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

1. **Test Plan Version 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Code**

A baseline for commenting is before any function add this:

//Description: What does the function do

//Pre-condition: What do input do you need for the function to work

//Post-condition: What is the end result of the function or what do you get out of the function

Also the beginning of your program should have these comments:

//Program Name:

//Programmer Name:

//Description:

//Date Created:

1. **Updated Algorithm**
2. **Test Plan Version 3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Screenshots**
2. **Error Log**

|  |  |  |
| --- | --- | --- |
| Error Type | Cause of Error | Solution to Error |
|  |  |  |
|  |  |  |

1. **Status**