ต้นไม้ค้นหาแบบทวิภาค

(Binary Search Tree)

สมชาย ประสิทธิ์จูตระกูล

Translated to English by

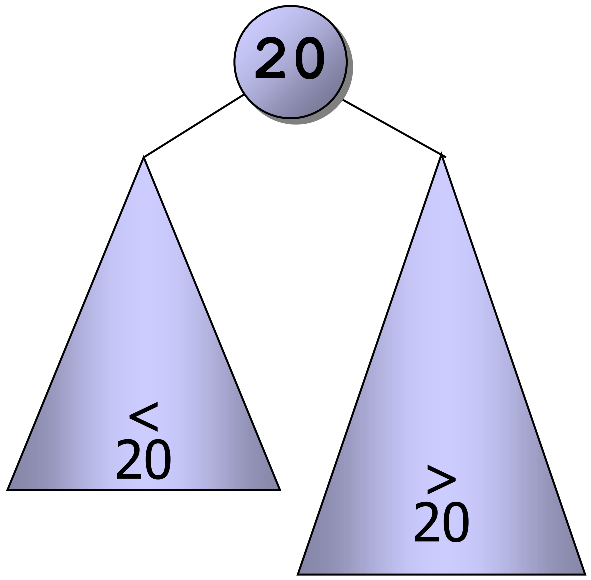
Nuttapong Chentanez

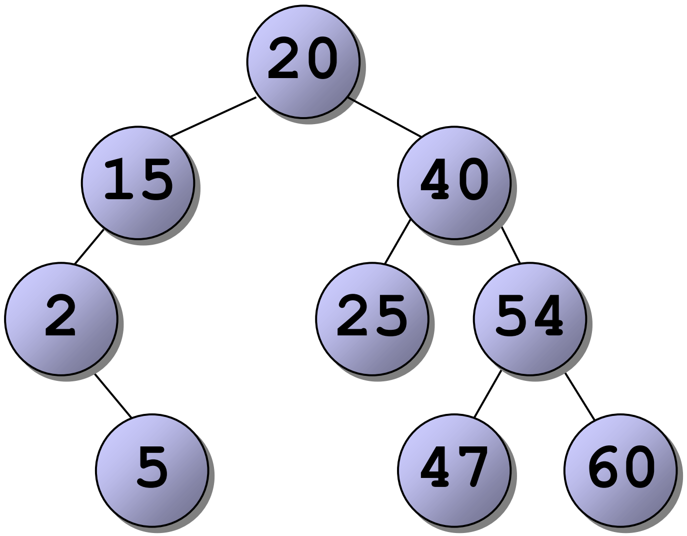
Topic

- Definition of binary search tree
- Structure of binary search tree
- Operations
  - Search data, find minimum, maximum
  - Insert data
  - Delete data
  - Sort data with binary tree

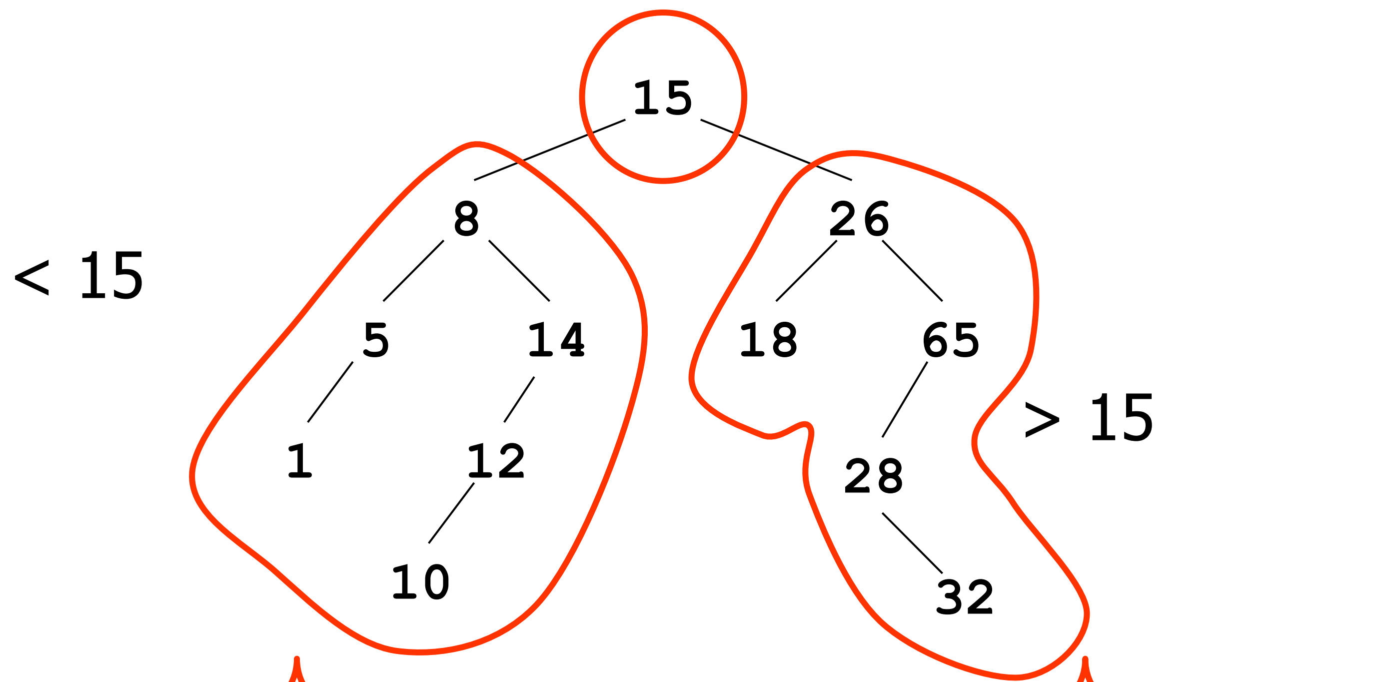
Binary Search Tree

- It's a binary tree
- Data are stored at nodes
- Data at the left child is smaller than data at parent
- Data at the right child is larger than data at parent
- Every subtree is binary search tree





