next = header;
  }
```



เมท็อดที่เหมือนกับ SinglyLinkedList

- size()
- isEmpty()
- indexOf(e)
- contains(e)
- nodeAt(i)
- get(i)
- set(i,e)

เมท็อดที่ไม่เหมือน
(เพราะต้องจัดการ prev ด้วย)
- add(e)
- add(i,e)
- remove(e)
- remove(i)



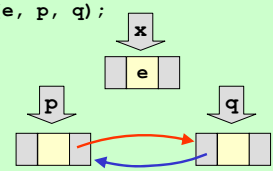
add(e), add(i, e)

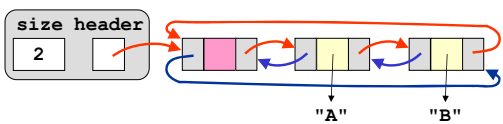
```
private void addBefore(LinkedNode q, Object e) {
  LinkedNode p = q.prev;
  LinkedNode x = new LinkedNode(e, p, q);
  p.next = q.prev = x;
  ++size;
}
public void add(Object e) {
  addBefore(header, e);
}
public void add(int i, Object e) {
  addBefore(nodeAt(i), e);
}
```

Θ(1)

Θ(1)

O(*n*)





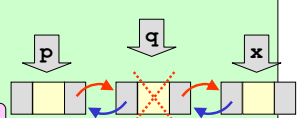
remove(e), remove(i)

```
private void removeNode(LinkedNode q) {
  LinkedNode p = q.prev;
  LinkedNode x = q.next;
  p.next = x;
  x.prev = p;
  --size;
}
public void remove(int i) {
  removeNode(nodeAt(i));
}
public void remove(Object e) {
  LinkedNode q = header.next;
  while (q != header) {
    if (q.element.equals(e)) { removeNode(q); break; }
    q = q.next;
  }
}
```

Θ(1)

O(*n*)

O(*n*)



เวกเตอร์มากเลขศูนย์ (Sparse Vector)

- นิยามของเวกเตอร์มากเลขศูนย์
- การสร้างเวกเตอร์มากเลขศูนย์ด้วยรายการ
- รายละเอียดของบริการต่าง ๆ

เวกเตอร์มากเลขศูนย์

- ต้องการใช้เวกเตอร์ เช่น (0,3,0,0,0,0,4)
- ใช้อาเรย์ double[ ] x = {0,3,0,0,0,0,4};
- ถ้าเวกเตอร์มีจำนวนส่วนใหญ่เป็น 0
- เก็บเป็นรายการของตัวที่ไม่ใช่ 0 ⟨(1,3),(6,4)⟩

```
SparseVector v1 = new SparseVector(7);
v1.set(1,3);                    // (0,3,0,0,0,0,0)
v1.set(6,4);                    // (0,3,0,0,0,0,4)
SparseVector v2 = v1.add(v1); // (0,6,0,0,0,0,8)
v2.set(2,3);                    // (0,6,3,0,0,0,8)
double d = v1.dot(v2);
```

```
v1 = (0,3,0,0,0,0,4)
v2 = (0,6,3,0,0,0,8)
v1·v2 =   18   +  32 = 50
```

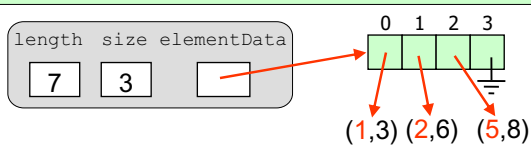
บริการของ SparseVector

```
public class SparseVector {
  public int length() {...}
  public double get(int i) {...}
  public void set(int i, double x) {...}
  public SparseVector add(SparseVector v) {...}
  public double dot(SparseVector v) {...}
  public SparseVector multiply(double x) {...}
}
```


สร้างด้วยแถวลำดับ

- elementData เก็บแถวลำดับ
  - แต่ละช่องเก็บคู่ลำดับ (index, value) เก็บเรียง index จากน้อยไปมาก
- size เก็บจำนวนคู่ลำดับในรายการ
- length เก็บความยาว vector

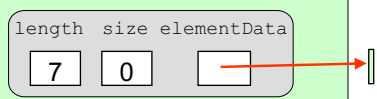
```
SparseVector v1 = new SparseVector(7);
v1.set(1,3);                  // (0,3,0,0,0,0,0)
v1.set(5,8);                  // (0,3,0,0,0,8,0)
v1.set(2,6);                  // (0,3,6,0,0,8,0)
```



SparseVector

```
public class SparseVector {
  private static class Element {
    int index;
    double value;
    Element(int i, double v) {
      this.index = i;  this.value = v;
    }
  }
  private int size;
  private int length;
  private Element[] elementData;
  public SparseVector(int length) {
    this.elementData = new Element[0];
    this.size = 0;
    this.length = length;
  }
  public int length() { return length; }
```

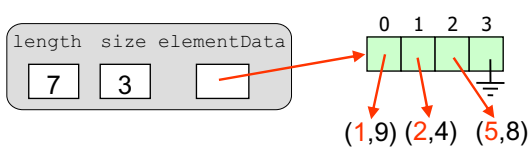
เก็บคู่ลำดับ (index, value)



SparseVector : get

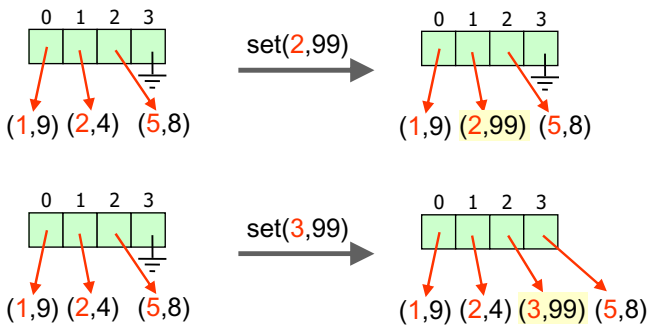
```
public class SparseVector {
  ...
  public double get(int index) {
    for(int i=0; i<size; i++) {
      if (elementData[i].index == index)
        return elementData[i].value;
      if (elementData[i].index > index) break;
    }
    return 0.0;
  }
```

$O(n)$



SparseVector : set

```
public class SparseVector {
  ...
  public void set(int index, double value) {
```



SparseVector : set

```
  public void set(int index, double value) {
    int i = 0;
    for ( ;i<size; i++ )
      if (elementData[i].index >= index) break;
    if (i<size && elementData[i].index == index)
      elementData[i].value = value;
    else
      add(i, index, value);
  }
  void add(int i, int index, double value) {
    if (value != 0) {
      ensureCapacity(size+1);
      for (int k=size; k>i; k--)
        elementData[k] = elementData[k-1];
      elementData[i] = new Element(index, value);
      ++size;
    }
  }
```

$O(n)$

SparseVector : dot

```
public double dot(SparseVector v2) {
  SparseVector v1 = this;
  double r = 0;
  for (int i = 0; i<v1.length(); i++)
    r += v1.get(i) * v2.get(i);
  return r;
}
```

```
v1 = (0,3,0,0,1,4,0)
v2 = (0,6,3,0,0,3,0)
r = 0*0 + 3*6 + 0*3 +
    0*0 + 1*0 + 4*3 + 0*0
```

- เสีย O(n) ในแต่ละครั้งที่ get
- ต้องเรียก m ครั้ง (m คือความยาวเวกเตอร์)
- รวมใช้เวลา O(nm)

SparseVector : dot

```
public double dot(SparseVector v2) {
  SparseVector v1 = this;
  double r = 0;
  int i1 = 0, i2 = 0;
  while (i1 < v1.size && i2 < v2.size) {
    Element e1 = v1.elementData[i1];
    Element e2 = v2.elementData[i2];
    if      (e1.index < e2.index) i1++;
    else if (e1.index > e2.index) i2++;
    else {
      r += e1.value * e2.value;
      i1++; i2++;
    }
  }
  return r;
}
```

$O(n)$

```
    v1 = (0,3,0,0,0,4,0)
    v2 = (0,6,3,0,0,8,0)
v1·v2 =   18   +  32 = 50
```

SparseVector : add

```
public SparseVector add(SparseVector v2) {
  SparseVector v1 = this;
  SparseVector v3 = new SparseVector(v1.length());
  for (int i = 0; i<v1.length(); i++)
    v3.set(i, v1.get(i) + v2.get(i));
  return v3;
}
```

```
v1 = (0,3,0,0,1,4,0)
v2 = (0,6,3,0,0,3,0)
v3 = (0,9,3,0,1,7,0)
```

- เสีย O(n) ในแต่ละครั้งที่ get
- set ที่ปลายเวกเตอร์ เสีย O(n)
- ต้องเรียก m ครั้ง (m คือความยาวเวกเตอร์)
- รวมใช้เวลา O(nm)

O(n)

```
public SparseVector add(SparseVector v2) {
  SparseVector v1 = this;
  SparseVector v3 = new SparseVector(v1.length());
  int i1 = 0, i2 = 0, i3 = 0;
  while (i1 < v1.size && i2 < v2.size) {
    Element e1 = v1.elementData[i1], e2 = v2.elementData[i2];
    if (e1.index < e2.index)
      {v3.add(i3++, e1.index, e1.value); i1++;}
    else if (e1.index > e2.index)
      {v3.add(i3++, e2.index, e2.value); i2++;}
    else
      {v3.add(i3++, e1.index, e1.value+e2.value); i1++;i2++;}
  }
  while (i1 < v1.size) {
    Element e1 = elementData[i1++];
    v3.add(i3++, e1.index, e1.value);
  }
  while (i2 < v2.size) {
    Element e2 = elementData[i2++];
    v3.add(i3++, e2.index, e2.value);
  }
  return v3;
}
```

สรุป

- รายการคือที่เก็บข้อมูลที่ข้อมูลแต่ละตัวมีอันดับ
- สร้าง ArrayList คล้ายกับ ArrayCollection
- สร้าง SinglyLinkedList คล้ายกับ LinkedCollection
  - โยงเดี่ยว ไม่วน มีปมหัว
- สร้าง LinkedList
  - โยงคู่ วน มีปมหัว
- การใช้รายการเพื่อสร้างเวกเตอร์มากเลขศูนย์

กองซ้อน

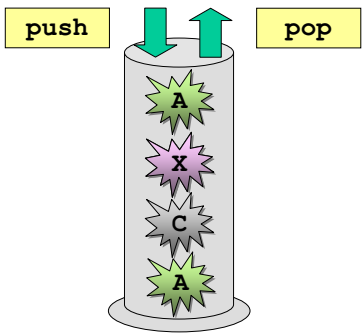
(Stack)

หัวข้อ

- นิยามกองซ้อน และอินเตอร์เฟส Stack
- การสร้างกองซ้อนด้วยอาเรย์
- ตัวอย่างการใช้งานกองซ้อน
  - การตรวจการเขียนวงเล็บเปิดปิด
  - การใช้กองซ้อนใน java virtual machine
  - การคำนวณค่าของนิพจน์เติมหลัง (postfix)
  - การเปลี่ยนนิพจน์เติมกลาง (infix) เป็นเติมหลัง

การเพิ่ม/ลบข้อมูลในกองซ้อน

- ข้อมูล เข้าหลัง ออกก่อน (Last-In First-Out)



กองซ้อน : stack

```
public interface Stack {
  public boolean isEmpty();
  public int size();
  public void push(Object e);
  public Object peek();
  public Object pop();
}
```

A B C * * D E * * *

C B E D A

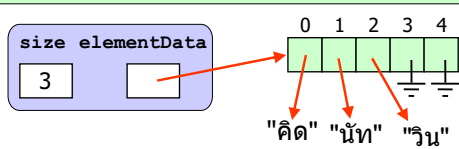
ArrayStack : สร้าง Stack ด้วยอาเรย์

- คล้าย ArrayCollection
- เพิ่มกับลบเฉพาะที่ด้านท้าย
- ขยายขนาดของอาเรย์ได้ เมื่อเต็ม

Stack s;	size	elementData
s = new ArrayStack(1);	0	
s.push("A");	1	A
s.push("B");	2	A B
s.push("C");	3	A B C
s.pop();	2	A B

ArrayStack : โครงสร้าง

```
public class ArrayStack implements Stack {
  private Object[] elementData;
  private int size;

  public ArrayStack(int cap) {
    elementData = new Object[cap];
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public int size() {
    return size;
  }
```



ArrayStack : push

```
public void push(Object e) {
  if (size == elementData.length) {
    Object[] a = new Object[2*size];
    for(int i=0; i<size; i++)
      a[i] = elementData[i];
    elementData = a;
  }
  elementData[size++] = e;
}
```

ข้อมูลเต็มอาเรย์ ก็ขยายขนาดอาเรย์เป็นสองเท่า

ถ้าไม่ต้องขยายขนาด : $\Theta(1)$
ถ้าต้องขยายขนาด : $O(n)$

ArrayStack : peek, pop

```
public Object peek() {
  if (isEmpty())
    throw new NoSuchElementException ();
  return elementData[size-1];
}
public Object pop() {
  Object e = peek();
  elementData[--size] = null;
  return e;
}
}
```

$\Theta(1)$

ตัวอย่างการใช้งาน Stack

- การตรวจสอบโครงสร้างแบบซ้อนกัน
  เช่น การใส่วงเล็บ ( ) { } [ ] ...
- การจัดเก็บตัวแปรและการเรียกเมท็อดของ jvm
- การประมวลผลนิพจน์ทางคณิตศาสตร์
- การทำ undo/redo
- การค้นคำตอบแบบ depth-first search
- ...

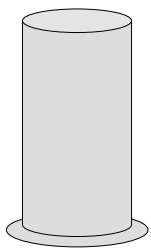
การตรวจสอบการใส่วงเล็บ

- ถูก : **({ ( ) [{ }] })**
- ผิด : เปิดปิดไม่ตรงกัน (**{]**)
- ผิด : มีปิดมากไป ({ ( ) }) **) }**
- ผิด : มีเปิดมากไป **(** { ( ) }
- วิธีทำ
  - อ่านมาทีละตัว
  - ถ้าเป็นวงเล็บเปิด ให้ push ลง stack
  - ถ้าเป็นวงเล็บปิด ให้ pop จาก stack มาตรวจสอบว่าเป็นวงเล็บเปิดที่ตรงกันวงเล็บปิดที่พบหรือไม่
  - เมื่อใดอยาก pop ถ้า isEmpty แสดงว่า ปิดมีมากไป
  - เมื่ออ่านเสร็จหมด stack ยังมีข้อมูล แสดงว่า เปิดมีมากไป

ตัวอย่าง ๑

- ({ ( ) [{ }] })

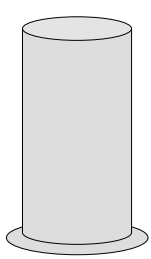
ถูกต้อง



ตัวอย่าง ๒

- ({ ( ) [{)] })

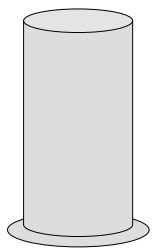
วงเล็บปิดไม่ตรงกับเปิดผิด !!



ตัวอย่าง ๓

- ({ ( ) })]

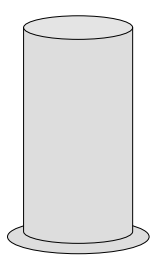
ยังไม่หมด
แต่ stack ว่าง
ผิด !!



ตัวอย่าง ๔

- ({ ( ) [{ }

หมดแล้ว
แต่ stack ไม่ว่าง
ผิด !!



โปรแกรมตรวจสอบการใส่วงเล็บ

```
public static boolean checkParentheses(String t) {
  String open = "{([", close = "})]";
  Stack s = new ArrayStack(100);
  for (int i = 0; i < t.length(); i++) {
    String token = t.substring(i, i + 1);
    if (open.indexOf(token) >= 0) {
      s.push(token);
    } else {
      int k = close.indexOf(token);
      if (k >= 0)
        if (s.isEmpty() ||
            !open.substring(k, k + 1).equals(s.pop()))
          return false;
    }
  }
  return s.isEmpty();
}
```

a.indexOf(b) คืนตำแหน่งในสตริง a ที่มี b ปรากฏอยู่ ถ้าหาไม่พบคืน –1

ผิด ถ้ากองซ้อนว่าง หรือวงเล็บปิดไม่ตรงกับเปิด

ผิด ถ้ากองซ้อนไม่ว่าง

การใช้กองซ้อนภายใน java virtual machine

- ตัว jvm ใช้กองซ้อน (java stack) ในการเก็บ
  - สถานะของการเรียกเมท็อด
  - พารามิเตอร์ และ local variables
  - ที่เก็บชั่วคราวเพื่อการคำนวณ (operand stack)

```
public class Jeng3 {
  public static void main(String[] a) {
    main(args);
  }
}
```

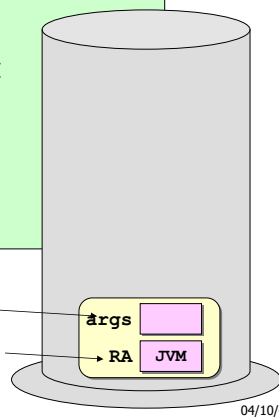
```
Exception in thread "main" java.lang.StackOverflowError
        at Jeng3.main(Jeng3.java:3)
```

Java Stack

```
01:public class StackFrame {
02:  public static void main(String[] args) {
03:    a(3, 2);
04:    b(5);
05:  }
06:  static void a(int x, int y) {
07:    int z = x/y;
08:    b(z);
09:  }
10:  static void b(int x) {
11:    ++x;
12:  }
13:}
```

กรอบกองซ้อน (Stack Frame)

parameters & local variables
return address



นิพจน์ Infix และ Postfix

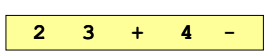
- infix (เติมกลาง)
  - a + b * c / d – 2, (a + b) * c / (d – 2)
  - ต้องกำหนดลำดับการทำก่อนหลังของ operators
  - ใช้วงเล็บช่วย
- postfix (เติมหลัง)
  - a b c * d / + 2 – , a b + c * d 2 - /
  - ลำดับการทำงานของ operators คือจาก ซ้ายไปขวา
  - ไม่จำเป็นต้องมีวงเล็บ

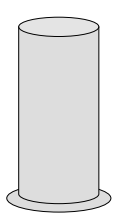
```
1   2   +   3   *   4   1   -   /
        3   3   *   4   1   -   /
                9   4   1   -   /
                9           3   /
                                3
```

การใช้กองซ้อนคำนวณค่าของนิพจน์เติมหลัง

- 2 3 + 4 5 – 6 * + มีค่าเท่าไร ?
- สามารถใช้ stack ช่วยหาค่าของนิพจน์ postfix
- วิธีทำ
  - ดูทีละตัวใน postfix จากซ้ายไปขวา
  - ถ้าเป็น operand ให้ push ของ stack
  - ถ้าเป็น operator ให้ pop operands จาก stack ตามที่ operator ต้องการมาประมวลผล แล้ว push ผลลัพธ์
  - ทำเสร็จ คำตอบจะอยู่ที่ top of stack

ตัวอย่าง





ผลลัพธ์อยู่บนกองซ้อน

การใช้กองซ้อนช่วยแปลง infix เป็น postfix

- input : infix expression
- output : postfix expression
- ขั้นตอนการทำงาน
  - ดูแต่ละตัวใน infix
  - ถ้าเป็น operand นำไปต่อท้าย output
  - ถ้าเป็น operator
    - อาจ pop operator ออกไปต่อท้าย output
    - push operator ลงกองซ้อน
  - เมื่อดู infix ครบตัวแล้ว
    - ให้ pop operators ทุกตัวออกไปต่อท้าย output

เก็บใน list

โครงของโปรแกรม

```
public static List infix2Postfix(List infix) {
  List postfix = new ArrayList(infix.size());
  Stack s = new ArrayStack(infix.size());
  for (int i = 0; i < infix.size(); ++i) {
    String token = (String) infix.get(i);
    if (!isOperator(token)) {
      postfix.add(token);
    } else {
      while (  ???  ) {
        postfix.add(s.pop());
      }
      s.push(token);
    }
  }
  while (!s.isEmpty()) postfix.add(s.pop());
  return postfix;
}
```

ดูทีละตัว

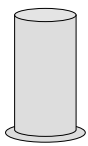
ถ้าเป็น operand, เพิ่มต่อในผลลัพธ์

ถ้าเป็น operator
อาจ pop operators ใน stack ออกเป็นผลลัพธ์
ตามด้วยการ push operator ตัวใหม่

ความสำคัญของ operators

output

input | 2 | * | 3 | + | 4 |



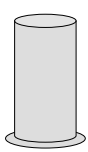
* มาก่อน และสำคัญกว่า +

pop ออก ถ้า operator บนกองซ้อนสำคัญกว่า operator ใหม่

ความสำคัญของ operators

output

input | 2 | + | 3 | - | 4 |



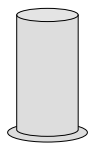
+ มาก่อน
และสำคัญเท่ากับ -

pop ออก ถ้า operator บนกองซ้อน
สำคัญไม่น้อยกว่า operator ใหม่

ความสำคัญของ operators

output

input | 2 | + | 3 | * | 4 |



+ มาก่อน แต่ * สำคัญกว่า +

push ทับ ถ้า operator ใหม่สำคัญกว่า operator บนกองซ้อน

การเปรียบเทียบความสำคัญของ operators

```
public static List infix2Postfix(List infix) {
  List postfix = new ArrayList(infix.size());
  Stack s = new ArrayStack(infix.size());
  for (int i = 0; i < infix.size(); ++i) {
    String token = (String) infix.get(i);
    if (!isOperator(token)) {
      postfix.add(token);
    } else {
      int p = priority(token);
      while (!s.isEmpty() && priority((String)s.peek())>= p) {
        postfix.add(s.pop());
      }
      s.push(token);
    }
  }
  while (!s.isEmpty()) postfix.add(s.pop());
  return postfix;
}
```

หาค่าความสำคัญของ operator ตัวใหม่ที่พบ

pop ออก ถ้า operator บนกองซ้อน สำคัญไม่น้อยกว่า operator ใหม่

การหาความสำคัญของ operators

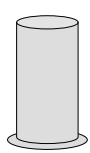
- ให้ ^ แทนการยกกำลัง 7
- ^ ทำก่อน + – * /
- * กับ / ทำก่อน + กับ –
- * มีความสำคัญเท่ากับ / 5
- + มีความสำคัญเท่ากับ – 3

```
private static String operators = "+-*/^";
private static int[] priority =  {3,3,5,5,7};
private static boolean isOperator(String x) {
  return operators.indexOf(x) >= 0;
}
private static int priority(String x) {
  return priority[operators.indexOf(x)];
}
```


ซับซ้อนขึ้นเมื่อมีวงเล็บใน infix

output

input | 2 | * | (| 3 | + | 4 |) | ...



- ภายในวงเล็บเสมือนเป็นนิพจน์ย่อย
- พบวงเล็บเปิด push เสมอ (มีความสำคัญมาก)
- วงเล็บเปิดในกองซ้อน มีความสำคัญน้อยมาก
- พบวงเล็บปิด ลุย pop จนกว่าจะพบวงเล็บเปิด

ความสำคัญของ operators กรณีมีวงเล็บ

- ขณะพบวงเล็บเปิด
  - มีความสำคัญมาก, จึง push (เสมอ
- แต่พอวงเล็บเปิดอยู่ในกองซ้อน
  - มีความสำคัญน้อยสุด, เพื่อให้ตัวอื่น push ทับ, ยกเว้น)

```
int p = priority(token);
while (!s.isEmpty() && priority((String)s.peek())>= p) {
  postfix.add(s.pop());
}
s.push(token);
```

```
int p = outPriority(token);
while (!s.isEmpty() && inPriority((String)s.peek())>= p) {
  postfix.add(s.pop());
}
if (token.equals(")")) s.pop(); else s.push(token);
```


ความสำคัญของ operators กรณีมีวงเล็บ

	+	–	*	/	^	(	)
ขณะที่พบในนิพจน์ (อยู่นอกกองซ้อน)	3	3	5	5	7	9	1
ขณะอยู่ในกองซ้อน	3	3	5	5	7	0	

```
private static String operators = "+-*/^()";
private static int[] outPriority =  {3,3,5,5,7,9,1};
private static int[] inPriority  =  {3,3,5,5,7,0};
private static int outPriority(String x) {
  return outPriority[operators.indexOf(x)];
}
private static int inPriority(String x) {
  return inPriority[operators.indexOf(x)];
}
```

operator ที่ทำจากซ้ายไปขวา

	+	–	*	/	^	(	)
ขณะที่พบในนิพจน์ (อยู่นอกกองซ้อน)	2	2	4	4	6	9	1
ขณะอยู่ในกองซ้อน	3	3	5	5	7	0	

2 + 3 + 4 + 5

2 3 + 4 + 5 +

```
int p = outPriority(token);
while (!s.isEmpty() && inPriority((String)s.peek())>= p) {
  postfix.add(s.pop());
}
if (token.equals(")")) s.pop(); else s.push(token);
```


operator ที่ทำจากขวาไปซ้าย

	+	–	*	/	^	(	)
ขณะที่พบในนิพจน์ (อยู่นอกกองซ้อน)	3	3	5	5	8	9	1
ขณะอยู่ในกองซ้อน	3	3	5	5	7	0	

2 ^ 3 ^ 4 ^ 5

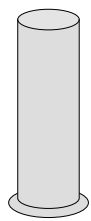
2 3 4 5 ^ ^ ^

