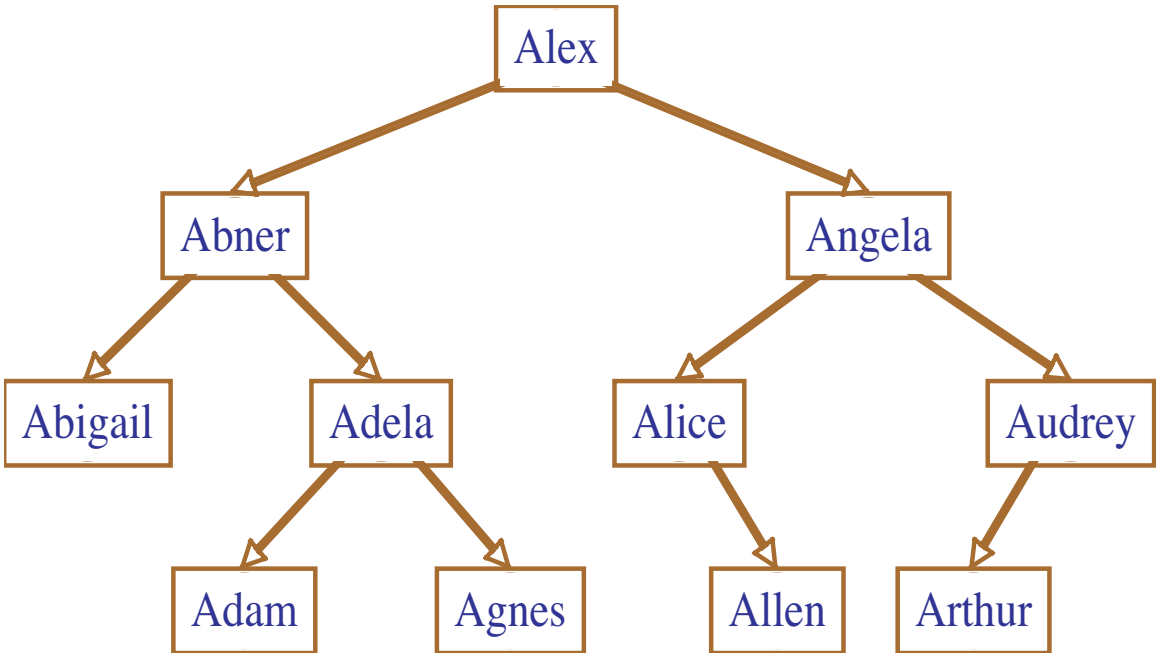
Worksheet 29:Solution: Binary Search Trees

In Preparation: Read Chapter 8 to learn more about the Bag data type, and chapter 10 to learn more about the basic features of trees. If you have not done so already, read Worksheets 21 and 22 for alternative implementation of the Bag.

In this worksheet we will start to explore how to make a useful container class using the idea of a binary tree. A *binary search tree* is a binary tree that has the following additional property: for each node, the values in all descendants to the left of the node are less than or equal to the value of the node, and the values in all descendants to the right are greater than or equal. The following is an example binary search tree:



Notice that an inorder traversal of a BST will list the elements in sorted order. The most important feature of a binary search tree is that operations can be performed by walking the tree from the top (the root) to the bottom (the leaf). This means that a BST can be used to produce a fast **Bag** implementation. For example, suppose you find out if the name "Agnes" is found in the tree shown. You simply compare the value to the root (Alex). Since Agnes comes before Alex, you travel down the left child. Next you compare "Agnes" to "Abner". Since it is larger, you travel down the right. Finally you find a node that matches the value you are searching, and so you know it is in the collection. If you find a null pointer along the path, as you would if you were searching for "Sam", you would know the value was not in the collection.

Adding a value to a binary search tree is easy. You simply perform the same type of traversal as described above, and when you find a null value you insert a new node. Try inserting the values "Amelia". Then try inserting "Sam".

Insertion is most easily accomplished by writing a private internal function that takes a Node and a value, and returns the new tree in which the Node has been inserted. In pseudo-code this routine is similar to the following:

```
Node add (Node start, E newValue)
  if start is null then return a new Node with newValue
  otherwise if newValue is less than the value at start then
    set the left child to be the value returned by add(leftChild, newValue)
  otherwise set the right child to be add(rightChild, newValue)
  return the current node
```

Removal is the most complex of the basic Bag operations. The difficulty is that removing a node leaves a "hole". Imagine, for example, removing the value "Alex" from the tree shown. What value should be used in place of the removed element?

The answer is the *leftmost child of the right node*. This is because it is this value that is the smallest element in the right subtree. The leftmost child of a node is the value found by running through left child Nodes as far as possible. The leftmost child of the original tree shown above is "Abigail". The leftmost child of the right child of the node "Alex" is the node "Alice". It is a simple matter to write a routine to find the value of the leftmost child of a node. You should verify that in each case if you remove an element the value of the node can be replaced by the leftmost child of the right node without destroying the BST property.

