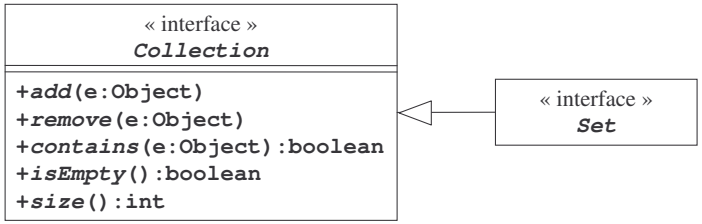
2110211 โครงสร้างข้อมูลเบื้องต้น

Binary Search Tree

สมชาย ประสิทธิ์จูตระกูล

กลับมาเรื่อง set

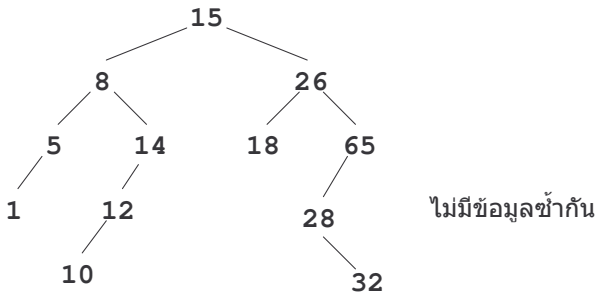



```
public interface Set extends Collection {

  /**
   * add a new element without duplication.
   *
   * @param element a new element
   */
  public void add(Object element);
}
```


สร้าง set ด้วย binary search tree

- binary search tree
  - เป็น binary tree
  - เก็บข้อมูลตาม internal nodes
  - ข้อมูลในต้นไม้ย่อยทางซ้ายต้องน้อยกว่าข้อมูลที่ราก
  - ข้อมูลในต้นไม้ย่อยทางขวาต้องมากกว่าข้อมูลที่ราก
  - ต้นไม้ย่อยทุก ๆ ต้นต้องเป็น binary search tree ด้วย

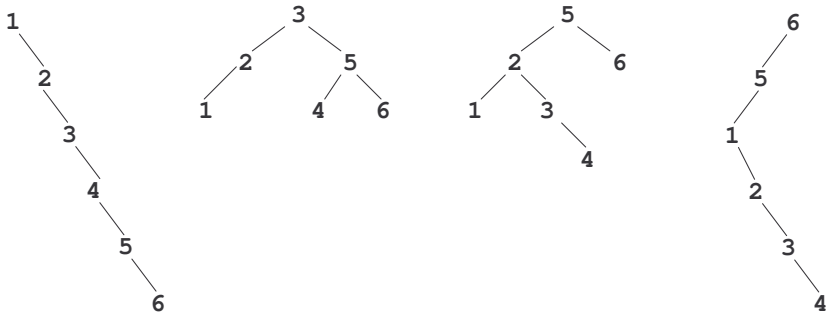


ไม่มีข้อมูลซ้ำกัน

ความสูงของ binary search tree

- ต้นไม้ bst ที่เก็บข้อมูลเหมือนกัน อาจมีรูปร่างต่างกัน
- bst ที่เก็บข้อมูลเหมือนกัน อาจมีความสูงต่างกัน

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



class BSTSet

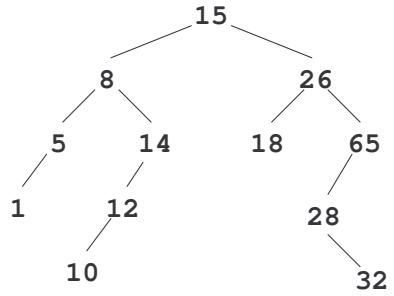
```
public class BSTSet implements Set {
  private static class Node {
    Object element;
    Node left;
    Node right;
    Node(Object e, Node l, Node r) {
      this.element = e;
      this.left = l;
      this.right = r;
    }
  }

  private Node root;
  private int size;

  public BSTSet() { }
  ...
```


contains(e)

- เริ่มการค้นที่ราก
  - ถ้า e มีค่าน้อยกว่าค่าที่ราก ก็ต้องลงไปค้นต่อทางซ้าย
  - ถ้า e มีค่ามากกว่าค่าที่ราก ก็ต้องลงไปค้นต่อทางขวา
  - ถ้า e มีค่าเท่ากับค่าที่ราก ก็แสดงว่าค้นพบแล้ว
  - ถ้าเจอ external nodes (null) ก็แสดงว่าค้นไม่พบ