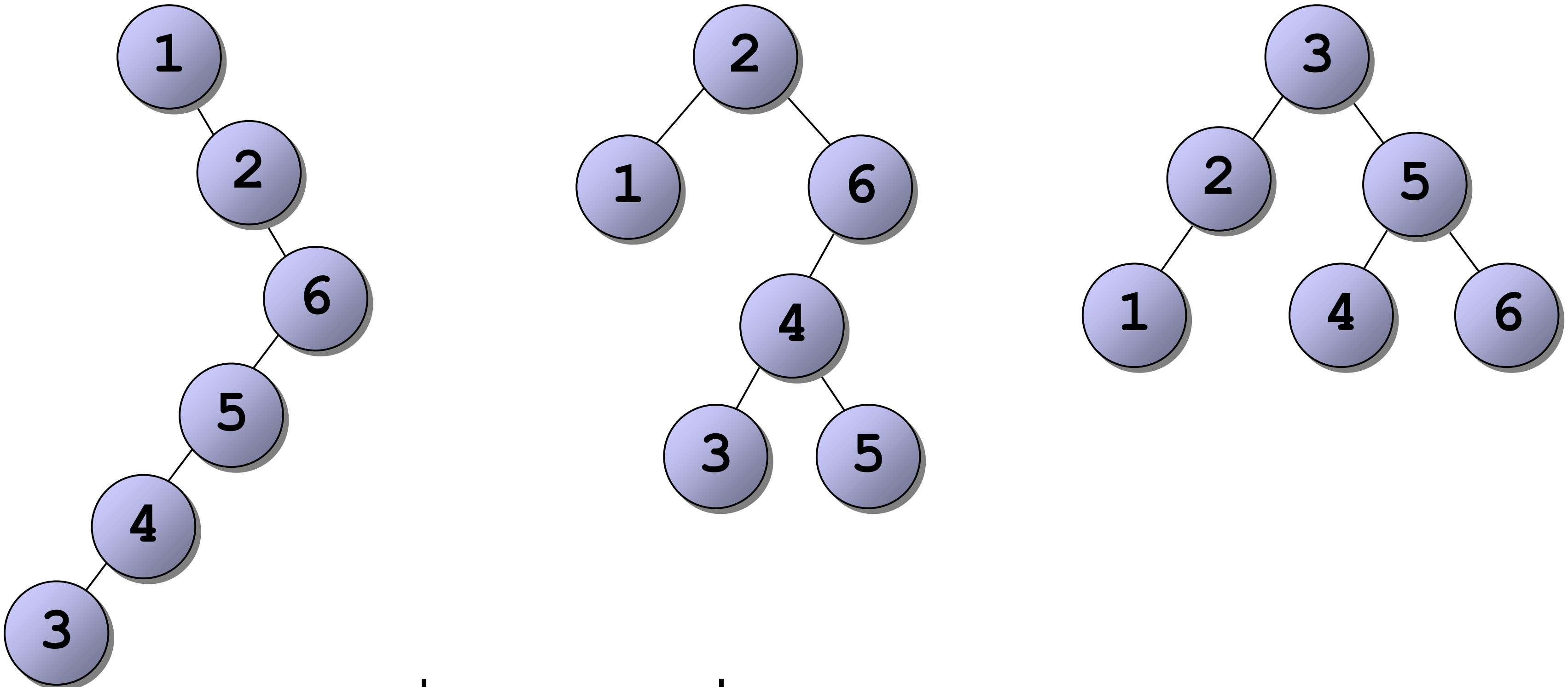
In-order Traversal



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

Smallest to Greatest

Many ways to store the same set of data



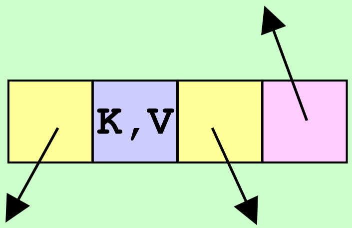
$$\lfloor \log_2 n \rfloor \leq h \leq n-1$$

