|  |  |  |
| --- | --- | --- |
| American University of Sharjah  College of Engineering  Department of Computer Science & Engineering  P. O. Box 26666, Sharjah, UAE |  | **Instructors:** Dr. Michel Pasquier  **Lab Instructor:** Praveena Kolli  **Office:** EB2-12  **Phone**: 971-6-5152352  **e-mail**: pkolli@aus.edu  **Semester**: Spring 2021 |

**CMP305L - Data Structures and Algorithms Lab**

**Lab. Assignment 12 – AVL-balanced Binary Search Trees**

***Objectives***

* To compare and appreciate the performance of AVL trees vs. BSTs and lists
* To understand balanced trees and balance measures, compute and compare them.

***Instructions***

* Do not use any static or global variable. Use recursion!

***Note:***

***Lab:*** Exercises 1 and 2 (10 marks)

***Bonus*:** Exercise 3 (1 mark)

***Exercise 1:***

The goal of this exercise is to demonstrate how efficient using a *balanced* Binary Search Tree is, compared to a plain Binary Search Tree (BST) or a much simpler alternative such as a list.

You are provided with the textbook implementations of AVL tree and BST classes, and the full code of the test program (also in Appendix 1). This program will repeatedly create AVL trees and BSTs of increasing sizes by inserting random data into them, and compute the min, max, and average height of the generated trees.

Run the test program and store the results in a CSV text file (feel free to modify the output part as you see fit). Use Excel to plot performance graphs i.e., min / max / avg heights as a function of *n*, for both AVL trees and BSTs (all on the same plot). You may wish to adjust experimental parameters to get better results (e.g., using more data and/or samples, generating more trees, etc.)

Submit your findings (results and graphs) and your comments (analysis) about the observed performance of AVL trees vs. Binary Search Trees vs. List data structures. Recall the height of a BST is a measure of the worst-case computational complexity for finding an element in it. (Note that the code offers the option to partially sort the input data; you may want to try that too…)

A few observations and analysis we can deduce from the graphs above:

1. AVL Trees are always faster than BST, regardless of sorting (Although the difference in performance is larger as the data gets sorted – More below). The curves for the min, max and average performance for AVL trees are always below the min, max and average performance for BSTs, which indicate that AVL trees always perform better than BSTs in all cases. Infact, the min curve of the BST is always above the max curve of the AVL, which means AVL’s worst case scenario would still be faster than BST’s best case.
2. Notice, with little to no sorting taking place, all the curves roughly follow the log­2N graph, since both AVL and BST would be closer to a balanced tree. However, as the data that’s inserted becomes sorted, BST becomes more like a linked list, hence the curve looks closer to a linear curve. While AVL trees still follow a log2N, which is much smaller than a linear curve for large values of N.

Hence, in general, AVL Trees perform better than BST for large values of N.