A companion routine (removeLeftmost) is a function to return a tree with the leftmost child removed. Again, traverse the tree until the leftmost child is found. When found, return the right child (which could possibly be null). Otherwise make a recursive call and set the left child to the value returned by the recursive call, and return the current Node.

Armed with these two routines, the general remove operation can be described as follows. Again it makes sense to write it as a recursive routine that returns the new tree with the value removed.

```
Node remove (Node start, E testValue)
  if start.value is the value we seek
    decrease the value of dataSize
    if right child is null
      return left child
    otherwise
      replace value of node with leftmost child of right child
      set right child to be removeLeftmost(right child)
  otherwise if testValue is smaller than start.value
    set left child to remove (left child, testValue)
  otherwise
    set right child to remove (right child, testValue)
  return current node
```

Try executing this function on each of the values of the original binary search tree in turn, and verifying that the result is a valid binary search tree.

Using the approach described, complete the following implementation:

struct node {
  TYPE value;
  struct node * left;
  struct node * right;
};
	struct BinarySearchTree {
	  struct node *root;
	  int size;
	};

void initBST(struct BinarySearchTree *tree) { tree->size = 0; tree->root = 0: }

void addBST(struct BinarySearchTree *tree, TYPE newValue) {
  tree->root = _nodeAddBST(tree->root, newValue); tree->size++; }

int sizeBST (struct binarySearchTree *tree) { return tree->size; }

struct node * _nodeAddBST (struct node *current, TYPE newValue) {

```
struct Node *node;

  if (current == 0) {
    node = (struct Node *)malloc(sizeof(struct Node));
    assert(node != 0);

    node->value  = newValue;
    node->left = node->rght = 0;

    return node;
  }

  if (LT(newValue, current->value))
      current->left = _addNode(current->left, newValue);
  else  current->right = _addNode(current->right, newValue);

  return current;

}
```

int containsBST (struct binarySearchTree *tree, TYPE d) {

```
    struct Node *cur = tree->root;

    while (cur != 0) {
      if (EQ(d, cur->value))
            return 1;          /* Return true if value found. */
      if (LT(d, cur->value))
            cur = cur->left;   /* Otherwise, go to the left     */
      else  cur = cur->right;  /*   or right, depending on val. */
  }

  return 0;  /* Return false if not found. */
```

```
}
void removeBST (struct binarySearchTree *tree, TYPE d) {
  if (containsBST(tree, d) {
    tree->root = _nodeRemoveBST(tree->root, d);
    tree->size--;
  }
}

TYPE _leftMostChild (struct node * current) {


    while (current->left != 0)
      current = current->left;
    return current->val;

}

struct node * _removeLeftmostChild (struct node *current) {


    struct Node *node;


 if(current->left == 0)
 {
  node = current->right;
  free(current);
  return node;

 }

  current->left = _removeLeftMost(current->left);
  return current;
}




}

void _nodeRemoveBST (struct node * current, TYPE d) {


  struct Node *node;

  if (LT(val, current->val) == 0) {     /* Found value. */
    if (current->rght == 0) {
      node = current->left;
      free(current);
      return node;
    }
    current->val  =       _leftMost(current->rght);
```

```
    current->rght = _removeLeftMost(current->rght);
  }
  else if (LT(val, current->val) < 0)
            current->left = _removeNode(current->left, val);
  else      current->rght = _removeNode(current->rght, val);

  return current;
}
```

On Your Own

1. What is the primary characteristic of a binary search tree?

2. Explain how the search for an element in a binary search tree is an example of the idea of divide and conquer.

3. Try inserting the values 1 to 10 in order into a BST. What is the height of the resulting tree?

4. Why is it important that a binary search tree remain reasonably balanced? What can happen if the tree becomes unbalanced?

5. What is the maximum height of a BST that contains 100 elements? What is the minimum height?

6. Explain why removing a value from a BST is more complicated than insertion.

7. Suppose you want to test our BST algorithms. What would be some good boundary value test cases?

8. Program a test driver for the BST algorithm and execute the operations using the test cases identified in the previous question.

9. The smallest element in a binary search tree is always found as the leftmost child of the root. Write a method getFirst to return this value, and a method removeFirst to modify the tree so as to remove this value.

10. With the methods described in the previous question, it is easy to create a data structure that stores values in a BST and implements the Priority Queue interface. Show this implementation, and describe the algorithmic execution time for each of the Priority Queue operations.

11. Suppose you wanted to add the equals method to our BST class, where two trees are considered to be equal if they have the same elements. What is the complexity of your operation?