ต้นไม้ค้นหาแบบทวิภาค

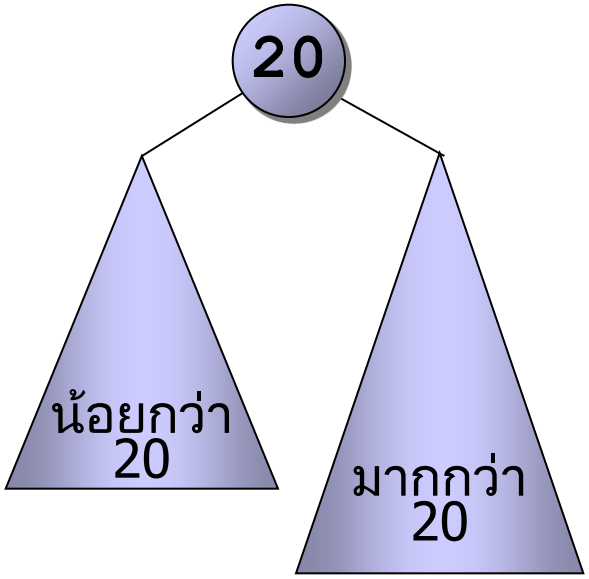
(Binary Search Tree)

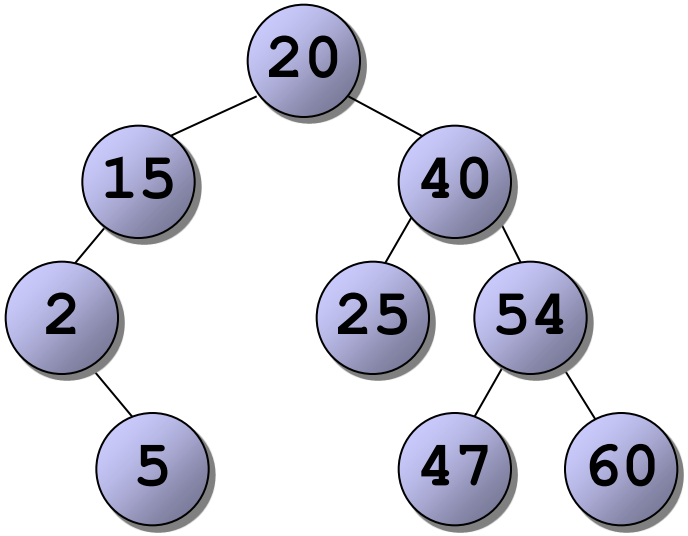
หัวข้อ

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

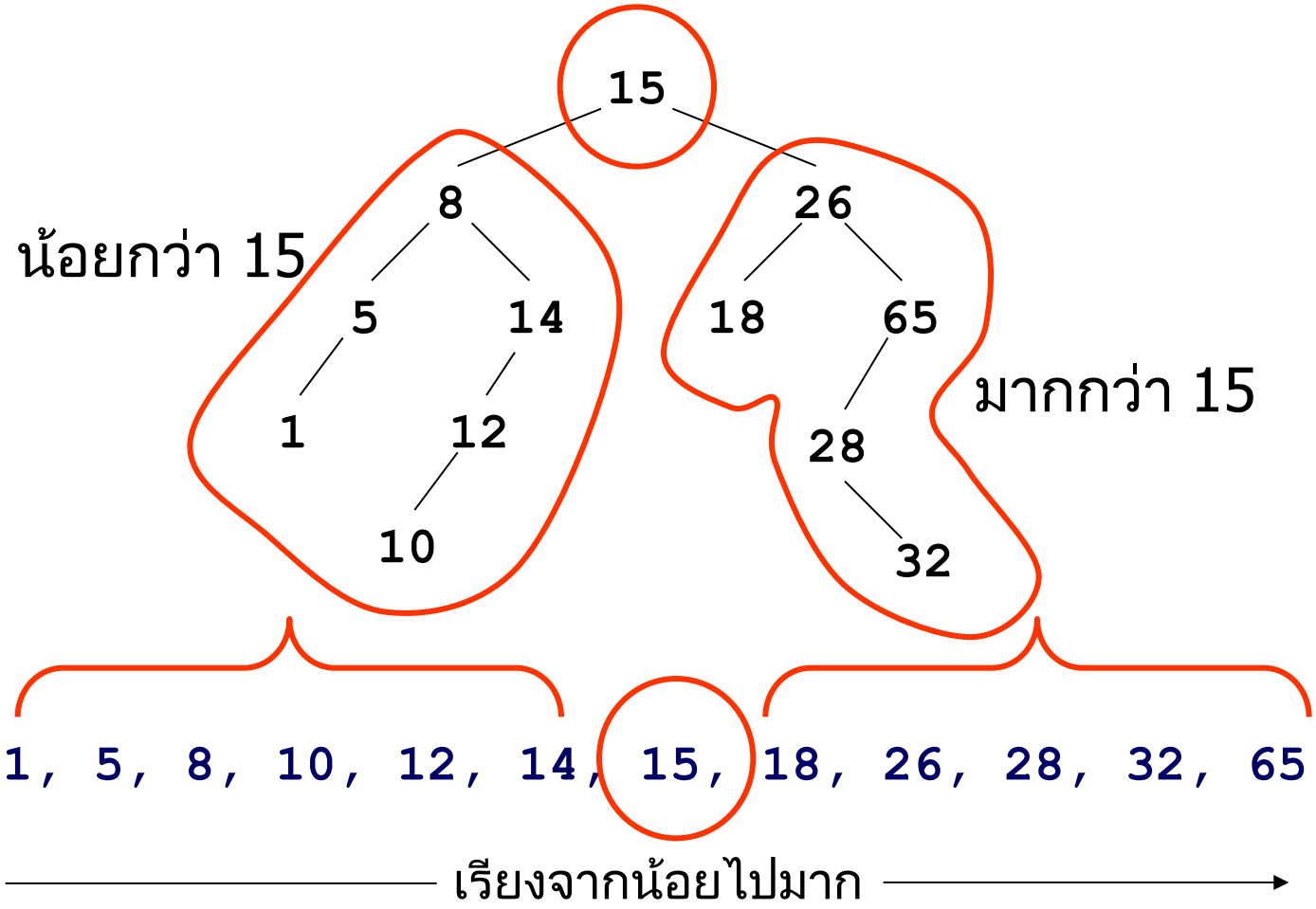
ต้นไม้ค้นหาแบบทวิภาค

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

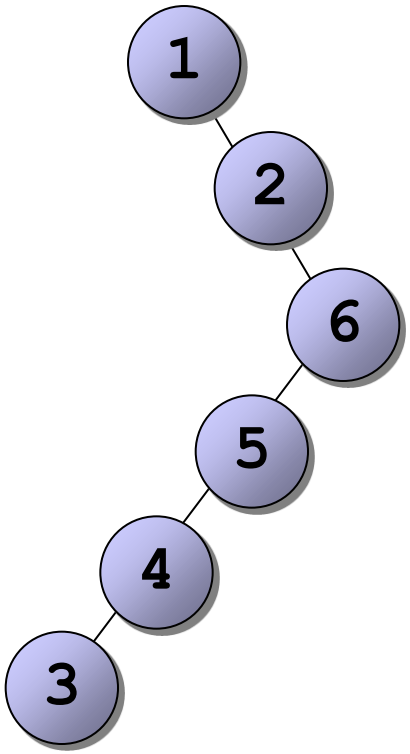


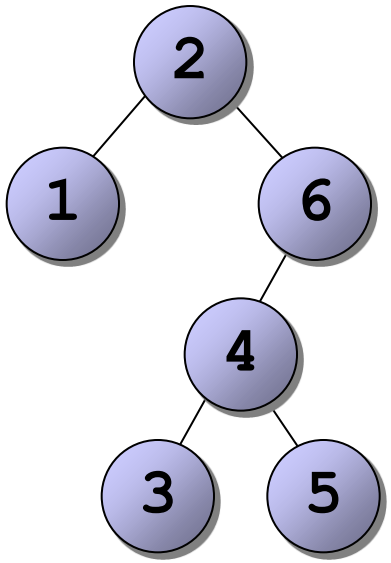


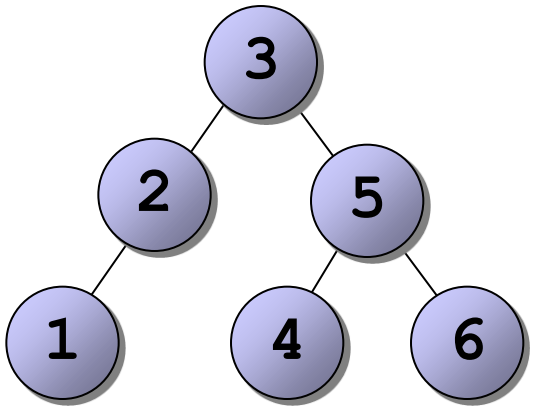
การแวะผ่านแบบตามลำดับ



ข้อมูลชุดเดียวกันเก็บได้หลายแบบ







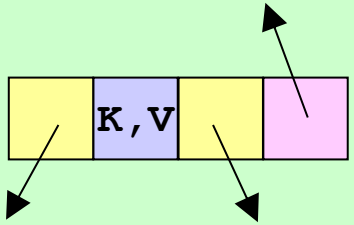
$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

map_bst

```
template <typename KeyT,
          typename MappedT,
          typename CompareT = std::less<KeyT> >
class map_bst {
protected:

  class node {
    friend class map_bst;
    ...
  };

  class tree_iterator {
    ...
  };

public:
  ...

};
```

node

```
class node {
  friend class map_bst;
protected:
  ValueT data;
  node  *left;
  node  *right;
  node  *parent;

  node() : data(ValueT()), left(NULL),
           right(NULL), parent( NULL ) { }

  node(const ValueT& data, node* left,
       node* right, node* parent) :  data (data),
       left(left), right(right), parent(parent) { }
};
```