```
int p = outPriority(token);
while (!s.isEmpty() && inPriority((String)s.peek())>= p) {
  postfix.add(s.pop());
}
if (token.equals(")")) s.pop(); else s.push(token);
```

ตัวอย่าง

output

input | 2 | + | 3 | - | 4 | ^ | 5 | ^ | 6 |



สรุป

- ประยุกต์กองซ้อนในการแก้ปัญหาหลากหลาย
- การดำเนินการหลัก : push / pop / peek
- สร้างกองซ้อนได้ง่ายด้วยอาเรย์
- ถ้าจองขนาดให้เพียงพอ การทำงานทุกครั้งเป็น $\Theta(1)$

แถวคอย

(Queue)

หัวข้อ

- นิยามแถวคอย และอินเตอร์เฟส Queue
- การสร้างแถวคอยด้วยอาเรย์
- ตัวอย่างการใช้งานแถวคอย
  - ที่พักข้อมูล
  - การเรียงลำดับแบบฐาน
  - การค้นคำตอบตามแนวกว้าง
  - การหาวิถีสั้นสุด

การเพิ่ม/ลบข้อมูลในแถวคอย

- ข้อมูล เข้าก่อน ออกก่อน (First-In First-Out)

แถวคอย : Queue

```
public interface Queue {
  public boolean isEmpty();
  public int size();
  public void enqueue(Object e);
  public Object peek();
  public Object dequeue();
}
```

A B C * * D E * * *

A B C D E

Queue คล้าย List

- Queue คือ list ที่เราเพิ่มปลายด้านหนึ่ง และลบที่ปลายอีกด้าน
- สร้าง queue ด้วย list แบบง่าย ๆ

```
public class ArrayListQueue implements Queue {
  private List list = new ArrayList(10);
  public boolean isEmpty() {return list.isEmpty();}
  public int size() {return list.size();}
  public void enqueue(Object e) {list.add(e);}
  public Object peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return list.get(0);
  }
  public Object dequeue() {
    Object e = peek();
    list.remove(0);
    return e;
  }
}
```

enqueue เพิ่มท้าย list ใช้เวลา O(1)
แต่ dequeue ลบหัว list ใช้เวลา O(n)

สร้าง Queue ด้วย LinkedList

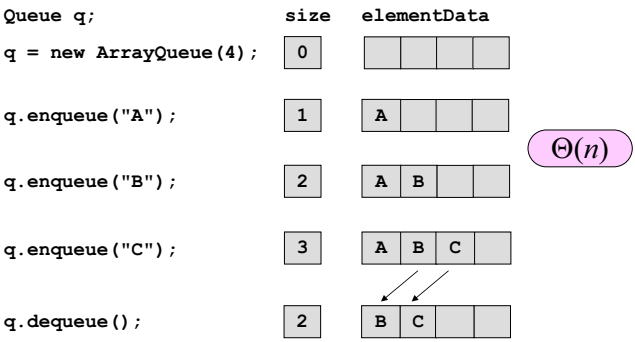
- เลือกใช้ Circular doubly linked list w/ header

```
public class LinkedListQueue implements Queue {
  private List list = new LinkedList(10);
  public boolean isEmpty() {return list.isEmpty();}
  public int size() {return list.size();}
  public void enqueue(Object e) {list.add(e);}
  public Object peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return list.get(0);
  }
  public Object dequeue() {
    Object e = peek();
    list.remove(0);
    return e;
  }
}
```

enqueue เพิ่มท้าย list ใช้เวลา O(1)
dequeue ลบหัว list ใช้เวลา O(1)

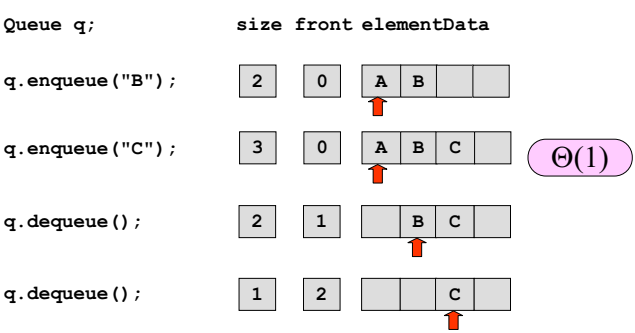
ArrayQueue : สร้าง Queue ด้วยอาเรย์

- เพิ่มที่ท้ายคิว ลบที่หัวคิว
- ให้หัวคิวอยู่ที่ index 0 เสมอ
- ตอนลบต้องใช้เวลา Θ(n)



ArrayQueue : ตำแหน่งหัวคิวเปลี่ยนได้

- จำ index ของหัวคิว
- ลบ : อย่าย้ายข้อมูล แต่ใช้วิธีการเลื่อนตำแหน่งหัวคิว

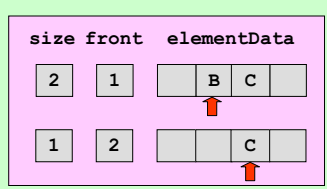


ArrayQueue

```
public class ArrayQueue implements Queue {
  private Object[] elementData;
  private int size;
  private int front;

  public ArrayQueue(int cap) {
    elementData = new Object[cap];
    size = front = 0;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public int size() {
    return size;
  }
  ...
```

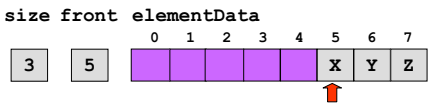

enqueue, peek, dequeue

```
public class ArrayQueue implements Queue {
  private Object[] elementData;
  private int size;
  private int front;
  ...
  public void enqueue(Object e) {
    // ... ขยายอาเรย์ ถ้าเต็ม
    elementData[front + size] = e; size++;
  }
  public Object peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return elementData[front];
  }
  public Object dequeue() {
    Object e = peek();
    elementData[front++] = null;
    size--;
    return e;
  }
  ...
```

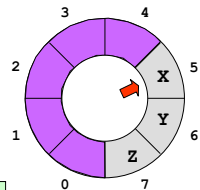


มองอาเรย์เป็นวงวน

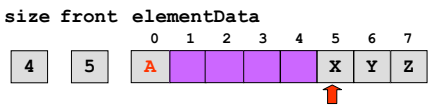
- ถ้าตัวท้ายคิวอยู่ท้ายอาเรย์ เติมตัวใหม่ไม่ได้
- มองอาเรย์ให้เป็นแบบวงวน จะใช้เนื้อที่ได้เต็มที่

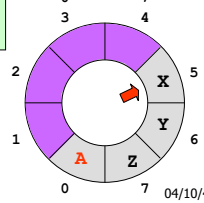



```
q.enqueue("A");
```



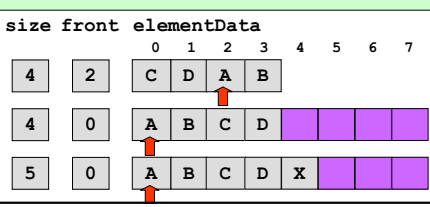

```
b = (front + size) % elementData.length;
elementData[b] = e; size++;
```





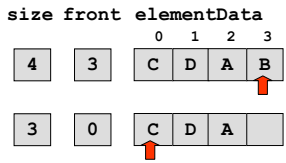
ArrayQueue : enqueue

```
public class ArrayQueue implements Queue {
  ...
  public void enqueue(Object e) {
    if (size == elementData.length) {
      Object[] a = new Object[2 * elementData.length];
      for (int i = 0, j = front; i < size;
           i++, j = (j+1)%elementData.length)
        a[i] = elementData[j];
      front = 0; elementData = a;
    }
    int b = (front + size) % elementData.length;
    elementData[b] = e; size++;
  }
```



ArrayQueue : dequeue

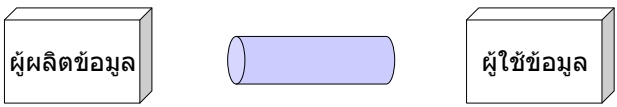
```
public Object dequeue() {
  Object e = peek();
  elementData[front] = null;
  front = (front + 1) % elementData.length;
  size--;
  return e;
}
```



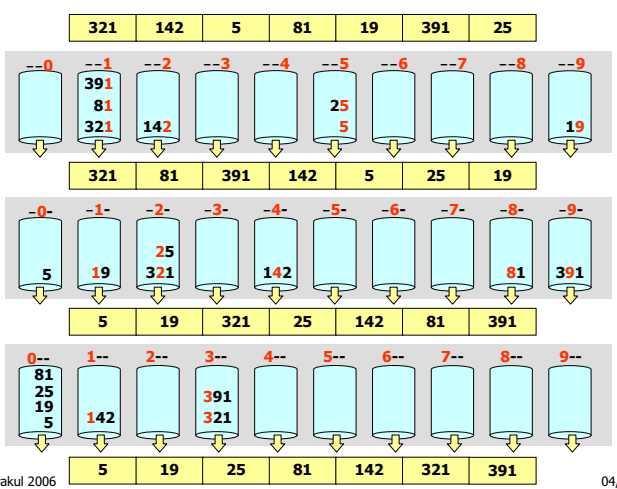
ตัวอย่างการใช้ queue

- เป็นที่พักข้อมูล
- การเรียงลำดับแบบฐาน
- การค้นคำตอบตามแนวกว้าง
- การหาวิถีสั้นสุด
- ...

การใช้แถวคอยเป็นที่พักข้อมูล



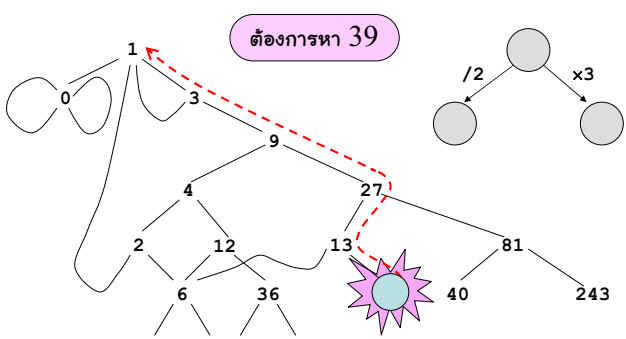
Radix Sort



ปัญหาคูณสามหารสอง

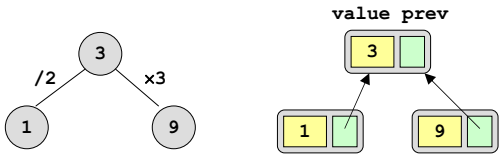
- ให้จำนวนเต็ม v
- เริ่มด้วย 1 จะต้องทำการ x3 และหรือ /2 (ปัดเศษทิ้ง) อย่างไร จึงมีค่าเท่ากับ v
- เช่น
  - v = 10 = 1x3x3x3x3/2/2/2
  - v = 31 = 1x3x3x3x3x3/2/2/2/2/2x3x3/2
- แก้ปัญหานี้อย่างไร ?
- ขอเสนอวิธีลุยทุกรูปแบบ

การค้นตามแนวกว้าง



$39 = 1 \times 3 \times 3 \times 3 / 2 \times 3$

ปมต่าง ๆ ระหว่างการค้น

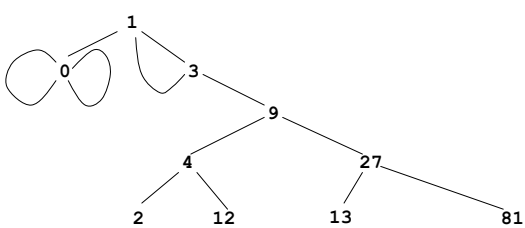



```
class Node {
  int value;
  Node prev;
  Node(int v, Node p) {
    this.value = v;
    this.prev = p;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Node)) return false;
    return this.value == ((Node) o).value;
  }
}
```

สอง nodes เท่ากันก็เมื่อ values ทั้งสองเท่ากัน

ใช้ Set และ Queue

- ใช้ queue เก็บเฉพาะ nodes ที่ยังไม่แตกกิ่ง
- ใช้ set เก็บทุก nodes ที่เคยผลิต

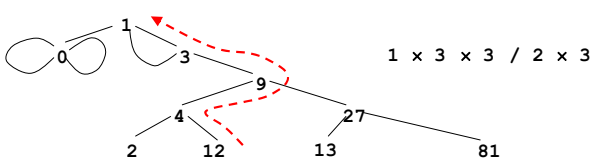


ตัวโปรแกรม

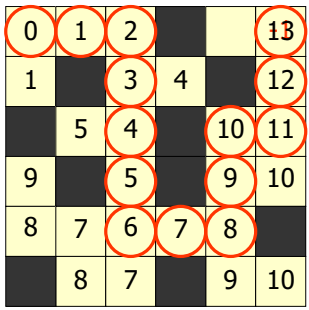
```
public static void bfsM3D2(int target) {
  Set set = new ArraySet(100);
  Queue q = new ArrayQueue(100);
  Node v = new Node(1,null);  // เริ่มด้วย 1
  q.enqueue(v); set.add(v);
  while( !q.isEmpty() ) {
    v = (Node) q.dequeue();
    if (v.value == target) break;
    Node v1 = new Node(v.value/2, v);  // ลองหารสอง (ปัดเศษ)
    Node v2 = new Node(v.value*3, v);  // ลองคูณสาม
    if (!set.contains(v1)) {q.enqueue(v1); set.add(v1);}
    if (!set.contains(v2)) {q.enqueue(v2); set.add(v2);}
  }
  if (v.value == target) showSolution(v);
}
```


การแสดงผลลัพธ์

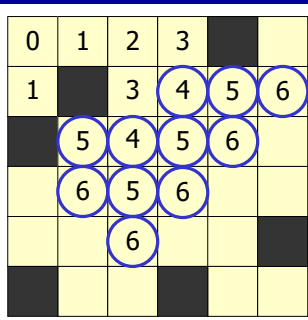
```
static void showSolution(Node v) {
  if (v.prev != null) {
    showSolution(v.prev);
    System.out.print((v.prev.value / 2 == v.value) ?
                     "/ 2" : "x 3");
  } else {
    System.out.print("1 ");
  }
}
```



การหาวิถีสั้นสุด



การใช้แถวคอยในการหาวิถีสั้นสุด



ใช้แถวคอยเก็บตำแหน่งของช่องที่รอขยาย

```
class Pos {
  int row, col;
  Pos(int r, int c) {row = r; col = c;}
}
```

โปรแกรมหาวิถีสั้นสุด

```
static void findPath(int[][] map, Pos source, Pos target) {
  map[source.row][source.col] = 0;   // ต้นทาง
  map[target.row][target.col] = -1;  // ปลายทาง
  Queue q = new ArrayQueue(map.length); q.enqueue(source);
  while (!q.isEmpty()) {
    Pos p = (Pos) q.dequeue();
    if (p.row == target.row && p.col == target.col) break;
    expand(map, q, p.row + 1, p.col, map[p.row][p.col]+1);
    expand(map, q, p.row - 1, p.col, map[p.row][p.col]+1);
    expand(map, q, p.row, p.col + 1, map[p.row][p.col]+1);
    expand(map, q, p.row, p.col - 1, map[p.row][p.col]+1);
  }
}
static void expand(int[][] map, Queue q,int r,int c,int k){
  if (r<0 || r>=map.length ||
      c<0 || c>=map[r].length || map[r][c] != 0) return;
  map[r][c] = k;
  q.enqueue(new Pos(r, c));
}
```


สรุป

- ประยุกต์แถวคอยในการแก้ปัญหาหลากหลาย
- การดำเนินการหลัก : enqueue / dequeue / peek
- สร้างแถวคอยได้ง่ายด้วยอาเรย์ (มองแบบวงวน)
- ถ้าจองขนาดให้เพียงพอ การทำงานทุกครั้งเป็น $\Theta(1)$

แถวคอยเชิงบุริมภาพ

(Priority Queues)

หัวข้อ

- นิยามแถวคอยเชิงบุริมภาพ
- การสร้างแถวคอยเชิงบุริมภาพด้วยฮีปแบบทวิภาค
- ตัวอย่างการใช้งานแถวคอยเชิงบุริมภาพ
  - การเรียงลำดับแบบฮีป
  - การเลือกข้อมูลตามอันดับ
  - รหัสฮัฟฟ์แมน

แถวคอยเชิงบุริมภาพ (Priority Queue)

- เป็นแถวคอยชนิดพิเศษ ที่ "ลัดคิว" ได้
- ข้อมูลมี "ความสำคัญ (priority)" กำกับ
  - ข้อมูลที่มี priority สูงสุด จะลัดคิวไปอยู่ที่หัวคิว
- มีบริการเหมือนของแถวคอย
  - isEmpty, size, peek, enqueue, และ dequeue
- dequeue : ลบข้อมูลที่มีความสำคัญสูงสุด
- peek : ขอข้อมูลที่มีความสำคัญสูงสุด

```
public interface PriorityQueue extends Queue{
  public Object dequeue();  // ลบข้อมูลตัวสำคัญที่สุด
  public Object peek();     // ขอดูข้อมูลตัวสำคัญที่สุด
}
```


ตัวอย่างการใช้งาน


```
PriorityQueue q = new BinaryHeap(10);
q.enqueue(new Integer(5));
q.enqueue(new Integer(3));
q.enqueue(new Integer(99));
q.dequeue();
q.enqueue(new Integer(4));
q.dequeue();
```

สร้าง Priority Queue แบบง่าย (แต่ช้า)

- มีรายการ (แบบ ArrayList) ไว้เก็บข้อมูล
- เรียกใช้บริการต่าง ๆ ของ ArrayList
- เพิ่มข้อมูลใหม่ไว้ท้ายรายการ : $\Theta(1)$
- ต้องวิ่งหาตัวมากสุดเอง ให้กับ peek และ dequeue

```
public class ArrayPQ implements PriorityQueue {
  private ArrayList list = new ArrayList(10);
  public boolean isEmpty() { return list.isEmpty(); }
  public int size() { return list.size(); }
  public void enqueue(Object e) { list.add(e); }
```


สร้าง Priority Queue แบบง่าย (แต่ช้า)

```
public class ArrayPQ implements PriorityQueue {
  ...
  public Object peek() { return list.get(maxIndex()); }
  public Object dequeue() {
    int max = maxIndex();
    Object result = list.get(max);
    list.remove(max);
    return result;
  }
  private int maxIndex() {
    if (isEmpty()) throw new NoSuchElementException();
    int max = 0;
    for (int i = 1; i < list.size(); i++) {
      Comparable d = (Comparable) list.get(i);
      if (d.compareTo(list.get(max)) < 0) max = i;
    }
    return max;
  }
}
```

d < list.get(max)

$\Theta(n)$

จาวา : Comparable

- อ็อบเจกต์ในจาวาที่เปรียบเทียบน้อยกว่ามากกว่าได้ ต้องเป็นออบเจ็กต์ของคลาสแบบ Comparable
- เป็นคลาสที่ implements Comparable

```
public interface Comparable {
  // คืนค่าลบ          ถ้า this น้อยกว่า obj
  // คืนค่า 0          ถ้า this เท่ากับ obj
  // คืนค่าบวก         ถ้า this มากกว่า obj
  public int compareTo(Object obj);
}
```

```
Date d1 = new Date(1998, 4, 12);
Date d2 = new Date(1998, 5, 31);
System.out.println(d1.compareTo(d2));
System.out.println(d2.compareTo(d1));
System.out.println(d1.compareTo(d1));
```

-1

+1

0

ตัวอย่างการเขียนคลาสให้ Comparable

```
public class Rectangle implements Comparable {
  private int width, height;
  ...
  public int compareTo(Object obj) {
    Rectangle that = (Rectangle) obj;
    int thisArea = width * height;
    int thatArea = that.width * that.height;
    return thisArea - thatArea;
  }
}
```

```
Rectangle r1 = new Rectangle(2, 4);
Rectangle r2 = new Rectangle(5, 1);
System.out.println(r1.compareTo(r2));
System.out.println(r2.compareTo(r1));
System.out.println(r1.compareTo(r1));
```

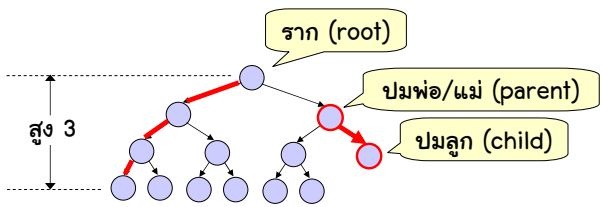
3

-3

0

ฮีปแบบทวิภาค (Binary Heap)

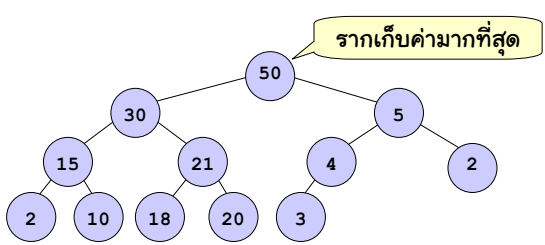
- enqueue : O(log n)
- dequeue : O(log n)
- peek : Θ(1)
- ใช้โครงสร้างแบบ balanced binary tree



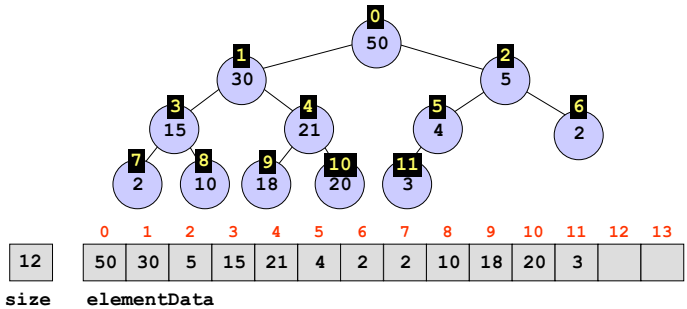
ต้นไม้แบบทวิภาคได้ดุลที่มี n ปม สูง $\lfloor \log_2 n \rfloor$

ฮีปแบบทวิภาค

- Binary Heap
  - มีโครงสร้างเป็น binary tree แบบได้ดุล
    - node เต็มทุกระดับ
    - ระดับล่างสุด เต็มจากซ้ายไปขวา
  - ข้อมูลของ parent node มีค่ามากกว่าของลูก ๆ



การสร้างฮีปแบบทวิภาคด้วยอาเรย์



- รากเก็บที่ index 0
- ลูกซ้ายของ node ที่ index k อยู่ที่ index 2k + 1
- ลูกขวาของ node ที่ index k อยู่ที่ index 2k + 2
- พ่อของ node ที่ index k อยู่ที่ index (k – 1) / 2

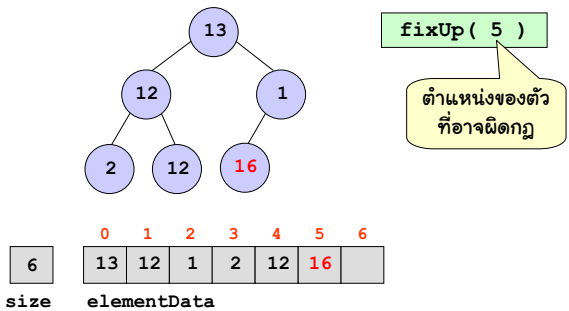
คลาส BinaryHeap

```
public class BinaryHeap implements PriorityQueue {
  private Object[] elementData;
  private int size;

  public BinaryHeap(int cap) {
    elementData = new Object[cap];
  }

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

  public int size() { return size; }

  public Object peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return elementData[0];
  }
  ...
}
```

รากคือตัวมากสุด
เก็บที่ index 0

enqueue(e) : การทำงาน

- นำ e ไปต่อเป็นใบถัดไป (เพิ่มท้ายในอาเรย์)
- สลับ e กับปมพ่อ จนกว่า e จะไม่มากกว่าพ่อ



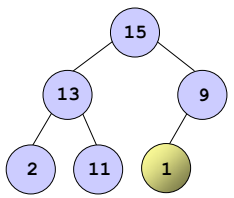
เมท็อด enqueue(e)

```
public void enqueue(Object e) {
  ensureCapacity(size+1);
  elementData[size] = e;
  fixUp(size++);
}
private void fixUp(int k) {
  while (k > 0) {
    int p = (k-1)/2;
    if (!greaterThan(k,p)) break;
    swap(k, p);
    k = p;
  }
}
boolean greaterThan(int i, int j) {
  Comparable e = (Comparable) elementData[i];
  return e.compareTo(elementData[j]) > 0;
}
void swap(int i, int j) {
  Object t = elementData[i];
  elementData[i] = elementData[j];
  elementData[j] = t;
}
```

สลับข้อมูลกับปมพ่อขึ้นไปเรื่อย ๆ จนกว่าจะไม่สำคัญกว่าของพ่อ

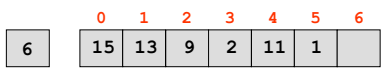
dequeue( ) : การทำงาน

- เก็บรากไว้เป็นคำตอบ
- ย้ายข้อมูลที่ใบล่างขวาสุด มาเก็บที่ราก
- สลับข้อมูลที่รากลงมา จนกว่าพ่อจะไม่น้อยกว่าลูก



`fixDown( 0 )`

ตำแหน่งของตัวที่อาจผิดกฎ



`size` `elementData`

เมท็อด dequeue( )

