CS 3310: Data and File Structures

Instructor: Dr. Ajay K. Gupta, Western Michigan University

Lab TA: Yu Guo

Matt Peter

**Software Life Cycle Report – Assignment 4**

**Phase 1: Specification**

Objectives:

1. Gain experience in manipulating binary search trees and using heaps
2. Analyze the advantages and disadvantages of binary search trees

Write a program that:

1. Generates 100 random integers and inserts them into a binary search tree
2. Deletes nodes of the tree that have maximum and minimum values for certain attributes
3. Computes the sum of the values for certain attributes and inserts a node that makes these values equal
4. Finds the tallest and shortest binary search trees out of a set of 20
5. Traverses a binary search tree in preorder, postorder, and inorder format

**Phase 2: Design**

2.1 Modules and their Basic Structure

1. Module 1: Class Hw4Main contains:
   1. public static void main (String[] args) – Calls methods from other classes and calculates and prints results from returned values
2. Module 2: Class BST contains:
   1. private class Node – Defines nodes used to store data
   2. public void insert (int keyValue) – Inserts a passed value into the binary search tree
   3. public void delete (int keyValue) – Deletes the node with the passed key value
   4. public int getHeight () – Returns the height of the binary search tree
   5. public int getSumKeys () – Prints and returns the sum of all the keys in the binary search tree
   6. private int findSumKeys (Node node) – Calculates and returns the sum of all the keys in the binary search tree
   7. public int getSumCounts () – Prints and returns the sum of all the counts in the binary search tree
   8. private int findSumCounts (Node node) – Calculates and returns the sum of all the counts in the binary search tree
   9. public void findKeys () – Prints the values of the kth smallest keys
   10. private int findKthSmallest (Node node, int k) – Finds and returns the value of the kth smallest key in the binary search tree
   11. public void printOrders () – Calls recursive methods to print the binary search tree in preorder, postorder, and inorder formats
   12. private void printInorder (Node node) – Prints the binary search tree in inorder format using recursion
   13. private void printPreorder (Node node) – Prints the binary search tree in preorder format using recursion
   14. private void printPostorder (Node node) – Prints the binary search tree in postorder format using recursion
   15. public void updateTree () – Calls recursive methods to update the values of the sizeOfSubtree and subtreeHeight attributes for all nodes in the binary search tree
   16. private int updateSize (Node node) – Updates the value of sizeOfSubtree for all nodes using recursion
   17. private int updateHeight (Node node) – Updates the value of subtreeHeight for all nodes using recursion
3. Module 3: Class Heap contains:
   1. private class Node – Defines nodes used to store data
   2. public void insertMaxHeap (int keyValue) – Inserts a passed value into the max-heap and adjusts the node’s position if necessary
   3. public void insertMinHeap (int keyValue) – Inserts a passed value into the min-heap and adjusts the node’s position if necessary
   4. public int getRootKeyValue () – Returns the key value of the root node
   5. public int getRootCounterValue () – Returns the count value of the root node
   6. private int findKey (Node node, int key, int heapPos) – Finds the position of a node with the specified key value in the heap (Returns -1 if the node doesn’t exist)

2.2 Pseudocode for the Modules

2.2.1 Pseudocode for Hw4Main

1a. Hw4Main Pseudocode Refinement #1:

// Method main – Calls methods from other classes and calculates and prints results from returned values

2a. Hw4Main Pseudocode Refinement #2:

// Method main – Calls methods from other classes and calculates and prints results from returned values

// Declare and Instantiate Random object and height trackers

// Enter a for loop that will run 20 tests

// Declare and Instantiate BST and Heap objects

// Generate 100 values and insert into BST and Heaps

// Print largest and smallest values and delete nodes

// Find sum of keys and counts, compute equalizing value, print results, and insert node into BST

// Find and print 5th, 20th, 30th, and 70th smallest keys

// Check height to see if min or max gets changed

// Check if this is the last test

// If so, print the max and min heights and the values of this last BST in preorder, postorder, and inorder formats

