CSCI 262
Data Structures

25 – Binary Search Trees

Review: Binary Trees

A binary tree is defined recursively:

```
+ = or

A binary tree is (empty)
a root node with a left child and a right child, each of which is a binary tree.
```

Review: Binary Tree Implementation

Just follow the recursive definition to get a simple implementation:

```
template <class T>
class binary_tree_node {
    public:
        T data;
        binary_tree_node<T>* left;
        binary_tree_node<T>* right;
    }
```

Binary Search Trees

Data structure for holding comparable elements
- Efficient searching, insertion, deletion
- Underlying structure for sets, maps

Items in left subtree < item at root;
root item < items in right subtree;
subtrees are BSTs.

In Order Traversal

Visit left subtree,
visit root,
visit right subtree.

```
template <class T>
void print_inorder(binary_tree_node<T>* root) {
    if (root != NULL) {
        print_inorder(root->left);
        cout << root->data << " ";
        print_inorder(root->right);
    }
}
```

Example:

search (root, "cherry")

```
template <class T>
binary_tree_node<T>* search(binary_tree_node<T>* root, T val) {
    if (root == NULL) return NULL;
    if (val == root->data) return root;
    if (val < root->data) return search(root->left, val);
    return search(root->right, val);
}
```

apple banana cherry guava lemon orange peach pear quince
Inserting

Where do we insert an item into the tree?

How would you add "fig" to this tree?

Answer: put it where you expect to find it!

Inserting
template <class T>
void insert(binary_tree_node<T>*& root, T val) {
    if (root == NULL) root = new binary_tree_node<T>(val);
    else if (val < root->info) insert(root->left, val);
    else if (val > root->info) insert(root->right, val);
}

Example:
insert(root, "fig")

Removing
template <class T>
void remove(binary_tree_node<T>* root, T val) {
    // this is trickier!
    ...
}

Removing

3 Cases when node is in tree:
1. No children
2. One child
3. Two children

Removing Case 1: No Child
1. Find the item
2. Detach and delete

Example: remove(root, "lemon")

Removing Case 2: One Child
1. Find the item
2. Link child to parent
3. Delete

Example: remove(root, "quince")
Removing Case 3: Two Children

1. Find the item
2. Swap with rightmost item in left subtree (why?)
3. Remove rightmost node in left subtree (Case 1 or 2)

Example: remove(root, "guava")

Removing: Code

```cpp
template <class T>
void remove(binary_tree_node<T>*& root, T val) {
    if (root == NULL) return NULL;
    if (val < root->data) remove(root->left, val);
    else if (val > root->data) remove(root->right, val);
    else { // item found!
        if (root->left == NULL || root->right == NULL) {
            binary_tree_node<T>* tmp;
            if (root->left == NULL) tmp = root->right;
            else tmp = root->left;
            delete root;
            root = tmp;
        }
        else {
            binary_tree_node<T>* tmp = root->left; // find rightmost node
            binary_tree_node<T>* parent = root;     // in left subtree
            while (tmp->right != NULL) {
                parent = tmp;
                tmp = tmp->right;
            }
            root->data = tmp->data;        // copy data to root
            if (parent == root)            // detach and delete rightmost node
                root->left = tmp->left;     // in left subtree
            else
                parent->right = tmp->left;
            delete tmp;
        }
    }
}
```

Practice With Trees Time

Analysis

What is the "big-O" complexity of:
– Searching?
– Inserting?
– Removing?

Complexity of Search

Height of Trees

So how high is a tree with N nodes?
Height-Balanced Trees (AVL)

Again, a recursive definition:

- Left and right subtrees are height-balanced
- Left and right subtrees differ in height by no more than 1

Which of these are height balanced?

Analysis on Balanced BSTs

When trees are balanced

- Each subtree contains roughly half the nodes
- Simplifies construction of recurrence relations

Recurrence for search:

\[ T(N) = O(1) + T(N/2) \]

Search, insert, delete all \( O(\log N) \)

Self Balancing BSTs

- Trees become unbalanced through series of inserts and deletes
- Self-balancing: perform \( O(\log N) \) or fewer operations to rebalance after insert, delete
- Examples of self-balancing BSTs:
  - Red-Black trees
  - AVL trees
  - Splay trees

Rotations

We change the balance at a node via a rotation:

This is the right rotation. The left rotation is the mirror image of this one.

If the tree shown was a BST, is the new tree a BST?

Rebalancing Example

Removing this node unbalances the tree at e.

Red-Black Trees

RB-Tree basics (so you can follow the video):

- No two consecutive red nodes on any descending path
- All leaves are black
- All paths from root to leaf have the same # of black nodes

Watch the video:

http://www.youtube.com/watch?v=vDHFF4wjWYU
Final Words

Why BSTs matter:
- Linux kernel: schedulers, ext3 filesystem, virtual memory, many more (Red-Black trees)
- Ordered set and map types (e.g., C++ STL, Java) (Red-Black trees again!)
- Database indexing (B-trees – not exactly BSTs, but related)
- Filesystem metadata indexing (B-trees or R-B)
- Lurking in your favorite OS?