

**CMSC 15200- Intro to Computer Science 2**  
**Summer 2009**  
**Homework #6 (8/14/2009)**  
**Due Date: 8/19/2009 (before class -- 1:30 pm)**

**Exercise 1** <<15 points>>

The homework files include a class-based implementation of a Stack ADT. The items in this stack can only be of integer type. You must modify this implementation using templates to allow the Stack ADT to contain *any* data type. A main.cpp file is provided to test your implementation. The expected output is the following:

```
9 8 7 6 5 4 3 2 1
First element is 9
9 8 7 6 5 4 3 2 1
Popped element 9
8 7 6 5 4 3 2 1
Popped element 8
7 6 5 4 3 2 1
Popped element 7
6 5 4 3 2 1
Popped element 6
5 4 3 2 1
Popped element 5
4 3 2 1
Popped element 4
3 2 1
Popped element 3
2 1
Popped element 2
1
Popped element 1
Stack is empty!
5 4 3 2 1
8 7 6 5 4 3 2 1
```

**Exercise 2** <<20 points>>

PhoneCorp and PhoneTech, the two biggest phone companies in the US, have just completed a corporate merger. They are now faced with the daunting task of *merging* their client data files into a single file. In particular, each company has a text file with the social security numbers of their clients (one 9-digit number in each line) in increasing order. Your task is to create a program that takes those two files and creates a new file with the numbers from both files, in increasing order. You can assume that the two file have no numbers in common (i.e. the two sets of clients are disjoint)

For example:

Your program must be run like this:

```
ex1 <clientfile1> <clientfile2> <result>
```

A skeleton **ex2.cpp** file is provided showing how to retrieve the command-line parameters.

For full credit (10 points otherwise), your file must perform the merge doing a *single pass* through each of the files, without loading them into memory.

Note: Two example files (`clients1` and `clients2`) are provided in the homework files. The result of correctly merging these two files is the following:

```
113624982
135648623
154564564
168943568
193215689
236584654
276546846
279461322
283216569
295464654
316243213
333218954
342654658
345654858
393546562
395221354
399321545
412135468
422354943
462213589
493215812
539565482
539635482
783213542
896546543
973213215
993219953
```

Hint 2: This is a good example of an exercise you should try to first solve with a more reduced problem set, before approaching the complete problem. For example, to familiarize yourself with the merging algorithm (without dealing with all the I/O messiness), try merging two 5-position arrays (preloaded with any integers you want,

as long as they are in increasing order) into a 10-position array. When you do start to add the I/O code, first give your algorithm a try with smaller files than the ones provided (with single-digit integers, for example, which are easier to check than 9-digit numbers).

### Exercise 3 <<5 points>>

Make a simple modification to Exercise 1 so that your program will be able to handle files with common numbers (i.e. the two companies share some clients in common, so the two sets of clients are *not* disjoint). For example:

Note: Two example files (`clients1_rep` and `clients2_rep`) are provided in the homework files. The result of correctly merging these two files is the following:

```
113624982
135648623
168943568
236584654
256684623
276546846
279461322
295464654
316243213
333218954
342654658
345654858
356698823
393546562
395221354
399321545
399645652
412135468
419654332
422354943
462213589
539565482
539635482
712315465
896546543
936532132
946546523
982132132
983213223
```

In the remaining exercises you will add missing functionality to a Binary Search Tree implementation. The structure and function declarations are the following (tree.h in the homework files):

```
struct TreeNode
{
    TreeNode* left;
    int value;
    TreeNode* right;
};

typedef TreeNode* Tree;

void createTree(Tree &t);

bool insert(Tree &t, int value);
void inorder(Tree &t);
void preorder(Tree &t);
void postorder(Tree &t);
bool find(Tree &t, int value);
int height(Tree &t);
bool remove(Tree &t, int value);
```

