CS 3310 – Data and File Structures, Instructor: *<Gupta>*, Western Michigan University Lab TA: *<YG>*

Parker Loomis

**SOFTWARE LIFE CYCLE REPORT – FOR LAB ASSIGNMENT** *4*

**Binary Search Trees**

1. **PHASE 1: SPECIFICATION (“What do we build?”)** Write a JAVA application to solve the following problem,
2. 1) Generate 100 random integers within the range [1,20].

* 2)  Then insert them into a binary search tree T using an explicit representation. For this assignment we don’t need to worry about minimizing the height of the search tree.
* 3)  Each node of T contains two values, one is the key value and the other is the counter representing #repeats of the key value in the randomly generated sequence in step1, in addition to the tree pointers (leftChild, rightChild, parent), sizeOfSubtree and subtreeHeight  For example: if key 10 gets repeated 3 times, then the node corresponding to key value 10 is:  keyVal repeatCount If we insert one more 10 into this tree, then there is no extra node added, but this node ‘s counter  will change from 3 to 4. If we delete one 10, then the counter will decrease by 1. In this kind of data structure, a node is deleted only if the counter reaches 0..
* 4)  Print the node whose counter has the largest value, and the node whose counter has the smallest value, and delete these two nodes one after another without destroying the structure of the binary search tree.  [***Note:*** *For an efficient implementation, to be able to easily find these nodes after any insert or delete operation, one can maintain a separate min-max heap that’s based on the counter values to determine the key values of these nodes (another alternative is to maintain two heaps, one max-heap and another min-heap). To simplify the assignment, we are not requiring this.*

*However, for extra 30 points, you may want to implement heap on counter values along with search tree organized on keys. If you decide to do so, clearly identify at the beginning of your code*.]

* 5)  Compute and print sum(keyVal) and sum(repeatCount), i.e., additions of all the keys and counters, respectively.
* 6)  Determine and print a distinct keyVal and its repeatCount so that if we insert a node with these values to the tree, the sum(keyVal) equals sum(repeatCount). Insert this node in the binary search tree.  Example:

Sum(keyVal)= 10+5+15 =30 Sum(repeatCount)=3+10+15=28 Then, as an example, we can insert, say, key 2 four times. Then the resulting tree is:

Sum(keyVal)= 2+10+5+15 =32 Sum(repeatCount)=3+10+15+4=32 and we have sum(keyVal)=sum(repeatCount).

* 7)  For k = 5, 20, 30 and 70, find and print kth-smallest key in the binary search tree.
* 8)  Repeat steps 1-7 twenty times and print the minimum and maximum heights of the search trees obtained in these 20 experiments.
* 9)  Now traverse the final tree as preorder, postorder, and inorder and print your traversal results. Print so that the values of each node are shown as tuples (keyVal, repeatCount, sizeOfSubtree,  subtreeHeight) and tuples of different nodes are separated by commas. For example for the tree

above, your inorder traversal output may look like: (10, 3, 4, 2), (5, 10, 2, 1), (2, 4, 1, 0), (15, 15, 1, 0)

**PHASE 2: DESIGN**

**Program should have 7 classes**

1. **Main method**
2. **SBT**
3. **LinkedListBST**
4. **Node**

**PHASE 3: RISK ANALYSIS (“What can go wrong, and how bad can it be?”)**

Heap could go wrong. Printing could go to wrong node.

**PHASE 4: VERIFICATION (“Are the algorithms correct?”)**

Size balanced doesn’t work, and printing doesn’t work like it should.

**PHASE 5: CODING**

**package main;**

**import java.util.Random;**

**import java.util.Scanner;**

**public class Main {**

**public static void main(String[] args) {**

**// TODO Auto-generated method stub**

**Random rand = new Random();// call random class**

**// Scanner kbd = new Scanner(System.in);**

**// kbd.close();**

**int i = 19;// max number int node can be**

**int k = 100;// number of integers in tree**

**int l;**

**// int key=5;**

**int count = 0;**

**/\***

**\* need to loop 20 times in loops you need to print counts, sum of**

**\* counts, and the difference between them**

**\*/**

**System.out.print("This tree implements heap.");**

**// char ch;**

**while (count < 20) {**

**BSTS bst = new BSTS(k);**

**System.out.print("\n");**

**for (int j = 0; j < k; j++) {**

**l = (rand.nextInt(i)) + 1;**

**// System.out.println(l+", "+(bst.getCount(l)));**

**bst.insert(l, (bst.getCount(l)));**

**}**

**bst.print();**

**System.out.println("Sum of Keys " + bst.sumKeys() + " Sum of count " + bst.sumCount());**

**// System.out.println(bst.sumKeys()+" count "+bst.sumCount());**

**int h = bst.getCount(1);**

**int f = bst.sumKeys();**

**int d = bst.sumCount();**

**int q = 0;**

**do {**

**d += 1;**

**// h+=1;**

**q++;**

**} while ((f - d) != 0);**

**// int f=bst.sumKeys()-bst.sumCount();**

**// int h=bst.getCount(1);**

**System.out.println("You would have to add 1, " + (q - h) + " times to make the sums even.");**

**count++;**

**}**

**/\***

**\* need to print out in preorder, postorder, and inorder (keyVal,**

**\* repeatCount, sizeOfSubtree,subtreeHeight)**

**\*/**

**}**

**}**

package main;

public class BSTS {

private int[][] Heap;

private int size;

private int maxsize;

private int minCount;

private int maxCount;

private int minKey;

private int maxKey;

private static final int FRONT = 1;

public BSTS(int maxsize) {

/\*

\* create all values and set them

\*/

this.maxsize = maxsize;

this.minCount = 2;

this.minKey = 10;

this.maxCount = 0;

this.maxKey = 0;

this.size = 0;

