Data Structures Lab

Keeping Your Balance
Assignment 2 Reviewed

● Assignment 2 is not graded yet
  ○ Hopefully over the weekend

● More trouble than in previous assignments
  ○ Trouble coding BST
  ○ Trouble testing BST

● Assignment 3 is effectively an extension to assignment 2
  ○ If your grade improves on assn3, I'll use it instead
Assignment 2 Reviewed

● Coding issues
  ○ This was a hard problem

● I have office hours - come to them!
  ○ Monday 1-2 (office)
  ○ Tuesday 1-2 (office)
  ○ Tuesday 5-6 (lab)
  ○ Wednesday 5-6 (lab)
  ○ Thursday (lab)
  ○ Friday (lab)

● If you can't make my hours or you need more, let me know
  ○ I will make time to help you
Assignment 2 Reviewed

- Testing issues
  - Make sure your data structure actually works!
  - Don't just run the problem tests

- The assignment problem is not the goal of the assignment
  - The goal is to code a data structure
  - The problem is just to provide motivation

- If the data structure doesn't work, the problem won't either
Assignment 2 Reviewed

- Don't work on the problem until your data structure works
  - Your first main method should have nothing to do with diamonds or taxes
  - Write a test method

- You can't test a tree unless you can see it
  - Write a print method
  - Required for assignment 3
BST Remove

- Hardest part of implementing BST
  - Most of you understood conceptually
  - Implementation problems
  - Reattaching parent

- References are your friend
  - (most of the time)

- Let work through a remove function
BST Remove

void BinTree::remove(int n) { remove(n, root); }

void BinTree::remove(int n, Node* & curr){
void BinTree::remove(int n) { remove(n, root); }

void BinTree::remove(int n, Node* & curr){
    if (curr == NULL) //n is not in this tree
        return;
    else if (n < curr->data) //n is in the left subtree
        remove(n, curr->left);
    else if (n > curr->data) //n is in the right subtree
        remove(n, curr->right);
    else if (n == curr->data){ //Ah, found it!
        // Code to handle the case where n is found
    }
}
What's wrong with this code?

//node has no children
if (curr → left == NULL && curr → right == NULL) {
    Node* & temp = curr;
    curr = NULL;
    delete temp;
}
What's wrong with this code?

//node has no children
if (curr → left == NULL && curr → right == NULL) {
    Node* temp = curr;
    curr = NULL;
    delete temp;
}
BST Remove

What do we need to change in this case?

//node has only left child
if (curr → left != NULL && curr → right == NULL) {
    Node* temp = curr;
    curr = NULL;
    curr = NULL;
    delete temp;
}
What do we need to change in this case?

//node has only left child
if (curr → left != NULL && curr → right == NULL) {
    Node* temp = curr;
    curr = curr → left;
    delete temp;
}
BST Remove

What's wrong with this code?

//node has two children
if (curr → left != NULL && curr → right != NULL)

//find the in-order predecessor
Node* & temp = curr->left;
while (temp→ right != NULL)
    temp = temp → right;

//swap up value and remove lower node
curr → value = temp → value;
remove(temp → value, temp);
What's wrong with this code?

//node has two children
if (temp → left != NULL && temp → right != NULL)

//find the in-order predecessor
Node* & temp = getMax(curr->left);

//while (curr → right != NULL) THIS CODE WILL CHANGE
// curr = curr → right;

//swap up value and remove lower node
curr → value = temp → value;
remove(temp → value, temp);
How do we feel about this function?

Node* getMax(Node* curr){
    if (curr \rightarrow right \neq \textbf{NULL})
        return getMax(curr \rightarrow right);
    else
        return curr;
}
How do we feel about this function?

```c
Node* & getMax(Node* & curr){
    if (curr → right != NULL)
        return getMax(curr → right);
    else
        return curr;
}
```
BST Remove

- This remove code is around 30 lines long
  - Plus a 5 line getMax function

- Some implementations necessarily require more space
  - Special cases for root
  - Switching between in-order predecessor/successor

- If your methods are over 100 lines long, there's a problem
  - Refactor your cases
  - Break into smaller functions
Assignment 3

- Due Friday, February 24
  - (One week from Friday)

- Implement a Balanced Search Tree
  - AVL Tree
  - 2-3-4 Tree
  - Red-Black Tree

- We'll be going over AVL implementation details
  - Feel free to challenge yourself with a different tree

- We'll be using C++ pseudocode
  - Feel free to challenge yourself with another language
Assignment 3

- Implement a balanced search tree
- Must support the following methods:
  - void insert(int n)
  - void remove(int n)
  - int next(int n)
  - void print()
- Must maintain tree balance
  - When inserting numbers in increasing order
  - When inserting numbers in decreasing order
  - When inserting numbers in random order
Assignment 3

