1. **Problem Statement**

Create a program that will organize integers into various search trees and perform actions such as add, delete, and search on them.

1. **Requirements**
   1. **Assumptions**

* Only one integer per line on the insert file
* Only one single character action code and one integer per line on the action document.
  1. **Specifications**
* For all file operations
  + Check if the file(s) exist or if it is empty and print a message
* All values read in from the insert file should be added to a binary search tree, an AVL tree, and a Splay tree.
* After each value is added, the trees should be displayed to the screen to show that it was done properly
* Keep track of the number of inserts, deletions, and searches operations performed on each tree after the initial insert list is processed.
  + These metrics should be printed for the user after all processing has completed

1. **Decomposition Diagram**

Main

Print the metrics after the program finishes all operations

Print any errors/exceptions that occur during operations

Print an updated tree after each operation is performed

Complete the operations in the operation file for each tree

Record the operations metrics for each type of tree and operation

Insert integers in to each tree

Operations file containing the operation followed by an integer

Initial insert file containing only integers

Input

Output

Process

1. **Test Strategy**
   1. File Handling
   2. Valid Data
      1. Valid Test Plan Files
   3. Invalid Data
      1. Invalid Test Plan Files
2. **Test Plan Version 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| File Handling | 1.1 | Test initial insert file that exists and is not empty |  |  |  |  |
| File Handling | 1.2 | Test an operation file that exists and is not empty |  |  |  |  |
| File Handling | 1.3 | Test initial insert file that exists but is empty |  |  |  |  |
| File Handling | 1.4 | Test operation file that exists but is empty |  |  |  |  |
| File Handling | 1.5 | Test initial insert file that does not exist |  |  |  |  |
| File Handling | 1.6 | Test operation file that does not exist |  |  |  |  |
| Valid Data | 2.1 | Use an initial insert file that contains valid integers with no duplicates or negatives |  |  |  |  |
| Valid Data | 2.2 | Use an initial insert file that contains valid integers with no duplicates and it has negatives |  |  |  |  |
| Valid Data | 2.3 | Use an initial insert file that contains valid integers with duplicates and negative values |  |  |  |  |
| Valid Data | 2.4 | Use an operation file that contains only insert operations and valid integers |  |  |  |  |
| Valid Data | 2.5 | Use an operation file that contains only deletion operations and valid integers that exist in the current trees |  |  |  |  |
| Valid Data | 2.6 | Use an operation file that contains only search operations and integers that exist in the current trees |  |  |  |  |
| Valid Data | 2.7 | Use an operation file that contains only deletion operations and only some of the integers will be contained in the current trees |  |  |  |  |
| Valid Data | 2.8 | Use an operation file that contains only search operations and only some of the integers will be contained in the current trees |  |  |  |  |
| Valid Data | 2.9 | Use an operations file that contains a mix of insert and deletion operations and some of the deletion integers will not exist in the trees |  |  |  |  |
| Valid Data | 2.10 | Use an operations file that contains a mix of insert and search operations and some of the search operation integers will not exist in trees |  |  |  |  |
| Valid Data | 2.11 | Use an operations file that contains a mix of deletion and search operations and some of the operation integers will not exist in the trees |  |  |  |  |
| Valid Data | 2.12 | Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. No duplicate insertion integers |  |  |  |  |
| Valid Data | 2.13 | Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. Contains duplicate insertion integers. |  |  |  |  |
| Invalid Data | 3.1 | Use an operations file that only contains characters that do not correspond to valid operations |  |  |  |  |
| Invalid Data | 3.2 | Use an operations file that contains a mix valid and invalid operation characters |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Initial Algorithm**

* Function: log
  + Accepts a reference to a string value as input and the name of the requested output file
  + Opens the requested output file in append mode
  + Writes the input string to the file
  + Closes the file
* Class: Binary Search Tree
  + Private node pointing to the root of the tree
  + Structure for a tree node
    - Attributes are:
      * Value that points to the left and right nodes
      * Value that points to a twin node
      * Value containing the data for the node
  + Function: Insert
    - Accepts an integer value to insert in the tree
    - Returns the number of operations performed
    - Integer for number of operations
    - Create a new tree node and set the data value equal to the inputted value
    - Create a current node equal to the root node
    - While current is not null
      * If the value is equal to the current node value
        + While current nodes twin is not null

Set current equal to current twin node

Increase operations by 1

* + - * + Set current twin node equal to the new node
      * If the value is less than the current nodes value
        + Set current equal to left node of current
        + Increase operations by 1
      * If the value is greater than current nodes value
        + Set current equal to the right node of current
        + Increase operations by 1
    - Set current node equal to the new node
    - Return the value of operations
  + Function: Search
    - Accepts an integer value to search for as a parameter
    - Returns the number of operations performed
    - Boolean value for found
    - Create integer value for number of operations performed
    - Create current node equal to the root node
    - While current is not null
      * If value of current equals input value
        + Increase operations by 1
        + Set fount equal to true
        + Break out of while loop
      * If value of inputted integer is greater than value of current
        + Set current equal to current right
        + Increase operations by 1
      * If value of inputted integer is less than value of current
        + Set current equal to the current left
        + Increase operations by 1
    - If found is true return the value of operations
    - If found is false return the value of operations \* -1
  + Function: getMinNode (Private)
    - Accepts an input node as a parameter
    - Create a new node equal to the inputted node
    - While the current node left is not null
      * Set current node equal to current’s left node
    - Return current
  + Function: delete (public)
    - Accepts an integer value to delete
    - Returns the number of operations performed
      * Number will be negative if the value was not found
    - Integer value for operations
    - Set the root node equal to the return of deleteRecursive
      * Pass the root node, inputted integer value, and the pointer to operations as parameters
    - Return the value of operations
  + Function: deleteRecursive(private, recursive)
    - Accepts a pointer to the tree node, the integer to be deleted, and a pointer to the number of operation integer
    - If the node is null
      * Return the node
    - If the node is greater than the input integer
      * Set left value of node left equal to deleteRecursive
        + Pass the operations value, left node of current value and the deletion integer as parameters to deleteRecursive
    - If the node is less than the input integer
      * Set left value of node right equal to deleteRecursive
        + Pass the operations value, right node of current value and the deletion integer as parameters to deleteRecursive
    - If the nodes value equals the inputted value
      * If the node left is null
        + Create temp pointer equal to the nodes right value
        + Deallocate the nodes memory
        + Return the temp node
      * If the node right is null
        + Create temp pointer equal to the nodes left value
        + Deallocate the nodes memory
        + Return the temp node
      * If nodes left and right are not null
        + Create a temp node equal to the return of getMinNode

Pass the nodes right value as a parameter to getMinNode

* + - * + Set the current nodes value equal to the temp nodes value
        + Set the current right node value equal to the return of deleteRecursive

Pass the current nodes right, temp nodes value, and the pointer to operation count as parameters

* + - Return the node
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * New line followed by the integer value of the node
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
* Class: AVL Tree
  + Tree node
    - Attributes are:
      * Integer for the value
      * Nodes that point to the left and right
      * Integer for the height of the tree
  + Function: rightRotation
    - Accepts a tree node as input parameter
    - Create temp1 node equal to the input node’s left value
    - Create temp2 node equal to the temp1 node’s right value
    - Set temp1 right equal to input node
    - Set input node left equal to temp2
    - Set input node height equal to the maximum height of its right or left subtrees + 1
    - Set the temp 1 height equal to the maximum height between its left and right subtrees + 1
    - Return tremp1 node
  + Function: doubleRightRotation
    - Accepts a tree node as an input parameter
    - Returns a tree node
    - Set left node of input node equal to the return of the function leftRotation
      * Pass the left node of the input node as a parameter
    - Set the input node equal to the return of the function rightRotation
      * Pass the input node as a parameter
    - Return the input node
  + Function: doubleLeftRotation
    - Accepts a tree node as an input parameter
    - Returns a tree node
    - Set right node of input node equal to the return of the function right rotation
      * Pass the right node of the input node as a parameter
    - Set the input node equal to the return of the function leftRotation
      * Pass the input node as a parameter
    - Return the input node
  + Function: leftRotation
    - Accepts a tree node as an input parameter
    - Create temp1 node equal to the input node’s right value
    - Create temp2 node equal to the input temp1’s left value
    - Set temp1 left equal to input node
    - Set input node right equal to temp2
    - Set input node height to the maximum height between its left and right subtrees + 1
    - Set temp1 height to the maximum height between its left and right subtrees + 1
    - Return temp1 node
  + Function: insertRecursive (private, recursive)
    - Accepts the integer value to insert, the tree node, and the pointer to operation count integer as parameters
    - Returns a pointer to a node
    - If the input node is null
      * set it equal to a new node with the value to insert
    - if insert value is less than the node’s value
      * set nodes left value equal to the return of insertRecursive
        + pass the value to insert, the node, and the pointer to the operation count as parameter values
      * if the height of left – right equals 2
        + if insert value is less than value of left node

set the node equal to the return of rotateRight function

pass the node as a parameter

* + - * + else

set the node equal to the return of doubleRotateRight function

pass the node as a parameter

* + - if insert value is greater than node’s value
      * set the node’s right equal to the return of insert function
        + pass the value to insert, the node, and the pointer to the operation count as parameters
      * if the height of right – left equals 2
        + if insert value is greater than right node value

set the node equal to the return of rotateLeft function

pass the node as an input parameter

* + - * + else

set the node equal to the return of doubleRotateLeft function

pass the node as an input parameter

* + - set the node height equal to the max height between left and right + 1
    - return the node
  + Function: insert (public)
    - Accepts integer value to insert as a parameter
    - Create integer for number of operations
    - Set root node equal to the return of insertRecursive
      * Pass value to insert, root node, and the pointer to operations variable as input parameters
    - Return the value of operations
  + Function: deleteRecursive (private, recursive)
    - Accepts integer value to delete, tree node, and pointer to the operations count as input parameters
    - Returns a tree node
    - If the root equals null
      * Return the input node
    - If the integer input is less than input nodes value
      * Set the input nodes left equal to the return of deleteRecursive
        + Pass left node of the input node, the deletion value, and the pointer to the number of operations as parameters
    - If the integer input is greater than input nodes value
      * Set the input nodes right equal to the return of deleteRecursive
        + Pass right node of the input node, the deletion value, and the pointer to the number of operations as parameters
    - If the nodes value equals the deletion value
      * If the input node has only one child or no children
        + Create a temp node
        + Set the left value of the temp node equal to the single child of the input node
        + If there are no children

Set temp node equal to the input node

Set the input node to null

* + - * + If there is one child

Set the root is equal to the temp node

* + - * + Deallocate the temp node from memory
      * If input node has two children
        + Create a temp node equal to the return of minNode

Pass the input node as a parameter

* + - * + Set the input nodes value equal to the temp nodes value
        + Set the input node’s right equal to the return of deleteRecursive

Pass the input node’s right, temp nodes integer value, and pointer to the operations integer as parameters

* + - If the input node is null
      * Return the value of the input node
    - Set the input node height equal to the max height between the left and right child + 1
    - If the height input nodes left – right equals 2
      * If the height of input node’s left’s left – input node’s right’s right equals 1
        + Return the return node of leftRotation

Pass the input node as a parameter

* + - * Else
        + Return the return node of doubleLeftRotate

Pass the input node as a parameter

* + - If the height of the right – left equals 2
      * If the height of right’s right – left’s left equals 1
        + Return the return of rightRotation

