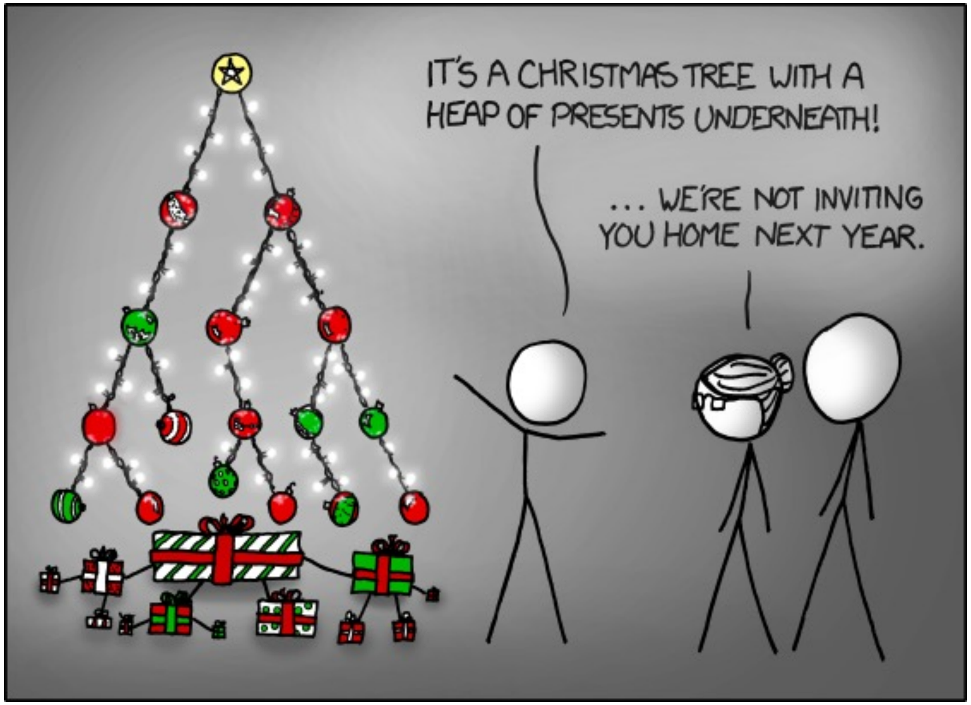
VE280 Programming and Elementary Data Structures

Paul Weng
UM-SJTU Joint Institute

Binary Search Tree and Set



Learning Objectives

- Understand what is a binary search tree
- Learn how to implement a binary search tree
- Learn how to implement a set using a binary search tree

Outline

- Binary Search Tree

- Implementing a set using a binary search tree

Binary Tree

- A binary tree
  - is empty, or
  - contains a root node, a left child, and a right child; both children are binary trees.

- Depth of a tree = height - 1

Implementation of Binary Tree

```
struct Node{
    int val;
    Node *left;
    Node *right;
}
class BinaryTree{
    Node *root;
public:
    ...
}
```

Pre-order Tree Traversal

```
void BinaryTree::_preorder(Node *n){
   if (!n){
      // perform some operations on root
      _preorder(root->left);
      _preorder(root->right);
   }
}
void BinaryTree::preorder(){
   _preorder(root);
}
```

Pre-order Tree Traversal

Assuming that the operation on the root node is printing its value, what is the output of pre-order tree traversal on this tree?

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

- **A.** 1234567.
- **B.** 1245367.
- **C.** 4251637.
- **D.** 4526731.

In-order Tree Traversal

```
void BinaryTree::_inorder(Node *n){
    if (!n){
        _inorder(root->left);
        // perform some operations on root
        _inorder(root->right);
    }
}
void BinaryTree::inorder(){
    _inorder(root);
}
```

In-order Tree Traversal

Assuming that the operation on the root node is printing its value, what is the output of in-order tree traversal on this tree?

- **A.** 1234567.
- **B.** 1245367.
- **C.** 4251637.
- **D.** 4526731.

```
      1
     / \
    2   3
   / \ / \
  4  5 6  7
```

Post-order Tree Traversal

```
void BinaryTree::_postorder(Node *n){
    if (!n){
        _postorder(root->left);
        _postorder(root->right);
        // perform some operations on root
    }
}
void BinaryTree::postorder(){
    _postorder(root);
}
```

Post-order Tree Traversal

Assuming that the operation on the root node is printing its value, what is the output of post-order tree traversal on this tree?

```
      1
     / \
    2   3
   / \ / \
  4  5 6  7
```

- **A.** 1234567.
- **B.** 1245367.
- **C.** 4251637.
- **D.** 4526731.

Binary Search Tree

- A binary search tree:
  - is a search tree,
  - if not empty, the key contained in its root node is larger than those contained in its left child and smaller than those contained in its right child,
  - Both left and right children are binary search trees.

- A binary search tree is balanced if the difference of the depths of its left and right subtrees is bounded by one.