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) { }
};
```

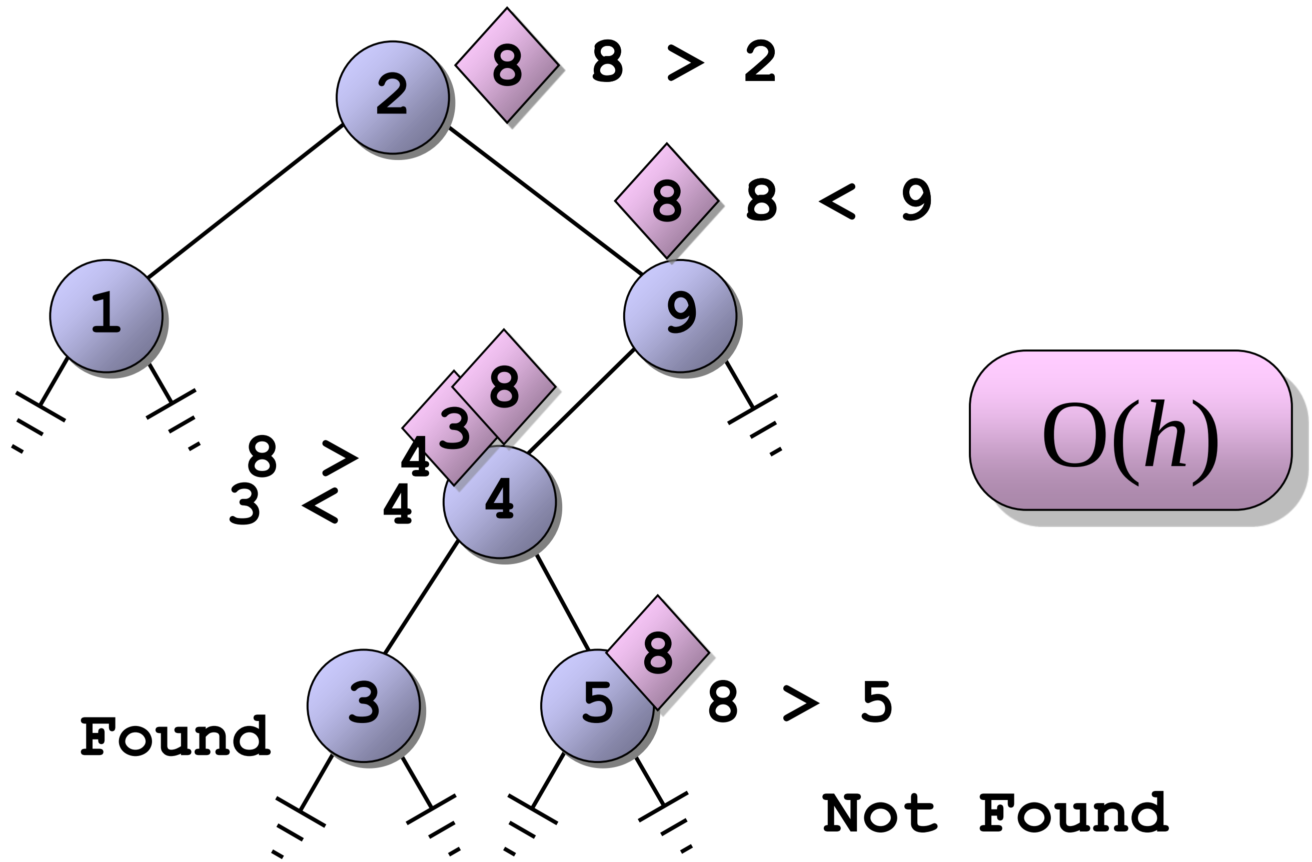


map_bst

```
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) { ... }
```

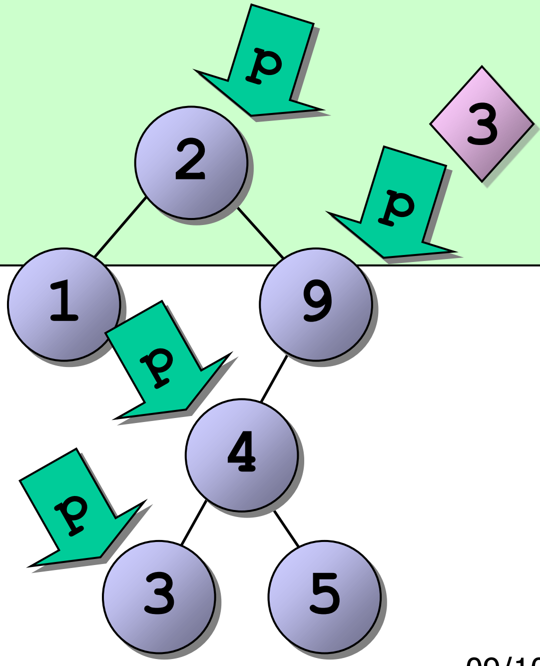
Search data

- Visit nodes and compare
- Utilize ordering rule to improve search



Finding data

```
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;
}
```



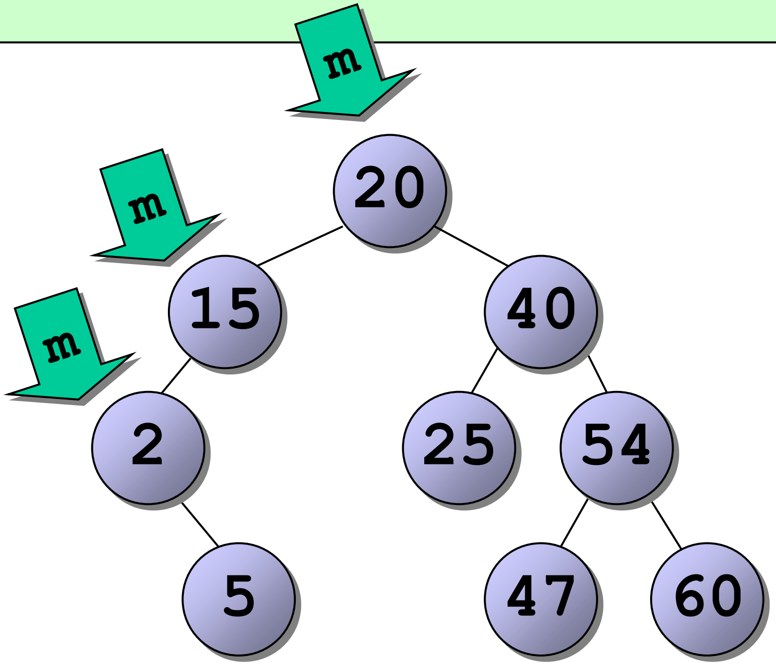
find_node