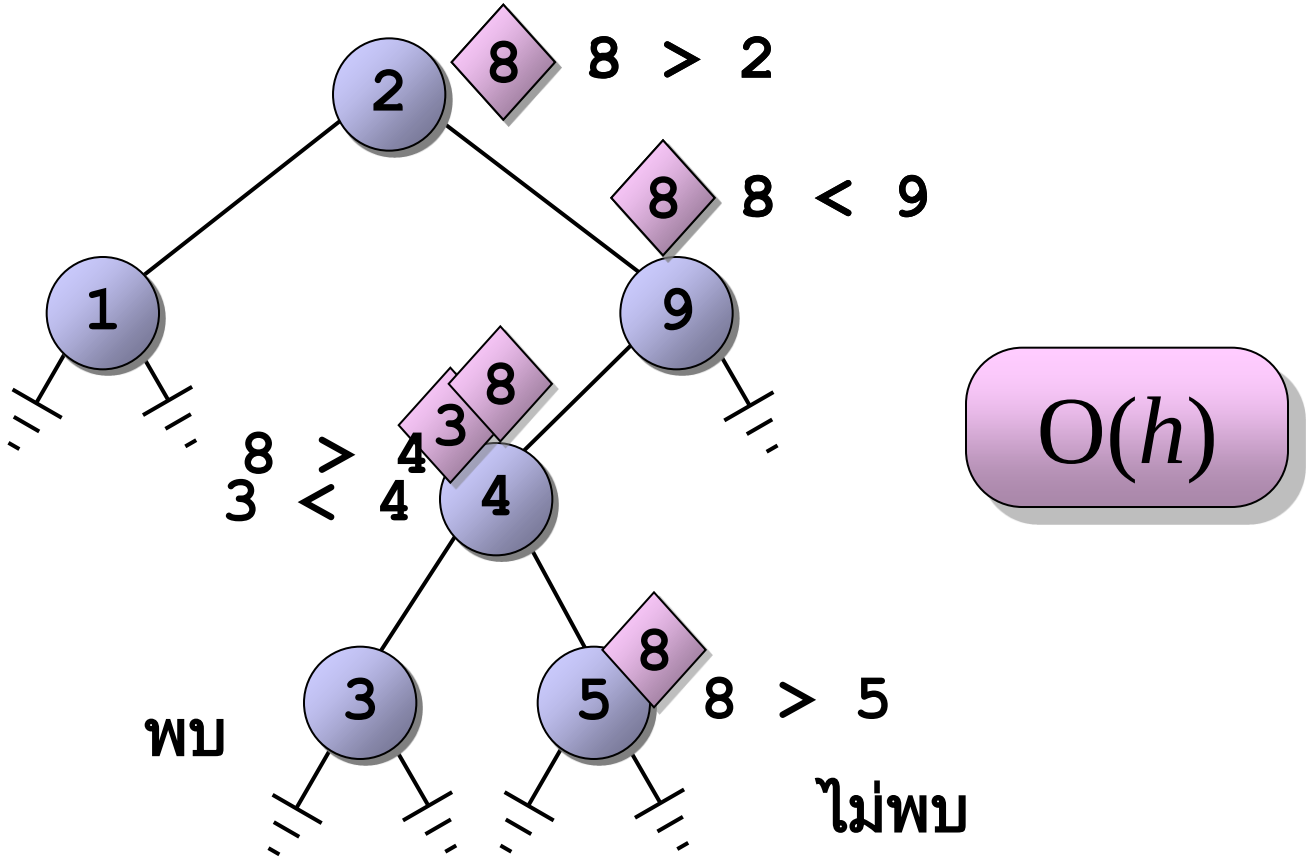


map_bst

```
class map_bst {
protected:
  node     *mRoot;
  CompareT  mLess;
  size_t    mSize;
public:
  map_bst(const map_bst<KeyT,MappedT,CompareT> & x){...}
  map_bst(const CompareT& c = CompareT() )          {...}
 ~map_bst() {...}
  map_bst<KeyT,MappedT,CompareT>&
   operator=(map_bst<KeyT,MappedT,CompareT> other) {...}
  bool     empty() { return mSize == 0;  }
  size_t   size()  { return mSize;  }
  iterator begin() { ... }
  iterator end()   { ... }
  void     clear() { ... }
  iterator find(const KeyT &key)        { ... }
  size_t   erase(const KeyT &key)       { ... }
  MappedT& operator[](const KeyT& key) { ... }
  pair<iterator,bool> insert(const ValueT& val) { ... }
```

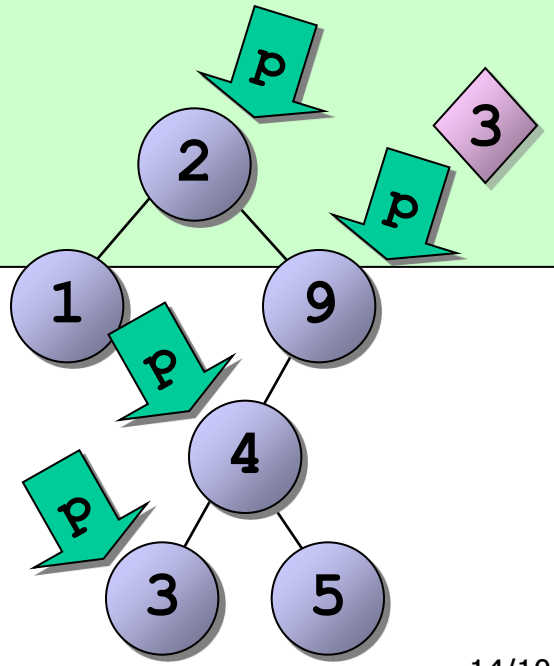
การค้นหาข้อมูล

- ใช้การแวะผ่านต้นไม้ ค่อย ๆ เปรียบเทียบ
- ใช้กฎการจัดเก็บช่วยในการค้น



การค้นหาข้อมูล

```
node* find_node(const KeyT& k, node* r, node* &parent){
  node *ptr = r;

  while (ptr != NULL) {
    if (k == ptr->data.first) return ptr;

    parent = ptr;
    ptr = (k < ptr->data.first) ?
           ptr->left : ptr->right;
  }
  return NULL;
}
```



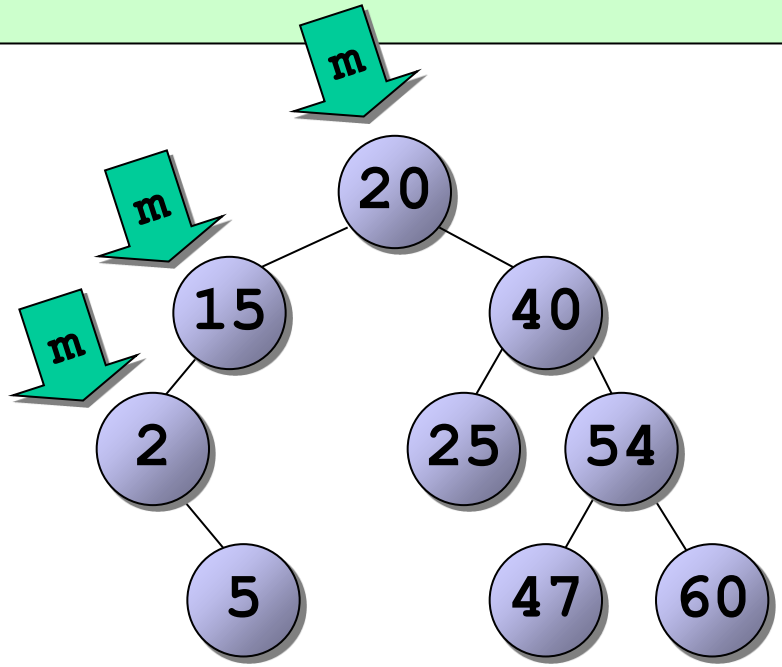
find_node

```
int compare(const KeyT& k1, const KeyT& k2) {
  if (mLess(k1, k2)) return -1;
  if (mLess(k2, k1)) return +1;
  return 0;
}
node* find_node(const KeyT& k,node* r, node* &parent){
  node *ptr = r;
  while (ptr != NULL) {
    int cmp = compare(k, ptr->data.first);
    if (cmp == 0) return ptr;
    parent = ptr;
    ptr = cmp < 0 ? ptr->left : ptr->right;
  }
  return NULL;
}
```

```
iterator find(const KeyT &key) {
  node *parent = NULL;
  node *ptr = find_node(key,mRoot,parent);
  return ptr == NULL ? end() : iterator(ptr);
}
```

find_min_node : การค้นหาข้อมูลตัวน้อยสุด

```
node* find_min_node(node* r) {
  node *min = r;
  while (min->left != NULL) {
    min = min->left;
  }
  return min;
}
```

