CS 106B, Lecture 21
Binary Search Trees
Plan for Today

• How to implement a Set
• Modifying Trees
  – Contains
  – getMin/getMax
  – Add
  – FreeTree
  – Removal
Exercise: contains

• Write a function `contains` that accepts a tree node pointer as its parameter and searches the tree for a given integer, returning `true` if found and `false` if not.

  • `contains(root, 87) → true`
  • `contains(root, 60) → true`
  • `contains(root, 63) → false`
  • `contains(root, 44) → false`
// Returns whether this BST contains the given integer.  
// Assumes that the given tree is in valid BST order.  
bool contains(TreeNode* node, int value) {  
  if (node == nullptr) {  
    return false;  // base case: not found here  
  } else if (node->data == value) {  
    return true;  // base case: found here  
  } else if (node->data > value) {  
    return contains(node->left, value);  
  } else {  // root->data < value  
    return contains(node->right, value);  
  }  
}
getMin/getMax

- Sorted arrays can find the smallest or largest element in $O(1)$ time.
- How could we get the same values in a binary search tree?
// Returns the minimum/maximum value from this BST.
// Assumes that the tree is a nonempty valid valid BST.

int getMin(TreeNode* root) {
    if (root->left == nullptr) {
        return root->data;
    } else {
        return getMin(root->left);
    }
}

int getMax(TreeNode* root) {
    if (root->left == nullptr) {
        return root->data;
    } else {
        return getMax(root->left);
    }
}
Adding to a BST

• Suppose we want to add new values to the BST below.
  – Where should the value 14 be added?
  – Where should 3 be added? 7?
  – If the tree is empty, where should a new value be added?
• Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:
Exercise: add

- Write a function **add** that adds a given integer value to the BST.
  - Add the new value in the proper place to maintain BST ordering.

  ```
  tree.add(root, 49);
  ```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}

• Must pass the current node by reference for changes to be seen.
Announcements

• Assn. 5 is due today

• Assn. 6 goes out today
  – Linked Lists. Give yourself plenty of time!

• One extra free late day for filling out the survey
  – Thanks for the feedback!
Free Tree

- To avoid leaking memory when discarding a tree, we must free the memory for every node.
  - Like most tree problems, often written *recursively*
  - must free the node itself, and its left/right subtrees

- this is another *traversal* of the tree
  - should it be pre-, in-, or post-order?

```
root

55
  29
  42
  36

87
  60
  91
  73
```
void freeTree(TreeNode* node) {
    if (node == nullptr) {
        return;
    }
    freeTree(node->left);
    freeTree(node->right);
    delete node;
}
• Suppose we want to **remove** values from the BST below.
  – Removing a leaf like 4 or 22 is easy.
  – What about removing 2? 19?
  – How can you remove a node with two large subtrees under it, such as 15 or 9?
Cases for removal

1. a leaf:
   - Remove with nullptr

2. a node with a **left child only**:
   - Replace with left child

3. a node with a **right child only**:
   - Replace with right child

```
remove(root, 17);
remove(root, 29);
remove(root, 55);
remove(root, 29);
remove(root, 42);
```
4. a node with both children:
   replace with min from right
   (replacing with max from left would also work)

```python
remove(root, 55);
```
Exercise: remove

• Add a function `remove` that accepts a root pointer and removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.

  • `remove(root, 73);`
  • `remove(root, 29);`
  • `remove(root, 87);`
  • `remove(root, 55);`
// Removes the given value from this BST, if it exists. 
// Assumes that the given tree is in valid BST order.
void remove(TreeNode*& node, int value) {
    if (node == nullptr) {
        return;
    } else if (value < node->data) {
        remove(node->left, value);  // too small; go left
    } else if (value > node->data) {
        remove(node->right, value);  // too big; go right
    } else {
        // value == node->data; remove this node!
        // (continued on next slide)
        ...
    }
}
remove solution

// value == node->data; remove this node!
if (node->right == nullptr) {
    // case 1 or 2: no R child; replace w/ left
    TreeNode* trash = node;
    node = node->left;
    delete trash;
} else if (node->left == nullptr) {
    // case 3: no L child; replace w/ right
    TreeNode* trash = node;
    node = node->right;
    delete trash;
} else {
    // case 4: L+R both; replace w/ min from right
    int min = getMin(node->right);
    remove(node->right, min);
    node->data = min;
}
Overflow

• We saw how to add to a binary search tree. Does it matter what order we add in?
  – Try adding: 50, 20, 75, 98, 80, 31, 150
  – Now add the same numbers but in sorted order: 20, 31, 50, 75, 80, 98, 150