Pass the input node as a parameter

* + - * Else
        + Return the return of doubleRightRotation

Pass the input node as a parameter

* + - Return the input node
  + Function: delete (public)
    - Accepts an integer of the value to delete
    - Create an integer for the operations count
    - Set the root equal to the return of deleteRecursive
      * Pass the deletion integer, root node and operations count as parameters
    - Return the operations count
  + Function: minNode (private, recursive)
    - Accepts a tree node as an input parameter
    - If the node is null
      * Return null
    - If the node’s left is null
      * Return the input node
    - Else
      * Return the return node from minNode
        + Pass the input node as a parameter
  + Function: Search
    - Accepts an integer value to search for as a parameter
    - Returns the number of operations performed
    - Boolean value for found
    - Create integer value for number of operations performed
    - Create current node equal to the root node
    - While current is not null
      * If value of current equals input value
        + Increase operations by 1
        + Set fount equal to true
        + Break out of while loop
      * If value of inputted integer is greater than value of current
        + Set current equal to current right
        + Increase operations by 1
      * If value of inputted integer is less than value of current
        + Set current equal to the current left
        + Increase operations by 1
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * New line followed by the integer value of the node
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
* Class: Splay Tree
  + Structure: tree node
    - Attributes:
      * Integer value
      * Pointer to the left, right, and parent nodes
  + Function: rotateRight
    - Accepts a tree node as input parameter
    - Create temp1 node equal to input node left
    - Create temp2 node equal to temp1 node right
    - Create parent1 node equal to input node parent
    - If parent1 is not null
      * If parent1 right node equals input node
        + Set parent1 node right equal to temp1
      * Else set parent node left equal to temp1
    - If temp2 is not null
      * Set temp2 parent equal to input node
    - Set temp1 parent equal to parent1
    - Set temp1 right equal to input node
    - Set input node parent equal to temp1
    - Set input node left equal to temp2
  + Function: rotateLeft
    - Accepts a tree node as input parameter
    - Create temp1 node equal to input node right
    - Create temp2 node equal to temp1 left
    - Create parent1 node equal to input node parent
    - If parent1 is not null
      * If parent1 right equals input node
        + Set parent1 right equal to temp1
      * Else
        + Set parent1 left equal to temp1
    - If temp2 is not null
      * Set temp2 parent equal to input node
    - Set temp1 parent equal to parent1
    - Set temp1 left equal to input node
    - Set input node parent equal to temp1
    - Set input node right equal to temp2
  + Function: zig
    - Accepts tree node as input parameter
    - Call rotationRight function
      * Pass the input node as a parameter
  + Function: zag
    - Accepts tree node as input parameter
    - Call rotationLeft function
      * Pass the input node as a parameter
  + Function: zigzag
    - Accepts tree node as input parameter
    - Call rotationRight on input node
    - Call rotationLeft on input node parent
  + Function: zigzig
    - Accepts tree node as input parameter
    - Call rotationRight on input node parent
    - Call rotationRight on input node
  + Function: zagzig
    - Accepts tree node as input parameter
    - Call rotationleft on input node
    - Call rotationRigth on input node parent
  + Function: zagzag
    - Accepts tree node as input parameter
    - Call rotationLeft on input node parent
    - Call rotationLeft on input node
  + Function: splay (public)
    - Accepts a tree node as input parameter
    - Create integer for operations count
    - While true
      * Create node for parent1 equal to parent of input node
      * Create node for parent2 equal to parent of parent1
      * If parent2 equals null
        + If parent1 left equals input node

Call zig function on parent1

* + - * + If parent1 right equals input node

Call zag function on parent1

* + - * + Increase operation count by 1
        + Break out of while loop
      * If parent2 left child equals parent1
        + If parent1 left child equals input node

Call zigzig function on parent1

* + - * + If parent1 right child equals input node

Call zagzig function on parent1

* + - * If parent2 right child equals parent1
        + If parent1 left child equals input node

Call zigzag function on parent1

* + - * + If parent1 right child equals input node

Call zagzag function on parent1

* + - Set the root value equal to input node
    - Return operations count
  + Function: insert (public)
    - Accepts an integer as an input parameter
    - Returns the operations count integer
    - Create integer for operations count
    - If root is null
      * Create a new node with the input value
      * Set root equal to the new node
    - Create a current node equal to the root node
    - While true
      * If current value equals input value
        + Create a twin chain node off of the current node
        + Break out of the loop
      * If input value is less than current value
        + If current left is null

Create a new node with the input value

Set current left equal to the new node

Break out of the loop

* + - * + Else

Increase operations count by 1

set current equal to current left

* + - * If input value is greater than current value
        + If current right is null

Create new node with the input value

Set current right equal to the new node

Break out of the loop

* + - * + Else

Increase operations count by 1

Set current equal to the current right

* + - Set operations equal to operations + the return of splay function on the current node
    - Return operations count value
  + Function: search (public)
    - Accepts integer to find as input parameter
    - Returns integer for the operations count
    - Create integer for the operations count
    - Create Boolean for fount, set equal to false
    - If root node is null
      * Return -1
    - Create current node equal to the root node
    - While current node is not null
      * If current value equals input value
        + Set found equal to true
        + Break out of loop
      * If current value is greater than input value
        + If current left is not null

Increase operations count by 1

Set current equal to current left

* + - * + Else break out of loop
      * If current value is less than input value
        + If current right value is not null

set current equal to current right

increase operations count by 1

* + - * + Else break out of the loop
    - set operations count equal to operations + return of splay function on the current node
    - if found is true
      * return the operations count
    - else
      * return operations count \* -1
  + Function: delete (public)
    - Accepts an integer as input parameter
    - Returns operations count preformed
    - Create integer for operation count
    - Create a new delete node
    - If root node is null
      * Set delete node equal to null
    - Create current node equal to the root node
    - While delete node is not null
      * If delete node value equals input value
        + Break out of loop
      * If delete node value is greater than input value
        + If delete node left is not null

Increase operations count by 1

Set delete node equal to current left

* + - * + Else break out of loop
      * If delete node value is less than input value
        + If delete node right value is not null

set delete node equal to current right

increase operations count by 1

* + - * + Else break out of the loop
    - If delete node is null
      * Return operation count \* -1
    - If delete node value does not equal input value
      * Return operations count \* -1
    - Set operation count equal to operation count + return of splay function on delete node
    - Create a temp1 node equal to delete node left
    - If temp1 is null
      * Set the root equal to delete node right
      * Set root parent equal to null
      * Deallocate delete node memory
      * Return operations count
    - While temp1 right is not null
      * Set temp1 equal to temp1 right
    - If delete node right is not null
      * Set temp1 right equal to delete node right
      * Set delete node right parent equal to temp1
    - Set the root equal to delete node left
    - Set root parent to null
    - Deallocate delete node memory
    - Return the operations count
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * New line followed by the integer value of the node
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
* Function: Main
  + Ask user to enter insert file
  + Attempt to open the insert file
    - If the file does not exist
      * Print that the file does not exist
      * Return from the function
    - If the file is empty
      * Print that the file is empty
      * Return from the function
  + Create object bst for binary search trees
  + Create object splay for splay trees
  + Create object avl for AVL trees
  + Create metrics 2- dimensional array of integers, 3x3
  + While the input file is not at the end
    - Create integer value equal to value on the current line in the file
    - Log the value to insert to each tree’s log file
    - Call bst insert function
      * Pass file integer value as parameter
    - Call bst print tree function
      * Log the result to the bst log file
    - Call the bst getHeight function
      * Log the result to the bst log file
    - Call avl insert function
      * Pass file integer value as parameter
    - Call the avl insert function
      * Log the result to the avl log file
    - Call the avl getHeight function
      * Log the result to the avl log file
    - Call splay insert function
      * Pass file integer value as parameter
    - Call the splay print tree function
      * Log the result to the splay log file
    - Call the splay getHeight function
      * Log the result to the splay log file
  + Ask the user to enter the name of the action file
  + Attempt to open the action file
    - If the file does not exist
      * Print that the file does not exist
      * Return from the function
    - If the file is empty
      * Print that the file is empty
      * Return from the function
  + While action file is not at the end
    - Get the action character and the integer value from the current line
    - If action character does not equal S, I, or D
      * Print that the action is invalid
      * Move to the next line in the file
    - If action character is S
      * Call bst search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the bst log file

Set metrics [1,0] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the bst log file

* + - * Call avl search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the avl log file

Set metrics [1,1] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the avl log file

* + - * Call splay search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the splay log file

Set metrics [1,2] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the splay log file

* + - If action character is I
      * Log the value to insert to each tree’s log file
      * Call bst insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,0] += to return of insert function
      * Call bst print tree function
        + Log the result to the bst log file
      * Call the bst getHeight function
        + Log the result to the bst log file
      * Call avl insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,1] += to return of insert function
      * Call the avl printTree function
        + Log the result to the avl log file
      * Call the avl getHeight function
        + Log the result to the avl log file
      * Call splay insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,2] += to return of insert function
      * Call the splay print tree function
        + Log the result to the splay log file
      * Call the splay getHeight function
        + Log the result to the splay log file
    - If action character is D
      * Call the bst delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the bst log file

Set metrics [2,0] += to return of delete function

Call bst printTree function and log the return of it

Call bst getHeight function and log the return of it

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the bst log file

* + - * Call the avl delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the avl log file

Set metrics [2,2] += to return of delete function

Call avl printTree function and log the return of it

Call avl getHeight function and log the return of it

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the avl log file

* + - * Call the splay delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the splay log file

Set metrics [2,2] += to return of delete function

Call splay printTree function and log the return of it

Call splay getHeight function and log the return of it

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the splay log file

* + Bst insert metrics to the screen (metrics [0,0])
  + Avl insert metrics to the screen (metrics [0,1])
  + Splay insert metrics to the screen (metrics[0,2])
  + Bst search metrics to the screen (metrics [1,0])
  + Avl search metrics to the screen (metrics [1,1])
  + Splay search metrics to the screen (metrics [1,2])
  + Bst delete metrics to the screen (metrics [2,0])
  + Avl delete metrics to the screen (metrics [2,1])
  + Splay delete metrics to the screen (metrics [2,2])

1. **Test Plan Version 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| File Handling | 1.1 | Test initial insert file that exists and is not empty | Initial\_2\_1.txt | Open file successfully |  |  |
| File Handling | 1.2 | Test an operation file that exists and is not empty | Operation\_2\_4.txt | Open file successfully |  |  |
| File Handling | 1.3 | Test initial insert file that exists but is empty | Initial\_empty.txt | Display that the file is empty and return from the function |  |  |
| File Handling | 1.4 | Test operation file that exists but is empty | Operation\_empty.txt | Display that the file is empty and return from the function |  |  |
| File Handling | 1.5 | Test initial insert file that does not exist | Somefile.txt | Display that the file does not exist and return form the function |  |  |
| File Handling | 1.6 | Test operation file that does not exist | Somefile.txt | Display that the file does not exist and return form the function |  |  |
| Valid Data | 2.1 | Use an initial insert file that contains valid integers with no duplicates or negatives | Initial\_1\_1.txt | Inserts the values into the three tree structures and outputs and logs the structures |  |  |
| Valid Data | 2.2 | Use an initial insert file that contains valid integers with no duplicates and it has negatives | Intial\_2\_2.txt | Inserts the values into the three tree structures and outputs and logs the structures |  |  |
| Valid Data | 2.3 | Use an initial insert file that contains valid integers with duplicates and negative values | Initial\_2\_3.txt | Inserts the values into the three tree structures and outputs and logs the structures |  |  |
| Valid Data | 2.4 | Use an operation file that contains only insert operations and valid integers | Initial\_2\_3.txt  operations\_2\_4.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.5 | Use an operation file that contains only deletion operations and valid integers that exist in the current trees | Initial\_2\_3.txt operations\_2\_5.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.6 | Use an operation file that contains only search operations and integers that exist in the current trees | Initial\_2\_3.txt operations\_2\_6.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.7 | Use an operation file that contains only deletion operations and only some of the integers will be contained in the current trees | Initial\_2\_3.txt operations\_2\_7.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.8 | Use an operation file that contains only search operations and only some of the integers will be contained in the current trees | Initial\_2\_3.txt Operations\_2\_8.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.9 | Use an operations file that contains a mix of insert and deletion operations and some of the deletion integers will not exist in the trees | Initial\_2\_3.txt Operations\_2\_9.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.10 | Use an operations file that contains a mix of insert and search operations and some of the search operation integers will not exist in trees | Initial\_2\_3.txt Operations\_2\_10.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.11 | Use an operations file that contains a mix of deletion and search operations and some of the operation integers will not exist in the trees | Initial\_2\_3.txt Operations\_2\_11.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.12 | Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. No duplicate insertion integers | Initial\_2\_3.txt  Operations\_2\_12.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Valid Data | 2.13 | Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. Contains duplicate insertion integers. | Initial\_2\_3.txt  Operations\_2\_12.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees |  |  |
| Invalid Data | 3.1 | Use an operations file that only contains characters that do not correspond to valid operations | Initial\_2\_3.txt  Operations\_3\_1.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees. Display the invalid operations and move to the next one in the file |  |  |
| Invalid Data | 3.2 | Use an operations file that contains a mix valid and invalid operation characters | Initial\_2\_3.txt  Operations\_3\_2.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees. Display the invalid operations and move to the next one in the file |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Code**