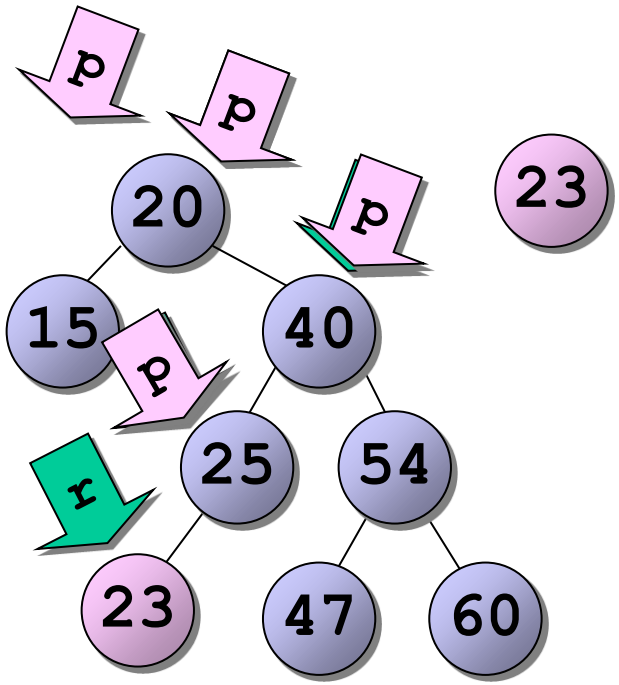


ลองเขียนการค้นตัวมากสุดเอง

find_max_node : การค้นหาข้อมูลตัวมากสุด

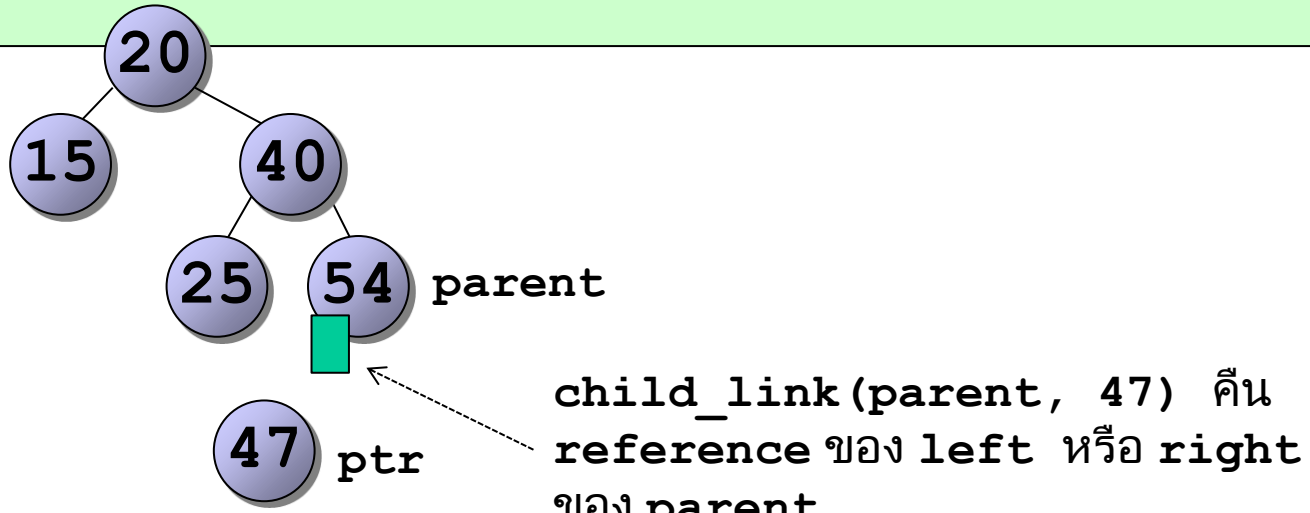
```
node* find_max_node(node* r) {
  node *max = r;
  while (max->right != NULL) {
    max = max->right;
  }
  return max;
}
```

insert: การเพิ่มข้อมูล



insert: การเพิ่มข้อมูล

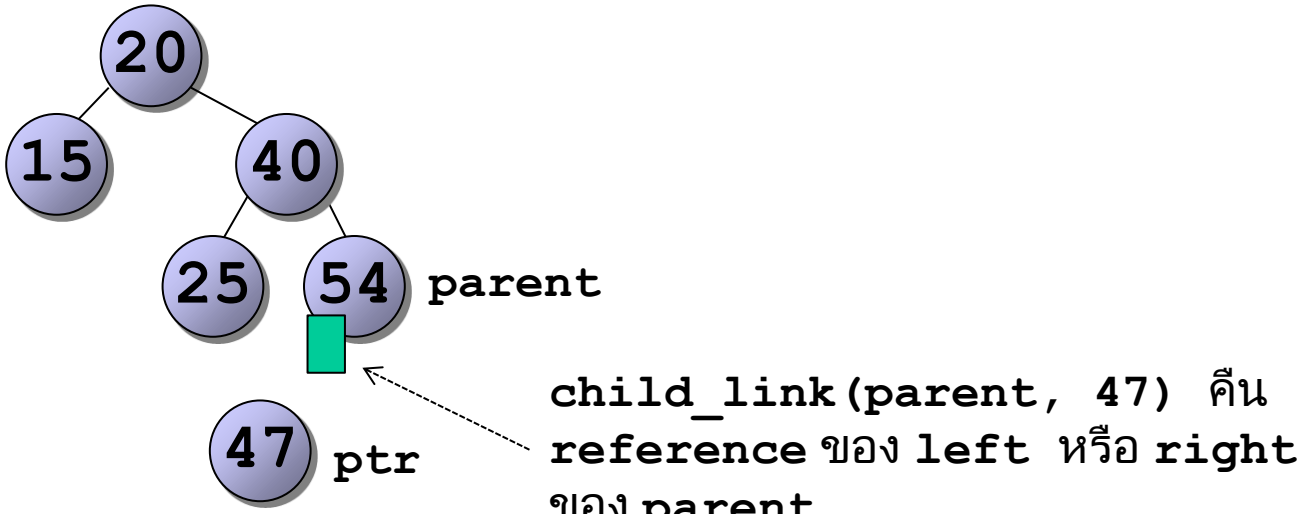
```
pair<iterator,bool> insert(const ValueT& val) {
  node *parent = NULL;
  node *ptr = find_node(val.first,mRoot,parent);
  bool not_found = (ptr==NULL);
  if (not_found) {
    ptr = new node(val,NULL,NULL,parent);
    child_link(parent, val.first) = ptr;
    mSize++;
  }
  return std::make_pair(iterator(ptr), not_found);
}
```



