Here are 14 binary tree problems in increasing order of difficulty. Some of the problems operate on binary search trees (aka "ordered binary trees") while others work on plain binary trees with no special ordering. The next section, Section 3, shows the solution code in C/C++. Section 4 gives the background and solution code in Java. The basic structure and recursion of the solution code is the same in both languages -- the differences are superficial.

Reading about a data structure is a fine introduction, but at some point the only way to learn is to actually try to solve some problems starting with a blank sheet of paper. To get the most out of these problems, you should at least attempt to solve them before looking at the solution. Even if your solution is not quite right, you will be building up the right skills. With any pointer-based code, it's a good idea to make memory drawings of a a few simple cases to see how the algorithm should work.

1. build123()

This is a very basic problem with a little pointer manipulation. (You can skip this problem if you are already comfortable with pointers.) Write code that builds the following little 1-2-3 binary search tree...

Write the code in three different ways...

- a: by calling newNode() three times, and using three pointer variables
- b: by calling newNode() three times, and using only one pointer variable
- c: by calling insert() three times passing it the root pointer to build up the tree

(In Java, write a build123() method that operates on the receiver to change it to be the 1-2-3 tree with the given coding constraints. See Section 4.)

```
struct node* build123() {
```

2. size()

This problem demonstrates simple binary tree traversal. Given a binary tree, count the number of nodes in the tree.

```
int size(struct node* node) {
```

3. maxDepth()

Given a binary tree, compute its "maxDepth" -- the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is 0, the maxDepth of the tree on the first page is 3.

```
int maxDepth(struct node* node) {
```

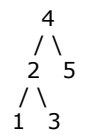
4. minValue()

Given a non-empty binary search tree (an ordered binary tree), return the minimum data value found in that tree. Note that it is not necessary to search the entire tree. A maxValue() function is structurally very similar to this function. This can be solved with recursion or with a simple while loop.

```
int minValue(struct node* node) {
```

5. printTree()

Given a binary search tree (aka an "ordered binary tree"), iterate over the nodes to print them out in increasing order. So the tree...



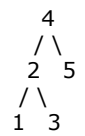
Produces the output "1 2 3 4 5". This is known as an "inorder" traversal of the tree.

Hint: For each node, the strategy is: recur left, print the node data, recur right.

```
void printTree(struct node* node) {
```

6. printPostorder()

Given a binary tree, print out the nodes of the tree according to a bottom-up "postorder" traversal -- both subtrees of a node are printed out completely before the node itself is printed, and each left subtree is printed before the right subtree. So the tree...

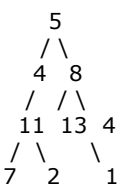


Produces the output "1 3 2 5 4". The description is complex, but the code is simple. This is the sort of bottom-up traversal that would be used, for example, to evaluate an expression tree where a node is an operation like '+' and its subtrees are, recursively, the two subexpressions for the '+'.

```
void printPostorder(struct node* node) {
```

7. hasPathSum()

We'll define a "root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). We'll say that an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths:



Root-to-leaf paths:
path 1: 5 4 11 7
path 2: 5 4 11 2
path 3: 5 8 13
path 4: 5 8 4 1

For this problem, we will be concerned with the sum of the values of such a path -- for example, the sum of the values on the 5-4-11-7 path is 5 + 4 + 11 + 7 = 27.

Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. (Thanks to Owen Astrachan for suggesting this problem.)

```
int hasPathSum(struct node* node, int sum) {
```

8. printPaths()

Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the "path so far" needs to be communicated between the recursive calls. **Hint:** In C, C++, and Java, probably the best solution is to create a recursive helper function printPathsRecur(node, int path[], int pathLen), where the path array communicates the sequence of nodes that led up to the current call. Alternately, the problem may be solved bottom-up, with each node returning its list of paths. This strategy works quite nicely in Lisp, since it can exploit the built in list and mapping primitives. (Thanks to Matthias Felleisen for suggesting this problem.)

Given a binary tree, print out all of its root-to-leaf paths, one per line.

```
void printPaths(struct node* node) {
```

9. mirror()

Change a tree so that the roles of the left and right pointers are swapped at every node.

```
 So the tree...
      4
     / \
    2   5
   / \
  1   3

 is changed to...
      4
     / \
    5   2
       / \
      3   1
```

The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the-new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree.

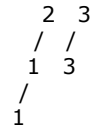
```
void mirror(struct node* node) {
```

10. doubleTree()

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

```
 So the tree...
    2
   / \
  1   3

 is changed to...
      2
     / \
```



As with the previous problem, this can be accomplished without changing the root node pointer.

void doubleTree(struct node* node) {

11. sameTree()

Given two binary trees, return true if they are structurally identical -- they are made of nodes with the same values arranged in the same way. (Thanks to Julie Zelenski for suggesting this problem.)

int sameTree(struct node* a, struct node* b) {

12. countTrees()

This is not a binary tree programming problem in the ordinary sense -- it's more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.)