```
public Object dequeue() {
  Object max = peek();
  elementData[0] = elementData[--size];
  elementData[size] = null;
  if (size > 1) fixDown(0);
  return max;
}
private void fixDown(int k) {
  int c;
  while ((c = 2 * k + 1) < size) {
    if (c+1 < size && greaterThan(c+1, c)) c++;
    if (!greaterThan(c, k)) break;
    swap(k, c);
    k = c;
  }
}
```

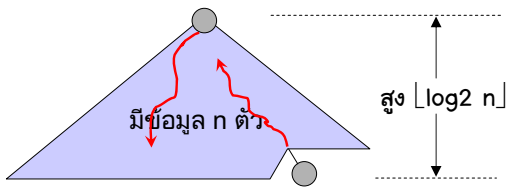
ตราบเท่าที่ยังมีลูกซ้าย

ถ้ามีลูกขวาและขวา > ซ้าย

เลิก เมื่อลูกไม่มากกว่าพ่อ

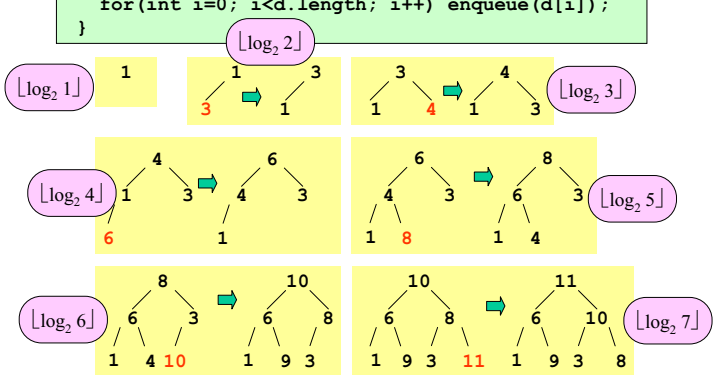
เวลาการทำงาน

- peek : O(1)
- enqueue : fixUp = O(h) = O(log n)
- dequeue : fixDown = O(h) = O(log n)



การสร้างฮีปแบบทวิภาคด้วยการค่อย ๆ เพิ่ม

```
public BinaryHeap(Object[] d) {
  for(int i=0; i<d.length; i++) enqueue(d[i]);
}
```

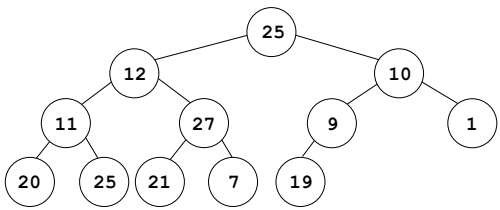


$$\leq \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \lfloor \log_2 3 \rfloor + \ldots + \lfloor \log_2 n \rfloor < \log_2 n! = O(n \log n)$$

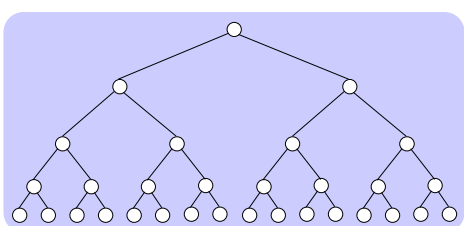
การสร้างฮีปแบบทวิภาคด้วยการค่อย ๆ ปรับ

```
public BinaryHeap(Object[] d) {
  elementData = (Object[]) d.clone();
  size = d.length;
  for(int i=size-1; i>=0; i--) fixDown(i);
}
```

0	1	2	3	4	5	6	7	8	9	10	11
25	12	10	11	27	9	1	20	25	21	7	19



การสร้างฮีปแบบทวิภาคด้วยการค่อย ๆ ปรับ



สูง จำนวนต้น

h

4 × 1

3 × 2

2 × 4

1 × 8

0 × 16

$k \times 2^{h-k}$

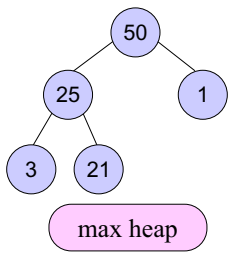
$$\leq \sum_{k=0}^{h} k2^{h-k}$$

fixdown ในต้นไม้สูง k เกิดการสลับข้อมูลไม่เกิน k ครั้ง

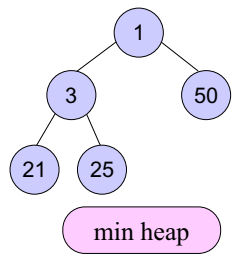
$$\sum_{k=0}^{h} k2^{h-k} = 2^h \sum_{k=0}^{h} k2^{-k} < 2^h \sum_{k=0}^{\infty} k2^{-k} = 2^{h+1} = O(n)$$

ฮีปมากสุด / ฮีปน้อยสุด (Max/Min Heap)

- ฮีปมากสุด
  - ข้อมูลของ parent node มีค่ามากกว่าของลูก ๆ
- ฮีปน้อยสุด
  - ข้อมูลของ parent node มีค่าน้อยกว่าของลูก ๆ



max heap



min heap

คลาส BinaryMinHeap

```
public class BinaryMinHeap   implements PriorityQueue {
  private Object[] elementData;
  private int size;

  public BinaryMinHeap(int cap) {
    elementData = new Object[cap];
  }

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

  public int size() { return size; }

  public Object peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return elementData[0];
  }
  ...
}
```

เปลี่ยนชื่อ

เหมือนเดิม

เหมือนเดิม

เหมือนเดิม

BinaryMinHeap : enqueue(e)

```
public void enqueue(Object e) {
  ensureCapacity(size+1);
  elementData[size] = e;
  fixUp(size++);
}
private void fixUp(int k) {
  while (k > 0) {
    int p = (k-1)/2;
    if (!lessThan(k,p)) break;
    swap(k, p);
    k = p;
  }
}
boolean lessThan(int i, int j) {
  Comparable e = (Comparable) elementData[i];
  return e.compareTo(elementData[j]) < 0;
}
void swap(int i, int j) {
  Object t = elementData[i];
  elementData[i] = elementData[j]
  elementData[j] = t;
}
```

ปมลูกไม่น้อยกว่าปมพ่อ

เปลี่ยนชื่อ

น้อยกว่า

BinaryMinHeap : dequeue( )

```
public Object dequeue() {
  Object max = peek();
  elementData[0] = elementData[--size];
  elementData[size] = null;
  if (size > 1) fixDown(0);
  return max;
}
private void fixDown(int k) {
  int c;
  while ((c = 2 * k + 1) < size) {
    if (c+1 < size && lessThan(c+1, c)) c++;
    if (!lessThan(c, k)) break;
    swap(elementData, k, c);
    k = c;
  }
}
```

เปลี่ยนมากกว่าเป็นน้อยกว่า

BinaryMinHeap : เขียนอีกแบบ

- ให้ BinaryMinHeap เป็นคลาสลูกของ BinaryHeap
- เปลี่ยน greaterThan ของคลาสลูก ให้เปรียบเทียบกลับรูปแบบจากที่ควรเป็น

```
public class BinaryMinHeap extends BinaryHeap {
  public BinaryMinHeap(int cap) {
    super(cap);
  }
  boolean greaterThan(int i, int j) {
    Comparable e = (Comparable) elementData[i];
    return e.compareTo(elementData[j]) < 0;
  }
}
```

ชื่อ greaterThan แต่เทียบน้อยกว่า

ตัวอย่างการใช้งาน

- การเรียงลำดับแบบฮีป (heap sort)
- การเลือกข้อมูลตามอันดับ
- รหัสฮัฟฟ์แมน
- การจำลองตามเหตุการณ์
- กลวิธีการแก้ไขปัญหาแบบ greedy
- ...

การเรียงลำดับแบบฮีป (Heap Sort)

- รับอาเรย์มาสร้างให้เป็นฮีปมากสุด
- เข้าวงวน dequeue ข้อมูล (ตัวมากสุด) เพื่อนำไปไว้ ณ ตำแหน่งหลังสุดของกลุ่ม

0	1	2	3	4
25	12	10	11	27

การเรียงลำดับแบบฮีป : โปรแกรม

```
public class BinaryHeap implements PriorityQueue {
  private Object[] elementData;
  private int size;
  ...
  public static void heapSort(Object[] data) {
    BinaryHeap h = new BinaryHeap(0);
    h.elementData = data;
    h.size = data.length;
    for (int k = h.size - 1; k >= 0; k--) {
      h.fixDown(k);
    }
    for (int k = h.size - 1; k > 0; k--) {
      data[k] = h.dequeue();
    }
  }
}
```

$O(n)$

$O(n \log n)$

การเลือกข้อมูลตามอันดับ

- รับอาเรย์ของข้อมูลจำนวน n ตัว
- ข้อมูลตัวที่น้อยสุดเป็นอันดับที่ k คือตัวใด ?
- ตัวอย่าง : ⟨ 2, 4, 1, 7, 9, 5, 6, 8, 10 ⟩ k = 4
- ⟨ 1, 2, 4, **5**, 6, 7, 8, 9, 10 ⟩
- วิธีที่ 1 : เรียงลำดับ, ตอบ a[k-1]

```
public static Object select(Object[] a, int k) {
  BinaryHeap.heapSort(a);
  return a[k-1];
}
```

$O(n \log n)$ $\Theta(1)$

วิธีที่ 2 : สร้างฮีปแล้วลบ k ครั้ง

- เปลี่ยนเป็นฮีปน้อยสุด
- ลบออกมา k ครั้ง
  ครั้งที่ k ก็คือตัวน้อยสุดอันดับ k

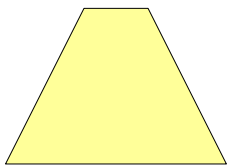
$O(n)$

$O(k \log n)$

```
public static Object select(Object[] a, int k) {
  PrioriyQueue h = new BinaryMinHeap(a);
  for (int i=0; i<k-1; i++) {
    h.dequeue();
  }
  return h.dequeue();
}
```

วิธีที่ 3 : ใช้ฮีปมากสุดขนาด k

7	13	21	5	9	11	44	6



k = 4

วิธีที่ 3 : ใช้ฮีปมากสุดขนาด k

```
public static Object select(Object[] a,  int k) {
  PrioriyQueue h = new BinaryMaxHeap(k);
  int j = 0;
  for (; j<k; j++) h.enqueue(a[j]);
  for (; j<a.length; j++) {
    Comparable e = (Comparable) a[j];
    if (e.compareTo(h.peek()) < 0) {
      h.dequeue();
      h.enqueue(e);
    }
  }
  return h.peek();
}
```

$O(n \log k)$

ฮีปขนาด k : การเพิ่ม/ลบใช้เวลา O(log k)

วิธีที่ 3 : ใช้กับ Iterator

```
public static Object select(Iterator i, int k) {
  PrioriyQueue h = new BinaryMaxHeap(k);
  int j = 0;
  for (; j<k; j++) h.enqueue(a[j]);
  for (; j<i.hasNext(); j++) {
    Comparable e = (Comparable) i.next();
    if (e.compareTo(h.peek) < 0) {
      h.dequeue();
      h.enqueue(e);
    }
  }
  return h.peek();
}
```

ใช้เนื้อที่เสริม k ช่อง

ในจาวา **Iterator** คืออ็อบเจกต์ที่ใช้แจกแจงข้อมูล
จากที่เก็บข้อมูลมีเมท็อด **hasNext** และ **next**

รหัสฮัฟฟ์แมน (Huffman Code)

	'ช'	'ม'	'ย'	'ป'	'ส'	'า'
จำนวน	40	21	15	14	8	2
รหัสแบบความยาวคงที่	000	001	010	011	100	101
รหัสแบบความยาวแปรได้	0	100	101	110	1110	1111

100010000101010001101

ส ม ช า ย ม า

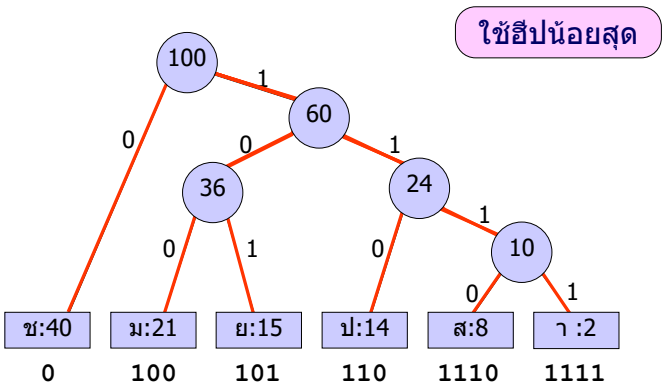
11101000111111011001111

ส ม ช า ย ม า

40x3 + 21x3 + 15x3 + 14x3 + 8x3 + 2x3 = 300

40x1 + 21x3 + 15x3 + 14x3 + 8x4 + 2x4 = 230

วิธีการหารหัสฮัฟฟ์แมน



รหัสฮัฟฟ์แมน

สรุป

- แถวคอยเชิงบุริมภาพเก็บข้อมูลตามความสำคัญ
- ได้รับการนำไปใช้งานมากมาย
- สร้างได้ด้วยฮีปแบบทวิภาค
- ให้บริการเพิ่ม/ลบข้อมูลในเวลา O(log n)

ต้นไม้แบบทวิภาค

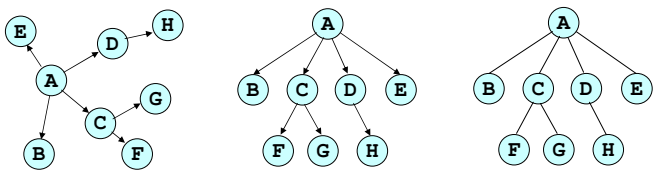
(Binary Trees)

หัวข้อ

- นิยามต้นไม้
- การสร้างต้นไม้
- ต้นไม้แบบทวิภาค
  - ต้นไม้รหัสฮัฟฟ์แมน
  - ต้นไม้นิพจน์
  - การแวะผ่าน
  - การคำนวณค่าของต้นไม้นิพจน์
  - การหาอนุพันธ์ของฟังก์ชันตัวแปรเดียว

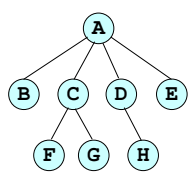
ต้นไม้ (Tree)

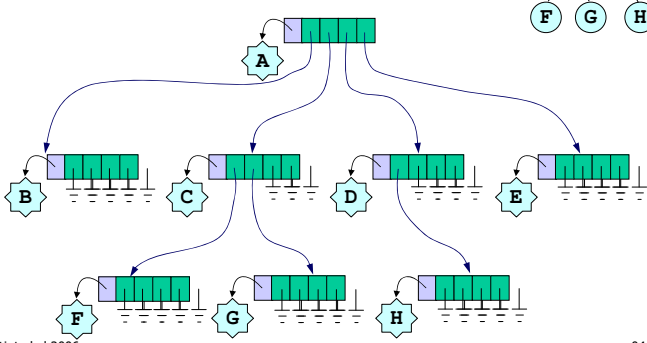
- ต้นไม้ประกอบด้วยปม (nodes) กับเส้นเชื่อม (edges)
- เส้นเชื่อมมีทิศทาง
- A เป็นปมพ่อของ B เมื่อมีเส้นเชื่อมจาก A ไปยัง B
- แต่ละปมมีปมพ่อได้เพียงปมเดียว (ยกเว้นปมพิเศษคือรากไม่มีพ่อ)
- ต้นไม้ที่มีปม v ปม ย่อมมี v – 1 เส้นเชื่อม



การสร้างต้นไม้ : ใช้อาเรย์เก็บลูก ๆ

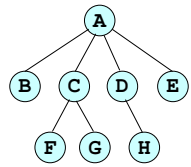
- ถ้ารู้จำนวนลูกมากสุดต่อปม
- ตัวเชื่อมส่วนมากเป็น null

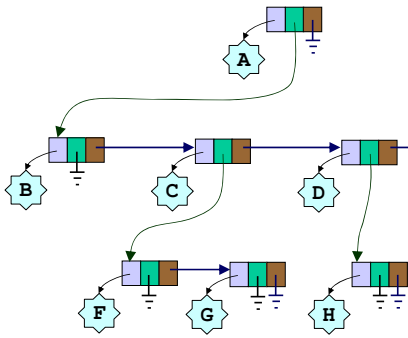




การสร้างต้นไม้ : ใช้รายการเก็บลูก ๆ

- มีตัวเชื่อมไปยังลูกคนโต
- มีตัวเชื่อมไปยังน้องคนถัดไป

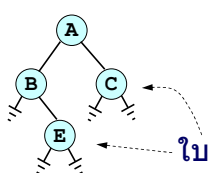


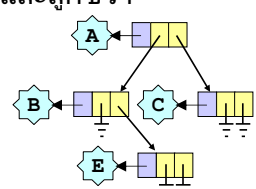



```
class Node {
  Object element;
  Node leftChild;
  Node nextSibling;
}
```


ต้นไม้แบบทวิภาค (Binary Tree)

- ทุกปมมีสองลูก : ลูกซ้าย และลูกขวา

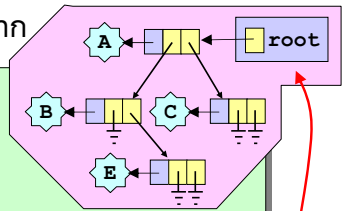





```
class Node {
  Object element;
  Node left, right;
  Node(Object e, Node l, Node r) {
    element = e; left = l; right = r;
  }
  boolean isLeaf() {
    return left == null && right == null;
  }
}
```

การสร้างต้นไม้แบบทวิภาค

- อ็อบเจกต์ต้นไม้เก็บเฉพาะราก




```
public class BinaryTree {
  static class Node {
    Object element;
    Node left;
    Node right;
    Node(Object e, Node l, Node r) {
      element = e; left = l; right = r;
    }
    boolean isLeaf() {
      return left == null && right == null;
    }
  }
  Node root;
  ...
```


รหัสฮัฟฟ์แมน (Huffman Code)

	'ช'	'ม'	'ย'	'ป'	'ส'	'า'
จำนวน	40	21	15	14	8	2
รหัสแบบความยาวคงที่	000	001	010	011	100	101
รหัสแบบความยาวแปรได้	0	100	101	110	1110	1111

10000100010101000 1101

ส ม ช า ย ม า

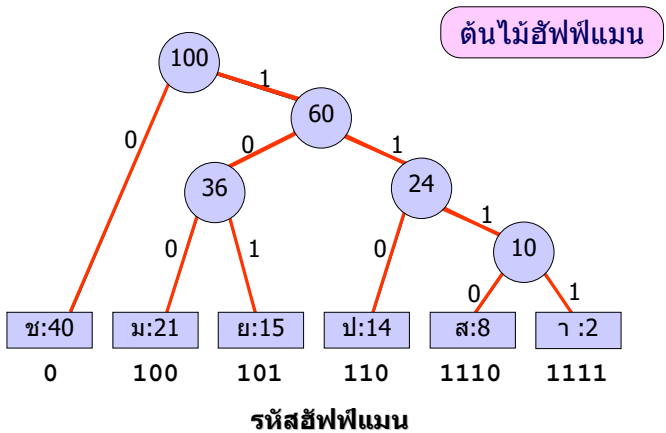
1110100011111011001111

ส ม ช า ย ม า

40x3 + 21x3 + 15x3 + 14x3 + 8x3 + 2x3 = 300

40x1 + 21x3 + 15x3 + 14x3 + 8x4 + 2x4 = 230

วิธีการหารหัสฮัฟฟ์แมน



รหัสฮัฟฟ์แมน

โปรแกรมการหาต้นไม้ฮัฟฟ์แมน

```
public static HuffmanTree coding(int[] freq) {
  BinaryMinHeap h = new BinaryMinHeap();
  for (int i = 0; i < freq.length; i++) {
    h.enqueue(new HuffmanTree(freq[i], null, null));
  }
  for (int i = 0; i < freq.length - 1; i++) {
    HuffmanTree t1 = (HuffmanTree) h.dequeue();
    HuffmanTree t2 = (HuffmanTree) h.dequeue();
    int f = t1.freq() + t2.freq();
    h.enqueue(new HuffmanTree(f, t1.root, t2.root));
  }
  return (HuffmanTree) h.dequeue();
}
```

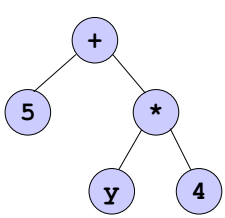
ต้นไม้ฮัฟฟ์แมน (HuffmanTree)

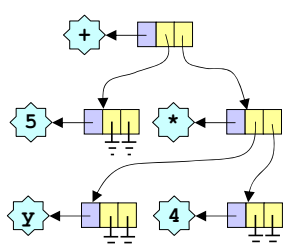
```
public class HuffmanTree extends BinaryTree
                         implements Comparable {
  ...
  public HuffmanTree(int freq, Node left, Node right) {
    root = new Node(new Integer(freq), left, right);
  }
  public int freq() {
    return ((Integer) root.element).intValue();
  }
  public int compareTo(Object obj) {
    return freq() - ((HuffmanTree) obj).freq();
  }
}
```


ต้นไม้นิพจน์ (Expression Tree)

- แทนนิพจน์ได้ด้วยต้นไม้
- ใบ : ตัวถูกดำเนินการ (operand)
- ปม : ตัวดำเนินการ (operator)

```
5 + y * 4
```

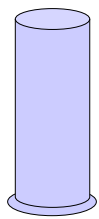




การสร้างต้นไม้นิพจน์

- พบ operand ให้ push ลงกองซ้อน
- พบ operator ให้ pop ออกมาเป็นลูกของปมใหม่ เพื่อ push ลงกองซ้อน

```
infix : 5  +  y  *  4
postfix : 5  y  4  *  +
```



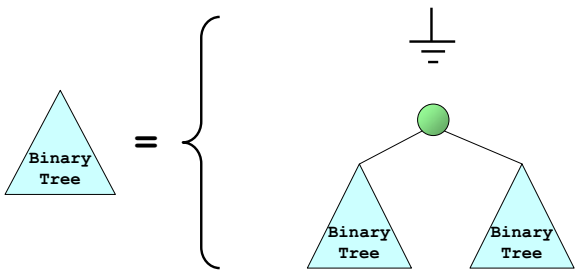
โปรแกรมการสร้างต้นไม้นิพจน์

```
public class Expression extends BinaryTree {
  public Expression(List infix) {
    List postfix = infix2Postfix(infix);
    Stack s = new ArrayStack(10);
    for (int i=0; i<postfix.size(); i++) {
      String token = (String)postfix.get(i);
      if (!isOperator(token)) {
        s.push(new Node(token, null, null));
      } else {
        Node right = (Node) s.pop();
        Node left = (Node) s.pop();
        s.push(new Node(token, left, right));
      }
    }
    root = (Node) s.pop();
  }
  ...
```

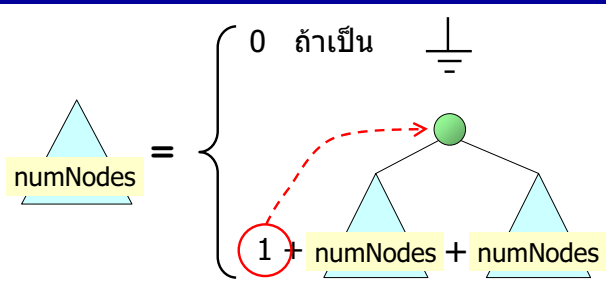
pop 2 ตัวเพราะเป็น binary operaor

มองต้นไม้แบบทวิภาคแบบเวียนเกิด

- Binary tree คือ
  - ต้นไม้ว่าง (null) หรือ
  - หนึ่งปม และลูกต้นซ้ายกับลูกต้นขวา

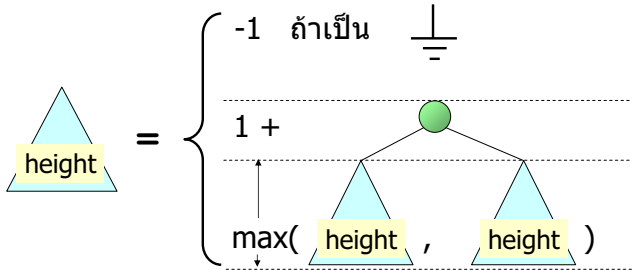


การหาจำนวนปมทั้งหมดของต้นไม้




```
int numNodes(Node node) {
  if ( node == null ) return 0;
  return 1 + numNodes(node.left) +
             numNodes(node.right);
}
```

การหาความสูงของต้นไม้

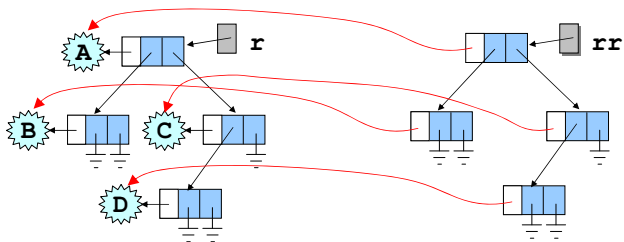



```
int height(Node node) {
  if ( node == null ) return -1;
  return 1 + Math.max(height(node.left),
                      height(node.right));
}
```


คลาส BinaryTree