```
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 : Find minimum

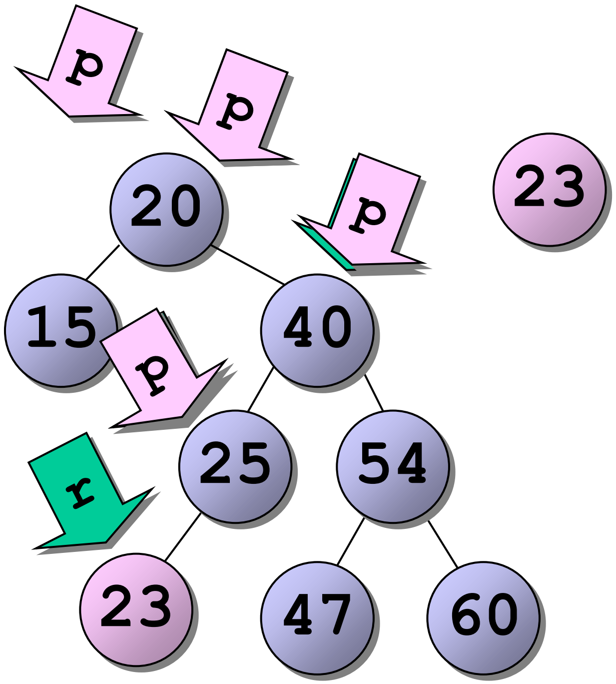
```
node* find_min_node(node* r) {
  node *min = r;
  while (min->left != NULL) {
    min = min->left;
  }
  return min;
}
```



find_max_node : Find maximum

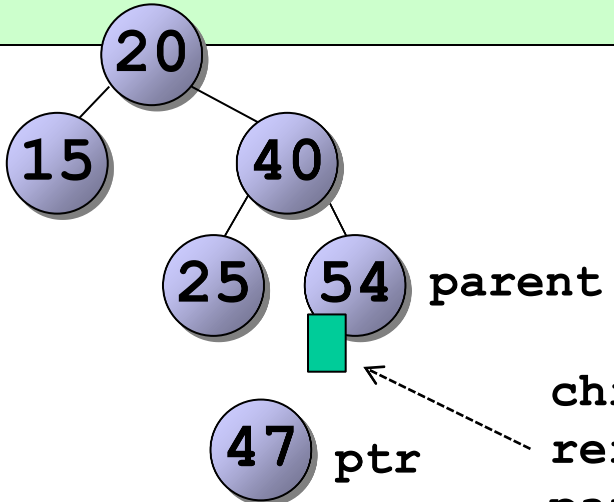
```
node* find_max_node(node* r) {
  node *max = r;
  while (max->right != NULL) {
    max = max->right;
  }
  return max;
}
```

insert: Insert data



insert: Insert data

```
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);
}
```

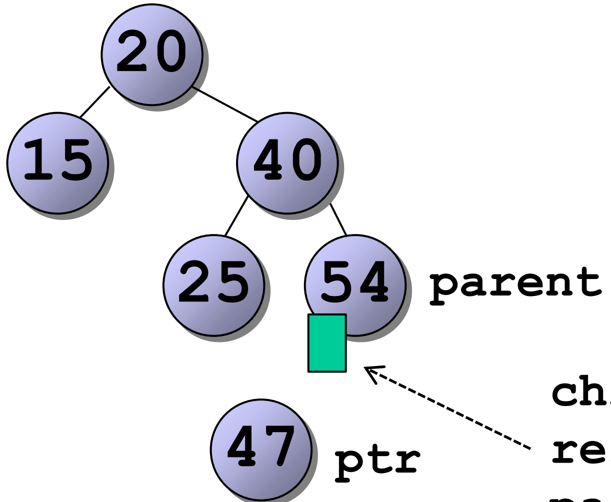


child_link(parent, 47) return **reference** of **left** or **right** of **parent**