child_link

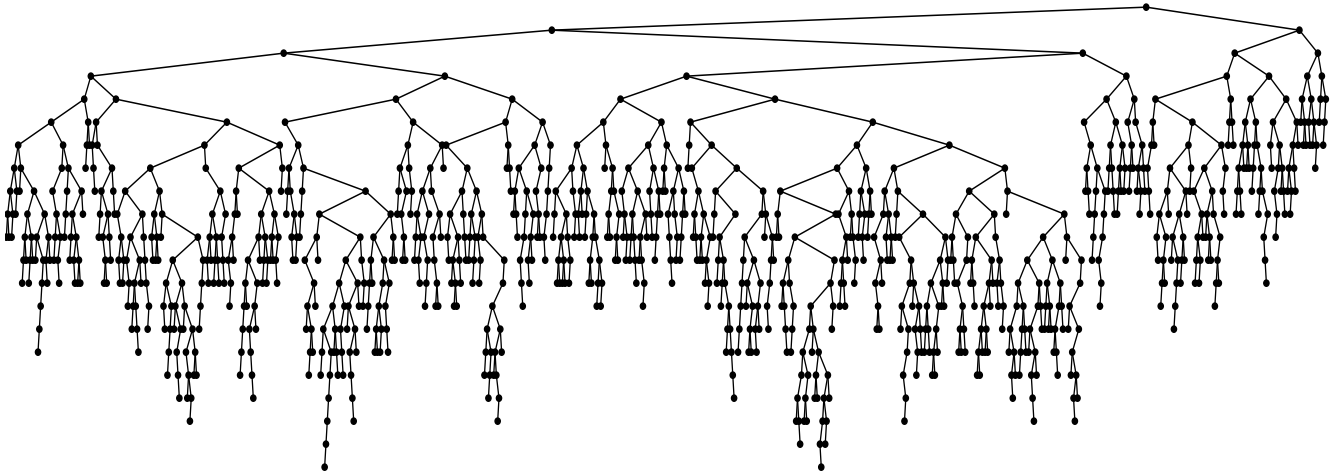
```
node* &child_link(node* parent, const KeyT& k) {
  if (parent == NULL) return mRoot;
  return mLess(k, parent->data.first) ?
         parent->left : parent->right;
}
```

node* & คือ **reference** ของ **pointer** ที่ชี้ไปยัง **node**



ต้นไม้ BSTree ที่สร้างจากข้อมูลสุ่ม

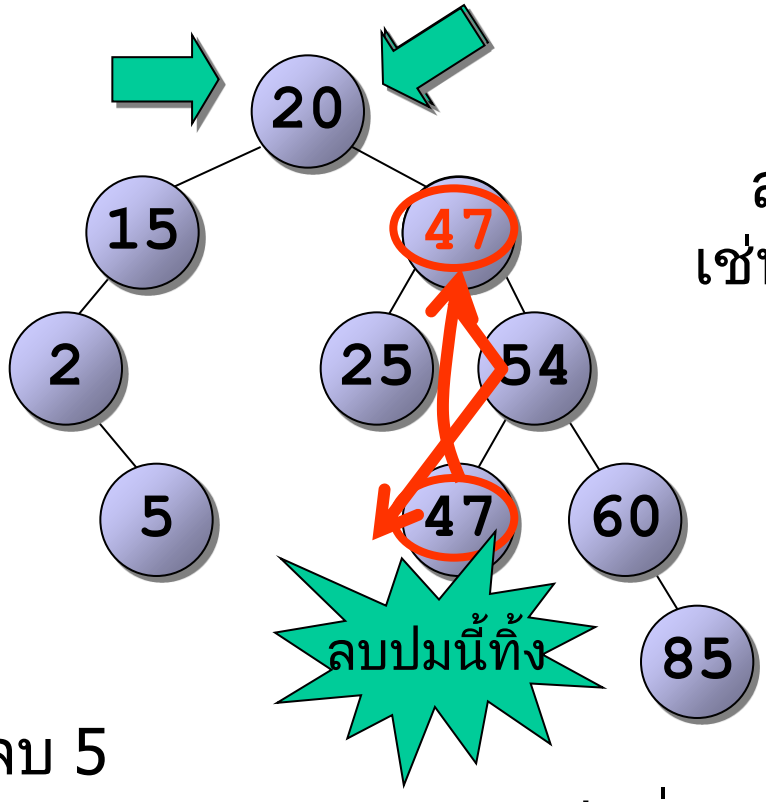
- ต้นไม้ที่เก็บข้อมูล n ตัว
- ช่วงของความสูง : $\lfloor \log_2 n \rfloor \leq h \leq n - 1$
- ถ้าสร้างจากข้อมูลสุ่ม สามารถวิเคราะห์ได้ว่า
  - ความลึกเฉลี่ยของปมภายใน $\approx 1.39 \log_2 n$
  - ความลึกเฉลี่ยของ null $\approx 2 + 1.39 \log_2 n$
  - ความสูง (ความลึกของใบล่างสุด) $\approx 2.99 \log_2 n$



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

การลบข้อมูล

- ค้นหาปมที่เก็บข้อมูลที่ต้องการลบ
- ลบปมที่เก็บข้อมูลนั้น หรือลบข้อมูลในปมนั้น



ลบปมที่มี 2 ลูก
เช่น ต้องการลบ 40

$O(h)$

ลบใบ
เช่น ต้องการลบ 5

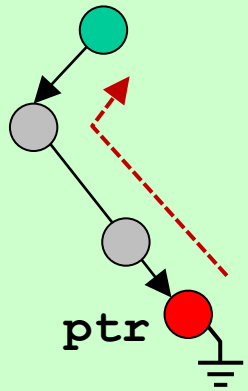
ลบปมที่มีลูกเดียว
เช่น ต้องการลบ 60

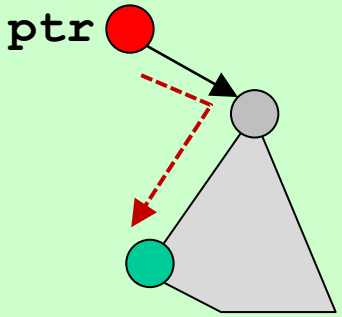
erase

```
size_t erase(const KeyT &key) {
  node *parent = NULL;
  node *ptr = find_node(key,mRoot,parent);
  if (ptr == NULL) return 0;
  if (ptr->left != NULL && ptr->right != NULL) {
    node *min = find_min_node(ptr->right);
    node *&link = child_link(min->parent,min->data.first);
    link = (min->left == NULL) ? min->right : min->left;
    if (link != NULL) link->parent = min->parent;
    swap(ptr->data.first, min->data.first);
    swap(ptr->data.second, min->data.second);
    ptr = min;
  } else {
    node * &link = child_link(ptr->parent, key);
    link = (ptr->left == NULL) ? ptr->right : ptr->left;
    if (link != NULL) link->parent = ptr->parent;
  }
  delete ptr;
  mSize--;
  return 1;
```

iterator (แจงข้อมูลแบบ inorder)