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#include "pch.h"

#include <iostream>

#include "BinarySearchTree.h"

#include "AvlTree.h"

#include "SplayTree.h"

#include <string>

#include <fstream>

using namespace std;

void log(const std::string &input, string fileName) {

ofstream filestream(fileName, std::ios\_base::app | std::ios\_base::out);

filestream.flush();

cout.flush();

std::cout << input << endl;

filestream << input << endl;

filestream.close();

}

/\*int main() {

AvlTree avl;

string tree;

int i, a, ia, ai, aa, ii, opCount;

cout << "Zig";

opCount = i = a = ia = ai = aa = ii = 0;

avl.insert(10, &opCount, &i, &a, &ia);

tree = avl.printTree();

cout << tree;

cout << "\n===========================" << endl;

opCount = i = a = ia = ai = aa = ii = 0;

avl.insert(5, &opCount, &i, &a, &ia);

tree = avl.printTree();

cout << tree;

cout << "\n===========================" << endl;

avl.insert(7, &opCount, &i, &a, &ia);

tree = avl.printTree();

cout << tree;

cout << "\n===========================" << endl;

system("pause");

}\*/

int main()

{

string bstLog = "bst\_3\_2.txt";

string avlLog = "avl\_3\_2.txt";

string splayLog = "splay\_3\_2.txt";

string metricLog = "metrics\_3\_2.txt";

cout << "Enter an initial input file: ";

string initFile;

getline(cin, initFile);

cout << endl;

ifstream initStream(initFile);

if (!initStream) {

cout << initFile + " Does not exist!" << endl;

system("pause");

return 0;

}

else if (initStream.peek() == ifstream::traits\_type::eof()) {

cout << initFile + " File is empty!" << endl;

system("pause");

return 0;

}

BinarySearchTree bst;

AvlTree avl;

SplayTree splay;

int metrics[3][3] = {{ 0,0,0 }, { 0,0,0 }, {0, 0, 0}};

string line;

while (getline(initStream, line)) {

int opCount;

string vis;

int rr = 0;

int ll = 0;

int lr = 0;

int rl = 0;

log("Inserting " + line, bstLog);

opCount = bst.insert(stoi(line));

vis = bst.printTree();

log(vis, bstLog);

log("Tree height is: " + to\_string(bst.maxHeight()), bstLog);

log("======================================================", bstLog);

log("Inserting " + line, avlLog);

opCount = avl.insert(stoi(line),&ll,&rr,&lr,&rl);

vis = avl.printTree();

log(vis, avlLog);

log("Tree height is: " + to\_string(avl.maxHeight()), avlLog);

log("right-right: " + to\_string(rr), avlLog);

log("right-left: " + to\_string(rl), avlLog);

log("left-right: " + to\_string(lr), avlLog);

log("left-left: " + to\_string(ll), avlLog);

log("======================================================", avlLog);

int i, a, ia, ai, aa, ii;

i = a = ia = ai = aa = ii = 0;

log("Inserting " + line, splayLog);

opCount = 0;

splay.insert(stoi(line), &opCount, &i, &a, &ia, &ai, &ii, &aa);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(bst.maxHeight()), splayLog);

log("======================================================", splayLog);

}

cout << "Enter an opertations file: ";

string opFile;

getline(cin, opFile);

cout << endl;

ifstream opStream(opFile);

if (!opStream) {

cout << opFile + " Does not exist!" << endl;

system("pause");

return 0;

}

else if (opStream.peek() == ifstream::traits\_type::eof()) {

cout << opFile + " File is empty!" << endl;

system("pause");

return 0;

}

while (getline(opStream, line)) {

char action = line.at(0);

int value = stoi(line.substr(2));

int opCount = 0;

string vis;

//cout << "[" << action << "]" << endl;

//cout << "[" << to\_string(value) << "]" << endl;

if (action == 'S') {

opCount = bst.search(value);

if (opCount > 0) {

log("Value found: " + to\_string(value), bstLog);

log("Operations: " + to\_string(opCount), bstLog);

log("======================================================", bstLog);

metrics[1][0] += opCount;

}

else {

log("Value not found: " + to\_string(value), bstLog);

log("Operations: " + to\_string(opCount \* -1), bstLog);

log("======================================================", bstLog);

metrics[1][0] += opCount \* -1;

}

opCount = avl.search(value);

if (opCount > 0) {

log("Value found: " + to\_string(value), avlLog);

log("Operations: " + to\_string(opCount), avlLog);

log("======================================================", avlLog);

metrics[1][1] += opCount;

}

else {

log("Value not found: " + to\_string(value), avlLog);

log("Operations: " + to\_string(opCount \* -1), avlLog);

log("======================================================", avlLog);

metrics[1][1] += opCount \* -1;

}

int i, a, ia, ai, aa, ii;

opCount = i = a = ia = ai = aa = ii = 0;

splay.search(value, &opCount, &i, &a, &ia, &ai, &ii, &aa);

if (opCount > 0) {

log("Found: " + to\_string(value), splayLog);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(splay.getMaxHeight()), splayLog);

log("zig: " + to\_string(i), splayLog);

log("zag: " + to\_string(a), splayLog);

log("zigzig: " + to\_string(ii), splayLog);

log("zigzag: " + to\_string(ia), splayLog);

log("zagzag: " + to\_string(aa), splayLog);

log("zagzig: " + to\_string(ai), splayLog);

log("======================================================", splayLog);

metrics[1][2] += opCount;

}

else {

log("Value not found: " + to\_string(value), splayLog);

log("Operations: " + to\_string(opCount \* -1), splayLog);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(splay.getMaxHeight()), splayLog);

log("======================================================", splayLog);

metrics[1][2] += opCount \* -1;

}

}

else if (action == 'I') {

log("Inserting " + to\_string(value), bstLog);

opCount = bst.insert(value);

vis = bst.printTree();

log(vis, bstLog);

log("Tree height is: " + to\_string(bst.maxHeight()), bstLog);

log("======================================================", avlLog);

metrics[0][0] += opCount;

int rr = 0;

int ll = 0;

int lr = 0;

int rl = 0;

log("Inserting " + to\_string(value), avlLog);

opCount = avl.insert(value, &ll, &rr, &lr, &rl);

vis = avl.printTree();

log(vis, avlLog);

log("Tree height is: " + to\_string(avl.maxHeight()), avlLog);

log("right-right: " + to\_string(rr), avlLog);

log("right-left: " + to\_string(rl), avlLog);

log("left-right: " + to\_string(lr), avlLog);

log("left-left: " + to\_string(ll), avlLog);

log("======================================================", avlLog);

metrics[0][1] += opCount;

int i, a, ia, ai, aa, ii;

i = a = ia = ai = aa = ii = 0;

log("Inserting " + to\_string(value), splayLog);

opCount = 0;

splay.insert(value, &opCount, &i, &a, &ia, &ai, &ii, &aa);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(splay.getMaxHeight()), splayLog);

log("zig: " + to\_string(i), splayLog);

log("zag: " + to\_string(a), splayLog);

log("zigzig: " + to\_string(ii), splayLog);

log("zigzag: " + to\_string(ia), splayLog);

log("zagzag: " + to\_string(aa), splayLog);

log("zagzig: " + to\_string(ai), splayLog);

log("======================================================", splayLog);

metrics[0][2] += opCount;

}

else if (action == 'D') {

opCount = bst.deleteValue(value);

if (opCount > 0) {

log("Deleted: " + to\_string(value), bstLog);

log("Operations: " + to\_string(opCount), bstLog);

vis = bst.printTree();

log(vis, bstLog);

log("Tree height is: " + to\_string(bst.maxHeight()), bstLog);

log("======================================================", bstLog);

metrics[2][0] += opCount;

}

else {

log("Could not delete: " + to\_string(value), bstLog);

log("Operations: " + to\_string(opCount \* -1), bstLog);

vis = bst.printTree();

log(vis, bstLog);

log("Tree height is: " + to\_string(bst.maxHeight()), bstLog);

log("======================================================", bstLog);

metrics[2][0] += opCount \* -1;

}

int rr = 0;

int ll = 0;

int lr = 0;

int rl = 0;

opCount = avl.deleteValue(value,&ll, &rr, &lr, &rl);

if (opCount > 0) {

log("Deleted: " + to\_string(value), avlLog);

log("Operations: " + to\_string(opCount), avlLog);

vis = avl.printTree();

log(vis, avlLog);

log("Tree height is: " + to\_string(avl.maxHeight()), avlLog);

log("right-right: " + to\_string(rr), avlLog);

log("right-left: " + to\_string(rl), avlLog);

log("left-right: " + to\_string(lr), avlLog);

log("left-left: " + to\_string(ll), avlLog);

log("======================================================", avlLog);

metrics[2][1] += opCount;

}

else {

log("Could not delete: " + to\_string(value), avlLog);

log("Operations: " + to\_string(opCount \* -1), avlLog);

vis = avl.printTree();

log(vis, avlLog);

log("Tree height is: " + to\_string(avl.maxHeight()), avlLog);

log("======================================================", avlLog);

metrics[2][1] += opCount \* -1;

}

int i, a, ia, ai, aa, ii;

opCount = i = a = ia = ai = aa = ii = 0;

splay.deleteNode(value, &opCount, &i, &a, &ia, &ai, &ii, &aa);

if (opCount > 0) {

log("Deleted: " + to\_string(value), splayLog);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(splay.getMaxHeight()), splayLog);

log("zig: " + to\_string(i), splayLog);

log("zag: " + to\_string(a), splayLog);

log("zigzig: " + to\_string(ii), splayLog);

log("zigzag: " + to\_string(ia), splayLog);

log("zagzag: " + to\_string(aa), splayLog);

log("zagzig: " + to\_string(ai), splayLog);

log("======================================================", splayLog);

metrics[2][2] += opCount;

}

else {

log("Could not delete: " + to\_string(value), splayLog);

log("Operations: " + to\_string(opCount \* -1), splayLog);

vis = splay.printTree();

log(vis, splayLog);

log("Tree height is: " + to\_string(splay.getMaxHeight()), splayLog);

log("======================================================", splayLog);

metrics[2][2] += opCount \* -1;

}

}

else {

cout << "[ERROR] Invalid input line: " << line << endl;

}

}

log("--------Inserts--------", metricLog);

log("BST: " + to\_string(metrics[0][0]), metricLog);

log("AVL: " + to\_string(metrics[0][1]), metricLog);

log("Splay: " + to\_string(metrics[0][2]), metricLog);

log("--------Searches--------", metricLog);

log("BST: " + to\_string(metrics[1][0]), metricLog);

log("AVL: " + to\_string(metrics[1][1]), metricLog);

log("Splay: " + to\_string(metrics[1][2]), metricLog);

log("--------Deletes--------", metricLog);

log("BST: " + to\_string(metrics[2][0]), metricLog);

log("AVL: " + to\_string(metrics[2][1]), metricLog);

log("Splay: " + to\_string(metrics[2][2]), metricLog);

system("pause");

}

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#pragma once

#include <string>

using namespace std;

class BinarySearchTree

{

private:

struct node {

int value;

node \*right;

node \*left;

node \*twin;

};

node\* root = NULL;

node\* getMinNode(node\* input);

node\* deleteRecursive(int x, node\* input, int \*count);

string printRecursive(node\* input, int spCount);

int getHeight(node\* input);

public:

BinarySearchTree();

~BinarySearchTree();

int insert(int x);

int search(int x);

int deleteValue(int x);

string printTree();

int maxHeight();

};

