แถวคอยเชิงบุริมภาพ

(Priority Queues)

หัวข้อ

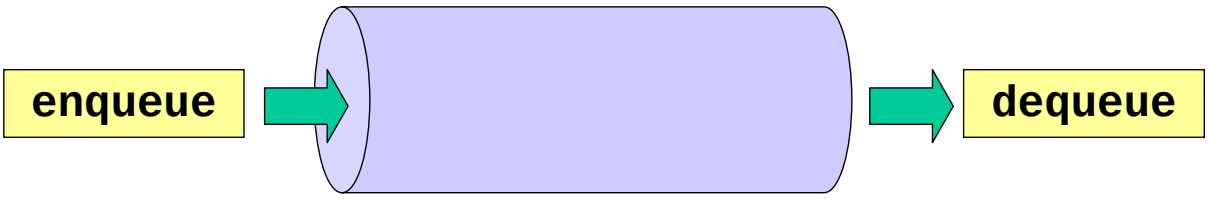
- นิยามแถวคอยเชิงบุริมภาพ
- การสร้างแถวคอยเชิงบุริมภาพด้วยฮีปแบบทวิภาค
- ตัวอย่างการใช้งานแถวคอยเชิงบุริมภาพ
  - การเรียงลำดับแบบฮีป
  - การเลือกข้อมูลตามอันดับ
  - รหัสฮัฟฟ์แมน