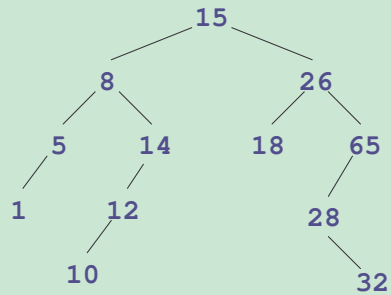


ออปเจกต์ที่เก็บต้องเป็นแบบ `Comparable`

contains(e)

```
public class BSTSet implements Set {
  private Node root;
  ...
  public boolean contains(Object e) {
    Node r = root;
    while( r != null ) {
      int cmp = ((Comparable) r.element).compareTo(e);
      if (cmp > 0)
        r = r.left;
      else if (cmp < 0)
        r = r.right;
      else
        return true;
    }
    return false;
  }
  ...
```

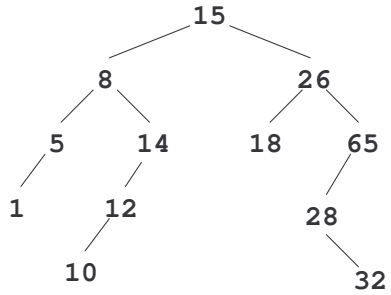
O(h)



contains(e) แบบ recursive

```
public class BSTSet implements Set {
  private Node root;
  ...
  public boolean contains(Object e) {
    return contains(root, e);
  }
  private static boolean contains(Node r, Object e) {
    if (r == null) return false;
    int cmp = ((Comparable) r.element).compareTo(e);
    if (cmp > 0)
      return contains(r.left, e);
    else if (cmp < 0)
      return contains(r.right, e);
    else
      return true;
  }
  ...
```

getMin(), getMax()



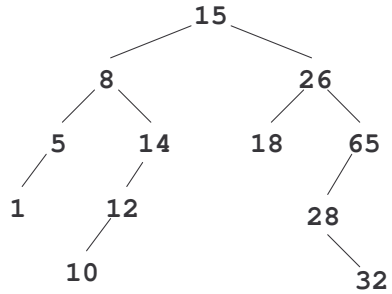
getMin(), getMax()

O(h)

```
public class BSTSet implements Set {
  private Node root;
  ...
  public Object getMin() {  // return null, if empty
    Node r = root;
    if (r == null) return null;
    while (r.left != null) r = r.left;
    return r.element;
  }

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


Inorder ใน Binary Search Trees

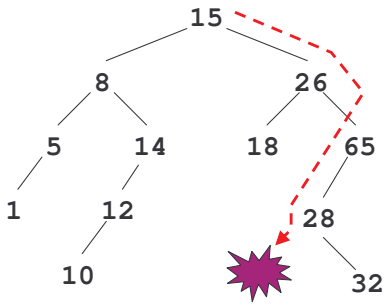


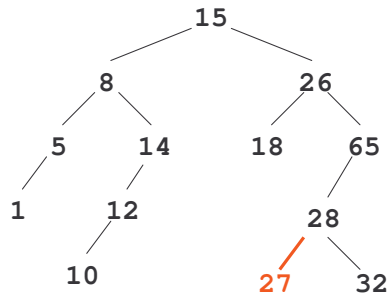
1, 5, 8, 14, 12, 10, 15, 18, 26, 65, 28, 32

เรียงจากน้อยไปมาก

การเพิ่มข้อมูล

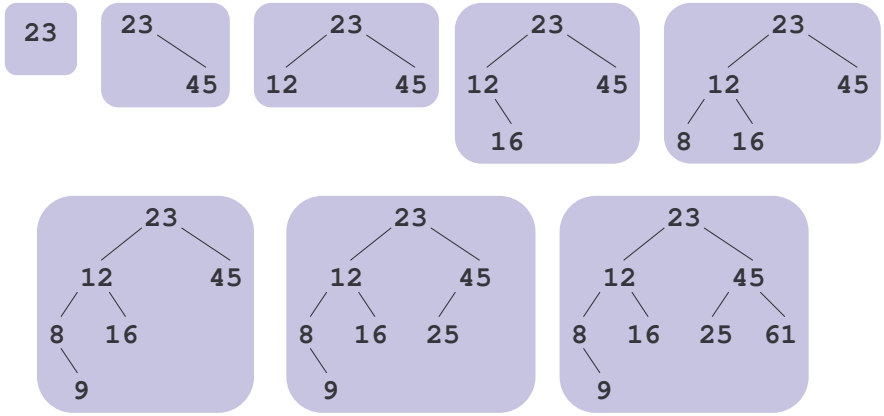
- อยากเพิ่ม 27
- ลองค้น 27, ต้องไม่พบที่ตำแหน่ง null หนึ่งในต้นไม้
- เพิ่ม 27 แทน null ณ ตำแหน่งนั้น





ตัวอย่างการเพิ่มข้อมูล

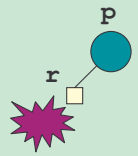
- เริ่มด้วยต้นไม้ว่าง แล้วเพิ่ม
- 23, 45, 12, 16, 8, 9, 25, 61



add(e)

O(h)

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



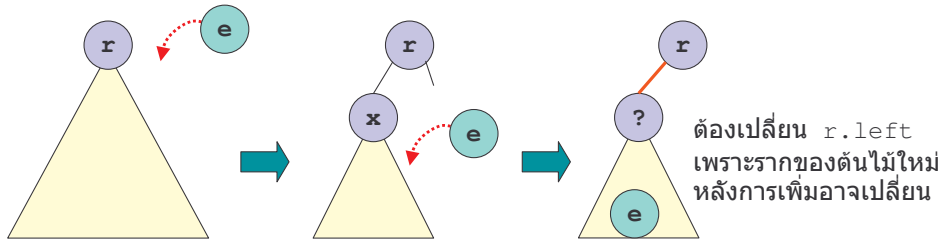
add(e) : แบบ recursive

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

เรียก `add(r,e)` เป็นการเพิ่ม `e` ในต้นไม้ที่มี `r` เป็นราก ผลที่ได้คือรากของต้นไม้ใหม่หลังการเพิ่ม

add(r, e)

`if (e < r)  r.left = add( r.left, e)`

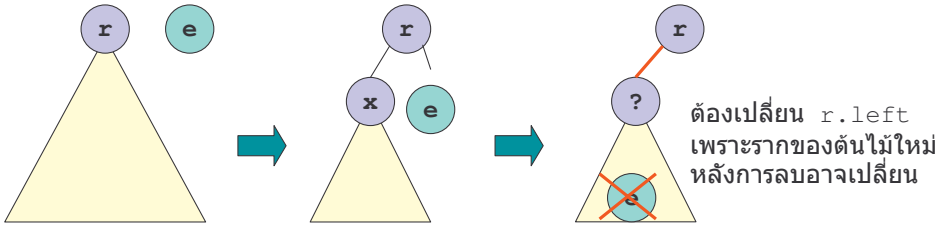


ก่อนเพิ่ม — หลังเพิ่ม

`add(r, e)` `add(r.left, e)` `r.left = add(r.left, e)`

การลบข้อมูล : remove(r, e)