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;
}
```

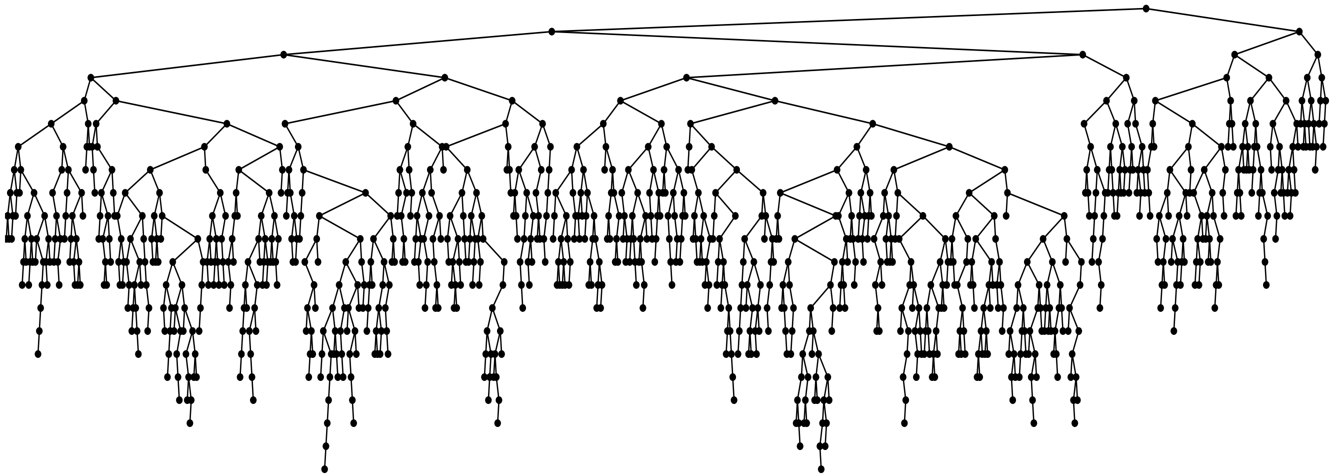
node* & is **reference** of **pointer** to a **node**



child_link(parent, 47) return **reference** of **left** or **right** of **parent**

BST constructed from random data

- Tree that stores n data points
- Has height : $\lfloor \log_2 n \rfloor \leq h \leq n-1$
- When constructed from random data, w
  - Average depth of internal node $\approx 1.39 \log_2 n$
  - Average depth of null $\approx 2 + 1.39 \log_2 n$
  - Height (Depth of deepest tree) $\approx 2.99 \log_2 n$

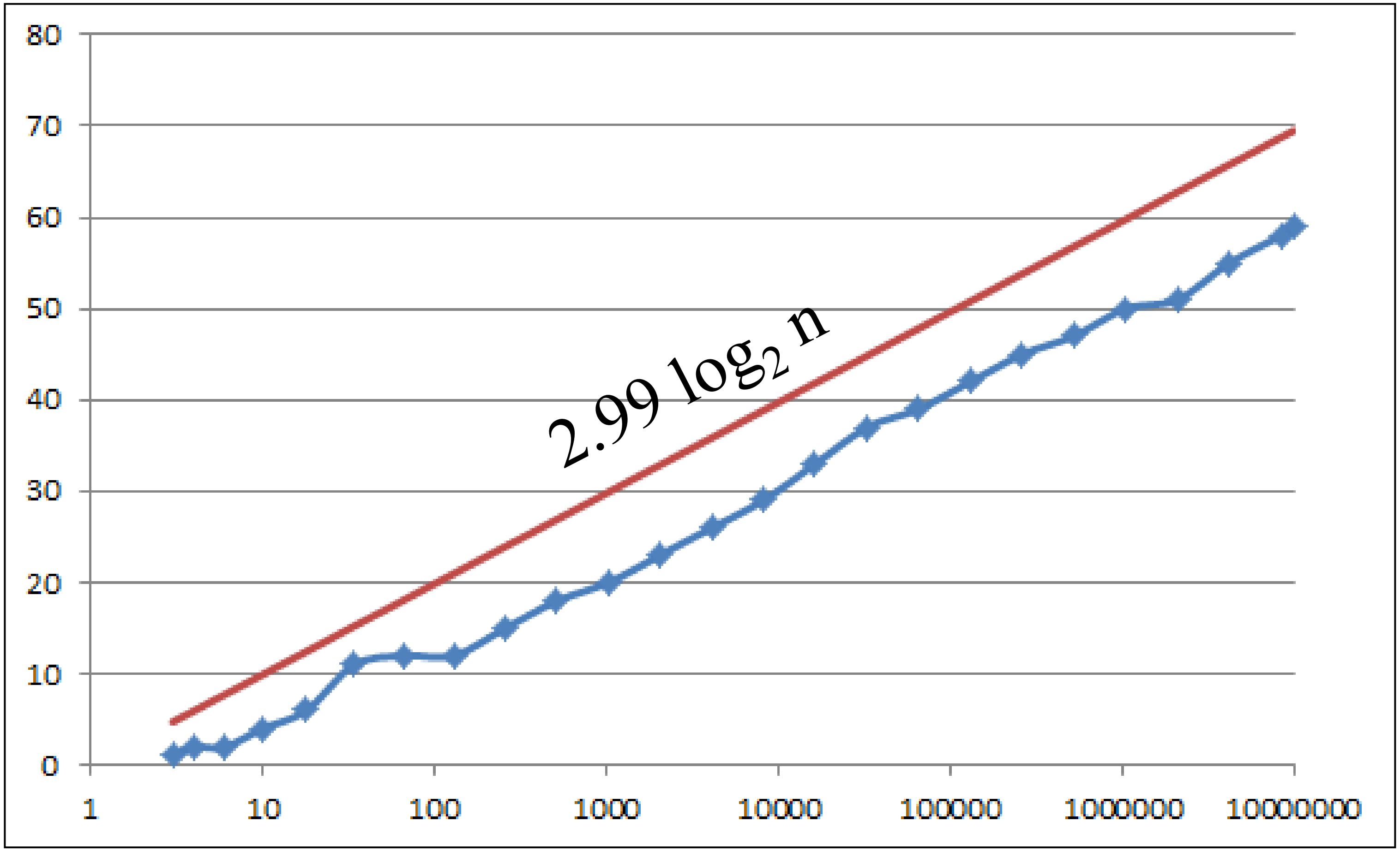


Devroye, L. 1986. A note on the height of binary search trees. *J. ACM 33*, 489–498.

Height measurement

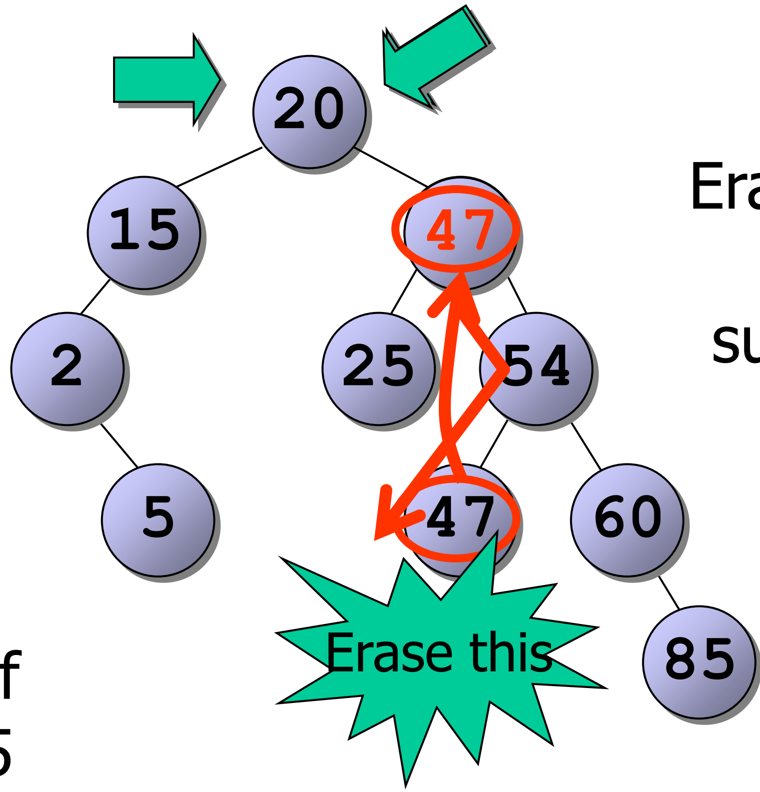
```
CP::map_bst<int,int> m;
int n = 10000000;
int *d = new int[n];
for (int i=0; i<n; i++) d[i] = i;
for (int i=0; i<n; i++) {
  int j = rand()*rand()%n;
  int t = d[i]; d[i] = d[j]; d[j] = t;
}