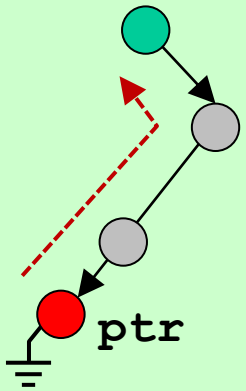
```
class tree_iterator {
protected:
  node* ptr;

public:
  tree_iterator& operator++() {
    if (ptr->right == NULL) {
      node *parent = ptr->parent;
      while (parent != NULL && parent->right == ptr) {
        ptr = parent;
        parent = ptr->parent;
      }
      ptr = parent;
    } else {
      ptr = ptr->right;
      while (ptr->left != NULL) ptr = ptr->left;
    }
    return (*this);
  }
```

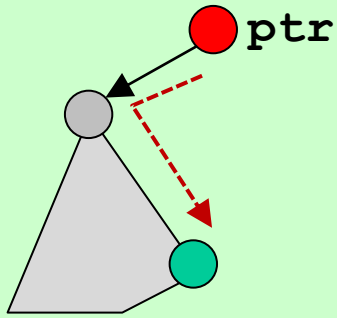




iterator (แจงข้อมูลแบบ inorder)

```
class tree_iterator {
protected:
  node* ptr;

public:
  tree_iterator& operator--() {
    if (ptr->right == NULL) {
      node *parent = ptr->parent;
      while (parent != NULL && parent->left == ptr) {
        ptr = parent;
        parent = ptr->parent;
      }
      ptr = parent;
    } else {
      ptr = ptr->left;
      while (ptr->right != NULL) ptr = ptr->right;
    }
    return (*this);
  }
```



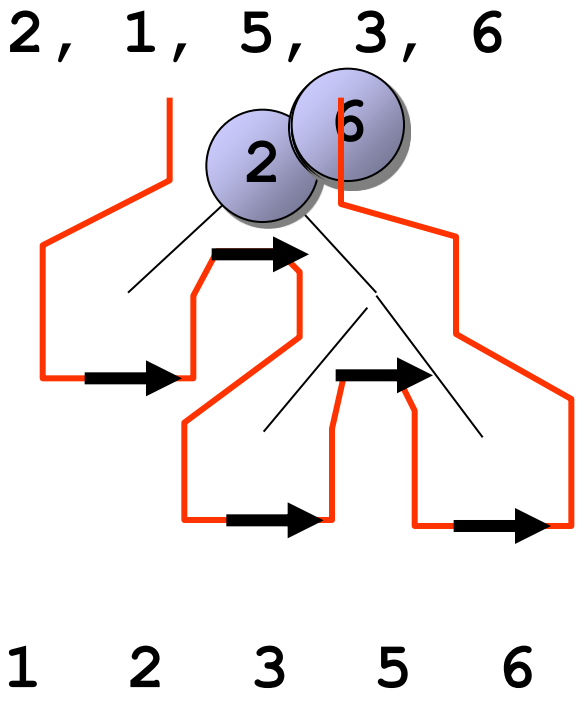


iterator

```
class tree_iterator {
protected:
  node* ptr;
public:
  tree_iterator() : ptr( NULL ) { }
  tree_iterator(node *a) : ptr(a) { }
  tree_iterator operator++(int) {
    tree_iterator tmp(*this); operator++(); return tmp;
  }
  tree_iterator operator--(int) {
    tree_iterator tmp(*this); operator--(); return tmp;
  }
  ValueT& operator*()  { return ptr->data;    }
  ValueT* operator->() { return &(ptr->data); }
  bool    operator==(const tree_iterator& other)
  { return other.ptr == ptr; }
  bool    operator!=(const tree_iterator& other)
  { return other.ptr != ptr; }
};
```

การเรียงลำดับข้อมูลแบบต้นไม้

- นำข้อมูลทั้งหมด มาสร้างต้นไม้ค้นหาแบบทวิภาค
- แวะผ่านต้นไม้นี้แบบตามลำดับ



tree_sort : การเรียงลำดับข้อมูล

```
void tree_sort(float *d, int n) {
  CP::map_bst<float,int> m;
  for (int i=0; i<n; i++) m[d[i]]++;
  int k = 0;
  for (auto& v : m) {
    for (int i=0; i<v.second; i++) {
      d[k++] = v.first;
    }
  }
}
```

$O(n \log n)$

เวลาการทำงานของการเพิ่ม ลบ ค้น

- find, find_min, find_max, insert, erase : 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)

