Outline

1. Expression Trees

2. Binary Search Trees
   - Searching a BST
   - Inserting into a BST
   - Deleting from a BST
   - Count-Balancing a BST

3. Assignment
Definition (Expression Tree)

A binary expression tree is a binary tree that is used to represent an expressions with binary operators. Each interior node represents an operator. At each interior node, the left subtree represents the left operand and the right subtree represents the right operand.

- As a consequence, each terminal node represents an operand.
- The order of operation is indicated by the structure of the tree.
For example, \((3 + 4) \times 5\) may be represented as
If there is more than one operator, the order of operation is indicated by the structure of the tree.
Perform an in-order traversal of the expression tree and print the nodes.
Traversals and Expression Trees

- Perform a post-order traversal to evaluate the expression tree.
Definition (Binary Search Tree)

A **binary search tree** is a binary tree with the following properties.

- There is a **total order** relation on the members in the tree.
- At every node, every member of the left subtree is less than or equal to the node value.
- At every node, every member of the right subtree is greater than or equal to the node value.
A Binary Search Tree

Binary Tree Applications
Robb T. Koether

Expression Trees

Binary Search Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a BST

Assignment
The `BinarySearchTree` class is implemented as a subclass of the `BinaryTree` class.
Mutators

- **void insert(const T& value);**
  Insert a new node containing the value into the binary search tree.

- **void remove(const T& value);**
  Remove the node containing the value from the binary search tree.
Binary Search Tree Interface

Other Member Functions

- `T* search(const T& value) const;`
  Search the binary search tree for the value. Return a pointer to the node where the value is found. Return `NULL` if the value is not found.

- `void countBalance();`
  Count-balance the binary search tree.
Searching a Binary Search Tree

- Beginning at the root node, apply the following steps recursively.
  - Compare the value to the node data.
  - If it is equal, you are done.
  - If it is less, search the left subtree.
  - If it is greater, search the right subtree.
  - If the subtree is empty, the value is not in the tree.
Inserting a Value into a BinarySearchTree

Inserting into a Binary Search Tree

- Beginning at the root node, apply the following steps recursively.
  - Compare the value to the node data.
  - If it is less (or equal), continue recursively with the left subtree.
  - If is is greater, continue recursively with the right subtree.
  - When the subtree is empty, attach the node as a subtree.
Deleting a Value from a BinarySearchTree

- Perform a search to locate the value.
- This node will have
  - Two children, or
  - One child, or
  - No child.
A Binary Search Tree
Deleting a Value from a Binary Search Tree

Case 1: No Child

- Delete the node.
Delete a Node with No Child
Delete a Node with No Child

- Binary Tree Applications
  - Robb T. Koether

- Expression Trees
- Binary Search Trees
  - Searching a BST
  - Inserting into a BST
  - Deleting from a BST
  - Count-Balancing a BST

- Assignment

```
50
30 80
20 40 60 90
70
110
130
140
100
120
```
Deleting a Value from a BinarySearchTree

Case 2: One Child

- Replace the node with the subtree of which the child is the root.
Delete a Node with Two Children

Binary Tree Applications
Robb T. Koether

Expression Trees

Binary Search Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a BST

Assignment
Delete a Node with Two Children

The diagram shows a binary search tree with the following structure:

- **Root**: 100
- **Left Child of Root**: 50
  - **Left Child of 50**: 30
    - **Left Child of 30**: 20
    - **Right Child of 30**: 40
  - **Right Child of 50**: 80
    - **Left Child of 80**: 60
    - **Right Child of 80**: 90
- **Right Child of Root**: 130
  - **Left Child of 130**: 120
  - **Right Child of 130**: 140
Delete a Node with Two Children

```
100
 /   \
50    130
 / \
30   80
 / \
20   90
 / \
10   120
```

Assignment

Delete a Node with Two Children
Case 3: Two Children

- Locate the next smaller value in the tree. This value is the rightmost value of the left subtree.
  - Move left one step.
  - Move right as far as possible.
- Swap this value with the value to be deleted.
- The node to be deleted now has at most one child.
Delete a Node with Three Children

The image shows a binary search tree with a red node at the root labeled 50. This node has three children: 30, 80, and 110. Node 30 has children 20 and 40, node 80 has children 60 and 90, and node 110 has children 120 and 130.
Delete a Node with Three Children
Delete a Node with Three Children

Binary Tree Applications
Robb T. Koether

Expression Trees

Binary Search Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a BST

Assignment

Diagram of a binary tree with a node marked for deletion with three children.
Delete a Node with Three Children

Binary Tree
Applications
Robb T.
Koether

Expression
Trees

Binary Search
Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a
BST

Assignment

```
30 80
20 40
60
90
10 70
110
130
140
100
120
```
Delete a Node with Three Children

Binary Tree
Applications

Robb T. Koether

Expression
Trees

Binary Search
Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a
BST

Assignment
Count-Balancing a BinarySearchTree

Write a function `moveNodeRight()` that will move the largest value of the left subtree to the right subtree.

The `moveNodeRight()` Function

- Locate the largest value in the left subtree.
- Delete it (but save the value).
- Place it at the root.
- Insert the old root value into the right subtree.
A Binary Search Tree

- Searching a BST
- Inserting into a BST
- Deleting from a BST
- Count-Balancing a BST

Binary Tree Applications
Robb T. Koether

Expression Trees
Binary Search Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a BST
Assignment
A Binary Search Tree

Binary Search Trees
- Searching a BST
- Inserting into a BST
- Deleting from a BST
- Count-Balancing a BST

Assignment
A Binary Search Tree

Binary Tree Applications

Robb T. Koether

Expression Trees

Binary Search Trees
Searching a BST
Inserting into a BST
Deleting from a BST
Count-Balancing a BST

Assignment
Count-Balancing a BinarySearchTree

Count-Balancing a Tree

- Write a similar function `moveNodeLeft()`.
- Apply either `moveNodeRight()` or `moveNodeLeft()` repeatedly at the root node until the tree is balanced at the root.
- Then apply these functions recursively, down to the leaves.
Suppose we wish to transmit the nodes of a balanced binary search tree to another computer and reconstruct the tree there.

In what order should the values be transmitted?
We could use an in-order traversal to transmit them.

At the receiving end, simply call `insert()` to insert each value into the tree.

The constructed tree will be identical to the original.

What do we get if we transmit the values using a pre-order traversal?

Using a post-order traversal?
BinarySearchTree Implementation

BinarySearchTree Class

- Download `binarysearchtree.h`
- Download and run `BinarySearchTreeTest.cpp`
Assignment

Homework

Read Section 12.4, pages 667 - 694.