```
public class BinaryTree {
  static class Node { ... }
  Node root;
  public int numNodes() {
    return numNodes(root);
  }
  public int height() {
    return height(root);
  }
  private int numNodes(Node node) {
    if ( node == null ) return 0;
    return 1 + numNodes(node.left) +
               numNodes(node.right);
  }
  private int height(Node node) {
    if ( node == null ) return -1;
    return 1 + Math.max(height(node.left),
                        height(node.right));
  }
```

การสำเนาต้นไม้




```
Node rr = copy(r);
```

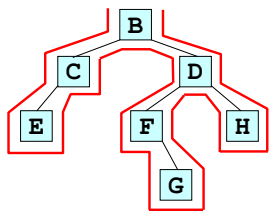

การสำเนาต้นไม้ (มองแบบเวียนเกิด)

```
public class Expression extends BinaryTree {
  ...
  // copy constructor
  public Expression(Expression e) {
    root = copy(e.root);
  }
  ...
}
```

```
Node copy(Node r) {
  if (r == null) return null;
  Node leftTree = copy(r.left);
  Node rightTree = copy(r.right);
  return new Node(r.element, leftTree, rightTree);
}
```

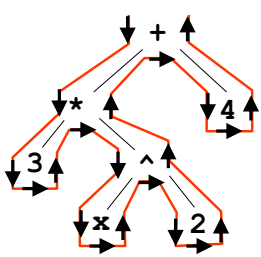
การแวะผ่านต้นไม้

- Tree traversal เป็นกระบวนการเข้าถึงปมต่าง ๆ ในต้นไม้ ปมละหนึ่งครั้งอย่างมีระเบียบ
  - แบบก่อนลำดับ (preorder) B, C, E, D, F, G, H
  - แบบตามลำดับ (inorder) E, C, B, F, G, D, H
  - แบบหลังลำดับ (postorder) E, C, G, F, H, D, B



การแวะผ่านต้นไม้นิพจน์

- แวะผ่านแบบก่อนลำดับ ได้นิพจน์แบบ prefix
- แวะผ่านแบบตามลำดับ ได้นิพจน์แบบ infix
- แวะผ่านแบบหลังลำดับ ได้นิพจน์แบบ postfix



+ * 3 ^ x 2 4

3 * x ^ 2 + 4

3 x 2 ^ * 4 +

การแวะผ่านต้นไม้ที่มี x เป็นราก

- แบบก่อนลำดับ
  - แวะ x
  - แวะผ่าน x.left
  - แวะผ่าน x.right
- แบบตามลำดับ
  - แวะผ่าน x.left
  - แวะ x
  - แวะผ่าน x.right
- แบบหลังลำดับ
  - แวะผ่าน x.left
  - แวะผ่าน x.right
  - แวะ x

```
void preOrder(Node x) {
  if (x == null) return;
  visit(x.element);
  preorder(x.left);
  preorder(x.right);
}
```

```
void inOrder(Node x) {
  if (x == null) return;
  inorder(x.left);
  visit(x.element);
  inorder(x.right);
}
```

```
void postOrder(Node x) {
  if (x == null) return;
  postorder(x.left);
  postorder(x.right);
  visit(x.element);
}
```


toArray( )

- คืนอาเรย์ที่บรรจุข้อมูลตามปมต่าง ๆ ทุกปมในต้นไม้

```
public class BinaryTree {
  Node root;
  ...
  public Object[] toArray() {
    int n = numNodes(root);
    Object[] a = new Object[n];
    toArray(root, a, 0);
    return a;
  }
  private int toArray(Node x, Object[] a, int k) {
    if (x == null) return k;
    a[k++] = x.element;
    k = toArray(x.left, a, k);
    k = toArray(x.right, a, k);
    return k;
  }
```

นำข้อมูลตามปมในต้นไม้ root ไปใส่ในอาเรย์ a เริ่มตั้งแต่ช่อง 0 เป็นต้นไป

มีโครงการทำงานแบบ preorder

เขียน toArray แบบใช้ Visitor

```
public abstract class Visitor {
  public abstract void visit(Object e);
}
```

```
void preOrder(Node r, Visitor v) {
  if (r == null) return;
  v.visit(r.element);
  preOrder(r.left, v);
  preOrder(r.right, v);
}
```

```
public Object[] toArray() {
  final Object[] a = new Object[numNodes()];
  Visitor v = new Visitor() {
                int k = 0;
                public void visit(Object e) {
                  a[k++] = e;
                }
              };
  preOrder(root, v);
  return a;
}
```

anonymous class

สร้างอ็อบเจกต์ของคลาส ที่เป็นคลาสลูกของ Visitor

ปรับปรุงให้ Visitor ยุติการแวะผ่านได้

```
public abstract class Visitor {
  private boolean done = false;
  public void done() {
    done = true;
  }
  public boolean isDone() {
    return done;
  }
  public abstract void visit(Object e);
}
```

```
void preOrder(Node r, Visitor v) {
  if (r == null || v.isDone()) return;
  v.visit(r.element);
  preOrder(r.left, v);
  preOrder(r.right, v);
}
```

การตรวจสอบว่าต้นไม้เก็บ x หรือไม่

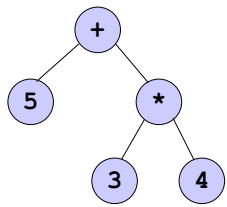
```
public boolean contains(final Object x) {
  Visitor v = new Visitor() {
                public void visit(Object e) {
                  if (e.equals(x)) done();
                }
              };
  preOrder(root, v);
  return v.isDone();
}
```


การแวะผ่านด้วย Visitor

```
public class BinaryTree {
  ...
  public void preOrder(Visitor v)  { preOrder(root, v);  }
  public void inOrder(Visitor v)   { inOrder(root, v);   }
  public void postOrder(Visitor v) { postOrder(root, v); }

  void preOrder(Node x, Visitor v) {
    if (x == null || v.isDone()) return;
    v.visit(x.element);
    preOrder(x.left, v);
    preOrder(x.right, v);
  }
  void inOrder(Node x, Visitor v) {
    if (x == null || v.isDone()) return;
    inOrder(x.left, v);
    v.visit(x.element);
    inOrder(x.right, v);
  }
  void postOrder(Node x, Visitor v) {
    ...
```

การคำนวณค่าของต้นไม้นิพจน์

- ต้องรู้ค่าของลูก ๆ ก่อน จึงคำนวณได้
- มีลักษณะคล้ายการแวะผ่านแบบหลังลำดับ



เมท็อดการคำนวณค่าของต้นไม้นิพจน์

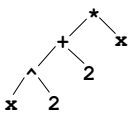
```
public class Expression extends BinaryTree {
  ...
  public double eval() {
    return eval(root);
  }
  private double eval(Node r) {
    if (r == null) return 0;
    if (r.isLeaf())
      return Double.parseDouble((String) r.element);
    double vLeft = eval(r.left);
    double vRight = eval(r.right);
    if (r.element.equals("+")) return vLeft + vRight;
    if (r.element.equals("-")) return vLeft - vRight;
    if (r.element.equals("*")) return vLeft * vRight;
    if (r.element.equals("/")) return vLeft / vRight;
    if (r.element.equals("^")) return Math.pow(vLeft,vRight);
    throw new IllegalStateException();
  }
```

การหาอนุพันธ์ฟังก์ชันตัวแปรเดียว

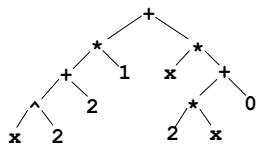
$$f(x) = (x^2 + 2)x$$

$$f'(x) = (x^2 + 2)\cdot 1 + x\cdot(2x + 0)$$

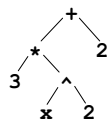
$$= 3x^2 + 2$$



f



f.diff()



f.simplify()

สูตรต่าง ๆ

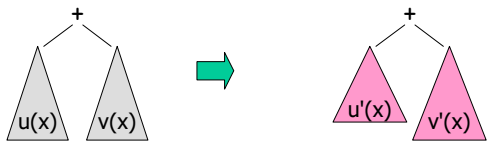
$$\left(u(x) + v(x)\right)' = u'(x) + v'(x)$$

$$\left(u(x)v(x)\right)' = v(x)u'(x) + u(x)v'(x)$$

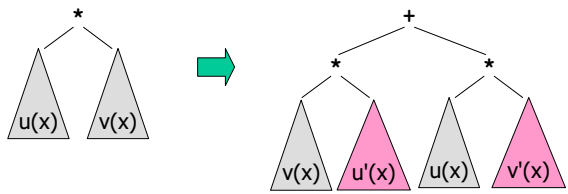
$$\left(\frac{u(x)}{v(x)}\right)' = \frac{v(x)u'(x) - u(x)v'(x)}{\left(v(x)\right)^2}$$

$$\left(\left(u(x)\right)^C\right)' = C\left(u(x)\right)^{C-1}u'(x)$$

อนุพันธ์ของต้นไม้นิพจน์ : +, *

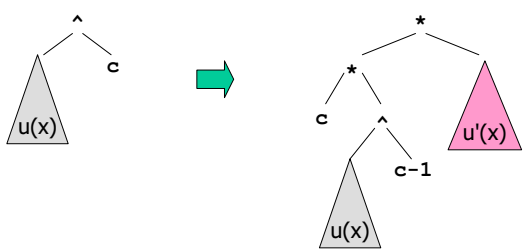


$$\left(u(x)+v(x)\right)' = u'(x)+v'(x)$$



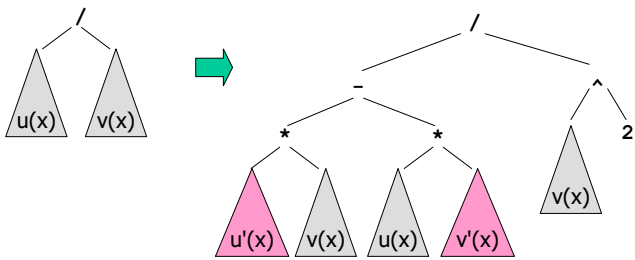
$$\left(u(x)v(x)\right)' = v(x)u'(x)+u(x)v'(x)$$

อนุพันธ์ของต้นไม้นิพจน์ : ^



$$\left(\left(u(x)\right)^C\right)' = C\left(u(x)\right)^{C-1}u'(x)$$

อนุพันธ์ของต้นไม้นิพจน์ : /



$$\left(\frac{u(x)}{v(x)}\right)' = \frac{v(x)u'(x) - u(x)v'(x)}{\left(v(x)\right)^2}$$

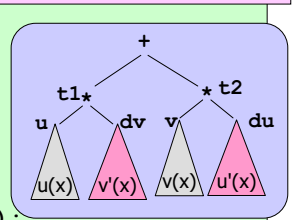
เมท็อดการหาอนุพันธ์

```
public class Expression extends BinaryTree {
  ...
  public void diff() {
    root = diff(root);
  }
  private Node diff(Node r) {
    if (r == null) return null;
    String s = (String) r.element;
    if ( r.isLeaf() ) {
      r.element = (s.equals("x") ? "1" : "0");
    } else {
      if (s.equals("+"))      r = diffSum(r);
      else if (s.equals("-")) r = diffSum(r);
      else if (s.equals("^")) r = diffExpo(r);
      else if (s.equals("*")) r = diffMult(r);
      else if (s.equals("/")) r = diffDiv(r);
    }
    return r;
  }
```

เมท็อดการหาอนุพันธ์ : +, *

```
public class Expression extends BinaryTree {
  ...
  private Node diffSum(Node r) {
    r.left = diff(r.left);
    r.right = diff(r.right);
    return r;
  }
  private Node diffMult(Node r) {
    Node u = copy(r.left);
    Node v = copy(r.right);
    Node du = diff(r.left);
    Node dv = diff(r.right);
    Node t1 = new Node("*", u, dv);
    Node t2 = new Node("*", v, du);
    return new Node("+", t1, t2);
  }
```

$$\left(u(x)+v(x)\right)' = u'(x)+v'(x)$$



$$\left(u(x)v(x)\right)' = v(x)u'(x)+u(x)v'(x)$$

สรุป

- ต้นไม้เป็นโครงสร้างในการจัดเก็บข้อมูล
  - สร้างต้นไม้ได้ด้วยการโยงปมของต้นไม้
- ต้นไม้แบบทวิภาคเป็นต้นไม้ที่แต่ละปมมี 2 ลูก
  - มองต้นไม้ใหญ่ประกอบด้วยต้นไม้ย่อย
  - ทำให้เขียนเมท็อดได้แบบเวียนเกิด
  - การประมวลข้อมูลตามปม กระทำได้ด้วยการแวะผ่านแบบก่อนลำดับ ตามลำดับ หรือหลังลำดับ

ต้นไม้ค้นหาแบบทวิภาค

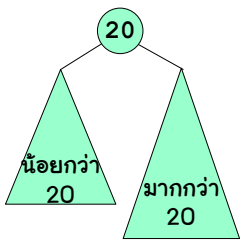
(Binary Search Tree)

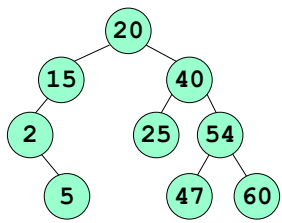
หัวข้อ

- นิยามต้นไม้ค้นหาแบบทวิภาค
- โครงสร้างของต้นไม้ค้นหาแบบทวิภาค
- บริการต่าง ๆ
  - การค้นหาข้อมูล ตัวน้อยสุด ตัวมากสุด
  - การเพิ่ม
  - การลบ
  - การเรียงลำดับข้อมูลโดยใช้ต้นไม้ค้นหาแบบทวิภาค
- การสร้างเซตและคอลเล็กชัน ด้วยต้นไม้ค้นหาแบบทวิภาค

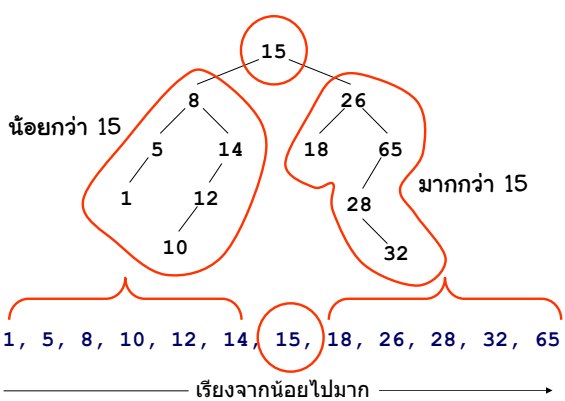
ต้นไม้ค้นหาแบบทวิภาค

- เป็นต้นไม้แบบทวิภาค
- เก็บข้อมูลตามปม
- ข้อมูลในต้นไม้ย่อยทางซ้ายต้องน้อยกว่าข้อมูลที่ราก
- ข้อมูลในต้นไม้ย่อยทางขวาต้องมากกว่าข้อมูลที่ราก
- ต้นไม้ย่อยทุก ๆ ต้นต้องเป็น binary search tree ด้วย

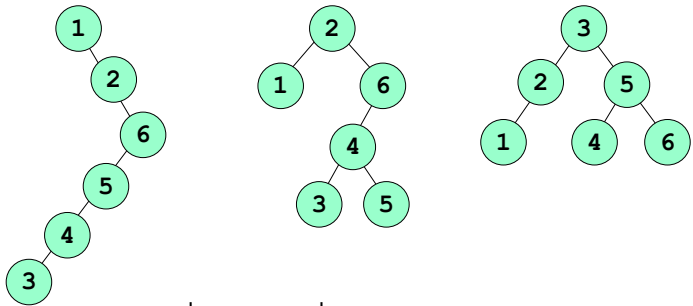




การแวะผ่านแบบตามลำดับ



ข้อมูลชุดเดียวกันเก็บได้หลายแบบ



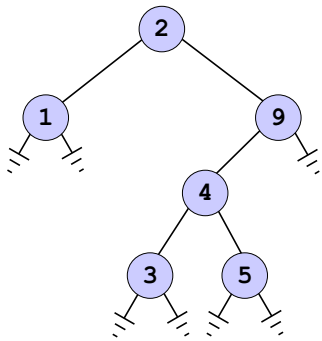
$$\lfloor \log_2 n \rfloor \le h \le n-1$$

class BSTree

```
public class BSTree extends BinaryTree {
  int size;

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

  int compare(Object a, Object b) {
    return ((Comparable)a).compareTo(b);
  }

  public Object get(Object e) {...}
  public Object getMin() {...}
  public Object getMax() {...}
  public void add(Object e) {...}
  public void remove(Object e) {...}
  public static treeSort(Object[] data) {...}
  ...
```

การค้นหาข้อมูล

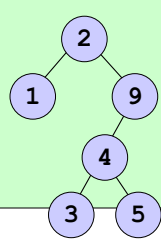
- ใช้การแวะผ่านต้นไม้ ค่อย ๆ เปรียบเทียบ
- ใช้กฎการจัดเก็บช่วยในการค้น



$O(h)$

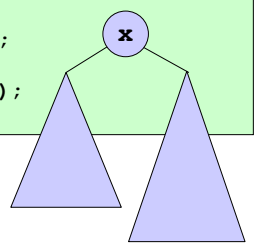
get : การค้นหาข้อมูล

```
public class BSTree extends BinaryTree {
  ...
  public Object get(Object e) {
    Node node = getNode(root, e);
    return node == null ? null : node.element;
  }
  Node getNode(Node r, Object e) {
    while (r != null) {
      int cmp = compare(e, r.element);
      if (cmp == 0) return r;
      if (cmp < 0)
        r = r.left;
      else
        r = r.right;
    }
    return null;
  }
```



การค้นหาข้อมูลแบบเวียนเกิด

```
public class BSTree extends BinaryTree {
  ...
  Node getNode(Node r, Object e) {
    if (r == null) return null;
    int cmp = compare(e, r.element);
    if (cmp == 0) return r;
    if (cmp < 0)
      return getNode(r.left, e);
    else
      return getNode(r.right, e);
  }
```

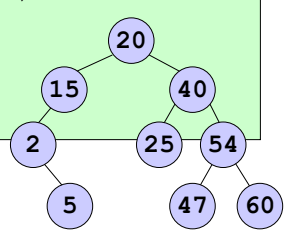


getMin : การค้นหาข้อมูลตัวน้อยสุด

- ปมใดมีลูกทางซ้าย ย่อมมีข้อมูลที่น้อยกว่า
- หาตัวน้อยสุด ทำได้โดยเริ่มที่รากแล้วลงไปทางซ้าย จนกว่าจะพบปมที่ไม่มีลูกซ้าย

$O(h)$

```
public Object getMin() {
  Node r = root;
  if (r == null) return null;
  while (r.left != null) {
    r = r.left;
  }
  return r.element;
}
```

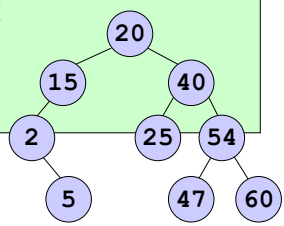


getMax : การค้นหาข้อมูลตัวมากสุด

- ปมใดมีลูกทางขวา ย่อมมีข้อมูลที่มากกว่า
- หาตัวมากสุด ทำได้โดยเริ่มที่รากแล้วลงไปทางขวา จนกว่าจะพบปมที่ไม่มีลูกขวา

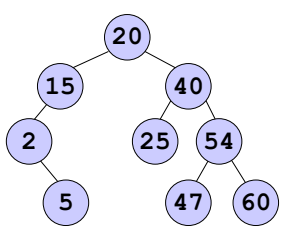
```
public Object getMax() {
  Node r = root;
  if (r == null) return null;
  while (r.right != null) {
    r = r.right;
  }
  return r.element;
}
```

$O(h)$



การเพิ่มข้อมูล

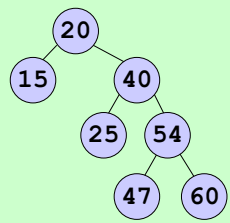
- สร้างปมใหม่, เพิ่มเข้าในต้นไม้
- เพิ่มเป็นใบใหม่ ณ ตำแหน่งที่ได้จากการค้น



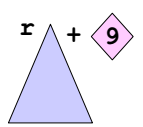
$O(h)$

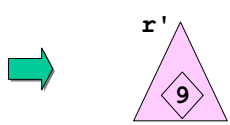
add : การเพิ่มข้อมูล

```
public void add(Object e) {
  Node newNode = new Node(e, null, null);
  if (root == null) root = newNode;
  else {
    Node p = null, r = root;
    while( r != null ) {
      int cmp = compare(e, r.element);
      if      (cmp < 0) {p = r; r = r.left;}
      else if (cmp > 0) {p = r; r = r.right;}
      else return;
    }
    if (compare(e, p.element) < 0)
      p.left = newNode;
    else
      p.right = newNode;
  }
  ++size;
}
```

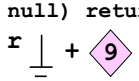


การเพิ่มข้อมูลแบบเวียนเกิด

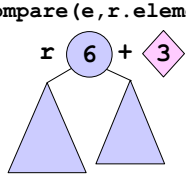


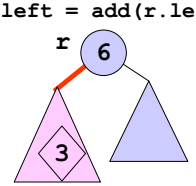



```
if (r == null) return new Node(e, null, null)
```




```
if (compare(e,r.element)<0) r.left = add(r.left,e);
```





add : แบบเวียนเกิด

```
public void add(Object e) {
  root = add(root, e);
}
Node add(Node r, Object e) {
  if (r == null) {
    r = new Node(e, null, null);
    ++size;
  } else {
    int cmp = compare(e, r.element);
    if (cmp < 0)
      r.left = add(r.left, e);
    else if (cmp > 0)
      r.right = add(r.right, e);
  }
  return r;
}
```

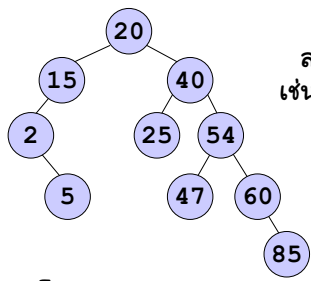

ลักษณะของต้นไม้ขึ้นกับลำดับการเพิ่ม

`1, 2, 6, 3, 5` `2, 1, 5, 3, 6`

ความสูงของต้นไม้ขึ้นกับลำดับ
ของข้อมูลที่เพิ่มใส่ต้นไม้

การลบข้อมูล

- ค้นหาปมที่เก็บข้อมูลที่ต้องการลบ
- ลบปมที่เก็บข้อมูลนั้น หรือลบข้อมูลในปมนั้น



ลบปมที่มี 2 ลูก
เช่น ต้องการลบ 40

$O(h)$

ลบใบ
เช่น ต้องการลบ 5

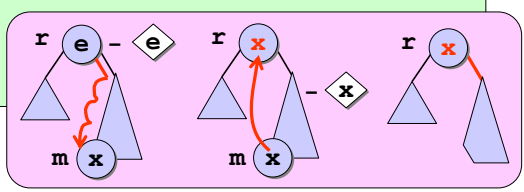
ลบปมที่มีลูกเดียว
เช่น ต้องการลบ 60

remove : แบบเวียนเกิด

```
public void remove(Object e) {
  root = remove(root, e);
}
Node remove(Node r, Object e) {
  if (r == null) return r;
  int cmp = compare(e, r.element);
  if (cmp < 0) {
    r.left = remove(r.left, e);
  } else if (cmp > 0) {
    r.right = remove(r.right, e);
  } else {
    // พบแล้ว ลบที่นี่



  }
  return r;
}
```

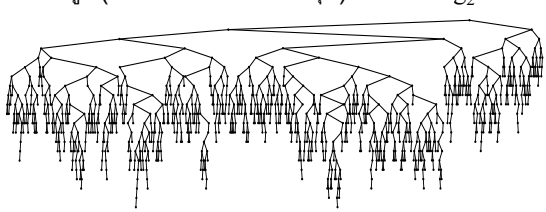
remove : แบบเวียนเกิด

```
Node remove(Node r, Object e) {
  ...
  } else {
    if (r.left == null || r.right == null) {
      r = (r.left == null ? r.right : r.left);
      --size;
    } else {
      Node m = r.right;
      while (m.left != null) m = m.left;
      r.element = m.element;
      r.right = remove(r.right, m.element);
    }
  }
  return r;
}
```



ต้นไม้ BSTree ที่สร้างจากข้อมูลสุ่ม

- ต้นไม้ที่เก็บข้อมูล n ตัว
- ช่วงของความสูง : $\lfloor \log_2 n \rfloor \le h \le n - 1$
- ถ้าสร้างจากข้อมูลสุ่ม สามารถวิเคราะห์ได้ว่า
  - ความลึกเฉลี่ยของปมภายใน $\approx 1.39 \log_2 n$
  - ความลึกเฉลี่ยของ null $\approx 2 + 1.39 \log_2 n$
  - ความสูง (ความลึกของใบล่างสุด) $\approx 2.99 \log_2 n$



Devroye, L. 1986. A note on the height of binary search trees. *J. ACM 33*, 489–498.

เวลาการทำงานของการเพิ่ม ลบ ค้น

- get, getMin, getMax, add, remove : O(h)
- ต้นไม้มีความสูงในช่วง $\lfloor \log_2 n \rfloor \leq h \leq n - 1$
- กรณีเร็วสุด (ต้นไม้เตี้ยสุด) : O(log n)
- กรณีช้าสุด (ต้นไม้สูงสุด) : O(n)
- กรณีเฉลี่ย (เมื่อต้นไม้สร้างจากข้อมูลสุ่ม) : O(log n)

การเรียงลำดับข้อมูลแบบต้นไม้

- นำข้อมูลทั้งหมด มาสร้างต้นไม้ค้นหาแบบทวิภาค
- แวะผ่านต้นไม้นี้แบบตามลำดับ