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#include "pch.h"

#include "BinarySearchTree.h"

#include <cstdlib>

BinarySearchTree::BinarySearchTree()

{

}

BinarySearchTree::~BinarySearchTree()

{

}

int BinarySearchTree::getHeight(node\* input)

{

if (input == NULL)

return 0;

else

{

// find depth of left and right nodes

int lDepth = getHeight(input->left);

int rDepth = getHeight(input->right);

if (lDepth > rDepth)

return(lDepth + 1);

else return(rDepth + 1);

}

}

int BinarySearchTree::insert(int x)

{

int opCount = 0;

node\* newNode = new node();

newNode->twin = NULL;

newNode->left = NULL;

newNode->right = NULL;

newNode->value = x;

if (root == NULL) {

root = newNode;

opCount = 1;

}

else {

node\* current = root;

while (current != NULL) {

if (x == current->value) {

// handle the twin node scenario

while (current->twin != NULL) {

current = current->twin;

opCount++;

}

current->twin = newNode;

return opCount;

}

// go to the left

else if (x < current->value) {

if (current->left == NULL) {

current->left = newNode;

opCount++;

return opCount;

}

else{

current = current->left;

opCount++;

}

}

// go to the right

else if (x > current->value) {

if (current->right == NULL) {

current->right = newNode;

opCount++;

return opCount;

}

else {

current = current->right;

opCount++;

}

}

}

current = newNode;

}

return opCount;

}

int BinarySearchTree::search(int x)

{

int opCount = 0;

bool found = false;

node\* current = root;

while (current != NULL) {

if (x == current->value) {

opCount++;

found = true;

break;

}

// go left

else if (x < current->value) {

current = current->left;

opCount++;

}

// go right

else if (x > current->value) {

current = current->right;

opCount++;

}

}

if (found) {

return opCount;

}

else {

//not found so return a -value

return opCount \* -1;

}

}

BinarySearchTree::node \* BinarySearchTree::getMinNode(node \* input)

{

node \*current = input;

while (current->left != NULL) {

current = current->left;

}

return current;

}

BinarySearchTree::node \* BinarySearchTree::deleteRecursive(int x, node \* input, int \* count)

{

if (input == NULL) {

return input;

}

else if (input->value > x) {

\*count+=1;

input->left = deleteRecursive(x, input->left, count);

}

else if (input->value < x) {

\*count+= 1;

input->right = deleteRecursive(x, input->right, count);

}

else if (input->value == x) {

if (input->twin != NULL) {

node\* temp = input;

while (temp->twin->twin != NULL) {

temp = temp->twin;

\*count += 1;

}

free(temp->twin);

temp->twin = NULL;

}

else {

if (input->left != NULL && input->right != NULL) { // neither the left or right side is empty

node\* temp = getMinNode(input->right);

input->value = temp->value;

input->right = deleteRecursive(temp->value, input->right, count);

count += 1;

}

else if (input->left == NULL) {

node\* temp = input->right;

free(input); // deallocate the memory for the node

return temp;

}

else if (input->right == NULL) {

node\* temp = input->left;

free(input);

return temp;

}

}

}

return input;

}

string BinarySearchTree::printRecursive(node\* input, int spCount)

{

string temp;

if (input != NULL) {

spCount += 5;

temp += printRecursive(input->right, spCount);

temp += "\n";

for (int i = 0; i < spCount; i++) {

temp += " ";

}

temp += to\_string(input->value);

if (input->twin != NULL) {

int tcount = 0;

node\* x = input;

while (x->twin != NULL) {

tcount += 1;

x = x->twin;

}

temp += "(" + to\_string(tcount);

temp += ")";

}

temp += printRecursive(input->left, spCount);

}

return temp;

}

int BinarySearchTree::deleteValue(int x)

{

int opCount = 0;

root = deleteRecursive(x, root, &opCount);

return opCount;

}

string BinarySearchTree::printTree()

{

string structure = printRecursive(root, 0);

return structure;

}

int BinarySearchTree::maxHeight()

{

return getHeight(root);

}

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#pragma once

#include <string>

using namespace std;

class AvlTree

{

private:

struct node {

int value;

node \*right = NULL;

node \*left = NULL;

node \*twin = NULL;

int height = 1;

};

node\* root = NULL;

node\* llRotation(node\* input);

node\* rrRotation(node\* input);

node\* lrRotation(node\* input);

node\* rlRotation(node\* input);

node\* insertRecursive(int x, node\* input, int \*opCount, int \* ll, int \* rr, int \*lr, int \*rl);

node\* deleteRecursive(int x, node\* input, int \* opCount, bool \* found, int \* ll, int \* rr, int \*lr, int \*rl);

node\* getMinNode(node\* input);

int getHeightDif(node\* input);

int getHeight(node\* input);

string printRecursive(node\* input, int spCount);

int getMaxHeight(node\* input);

public:

AvlTree();

~AvlTree();

string printTree();

int insert(int x, int \* ll, int \* rr, int \*lr, int \*rl);

int search(int x);

int deleteValue(int x, int \* ll, int \* rr, int \*lr, int \*rl);

int maxHeight();

};

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#include "pch.h"

#include "AvlTree.h"

#include <algorithm>

#include <cstdlib>

using namespace std;

AvlTree::AvlTree()

{

}

AvlTree::~AvlTree()

{

}

//Description: Public function that returns a string with a horizontally structured tree

string AvlTree::printTree()

{

string structure = printRecursive(root, 0);

return structure;

}

//Description: Public function that inserts an integer value into the AVL tree

int AvlTree::insert(int x, int \* ll, int \* rr, int \*lr, int \*rl)

{

int opCount = 0;

root = insertRecursive(x, root, &opCount, ll, rr, lr, rl);

return opCount;

}

//Description: public Function that searches through the AVL tree for the requested value.

// If the value can't be found then a negative operation cound will be returned

int AvlTree::search(int x)

{

int opCount = 0;

bool found = false;

node\* current = root;

while (current != NULL) {

if (x == current->value) {

opCount++;

found = true;

break;

}

else if (x < current->value) {

current = current->left;

opCount++;

}

else if (x > current->value) {

current = current->right;

opCount++;

}

}

if (found) {

return opCount;

}

else {

//not found so return a -value

return opCount \* -1;

}

return 0;

}

// Description: Public function that removes a value from the AVL tree.

int AvlTree::deleteValue(int x, int \* ll, int \* rr, int \*lr, int \*rl)

{

bool found = false;

int opCount = 0;

root = deleteRecursive(x, root, &opCount, &found, ll, rr, lr, rl);

if(found == true){

return opCount;

}

else {

return opCount \* -1;

}

}

//Description: Public function to get the maxHeight of the root node

int AvlTree::maxHeight()

{

return getMaxHeight(root);

}

//Description: Performs a left-left rotation on the inputted node

AvlTree::node \* AvlTree::llRotation(node \* input)

{

node\* temp1 = input->left;

node\* temp2 = temp1->right;

temp1->right = input;

input->left = temp2;

input->height = max(getHeight(input->left), getHeight(input->right)) + 1;

temp1->height = max(getHeight(input->left), getHeight(input->right)) + 1;

return temp1;

}

//Description: Performs a right-right rotation on the inputted node

AvlTree::node \* AvlTree::rrRotation(node \* input)

{

node\* temp1 = input->right;

node\* temp2 = temp1->left;

temp1->left = input;

input->right = temp2;

input->height = max(getHeight(input->left), getHeight(input->right)) + 1;

temp1->height = max(getHeight(input->left), getHeight(input->right)) + 1;

return temp1;

}

//Description: Performs a left-right rotation on the inputted node

AvlTree::node \* AvlTree::lrRotation(node \* input)

{

input->left = rrRotation(input->left);

input = llRotation(input);

return input;

}

//Description: Perfoms a right-left rotation on the inputted node

AvlTree::node \* AvlTree::rlRotation(node \* input)

{

input->right = llRotation(input->right);

input = rrRotation(input);

return input;

}

//Description: Recursively traverses throught the AVL tree to insert the inputted value in the correct position.

// After value is inserted, the proper rotations are done to ensure the tree is balanced properly

AvlTree::node \* AvlTree::insertRecursive(int x, node \* input, int \* opCount, int \* ll, int \* rr, int \*lr, int \*rl)

{

//node\* y = root;

if (input == NULL) {

node\* newNode = new node();

newNode->value = x;

return newNode;

}

else if (x < input->value) {

\*opCount+=1;

input->left = insertRecursive(x, input->left, opCount, ll, rr, lr, rl);

}

else if (x > input->value) {

\*opCount+= 1;

input->right = insertRecursive(x, input->right, opCount, ll, rr, lr, rl);

}

else if (x == input->value) {

node\* current = input;

node\* newNode = new node();

newNode->value = x;

// handle the twin node scenario

while (current->twin != NULL) {

current = current->twin;

\*opCount+=1;

}

current->twin = newNode;

return input;

}

input->height = max(getHeight(input->left), getHeight(input->right)) + 1;

int hDiff = getHeightDif(input);

if (hDiff > 1 && x < input->left->value) {

\*ll += 1;

return llRotation(input);

}

if (hDiff > 1 && x > input->left->value) {

\*lr += 1;

return lrRotation(input);

}

if (hDiff < -1 && x > input->right->value) {

\*rr += 1;

return rrRotation(input);

}

if (hDiff < -1 && x < input->right->value) {

\*rl += 1;

return rlRotation(input);

}

return input;

}

//Description: Recursively traverses throught the tree to delete the requested value. If the value isn't found, then the opCount will be negative.

AvlTree::node \* AvlTree::deleteRecursive(int x, node \* input, int \* opCount, bool \* found, int \* ll, int \* rr, int \*lr, int \*rl)

{

node\* y = root;

if (input == NULL) {

return input;

}

else if (x < input->value) {

\*opCount += 1;

input->left = deleteRecursive(x, input->left, opCount, found, ll, rr, lr, rl);

}

else if (x > input->value) {

\*opCount += 1;

input->right = deleteRecursive(x, input->right, opCount, found, ll, rr, lr, rl);

}

else if (x == input->value) {

\*found = true;

if (input->twin != NULL) {

node\* temp = input;

while (temp->twin->twin != NULL) {

temp = temp->twin;

\*opCount += 1;

}

free(temp->twin);

temp->twin = NULL;

}

else if (input->left == NULL || input->right == NULL) {

node\* temp = input->left ? input->left : input->right;

if (temp == NULL) {

temp = input;

input = NULL;

}

else {

input->value = temp->value;

input->height = temp->height;

input->left = temp->left;

input->right = temp->right;

input->twin = temp->twin;

}

free(temp);

temp = NULL;

}

else {

node \*temp = getMinNode(input->right);

input->value = temp->value;

input->right = deleteRecursive(temp->value, input->right, opCount, found, ll, rr, lr, rl);

}

}

if (input == NULL) {

return input;

}

input->height = max(getHeight(input->left), getHeight(input->right)) + 1;

int hDiff = getHeightDif(input);

if (hDiff > 1 && getHeightDif(input->left) >= 0) {

\*ll+=1;

return llRotation(input);

}

if (hDiff > 1 && getHeightDif(input->left) < 0) {

\*lr+=1;

return lrRotation(input);

}

if (hDiff < -1 && getHeightDif(input) <= 0) {

\*rr+=1;

return rrRotation(input);

}

if (hDiff < -1 && getHeightDif(input) > 0) {

\*rl+=1;

return rlRotation(input);

}

return input;

}

//Description: Finds the minimum valued node under the inputted node

AvlTree::node \* AvlTree::getMinNode(node \* input)

{

node \*current = input;

while (current->left != NULL) {

current = current->left;

}

return current;

}

//Description: Gets the difference in height of the right an left child nodes of the inputted node

int AvlTree::getHeightDif(node \* input)

{

if (input == NULL) {

return 0;

}

int left = getHeight(input->left);

int right = getHeight(input->right);

return left-right;

}

//Description: Gets the current height of the inputted node

int AvlTree::getHeight(node \* input)

{

if (input == NULL) {

return 0;

}

else {

return input->height;