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));
```

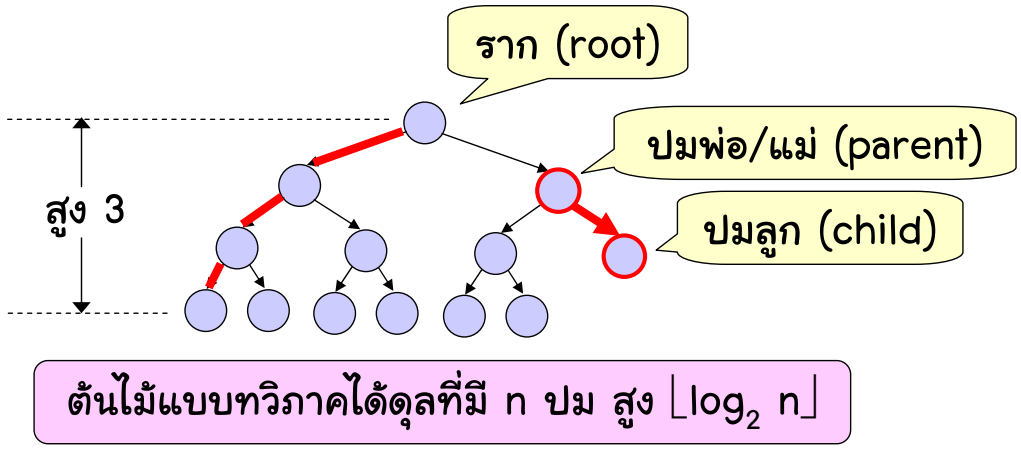
3

-3

0

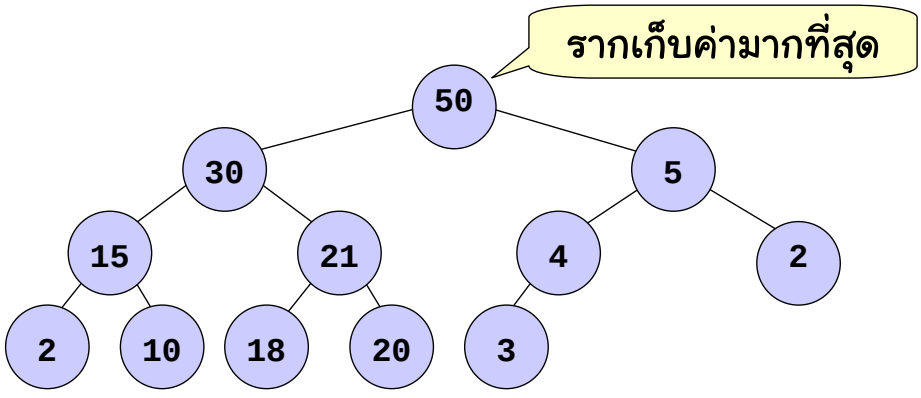
ฮีปแบบทวิภาค (Binary Heap)

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

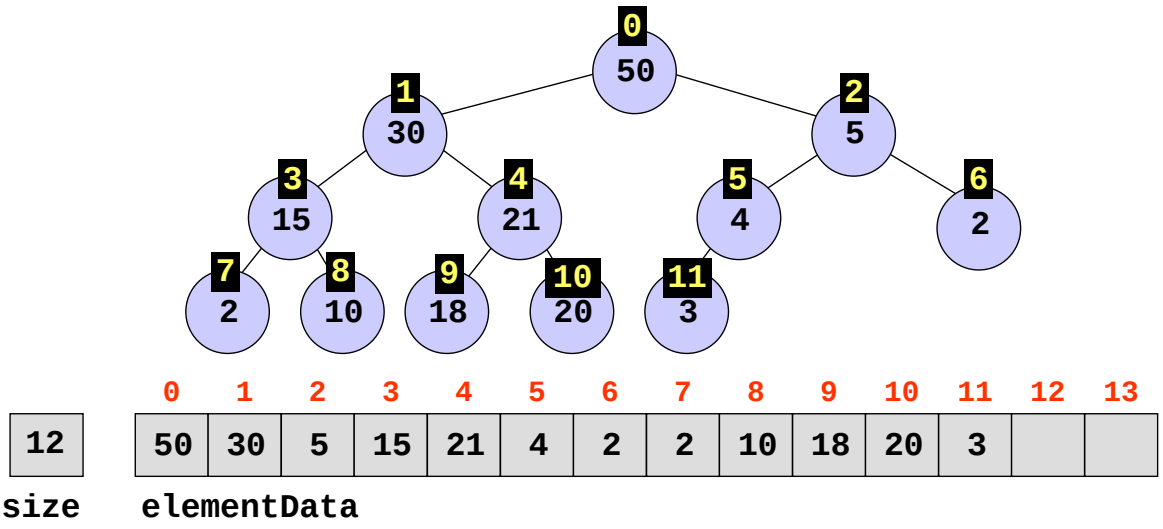


ฮีปแบบทวิภาค

- 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

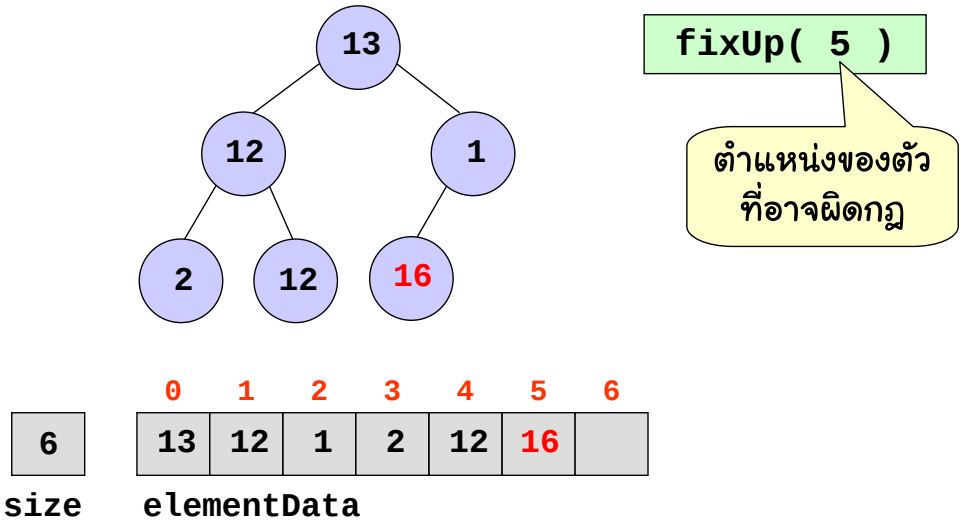
คลาส BinaryHeap