for (int i=0; i<n; i++) {
  std::cout << d[i] << ",";
  m[d[i]] = 1;
  if (i % 100000 == 0) {
    cout << m.size() << "\t" << m.height() << endl;
  }
}
cout <<  m.size() << "\t" << m.height() << endl;
```

Experimental Data



Erase data

- Search node that store data to be erased
- Erase node or erase data in that node



Erase node with 2 children such as node 40

O(h)

Erase leaf such as 5

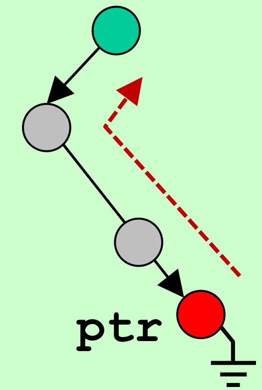
Erase node with 1 child such as 60

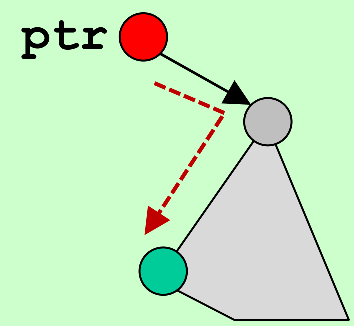
erase

```
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 (enumerate with inorder traversal)

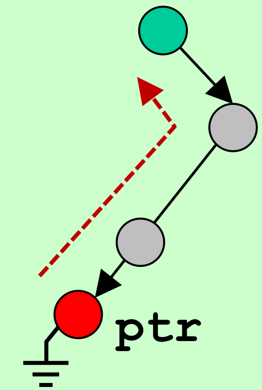
```
class tree_iterator {
protected:
  node* ptr;

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);
  }
```

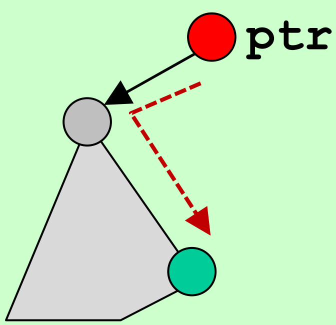




iterator (enumerate with inorder traversal)

```
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);
  }
```



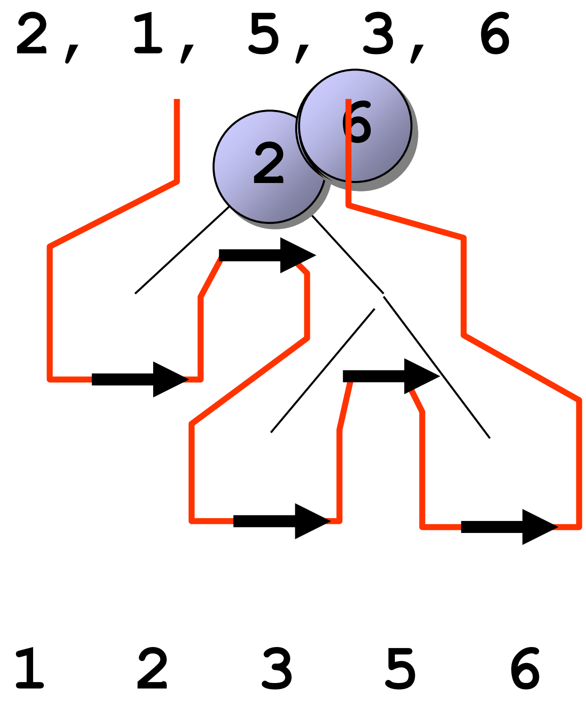


iterator

```
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; }
};
```

Sorting data with binary search tree

- Insert all data into binary search tree
- Visit tree in an in-order fashion



tree_sort : sorting data

```
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;
    }
  }
}
```