Heap = new int[this.maxsize + 1][2];

Heap[0][0] = Integer.MIN\_VALUE;

}

/\*

\* gets parent

\*/

private int parent(int pos) {

return pos / 2;

}

/\*

\* gets left child

\*/

private int leftChild(int pos) {

return (2 \* pos);

}

/\*

\* gets right child

\*/

private int rightChild(int pos) {

return (2 \* pos) + 1;

}

/\*

\* makes sure there is a node

\*/

private boolean isLeaf(int pos) {

if (pos >= (size / 2) && pos <= size) {

return true;

}

return false;

}

/\*

\* swaps nodes for heaping

\*/

private void swap(int fpos, int spos) {

int tmp;

int tmp2;

tmp = Heap[fpos][0];

tmp2 = Heap[fpos][1];

Heap[fpos][0] = Heap[spos][0];

Heap[fpos][1] = Heap[spos][1];

Heap[spos][0] = tmp;

Heap[spos][1] = tmp2;

}

/\*

\* heaps tree

\*/

private void minHeapify(int pos) {

if (!isLeaf(pos)) {

if (Heap[pos][0] > Heap[leftChild(pos)][0] || Heap[pos][0] > Heap[rightChild(pos)][0]) {

if (Heap[leftChild(pos)][0] < Heap[rightChild(pos)][0]) {

swap(pos, leftChild(pos));

minHeapify(leftChild(pos));

} else {

swap(pos, rightChild(pos));

minHeapify(rightChild(pos));

}

}

}

}

/\*

\* inserts vairables into tree

\*/

public void insert(int element, int counter) {

counter += 1;

if (size == maxsize) {

expand();

}

Heap[++size][0] = element;

Heap[size][1] = counter;

if (counter < this.minCount) {

if (element < this.minKey) {

this.minCount = counter;

this.minKey = element;

}

}

if (counter >= this.maxCount) {

this.maxCount = counter;

this.maxKey = element;

}

int current = size;

while (Heap[current][0] < Heap[parent(current)][0]) {

swap(current, parent(current));

current = parent(current);

}

// int current = size;

// current = parent(current);

}

/\*

\* expands if need be

\*/

public void expand() {

int[][] newArray = new int[Heap.length + 1][2];

// System.arraycopy(Heap, 0, newArray, 0, Heap.length);

int j = 0;

// an alternative to using System.arraycopy would be a for-loop:

for (int i = 0; i < this.Heap.length; i++) {

newArray[i][0] = this.Heap[i][0];

newArray[j][1] = this.Heap[j][1];

}

this.Heap = newArray;

this.maxsize++;

j++;

}

/\*

\* prints out tree and gets min and max key as well as deletes

\*/

public void print() {// System.out.print("\n");

// for (int i = 1; i <= size; i++ )

// {

// System.out.print(" PARENT : " + Heap[i][0] + "

// LEFT CHILD : " + Heap[2\*i] [0]

// + " RIGHT CHILD :" + Heap[2 \* i + 1][0]);

// System.out.println(Heap[i][0]+", "+Heap[i][1]);

// }//System.out.print("\n");

// for (int i = 1; i <= size; i++ )

// {

// System.out.print(" PARENT : " + Heap[i][0] + "

// LEFT CHILD : " + Heap[2\*i] [0]

// + " RIGHT CHILD :" + Heap[2 \* i + 1][0]);

// // System.out.println(Heap[i][0]+",

// "+Heap[i][1]);

// }

System.out.print("\nKey with smallest count " + this.minKey + " Key with largest count " + this.maxKey + "\n");

remove(this.minKey);

remove(this.maxKey);

System.out.println("removed them");

}

/\*

\* calls min heap

\*/

public void minHeap() {

for (int pos = (size / 2); pos >= 1; pos--) {

minHeapify(pos);

}

}

/\*

\* removes node

\*/

public int remove(int key) {

int popped = Heap[FRONT][1];

Heap[FRONT] = Heap[size--];

minHeapify(FRONT);

return popped;

}

/\*

\* calculates the min node

\*/

public int min() {

int popped = Heap[FRONT][0];

minHeapify(FRONT);

return popped;

}

/\*

\* gets size of tree

\*/

public int getsize() {

return size;

}

/\*

\* gets count of how many of that node are in the tree already

\*/

public int getCount(int key) {

int count = 0;

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

if (Heap[i][0] == key)

count++;