```
if (e < r)  r.left = remove( r.left, e)
```



ต้องเปลี่ยน `r.left` เพราะรากของต้นไม้ใหม่หลังการลบอาจเปลี่ยน

ก่อนลบ

`remove(r, e)`

`remove(r.left, e)`

หลังเพิ่ม

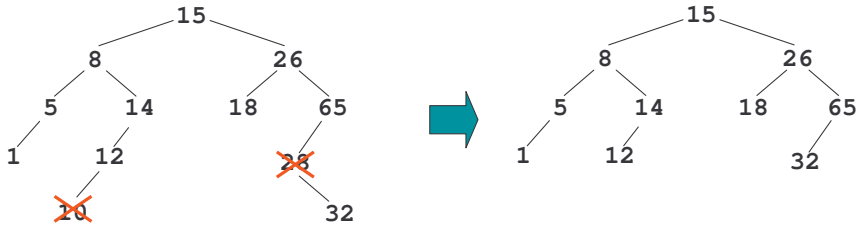
`r.left = remove(r.left, e)`

remove(e) : แบบ recursive

```
public void remove(Object e) {
  root = remove(root, e);
}
private Node remove(Node r, Object e) {
  if (r != null) {
    int cmp = ((Comparable) r.element).compareTo(e);
    if (cmp > 0) {
      r.left = remove(r.left, e);
    } else if (cmp < 0) {
      r.right = remove(r.right, e);
    } else {

       // พบ node ที่จะลบแล้ว การลบเกิดขึ้นที่นี่
    }
  }
  return r;
}
```

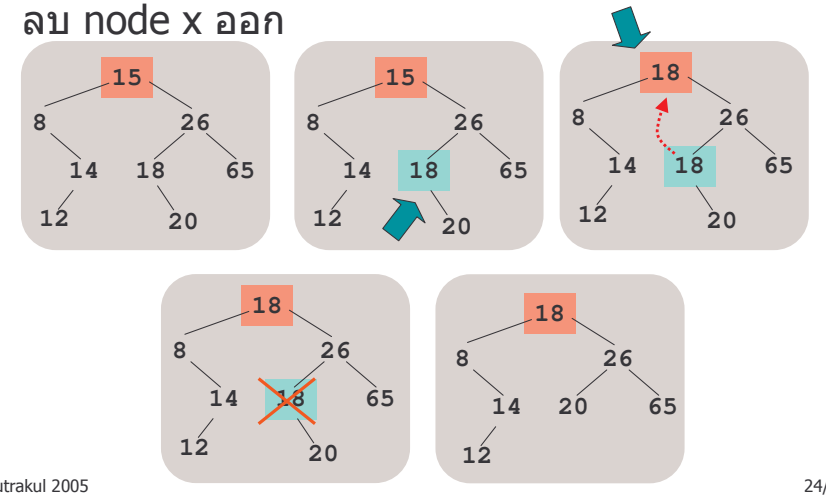

การลบ e เมื่อ e มี 1 หรือ 2 ลูกที่เป็น null