}

}

//Description: Recursively builds a string that represents a horizantal structure of the tree

string AvlTree::printRecursive(node \* input, int spCount)

{

string temp;

if (input != NULL) {

spCount += 5;

temp += printRecursive(input->right, spCount);

temp += "\n";

for (int i = 0; i < spCount; i++) {

temp += " ";

}

temp += to\_string(input->value);

if (input->twin != NULL) {

int tcount = 0;

node\* x = input;

while (x->twin != NULL) {

tcount += 1;

x = x->twin;

}

temp += "(" + to\_string(tcount);

temp += ")";

}

temp += printRecursive(input->left, spCount);

}

return temp;

}

//Description: Gets the maximum height of the inputted node

int AvlTree::getMaxHeight(node \* input)

{

if (input == NULL)

return 0;

else

{

/\* compute the depth of each subtree \*/

int lDepth = getMaxHeight(input->left);

int rDepth = getMaxHeight(input->right);

/\* use the larger one \*/

if (lDepth > rDepth)

return(lDepth + 1);

else return(rDepth + 1);

}

}

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#pragma once

#include <string>

using namespace std;

class SplayTree

{

private:

struct node {

int value;

node\* left = NULL;

node\* right = NULL;

node\* twin = NULL;

node\* parent = NULL;

};

node\* root = NULL;

void rotateRight(node\* input);

void rotateLeft(node\* input);

/\*void zig(node\* input);

void zag(node\* input);

void zigzag(node\* input);

void zigzig(node\* input);

void zagzig(node\* input);

void zagzag(node\* input);\*/

string printRecursive(node\* input, int spCount);

int getHeight(node\* input);

public:

SplayTree();

~SplayTree();

void splay(node\* T, int\* opCount, int\* i, int\* a, int\* ia, int\* ai, int\* ii, int\* aa);

void insert(int x, int\* opCount, int\* i, int\* a, int\* ia, int\* ai, int\* ii, int\* aa);

void search(int x, int\* opCount, int\* i, int\* a, int\* ia, int\* ai, int\* ii, int\* aa);

void deleteNode(int x, int\* opCount, int\* i, int\* a, int\* ia, int\* ai, int\* ii, int\* aa);

string printTree();

int getMaxHeight();

};

//Program Name: Trees and more Trees

//Programmer Name: Jason Hogan

//Description: Reads integers from an inital insert file to bst, avl, and splay tree. then performs operations based on the requested action in the operations file

//Date Created: 2/23/18

#include "SplayTree.h"

#include <cstdlib>

#include "pch.h"

SplayTree::SplayTree()

{

}

SplayTree::~SplayTree()

{

}

void SplayTree::splay(node \* T, int \* opCount, int \* i, int \* a, int \* ia, int \* ai, int \* ii, int \* aa)

{

while (true)

{

node \*p = T->parent;

if (!p) break;

node \*pp = p->parent;

if (!pp)//Zig

{

if (p->left == T)

rotateRight(p);

else

rotateLeft(p);

break;

}

if (pp->left == p)

{

if (p->left == T)

{

// zig zig

\*ii += 1;

rotateRight(pp);

rotateRight(p);

}

else

{

// zag zig

\*ai += 1;

rotateLeft(p);

rotateRight(pp);

}

}

else

{

if (p->left == T)

{

// zig zag

\*ia += 1;

rotateRight(p);

rotateLeft(pp);

}

else

{

// zag zag

\*aa += 1;

rotateLeft(pp);

rotateLeft(p);

}

}

}

root = T;

}

void SplayTree::insert(int x, int \* opCount, int \* i, int \* a, int \* ia, int \* ai, int \* ii, int \* aa)

{

node\* newNode = new node;

newNode->value = x;

if (root == NULL) {

\*opCount += 1;

root = newNode;

return;

}

node\* current = root;

while (true) {

\*opCount += 1;

if (current->value == x) {

node\* temp = current;

// handle the twin node scenario

while (temp->twin != NULL) {

temp = temp->twin;

\*opCount += 1;

}

temp->twin = newNode;

break;

}

else if (x < current->value) {

if (current->left == NULL) {

current->left = newNode;

current->left->parent = current;

current = current->left;

break;

}

else {

current = current->left;

}

}

else if (x > current->value) {

if (current->right == NULL) {

current->right = newNode;

current->right->parent = current;

current = current->right;

break;

}

else {

current = current->right;

}

}

}

//bring the newly inserted node to the top

splay(current, opCount, i, a, ia, ai, ii, aa);

}

void SplayTree::search(int x, int \* opCount, int \* i, int \* a, int \* ia, int \* ai, int \* ii, int \* aa)

{

bool found = false;

if (root == NULL) {

\*opCount = 0;

return;

}

node\* current = root;

while (current != NULL && found == false) {

\*opCount += 1;

if (current->value == x) {

found = true;

break;

}

else if (current->value > x) {

current = current->left;

}

else if (current->value < x) {

current = current->right;

}

}

if (!found) {

\*opCount = \*opCount \* -1;

return;

}

splay(current, opCount, i, a, ia, ai, ii, aa);

}

void SplayTree::deleteNode(int x, int \* opCount, int \* i, int \* a, int \* ia, int \* ai, int \* ii, int \* aa)

{

bool found = false;

if (root == NULL) {

\*opCount = 0;

return;

}

node\* current = root;

while (current != NULL && found == false) {

\*opCount += 1;

if (current->value == x) {

found = true;

break;

}

else if (current->value > x) {

current = current->left;

}

else if (current->value < x) {

current = current->right;

}

}

if (!found) {

\*opCount = \*opCount \* -1;

return;

}

splay(current, opCount, i, a, ia, ai, ii, aa);

node\* temp1 = current->left;

if (temp1 == NULL) {

root = current->right;

root->parent = NULL;

free(temp1);

return;

}

while (temp1->right != NULL) {

temp1 = temp1->right;

}

if (current->right != NULL) {

temp1->right = current->right;

current->right->parent = temp1;

}

root = current->left;

root->parent = NULL;

free(current);

}

string SplayTree::printTree()

{

string structure = printRecursive(root, 0);

return structure;

}

int SplayTree::getMaxHeight()

{

return getHeight(root);

}

void SplayTree::rotateRight(node \* input)

{

node \*T = input->left;

node \*B = T->right;

node \*D = input->parent;

if (D)

{

if (D->right == input) D->right = T;

else D->left = T;

}

if (B)

B->parent = input;

T->parent = D;

T->right = input;

input->parent = T;

input->left = B;

}

void SplayTree::rotateLeft(node \* input)

{

node \*T = input->right;

node \*B = T->left;

node \*D = input->parent;

if (D)

{

if (D->right == input) D->right = T;

else D->left = T;

}

if (B)

B->parent = input;

T->parent = D;

T->left = input;

input->parent = T;

input->right = B;

}

/\*void SplayTree::zig(node \* input)

{

rotateRight(input);

}

void SplayTree::zag(node \* input)

{

rotateLeft(input);

}

void SplayTree::zigzag(node \* input)

{

rotateRight(input);

rotateLeft(input->parent);

}

void SplayTree::zigzig(node \* input)

{

rotateRight(input->parent);

rotateRight(input);

}

void SplayTree::zagzig(node \* input)

{

rotateLeft(input);

rotateRight(input->parent);

}

void SplayTree::zagzag(node \* input)

{

rotateLeft(input->parent);

rotateLeft(input);

}\*/

string SplayTree::printRecursive(node \* input, int spCount)

{

string temp;

if (input != NULL) {

spCount += 5;

temp += printRecursive(input->right, spCount);

temp += "\n";

for (int i = 0; i < spCount; i++) {

temp += " ";

}

temp += to\_string(input->value);

if (input->twin != NULL) {

int tcount = 0;

node\* x = input;

while (x->twin != NULL) {

tcount += 1;

x = x->twin;

}

temp += "(" + to\_string(tcount);

temp += ")";

}

temp += printRecursive(input->left, spCount);

}

return temp;

}

int SplayTree::getHeight(node \* input)

{

if (input == NULL)

return 0;

else

{

/\* compute the depth of each subtree \*/

int lDepth = getHeight(input->left);

int rDepth = getHeight(input->right);

/\* use the larger one \*/

if (lDepth > rDepth)

return(lDepth + 1);

else return(rDepth + 1);

}

}

1. **Updated Algorithm**

* Function: log
  + Accepts a reference to a string value as input and the name of the requested output file
  + Opens the requested output file in append mode
  + Writes the input string to the file
  + Closes the file
* Class: Binary Search Tree
  + Private node pointing to the root of the tree
  + Structure for a tree node
    - Attributes are:
      * Value that points to the left and right nodes
      * Value that points to a twin node
      * Value containing the data for the node
  + Function: Insert
    - Accepts an integer value to insert in the tree
    - Returns the number of operations performed
    - Integer for number of operations
    - Create a new tree node and set the data value equal to the inputted value
    - if root node is null
      * set root equal to the new node
      * increase operation count by 1
    - else
      * Create a current node equal to the root node
      * While current is not null
        + If the value is equal to the current node value

While current nodes twin is not null

Set current equal to current twin node

Increase operations by 1

Set current twin node equal to the new node

Return operations count

* + - * + If the value is less than the current nodes value

Set current equal to left node of current

Increase operations by 1

* + - * + If the value is greater than current nodes value

Set current equal to the right node of current

Increase operations by 1

* + - * Set current node equal to the new node
    - Return the value of operations
  + Function: Search
    - Accepts an integer value to search for as a parameter
    - Returns the number of operations performed
    - Boolean value for found
    - Create integer value for number of operations performed
    - Create current node equal to the root node
    - While current is not null
      * If value of current equals input value
        + Increase operations by 1
        + Set found equal to true
        + Break out of while loop
      * If value of inputted integer is greater than value of current
        + Set current equal to current right
        + Increase operations by 1
      * If value of inputted integer is less than value of current
        + Set current equal to the current left
        + Increase operations by 1
    - If found is true return the value of operations
    - If found is false return the value of operations \* -1
  + Function: getMinNode (Private)
    - Accepts an input node as a parameter
    - Create a current node equal to the inputted node
    - While the current node left is not null
      * Set current node equal to current’s left node
    - Return current
  + Function: delete (public)
    - Accepts an integer value to delete
    - Returns the number of operations performed
      * Number will be negative if the value was not found
    - Integer value for operations
    - Set the root node equal to the return of deleteRecursive
      * Pass the root node, inputted integer value, and the reference to operations as parameters
    - Return the value of operations
  + Function: deleteRecursive(private, recursive)
    - Accepts a pointer to the tree node, the integer to be deleted, and a pointer to the number of operation integer
    - If the node is null
      * Return the node
    - If the node is greater than the input integer
      * Set left ~~value~~ pointer of node left equal to deleteRecursive
        + Pass the operations value, left node of current value and the deletion integer as parameters to deleteRecursive
    - If the node is less than the input integer
      * Set right ~~value~~ pointer of node right equal to deleteRecursive
        + Pass the operations value, right node of current value and the deletion integer as parameters to deleteRecursive
    - If the nodes value equals the inputted value
      * If the node left is null
        + Create temp pointer equal to the nodes right value
        + Deallocate the nodes memory
        + Return the temp node
      * If the node right is null
        + Create temp pointer equal to the nodes left value
        + Deallocate the nodes memory
        + Return the temp node
      * If nodes left and right are not null
        + Create a temp node equal to the return of getMinNode

Pass the nodes right value as a parameter to getMinNode

* + - * + Set the current nodes value equal to the temp nodes value
        + Set the current right node value equal to the return of deleteRecursive

Pass the current nodes right, temp nodes value, and the pointer to operation count as parameters