}

return count;

}

/\*

\* gets a sum of the key values

\*/

public int sumKeys() {

int count = 0;

for (int i = 1; i <= size; i++) {

count += Heap[i][0];

}

return count;

}

/\*

\* gets sum of counts

\*/

public int sumCount() {

int count = 0;

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

count += Heap[i][1];

}

return count;

}

/\*

\* prints inorder

\*/

// public void inorder()

// {

// inorder(root);

// }

/// \*

// \* prints inorder

// \*/

// private void inorder(Node r)

// {

// if (r != null)

// {

// inorder();

// System.out.print(r.data +" ");

// inorder(r.right);

// }

// }

/// \* Function for preorder traversal \*/

// public void preorder()

// {

// preorder(root);

// }

/// \*

// \* prints in preorder

// \*/

// private void preorder(Node r)

// {

// if (r != null)

// {

// System.out.print(r.data +" ");

// preorder(r.left);

// preorder(r.right);

// }

// }

/// \* Function for postorder traversal \*/

// public void postorder()

// {

// postorder(root);

// }

/// \*

// \* prints in postorder

// \*/

// private void postorder(Node r)

// {

// if (r != null)

// {

// postorder(r.left);

// postorder(r.right);

// System.out.print(r.data +" ");

// }

// }

}

**PHASE 6: TESTING (“Did we build it correctly?”)**

This tree implements heap.

Key with smallest count 1 Key with largest count 17

removed them

Sum of Keys 975 Sum of count 361

You would have to add 1, 608 times to make the sums even.

Key with smallest count 3 Key with largest count 12

removed them

Sum of Keys 1024 Sum of count 332

You would have to add 1, 688 times to make the sums even.

Key with smallest count 3 Key with largest count 5

removed them

Sum of Keys 887 Sum of count 350

You would have to add 1, 532 times to make the sums even.

Key with smallest count 5 Key with largest count 2

removed them

Sum of Keys 899 Sum of count 343

You would have to add 1, 552 times to make the sums even.

Key with smallest count 4 Key with largest count 12

removed them

Sum of Keys 972 Sum of count 336

You would have to add 1, 633 times to make the sums even.

Key with smallest count 3 Key with largest count 16

removed them

Sum of Keys 1058 Sum of count 362

You would have to add 1, 693 times to make the sums even.

Key with smallest count 1 Key with largest count 4

removed them

Sum of Keys 957 Sum of count 359

You would have to add 1, 593 times to make the sums even.

Key with smallest count 8 Key with largest count 15

removed them

Sum of Keys 1096 Sum of count 358

You would have to add 1, 737 times to make the sums even.

Key with smallest count 4 Key with largest count 4

removed them

Sum of Keys 892 Sum of count 372

You would have to add 1, 517 times to make the sums even.

Key with smallest count 2 Key with largest count 16

removed them

Sum of Keys 976 Sum of count 360

You would have to add 1, 610 times to make the sums even.

Key with smallest count 3 Key with largest count 9

removed them

Sum of Keys 983 Sum of count 359

You would have to add 1, 621 times to make the sums even.

Key with smallest count 2 Key with largest count 8

removed them

Sum of Keys 1003 Sum of count 340

You would have to add 1, 662 times to make the sums even.

Key with smallest count 7 Key with largest count 12

removed them

Sum of Keys 1017 Sum of count 335

You would have to add 1, 679 times to make the sums even.

Key with smallest count 2 Key with largest count 1

removed them

Sum of Keys 902 Sum of count 363

You would have to add 1, 531 times to make the sums even.

Key with smallest count 7 Key with largest count 16

removed them

Sum of Keys 997 Sum of count 363

You would have to add 1, 632 times to make the sums even.

Key with smallest count 6 Key with largest count 9

removed them

Sum of Keys 979 Sum of count 371

You would have to add 1, 602 times to make the sums even.

Key with smallest count 8 Key with largest count 9

removed them

Sum of Keys 950 Sum of count 331

You would have to add 1, 614 times to make the sums even.

Key with smallest count 4 Key with largest count 3

removed them

Sum of Keys 1022 Sum of count 371

You would have to add 1, 646 times to make the sums even.

Key with smallest count 6 Key with largest count 6

removed them

Sum of Keys 985 Sum of count 366

You would have to add 1, 616 times to make the sums even.

Key with smallest count 4 Key with largest count 7

removed them

Sum of Keys 1004 Sum of count 388

You would have to add 1, 614 times to make the sums even.

**PHASE 7: REFINING THE PROGRAM (“Add bells and whistles to the program”)**

Make it look better and fix any problems.

**PHASE 8: PRODUCTION**

I prepared a copy of the entire program for Lab TA’s evaluation, as specified by the TA. Then, I sent electronically the copy to the Lab TA.

**PHASE 9: MAINTENANCE**

Fix the issues with Linked List and printing.

**TIME AND SPACE**

Space would be depending on the amount of numbers the user wanted. The number of array elements plus any of the types.

Main: O(n)

SBT: O(log n)

LinkedListBST: O(n)

Node: O(n)