2.2.2 Pseudocode for BST

2a. BST Pseudocode Refinement #1:

// Class Node – Defines nodes used to store data

// Method insert – Inserts a passed value into the binary search tree

// Method delete – Deletes the node with the passed key value

// Method getHeight – Returns the height of the binary search tree

// Method getSumKeys – Prints and returns the sum of all the keys in the binary search tree

// Method findSumKeys – Calculates and returns the sum of all the keys in the binary search tree

// Method getSumCounts – Prints and returns the sum of all the counts in the binary search tree

// Method findSumCounts – Calculates and returns the sum of all the counts in the binary search tree

// Method findKeys – Prints the values of the kth smallest keys

// Method findKthSmallest – Finds and returns the value of the kth smallest key in the binary search tree

// Method printOrders – Calls recursive methods to print the binary search tree in preorder, postorder, and inorder formats

// Method printInorder – Prints the binary search tree in inorder format using recursion

// Method printPreorder – Prints the binary search tree in preorder format using recursion

// Method printPostorder – Prints the binary search tree in postorder format using recursion

// Method updateTree – Calls recursive methods to update the values of the sizeOfSubtree and subtreeHeight attributes for all nodes in the binary search tree

// Method updateSize – Updates the value of sizeOfSubtree for all nodes using recursion

// Method updateHeight – Updates the value of subtreeHeight for all nodes using recursion

2b. BST Pseudocode Refinement #2:

// Class Node – Defines nodes used to store data

// Declare all attributes (Nodes and integers)

// Have constructor that sets Nodes to null and integers to “empty” values

// Have constructor that has integer as argument that is assigned to key attribute, sets Nodes to null, and sets other integers to single Node values

// Method insert – Inserts a passed value into the binary search tree

// If a Node already exists…

// Find where this key value should exist

// If it already exists, increment that Node’s count value

// If not, create a new Node and assign the key value to it

// If there are no Nodes yet (this is the first value)

// Assign this key value to the root Node

// Method delete – Deletes the node with the passed key value

// Find where the passed key value should exist

// If it does exist…

// Enter a loop that will find the highest key value that is less than or lowest key value that is greater than the key value that was deleted

// Continue this loop until a leaf Node is reached

// “Delete” this resulting empty leaf Node

// Method getHeight – Returns the height of the binary search tree

// Return the value of subtreeHeight for the root Node

// Method getSumKeys – Prints and returns the sum of all the keys in the binary search tree

// Call the findSumKeys method

// Print the returned result

// Return the result

// Method findSumKeys – Calculates and returns the sum of all the keys in the binary search tree

// Recursively go through all the Nodes, adding all their key values to a single value

// Method getSumCounts – Prints and returns the sum of all the counts in the binary search tree

// Call the findSumCounts method

// Print the returned result

// Return the result

// Method findSumCounts – Calculates and returns the sum of all the counts in the binary search tree

// Recursively go through all the Nodes, adding all their count values to a single value

// Method findKeys – Prints the values of the kth smallest keys

// Call the findKthSmallest method 4 times, passing 5, 20, 30, and 70 as arguments

// Method findKthSmallest – Finds and returns the value of the kth smallest key in the binary search tree

// Recursively go through all the Nodes in order

// Add up all the Node key values as they’re gone through

// Once the sum is equal to or greater than the argument k, the last key value whose count value was added to the sum is returned

// Method printOrders – Calls recursive methods to print the binary search tree in preorder, postorder, and inorder formats

// Call printInorder, printPreorder, and printPostorder methods

// Method printInorder – Prints the binary search tree in inorder format using recursion

// Recursively go through the binary search tree in inorder format, printing the key, count, sizeOfSubtree, and subtreeHeight values of each Node

// Method printPreorder – Prints the binary search tree in preorder format using recursion

// Recursively go through the binary search tree in preorder format, printing the key, count, sizeOfSubtree, and subtreeHeight values of each Node

// Method printPostorder – Prints the binary search tree in postorder format using recursion

