Lecture Notes 16 – Binary search trees

CSS 501 – Data Structures and Object-Oriented Programming – Professor Clark F. Olson

Reading: Carrano, Chapter 10.3-10.4

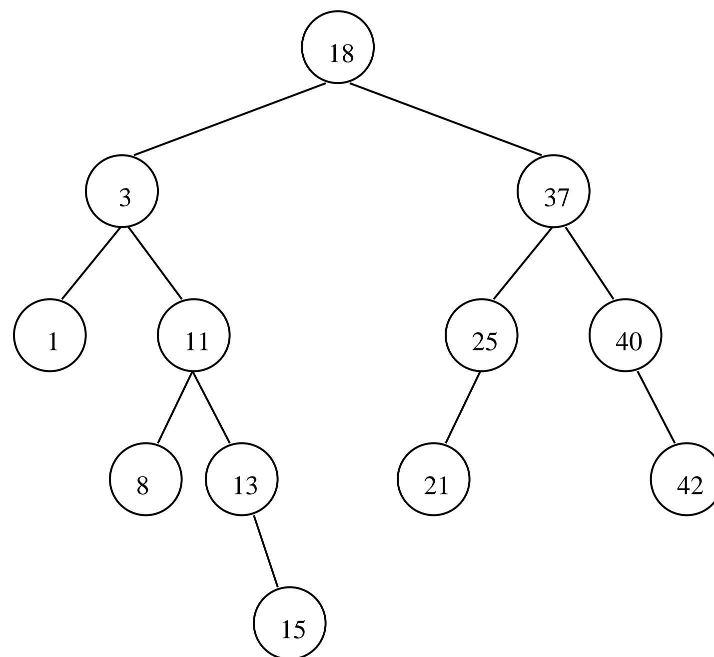
Binary search trees

A *binary search tree* is a special type of binary tree. For each node *n* in a binary search tree, the value of the item stored by *n* is greater than all of the values in the left subtree and less than all of the values in the right subtree. This implies that **all subtrees must be binary search trees**. This type of tree is useful for performing fast lookup (like binary search) and can be used to sort arrays. The primary advantage of this structure over an array is that you can efficiently insert and delete items. The disadvantage is that you don't have directed access to the *i*th element in the sorted list. Note that items that are equal can be put on the left or right or discarded, depending on the application.

In a general binary tree, it is time consuming to find a particular item, since we may need to traverse the entire tree to find the item that we are looking for. This is much easier in a binary search tree, see we know which subtree the item must be in for any ancestor node. This is similar to the difference between sequential search and binary search, except that a general binary search tree does not guarantee that the problem size is halved (one subtree may be much bigger than the other is).

Creating a binary search tree

Let's create a binary search tree storing the following values: 18 3 37 11 25 21 40 8 13 1 42 15



Programming a binary search tree

Once we have a binary search tree, we can find an item efficiently (usually) by taking the appropriate branch of the tree at each step:

```
const Object * retrieve(const BinaryTreeNode *root, const Object &key) {
    if (root == NULL) return NULL;
    else if (key == *root->item)
        return root->item;
    else if (key < *root->item)
        return retrieve(root->leftChild, key);
    else
        return retrieve(root->rightChild, key);
}
```

Inserting an item is straightforward:

```
void insert(BinaryTreeNode *&root, Object *item) {
    if (root == NULL) {
        root = new BinaryTreeNode;
        root->item = item;
        root->leftChild = NULL;
        root->rightChild = NULL;
    }
    else if (*item < *root->item)
        insert(root->leftChild, item);
    else
        insert(root->rightChild, item);
    // What if they are equal?
}
```

This can be done non-recursively, but the code is messier:

```
void insert(BinaryTreeNode *&root, Object *item) {
    BinaryTreeNode *node = new BinaryTreeNode;
    node->item = item;
    node->leftChild = NULL;
    node->rightChild = NULL;

    if  (root == NULL)
        root = node;
    else {
        BinaryTreeNode *cur = root;
        while (true) {
            if (*item < *cur->item)
                if  (cur->leftChild == NULL) {
                    cur->leftChild = node;
                    return;
                }
                else cur = cur->leftChild;
            else
                if (cur->rightChild == NULL) {
                    cur->rightChild = node;
                    return;
                }
                else cur = cur->rightChild;
        }
    }
}
```

When you want to remove an item from a binary search tree, finding the item is easy, but deleting it is a little trickier if the node has two children. If the node has zero or one children, we can delete the node easily and replace the pointer to it (with the child if one exists). If the node containing the item has two children, we must find a replacement item to place in the node. This replacement item is either the largest descendent of the left child or the smallest descendent of the right child (which are the next smaller and next larger item in the tree).

```
bool deleteNode(BinaryTreeNode *&root, const Object &item)
{
        if (root == NULL)
                return false;
        else if (item == *root->item)
        {
                deleteRoot(root);
                return true;
        }
        else if (item < *root->item)
                return deleteNode(root->leftChild, item);
        else
                return deleteNode(root->rightChild, item);
}

void deleteRoot(BinaryTreeNode *&root)
{
        if (root->leftChild == NULL && root->rightChild == NULL)
        {
                delete root->item;
                delete root;
                root = NULL;
        }
        else if (root->leftChild == NULL)
        {
                BinaryTreeNode *tmp = root;
                root = root->rightChild;
                delete tmp->item;
                delete tmp;
        }
        else if (root->rightChild == NULL)
        {
                BinaryTreeNode *tmp = root;
                root = root->leftChild;
                delete tmp->item;
                delete tmp;
        }
        else
        {
                delete root->item;
                root->item = findAndDeleteMostLeft(root->rightChild);
        }
}
```

```
// Pre-condition: root is not NULL
Object *findAndDeleteMostLeft(BinaryTreeNode *&root)
{
	if (root->leftChild == NULL)
	{
		Object  *item = root->item;
		BinaryTreeNode *junk = root;
		root = root->rightChild;
		delete junk;
		return item;
	}
	else
		return findAndDeleteMostLeft(root->leftChild);
}
```

Binary search tree efficiency

The efficiency of operations on a binary search depends on how balanced the tree is when it is built. Like quick sort, if a value that is smaller (or larger) than all of the current elements is inserted at each step, then we will get poor efficiency. We will discuss balanced binary tree structures that deal with this problem in a few weeks. In the worst case, insertion, deletion, and retrieval take O(n) time, where n is the number of nodes in the tree. In the best case, the tree will be perfectly balanced (for each node, the two subtrees will never have a height difference greater than one). In this case, the height of the tree is O(log n) and insertion, deletion, and retrieval can all be performed in O(log n) time, since, at worst, they require traversing to a leaf of the tree. It turns out that the average case (if it is assumed that each relative position in sorted order is equally likely at each step,) is also O(log n). However, special cases, such as arrays that are already sorted, can cause problems. Traversal of the tree is always O(n), since you have to visit each node.

Sorting is easy with the help of a binary search tree. Simply insert each item into the tree and read them back out using an inorder traverse. How long does this take? In the worst case, it will require O(n^2) to insert them all and O(n) to read them back out. It the best/average case, it requires O(log n) to insert each one and O(n) to read them back out for a total of O(n log n). Tree sort is very similar to quick sort. The array is essentially partitioned by the side of the root on which that each element lies.

Practice problems (optional):

Show the result of inserting 31, 17, 40, 67, 99, 23, 55, 78 in an initially empty binary search tree. Show the tree after deleting the root.

Implement a findMin or findMax method for a binary search tree.

Write a tree sort algorithm to sort a vector by creating a binary search tree and then performing an inorder traversal of the tree.