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 3**

**Phase 1: Specification**

Objectives:

1. Write a program that uses multiple techniques to implement heaps
2. Analyze the advantages and disadvantages of explicit and implicit representations of trees
3. Compare the results you get in terms of theoretical and empirical complexities

Write a program that:

1. Performs n random operations of insert, deletemin, and findmin on an SBT in its explicit representation and an array-based CBT, both in min-heap form
2. Have each node in both trees contain a key and a value
3. Print both trees when their heap-sizes reach 30 for the first time, with the CBT being printed in the order values are stored in the array, and the SBT being printed using both preorder and inorder traversals.

**Phase 2: Design**

2.1 Modules and their Basic Structure

1. Module 1: Class Hw3Main contains:
   1. public static void main (String[] args) – Gets information from the user and randomly calls methods to perform operations
   2. private static void printCBT (CBT mainCBT) – Shows effects of each operation on CBT with 30 values
   3. private static void printPreOrderSBT (SBT mainSBT) – Shows effects of each operation on SBT with 30 values (printed in preorder format)
   4. private static void printInOrderSBT (SBT mainSBT) – Shows effects of each operation on SBT with 30 values (printed in inorder format)
   5. private static void printCBTArray (int[][] arr, int size) – Prints values in passed array
   6. private static void printSBTArray (int[][] arr, int size) – Prints values in passed array
2. Module 2: Class CBT contains:
   1. public CBT () – Instantiates the array and assigns default values
   2. public int[] findMin () – Returns lowest key/value pair in CBT
   3. public void deleteMin () – Removes lowest key/value pair from CBT
   4. public void insert (int key, int value) – Inserts passed key/value pair into CBT
   5. public int[][] getCBT () – Returns CBT in ordered format
3. Module 3: Class SBT contains:
   1. private class Node – Defines nodes used to store data
   2. public int[] findMin () – Returns lowest key/value/size set in SBT
   3. public void deleteMin () – Removes lowest key/value/size set from SBT
   4. public void insert (int key, int value) – Inserts passed key/value pair into SBT
   5. public int[][] getSBT (String type) – Returns SBT in format based on String arg
   6. private void preOrderGetSBT (int[][] data, Node node) – Creates array with nodes from tree in preorder format
   7. private void inOrderGetSBT (int[][] data, Node node) – Creates array with nodes from tree in inorder format
   8. private void insertionOrderGetSBT (int[][] data, Node node, int pos) – Creates array with nodes from tree in order they were inserted

2.2 Pseudocode for the Modules

2.2.1 Pseudocode for Hw3Main

1a. Hw3Main Pseudocode Refinement #1:

// Method main – Gets information from the user and randomly calls methods to perform operations

// Method printCBT – Shows effects of each operation on CBT with 30 values

// Method printPreOrderSBT – Shows effects of each operation on SBT with 30 values (printed in preorder format)

// Method printInOrderSBT – Shows effects of each operation on SBT with 30 values (printed in inorder format)

// Method printCBTArray – Prints values in passed array

// Method printSBTArray – Prints values in passed array

2a. Hw3Main Pseudocode Refinement #2:

// Method main – Gets information from the user and randomly calls methods to perform operations

// Declare and Instantiate Scanner and Random objects

// Declare and Instantiate SBT and CBT objects

// Declare and Instantiate other variables

// Obtain sequence length from user

// Enter a loop that will perform operations

// Randomly decide operation

// Perform operation

// Check if size of tree is 30

// If so, call printing methods

// Set a flag so it cannot be printed again

// If not, continue with loop

// Method printCBT – Shows effects of each operation on CBT with 30 values

// Declare testing CBT object and get current values of main CBT

// Do the following for each operation in CBT

// Print the array before the operation

// Run the operation

// Print the array after the operation

// Method printPreOrderSBT – Shows effects of each operation on SBT with 30 values (printed in preorder format)