```
2, 1, 5, 3, 6
```

```
1  2  3  5  6
```

treeSort : การเรียงลำดับข้อมูล

```
public class BSTree extends BinaryTree {
  ...
  public static void treeSort(final Object[] data) {
    BSTree t = new BSTree();
    for (int i=0; i<data.length; i++) {
      t.add(data[i]);
    }
    t.inOrder(new Visitor() {
                int k = 0;
                public void visit(Object e) {
                  data[k++] = e;
                } } );
  }
  ...
```

เพิ่ม n ครั้งใช้เวลา O(n log n)

แวะผ่าน n ปม ใช้เวลา O(n)

$O(n \log n)$

การสร้างเซตด้วย BSTree

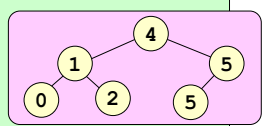
```
public class BSTSet implements Set {
  protected BSTree tree = new BSTree();
  public int size() {
    return tree.size();
  }
  public boolean isEmpty() {
    return tree.isEmpty();
  }
  public boolean contains(Object e) {
    return tree.get(e) != null;
  }
  public void add(Object e) {
    tree.add(e);
  }
  public void remove(Object e) {
    tree.remove(e);
  }
}
```

BSTree ไม่เก็บตัวซ้ำอยู่แล้ว

การสร้างคอลเล็กชันด้วย BSTree

- ปรับให้ BSTree เก็บตัวซ้ำ โดยให้ไปเพิ่มทางซ้าย
- ถ้ามีการลบ ตัวซ้ำอาจอยู่ทางขวาก็ได้ (ก็ไม่เป็นไร)

```
Node add(Node r, Object e) {
  if (r == null) {
    r = new Node(e, null, null);
    ++size;
  } else {
    int cmp = compare(e, r.element);
    if (cmp <= 0)
      r.left = add(r.left, e);
    else
      r.right = add(r.right, e);
  }
  return r;
}
```



สรุป

- ต้นไม้ค้นหาแบบทวิภาคมีจัดเก็บข้อมูลโดยอาศัยการเปรียบเทียบความมากกว่าน้อยกว่าของข้อมูล
- สามารถลดปริมาณข้อมูลที่ต้องพิจารณาได้ทีละมากๆ ระหว่างการเพิ่ม ลบ และค้นหา
- เวลาการทำงานขึ้นกับลักษณะของต้นไม้
- โชคดีทำงานเร็ว O(log n), โชคร้ายทำงานช้า O(n)
- เป็นโครงสร้างพื้นฐานของโครงสร้างข้อมูลอื่น ๆ ที่ซับซ้อนและมีประสิทธิภาพกว่า

ต้นไม้เอวีแอล

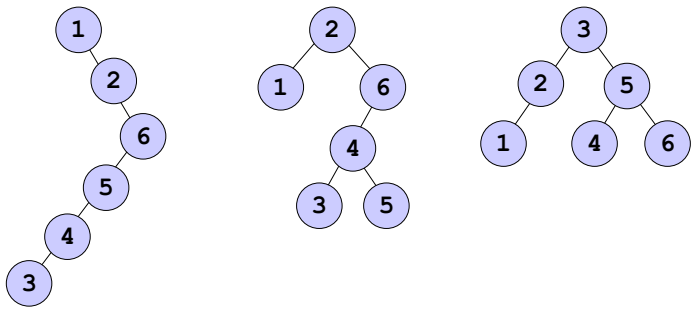
(AVL Tree)

หัวข้อ

- นิยามต้นไม้เอวีแอล
- การวิเคราะห์ความสูงของต้นไม้เอวีแอล
- การปรับต้นไม้เอวีแอลให้สูงสมดุล
- กระบวนการหมุนปม

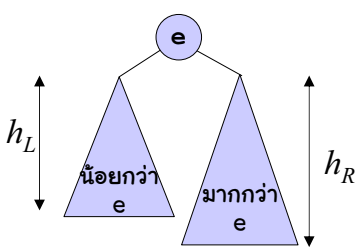
ต้นไม้ค้นหาแบบทวิภาค

- เวลาการทำงานเป็น $O(h)$
- $\lfloor \log_2 n \rfloor \leq h \leq n-1$
- โชคดีทำงานเร็ว $O(\log n)$, โชคร้ายทำงานช้า $O(n)$



ต้นไม้เอวีแอล

- AVL = Binary Search Tress + กฎความสูงสมดุล

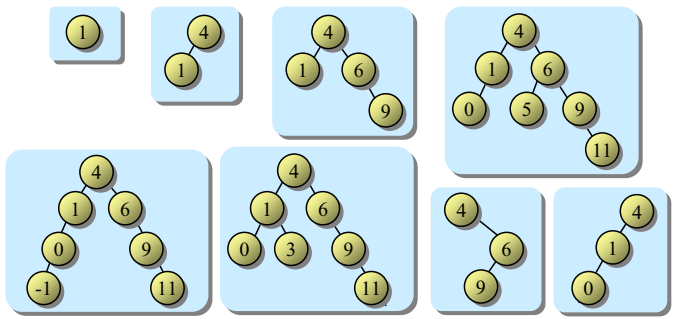


$$|h_L - h_R| \leq 1$$

ต้นไม้ย่อยทุกต้นต้องเป็นไปตามกฎ

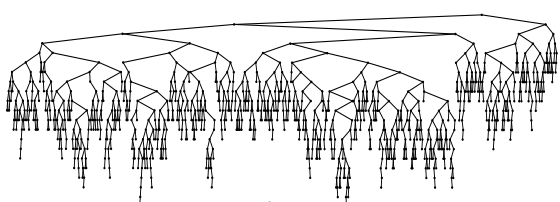
AVL : Adelson-Velskii and Landis

ตัวอย่างต้นไม้เอวีแอล

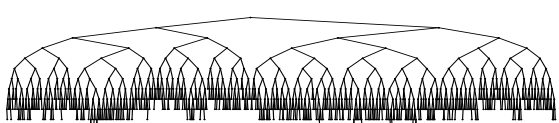


ต้นไม้ว่าง (null) สูง -1

ต้นไม้ค้นหาแบบทวิภาคกับเอวีแอล



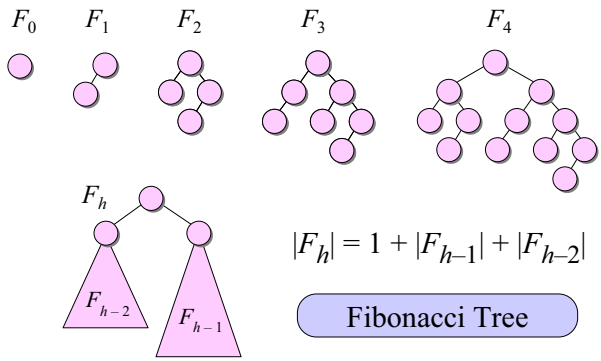
ต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่ม 1000 ตัว



ต้นไม้เอวีแอลที่สร้างจากข้อมูลสุ่ม 1000 ตัว

ต้นไม้เอวีแอลสูงเท่าใด ?

- ให้ F_h คือต้นไม้เอวีแอลซึ่งสูง h ที่มีจำนวนปมน้อยสุด



$$|F_h| = 1 + |F_{h-1}| + |F_{h-2}|$$

Fibonacci Tree

ความสูงของต้นไม้ฟิโบนักชี

$$|F_h| = 1 + |F_{h-1}| + |F_{h-2}|$$

$$n_h = 1 + n_{h-1} + n_{h-2} \quad h \geq 2,\ n_0 = 1, n_1 = 2$$

$$n_h = \alpha_1 \phi^h + \alpha_2 \hat{\phi}^h - 1,\ \phi = 1.618..., \hat{\phi} = -0.618$$

$$n_h \approx \alpha_1 \phi^h$$

$$h \approx \frac{1}{\log_2 \phi}(\log_2 n_h)$$

สรุป :
ต้นไม้เอวีแอลที่มี n ปม
สูงไม่เกิน 1.44 $\log_2 n$

$$h \approx 1.44(\log_2 n_h)$$

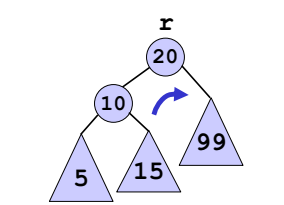
$$\lfloor \log_2 n \rfloor \leq h_{AVL} \leq 1.44 \log_2 n$$

ทำอย่างไรให้เป็นไปตามกฎของ AVL

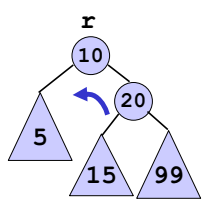
- การเพิ่ม/ลบข้อมูลทำเหมือน BSTree
- แต่หลังการเพิ่ม/ลบ อาจทำให้ผิดกฎสูงสมดุล
- ถ้าผิดกฎ ต้องปรับต้นไม้

การหมุนปม

- การปรับต้นไม้อาศัยการหมุน (rotation)
- การหมุนปมยังคงรักษาความเป็นต้นไม้ค้นหา



`rotateLeftChild(r)`

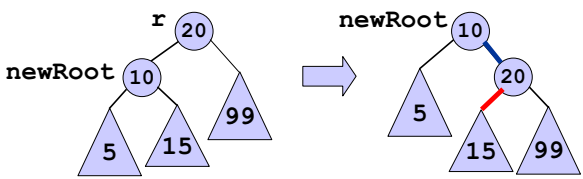


`rotateRightChild(r)`

rotateLeftChild(r)

```
public class BSTree extends BinaryTree {
  ...
  Node rotateLeftChild(Node r) {
    Node newRoot = r.left;
    r.left = newRoot.right;
    newRoot.right = r;
    return newRoot;
  }
  ...
```

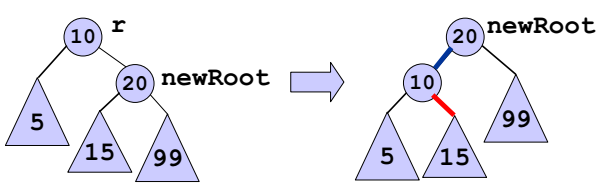
$\Theta(1)$



rotateRightChild(r)

```
public class BSTree extends BinaryTree {
  ...
  Node rotateRightChild(Node r) {
    Node newRoot = r.right;
    r.right = newRoot.left;
    newRoot.left = r;
    return newRoot;
  }
  ...
```

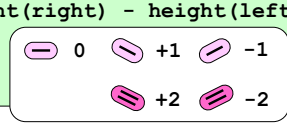
$\Theta(1)$



โครงสร้างปมของต้นไม้เอวีแอล

```
public class AVLTree extends BSTree {
  private static class AVLNode extends Node {
    private int height;
    AVLNode (Object e, Node left, Node right) {
      super(e, left, right);
      setHeight();
    }
    void setHeight() {
      height = 1+Math.max(height(left),height(right));
    }
    int height(Node n) {
      return (n == null ? -1 : ((AVLNode) n).height);
    }
    int balanceValue() {
      return height(right) - height(left);
    }
  }
  ...
```

แต่ละปมมีความสูงกำกับ



add และ remove ใช้ rebalance

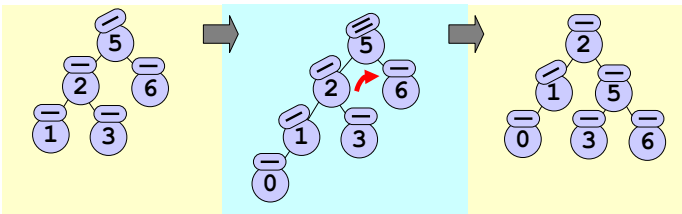
```
public class AVLTree extends BSTree {
  ...
  Node add(Node r, Object e) {
    if (r == null) {
      r = new AVLNode(e, null, null);
      ++size;
    } else {
      r = super.add(r,e);
      r = rebalance(r);
    }
    return r;
  }
  Node remove(Node r, Object e) {
    r = super.remove(r,e);
    r = rebalance(r);
    return r;
  }
```

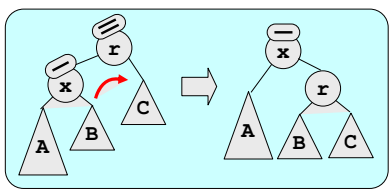
เพิ่มตามปกติ แล้วค่อยปรับ

rebalance มี 4 กรณี

ลบตามปกติ แล้วค่อยปรับ

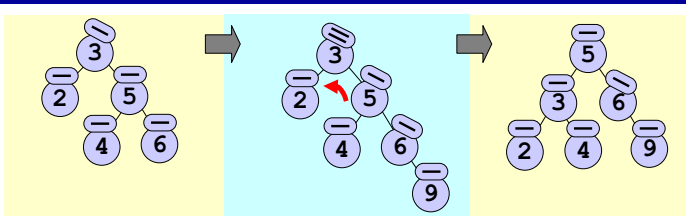
rebalance : กรณีที่ 1

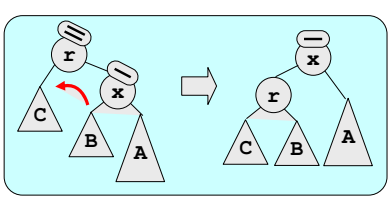




`rotateLeftChild(r)`

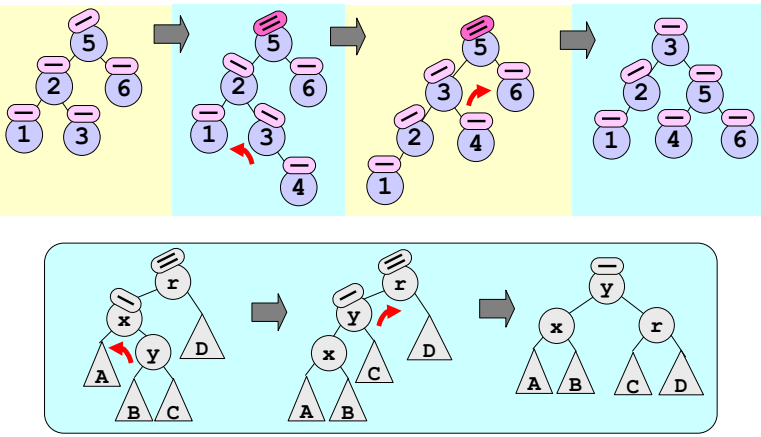
rebalance : กรณีที่ 2





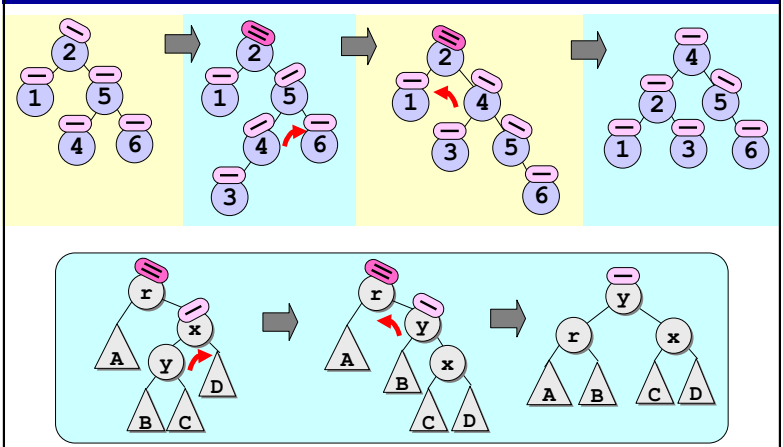
`rotateRightChild(r)`

rebalance : กรณีที่ 3



`rotateRightChild(r.left)` `rotateLeftChild(r)`

rebalance : กรณีที่ 4



`rotateLeftChild(r.right)` `rotateRightChild(r)`

rebalance

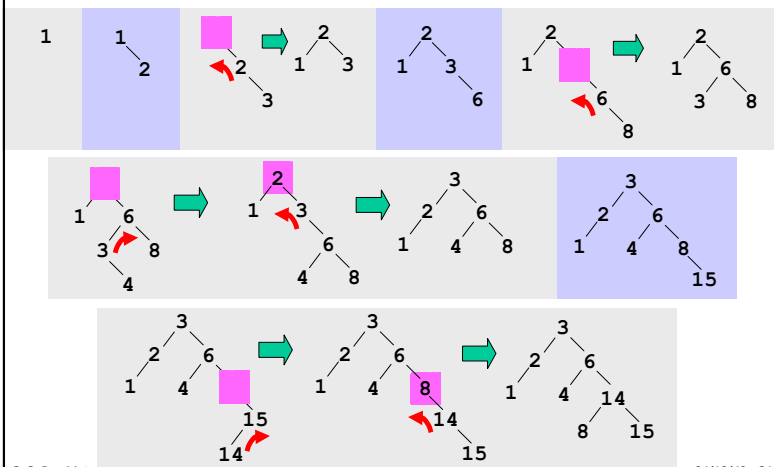
```
private Node rebalance(Node r) {
  if (r == null) return r;
  int balance = ((AVLNode)r).balanceValue();
  if (balance == -2) {
    if (((AVLNode)r.left).balanceValue() == 1)
      r.left = rotateRightChild(r.left);
    r = rotateLeftChild(r);
  } else if (balance == 2) {
    if (((AVLNode)r.right).balanceValue() == -1)
      r.right = rotateLeftChild(r.right);
    r = rotateRightChild(r);
  }
  ((AVLNode)r).setHeight();
  return r;
}
```


อย่าลืมปรับความสูงหลังการหมุน