* + - Return the node
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * ~~New line followed by~~ the integer value of the node
    - If the node has a twin node
      * Create integer for twin count
      * New node pointer equal to current node
      * While new node’s twin is not null
        + Increase twin count by 1
        + Set new node equal to it’s twin node
      * Add “(“ + twin count + “)” the string
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
  + Function: getHeight (private, recursive)
    - Accepts a node as input parameter
    - If the input is null
      * Return 0
    - Left depth value equal to return of get height
      * Pass left value of input as parameter
    - Right depth value equal to return of get height
      * Pass right value of input as parameter
    - If left > right
      * Return left depth + 1
    - Else return right depth + 1
  + Function: maxHeight (public)
    - Returns the value of getHeight
      * Pass the root value as a parameter
* Class: AVL Tree
  + Tree node
    - Attributes are:
      * Integer for the value
      * Nodes that point to the left and right
      * Integer for the height of the tree
  + Function: llRotation
    - Accepts a tree node as input parameter
    - Create temp1 node equal to the input node’s left value
    - Create temp2 node equal to the temp1 node’s right value
    - Set temp1 right equal to input node
    - Set input node left equal to temp2
    - Set input node height equal to the maximum height of its right or left subtrees + 1
    - Set the temp 1 height equal to the maximum height between its left and right subtrees + 1
    - Return tremp1 node
  + Function: lrRotation
    - Accepts a tree node as an input parameter
    - Returns a tree node
    - Set left node of input node equal to the return of the function rrRotation
      * Pass the left node of the input node as a parameter
    - Set the input node equal to the return of the function llRotation
      * Pass the input node as a parameter
    - Return the input node
  + Function: rlRotation
    - Accepts a tree node as an input parameter
    - Returns a tree node
    - Set right node of input node equal to the return of the function llRotation
      * Pass the right node of the input node as a parameter
    - Set the input node equal to the return of the function rrRotation
      * Pass the input node as a parameter
    - Return the input node
  + Function: rrRotation
    - Accepts a tree node as an input parameter
    - Create temp1 node equal to the input node’s right value
    - Create temp2 node equal to the input temp1’s left value
    - Set temp1 left equal to input node
    - Set input node right equal to temp2
    - Set input node height to the maximum height between its left and right subtrees + 1
    - Set temp1 height to the maximum height between its left and right subtrees + 1
    - Return temp1 node
  + Function: insertRecursive (private, recursive)
    - Accepts the integer value to insert, the tree node, and the pointer to operation count integer, and pointer to ll, rr, lr, rl count values as parameters
    - Returns a pointer to a node
    - If the input node is null
      * ~~set it equal to a new node with the value to insert~~
      * Return a new node with the value to insert
    - if insert value is less than the node’s value
      * set nodes left value equal to the return of insertRecursive
        + pass the value to insert, the node, and the pointer to the operation, ll, rr, lr, rl counts as parameter values
      * if the height of left – right equals 2
        + if insert value is less than value of left node

set the node equal to the return of llRotation function

pass the node as a parameter

* + - * + else

set the node equal to the return of lrRotation function

pass the node as a parameter

* + - if insert value is greater than node’s value
      * set the node’s right equal to the return of insert function
        + pass the value to insert, the node, and the pointer to the operation count as parameters
      * if the height of right – left equals 2
        + if insert value is greater than right node value

set the node equal to the return of rrRotation function

pass the node as an input parameter

* + - * + else

set the node equal to the return of rlRotation function

pass the node as an input parameter

* + - set the node height equal to the max height between left and right + 1
    - return the node
  + Function: insert (public)
    - Accepts integer value to insert and pointers to ll, rr, lr, rl count values as a parameter
    - Create integer for number of operations
    - Set root node equal to the return of insertRecursive
      * Pass value to insert, root node, the pointer to operations variable, and pointers to ll, rr, lr, rl count values as input parameters
    - Return the value of operations
  + Function: deleteRecursive (private, recursive)
    - Accepts integer value to delete, tree node, pointer to the operations count, and pointers to ll, rr, lr, rl count values as input parameters
    - Returns a tree node
    - If the integer input is less than input nodes value
      * Set the input nodes left equal to the return of deleteRecursive
        + Pass left node of the input node, the deletion value, and the pointer to the number of operations as parameters
    - If the integer input is greater than input nodes value
      * Set the input nodes right equal to the return of deleteRecursive
        + Pass right node of the input node, the deletion value, and the pointer to the number of operations as parameters
    - If the nodes value equals the deletion value
      * If the input node has only one child or no children
        + Create a temp node
        + Set the left value of the temp node equal to the single child of the input node
        + If there are no children

Set temp node equal to the input node

Set the input node to null

* + - * + If there is one child

Set the root is equal to the temp node

* + - * + Deallocate the temp node from memory
      * If input node has two children
        + Create a temp node equal to the return of minNode

Pass the input node as a parameter

* + - * + Set the input nodes value equal to the temp nodes value
        + Set the input node’s right equal to the return of deleteRecursive

Pass the input node’s right, temp nodes integer value, and pointer to the operations integer as parameters

* + - If the input node is null
      * Return the value of the input node
    - Set the input node height equal to the max height between the left and right child + 1
    - If the height input nodes left – right equals 2
      * If the height of input node’s left’s left – input node’s right’s right equals 1
        + Return the return node of leftRotation

Pass the input node as a parameter

* + - * Else
        + Return the return node of doubleLeftRotate

Pass the input node as a parameter

* + - If the height of the right – left equals 2
      * If the height of right’s right – left’s left equals 1
        + Return the return of rightRotation

Pass the input node as a parameter

* + - * Else
        + Return the return of doubleRightRotation

Pass the input node as a parameter

* + - Return the input node
  + Function: delete (public)
    - Accepts an integer of the value to delete
    - Create an integer for the operations count
    - Set the root equal to the return of deleteRecursive
      * Pass the deletion integer, root node and operations count as parameters
    - Return the operations count
  + Function: minNode (private, recursive)
    - Accepts a tree node as an input parameter
    - If the node is null
      * Return null
    - If the node’s left is null
      * Return the input node
    - Else
      * Return the return node from minNode
        + Pass the input node as a parameter
  + Function: Search
    - Accepts an integer value to search for as a parameter
    - Returns the number of operations performed
    - Boolean value for found
    - Create integer value for number of operations performed
    - Create current node equal to the root node
    - While current is not null
      * If value of current equals input value
        + Increase operations by 1
        + Set fount equal to true
        + Break out of while loop
      * If value of inputted integer is greater than value of current
        + Set current equal to current right
        + Increase operations by 1
      * If value of inputted integer is less than value of current
        + Set current equal to the current left
        + Increase operations by 1
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * ~~New line followed by~~ the integer value of the node
    - If the node has a twin node
      * Create integer for twin count
      * New node pointer equal to current node
      * While new node’s twin is not null
        + Increase twin count by 1
        + Set new node equal to it’s twin node
      * Add “(“ + twin count + “)” the string
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
* Class: Splay Tree
  + Structure: tree node
    - Attributes:
      * Integer value
      * Pointer to the left, right, and parent nodes
  + Function: rotateRight
    - Accepts a tree node as input parameter
    - Create temp1 node equal to input node left
    - Create temp2 node equal to temp1 node right
    - Create parent1 node equal to input node parent
    - If parent1 is not null
      * If parent1 right node equals input node
        + Set parent1 node right equal to temp1
      * Else set parent node left equal to temp1
    - If temp2 is not null
      * Set temp2 parent equal to input node
    - Set temp1 parent equal to parent1
    - Set temp1 right equal to input node
    - Set input node parent equal to temp1
    - Set input node left equal to temp2
  + Function: rotateLeft
    - Accepts a tree node as input parameter
    - Create temp1 node equal to input node right
    - Create temp2 node equal to temp1 left
    - Create parent1 node equal to input node parent
    - If parent1 is not null
      * If parent1 right equals input node
        + Set parent1 right equal to temp1
      * Else
        + Set parent1 left equal to temp1
    - If temp2 is not null
      * Set temp2 parent equal to input node
    - Set temp1 parent equal to parent1
    - Set temp1 left equal to input node
    - Set input node parent equal to temp1
    - Set input node right equal to temp2
  + ~~Function: zig~~
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationRight function~~
      * ~~Pass the input node as a parameter~~
  + ~~Function: zag~~
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationLeft function~~
      * ~~Pass the input node as a parameter~~
  + ~~Function: zigzag~~
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationRight on input node~~
    - ~~Call rotationLeft on input node parent~~
  + ~~Function: zigzig~~
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationRight on input node parent~~
    - ~~Call rotationRight on input node~~
  + ~~Function: zagzig~~
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationleft on input node~~
    - ~~Call rotationRigth on input node parent~~
  + ~~Function: zagzag~~ 
    - ~~Accepts tree node as input parameter~~
    - ~~Call rotationLeft on input node parent~~
    - ~~Call rotationLeft on input node~~
  + Function: splay (public)
    - Accepts a tree node as input parameter
    - Create integer for operations count
    - While true
      * Create node for parent1 equal to parent of input node
      * Create node for parent2 equal to parent of parent1
      * If parent2 equals null
        + If parent1 left equals input node

Zig

Call rotateRight function on parent1

* + - * + If parent1 right equals input node

Zag

Call rotateRight function on Parent1

* + - * + Increase operation count by 1
        + Break out of while loop
      * If parent2 left child equals parent1
        + If parent1 left child equals input node

ZigZig

Call rotateRight function on parent2

Call rotateRight function on parent1

* + - * + If parent1 right child equals input node

ZagZig

Call rotateLeft function on parent1

Call rotate right function on parent2

* + - * If parent2 right child equals parent1
        + If parent1 left child equals input node

ZigZag

Call rotateRight function on parent1

Call rotateLeft function on parent2

* + - * + If parent1 right child equals input node

ZagZag

Call rotateLeft function on parent2

Call rotateLeft function on parent1

* + - Set the root value equal to input node
    - Return operations count
  + Function: insert (public)
    - Accepts an integer as an input parameter
    - Returns the operations count integer
    - Create integer for operations count
    - If root is null
      * Create a new node with the input value
      * Set root equal to the new node
    - Create a current node equal to the root node
    - While true
      * If current value equals input value
        + Create a twin chain node off of the current node
        + Break out of the loop
      * If input value is less than current value
        + If current left is null

Create a new node with the input value

Set current left equal to the new node

Break out of the loop

* + - * + Else

Increase operations count by 1

set current equal to current left

* + - * If input value is greater than current value
        + If current right is null

Create new node with the input value

Set current right equal to the new node

Break out of the loop

* + - * + Else

Increase operations count by 1

Set current equal to the current right

* + - Set operations equal to operations + the return of splay function on the current node
    - Return operations count value
  + Function: search (public)
    - Accepts integer to find as input parameter
    - Returns integer for the operations count
    - Create integer for the operations count
    - Create Boolean for found, set equal to false
    - If root node is null
      * Return -1
    - Create current node equal to the root node
    - While current node is not null
      * If current value equals input value
        + Set found equal to true
        + Break out of loop
      * If current value is greater than input value
        + If current left is not null

Increase operations count by 1

Set current equal to current left

* + - * + Else break out of loop
      * If current value is less than input value
        + If current right value is not null

set current equal to current right

increase operations count by 1

* + - * + Else break out of the loop
    - set operations count equal to operations + return of splay function on the current node
    - if found is true
      * return the operations count
    - else
      * return operations count \* -1
  + Function: delete (public)
    - Accepts an integer as input parameter
    - Returns operations count preformed
    - Create integer for operation count
    - Create a new delete node
    - If root node is null
      * Set delete node equal to null
    - Create current node equal to the root node
    - While current node is not null
      * If current node value equals input value
        + Break out of loop
      * If current node value is greater than input value
        + If current node left is not null