// Declare testing SBT object and get current values of main SBT

// Do the following for each operation in SBT

// Print the array before the operation (preorder format)

// Run the operation

// Print the array after the operation (preorder format)

// Method printInOrderSBT – Shows effects of each operation on SBT with 30 values (printed in inorder format)

// Declare testing SBT object and get current values of main SBT

// Do the following for each operation in SBT

// Print the array before the operation (inorder format)

// Run the operation

// Print the array after the operation (inorder format)

// Method printCBTArray – Prints values in passed array

// Enter a for loop that will go through all the values in the passed CBT array

// Print each value in the desired format

// Method printSBTArray – Prints values in passed array

// Enter a for loop that will go through all the values in the passed SBT array

// Print each value in the desired format

2.2.2 Pseudocode for CBT

2a. CBT Pseudocode Refinement #1:

// Method CBT – Instantiates array and assigns default values

// Method findMin – Returns lowest key/value pair in CBT

// Method deleteMin – Removes lowest key/value pair from CBT

// Method insert – Inserts passed key/value pair into CBT

// Method getCBT – Returns CBT in ordered format

2b. CBT Pseudocode Refinement #2:

// Method CBT – Instantiates array and assigns default values

// Instantiate array of size 1

// Assign default value (greater than any possible value)

// Method findMin – Returns lowest key/value pair in CBT

// Check if value exists in array

// If so, return value at position 0

// Otherwise, return an “empty” value

// Method deleteMin – Removes lowest key/value pair from CBT

// Check if value exists in array

// If so…

// Decrement number of values in array

// Adjust size of binary tree (if necessary)

// Move last value to root and clear previous position

// Heapify the tree

// Method insert – Inserts passed key/value pair into CBT

// Adjust size of binary tree (if necessary)

// Put passed key/value in first open position in array

// Heapify the tree and increment number of values in array

// Method getCBT – Returns CBT in ordered format

// Create array of set size

// Copy binary tree into new array

2.2.3 Pseudocode for SBT

3a. SBT Pseudocode Refinement #1:

// Class Node – Defines nodes used to store data

// Method findMin – Returns lowest key/value/size set in SBT

// Method deleteMin – Removes lowest key/value/size set from SBT

// Method insert – Inserts passed key/value pair into SBT

// Method getSBT – Returns SBT in format based on String arg

// Method preOrderGetSBT – Creates array with nodes from tree in preorder format

// Method inOrderGetSBT – Creates array with nodes from tree in inorder format

// Method insertionOrderGetSBT – Creates array with nodes from tree in order they were inserted

3b. SBT Pseudocode Refinement #2:

// Class Node – Defines nodes used to store data

// Declare attributes that will contain left child, right child, and parent nodes, as well as key, value, and size

// Node constructor with no arguments that sets nodes to null and key, value, and size to “empty” values

// Node constructor with key and value arguments that sets nodes to null, key and value to passed arguments, and size to “empty” value

// Method findMin – Returns lowest key/value/size set in SBT

// Returns root’s key, value, and size

// Method deleteMin – Removes lowest key/value/size set from SBT

// If tree has more than 1 node…

// Create a temporary node

// Find the lowest node based on size and right default

// Move the lowest node’s information to the root

// Delete the lowest node

// Heapify the tree

// If the tree has only 1 node, simply replace the root node’s values with the default values

// Method insert – Inserts passed key/value pair into SBT

// If tree has a node…

// Create a temporary node

// Find the place where the next node is inserted

// Create a new node there

// Heapify the tree

// If the tree doesn’t have any nodes, assign the values to the root node

// Method getSBT – Returns SBT in format based on String arg

// Create an empty binary tree

// Check what order the values are wanted in

// Call the appropriate method

// Return the binary tree with values ordered correctly

// Method preOrderGetSBT – Creates array with nodes from tree in preorder format

// Assign values to array while going through tree in preorder format (using recursion)

// Method inOrderGetSBT – Creates array with nodes from tree in inorder format

// Assign values to array while going through tree in inorder format (using recursion)