```
public class AVLTree extends BSTree {
  ...
  Node rotateLeftChild(Node r) {
    r = super.rotateLeftChild(r);
    ((AVLNode) r.right).setHeight();
    ((AVLNode) r).setHeight();
    return r;
  }
  Node rotateRightChild(Node r) {
    r = super.rotateRightChild(r);
    ((AVLNode) r.left).setHeight();
    ((AVLNode) r).setHeight();
    return r;
  }
}
```

ตัวอย่าง

1, 2, 3, 6, 8, 4, 15, 14



สรุป

- ต้นไม้เอวีแอลคือต้นไม้ค้นหาที่ถูกควบคุมความสูง
- ผลต่างความสูงของลูกสองข้างห้ามเกินหนึ่ง
- พิสูจน์ได้ว่า $\lfloor \log_2 n \rfloor \leq h < 1.44 \log_2 n$
- แต่ละปมเก็บความสูงไว้ตรวจสอบ
- ถ้าหลังเพิ่ม/ลบข้อมูลแล้วผิดกฎ, ให้ปรับต้นไม้
- การปรับต้นไม้อาศัยการหมุนปม
- เวลาการทำงานของการเพิ่ม ลบ และค้นเป็น $O(\log n)$

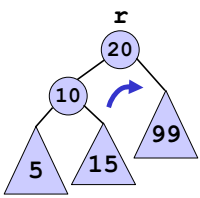
ต้นไม้ค้นหาแบบอื่น ๆ

หัวข้อ

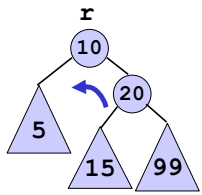
- ต้นไม้ทรีพ (treap)
- ต้นไม้บาน (splay tree)
- ต้นไม้ได้ดุล 2-3-4 (balanced 2-3-4 tree)
- ต้นไม้แดงดำ (red-black tree)

การหมุนปม

- ใช้หมุนปมให้สูงขึ้นหรือต่ำลง
- หมุนแล้วยังคงรักษาความเป็นต้นไม้ค้นหา
- ใช้เวลาคงตัว



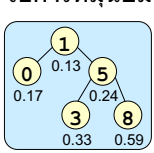
`rotateLeftChild(r)`



`rotateRightChild(r)`

ต้นไม้ทรีพ (Treap)

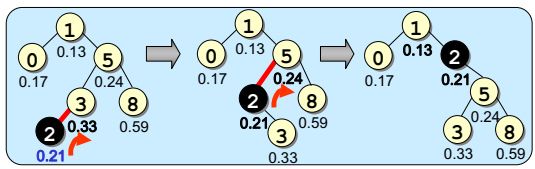
- นำแนวคิดของฮีปเพิ่มในต้นไม้ค้นหาแบบทวิภาค
- แต่ละปมมีข้อมูลสองตัว
  - ตัวข้อมูลจริง ของต้นไม้ค้นหาแบบทวิภาค
  - ข้อมูลเสริม เมื่อมองต้นไม้นี้เป็นฮีปแบบน้อยสุด
- ต้องรักษาโครงสร้างของต้นไม้ให้คงคุณสมบัติ
  - ข้อมูลจริงของต้นไม้ซ้ายน้อยกว่าราก ของต้นขวามากกว่าราก
  - ข้อมูลเสริมของปมพ่อต้องน้อยกว่าของลูก
  - ใช้การหมุนปม สลับความเป็นพ่อลูกได้



ข้อมูลเสริมของฮีปได้มาจาก
การสุ่มจำนวนจริงตอนสร้างปม

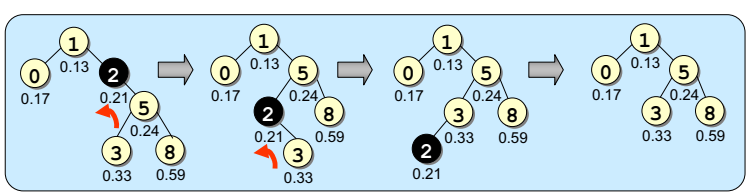
การเพิ่มข้อมูล

- ต้องการเพิ่ม x
  - เพิ่มปมใหม่เก็บ x เหมือนการเพิ่มในต้นไม้ค้นหาแบบทวิภาค
  - สุ่มจำนวนจริงเป็นข้อมูลเสริมกำกับปมใหม่
  - หมุนปม x ขึ้นไปเรื่อย ๆ จนกว่าจะไม่ผิดอันดับแบบฮีป



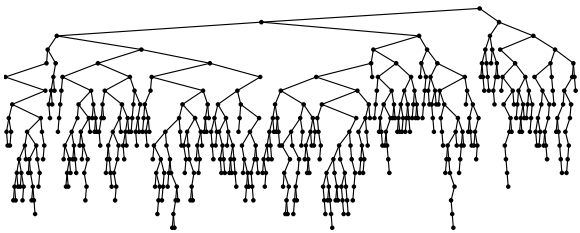
การลบข้อมูล

- ต้องการลบ x
  - ค้นปมที่เก็บ x
  - หมุนปมที่เก็บ x ลงไปเป็นใบ
  - ลบใบนั้นทิ้ง



สรุป : ต้นไม้ทรีพ

- ได้ต้นไม้สูงเฉลี่ยเป็น O(log n)
- การเพิ่ม ลบ ค้น ในกรณีเฉลี่ยใช้เวลา O(log n)

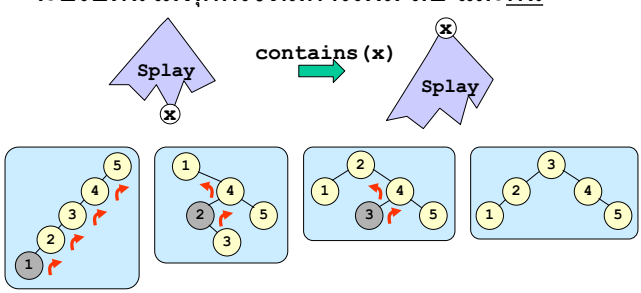


เพิ่มข้อมูลตามลำดับ 1,2,3,...,500

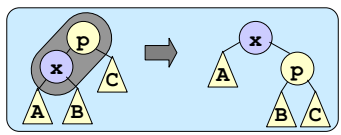
Treap = Binary search Tree + Heap

ต้นไม้บาน (Splay Tree)

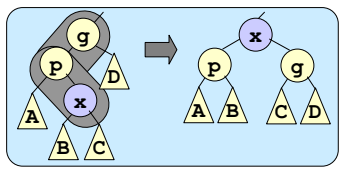
- เป็นต้นไม้ค้นหาแบบทวิภาคที่ปรับโครงสร้างตัวเอง
- จะปรับต้นไม้ทุกครั้งที่มีการเพิ่ม ลบ และค้น



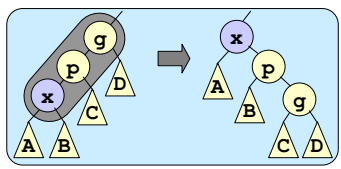
การปรับต้นไม้



zig

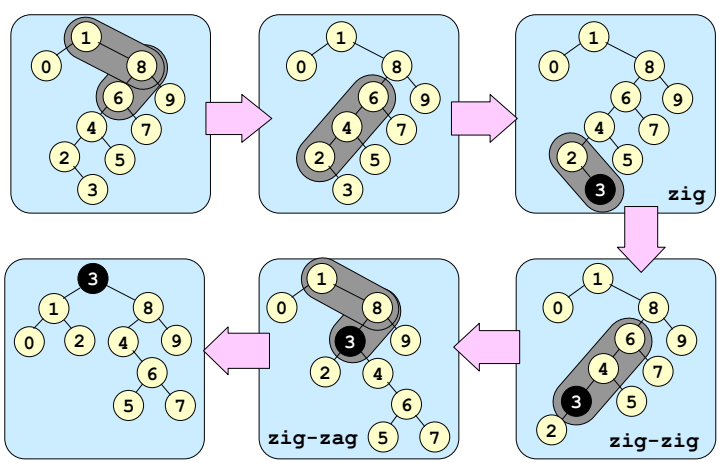


zig-zag



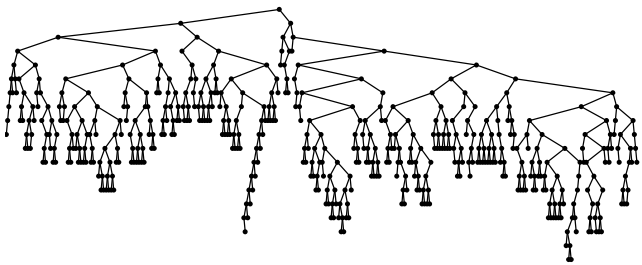
zig-zig

การค้นข้อมูล



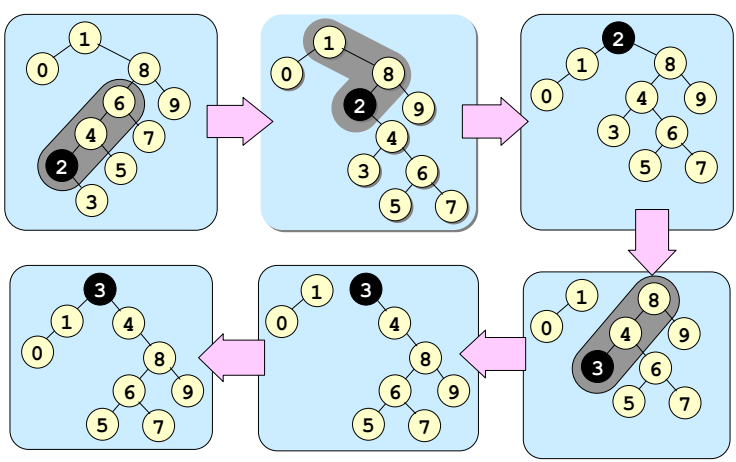
การเพิ่มข้อมูล

- เพิ่มตามปกติ (เหมือนของต้นไม้ค้นหาแบบทวิภาค)
- splay ปมข้อมูลใหม่ ขึ้นมาเป็นราก



เพิ่มข้อมูลตามลำดับ 1,2,3,...,500

การลบข้อมูล

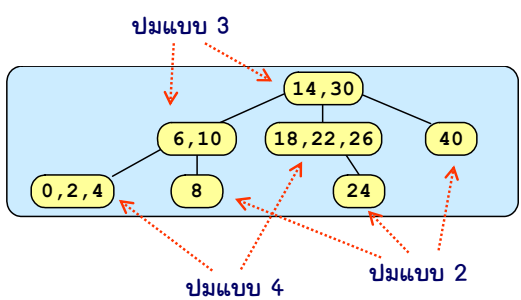


สรุป : ต้นไม้บาน

- ต้นไม้บานไม่ได้ประกันความสูงของต้นไม้
- ข้อมูลตัวที่ถูกค้นบ่อย ๆ จะอยู่บน ๆ จึงถูกค้นในอนาคตได้เร็ว
- ไม่ต้องข้อมูลเสริมใด ๆ ตามปม
- ประกันว่าเวลาสะสมของการให้บริการ add, remove, contains ต่าง ๆ จำนวน m ครั้ง เป็น O(m log n) โดยที่ n เป็นจำนวนข้อมูลของต้นไม้บาน
- แสดงว่าถัวเฉลี่ยต่อ operation คือ O(log n)

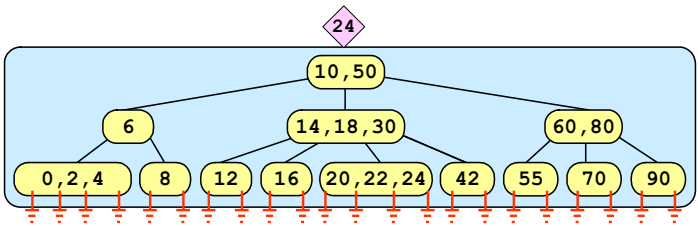
ต้นไม้ 2-3-4

- binary tree : ต้นไม้เตี้ยสุด $\log_2 n$
- m-ary tree : แต่ละปมมี m ลูก ต้นไม้เตี้ยสุด $\log_m n$
- ต้นไม้ 2-3-4 : แต่ละปมมี 2, 3, หรือ 4 ลูกได้
- ปมแบบ k เก็บข้อมูล k – 1 ตัว เรียงจากน้อยไปมาก



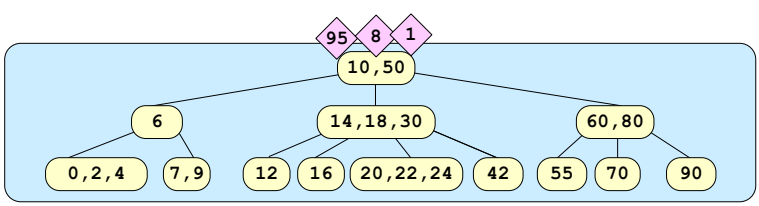
ต้นไม้ได้ดุล 2-3-4

- คือต้นไม้ 2-3-4 ที่ null อยู่ระดับเดียวกันหมด
- $\lfloor \log_4 n \rfloor \le h \le \lfloor \log_2 n \rfloor$



การเพิ่มข้อมูล

- ต้องการเพิ่ม x
  - หา x จนจบที่ใบ
  - ถ้าใบนั้นเป็นปมแบบ 2 ก็แทรก x ให้เป็นปมแบบ 3
  - ถ้าใบนั้นเป็นปมแบบ 3 ก็แทรก x ให้เป็นปมแบบ 4
  - ถ้าใบนั้นเป็นปมแบบ 4 แทรกไม่ได้ให้แตกปม และนำตัวกลางไปเพิ่มในปมระดับบน

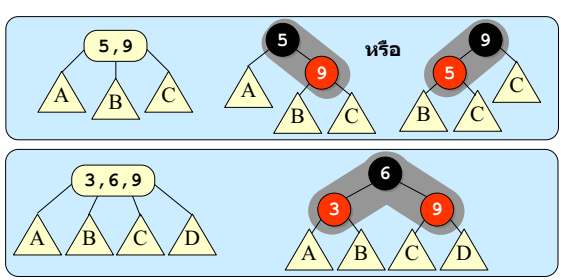


สรุป : ต้นไม้ได้ดุล 2-3-4

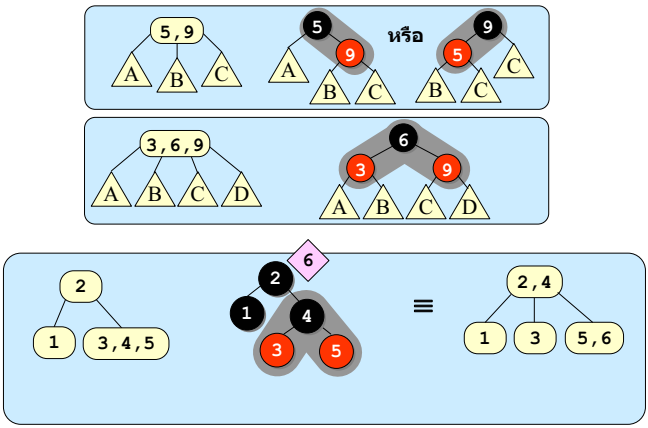
- เป็นต้นไม้ที่ประกันความสูง O(log n)
- เวลาในการค้น เพิ่ม ลบ เป็น O(log n)
- การลบจะซับซ้อนกว่า ทำได้โดยอาศัยการรวมปม ย้ายข้อมูลระหว่างปมญาติสนิท
- เป็นต้นไม้พื้นฐานของต้นไม้บี (B-tree) ซึ่งเป็นต้นไม้สำคัญในการการจัดเก็บฐานข้อมูลขนาดใหญ่ในฮาร์ดดิสก์ (แต่ละปมใน B-tree มีลูกได้เป็นร้อย)

ต้นไม้แดงดำ (Red-Black Tree)

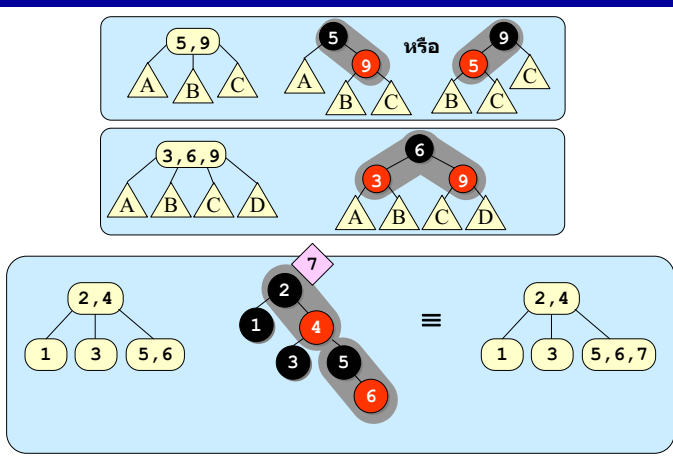
- มีสีกำกับปม, ปมแดงกับปมดำ
- เป็นรูปแบบการสร้างต้นไม้ 2-3-4 ด้วยปมแบบ 2
  - ปมแบบ 3 แทนด้วยปมแบบ 2 สองปม
  - ปมแบบ 4 แทนด้วยปมแบบ 2 สามปม



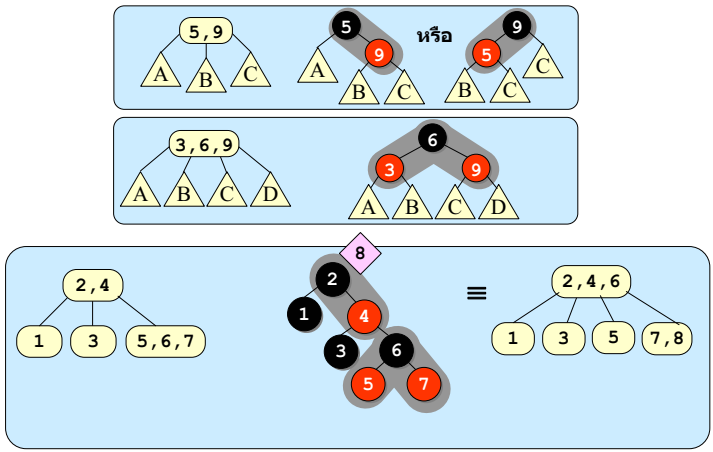
การเพิ่มข้อมูล



การเพิ่มข้อมูล



การเพิ่มข้อมูล



สรุป : ต้นไม้แดงดำ

- เป็นวิธีการสร้างต้นไม้ได้ดุล 2-3-4 แบบหนึ่ง
- แต่ละปมต้องใช้ 1 บิตเก็บสีกำกับปม
- ปมแบบ 3 และแบบ 4 เปลี่ยนเป็นแบบ 2
- ดังนั้นสูงอย่างมาก 2 เท่าของต้นไม้ได้ดุล 2-3-4
- เวลาในการค้น เพิ่ม ลบ เป็น O(log n)
- เป็นต้นไม้ค้นหาที่ใช้ในคลาสมาตรฐานของจาวา

สรุป

- ต้นไม้ค้นหามีอีกหลายแบบ
- แบบประกันความสูง
  - ต้นไม้เอวีแอล, ต้นไม้ได้ดุล 2-3-4, ต้นไม้แดงดำ
- แบบประกันประสิทธิภาพในกรณีเฉลี่ย
  - ต้นไม้ทรีพ
- แบบประกันประสิทธิภาพในกรณีถัวเฉลี่ย
  - ต้นไม้บาน

ตารางแฮช

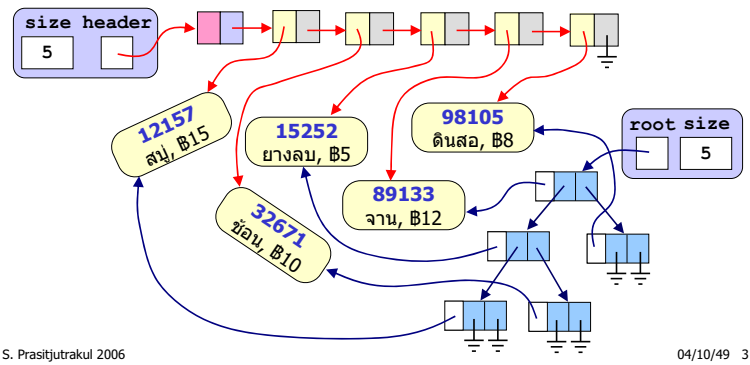
(Hash Tables)

หัวข้อ

- การใช้ตารางเก็บข้อมูลด้วยฟังก์ชันดัชนี
- การเก็บข้อมูลแบบแยกกันโยง
- ฟังก์ชันแฮช
- กลวิธีการเขียนฟังก์ชันแฮช
- การแฮชในจาวา
- การกำหนดเลขที่อยู่เปิด
- การเกาะกลุ่มของข้อมูล

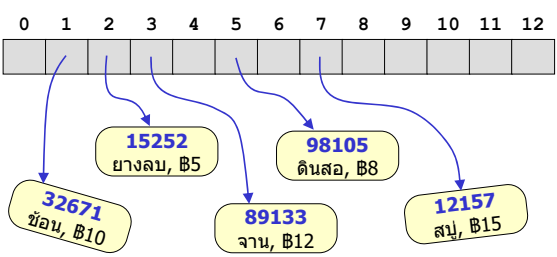
ที่เก็บข้อมูล

- เก็บในรายการ (list) : O(n)
- เก็บในต้นไม้เอวีแอล : O(log n)
- ทำอย่างไรให้เร็วกว่านี้ ?



ใช้ฟังก์ชันดัชนีคำนวณตำแหน่ง

- key ของข้อมูลคือส่วนของข้อมูลที่ใช้ในการค้น
- มีตารางซึ่งแต่ละช่องเป็นที่เก็บข้อมูล
- หา f(key) เพื่อแปลง key ไปเป็น index ของตาราง
- ฟังก์ชันดัชนีหาไม่ยาก ถ้าจองตารางขนาดใหญ่ ๆ




```
f(key) = key % 10
```

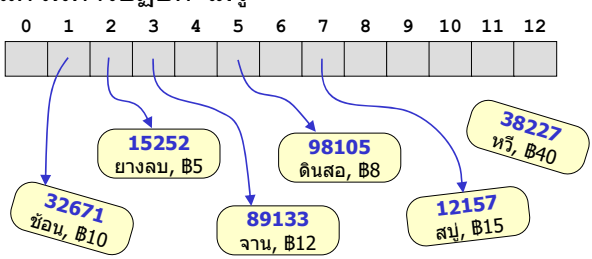
Table

```
public class Table implements Set {
  private Object[] table;
  private int size = 0;

  public Table(int m) { table = new Object[m]; }
  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }

  public void add(Object x) {
    if (table[f(x)] == null)
      {++size; table[f(x)] = x;}
  }
  public void remove(Object x) {
    if (table[f(x)] != null && table[f(x)].equals(x))
      {--size; table[f(x)] = null;}
  }
  public void contains(Object x) {
    return table[f(x)] != null && table[f(x)].equals(x);
  }
  private int f(Object x) { ... }
}
```

Θ(1)

ฟังก์ชันดัชนีนั้นหายาก

- เมื่อต้องเก็บอย่างประหยัด
- เมื่อต้องประกันว่าไม่เกิดการ "ชน"
- ถ้ารู้ชุดข้อมูลที่จะจัดเก็บก่อน ก็อาจหาสูตรที่ไม่ชนได้
- แต่ในทางปฏิบัติ ไม่รู้



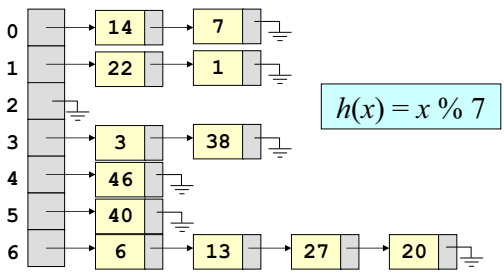
f(key) = key % 10

เปลี่ยนกลยุทธ์ : อนุญาตให้ชนได้

- จะได้เก็บข้อมูลในตารางที่ไม่ใหญ่มาก
- แต่ต้องหาวิธีแก้ไขปัญหาการชน ที่ทำงานได้เร็ว ๆ

Separate Chaining

- จัดเก็บกลุ่มข้อมูลที่ชนกันไว้ในรายการเดียวกัน



SeparateChaining

```
public class SeparateChaining {
  private LinkedList[] table;
  private int size = 0;

  public SeparateChaining(int m) {
    LinkedList[] table = new LinkedList[m];
    for (int i=0; i<table.length; i++)
      table[i] = new LinkedList();
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  ...
```

SeparateChaining

```
public class SeparateChaining {
  ...
  public boolean contains(Object x) {
    return table[h(x)].contains(x);
  }
  public void add(Object x) {
    table[h(x)].add(0, x);
    ++size;
  }
  public void remove(Object x) {
    int i = h(x);
    int s = table[i].size();
    table[i].remove(x);
    if (s > table[i].size()) size--;
  }
  private int h(Object x) {  ...  }
}
```

add ใช้เวลาคงตัว

contains และ remove ใช้เวลาแปรตามความยาว list

การกระจายของข้อมูล

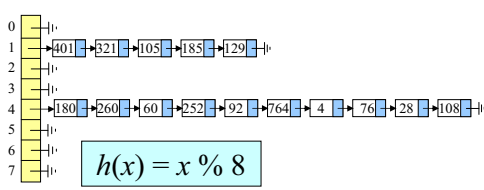
- ถ้าข้อมูลกระจายทั่วตาราง
  - แต่ละช่องเก็บรายการยาว ≈ λ
  - ถ้า λ น้อย ค้นหาได้เร็ว
- ถ้าไม่กระจาย
  - มีบางรายการยาวเกิน λ มาก
  - การค้นหาช้าเหมือนเก็บด้วย list

load factor

$\lambda = n / m$

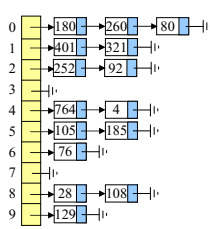
ปริมาณข้อมูล

ขนาดของตาราง



$h(x) = x \% 8$

$h(x) = x \% 10$

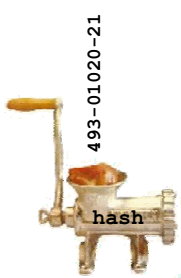


การกระจายของข้อมูล

- ขึ้นกับ
  - x : คีย์ของข้อมูล
  - h(x) : ฟังก์ชันการแปลงคีย์เป็นเลขที่ช่องของตาราง
- ถ้ากลุ่มข้อมูลมีคีย์ x ที่มีค่ากระจายอยู่แล้ว
  - ถ้าใช้ตาราง 100 ช่อง, ก็ให้ h(x) = x % 100
  - ถ้าใช้ตาราง 2^k ช่อง, ก็ให้ h(x) = k บิตทางขวาของ x
- ถ้ากลุ่มข้อมูลมีคีย์ที่มีค่าเป็นระเบียบ
  - รหัสนักศึกษา, รหัสประจำตัวบัตรประชาชน, ...
  - ต้องออกแบบ h(x) ให้ทำ x ที่มีระเบียบให้ "เละ"
  - เรียก h(x) ว่าฟังก์ชันแฮช (Hash function)

ฟังก์ชันแฮช (Hash Function)

- www.webster.com
  - hash : to chop (as meat and potatoes) into small pieces
- สอ เสถบุตร
  - สับ, แหลก, นำมาโขลกเข้าด้วยกัน



493-01020-21 → 10291
493-87628-21 → 76102
473-12332-21 → 40001
463-09872-21 → 00012

ตัวอย่างฟังก์ชันแฮช

```
static int h1(int x) {
  long hash = (2654435769L * x) & 0xFFFFFFFFL;
  return (int) (hash >> 22);
}
```

```
static int h2(int x) {
  x = ~x + (x << 15);
  x ^= (x >>> 11);
  x += (x << 3);
  x ^= (x >>> 5);
  x += (x << 10);
  x ^= (x >>> 16);
  return x;
}
```

x	1	2	3	4	5	6	7	8
h1(x)	632	241	874	483	92	725	334	966
h2(x)	500	1001	507	978	486	1014	403	933

กลวิธีการเขียนฟังก์ชันแฮช

- การวิเคราะห์เลขโดด (digit analysis)
- การคูณ (multiplicative hashing)
- การพับ (folding)
- การหาร (modulus hashing)

การวิเคราะห์เลขโดด (Digit Analysis)

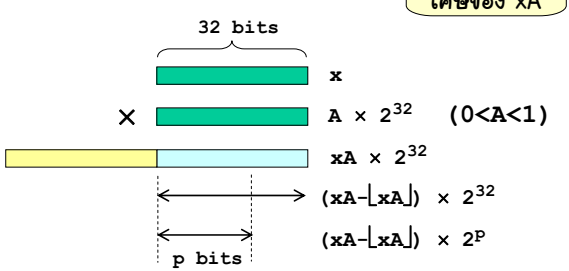
- คัดเลือกเลขโดดบางหลักของคีย์มาพิจารณา
- มั่นใจว่าที่ตัดไปไม่ทำให้เกิดความเอนเอียงในการกระจายของคีย์
- เช่น
  - รหัสนิสิตวิศวฯ ป.ตรี มีรูปแบบ : xx3xxxxx21
  - ก็ตัดเลข 3 และ 21 ออกจากการพิจารณา
  - k = 4830109521,
  - k1 = $\lfloor k / 100 \rfloor$ // k1 = 48301095
  - k2 = $\lfloor k1 / 10^6 \rfloor$ // k2 = 48
  - k3 = $k2*10^5 + k1 \% 10^5$ // k3 = 4801095

การคูณ (Multiplicative Hashing)

- คูณคีย์ด้วยจำนวนจริง A ที่มีค่าระหว่าง (0,1)
- นำเศษมาคูณกับขนาดของตาราง (m = 2^p)

$$h(x) = \lfloor m\left(xA - \lfloor xA \rfloor\right) \rfloor$$

ส่วนที่เป็นเศษของ xA



การคูณ : Fibonacci Hashing

- ถ้า A = golden ratio 0.6180339887... จะแยกคีย์ที่มีค่าใกล้กันออกจากกันได้ดี $\hat{\phi} = \frac{\sqrt{5}-1}{2}$

```
int multHash(int x, int p) {
  long s = 2654435769L;
  long hash = (s * x) & 0xFFFFFFFFL;
  return (hash >> (32-p));
}
```

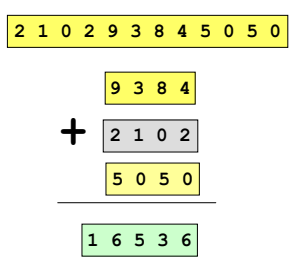
0.6180339887 × 2^{32}

```
for (int i = 0; i < 10; i++) {
  System.out.print(multHash(i, 16)+",");
}
```

```
0,40503,15470,55974,30941,5909,46412,21380,61883,36851
```


การพับ (Folding)

- แบ่งคีย์ออกเป็นส่วนๆ แล้วนำมา "รวม" กัน
- "รวม" ≡ บวก, xor, ...



การหาร (Modulus Hashing)

- $h(x) = x \% p$
- ไม่ควรเลือก
  - $p = 10^q$ เพราะเลือกเฉพาะ q หลักขวา ถ้าคีย์เป็นฐานสิบ
  - $p = 2^q$ เพราะเลือกเฉพาะ q บิตขวา
  - p ที่มีค่าน้อย ๆ เป็นตัวประกอบ
    - ถ้า c คือตัวประกอบร่วมของ p และ x
    - ค่า $x \% p$ จะเป็นจำนวนเท่าของ c
    - ถ้า c มีค่าน้อย ๆ จะมีคีย์จำนวนมากที่ได้ $x \% p$ มีค่าเป็นจำนวนเท่าของตัวประกอบนั้น ซึ่งไม่กระจาย
- โดยทั่วไปเลือก p ที่เป็นจำนวนเฉพาะ

ข้อมูลใด ๆ ก็เปลี่ยนเป็นจำนวนเต็มได้

- double หรือ float → จำนวนเต็ม
  - Double.doubleToLongBits(d)
  - Float.floatToIntBits(f)
- boolean : true → 1, false → 0
- สตริง → จำนวนเต็ม
  - ข้อมูลเป็นสตริงภาษาอังกฤษตัวใหญ่ ก็มองเป็นเลขฐาน 26 "DATA" → $3x26^3 + 0x26^2 + 19x26^1 + 0x26^0 = 53222$
- อ็อบเจกต์ → จำนวนเต็ม
  - แปลงข้อมูลภายในให้เป็นจำนวนเต็มแล้วนำมา "รวม" กัน

ตัวอย่าง

```
public int h(String s) {
  int hash = 0;
  for (int i=0; i<s.length(); i++)
    hash = 31 * hash + s.charAt(i);
  return (hash & 0x7FFFFFFF);
}
```

