Working with Binary Search Trees

MAT 102 - Data Structures
Friday, April 11, 2003
Binary Search Trees

• A binary search tree has the property that the item in a node is greater than every item in the node’s left subtree and less than every item in the node’s right subtree.

• If you display a binary search tree using “inorder” traversal, the items will be shown from smallest to largest.
Inserting an Item

• One very useful technique for inserting a new item into a binary search tree is to use recursion.
  – First compare the new item to the root
  – If the new item is smaller, insert the item into the root’s left subtree.
  – If the new item is larger, insert the item into the root’s right subtree.
void BinaryTree::insert(TreeNode *&treePtr,
    const TreeItemType &newItem)
{
    if (treePtr == NULL)
    {
        cout << "adding node" << endl;
        treePtr = new TreeNode(newItem, NULL, NULL);
        if (treePtr == NULL)
            panic("Error: Could not allocate memory for new node.");
    }
    else if (newItem < treePtr->item)
    {
        cout << "inserting into left subtree" << endl;
        insert(treePtr->leftChildPtr, newItem);
    }
    else if (newItem > treePtr->item)
    {
        cout << "inserting into right subtree" << endl;
        insert(treePtr->rightChildPtr, newItem);
    }
    else
    {
        cout << "item already exists in the tree" << endl;
    }
}
Restricting access when working with types implemented using pointers

• When you implement an ADT with pointers, you must be careful not to let any pointers be accessed by public functions.
• If a pointer’s value is changed in an uncontrolled way, data can be lost and memory can be leaked.
• One way to restrict pointer access but retain the power of pointers is to use a wrapper function.
• A wrapper function is a public function whose only purpose is to call a private function with restricted arguments.
// this function is public and calls the
// protected/private function insert with
// the pointer “root”

void BinaryTree::insertItem(const TreeItemType &newItem)
{
    insert(root, newItem);
}

// recall that in the BinaryTree class, the
// private variable “root” is a pointer to the
// top of the tree

// then the insert function will step through
// the right or left subtree of root based
// on whether newItem is larger or smaller than
// the item at the root of the tree
Deleting an Item

• Deleting an item from a binary search tree is a little more complicated than inserting an item.

• First, of course, you find the item to be deleted. Then there are three cases to consider.
  – The item to be deleted is at a leaf node.
  – The item to be deleted has one child.
  – The item to be deleted has two children.
Case one, deleting a leaf

- If the item to be deleted is at a leaf node, you can simply change the pointer that points to the item to NULL and free the space for that node.
Case two, the item has one child

- If the item to be deleted has one child, you can simply let the item’s parent “adopt” the item’s child.
Case three, the item has two children

• If the item to be deleted (call it’s node N) has two children, follow the following steps:
  – locate another node M that is easier to remove from the tree than the node N
  – copy the item that is in M to N, thus effectively deleted the item originally in node N
  – remove the node M from the tree

• This is not a trivial process, not any node M will do.