สรุป

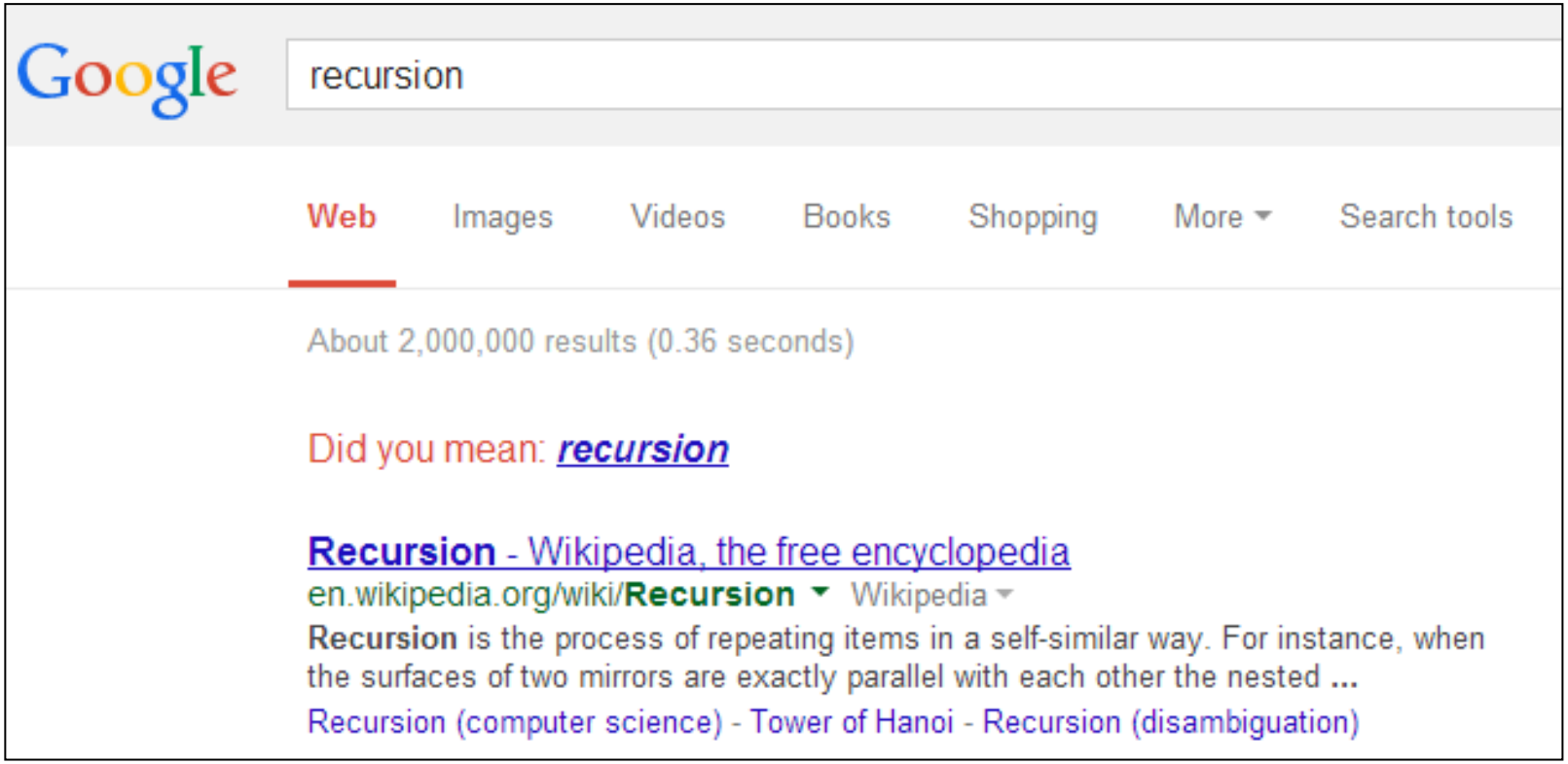
- ต้นไม้ค้นหาแบบทวิภาคมีจัดเก็บข้อมูลโดยอาศัยการเปรียบเทียบความมากกว่าน้อยกว่าของข้อมูล
- สามารถลดปริมาณข้อมูลที่ต้องพิจารณาได้ทีละมาก ๆ ระหว่างการเพิ่ม ลบ และค้นหา
- เวลาการทำงานขึ้นกับลักษณะของต้นไม้
- โชคดีทำงานเร็ว O(log n), โชคร้ายทำงานช้า O(n)
- เป็นโครงสร้างพื้นฐานของโครงสร้างข้อมูลอื่น ๆ ที่ซับซ้อนและมีประสิทธิภาพกว่า

Thinking Recursively

Recursion, *see Recursion*.[2]

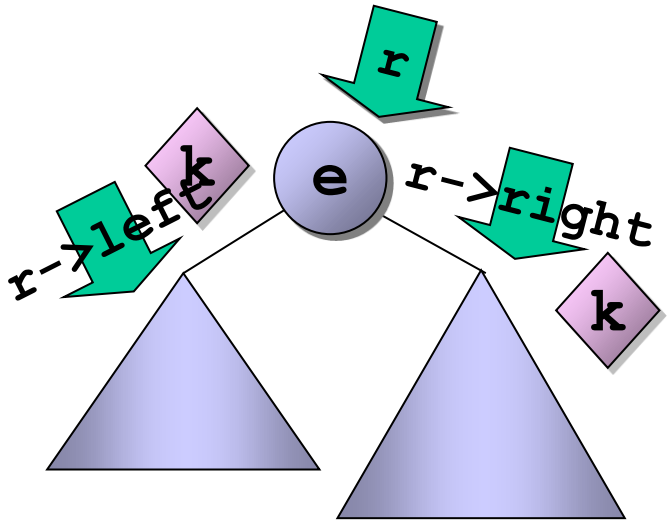
"To understand recursion,

you must understand recursion."[2]

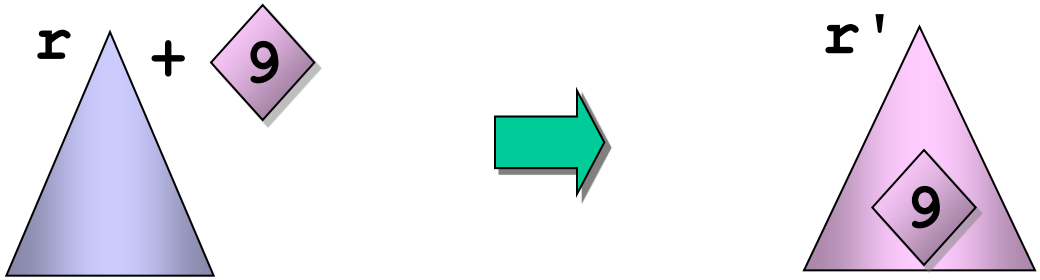


การค้นหาข้อมูลแบบเวียนเกิด

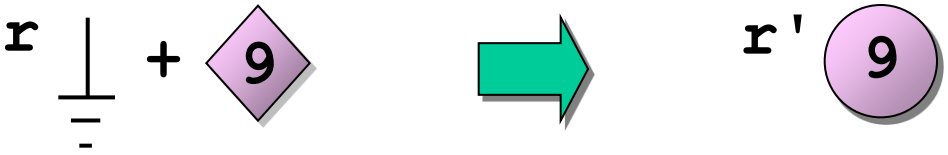
```
node* find_node(const KeyT& k, node* r, node* &parent){
  if (r == NULL) return NULL;
  int cmp = compare(k, r->data.first);
  if (cmp == 0) return r;
  parent = r;
  return find_node(k,
                   cmp < 0 ? r->left : r->right,
                   parent);
}
```