To test your tree implementation, a main.cpp is provided in the homework files. Running this program with all the exercises correctly implemented should yield the following:

```
CREATING AND INSERTING
-----
Could not insert value 15. Already in tree.
Could not insert value 17. Already in tree.
Could not insert value 23. Already in tree.
The height of the tree is 3

TRAVERSALS
-----
4 10 12 13 15 17 23 25 29
15 10 4 13 12 23 17 29 25
4 12 13 10 17 25 29 23 15

BINARY SEARCH
-----
Value 10 is contained in the tree.
Value 29 is contained in the tree.
Value 48 is NOT contained in the tree.

REMOVAL OF NODES
-----
4 10 12 13 15 17 23 25 29
Removing node 25:
```

```
4 10 12 13 15 17 23 29
The height of the tree is 3
Removing node 13:
4 10 12 15 17 23 29
The height of the tree is 2
Removing node 15:
4 10 12 17 23 29
The height of the tree is 2
```

When debugging your program, take into account that the tree used in the main.cpp program is exactly the same as the one shown in the previous page.

## Exercise 4 <<10 points>>

The provided implementation already includes an insert function:

```
bool insert(Tree &t, int value);
```

However, it uses an iterative algorithm, which is not as readable as the recursive version of the algorithm. You must reimplement the insertion algorithm using a recursive algorithm. You are not allowed to modify the expected behaviour of the function, so don't forget that the function *must* return true if the insertion was successful and false if the tree already contains the specified value.

## Exercise 5 <<10 points>>

Implement the following function:

```
int height(Tree &t);
```

This function returns the height of the tree, which is the length of the path from the root of the tree to the lowest leaf node. Hint: Finding the height is a very simple problem if you come up with a recursive definition of a tree's height.

## Exercise 6 <<10 points>>

Implement the remove function:

```
bool remove(Tree &t, int value);
```

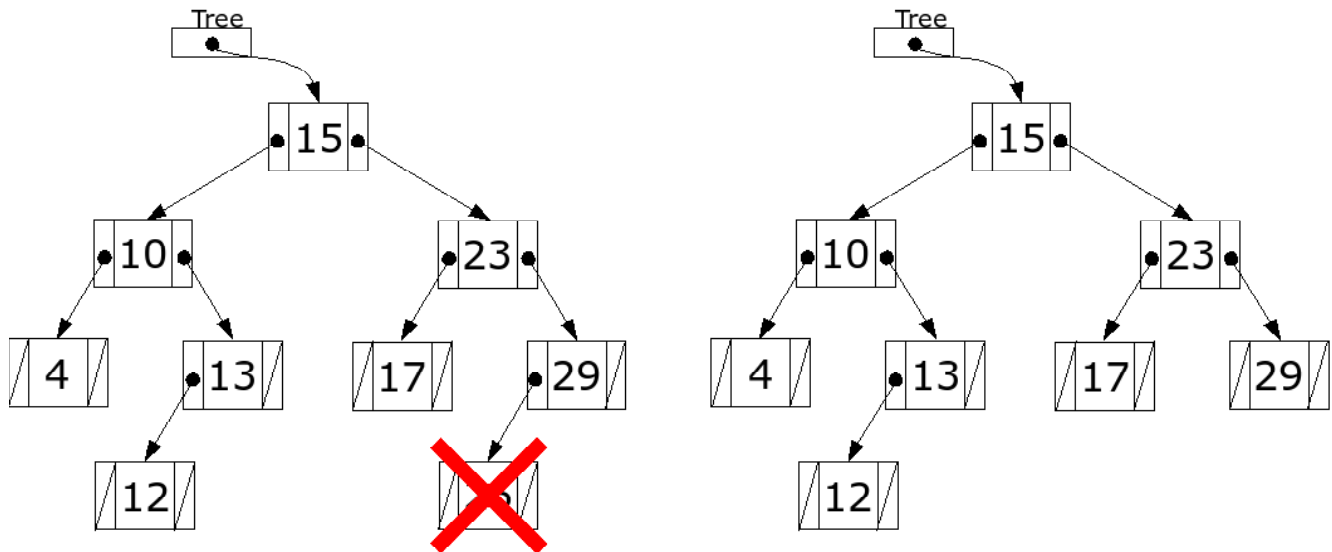
This function removes the node with the specified value. This is more complicated than the deleteSubtree method shown in the book, since we are not just cutting off an entire branch of the tree. We want to remove a specific node, and make sure that the result is still a tree (e.g., think about what would happen if you removed node 23 in the tree; what would happen to its left and right subtrees?). The function will return true

if the removal was successful and false if no such value was found.