Implementation of Binary Search Tree

```
struct Node{
    int key;
    Node *left;
    Node *right;
}
class BinarySearchTree{
    Node *root;
public:
    ...
    void add(int k);
    bool contains(int k);
    void delete(int k);
}
```

Implementation of `add`

```
Node *BinarySearchTree::_add(Node *n, int k);
   if (!n){
      Node *nn = new Node(k, NULL, NULL);
      return nn;
   }
   if (n->key > k){
      n->left = _add(n->left, k);
      return n;
   }
   if (n->key < k){
      n->right = _add(n->right, k);
      return n;
   }
}
void BinarySearchTree::add(int k);
   root = _add(root, k);
}
```

Implementation of `contains`

```
Node *BinarySearchTree::_contains(Node *n, int k);
   if (!n){
      return False;
   }
   if (n->key > k)
      return _contains(n->left, k);
   if (n->key < k)
      return _contains(n->right, k);
   if (n->key == k)
      return True;
}
bool BinarySearchTree::contains(int k);
   return _contains(root, k);
}
```

Implementation of `delete`

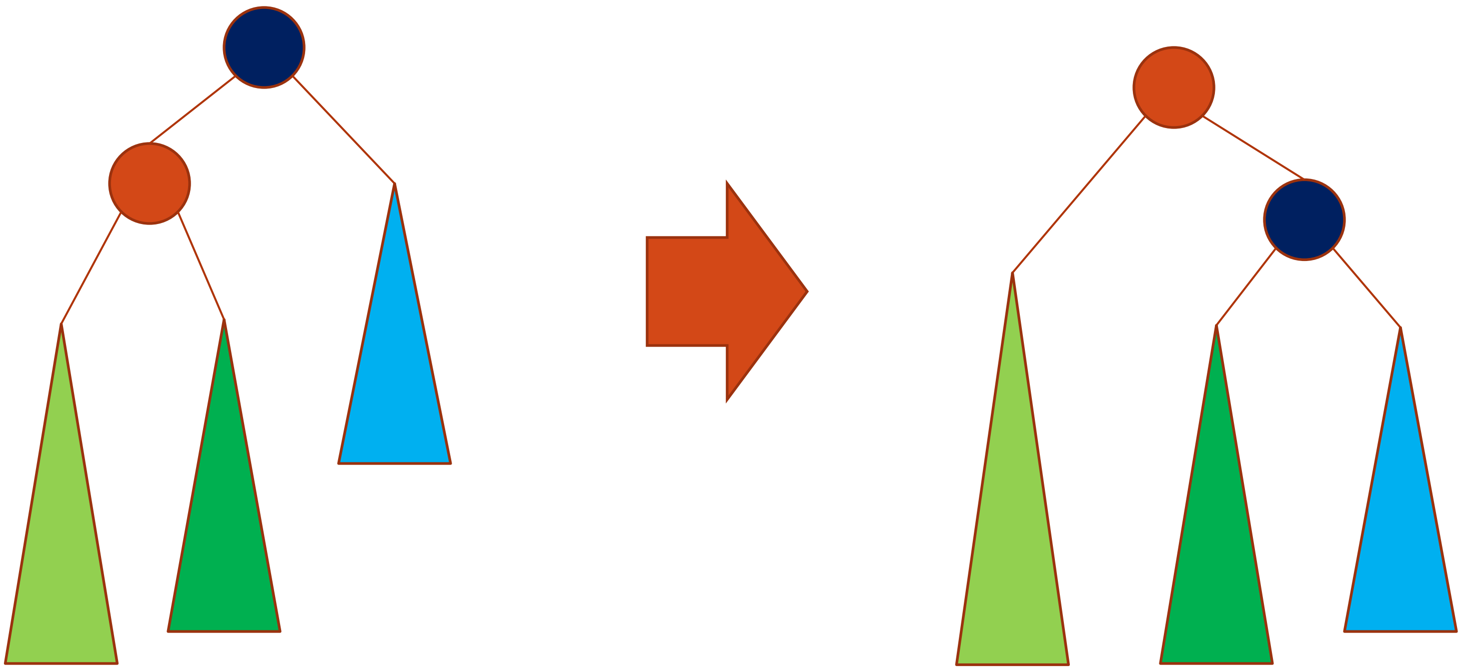
```
Node *BinarySearchTree::_delete(Node *n,
int k){
    if (!n) return NULL;
    if (n->key > k){
        n->left = _delete(n->left, k);
        return n;
    }
    if (n->key < k){
        n->right = _delete(n->right, k);
        return n;
    }
    if (n->key == k){
        if (!n->left && !n->right)
            return NULL;
        if (!n->left){
            Node *right = n->right;
            delete n;
            return right;
        }
        if (!n->right){
            Node *left = n-> left;
            delete n; return left;
        }
        n->key = minKey(n->right)
        n->right = _delete(n->right,
                          n->key);
    }
}
Node *BinarySearchTree::delete(int k){
    if (!root) return;
    root = _delete(root, k);
}
Node *BinarySearchTree::minKey(Node
*n){
    if (!n) throw exception;
    if (!n->left) return n->key;
    return minKey(n->left);
}
```

Complexity of `add`, `contains`, `delete`

- The number of nodes visited is in O(depth of tree)
- If tree is balanced, depth is in O(log(n)) where n=# values
- Therefore, for balanced binary search tree, the computational complexity is in O(log(n))

Balance a binary search tree

- Possibly needed after a call of `add` or `delete`
- Example:



Implementation of Set

```
class Set{
    BinarySearchTree bst;
public:
    ...
    void add(int k);
    bool contains(int k);
    void delete(int k);
}
```