```
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



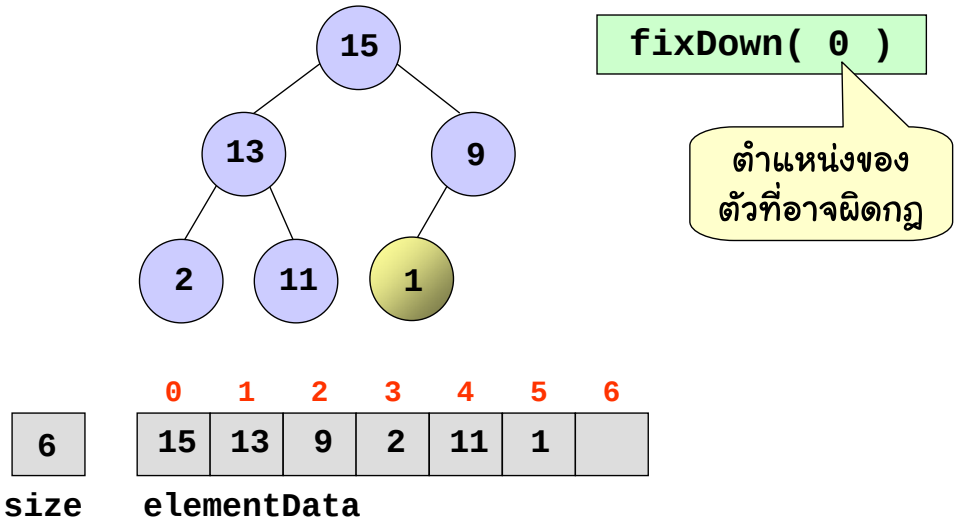
เมท็อด enqueue(e)

```
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;
}
```

สลับข้อมูลกับปมพ่อขึ้นไป
เรื่อย ๆ จนกว่าจะไม่
สำคัญกว่าของพ่อ

dequeue( ) : การทำงาน

- เก็บรากไว้เป็นคำตอบ
- ย้ายข้อมูลที่ใบล่างขวาสุด มาเก็บที่ราก
- สลับข้อมูลที่รากลงมา จนกว่าพ่อจะไม่น้อยกว่าลูก



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;
  }
}
```

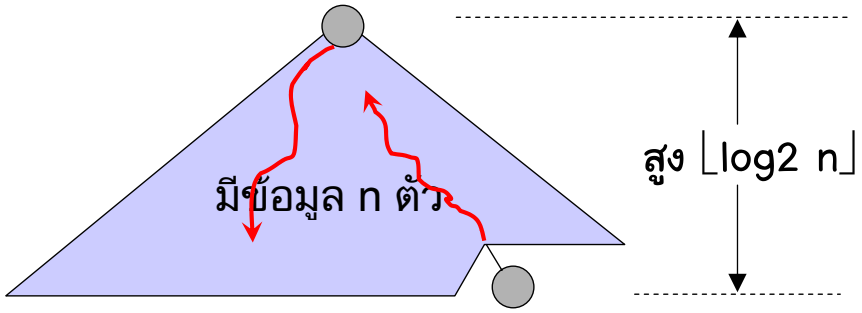
ตราบเท่าที่ยังมีลูกซ้าย

ถ้ามีลูกขวาและขวา > ซ้าย

เลิก เมื่อลูกไม่มากกว่าพ่อ

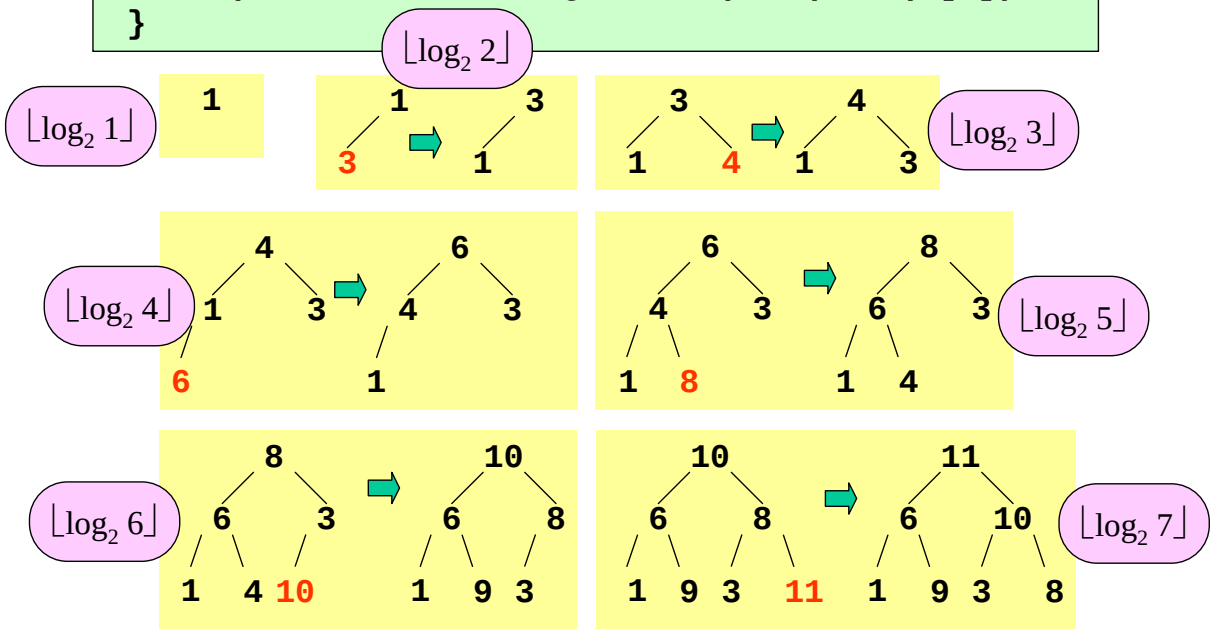
เวลาการทำงาน

- 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]);
}
```