When removing a node from the tree, you will encounter three different scenarios:

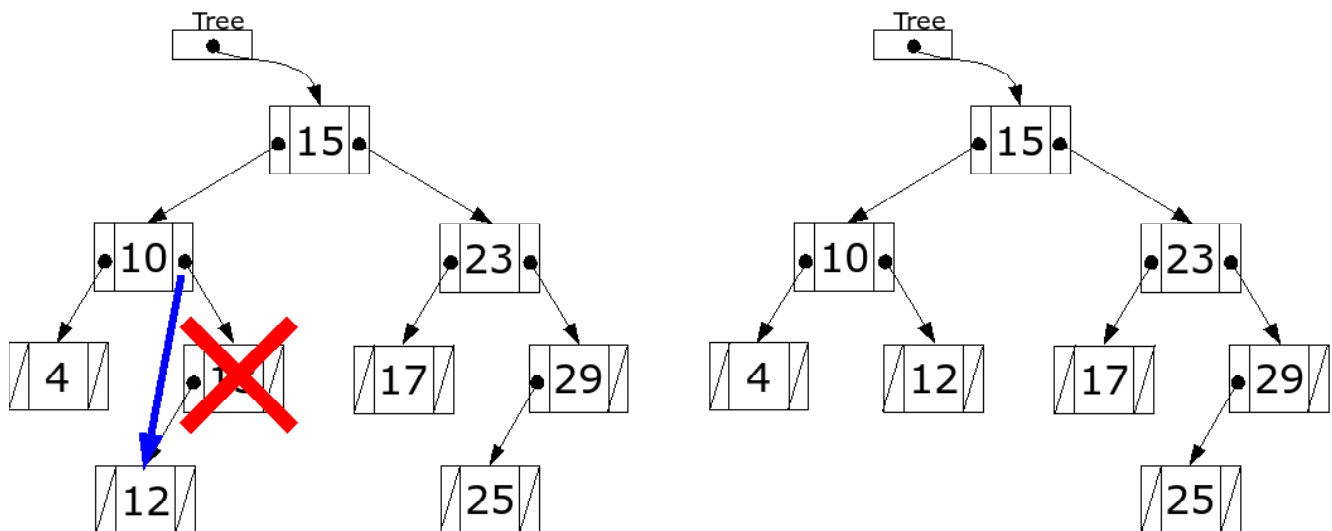
### **Removing a leaf**

This is the simplest case. Simply free the memory allocated to the node, and make sure you update the parent node to indicate that it no longer has a subtree.



### **Removing a node with a single child**

In this case, we will need to give the orphaned child a new parent. We just need to modify the node's parent so its subtree will be the orphaned subtree.



### **Removing a node with two children**

This is the most complex scenario. If we want to remove a node with two children, the first thing we need to do is find the largest node of all the nodes in the left-subtree

("largest of smaller", or LoS). For example let's suppose we want to remove node 15. The largest node in the left sub-tree is 13. We must change the value of the node we want to remove to be the value of the LoS. Finally, we will need to remove the LoS node. Take into account that, by definition, the LoS must be a node with a single child or no children, since there are no nodes with values larger than it (i.e., it will not have a right subtree). So, we will be able to remove the LoS node using any of the two previous algorithms. Note how, in this case, the node we want to remove doesn't actually get removed (we just change its value to the LoS node's value, and then remove the LoS node).