Suppose you are building an N node binary search tree with the values 1..N. How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values. For example, countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it's just a counting problem.

int countTrees(int numKeys) {

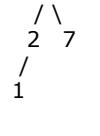
Binary Search Tree Checking (for problems 13 and 14)

This background is used by the next two problems: Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all of the nodes in its left tree must be <= the node, and all of the nodes in its right subtree must be > the node. Consider the following four examples...

a. 5 -> TRUE

b. 5 -> FALSE, because the 6 is not ok to the left of the 5

c. 5 -> TRUE



d. 5 -> FALSE, the 6 is ok with the 2, but the 6 is not ok with the 5

```
  2   7
 / \
1   6
```

For the first two cases, the right answer can be seen just by comparing each node to the two nodes immediately below it. However, the fourth case shows how checking the BST quality may depend on nodes which are several layers apart -- the 5 and the 6 in that case.

13 isBST() -- version 1

Suppose you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree (see problem 3 above). Write an isBST() function that returns true if a tree is a binary search tree and false otherwise. Use the helper functions, and don't forget to check every node in the tree. It's ok if your solution is not very efficient. (Thanks to Owen Astrachan for the idea of having this problem, and comparing it to problem 14)

Returns true if a binary tree is a binary search tree.

```
int isBST(struct node* node) {
```

14. isBST() -- version 2

Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTRecur(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX -- they narrow from there.

```
/*
 Returns true if the given tree is a binary search tree
 (efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTRecur(node, INT_MIN, INT_MAX));
}

/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
int isBSTRecur(struct node* node, int min, int max) {
```

15. Tree-List

The Tree-List problem is one of the greatest recursive pointer problems ever devised, and it happens to use binary trees as well. CLibarary #109 http://cslibrary.stanford.edu/109/ works through the Tree-List problem in detail and includes solution code in C and Java. The problem requires an understanding of binary trees, linked lists, recursion, and pointers. It's a great problem, but it's complex.

Section 3 -- C/C++ Solutions

Make an attempt to solve each problem before looking at the solution -- it's the best way to learn.

1. Build123() Solution (C/C++)

```
// call newNode() three times
struct node* build123a() {
  struct node* root = newNode(2);
  struct node* lChild = newNode(1);
  struct node* rChild = newNode(3);

  root->left = lChild;
  root->right= rChild;

  return(root);
}

// call newNode() three times, and use only one local variable
struct node* build123b() {
  struct node* root = newNode(2);
  root->left = newNode(1);
  root->right = newNode(3);

  return(root);
}


/*
 Build 123 by calling insert() three times.
 Note that the '2' must be inserted first.
*/
struct node* build123c() {
  struct node* root = NULL;
  root = insert(root, 2);
  root = insert(root, 1);
  root = insert(root, 3);
  return(root);
}
```

2. size() Solution (C/C++)

```
/*
 Compute the number of nodes in a tree.
*/
int size(struct node* node) {
  if (node==NULL) {
    return(0);
  } else {
    return(size(node->left) + 1 + size(node->right));
  }
}
```

3. maxDepth() Solution (C/C++)

```
/*
 Compute the "maxDepth" of a tree -- the number of nodes along
 the longest path from the root node down to the farthest leaf node.
*/
int maxDepth(struct node* node) {
  if (node==NULL) {
    return(0);
  }
  else {
    // compute the depth of each subtree
```

```
    int lDepth = maxDepth(node->left);
    int rDepth = maxDepth(node->right);

    // use the larger one
    if (lDepth > rDepth) return(lDepth+1);
    else return(rDepth+1);
  }
}
```

4. minValue() Solution (C/C++)

```
/*
 Given a non-empty binary search tree,
 return the minimum data value found in that tree.
 Note that the entire tree does not need to be searched.
*/
int minValue(struct node* node) {
  struct node* current = node;

  // loop down to find the leftmost leaf
  while (current->left != NULL) {
    current = current->left;
  }

  return(current->data);
}
```

5. printTree() Solution (C/C++)

```
/*
 Given a binary search tree, print out
 its data elements in increasing
 sorted order.
*/
void printTree(struct node* node) {
  if (node == NULL) return;

  printTree(node->left);
  printf("%d ", node->data);
  printTree(node->right);
}
```

6. printPostorder() Solution (C/C++)

```
/*
 Given a binary tree, print its
 nodes according to the "bottom-up"
 postorder traversal.
*/
void printPostorder(struct node* node) {
  if (node == NULL) return;

  // first recur on both subtrees
  printTree(node->left);
  printTree(node->right);

  // then deal with the node
  printf("%d ", node->data);
```

```
}
```

7. hasPathSum() Solution (C/C++)

```
/*
 Given a tree and a sum, return true if there is a path from the root
 down to a leaf, such that adding up all the values along the path
 equals the given sum.

 Strategy: subtract the node value from the sum when recurring down,
 and check to see if the sum is 0 when you run out of tree.
*/
int hasPathSum(struct node* node, int sum) {
  // return true if we run out of tree and sum==0
  if (node == NULL) {
    return(sum == 0);
  }
  else {
  // otherwise check both subtrees
    int subSum = sum - node->data;
    return(hasPathSum(node->left, subSum) ||
           hasPathSum(node->right, subSum));
  }
}
```