// Recursively go through the binary search tree in postorder format, printing the key, count, sizeOfSubtree, and subtreeHeight values of each Node

// Method updateTree – Calls recursive methods to update the values of the sizeOfSubtree and subtreeHeight attributes for all nodes in the binary search tree

// Call updateSize and updateHeight methods

// Method updateSize – Updates the value of sizeOfSubtree for all nodes using recursion

// Recursively go through the binary search tree

// If a Node is a leaf, set its sizeOfSubtree value to 1

// Otherwise, set the Node’s sizeOfSubtree value to the sum of its children’s sizeOfSubtree values plus 1

// Method updateHeight – Updates the value of subtreeHeight for all nodes using recursion

// Recursively go through the binary search tree

// If a Node is a leaf, set its subtreeHeight to 0

// Otherwise, set the Node’s subtreeHeight value to the greater of its children’s subtreeHeight values plus 1

2.2.3 Pseudocode for Heap

3a. Heap Pseudocode Refinement #1:

// Class Node – Defines nodes used to store data

// Method insertMaxHeap – Inserts a passed value into the max-heap and adjusts the node’s position if necessary

// Method insertMinHeap – Inserts a passed value into the min-heap and adjusts the node’s position if necessary

// Method getRootKeyValue – Returns the key value of the root node

// Method getRootCounterValue – Returns the count value of the root node

// Method findKey – Finds the position of a node with the specified key value in the heap (Returns -1 if the node doesn’t exist)

3b. Heap Pseudocode Refinement #2:

// Class Node – Defines nodes used to store data

// Declare all attributes (Nodes and integers)

// Have constructor that sets Nodes to null and integers to “empty” values

// Have constructor that has integer as argument that is assigned to key attribute, sets Nodes to null, and sets count to 1

// Method insertMaxHeap – Inserts a passed value into the max-heap and adjusts the node’s position if necessary

// If there’s at least one value in the heap…

// Call the findKey method to see if the key exists

// If the key exists in the heap…

// Calculate the steps it will take to get to the key’s Node in the heap

// Get to the key’s Node

// Increment its count value

// Adjust the Node’s position if necessary

// If the key doesn’t exist in the heap…

// Calculate the steps it will take to get to the Node where the key should go in the heap

// Get to the parent of where the key’s Node should go

// Insert the Node

// If there aren’t any values in the heap…

// Create a root Node

// Method insertMinHeap – Inserts a passed value into the min-heap and adjusts the node’s position if necessary

// If there’s at least one value in the heap…

// Call the findKey method to see if the key exists

// If the key exists in the heap…

// Calculate the steps it will take to get to the key’s Node in the heap

// Get to the key’s Node

// Increment its count value

// Adjust the Node’s position if necessary

// If the key doesn’t exist in the heap…

// Calculate the steps it will take to get to the Node where the key should go in the heap

// Get to the parent of where the key’s Node should go

// Insert the Node

// Adjust the Node’s position if necessary

// If there aren’t any values in the heap…

// Create a root Node

// Method getRootKeyValue – Returns the key value of the root node

// Return the root Node’s key value

// Method getRootCounterValue – Returns the count value of the root node

// Return the root Node’s count value

// Method findKey – Finds the position of a node with the specified key value in the heap (Returns -1 if the node doesn’t exist)

// Recursively go through the heap

// Return the position in the heap if the key is found

// Otherwise, return -1

**Phase 3: Risk Analysis**

There are no known risks associated with this application.

**Phase 4: Verification**

All the steps of the algorithm were checked to ensure correct results in all circumstances. A variety of tests with a wide range of values were also done to check for correct output.

**Phase 5: Coding**

Program code can be found in the .java files and contains in-line comments as well as a Javadoc.

**Phase 6: Testing**

All program results are printed to the IDE console.

**Phase 7: Refining the Program**

All required features are included in the program so no refinements are needed.

**Phase 8: Production**

A zip file containing source files, a Javadoc, and test data have been submitted.

**Phase 9: Maintenance**

Any maintenance can be performed once feedback has been obtained.