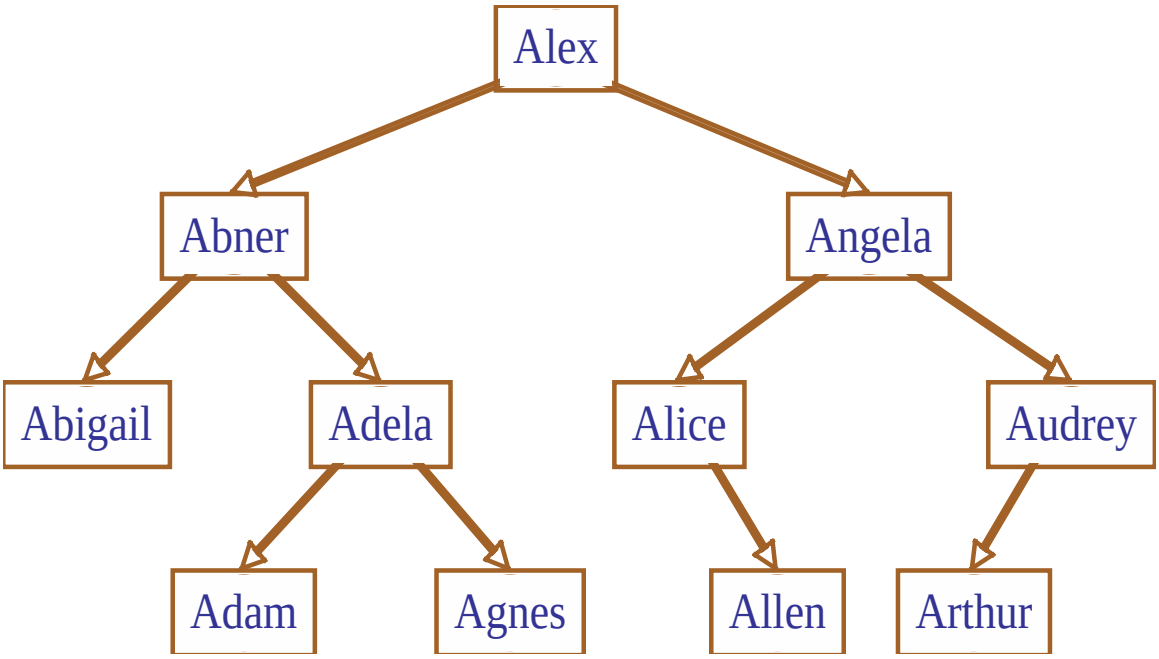
Worksheet 29: Binary Search Trees

Group 11
Michael Sigel did NOT participate

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.

worksheet 29: Binary Search Trees Name:

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 * newnode;
      if (current == 0) {
            struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
            assert(node != 0);
            newnode ->value = newValue;
            newnode ->left = newnode ->rght = 0;
            return newnode;
      }
      if (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 *current = tree->root;

       while (current != 0) {
              if (d == current->value){
              return 1;
              }
              if (d < current->value){
              current = current->left;
              }
              else current = current->right;
       }
       return 0;
}



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

}

```
//SEARCH LEFT MOST FUNCTION

TYPE _leftMostChild (struct Node * current) {
assert(!current);
TYPE temp;
Struct *Node search = current;
 while (search ->left != NULL){
            search = search ->left;
}
Temp = search ->value;
Return temp;
}


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;
}


struct Node * _nodeRemoveBST (struct Node * current, TYPE d) {
      struct Node *node;
      if (d == current->value){
            if (current->right == NULL){
                  return current->left;
            }
            else{
                  current->value = _leftMostChild(current->right);
                  current->right = _removeLeftMostChild(current->right);
            }
      else if (d < current->value){
            current->left = _removeNode(current->left, d);
      }
      else
            current->right = _removeNode(current->right, d);

      return current;
}
```

worksheet 29: Binary Search Trees Name:

On Your Own

1. What is the primary characteristic of a binary search tree?
   It provide rapid (log *n*) storage and retrieval of information because every comparison can eliminate half of the elements remaining to be searched. In the worst case, this process continues until only one item is left to examine.
2. Explain how the search for an element in a binary search tree is an example of the idea of divide and conquer.
   In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.
   Binary search tree starts with the root and branches out either to the left or to the right depending if the next value added to the binary tree if higher or lower.

3. Try inserting the values 1 to 10 in order into a BST. What is the height of the resulting tree? 9

4. Why is it important that a binary search tree remain reasonably balanced?

   When binary search tree is balanced, the binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes O(log n).
   What can happen if the tree becomes unbalanced? If the tree is not balanced than locating an item will take O(n) iterations.

5. What is the maximum height of a BST that contains 100 elements? What is the minimum height?
   The maximum height is then the tree is not branching at all (all values grow either to the left or all values grow to the right). In this case the height will be 99.
   The minimum height is achieved when we have full binary tree and each node has two children. A full binary tree of a given height h has 2h− 1 nodes. Solving this for h, we get h=(log2(n+1)), where n is # of nodes, using the log(base)A = log(A)/ log(base)propert, we have h = log(n+1) /log(2)
   In our case: h = log(n+1) /log(2) = log(100+1) /log(2) = 6.65.
   Do we round the number to 6 or 7? Do not know. I tried to draw it and it looks like we round up, so the answer is 7

   http://www.cise.ufl.edu/~sahni/cop3530/slides/lec206.pdf

6. Explain why removing a value from a BST is more complicated than insertion.
   Removing a value from a BST is more complicated than insertion because there are more steps. It is fairly simple to remove a node if it is a leaf. However, it gets more complicated when it has children. First you have to search the tree for the value, once found it is removed. Then you have to find the leftmost child of the right subtree and place it where the first node was removed.

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

   The max value in left subtree- it needs to be smaller than the node.

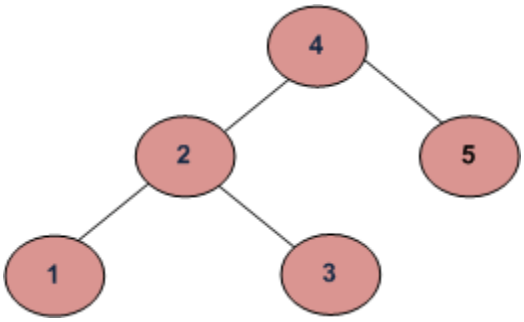
   The min value in right subtree- it needs to be greater than the node

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

   http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/

   A binary search tree (BST) is a node based binary tree data structure which has the following properties.
   • The left subtree of a node contains only nodes with keys less than the node's key.
   • The right subtree of a node contains only nodes with keys greater than the node's key.
   • Both the left and right subtrees must also be binary search trees.

   From the above properties it naturally follows that:
   • Each node (item in the tree) has a distinct key.



   For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
/* 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 */
  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);
  }
```

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

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.

```
/* 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);
}
```

To remove the smallest value, Ill just take the function provided in the worksheet:

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

And change it to something like this;

```
void removeMin (struct binarySearchTree *tree) {
  struct node* current = node;
  type min = minValue(node);
   if (containsBST(tree, min) {
      tree->root = _nodeRemoveBST(tree->root, min);
      tree->size--;
   }
}
```

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?