// Method insertionOrderGetSBT – Creates array with nodes from tree in order they were inserted

// Assign values to array while going through tree in order that values were inserted (using recursion)

**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**

|  |  |  |
| --- | --- | --- |
| Input/Output Analysis | | |
|
| **Method** | **Input Size** | **Output Size** |
| Hw3Main | 0 | 0 |
| findMin (CBT) | 0 | n |
| deleteMin (CBT) | 0 | 0 |
| insert (CBT) | 2 | 0 |
| findMin (SBT) | 0 | n |
| deleteMin (SBT) | 0 | 0 |
| insert (SBT) | 2 | 0 |

The table above shows an analysis of the input and output for each method. The value n that is output by the findMin functions depends on how many values are in the tree at the time it is returned.

The table below shows an analysis of the space used within each method as well as the big O notation for each of the resulting equations. The reason for needing n space in the deleteMin and insert methods for CBT is due to the need for temporary arrays used to hold values while array sizes are being changed.

|  |  |  |
| --- | --- | --- |
| Space Complexity Analysis | | |
|
| **Method** | **Space Usage** | **Space Complexity** |
| Hw3Main | 11 | O(1) |
| findMin (CBT) | 2 | O(1) |
| deleteMin (CBT) | 2n + 3 | O(n) |
| insert (CBT) | n + 3 | O(n) |
| findMin (SBT) | 3 | O(1) |
| deleteMin (SBT) | 10 | O(1) |
| insert (SBT) | 14 | O(1) |

|  |  |  |
| --- | --- | --- |
| Time Complexity Analysis | | |
|
| **Method** | **Instruction Count** | **Time Complexity** |
| Hw3Main | (22/5)n^2 + (46/5)n\*log(n) + 40n + 10 | O(n^2) |
| findMin (CBT) | 3 | O(1) |
| deleteMin (CBT) | 12n + 10 log(n) + 11 | O(n) |
| insert (CBT) | 10n + 7 log(n) + 3 | O(n) |
| findMin (SBT) | 2 | O(1) |
| deleteMin (SBT) | 16 log(n) + 9 | O(log(n)) |
| insert (SBT) | 13 log(n) + 7 | O(log(n)) |

Most of the time complexities that I calculated (shown in table above) ended up agreeing fairly well with the data that I collected (shown in the graphs on the next 2 pages). As it can be seen, the time complexity for SBTs is slightly better than CBTs in all tests besides the findMin since the CBTs use arrays while the SBTs use explicit representations. Had I simply set the arrays to a size which the tree would never cross (such as the number of operations, so even if all the operations were insert, the tree would never be bigger than the array), than there would be very little difference between the SBTs and CBTs. This shows in the findMin graph where the CBT actually ends up being better than the SBT since data is not moved. However, since the arrays require being made bigger and smaller as the size of the tree changes (to conserve space), extra time is needed to allocate space for the new array and transfer values from the old array to the new one. This results in the CBTs taking increasing extra time as the number of operations increases. In this way, SBTs are better than CBTs (however, SBTs can require much more space when it comes to needing references between nodes).

I also noticed that while the insert and deleteMin methods for CBTs are of time complexity O(n), they decrease as the number of operations increases. I’m assuming that this is due to the fact that I have my arrays set up to change size between powers of 2 depending on how many values are in the array. As the array grows bigger (which is more likely when you have more operations), the array changes size less often, resulting in a lower average time. This would probably be much different if I were to declare the array to be large only once at the very beginning of the program (which would probably result in a lower, more constant time).

A similar thing occurs with the total execution time. While it would be expected to be linear, it instead resembles more of a log(n) form. I believe this is due to the fact that O(n^2) is a worst-case scenario in which you have all insert and deleteMin operations, each taking O(n) time. Instead, with a bunch of findMin operations mixed in (taking O(1)), the total execution time is lowered as a result.

**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.