$O(n^2)$

```
void tree_sort(float *d, int n) {
  shuffle(d, d+n, default_random_engine(123));
  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;
    }
  }
}
```

$O(n \log n)$

Running time of insert, erase, search

- find, find_min, find_max, insert, erase : O(h)
- Tree has height $\lfloor \log_2 n \rfloor \leq h \leq n - 1$
- Best case (shortest tree) : O(log n)
- Worst case (tallest tree) : O(n)
- Average case (when build tree from random data) : O(log n)

Summary

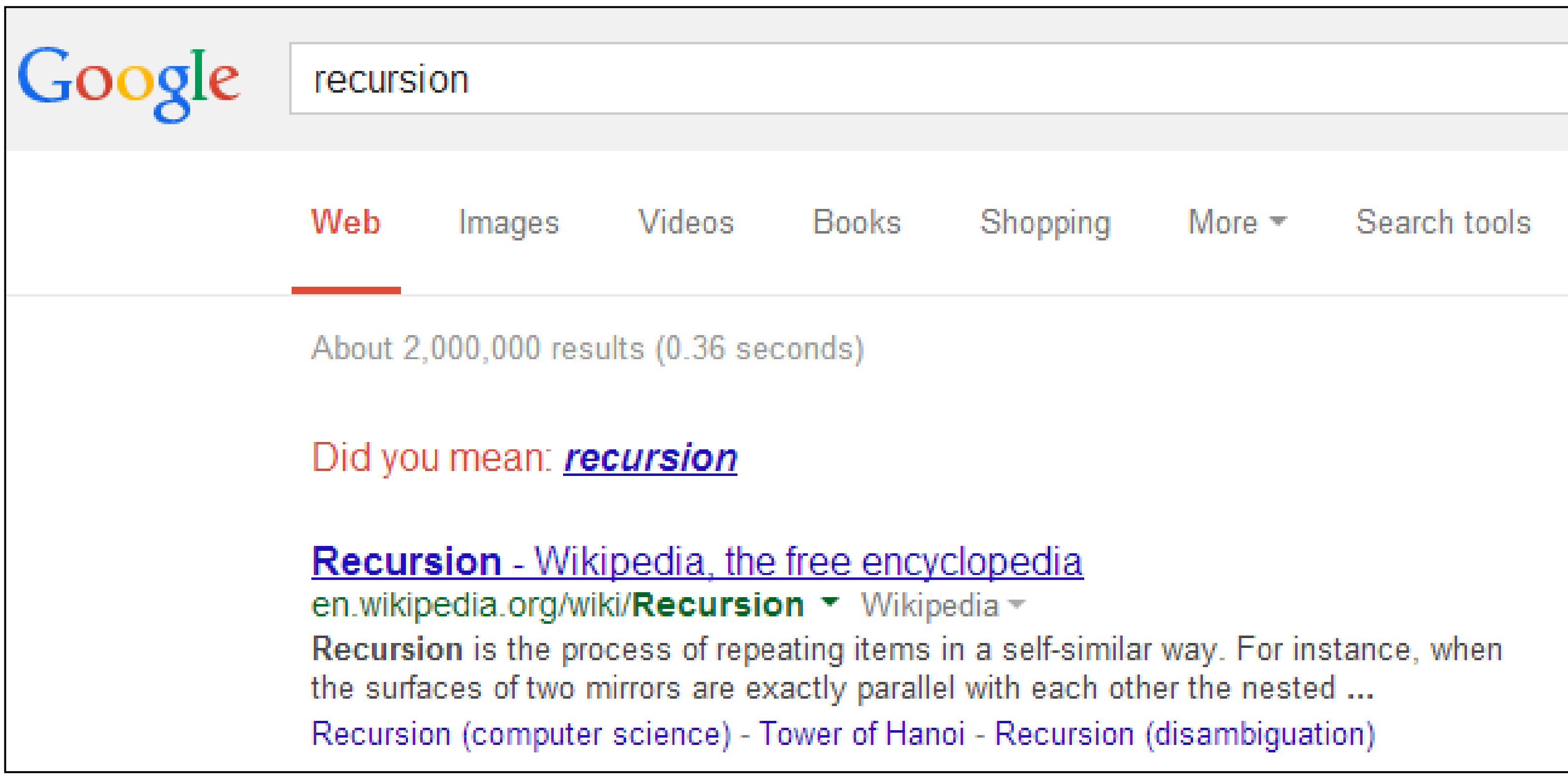
- Binary search tree store data based on comparison result
- Can reduce data needed consideration during insert, erase and search
- Running time depend on the shape of the tree
- Best case O(log n), worst case O(n)
- Is a basic of more complicated structure with better performance

Thinking Recursively

Recursion, *see Recursion*.[2]

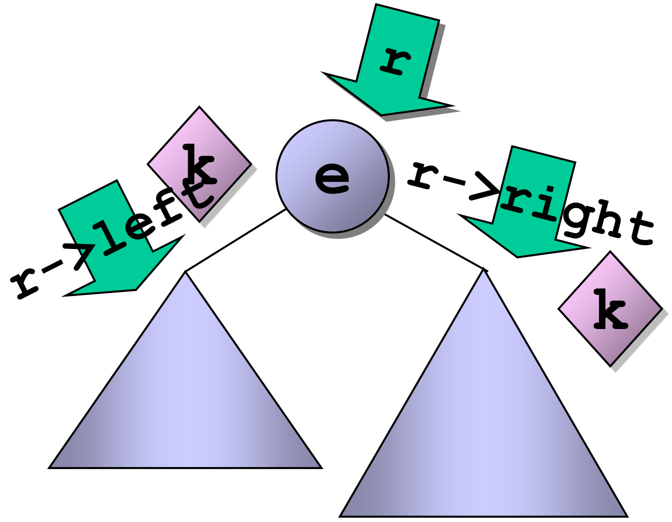
"To understand recursion,

you must understand recursion."[2]

