Worksheet 29 - Group 1

Worksheet Group 1 Members

Marc Clinedinst: clinedim@onid.oregonstate.edu
Kelby Faessler: faesslek@onid.oregonstate.edu
James Fitzwater: fitzwatj@onid.oregonstate.edu
Tom Gariepy: gariepyt@onid.oregonstate.edu
Sean Reilly: reillys@onid.oregonstate.edu
Joseph Struth: struthj@onid.oregonstate.edu

Collaborators

Marc, Kelby, James, Tom, Sean, Joseph

Worksheet 29: Binary Search Trees

In this assignment, we implement several different functions which control the behavior of a binary search tree that is implemented using dynamic memory. More specifically, we implement the `_addNode`, `_leftMostChild`, `_nodeRemoveBST`, `_removeLeftmostChild`, and `containsBSTree` functions. Each of these functions is defined below and is accompanied by comments where needed.

```
/*
      This is a recursive function which is responsible for adding a new node to a binary search
      tree.  If the current node is null, the function allocates memory for a new node, sets the
      value of the node to the passed value, and sets the left and right pointers to null. The
      function then returns the new node.  If the current node is not null, then the function
      uses recursive calls to find the appropriate null node.  Specifically, if the passed value
      is less than the current node's value, the function sets current->left to the result of
      calling the _addNode function with current->left and value as arguments.  Likewise, if
      the passed value is greater than or equal to the node's value, the function sets
```

```
     current->right to the result of calling the _addNode function with current-right and value
     as arguments.  Finally, the function returns current.
*/
struct Node *_addNode(struct Node *current, TYPE value) {
     struct Node *newNode;

     if (current == 0) {
          newNode = malloc(sizeof(struct Node));
          assert(newNode != 0);
          newNode->value = value;
          newNode->left = newNode->right = 0;
          return newNode;
     } else if (value < current->value) {
          current->left = _addNode(current->left, value);
     } else {
          current->right = _addNode(current->right, value);
     }

     return current;
}

/*
     This function returns the leftmost child of a given node within a binary search tree.
     The function first uses an assertion check to confirm that the passed node is not null.
     If this check fails, an assertion error occurs. Otherwise, the function iterates through
     the left nodes until the next left node is null.  The function then returns the value of
     the current left node.
*/
TYPE _leftMostChild(struct Node *current) {
     assert(current != 0);
```

```
	while (current->left != 0) {
		current = current->left;
	}

	return current->value;
}

/*
	This is a recursive function which removes a particular node from a binary search
	tree if it contains a passed value.  The function checks to see if the current node
	contains the target value.  If it does and the current node's right node is null,
	then the node's left node is returned.  If the current node contains the target value and
	the right node is not null, then the current value is set to the leftmost child of the
	node's right child, and the right child is set to the value returned from the
	_removeLeftmostChild function when called on current->right.  If the node does not contain
	the target value, the target value is compared to the node's value.  If the target value
	is less than the target value, the node's left value is set to the result of calling
	_nodeRemoveBST on the node's left node.  LIkewise, if the target value is greater than or
	equal to the target value, the node's right value is set to the result of calling
	_nodeRemoveBST on the node's right node.
*/
struct Node *_nodeRemoveBST(struct Node *current, TYPE value) {
	if (current->value == value) {
		if (current->right == 0) {
			struct Node *temp = current->left;
			free(current);
			return temp;
		} else {
			current->value = _leftMostChild(current->right);
			current->right = _removeLeftmostChild(current->right);
		}
```

```
	} else if (value < current->value) {
		current->left = _nodeRemoveBST(current->left, value);
	} else {
		current->right = _nodeRemoveBST(current->right, value);
	}

	return current;
}

/*
	This is a recursive function which removes the leftmost child of the current node.
	It first uses an assertion statement to check that the current node is not null. If
	this assertion check fails, an assertion error occurs.  Otherwise, the current node's
	right node is stored in a temporary variable. If the current node's left node is null,
	then the current node's memory is freed and the temporary variable is returned. Otherwise,
	the current node's left node is set to the result of calling _removeLeftmostChild and
	passing the node's left node.
*/
struct Node *_removeLeftmostChild(struct Node *current) {
	assert(current != 0);

	struct Node *temp = current->right;

	if (current->left == 0) {
		free(current);
		return temp;
	} else {
		current->left = _removeLeftmostChild(current->left);
	}

	return current;
```

```
}

/*
     This function returns an integer value indicating whether a binary search tree contains a
     particular value.  If the function returns 1, then the binary search tree contains the
     target value; if it returns 0, then the binary search tree does not contain the value.
     The function loops through a series of nodes, beginning with the root node, to attempt to
     find the target value.  During each iteration, the function compares the target value to
     the current node's value. If the values are equal, then the function returns 1.  If the
     target value is less than the current node's value, then current is set to current's left
     node. Otherwise, current is set to current's right value. Once a null node is reached, the
     function returns 0 to indicate that the value is not in the binary search tree.
*/
int containsBSTree(struct BSTree *tree, TYPE value) {
     struct Node *current = tree->root;

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

     return 0;
}
```

In addition to defining the functions above, we also need to respond to several questions about the binary search tree. These are listed below and are accompanied by answers.

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

   A linked list made of structs that contain a pointer to a left and right node.The left node is made of an element considered less than the current element, while the right node is considered greater than.

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

   Because the nodes consist of a less than and a greater than, a tree search will determine if the search value is less than, greater than, or equal to the current element. If they are not equal, the program will follow the appropriate node and remove all elements in the other node.

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

   The height of the resulting tree would be 9.