8. printPaths() Solution (C/C++)

```
/*
 Given a binary tree, print out all of its root-to-leaf
 paths, one per line. Uses a recursive helper to do the work.
*/
void printPaths(struct node* node) {
  int path[1000];

  printPathsRecur(node, path, 0);
}

/*
 Recursive helper function -- given a node, and an array containing
 the path from the root node up to but not including this node,
 print out all the root-leaf paths.
*/
void printPathsRecur(struct node* node, int path[], int pathLen) {
  if (node==NULL) return;

  // append this node to the path array
  path[pathLen] = node->data;
  pathLen++;

  // it's a leaf, so print the path that led to here
  if (node->left==NULL && node->right==NULL) {
    printArray(path, pathLen);
  }
  else {
  // otherwise try both subtrees
    printPathsRecur(node->left, path, pathLen);
    printPathsRecur(node->right, path, pathLen);
  }
}
```

```
// Utility that prints out an array on a line.
void printArray(int ints[], int len) {
  int i;
  for (i=0; i<len; i++) {
    printf("%d ", ints[i]);
  }
  printf("\n");
}
```

9. mirror() Solution (C/C++)

```
/*
 Change a tree so that the roles of the
 left and right pointers are swapped at every node.

 So the tree...
       4
      / \
     2   5
    / \
   1   3

 is changed to...
       4
      / \
     5   2
        / \
       3   1
*/
void mirror(struct node* node) {
  if (node==NULL) {
    return;
  }
  else {
    struct node* temp;

    // do the subtrees
    mirror(node->left);
    mirror(node->right);

    // swap the pointers in this node
    temp = node->left;
    node->left = node->right;
    node->right = temp;
  }
}
```

10. doubleTree() Solution (C/C++)

```
/*
 For each node in a binary search tree,
 create a new duplicate node, and insert
 the duplicate as the left child of the original node.
 The resulting tree should still be a binary search tree.

 So the tree...
    2
   / \
  1   3
```

```
 Is changed to...
       2
      / \
     2   3
    /   /
   1   3
  /
 1

*/
void doubleTree(struct node* node) {
  struct node* oldLeft;

  if (node==NULL) return;

  // do the subtrees
  doubleTree(node->left);
  doubleTree(node->right);

  // duplicate this node to its left
  oldLeft = node->left;
  node->left = newNode(node->data);
  node->left->left = oldLeft;
}
```

11. sameTree() Solution (C/C++)

```
/*
 Given two trees, return true if they are
 structurally identical.
*/
int sameTree(struct node* a, struct node* b) {
  // 1. both empty -> true
  if (a==NULL && b==NULL) return(true);

  // 2. both non-empty -> compare them
  else if (a!=NULL && b!=NULL) {
    return(
      a->data == b->data &&
      sameTree(a->left, b->left) &&
      sameTree(a->right, b->right)
    );
  }
  // 3. one empty, one not -> false
  else return(false);
}
```

12. countTrees() Solution (C/C++)

```
/*
 For the key values 1...numKeys, how many structurally unique
 binary search trees are possible that store those keys.

 Strategy: consider that each value could be the root.
 Recursively find the size of the left and right subtrees.
*/
int countTrees(int numKeys) {
```

```
  if (numKeys <=1) {
    return(1);
  }
  else {
    // there will be one value at the root, with whatever remains
    // on the left and right each forming their own subtrees.
    // Iterate through all the values that could be the root...
    int sum = 0;
    int left, right, root;

    for (root=1; root<=numKeys; root++) {
      left = countTrees(root - 1);
      right = countTrees(numKeys - root);

      // number of possible trees with this root == left*right
      sum += left*right;
    }

    return(sum);
  }
}
```

13. isBST1() Solution (C/C++)

```
/*
 Returns true if a binary tree is a binary search tree.
*/
int isBST(struct node* node) {
  if (node==NULL) return(true);

  // false if the max of the left is > than us

  // (bug -- an earlier version had min/max backwards here)
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  // false if the min of the right is <= than us
  if (node->right!=NULL && minValue(node->right) <= node->data)
    return(false);

  // false if, recursively, the left or right is not a BST
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  // passing all that, it's a BST
  return(true);
}
```

14. isBST2() Solution (C/C++)

```
/*
 Returns true if the given tree is a binary search tree
 (efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}
```

```
/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
int isBSTUtil(struct node* node, int min, int max) {
  if (node==NULL) return(true);


  // false if this node violates the min/max constraint
  if (node->data<min || node->data>max) return(false);


  // otherwise check the subtrees recursively,
  // tightening the min or max constraint
  return
    isBSTUtil(node->left, min, node->data) &&
    isBSTUtil(node->right, node->data+1, max)
  );
}
```