Increase operations count by 1

Set current node equal to current left

* + - * + Else break out of loop
      * If current node value is less than input value
        + If delete node right value is not null

set current node equal to current right

increase operations count by 1

* + - * + Else break out of the loop
    - If delete node is null
      * Return operation count \* -1
    - If delete node value does not equal input value
      * Return operations count \* -1
    - Set operation count equal to operation count + return of splay function on delete node
    - Create a temp1 node equal to delete node left
    - If temp1 is null
      * Set the root equal to delete node right
      * Set root parent equal to null
      * Deallocate delete node memory
      * Return operations count
    - While temp1 right is not null
      * Set temp1 equal to temp1 right
    - If delete node right is not null
      * Set temp1 right equal to delete node right
      * Set delete node right parent equal to temp1
    - Set the root equal to delete node left
    - Set root parent to null
    - Deallocate delete node memory
    - Return the operations count
  + Function: printRecursive (private, recursive)
    - Accepts a tree node and the space count integer as input parameters
    - Returns a string with the structured tree
    - Create a string for the structure
    - If the node is null
      * Return empty string
    - Increase space count by 5
    - Add the return printRecursive to the end of the structure string
      * Pass the node right and space count as parameters
    - Add the following to the end of structure string
      * New line
      * Number of spaces defined in space count
      * ~~New line followed by~~ the integer value of the node
    - If the node has a twin node
      * Create integer for twin count
      * New node pointer equal to current node
      * While new node’s twin is not null
        + Increase twin count by 1
        + Set new node equal to it’s twin node
      * Add “(“ + twin count + “)” the string
    - Add the return of printRecursive to the end of the structure string
      * Pass the node left and space count as parameters
    - Return structure string
  + Function: printTree (public)
    - Returns a string containing the structure of the tree
    - Create a new string for the structure
    - Set the structure string equal to the return of the printRecursive function
      * Pass the root node and a space count of 0 as parameters
    - Return the value of the structure string
* Function: Main
  + Create string values for
    - Bst log
    - Avl log
    - Splay log
    - Metrics log
  + Ask user to enter insert file
  + Attempt to open the insert file
    - If the file does not exist
      * Print that the file does not exist
      * Return from the function
    - If the file is empty
      * Print that the file is empty
      * Return from the function
  + Create object bst for binary search trees
  + Create object splay for splay trees
  + Create object avl for AVL trees
  + Create metrics 2- dimensional array of integers, 3x3
  + While the input file is not at the end
    - Create integer value equal to value on the current line in the file
    - Log the value to insert to each tree’s log file
    - Call bst insert function
      * Pass file integer value as parameter
    - Call bst print tree function
      * Log the result to the bst log file
    - Call the bst getHeight function
      * Log the result to the bst log file
    - Call avl insert function
      * Pass file integer value as parameter
    - Call the avl insert function
      * Log the result to the avl log file
    - Call the avl getHeight function
      * Log the result to the avl log file
    - Call splay insert function
      * Pass file integer value as parameter
    - Call the splay print tree function
      * Log the result to the splay log file
    - Call the splay getHeight function
      * Log the result to the splay log file
  + Ask the user to enter the name of the action file
  + Attempt to open the action file
    - If the file does not exist
      * Print that the file does not exist
      * Return from the function
    - If the file is empty
      * Print that the file is empty
      * Return from the function
  + While action file is not at the end
    - Get the action character and the integer value from the current line
    - If action character does not equal S, I, or D
      * Print that the action is invalid
      * Move to the next line in the file
    - If action character is S
      * Call bst search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the bst log file

Set metrics [1,0] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the bst log file

* + - * Call avl search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the avl log file

Set metrics [1,1] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the avl log file

* + - * Call splay search function
        + If return is positive or 0

Log “Found value: [value]” and the return of search function to the splay log file

Set metrics [1,2] += to return of search function

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the splay log file

* + - Else If action character is I
      * Log the value to insert to each tree’s log file
      * Call bst insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,0] += to return of insert function
      * Call bst print tree function
        + Log the result to the bst log file
      * Call the bst getHeight function
        + Log the result to the bst log file
      * Call avl insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,1] += to return of insert function
      * Call the avl printTree function
        + Log the result to the avl log file
      * Call the avl getHeight function
        + Log the result to the avl log file
      * Call splay insert function on the file integer
        + Pass file integer value as parameter
        + Set metrics [0,2] += to return of insert function
      * Call the splay print tree function
        + Log the result to the splay log file
      * Call the splay getHeight function
        + Log the result to the splay log file
    - Else If action character is D
      * Call the bst delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the bst log file

Set metrics [2,0] += to return of delete function

Call bst printTree function and log the return of it

Call bst getHeight function and log the return of it

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the bst log file

* + - * Call the avl delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the avl log file

Set metrics [2,2] += to return of delete function

Call avl printTree function and log the return of it

Call avl getHeight function and log the return of it

* + - * + If return is negative

Log “Value not fount: [value]” and the return of search function \* -1 to the avl log file

* + - * Call the splay delete function on file integer
        + If return is positive or 0

Log “Deleted value: [value]” and the return of search function to the splay log file

Set metrics [2,2] += to return of delete function

Call splay printTree function and log the return of it

Call splay getHeight function and log the return of it

* + - * + If return is negative

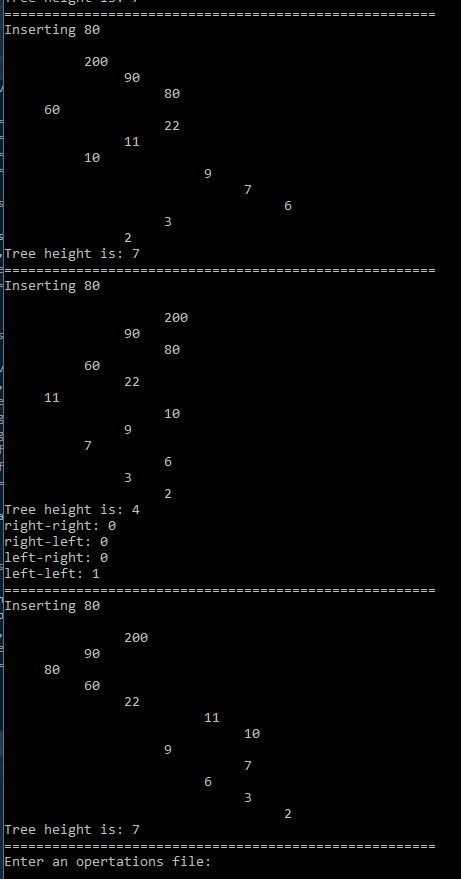
Log “Value not fount: [value]” and the return of search function \* -1 to the splay log file

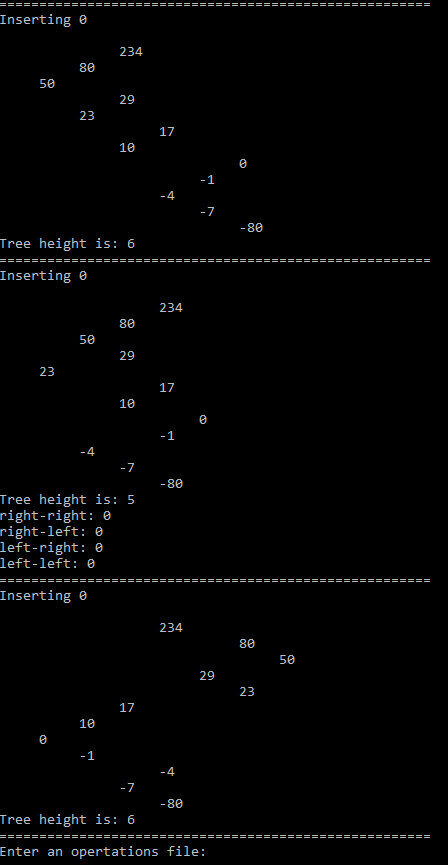
* + Bst insert metrics to the screen (metrics [0,0])
  + Avl insert metrics to the screen (metrics [0,1])
  + Splay insert metrics to the screen (metrics [0,2])
  + Bst search metrics to the screen (metrics [1,0])
  + Avl search metrics to the screen (metrics [1,1])
  + Splay search metrics to the screen (metrics [1,2])
  + Bst delete metrics to the screen (metrics [2,0])
  + Avl delete metrics to the screen (metrics [2,1])
  + Splay delete metrics to the screen (metrics [2,2])