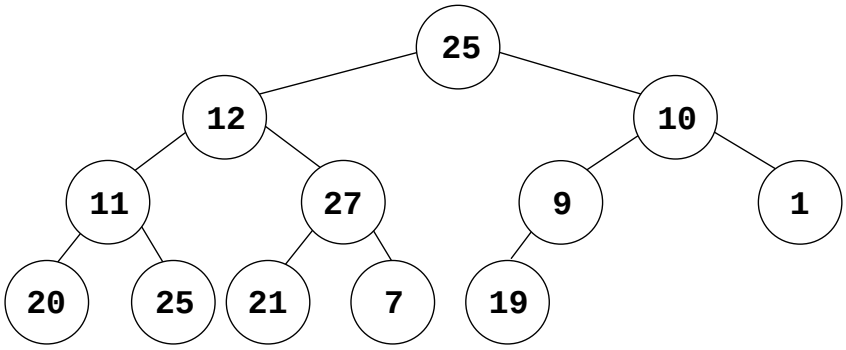


$$\leq \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \lfloor \log_2 3 \rfloor + \ldots + \lfloor \log_2 n \rfloor < \log_2 n! = O(n \log n)$$

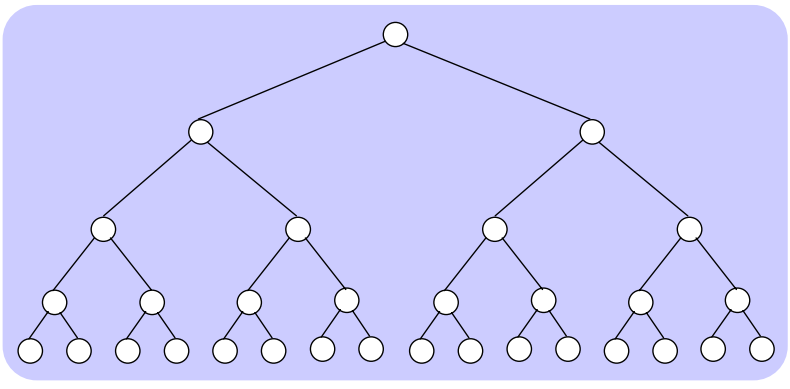
การสร้างฮีปแบบทวิภาคด้วยการค่อย ๆ ปรับ

```
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

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

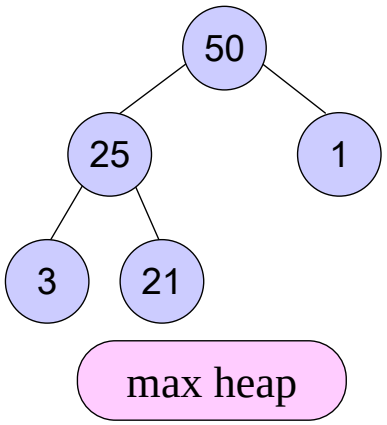
$$\leq \sum_{k=0}^{h} k2^{h-k}$$

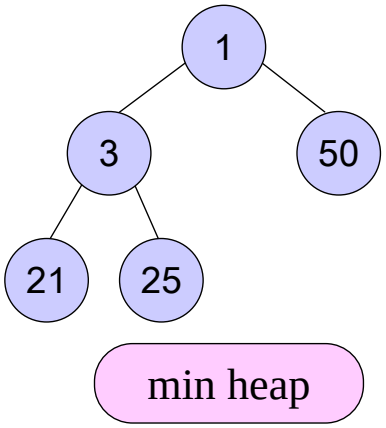
fixdown ในต้นไม้สูง k เกิดการสลับข้อมูลไม่เกิน k ครั้ง

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

ฮีปมากสุด / ฮีปน้อยสุด (Max/Min Heap)

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





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

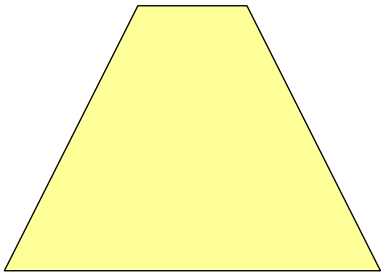
$O(n)$

$O(k \log n)$