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

   It is important to keep a BST balanced because it shortens the runtime to search for an element in the tree. The more unbalanced a BST is, the longer the search runtime becomes.

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

   The maximum height of a BST of 100 elements would be 99. The minimum would be 7.

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

When you remove an element in a BST, the program must determine if the value needs to be replaced with a descendant of that value. If it does, that descendant may also need to be replaced, and the pattern goes on.

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

   Test to see how the BST algorithm handles a series of values in ascending order (ex. 15-50 in that order).

   Test to see how the BST algorithm handles a series of values in descending order (ex. 50-15 in that order).

   Test to see how the BST algorithm handles a mixed series of values. (ex. 5, 20, 25, 10, 15, 4, 6, 90, 100).
   Try to delete a leaf variable. (ex. remove 100)
   Try to delete an internal node (ex. remove 20)
   Try to delete the header variable (e. remove 5)
   Test to see how the BST algorithm handles duplicate values. (ex. add 90)

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

```
void bstTest() {
     BinarySearchTree *ascBST;

     initBST(ascBST);

     /*Test in order*/
     for (int count = 15; count <= 50; count++) {
          addBST(ascBST, count);
          if (containsBST(ascBST, count) != 0) {
               printf("%d has been added to the tree.\n", count);
          }
     }
```

```
/*Test in reverse order*/
BinarySearchTree *descBST;

initBST(descBST);
for (int count = 50; count >= 15; count--) {
     addBST(ascBST, count);
     if (containsBST(ascBST, count) != 0) {
          printf("%d has been added to the tree.\n", count);
     }
}

/*Test in mixed order*/
BinarySearchTree *randBST;

initBST(randBST);

addBST(ascBST, 5);
addBST(ascBST, 20);
addBST(ascBST, 25);
addBST(ascBST, 10);
addBST(ascBST, 15);
addBST(ascBST, 4);
addBST(ascBST, 6);
addBST(ascBST, 90);
addBST(ascBST, 100);


if (containsBST(randBST, 5) != 0) {
          printf("5 has been added to the tree.\n");
}
if (containsBST(randBST, 20) != 0) {
```

```
          printf("20 has been added to the tree.\n");
}
if (containsBST(randBST, 25) != 0) {
          printf("25 has been added to the tree.\n");
}
if (containsBST(randBST, 10) != 0) {
          printf("10 has been added to the tree.\n");
}
if (containsBST(randBST, 15) != 0) {
          printf("15 has been added to the tree.\n");
}
if (containsBST(randBST, 4) != 0) {
          printf("4 has been added to the tree.\n");
}
if (containsBST(randBST, 6) != 0) {
          printf("6 has been added to the tree.\n");
}
if (containsBST(randBST, 90) != 0) {
          printf("90 has been added to the tree.\n");
}
if (containsBST(randBST, 100) != 0) {
          printf("100 has been added to the tree.\n");
}

/*Remove leaf*/
removeBST(randBST, 100);
if (containsBST(randBST, 100) == 0) {
     printf("100 has sucessfully been removed.\n");
}

/*Remove internal node*/
```

```
	removeBST(randBST, 20);
	if (containsBST(randBST, 20) == 0) {
		printf("100 has sucessfully been removed.\n");
	}

	/*Remove header*/
	removeBST(randBST, 5);
	if (containsBST(randBST, 5) == 0) {
		printf("5 has sucessfully been removed.\n");
	}


	/*Check to see how the tree handles duplicates*/
	addBST(randBST, 90);

}
```

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.

```
TYPE getFirst(struct BinarySearchTree *tree) {
	Node *curr;
	curr = tree->root;
	while (curr->left != 0) {
		curr = curr->left;
	}

	return curr->value;
}
```

```
void removeFirst(struct BinarySearchTree *tree) {
      Node *curr;
      curr = tree->root;
      while (curr->left != 0) {
            curr = curr->left;
      }

      removeBST(tree, curr->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.

    All that would need to be done is to use the BST struct along with the addBST, getFirst, and removeFirst.

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?

    In order to clearly understand the question, we wrote out a version of the isEqual function.

```
void isEqual (struct BinarySearchTree *tree1, struct BinarySearchTree *tree2) {
      int total = 0;

      if (tree1->size == tree2->size) {
            total = isEqu(tree1->root, tree2);
```

```
            if (total == tree2->size) {
                  printf("These two are equal.\n");
            }
            else {
                  printf("These two are not equal.\n");
            }
      }
      else {
            printf("These two are not equal.\n");
      }
}

int isEqu(struct node *tree1, struct BinarySearchTree *tree2) {
      if (tree1->left != 0 && tree1->right != 0) {
            if (containsBST(tree2, tree1value) != ) {
                  return 1 + isEqu(tree1->left, tree2) + isEqu(tree1->right, tree2);
            }
            else {
                  return 0 + isEqu(tree1->left, tree2) + isEqu(tree1->right, tree2);
            }
      }
      else if (tree1->left != 0) {
            if (containsBST(tree2, tree1value)) {
                  return 1 + isEqu(tree1->left, tree2);
            }
            else {
                  return 0 + isEqu(tree1->left, tree2);
            }
      }
      else if (tree1->right != 0) {
            if (containsBST(tree2, tree1value)) {
```

```
                return 1 + isEqu(tree1->right, tree2);
            }
            else {
                return 0 + isEqu(tree1->right, tree2);
            }
        }
        else {
            if (containsBST(tree2, tree1value)) {
                return 1;
            }
            else {
                return 0;
            }
        }
}
```

Because the program would have to manually find every value in the first BST and determine if the second tree contains each value, the complexity would be O(nlogn).

Piazza Discussion

https://piazza.com/class/ib2kus4hsie528?cid=173