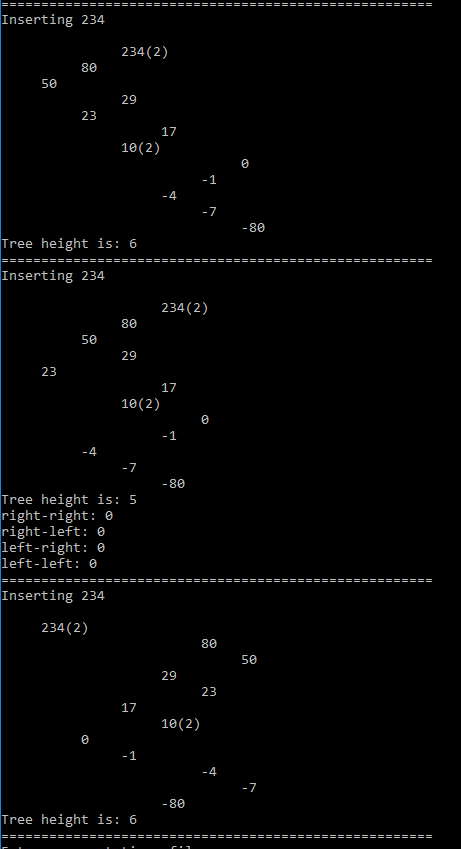
1. **Test Plan Version 3**

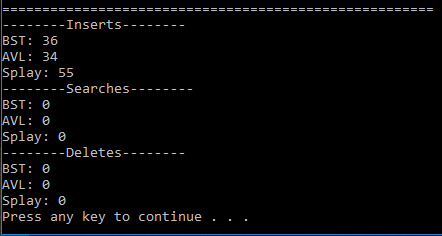
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| File Handling | 1.1 | Test initial insert file that exists and is not empty | Initial\_2\_1.txt | Open file successfully | Opened the file Bst\_2\_1.txt  Avl\_2\_1.txt  Splay\_2\_1.txt | Pass |
| File Handling | 1.2 | Test an operation file that exists and is not empty | Operation\_2\_4.txt | Open file successfully | Opened the file | Pass |
| File Handling | 1.3 | Test initial insert file that exists but is empty | Initial\_empty.txt | Display that the file is empty and return from the function | Displayed that the file was empty | Pass |
| File Handling | 1.4 | Test operation file that exists but is empty | Operation\_empty.txt | Display that the file is empty and return from the function | Displayed that the file was empty | Pass |
| File Handling | 1.5 | Test initial insert file that does not exist | Somefile.txt | Display that the file does not exist and return form the function | Printed to the screen that the file does not exist | Pass |
| File Handling | 1.6 | Test operation file that does not exist | Somefile.txt | Display that the file does not exist and return form the function | Displayed that the file did not exist | Pass |
| Valid Data | 2.1 | Use an initial insert file that contains valid integers with no duplicates or negatives | Initial\_2\_1.txt | Inserts the values into the three tree structures and outputs and logs the structures | Added all values into tree properly.  Bst\_2\_1.txt  Avl\_2\_1.txt  Splay\_2\_1.txt | Pass |
| Valid Data | 2.2 | Use an initial insert file that contains valid integers with no duplicates and it has negatives | Intial\_2\_2.txt | Inserts the values into the three tree structures and outputs and logs the structures | Added all values into tree properly.  Bst\_2\_2.txt  Avl\_2\_2.txt  Splay\_2\_2.txt | Pass |
| Valid Data | 2.3 | Use an initial insert file that contains valid integers with duplicates and negative values | Initial\_2\_3.txt | Inserts the values into the three tree structures and outputs and logs the structures | Added all values into tree properly.  Bst\_2\_3.txt  Avl\_2\_3.txt  Splay\_2\_3.txt | pass |
| Valid Data | 2.4 | Use an operation file that contains only insert operations and valid integers | Initial\_2\_3.txt  operations\_2\_4.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_4.txt  Avl\_2\_4.txt  Splay\_2\_4.txt  Metrics\_2\_4.txt | Pass |
| Valid Data | 2.5 | Use an operation file that contains only deletion operations and valid integers that exist in the current trees | Initial\_2\_3.txt operations\_2\_5.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_5.txt  Avl\_2\_5.txt  Splay\_2\_5.txt  Metrics\_2\_5.txt | Pass |
| Valid Data | 2.6 | Use an operation file that contains only search operations and integers that exist in the current trees | Initial\_2\_3.txt operations\_2\_6.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_6.txt  Avl\_2\_6.txt  Splay\_2\_6.txt  Metrics\_2\_6.txt | Pass |
| Valid Data | 2.7 | Use an operation file that contains only deletion operations and only some of the integers will be contained in the current trees | Initial\_2\_3.txt operations\_2\_7.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_7.txt  Avl\_2\_7.txt  Splay\_2\_7.txt  Metrics\_2\_7.txt | pass |
| Valid Data | 2.8 | Use an operation file that contains only search operations and only some of the integers will be contained in the current trees | Initial\_2\_3.txt Operations\_2\_8.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_8.txt  Avl\_2\_8.txt  Splay\_2\_8.txt  Metrics\_2\_8.txt | Pass |
| Valid Data | 2.9 | Use an operations file that contains a mix of insert and deletion operations and some of the deletion integers will not exist in the trees | Initial\_2\_3.txt Operations\_2\_9.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_9.txt  Avl\_2\_9.txt  Splay\_2\_9.txt  Metrics\_2\_9.txt | Pass |
| Valid Data | 2.10 | Use an operations file that contains a mix of insert and search operations and some of the search operation integers will not exist in trees | Initial\_2\_3.txt Operations\_2\_10.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_10.txt  Avl\_2\_10.txt  Splay\_2\_10.txt  Metrics\_2\_10.txt | Pass |
| Valid Data | 2.11 | Use an operations file that contains a mix of deletion and search operations and some of the operation integers will not exist in the trees | Initial\_2\_3.txt Operations\_2\_11.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_11.txt  Avl\_2\_11.txt  Splay\_2\_11.txt  Metrics\_2\_11.txt | Pass |
| Valid Data | 2.12 | Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. No duplicate insertion integers | Initial\_2\_3.txt  Operations\_2\_12.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_2\_12.txt  Avl\_2\_12.txt  Splay\_2\_12.txt  Metrics\_2\_12.txt | Pass |
| ~~Valid Data~~ | ~~2.13~~ | ~~Use an operations file that contains a mix of insert, deletion, and search operations. Some of the operations integers will not exist in the trees. Contains duplicate insertion integers.~~ | ~~Initial\_2\_3.txt~~  ~~Operations\_2\_12.txt~~ | ~~Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees~~ |  |  |
| Valid Data | 2.14 | Run the required initial insert file | required\_init.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_req.txt  Avl\_req.txt  Splay\_req.txt  Metrics\_req.txt | Pass |
| Valid Data | 2.15 | Run the required operation file | required\_op.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees | Added all values into tree properly.  Bst\_req.txt  Avl\_req.txt  Splay\_req.txt  Metrics\_req.txt | Pass |
| Valid Data | 2.16 | Test zig | Insert 10 then 15 | Performs zig op | Performed zig op. See screenshot for structure | pass |
| Valid Data | 2.17 | Test zag | Insert 10 then 5 | Performs zag op | Performed zag op. See screenshot for structure | pass |
| Valid Data | 2.18 | Test zagzig continue from 2.16 | Insert 7 | Performs zagzig op | Performed zagZig op. See screenshot for structure | pass |
| Valid Data | 2.19 | Test zigzag continue from 2.17 | Insert 7 | Performs zigzag op | Performed zigzag operation. See screenshot for structure | Pass |
| Valid Data | 2.20 | Test zagzag continue from 2.17 | Insert 3 | Performs zigzag op | Performed zigzag operation. See screenshot for structure | Pass |
| Valid Data | 2.21 | Test zigzig continue from 2.16 | Insert 20 | Performs zigzag op | Performed zigZig op. See screenshot for structure | Pass |
| Valid data | 2.22 | Test left-left rotation Avl tree | Insert 10, 5, and 2 | Performs left-left rotation | Performed left-left rotation. See screenshot for structure | Pass |
| Valid Data | 2.23 | Test right-right rotation avl tree | Insert 2, 5, 10 | Performs right-right Rotation | Performed right-right rotation. See screenshot for structure | Pass |
| Valid Data | 2.24 | Test right-left rotation avl tree | Insert 5, 10, 7 | Performs right-left rotation avl tree | Performed right-left rotation. See screenshot for structure | Pass |
| Valid data | 2.25 | Test left-right rotation avl tree | Insert 10, 5, 7 | Performs left-right rotation avl tree | Performed left-right rotation. See screenshot for structure | pass |
| Invalid Data | 3.1 | Test only invalid operations | Initial\_2\_3.txt  Operations\_3\_1.txt | Inserts all values into the three tree structures and outputs and logs all error messages caught in the operations file | Added all values into tree properly and caught all errors.  Bst\_3\_1.txt  Avl\_3\_1.txt  Splay\_3\_1.txt  Metrics\_3\_1.txt | Pass |
| Invalid Data | 3.2 | Use an operations file that contains a mix valid and invalid operation characters | Initial\_2\_3.txt  Operations\_3\_2.txt | Inserts the values into the three tree structures and outputs and logs the structures. Performs the correct operations on the trees. Display the invalid operations and move to the next one in the file | Added all values into tree properly and caught all errors.  Bst\_3\_2.txt  Avl\_3\_2.txt  Splay\_3\_2.txt  Metrics\_3\_2.txt | pass |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

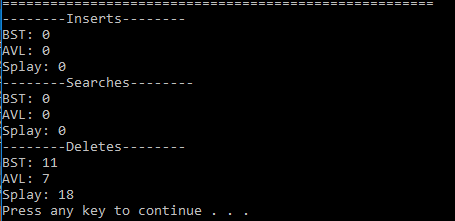
1. **Screenshots**

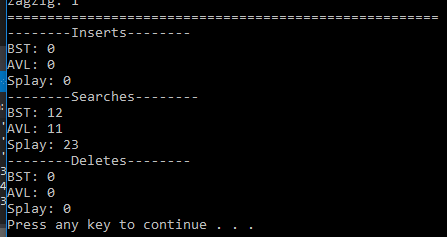
 Test case 2.1 & 1.1

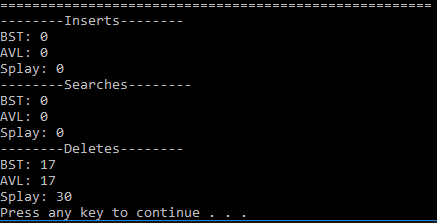
Test case 2.2

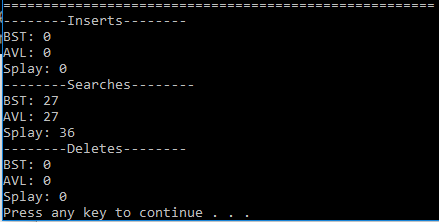
 Test case 2.3 & initial 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12

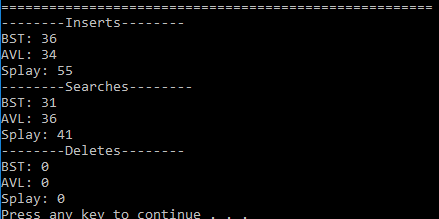
 Test 2.4

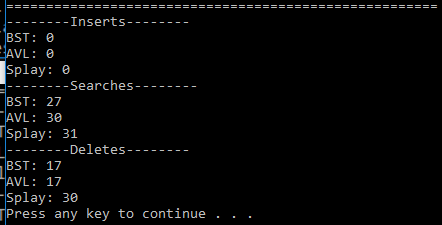
 Test 2.5

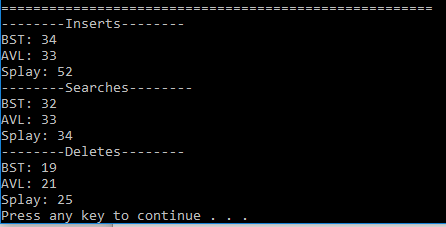
 Test 2.6

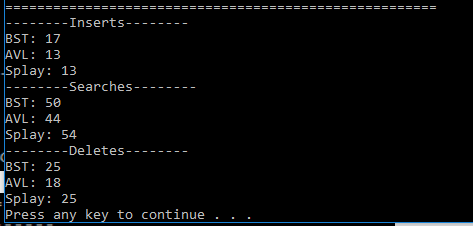
 Test 2.7

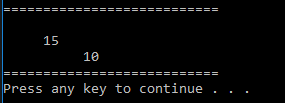
 Test 2.8

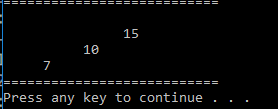
 Test 2.10

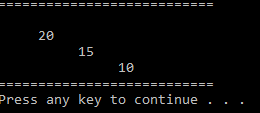
 Test 2.11

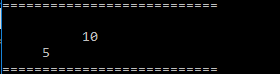
 Test 2.12

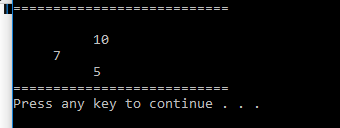
 Test 2.14 and 2.15

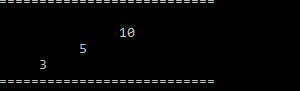
 Test 2.16

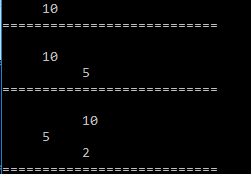
 Test 2.18

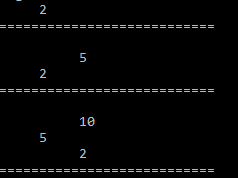
 Test 2.21

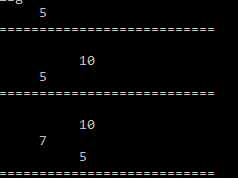
 Test 2.17

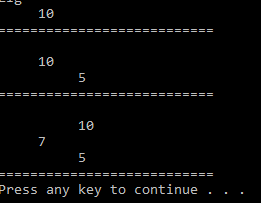
 Test 2.19

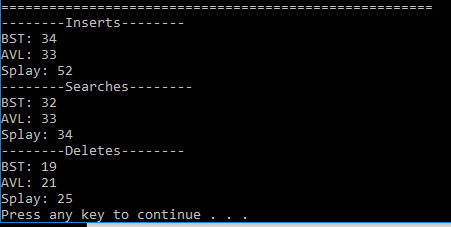
 Test 2.20

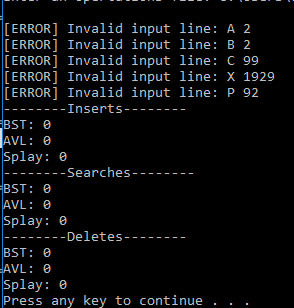
 Test 2.22

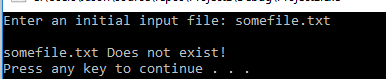
 Test 2.23

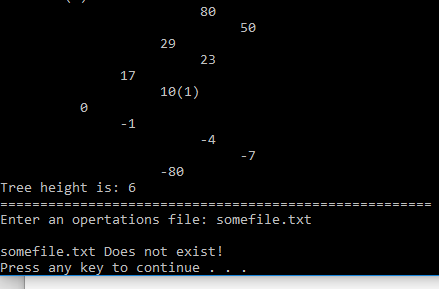
 Test 2.24

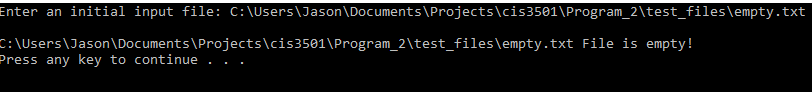
 Test 2.25

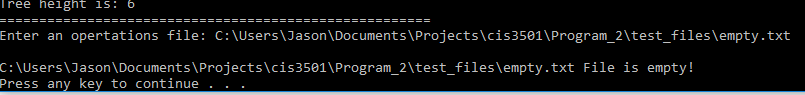
 Test 3.2

 Test 3.1

 Test 1.5

 Test 1.6

 Test 1.3

 Test 1.4

1. **Error Log**

|  |  |  |
| --- | --- | --- |
| Error Type | Cause of Error | Solution to Error |
| Logical (probably) | Not completely sure. The zig, zag, zigzag, etc. functions that I attempted to implement in the splay tree class kept causing pointer issues with the tree structure. | Eliminated the separate functions and implemented the logic directly in the splay function. |
|  |  |  |

1. **Status**

The program works in its current state and outputs all actions neatly to a log file for each type of tree.