```
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

100001000101010001101

ส ม ช า ย ม า

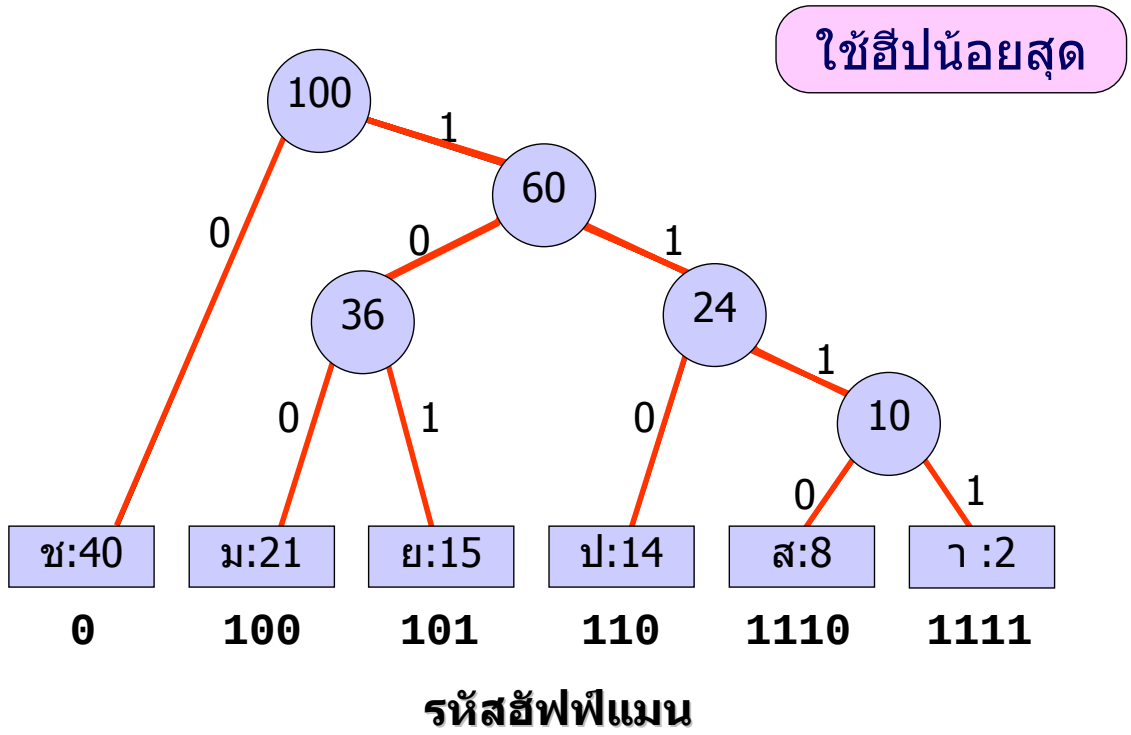
111010001111011001111

ส ม ช า ย ม า

40×3 + 21×3 + 15×3 + 14×3 + 8×3 + 2×3 = 300

40×1 + 21×3 + 15×3 + 14×3 + 8×4 + 2×4 = 230

วิธีการหารหัสฮัฟฟ์แมน



รหัสฮัฟฟ์แมน

สรุป

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