```
public int h(Point2D p) {
  long bits = Double.doubleToLongBits(p.getX());
  bits ^= Double.doubleToLongBits(p.getY()) * 31;
  hash = (((int) bits) ^ ((int) (bits >> 32)));
  return (hash & 0x7FFFFFFF);
}
```


การแฮชเอกภพ (Universal Hashing)

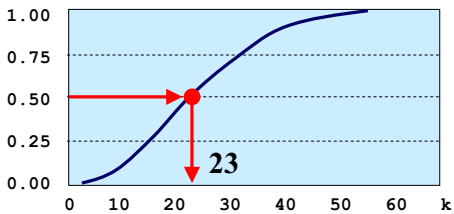
- วิธี hash ที่ผ่านมา เดาพฤติกรรมได้
  - ชุดข้อมูลที่ชนกันมากวันนี้ ก็จะชนกันมากตลอดไป
- ใช้สูตร $h(x) = ((ax + b) \% p) \% m)$
  - $x \in \{0, 1, ..., u-1\}$, u คือจำนวนคีย์ที่เป็นไปได้
  - m คือขนาดตาราง
  - หา p ซึ่งคือจำนวนเฉพาะหนึ่งตัวในช่วง $[u, 2u)$
  - $0 < a < p$ และ $0 \le b < p$
- สุ่มเลือกค่า a และ b ก่อนใช้งาน
  - ชุดข้อมูลที่ชนกันมากวันนี้ อาจชนกันน้อยวันหน้า
  - สามารถพิสูจน์ได้ว่า จำนวนการชนเฉลี่ยเท่ากับ λ

ปฏิทรรศน์วันเกิด (Birthday Paradox)

- ต้องมีคนในห้องกี่คนขึ้นไป จึงจะโอกาสเกินครึ่งที่จะมีคนเกิดวันเดือนเดียวกันสองคนขึ้นไป

k คน โอกาสที่มีวันเกิดไม่ซ้ำกัน = $\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)$

$$1-\left(\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)\right)>0.5$$



ฟังก์ชันแฮชในจาวา

- คลาส Object มีเมท็อดชื่อ hashCode() โดยที่
  - ถ้า x.equals(y) เป็นจริง, x.hashCode() ต้อง == y.hashCode()
- hashCode ที่คลาส Object คืนค่าตำแหน่งเริ่มต้นของออปเจกต์ในหน่วยความจำ
  - ออปเจกต์ต่างกัน ได้ hashCode ต่างกัน
  - value object ควร overrides hashCode ให้ออปเจกต์สองตัวที่มีค่าเท่ากันต้องมี hashCode เหมือนกัน

```
System.out.println(new Integer(1234).hashCode());
System.out.println(new Integer(1234).hashCode());
System.out.println(new Object().hashCode());
System.out.println(new Object().hashCode());
```

```
1234
1234
8222510
18581223
```

ตัวอย่างการเขียน hashCode()

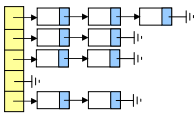
```
public class Point2D {
  private double x, y;
  ...
  public int hashCode() {
    long bits = Double.doubleToLongBits(x);
    bits ^= Double.doubleToLongBits(y) * 31;
    return (((int) bits) ^ ((int) (bits >> 32)));
  }
}
```

```
public class Book {
  private String name;
  private String publisher;
  private double price;
  ...
  public int hashCode() {
    return name.hashCode() ^ publisher.hashCode();
  }
}
```

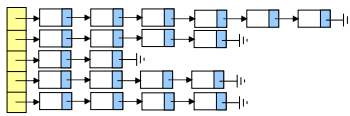

อีกครั้ง : SeparateChaining

```
public class SeparateChaining {
  ...
  public boolean contains(Object e) {
    return table[h(e)].contains(e);
  }
  public void add(Object e) {
    table[h(e)].add(0, e);
    ++size;
  }
  public void remove(Object e) {
    int i = h(e);
    int s = table[i].size();
    table[i].remove(e);
    if (s > table[i].size()) size--;
  }
  private int h(Object x) {
    return Math.abs(x.hashCode()) % table.length;
  }
}
```

Rehashing

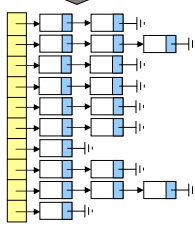


$\lambda = 1.8$



$\lambda = 4$

Rehashing



$\lambda = 2$

ถ้าฟังก์ชันแฮชกระจายดี

การลบและค้นใช้เวลา $O(\lambda)$

ถ้าควบคุม λ ไม่ให้เกินค่าคงตัว k

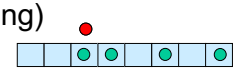
การลบและค้นใช้เวลาคงตัว

SeparateChaining : Rehash

```
public void add(Object e) {
  table[h(e)].add(0, e);
  ++size;
  if (size/table.length >= threshold) rehash();
}
private void rehash() {
  LinkedList[] oldTable = table;
  table = new LinkedList[2*table.length];
  for (int i=0; i<table.length; i++)
    table[i] = new LinkedList();
  for (int i=0; i<oldTable.length; i++) {
    Object[] items = oldTable[i].toArray();
    for (int j=0; j<items.length; j++) {
      table[ h(items[j]) ].add(0, items[j]);
    }
  }
}
```

การแก้ปัญหาการชนแบบอื่น

- แบบแยกกันโยง (separate chaining)
  - แต่ละช่องในตารางเก็บรายการโยงของข้อมูล
  - ข้อมูลที่ชนกันเก็บอยู่ด้วยกัน ไม่กระทบข้อมูลอื่น
  - เปลืองตัวโยง (links)
- แบบเลขที่อยู่เปิด (open addressing)

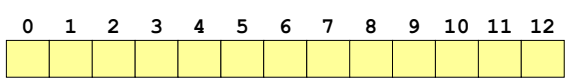


  - แต่ละช่องในตารางเก็บข้อมูล
  - ถ้าชน ก็หาช่องว่างใหม่ในตารางเพื่อเก็บข้อมูล
  - $\lambda = n/m \leq 1$ เสมอ ต้องคุมไม่ให้เกินเกณฑ์ ($\lambda \leq 0.5$)
  - มีหลายวิธีในการหาช่องว่างใหม่ในตาราง เมื่อเกิดการชน
    - การตรวจเชิงเส้น (linear probing)
    - การตรวจกำลังสอง (quadratic probing)
    - การตรวจสองชั้น (double hashing)

การตรวจเชิงเส้น (Linear Probing)

- เมื่อชน หาช่องว่างถัดไปด้วยวิธีดูตัวถัดไปเรื่อย ๆ
- ให้ $h_j(x)$ คือช่องที่ probe หลังจากชนครั้งที่ j
- $h_0(x) = h(x)$ คือช่องที่ hash เริ่มต้น (home address)

$$h_j(x) = (h(x) + j) \% m \qquad h_j(x) = (h_{j-1}(x) + 1) \% m$$



ใช้ `h(x) = x % 13` แล้วเพิ่มข้อมูลที่มีคีย์ตามลำดับดังนี้

`17 32 26 7 4 43 12 11 24`

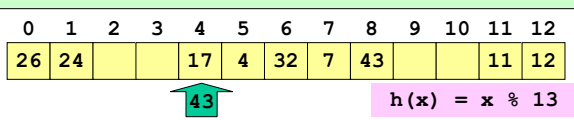
LinearProbingHashSet

```
public class LinearProbingHashSet implements Set {
  private Object[] table;
  private int size = 0;
  public LinearProbingHashSet(int m) {
    table = new Object[m];
  }
  public boolean contains(Object e) {
    return table[indexOf(e)] != null;
  }
  private int indexOf(Object e) {
    int h = h(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + 1) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
```


LinearProbingHashSet

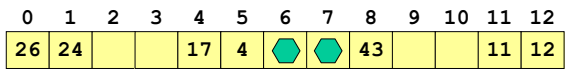
```
  public void add(Object e) {
    int i = indexOf(e);
    if (table[i] == null) {
      table[i] = e;
      ++size;
    }
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (table[i] != null) {
      table[i] = null;
      --size;
    }
  }
```

ผิด



สถานะของช่องเก็บข้อมูล

- แต่ละช่องมี 3 สถานะ
  - ช่องว่าง ๆ ไม่เคยมีข้อมูลมาเก็บเลย `table[i] == null`
  - ช่องที่เก็บข้อมูลที่ถูกลบไปแล้ว `table[i] == DELETED`
  - ช่องที่มีข้อมูลเก็บอยู่ `table[i] != null && != DELETED`




```
public void remove(Object e) {
  int i = indexOf(e);
  if (table[i] != null) {
    table[i] = DELETED;
    --size;
  }
}
```

DELETED เป็นอ็อบเจกต์ที่ไม่เท่ากับอ็อบเจกต์อื่น การทำงานใน `indexOf` จะค้นต่อไปเมื่อพบ DELETED

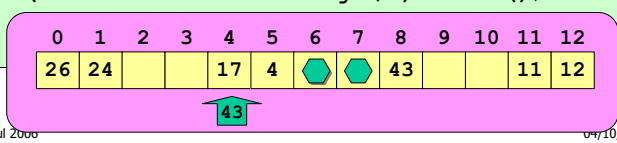
Rehash

```
public class LinearProbingHashSet implements Set {
  private static final Object DELETED = new Object();
  private Object[] table;
  private int size = 0;
  private int numNonNulls = 0;

  public void add(Object e) {
    int i = indexOf(e);
    if (table[i] == null) {
      table[i] = e;
      ++size; ++numNonNulls;
      if (numNonNulls > table.length/2) rehash();
    }
  }
  private void rehash() {
    Object[] old = table;
    table = new Object[4*size];
    size = numNonNulls = 0;
    for(int i = 0; i< old.length; i++)
      if (old[i] != null && old[i] != DELETED) add(old[i]);
  }
```

ไม่ได้ใช้ช่อง DELETED มาใช้ใหม่เลย

การนำช่อง DELETED มาใช้ใหม่

```
public void add(Object e) {
  int empty = -1;
  int h = h(e);
  for (int j=0; j<table.length; j++) {
    if (table[h] == DELETED && empty == -1) empty = h;
  int h = indexOf(e);                 ull || table[h].equals(e)) break;
    h = (h + 1) % table.length;
  }
  if (table[h] == null) {
    if (empty != -1) h = empty;
    table[h] = e;
    ++size; if (empty == -1) ++numNonNulls;
    if (numNonNulls > table.length/2) rehash();
  }
}
```



การเกาะกลุ่มปฐมภูมิ (Primary Clustering)

- ถ้าใช้ linear probling แล้วเพิ่มข้อมูลตัวใหม่อีกตัว ลงตารางข้างล่างนี้ อยากทราบว่า ข้อมูลใหม่นี้จะถูกนำไปเก็บไว้ที่ช่องใดด้วยความน่าจะเป็นสูงสุด

การตรวจกำลังสอง (Quadratic Probing)

- เพื่อขจัดการเกาะกลุ่มปฐมภูมิ
- หลีกเลี่ยงการตรวจช่องติด ๆ กัน
- ให้ตรวจแบบก้าวกระโดดห่าง ๆ

+1, +3, +5, +7,...

$h_j(x) = (h(x) + j^2) \% m$

$h_j(x) = (h_{j-1}(x) + 2j - 1) \% m$

$$
\begin{aligned}
h_j(x) &= (h(x)+j^2) \% m \\
h_{j-1}(x) &= (h(x)+(j-1)^2) \% m \\
h_j(x) - h_{j-1}(x) &= (j^2 - (j-1)^2) \% m \\
&= (j^2 - j^2 + 2j - 1) \% m \\
h_j(x) &= (h_{j-1}(x) + 2j - 1) \% m
\end{aligned}
$$

การตรวจกำลังสองไม่ตรวจทุกช่อง

- ลองเพิ่ม 30 อีกตัว ($h(x) = x \% 13$)

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1		17	4	5		7	8				

h(x) = 4
(4+1²)%13 = 5
(4+2²)%13 = 8
(4+3²)%13 = 0
(4+4²)%13 = 7
(4+5²)%13 = 3
(4+6²)%13 = 1

(4+7²)%13 = 1
(4+8²)%13 = 3
(4+9²)%13 = 7
(4+10²)%13 = 0
(4+11²)%13 = 8
(4+12²)%13 = 5
(4+13²)%13 = 4
...

มีช่องว่างอาจหาไม่พบ

เมื่อตารางมีขนาดเป็นจำนวนเฉพาะ

- การตรวจกำลังสองจะดูอย่างน้อยครึ่งหนึ่งของตาราง
- ดังนั้น ถ้า load factor ≤ ½ ก็สบายใจได้ว่าจะหาช่องว่างพบเมื่อเพิ่มข้อมูล
- พิสูจน์ : ให้ $0 \le i < j \le \lfloor m/2 \rfloor$ ถ้าข้างบนไม่จริง ต้องมีการ probe ครั้งที่ i และ j ที่ดูช่องซ้ำกัน

$$\begin{aligned} h(x) + j^2 &\equiv h(x) + i^2 && \bmod m \\ j^2 &\equiv i^2 && \bmod m \\ (j^2 - i^2) &\equiv 0 && \bmod m \\ (j - i)(j + i) &\equiv 0 && \bmod m \end{aligned}$$

- เป็นไปไม่ได้ : $(j - i)$ ไม่เป็น 0, $(j+i)$ ก็ไม่เป็น m อีกทั้ง $(j - i)(j+i) \% m \neq 0$ เพราะทั้งสองพจน์ $< m$ และ m เป็นจำนวนเฉพาะ

QuadraticProbingHashSet

- เหมือน LinearProbingHashSet ต่างกันแค่เปลี่ยน

```
h = (h + 1) % table.length
```

เป็น

```
h = (h + 2*j-1) % table.length
```

- ต้อง rehash เมื่อ load factor เกินครึ่ง
- ขนาดของตารางเป็นจำนวนเฉพาะตลอด

construct และ rehash

```
import java.math.BigInteger;
public class QuadraticProbingHashSet implements Set {
  private static final Object DELETED = new Object();
  private Object[] table;
  private int size, numNonNulls;

  public QuadraticProbingHashSet(int m) {
    table = new Object[nextPrime(m)];
  }
  private int nextPrime(int n) {
    BigInteger bi = new BigInteger(Integer.toString(n));
    return bi.nextProbablePrime().intValue();
  }
  private void rehash() {
    Object[] old = table;
    table = new Object[nextPrime(4*size)];
    size = numNonNulls = 0;
    for(int i = 0; i< old.length; i++)
      if (old[i] != null && old[i] != DELETED) add(old[i]);
  }
  ...
```

เช่น n=5000 จะได้ 5003

indexOf

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

  public boolean contains(Object e) {
    return table[indexOf(e)] != null;
  }
  private int indexOf(Object e) {
    int h = h(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + 2*j - 1) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
  ...
```

$$h_j(x) = (h_{j-1}(x) + 2j - 1) \% m$$

remove และ add

```
public void remove(Object e) {
  int i = indexOf(e);
  if (table[i] != null) {
    table[i] = DELETED; --size;
  }
}
public void add(Object e) {
  int empty = -1;
  int h = h(e);
  for (int j=0; j<table.length; j++) {
    if (table[h] == DELETED && empty == -1) empty = h;
    if (table[h] == null || table[h].equals(e)) break;
    h = (h + 2*j-1) % table.length;
  }
  if (table[h] == null) {
    if (empty != -1) h = empty;
    table[h] = e;
    ++size; if (empty == -1) ++numNonNulls;
    if (numNonNulls > table.length/2) rehash();
  }
}
```


การเกาะกลุ่ม

- การเกาะกลุ่มปฐมภูมิ (primary clustering)
  - เห็นได้ด้วยตา ข้อมูลอยู่ติด ๆ กัน
  - กลุ่มที่โต ยิ่งมีโอกาสโตขึ้น
  - การค้นจะช้าเหมือนการค้นแบบลำดับ
- การเกาะกลุ่มทุติยภูมิ (secondary clustering)
  - ข้อมูลที่มี $h(x)$ เดียวกัน จะตรวจช่องในตารางเหมือนกัน
  - ระยะกระโดดของการตรวจแปรตามหมายเลขครั้งที่ชน
  - $h_j(x) = (h(x) + j) \% m$, $h_j(x) = (h(x) + j^2) \% m$
  - แก้ปัญหานี้ได้ โดยให้ข้อมูลที่มี $h(x)$ เดียวกัน ไม่จำเป็นต้องมีระยะโดดของการตรวจเหมือนกัน
  - ให้ระยะกระโดดคำนวณจากค่าของข้อมูล

การแฮชสองชั้น (Double Hashing)

- ใช้ฟังก์ชันแฮชอีกตัวเพื่อคำนวณระยะกระโดด
- ทำให้ชุดข้อมูลที่แฮชไปที่ช่องเดียวกัน อาจมีระยะกระโดดต่างกัน

$h_j(x) = (h(x) + j \cdot g(x)) \% m$ $h_j(x) = (h_{j-1}(x) + g(x)) \% m$

- โดยที่ $g(x) \% m \neq 0$ (เพื่อไม่ให้ย่ำอยู่กับที่) เช่น
  - $g(x) = R - (x \% R)$ R เป็นจำนวนเฉพาะ และ $R < m$
- และ ตัวหารร่วมมากของ $g(x)$ และ m ต้องเป็น 1 จะได้ตรวจทุกช่องในตาราง
  - ประกันเงื่อนไขนี้ได้โดยให้ m เป็นจำนวนเฉพาะ
  - $h(x) = 0, g(x) = 4, m = 8$ จะตรวจช่อง 0 และ 4 เท่านั้น
  - $h(x) = 0, g(x) = 4, m = 7$ จะตรวจช่อง 0, 4, 1, 5, 2, 6, 3

DoubleHashingHashSet

```
public class DoubleHashingHashSet implements Set {
  ...

  private int indexOf(Object e) {
    int h = h(e);
    int g = g(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + g) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
  private int g(Object e) {
    return 11 - (Math.abs(e.hashCode()) % 11);
  }
  ...
```

$h_j(x) = (h_{j-1}(x) + g(x)) \% m$

เปรียบเทียบการเกาะกลุ่ม

การตรวจเชิงเส้น (linear probing)

การตรวจกำลังสอง (quadratic probing)

การแฮชสองชั้น (double hashing)

$\lambda = 0.8$

เปรียบเทียบจำนวนการตรวจเฉลี่ย

- การตรวจเชิงเส้นตรวจจำนวนช่องมากกว่าแบบอื่น
- การตรวจกำลังสองและแบบสองชั้นใกล้เคียงกัน
- ถ้า $\lambda \leq 0.5$ ทั้งสามแบบไม่ต่างกันมาก

	Linear Probing		Quadratic Probing		Double Hashing	
	พบ	ไม่พบ	พบ	ไม่พบ	พบ	ไม่พบ
$\lambda = 0.3$	1.21	1.52	1.21	1.47	1.19	1.43
$\lambda = 0.4$	1.33	1.89	1.31	1.75	1.28	1.67
$\lambda = 0.5$	1.50	2.50	1.43	2.14	1.39	2.02
$\lambda = 0.6$	1.75	3.63	1.59	2.72	1.53	2.54
$\lambda = 0.7$	2.16	6.02	1.82	3.70	1.74	3.44
$\lambda = 0.8$	3.00	12.84	2.16	5.64	2.05	5.32
$\lambda = 0.9$	5.44	49.70	2.79	11.37	2.67	11.63

เปรียบเทียบจำนวนการตรวจเฉลี่ย

	จำนวนการตรวจเฉลี่ย	
	หาพบ	หาไม่พบ
แบบแยกกันโยง ($\lambda \geq 0$)	$1 + \lambda/2$	$1 + \lambda$
การตรวจเชิงเส้น ($0 \leq \lambda \leq 1$)	$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
การแฮชสองชั้น ($0 \leq \lambda \leq 1$)	$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$	$\frac{1}{1-\lambda}$

ถาม : เก็บข้อมูลโดยใช้การตรวจเชิงเส้น ถ้าต้องการตรวจโดยเฉลี่ยไม่เกิน 5 ครั้ง ต้องควบคุมให้ตารางแฮชมี λ เป็นเท่าใด

ตอบ : $5 \geq \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ $9 \geq \frac{1}{(1-\lambda)^2}$ $1-\lambda \geq \sqrt{1/9}$ $\lambda \leq 2/3$

เปรียบเทียบเวลาการทำงาน

```
1117=1x3x3x3/2x3x3x3x3x3/2/2x3x3/2/2/2/2/2x3x3x3/2/2/2x3/2
```

สร้าง Set ด้วย	เวลาการทำงาน (ms)
ArraySet	164987
BSTSet	1112
AVLSet	430
LinearProbingHashSet	1903
QuadraticProbingHashSet	390
SeparateChainingHashSet	350

ตอนทำงานเสร็จ set มีข้อมูลจำนวน 73816 ตัว

เปรียบเทียบเนื้อที่

- แบบตรวจเชิงเส้น
  - ต้องการเก็บ 1200 ตัว
  - ให้ λ = 0.5, m = 2400
  - ตารางคืออาเรย์ของ object reference (ช่องละ 4 ไบต์)
  - ตารางใช้เนื้อที่ 4*2400 = 9600 ไบต์
  - ใช้เนื้อที่ทั้งหมด 9600 ไบต์
  - λ = 0.5, จากสูตรได้จำนวนการตรวจเฉลี่ยเป็น 2.5 ช่อง
- แบบแยกกันโยง
  - ต้องการตรวจเฉลี่ยจำนวน 2.5 ช่อง
  - จากสูตร ต้องให้ λ = 1.5
  - ต้องการเก็บ 1200 ตัว
  - ต้องมีตาราง 1200/1.5 = 800 ช่อง
  - ตารางใช้เนื้อที่ 4*800 = 3200 ไบต์
  - ถ้าใช้ singly linked list ไม่มีปมหัว ต้องมี 1200 ปม ใช้เนื้อที่ 1200*8 = 9600 ไบต์
  - รวมเป็น 3200+9600 = 12800 ไบต์

$$\#\text{probes} = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

$$\#\text{probes} = 1 + \lambda$$

แบบแยกกันโยงกับแบบเลขที่อยู่เปิด

- แบบแยกกันโยง
  - เปลืองตัวโยง
  - ไม่มีข้อจุกจิกเรื่องการลบข้อมูล
  - กลุ่มข้อมูลที่ชนกันเองมีผลกระทบในกลุ่มกันเอง ไม่มีผลต่อกลุ่มอื่น
- แบบกำหนดเลขที่อยู่เปิด
  - ประหยัดกว่า ถึงแม้จะมี λ ต่ำ
  - ข้อมูลอยู่ใกล้กัน ระบบ cache ทำให้เข้าถึงข้อมูลได้เร็วกว่าแบบโยงไปมา
  - การลบข้อมูลมีผลต่อการเกาะกลุ่มข้อมูล
  - ข้อมูลที่ชนกันจะมีผลกระทบกับข้อมูลอื่น ๆ

ข้อควรระวัง

- ไม่เหมาะกับบริการที่เกี่ยวข้องกับอันดับของข้อมูล
  - getMin, getMax, ...
  - ต้องค้นทั้งตาราง $\Theta(m+n)$
- ต้องระวังเรื่องฟังก์ชันแฮช

```
public class Book {
  private String isbn;
  ...
  public int hashCode() {
    return isbn.hashCode() | &7FFFFFFFH;
  }
}
```

สรุป

- การค้น เพิ่ม ลบข้อมูลในตารางแฮชทำได้รวดเร็ว
- สามารถปรับเวลาการทำงานให้เร็วขึ้น ด้วยการใช้เนื้อที่เข้าแลก เพื่อให้ได้ λ ที่เหมาะสม
- ฟังก์ชันแฮชมีผลต่อประสิทธิภาพการทำงาน

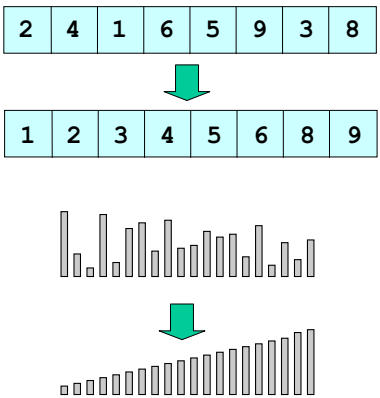
การเรียงลำดับข้อมูล

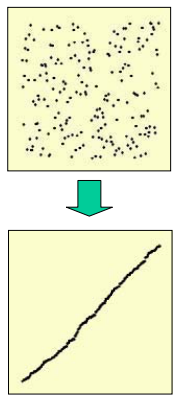
(Sorting)

หัวข้อ

- การเรียงลำดับแบบต่าง ๆ
  - เลือก, ฟอง, แทรก, เชลล์, ฮีป, ผสาน, เร็ว
- ประสิทธิภาพการเรียงลำดับ
  - เวลาการทำงาน
  - ปริมาณหน่วยความจำ
- การเปรียบเทียบการเรียงลำดับแบบต่าง ๆ

การเรียงลำดับ (Sorting)





วิธีการเรียงลำดับ

- เลือก : Selection sort $\Theta(n^2)$
- ฟอง : Bubble sort $O(n^2)$
- แทรก : Insertion sort $O(n^2)$
- เชลล์ : Shell sort $O(n^{1.xx})$
- ฮีป : Heap sort O(n log n)
- ผสาน : Merge sort O(n log n)
- เร็ว : Quick sort O(n log n) avg.