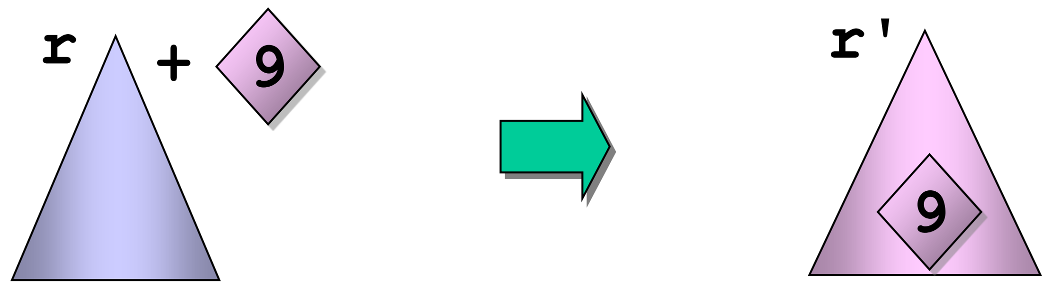


Recursive search

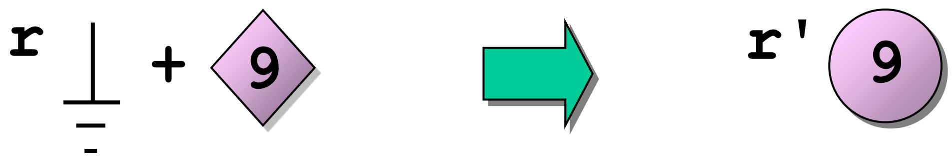
```
node* find_node(const KeyT& k, node* r, node* &parent){
  if (r == NULL) return NULL;
  int cmp = compare(k, r->data.first);
  if (cmp == 0) return r;
  parent = r;
  return find_node(k,
                   cmp < 0 ? r->left : r->right,
                   parent);
}
```



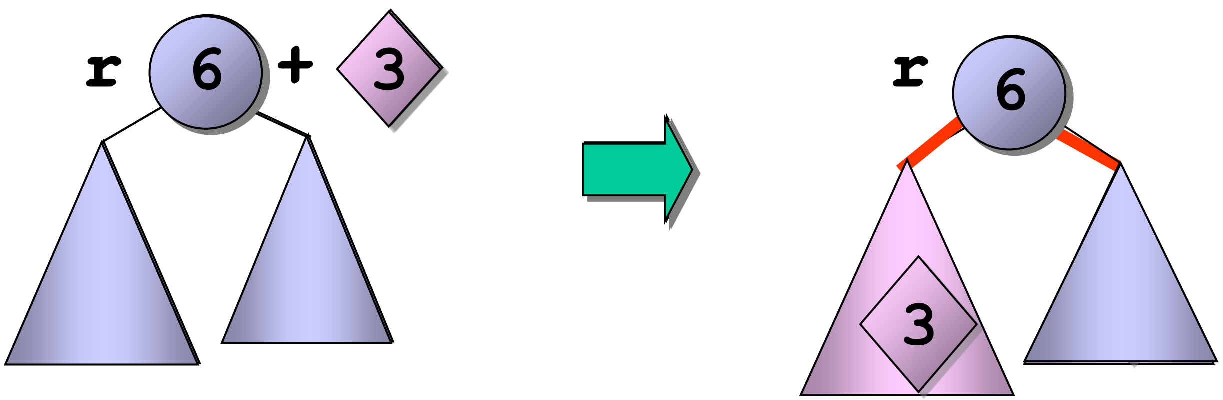
Recursive insert




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




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



insert : recursive

```
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 : recursive

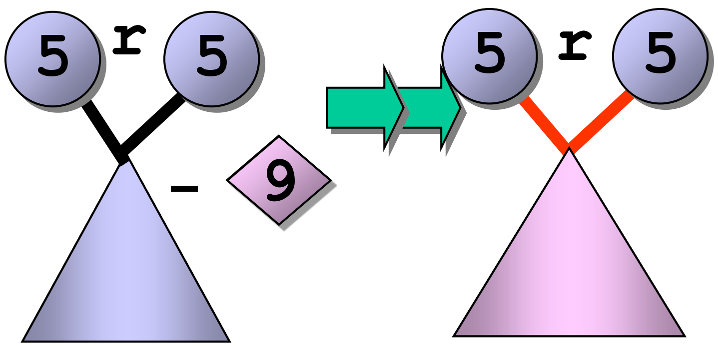
```
node* insert(const ValueT& val, node *r, node * &ptr) {
  if (r == NULL) {
    mSize++;
    ptr = r = new node(val,NULL,NULL,NULL);
  } else {
    int cmp = compare(val.first, r->data.first);
    if (cmp == 0) ptr = r;
    else if (cmp <  0) {
      r->set_left( insert(val, r->left, ptr) );

    } 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 : recursive

```
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 : recursive



erase : recursive

```
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) r
  int cmp = compa
  if (cmp < 0) {
    r->set_left(e
  } else if (cmp >
    r->set_right(e
  } 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;
}
```