***Exercise 2:***

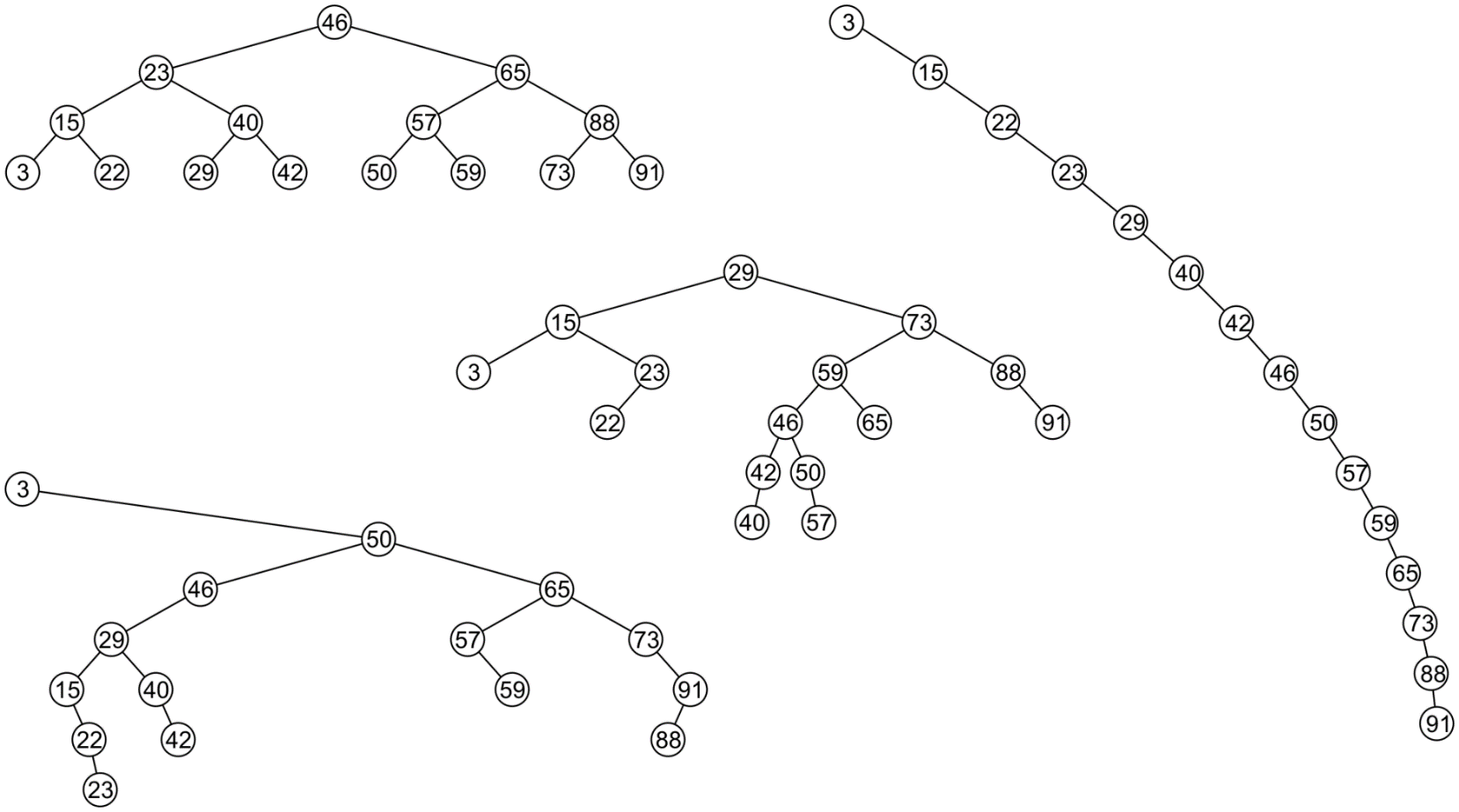
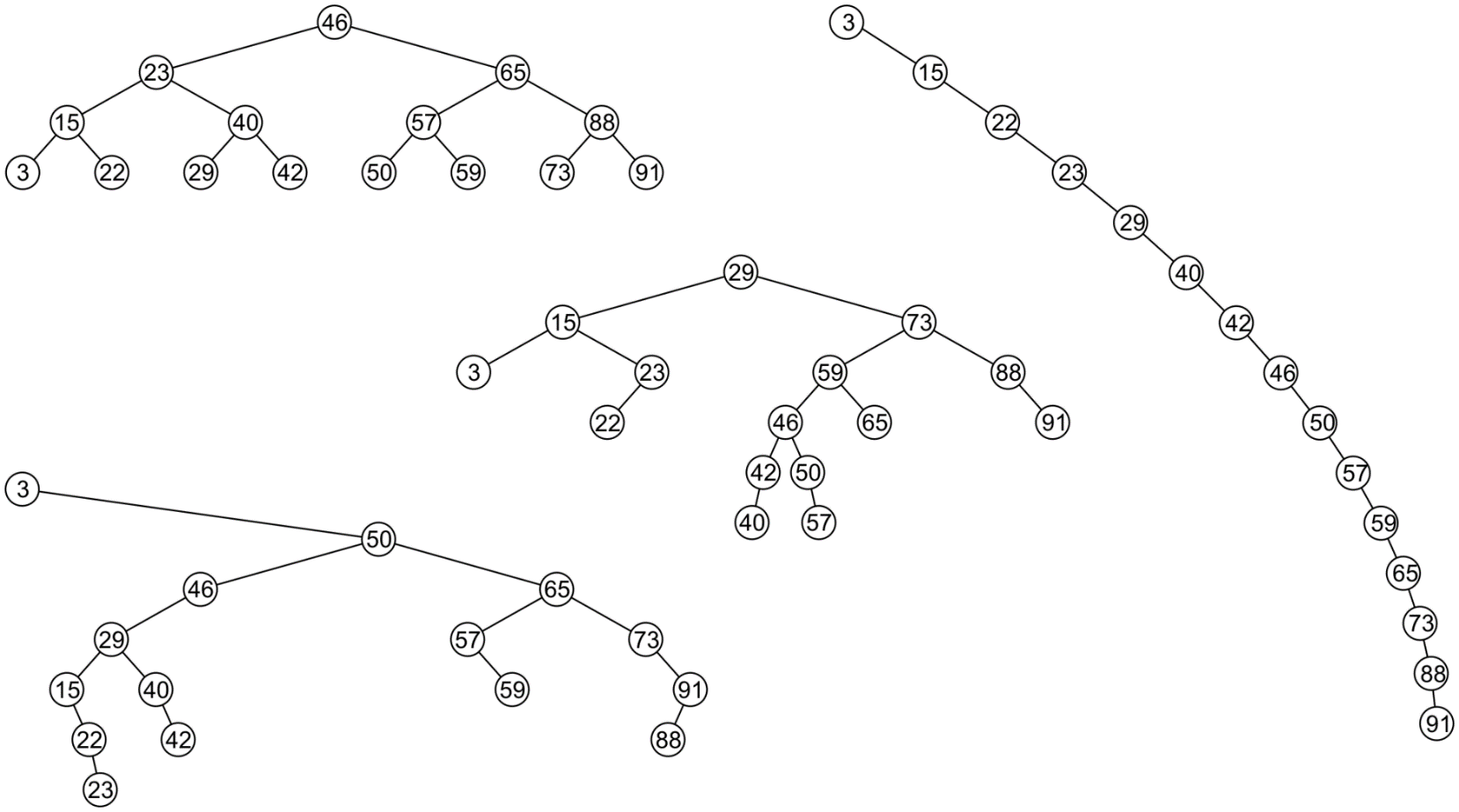
Calculating a *balance* measure helps to determine how well a binary tree is AVL-balanced, as per the following steps – for any given node:

* calculate *d* = **|** height of the left subtree – height of the right subtree **|**
* return 0 if *d* is 0 or 1 (the tree *is* AVL-balanced)
* return *d-*1 if *d* > 1 (the tree is *not* AVL-balanced; *d* tells us how badly)

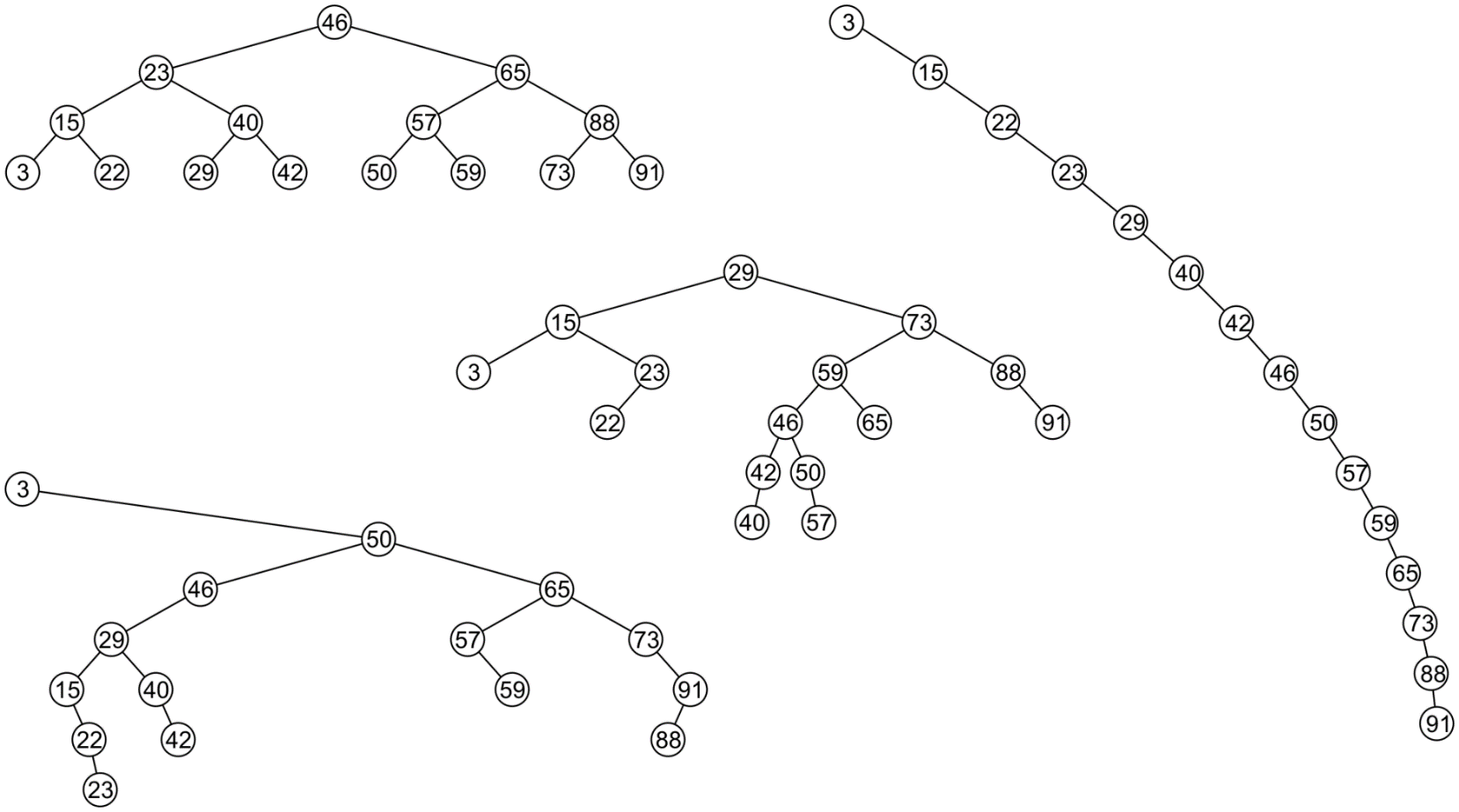
First, update the BinaryNode structure with an integer attribute (with a default of 0) that represents the balance value at that node. Then, write the following member functions of the Binary Search Tree class:

* void updateAvlBalance() that recursively computes the balance value at each node.
* void printAvlBalance(int threshold) that recursively prints the node element   
  and balance value of all nodes whose balance exceeds the given threshold.

Use the code given in Appendix 2 to create a Binary Search Tree for each of the four test cases below. Call updateAvlBalance() on them and use printAvlBalance() to show the results i.e., all nodes with a balance threshold larger than 1.



BST #1 BST #2



***Diagram

Description automatically generated***

BST #3 BST #4

***Bonus:***

***Exercise 3:***

For Balance-Bounded or BB() Trees, the balance measure is the *weight* ratio i.e., the ratio of empty nodes in the left sub-tree over the total number of empty nodes in the tree.

Assuming we use  =0.25, for instance, a node will be balanced if the ratio is between 0.25 and 0.75. A binary tree is balanced if all of its nodes are balanced.

Write the following member functions of the Binary Search Tree class:

* void updateBbaBalance() that recursively computes the balance value at each node.
* void printBBaBalance(int alpha) that recursively prints the node element   
  and balance value of all nodes whose balance is not within acceptable range.

Test the four Binary Search Trees created in Ex. 2, show the results, and indicate whether any tree is BB() balanced but not AVL balanced, or conversely. (Use  =0.25.)

***Appendix: Ex 1***

int main() {

const int DATASIZE = 3000, *// total number of elements to be stored*

NBSAMPLES = 30, *// number of sample points for benchmarking*

NBTREES = 100, *// number of trees generated for averaging*

SORTED = 0; *// percentage of input data that is sorted*

vector<int> data;

for (int i = 1; i <= DATASIZE; i++) data.push\_back(i);

AvlTree<int> avl; BinarySearchTree<int> bst;

ofstream out("results.csv");

for (int n = DATASIZE/NBSAMPLES; n <= DATASIZE; n += DATASIZE/NBSAMPLES) {

int avlHeight, avlMax = 0, avlMin = DATASIZE, avlAvg = 0,

bstHeight, bstMax = 0, bstMin = DATASIZE, bstAvg = 0;

for (int k = 1; k <= NBTREES; k++) {

random\_shuffle(data.begin(), data.begin() + n);

if (SORTED > 0)

partial\_sort(data.begin(), data.begin()+n\*SORTED/100, data.end());

*///for (int i = 0; i < n; i++) cout << " " << data[i]; cout << endl;*

avl.makeEmpty(); bst.makeEmpty();

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

avl.insert(data[i]); bst.insert(data[i]);

}

avlHeight = avl.height(); avlAvg += avlHeight;

avlMin = min(avlHeight,avlMin); avlMax = max(avlHeight,avlMax);

bstHeight = bst.height(); bstAvg += bstHeight;

bstMin = min(bstHeight,bstMin); bstMax = max(bstHeight,bstMax);

}

string result = to\_string(n) + "\t |\t " + to\_string(avlMin)

+ "\t " + to\_string(avlAvg/NBTREES) + "\t "

+ to\_string(avlMax) + "\t |\t " + to\_string(bstMin) + "\t "

+ to\_string(bstAvg/NBTREES) + "\t " + to\_string(bstMax) + "\n";

cout << result;

out << result;

}

string header = "N\t |\tavlMin\tavlAvg\tavlMax\t |\tbstMin\tbstAvg\tbstMax\n";

cout << header;

out << header;

return 0;

}

***Appendix: Ex 2***

template <typename Object> void TestBalance(Object data[], int size) {

BinarySearchTree<Object> bst;

for (int i = 0; i < size; i++) bst.insert(data[i]);

//-> update balance values and print nodes that are not balanced

}

int main() {

TestBalance(new int[15] {46,23,65,15,40,57,88,3,22,29,42,50,59,73,91}, 15);

TestBalance(new int[15] {29,15,73,3,23,59,88,22,46,65,91,42,50,40,57}, 15);

TestBalance(new int[15] {3,50,46,65,29,57,73,15,40,59,91,22,42,88,23}, 15);

TestBalance(new char[7] {'R','U','H','C','K','A','E'}, 7);

return 0;

}