อาศัยการนำข้อมูลมาเปรียบเทียบกันทีละคู่

lessThan, swap

```
public class ArrayUtil {
  ...
  private static boolean lessThan(Object a, Object b) {
    return ((Comparable)a).compareTo(b) < 0;
  }

  private static void swap(Object[] d, int i, int j) {
    Object t = d[i]; d[i] = d[j]; d[j] = t;
  }
  ...
}
```


การเรียงลำดับแบบเลือก (Selection Sort)

- เรียงลำดับข้อมูล d[0] ถึง d[k]
- หาตัวมากสุด
- สลับตัวมากสุดกับตัวท้ายของกลุ่ม
- ขนาดของกลุ่มลดลงหนึ่ง
- ไปเรียงลำดับ d[0] ถึง d[k-1]
- ...
- ทำจนข้อมูลของกลุ่มเหลือตัวเดียว

การเรียงลำดับแบบเลือก : โปรแกรม

```
public static void selectionSort(Object[] d) {
  for (int k=d.length-1; k>0; k--) {
    int m = k;
    for (int j=0; j<k; j++)
      if (lessThan(d[m],d[j])) m = j;
    swap(d, m, k);  // d[m] ←→ d[k]
  }
}
```

$\Theta(n^2)$

รอบที่	จำนวนข้อมูล	#(lessThan)
1	n	n-1
2	n-1	n-2
...	...	...
n-1	2	1

Σ = n(n-1)/2

#swaps = n-1

การเรียงลำดับแบบฟอง (Bubble Sort)

- เรียงลำดับข้อมูล d[0] ถึง d[k]
- เปรียบเทียบคู่ที่ติดกัน จาก 0 ถึง k สลับกันถ้ากลับลำดับ
- ขนาดของกลุ่มลดลงหนึ่ง
- ไปเรียงลำดับ d[0] ถึง d[k-1]
- ...
- ทำจนข้อมูลของกลุ่มเหลือตัวเดียว

การเรียงลำดับแบบฟอง : โปรแกรม

```
public static void bubbleSort(Object[] d) {
  for (int k=d.length; k>1; k--)
    for (int j=1; j<k; j++)
      if (lessThan(d[j], d[j-1])) swap(d, j-1, j);
}
```

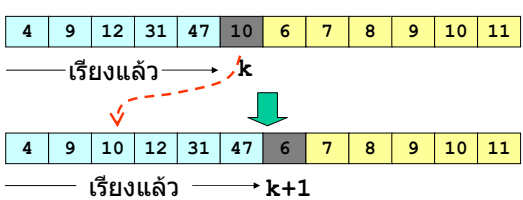
$\Theta(n^2)$

```
public static void bubbleSort(Object[] d)
  for (int k=d.length; k>1; k--) {
    boolean sorted = true;
    for (int j=1; j<k; j++) {
      if (lessThan(d[j], d[j-1])) {
        swap(d, j-1, j);
        sorted = false;
      }
    }
    if (sorted) break;
  }
}
```

$O(n^2)$

การเรียงลำดับแบบแทรก (Insertion Sort)

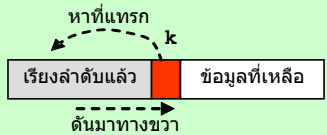
- ต้องการเรียงลำดับ d[0] ถึง d[m]
- พิจารณาข้อมูลตั้งแต่ตัวที่ 1 ถึง m
- เมื่อกำลังพิจารณาข้อมูลตัวที่ k
  - ข้อมูลทางซ้ายของ k : d[0] ถึง d[k-1] เรียงแล้ว
  - หาที่แทรกให้ d[k] ในช่วง 0 ถึง k



การเรียงลำดับแบบแทรก : โปรแกรม

```
public static void insertionSort(Object[] d) {
  for (int k=1; k<d.length; k++) {
    Object t = d[k];
    int j = k-1;
    while (j>=0 && lessThan(t, d[j])) {
      d[j+1] = d[j];
      j--;
    }
    d[j+1] = t;
  }
}
```

$O(n^2)$



รอบที่	จำนวนข้อมูลทางซ้าย	#(lessThan)
1	1	1
2	2	1 ถึง 2
...	...	...
n-1	n-1	1 ถึง n-1

Σ = n-1 ถึง n(n-1)/2

การเรียงลำดับแบบเชลล์ : แนวคิด

- การเรียงลำดับแบบแทรก
  - ช้า เพราะเลื่อนข้อมูลไปช่องถัดไป ทีละตำแหน่ง
- การเรียงลำดับแบบเชลล์
  - แบ่งข้อมูลเป็น h ชุด แบบตัวเว้น h – 1 ตัว
  - sort แต่ละชุดด้วยการเรียงลำดับแบบแทรก
  - ถ้า h == 1 ก็เสร็จ ไม่เช่นนั้น ลดค่า h ลง กลับไปข้อ 1
- หมายเหตุ
  - รอบสุดท้ายเป็นการเรียงลำดับแบบแทรกชุดเดียว

Donald Shell, 1959

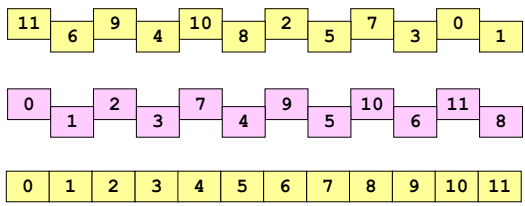
การเรียงลำดับแบบเชลล์ : ตัวอย่าง

- แบ่งเป็น 2 ชุด
- เรียงลำดับแต่ละชุดด้วยแบบแทรก
- รวมกับเป็นชุดเดียว แล้วเรียงลำดับทั้งชุด
- ถ้าเรียงลำดับครั้งเดียวเสร็จด้วยแบบแทรก

15+14, 20 → 49

59

h=1,2



การเรียงลำดับแบบเชลล์ : โปรแกรม

```
public static void shellSort(Object[] d) {
  for (int h=d.length/2; h>0; h/=2) {
    for (int m = 0; m < h; m++) {
      for (int k = h+m  ; k < d.length; k += h) {
        Object t = d[k];  int j = k - h;
        while( j >= 0 && lessThan(t, d[j]) ) {
          d[j + h] = d[j];  j -= h;
        }
        d[j + h] = t;
      }
    }
  }
}
```

h-sequence อะไรดี ?

- ชุดที่ดีมักมีค่าลดลงเป็นเท่า ๆ
  - Shell : 1,2,4,8, ..., 2^m,... $O(n^2)$
  - Hibbard : 1,3,7,15, ..., $2^m - 1$, ... $O(n^{3/2})$
  - Knuth : 1,4,13,40,121, ..., $(3^m - 1)/2$, ... $O(n^{3/2})$
  - Sedgewick : 1,8,23,77, ..., $4^{m+1}+3\cdot 2^m+1$,... $O(n^{4/3})$
- ยังไม่มีผู้ใดพบ h-sequence ที่ดีสุด

```
public static void shellSort(Object[] d) {
  for (int h =d.length/2; h>0; h/=2) {
    for (int m = 0; m < h; m++) {
      ...
```

```
h = h==2 ? 1 : (int)(h/2.2)
```

ของ Gonnet

ทำไมแบบเชลล์ถึงเร็ว

- แบบแทรก : ข้อมูลย้ายตำแหน่งช้า
- แบบเชลล์
  - เริ่มด้วยค่า h ที่มาก แล้วลดลง ๆ
  - h มีค่ามาก : ข้อมูลย้ายตำแหน่งเร็ว
  - ข้อมูลเข้าใกล้ตำแหน่งที่ควรจะอยู่ได้อย่างรวดเร็ว
  - ใช้แบบแทรกเป็นเครื่องมือเพราะ "ยิ่งเรียงยิ่งเร็ว"
  - รอบหลัง ๆ, h น้อย ๆ, เรียงมาก, ก็ทำงานเร็ว

0	1	2	3	4	5	6	7	8	9	10	11
10	12	21	5	8	2	6	9	0	9	10	11

เปรียบเทียบเวลาการเรียงลำดับ

ข้อมูลเริ่มต้นเรียงลำดับอยู่แล้ว (เวลาเป็น ms)

n	เลือก	ฟอง	แทรก	เชลล์
200	0.44	0.01	0.01	0.03
400	1.80	0.01	0.02	0.06
800	7.01	0.02	0.03	0.18
1,600	28.59	0.04	0.06	0.29
3,200	112.64	0.07	0.11	0.73
6,400	459.64	0.14	0.23	1.54
12,800	1814.66	0.28	0.45	3.48
25,600	7240.25	0.57	1.00	12.10
51,200	29179.25	1.25	1.91	17.31
102,400	123337.25	2.64	3.91	61.43

เปรียบเทียบเวลาการเรียงลำดับ

ข้อมูลเริ่มต้นเรียงกลับลำดับ (เวลาเป็น ms)

n	เลือก	ฟอง	แทรก	เชลล์
200	0.44	0.81	0.55	0.06
400	1.80	3.23	2.29	0.15
800	7.36	12.86	9.73	0.29
1,600	28.43	51.01	35.18	0.64
3,200	112.62	210.30	141.22	1.36
6,400	458.97	844.04	573.99	4.57
12,800	1816.83	3349.86	2266.79	6.55
25,600	7250.50	13449.50	8988.00	15.49
51,200	29249.75	53754.75	36625.25	31.10
102,400	124168.50	221318.25	153943.75	102.35

เปรียบเทียบเวลาการเรียงลำดับ

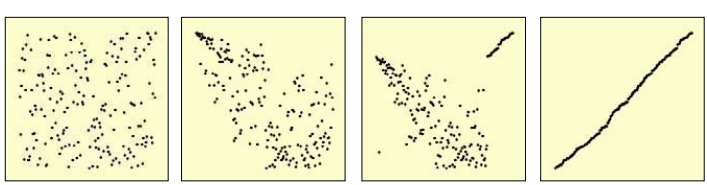
ข้อมูลเริ่มต้นเป็นแบบสุ่ม(เวลาเป็น ms)

n	เลือก	ฟอง	แทรก	เชลล์
200	0.45	0.19	0.30	0.08
400	1.82	3.13	1.10	0.19
800	7.02	15.38	4.38	0.45
1,600	31.01	49.71	16.93	1.10
3,200	112.86	207.47	74.17	2.49
6,400	452.57	823.90	284.56	5.83
12,800	1834.78	3318.25	1147.65	12.11
25,600	7273.00	13349.25	4639.25	27.83
51,200	30183.50	56241.00	20123.75	62.43
102,400	200588.50	332638.25	171231.25	184.83

การเรียงลำดับแบบฮีป (Heap Sort)

- รับอาเรย์มาสร้างให้เป็นฮีปมากสุด
- เข้าวงวน dequeue ข้อมูล (ตัวมากสุด) เพื่อนำไปไว้ ณ ตำแหน่งหลังสุดของกลุ่ม

0	1	2	3	4
25	12	10	11	27



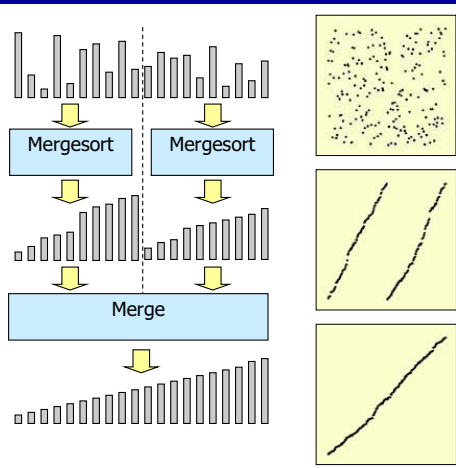
การเรียงลำดับแบบฮีป : โปรแกรม

```
public static void heapSort(Object[] d) {
  int size = d.length;
  for(int k=size/2-1; k>=0; k--)
    fixDown(d, size, k);
  for(int k=size-1; k>0; k--) {
    swap(d, 0, k);
    fixDown(d, --size, 0);
  }
}
static void fixDown(Object[] d, int size, int k) {
  int c;
  while ((c = 2 * k + 1) < size) {
    if (c < size-1 && lessThan(d[c], d[c+1])) c++;
    if (!lessThan(d[k], d[c])) break;
    swap(d, c, k);
    k = c;
  }
}
```

$O(n)$

$O(n \log n)$

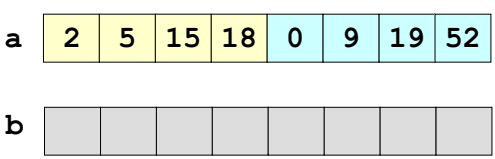
การเรียงลำดับแบบผสาน (Merge Sort)



การเรียงลำดับแบบผสาน : โปรแกรม

```
public static void mergeSort(Object[] d) {
  mSortR(d, 0, d.length-1,(Object[]) d.clone());
}

private static
void mSortR(Object[] d,
            int left, int right, Object[] t) {
  if (left < right) {
    int m = (left + right)/2;
    mSortR(t, left, m, d);
    mSortR(t, m + 1, right, d);
    merge(t, left, m, right, d);

  }
}
```

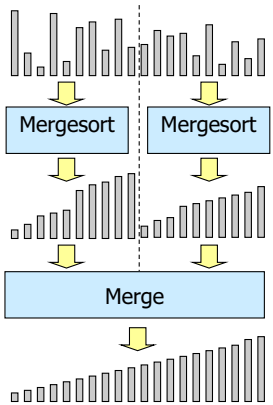

การผสาน (merge)




```
static void merge(Object[] a, int left, int mid,
                  int right, Object[] b) {
  int i = left, j = mid+1;
  for (int k = left; k <= right; k++) {
    if (i > mid)    {b[k] = a[j++]; continue;}
    if (j > right) {b[k] = a[i++]; continue;}
    b[k] = lessThan(a[i], a[j]) ? a[i++] : a[j++];
  }
}
```

การเรียงลำดับแบบผสาน : วิเคราะห์

ให้ $c(n)$ คือจำนวนครั้งของการเปรียบเทียบเพื่อเรียงลำดับข้อมูล n ตัว



$c(n/2) + c(n/2)$

$c(n) \geq 2c(n/2) + n/2$

$c(n) \leq 2c(n/2) + n - 1$

$n/2$ ถึง $n - 1$

ขอบเขตบนของจำนวนการเปรียบเทียบ

$$
\begin{aligned}
c(n) &\leq 2c(n/2) + n - 1 \\
&\leq 2(\, 2c(n/4) + n/2 - 1\,) + n - 1 \\
&= 4c(n/4) + n - 2 + n - 1 \\
&\leq 4(\, 2c(n/8) + n/4 - 1\,) + n - 2 + n - 1 \\
&= 8c(n/8) + n - 4 + n - 2 + n - 1 \\
&\quad \ldots \\
&\leq 2^k c(n/2^k) + n - 2^{k-1} + \ldots + n - 2 + n - 1 \\
&= 2^k c(n/2^k) + nk - (2^k - 1) \\
&= n \log_2 n - n + 1 \\
&= O(n \log n)
\end{aligned}
$$

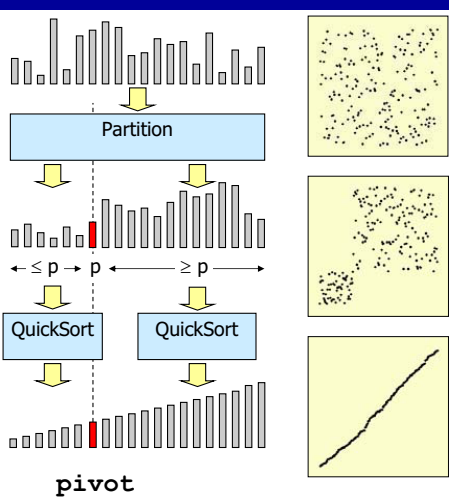
$c(1) = 0$

ให้ $n = 2^k$, $k = \log_2 n$

เวลาการทำงาน

- ขอบเขตบน : $c(n) \leq 2c(n/2) + (n-1) = O(n \log n)$
- ขอบเขตล่าง : $c(n) \geq 2c(n/2) + n/2 = \Omega(n \log n)$
- จำนวนการเปรียบเทียบ = $\Theta(n \log n)$
- จำนวนการย้าย $g(n) = 2g(n/2) + n = \Theta(n \log n)$
- เวลาการทำงาน = $\Theta(n \log n)$

การเรียงลำดับแบบเร็ว (Quick Sort)



การเรียงลำดับแบบเร็ว : โปรแกรม

```
public static void quickSort(Object[] d) {
  qSortR(d, 0, d.length-1);
}
static void qSortR(Object[] d, int left, int right){
  if (left < right) {
    int j = partition(d, left, right);
    qSortR(d, left, j - 1);
    qSortR(d, j + 1, right);
  }
}
```

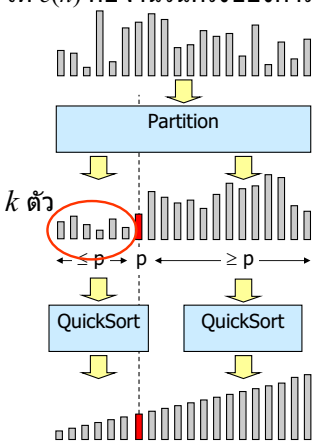

การแบ่งส่วน (partition)

d	12	5	15	18	0	9	11	52

```
static int partition(Object[] d, int left, int right) {
  Object p = d[left];
  int i = left, j = right + 1;
  while (i < j) {
    while (lessThan(p, d[--j]));
    while (lessThan(d[++i], p)) if (i == right) break;
    if (i < j) swap(d, i, j);
  }
  swap(d, left, j);
  return j;
}
```

การเรียงลำดับแบบเร็ว : วิเคราะห์

ให้ $c(n)$ คือจำนวนครั้งของการเปรียบเทียบเพื่อเรียงลำดับข้อมูล n ตัว



$n - 1$

$c(n) = c(k) + c(n - k - 1) + n - 1$

$c(k) + c(n - k - 1)$

จำนวนการเปรียบเทียบ : กรณีเร็วสุด

$$c(n) = c(k) + c(n - k - 1) + n - 1$$

$$\begin{aligned} c_{min}(n) &= c_{min}(\lfloor n/2 \rfloor) + c_{min}(n - \lfloor n/2 \rfloor - 1) + n - 1 \\ &\le 2c_{min}(n/2) + n - 1 \\ &= n \log_2 n - n + 1 \\ &= O(n \log n) \end{aligned}$$

จำนวนการเปรียบเทียบ : กรณีช้าสุด

$$c(n) = c(k) + c(n-k-1) + n - 1$$

$$\begin{aligned} c_{max}(n) &= c(0) + c_{max}(n-1) + n - 1 \\ &= c_{max}(n-1) + n - 1 \\ &= c_{max}(n-2) + n - 2 + n - 1 \\ &\ldots \\ &= c_{max}(1) + 1 + 2 + \ldots + n - 2 + n - 1 \\ &= n(n-1)/2 \\ &= \Theta(n^2) \end{aligned}$$

จำนวนการเปรียบเทียบ : กรณีเฉลี่ย

$$c(n) = c(k) + c(n-k-1) + n - 1$$

$$\begin{aligned} c_{\text{avg}}(n) &= \frac{1}{n}\sum_{k=0}^{n-1}\left(c_{\text{avg}}(k) + c_{\text{avg}}(n-k-1)\right) + (n-1) \\ &= \frac{2}{n}\sum_{k=0}^{n-1} c_{\text{avg}}(k) + (n-1) \end{aligned}$$

$$nc_{\text{avg}}(n) = 2\sum_{k=0}^{n-1} c_{\text{avg}}(k) + n(n-1)$$

$$(n-1)c_{\text{avg}}(n-1) = 2\sum_{k=0}^{n-2} c_{\text{avg}}(k) + (n-1)(n-2)$$

$$nc_{\text{avg}}(n) = (n+1)c_{\text{avg}}(n-1) + 2(n-1)$$

จำนวนการเปรียบเทียบ : กรณีเฉลี่ย

$$nc_{\text{avg}}(n) = (n+1)c_{\text{avg}}(n-1) + 2(n-1)$$

$$\begin{aligned}
\frac{c_{\text{avg}}(n)}{n+1} &= \frac{c_{\text{avg}}(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \\
&= \frac{c_{\text{avg}}(1)}{2} + 2\sum_{i=2}^{n} \frac{(i-1)}{i(i+1)} \\
&\approx 2\sum_{i=2}^{n} \frac{1}{(i+2)} \\
&= 2(\ln n + \mathrm{O}(1))
\end{aligned}$$

$$\begin{aligned}
c_{\text{avg}}(n) &= 2n\ln n + \mathrm{O}(n) \\
&\approx 1.39 n \log_2 n + \mathrm{O}(n) \\
&= \mathrm{O}(n \log n)
\end{aligned}$$

การเลือกตัวหลัก (pivot)

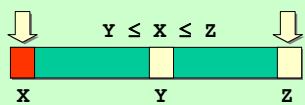
- ที่ผ่านมา : เลือกตัวซ้ายสุดเป็นตัวหลัก
- ถ้าข้อมูลเริ่มต้นเรียงลำดับอยู่แล้ว
  - หลังการแบ่งส่วน ชุดซ้ายมี 0 ตัวเสมอ
  - เกิดกรณีช้าสุด : $\Theta(n^2)$
  - ถ้าใช้โปรแกรมที่เขียนแบบเวียนเกิด
  - เกิดการเรียกเวียนเกิดซ้อน ๆ กันจำนวน n – 1 ครั้ง
  - มีโอกาสเกิด StackOverflowException
- สุ่มเลือกตัวหลัก
  - โอกาสเกิดกรณีช้าสุดมีน้อยมาก ๆ

```
static int partition(Object[] d, int left, int right) {
  int m = left + (int)(Math.random()*(right-left+1));
  swap(d, left, m);
  Object p = d[left];
  ...
```

การเลือกมัธยฐานสามเป็นตัวหลัก

- ใช้ตัวหลักที่ได้มาจากมัธยฐานของตัวซ้าย ขวา และกลางของข้อมูล

```
static int partition(Object[] d, int left, int right) {
  int c = (left + right)/2;
  if (lessThan(d[left],d[c])) swap(d, left, c);
  if (lessThan(d[right],d[c])) swap(d, c, right);
  if (lessThan(d[right],d[left])) swap(d, left, right);
  Object p = d[left];
  int i = left, j = right + 1;
  while (i < j) {
    while (lessThan(p, d[--j]));
    while (lessThan(d[++i], p)) if (i==right) break;
    if (i < j) swap(d, i, j);
  }
  swap(d, left, j);
  return j;
}
```



เปรียบเทียบเวลาเรียงลำดับ

ข้อมูลเริ่มต้นเรียงลำดับ (เวลาเป็น ms)

n	เชลล์	ฮีป	ผสาน	เร็ว	เร็ว (สุ่ม)	เร็ว (M3)
1000	0.20	0.96	0.38	11.66	0.48	0.29
10000	2.96	16.74	4.77	-	5.94	4.38
100000	64.67	168.72	58.99	-	67.60	46.87
1000000	1153.70	2005.57	724.59	-	787.78	503.23

เปรียบเทียบเวลาเรียงลำดับ

ข้อมูลเริ่มต้นเรียงกลับลำดับ (เวลาเป็น ms)

n	เชลล์	ฮีป	ผสาน	เร็ว	เร็ว (สุ่ม)	เร็ว (M3)
1000	0.40	0.77	0.39	10.51	0.49	0.30
10000	6.45	11.51	9.86	-	6.23	5.95
100000	113.26	154.61	60.88	-	71.68	48.66
1000000	1692.68	1997.14	745.68	-	809.86	541.56

เปรียบเทียบเวลาเรียงลำดับ

ข้อมูลเริ่มต้นแบบสุ่ม (เวลาเป็น ms)

n	เชลล์	ฮีป	ผสาน	เร็ว	เร็ว (สุ่ม)	เร็ว (M3)
1000	0.97	0.96	0.56	0.58	0.94	0.52
10000	12.82	15.53	10.27	6.88	15.37	8.89
100000	199.27	224.27	111.66	106.92	121.83	102.17
1000000	3143.03	4303.59	1682.07	1591.25	1810.50	1600.21

เปรียบเทียบเนื้อที่เสริม

- ใช้เนื้อที่เสริมเพื่อการเรียงลำดับ
  - $\Theta(1)$: selection, bubble, insertion, Shell, heap, quick
  - $\Theta(n)$: merge
- ใช้เนื้อที่เสริมเพราะเขียนแบบเวียนเกิด
  - quick : น้อยสุด $\Theta(\log n)$, มากสุด $\Theta(n)$
  - merge : $\Theta(\log n)$
  - สามารถเขียน quicksort และ mergesort แบบวนวน (ไม่ต้อง recursive) ได้

สรุป

- ไม่ควรใช้การเรียงลำดับแบบฟอง
- การเรียงลำดับแบบเลือกย้ายข้อมูลน้อยครั้งสุด
- ถ้าข้อมูลไม่มาก น่าใช้การเรียงลำดับแบบแทรก
- การเรียงลำดับแบบเชลล์เร็วมาก
- การเรียงลำดับแบบผสานเร็วกว่า ถ้าข้อมูลมาก แต่ต้องการเนื้อที่เสริม
- การเรียงลำดับแบบฮีปเป็น O(n log n) แต่ช้า
- การเรียงลำดับแบบเร็ว ต้องเขียนดี ๆ ถึงเร็ว และเร็วมาก ๆ