```
if (cmp > 0) {
  r.left = remove(r.left, e);
} else if (cmp < 0) {
  r.right = remove(r.right, e);
} else {
  if (r.left == null) r = r.right;
  else if (r.right == null) r = r.left;
  else {
     ...  // two children
  }
}
```


การลบ e เมื่อลูกทั้งสองของ e ไม่เป็น null

- หา x ซึ่งเป็น node ที่เก็บค่าน้อยสุดในต้นไม้ขวาของ e
- ย้ายค่าใน x นั้นมาไว้แทนที่ใน node ของ e
- ลบ node x ออก



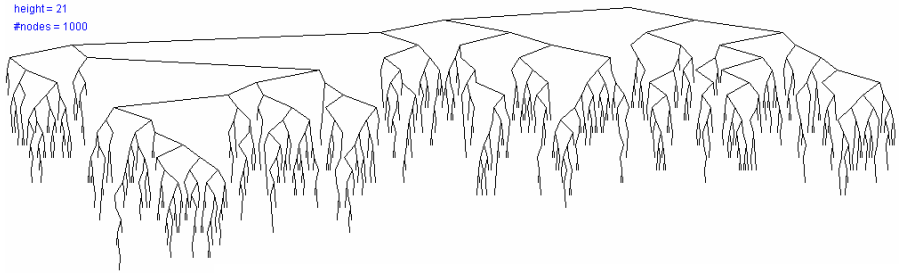
```
public void remove(Object e) {
  root = remove(root, e);
}
private Node remove(Node r, Object e) {
  if (r != null) {
    int cmp = ((Comparable) r.element).compareTo(e);
    if (cmp > 0) {
      r.left = remove(r.left, e);
    } else if (cmp < 0) {
      r.right = remove(r.right, e);
    } else {
      if (r.left == null || r.right == null) {
        r = (r.left == null ? r.right : r.left);
        --size;
      } else {
        r.element = getMinNode(r.right).element;
        r.right = remove(r.right, r.element);
      }
    }
  }
  return r;
}
```

O(h)

```
private Node getMinNode(Node r) {
  while( r.left != null ) r = r.left;
  return r;
}
```


เวลาการทำงาน : contains, add, remove

- เป็น O(h), $n - 1 \le h \le \lfloor \log_2 n \rfloor$
- best case : O(log n), worst case : O(n)
- มีผู้แสดงให้เห็นจริงว่า bst ที่สร้างจาก<u>ข้อมูลสุ่ม</u> จำนวน n ตัว จะสูงเฉลี่ย $\approx 4.31107 \ln n \approx 3 \log_2 n$
- ถ้าไม่แน่ใจในลักษณะของข้อมูลที่เก็บใน bst
  - เวลาการทำงานไม่แน่นอน โชคร้ายเป็น O(n)

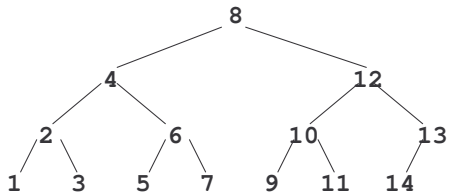


Devroye, L. 1986. A note on the height of binary search trees. *J. ACM 33*, 489–498.

การบ้าน

- เขียนเมท็อดดังต่อไปนี้ ใน BSTSet ให้สมบูรณ์
  - size, isEmpty, toString
- เขียน constructor ของ BSTSet ที่รับอาเรย์ของออปเจกต์ เพื่อสร้าง bst ที่ได้ดุล

```
int [] x = {1,2,3,5,6,4,10,11,9,8,7,14,12,13};
Integer [] d = new Integer[x.length];
for(int i=0; i<x.length; i++) d[i] = new Integer(x[i]);
BSTSet t = new BSTset(d);
```



การบ้าน

- ถ้าให้ Node เก็บจำนวน nodes ของต้นไม้ย่อยที่มี node นี้เป็นราก
- จงเขียนเมท็อด select(int k) ซึ่งคืนข้อมูลตัวที่เล็กสุดอันดับที่ k ของต้นไม้ ในเวลา O(log n)
- ที่ผ่านมา ข้อมูล e ตัวใหม่ที่เพิ่มจะอยู่ที่ใบใหม่ของต้นไม้ จงเขียน add ใหม่ที่จะทำให้ e มาอยู่ที่รากหลังการเพิ่ม