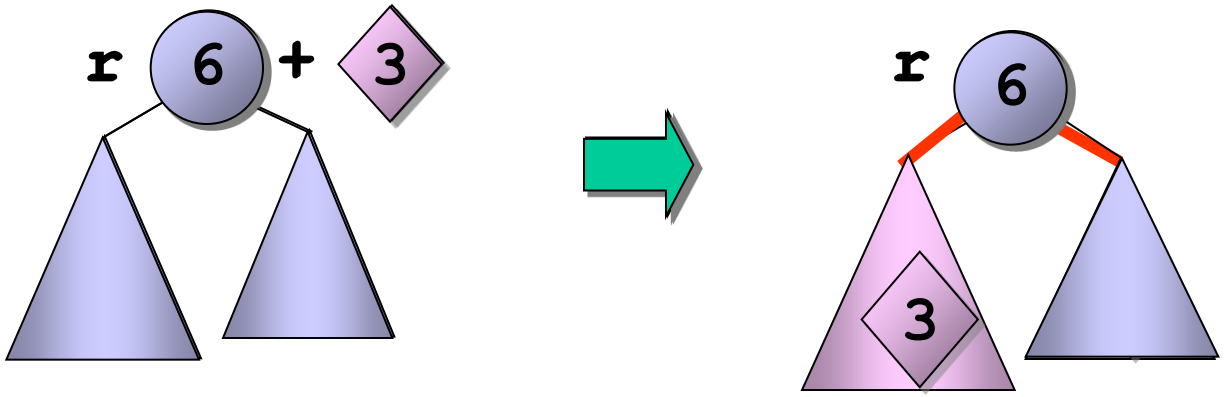
การเพิ่มข้อมูลแบบเวียนเกิด




```
if (r == NULL) return new node(x, NULL, NULL);
```




```
if (cmp(k, key(r))<0) r->left = insert(r->left, x);
if (cmp(k, key(r))>0) r->right = insert(r->right, x);
```



insert : แบบเวียนเกิด

```
node* insert(const ValueT& val, node *r, node * &ptr) {
  if (r == NULL) {
    mSize++;
    ptr = r = new node(val,NULL,NULL,NULL);
  } else {
    int cmp = compare(val.first, r->data.first);
    if (cmp == 0) ptr = r;
    else if (cmp <  0) {
      r->left = insert(val, r->left, ptr);
      if (r->left != NULL) r->left->parent = r;
    } else {
      r->right = insert(val, r->right, ptr);
      if (r->right != NULL) r->right->parent = r;
    }
  }
  return r;
}
```

```
class node {
  ...
  void set_left(node *n) {
    this->left = n;
    if (n != NULL) this->left->parent = this;
  }
```

insert : แบบเวียนเกิด

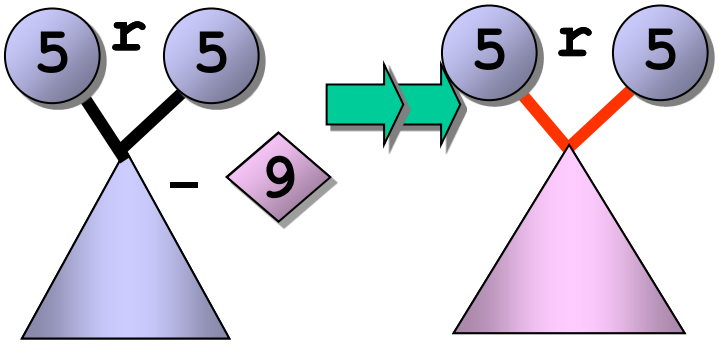
```
node* insert(const ValueT& val, node *r, node * &ptr) {
  if (r == NULL) {
    mSize++;
    ptr = r = new node(val,NULL,NULL,NULL);
  } else {
    int cmp = compare(val.first, r->data.first);
    if (cmp == 0) ptr = r;
    else if (cmp <  0) {
      r->set_left( insert(val, r->left, ptr) );

    } else {
      r->set_right( insert(val, r->right, ptr) );

    }
  }
  return r;
}
```

```
class node {
  ...
  void set_left(node *n) {
    this->left = n;
    if (n != NULL) this->left->parent = this;
  }
```

insert : แบบเวียนเกิด

```
node* insert(const ValueT& val, node *r, node * &ptr) {
  if (r == NULL) {
    mSize++;
    ptr = r = new node(val,NULL,NULL,NULL);
  } else {
    int cmp = compare(val.first, r->data.first);
    if (cmp == 0) ptr = r;
    else if (cmp <  0)
      r->set_left( insert(val, r->left, ptr) );
    else
      r->set_right( insert(val, r->right, ptr) );
  }
  return r;
}
pair<iterator,bool> insert(const ValueT& val) {
  node *ptr = NULL;
  size_t s = mSize;
  mRoot = insert(val, mRoot, ptr);
  mRoot->parent = NULL;
  return std::make_pair(iterator(ptr),(mSize > s));
}
```

remove : แบบเวียนเกิด



erase : แบบเวียนเกิด

```
node *erase(const KeyT &key, node *r) {
  if (r == NULL) return NULL;
  int cmp = compare(key, r->data.first);
  if (cmp < 0) {
    r->set_left( erase(key, r->left) );
  } else if (cmp > 0) {
    r->set_right( erase(key, r->right) );
  } else {
    if (r->left == NULL || r->right == NULL) {
      ...
    } else {
      ...
    }
  }
  return r;
}
size_t erase(const KeyT &key) {
  size_t s = mSize;
  mRoot = erase(key, mRoot);
  return s == mSize ? 0 : 1;
}
```

```
node *erase(const KeyT &key, node *r) {
  if (r == NULL) 
  int cmp = compa
  if (cmp < 0) {
    r->set_left(e
  } else if (cmp 
    r->set_right(
  } else {
    if (r->left == NULL || r->right == NULL) {
      node *n = r;
      r = (r->left == NULL ? r->right : r->left);
      delete n;
      mSize--;
    } else {
      node * m = r->right;
      while (m->left != NULL) m = m->left;
      swap(r->data.first, m->data.first);
      swap(r->data.second, m->data.second);
      r->set_right(erase(m->data.first, r->right));
    }
  }
  return r;
}
```