- Follow assignment naming conventions:
  - BalancedTree.h
  - BalancedTree.cpp
  - assn3.cpp

- Before starting assn3.cpp, write test.cpp
  - Extend it as you implement more functionality
  - Test as you code
Restructuring your BST

- To balance our tree, we need to be able to modify it
  - But still maintain order
  - Otherwise, we'll lose track of our elements
Restructuring your BST

- How could we restructure this tree?
Restructuring your BST

- How could we restructure this tree?
  - Right rotation
  - b becomes the root
  - d becomes its child
  - C switches ownership
Restructuring your BST

- We can rotate in either direction
- Rotating doesn't change tree order, only tree structure
Restructuring your BST

- Let's write `rotateRight`
Restructuring your BST

- Let's write `rotateRight`

```c
void rotateRight(node* & curr){
    Node* temp = curr->left;
    curr->left = temp->right;
    temp->right = curr;
    curr = temp;
}
```
Restructuring your BST

* Our current insert function inserts new nodes as leaves
  ○ What if we wanted to add new nodes at the root?
  ○ How could tree rotations help?

* Let's write `insertAtRoot`
  ○ Feel free to use `rotateLeft` and `rotateRight`
Restructuring your BST

if (curr == NULL)
    curr = new Node(n);

if (n < curr->data){
    insert(n, curr->left);
    rotateRight(curr);
}

else if (n > curr->data){
    insert(n, curr->right);
    rotateLeft(curr);
}

Recursively call insert on the way down

Restructure tree on the way up

This technique might be useful elsewhere...
Balancing your BST

- Tree rotations modify structure while maintaining order
- How can we use rotation to keep our tree balanced?
  - Left rotate moves nodes to the left
  - Right rotate moves nodes to the right
- Let's keep track of the balance of our tree
  - If it gets unbalanced, we'll rotate to rebalance it
- What does it mean for a tree to be balanced?
AVL Trees

- The height of a node's subtrees can't differ by more than one

- What operations cause a tree to become unbalanced?
  - Insert
  - Delete

- Spend time inserting and deleting to save time finding
  - All three operations are guaranteed $O(\log n)$
AVL Trees - Rebalancing

- How do we rebalance an unbalanced tree?
  - Depends on what it looks like
- Let's look at some trees
AVL Trees - Rebalancing

How could we balance this tree?
AVL Trees - Rebalancing

How could we balance this tree?
AVL Trees - Rebalancing

How about this one?
AVL Trees - Rebalancing

How about this one?

This won't work
AVL Trees - Rebalancing

How about this one?

Let's try multiple rotations
AVL Trees - Rebalancing

How about this one?

Right Rotate (at f)

Let's try multiple rotations
AVL Trees - Rebalancing

How about this one?

Let's try multiple rotations
AVL Trees - Rebalancing

- Right-Right
  - Rotate left at root

- Right-Left
  - Rotate right below root (creating a RR imbalance)
  - Rotate left at root

- Left-Left
  - Rotate right at root

- Left-Right
  - Rotate left below root (creating a LL imbalance)
  - Rotate right at root
AVL Trees - Implementation

- **Suggested functions:**
  - void rotateLeft(Node* & curr)
  - void rotateRight(Node* & curr)
  - void balance(Node* & curr)

- rotateLeft and rotateRight are atomic operations.

- balance determines which rotations are necessary to balance a given node, and performs them.
AVL Trees - Implementation

● How do we know when to rebalance?
  ○ Each node needs to keep track of extra information

● Balance factor
  ○ Difference between subtree heights
  ○ Easy to determine when to rebalance
  ○ Tricky to implement

● Subtree height
  ○ Need to manually determine balance factor
  ○ Easier to implement
AVL Trees - Implementation

- After modifying our tree, many nodes may be off balance
  - Where do we start rotating?

- Work from the bottom up
  - Assume lower down nodes are balanced
  - Make the same guarantee to parent nodes

- But we may have modified the bottom of a very large tree
  - Need to rebalance at every node on the way to the root
  - How do we trace our way back up?
AVL Trees - Implementation

- Recursion to the rescue!
  - (remember `insertAtRoot`?)

- Perform balance checks after each recursive call
  - Check balance factor
  - Rebalance if necessary
  - Update node height
AVL Trees - Implementation

● Make sure to test your tree thoroughly

● BST bugs
  ○ Elements are not inserted or removed correctly
  ○ Easy to test for

● Balance errors
  ○ Tree doesn't balance correctly
  ○ Harder to test for

● Print methods are your friend
  ○ Print elements of tree
  ○ Print heights of nodes in tree