CSC2001F 2024 Data Structures Assignment 2

Instructions

The goal of this assignment is to test the performance of the AVL Tree to determine if AVL trees really do balance nodes and provide good performance irrespective of the size of the data.

Dataset

The dataset to be used is the same as in Assignment 1.

Part 1: Program

Create an application as described below to store and retrieve data using the AVL binary search tree data structure.

Write an application **`GenericsKbAVLApp`** to read in the attached text file (`GenericsKB.txt`) and store the data items within an AVL Tree. Each data item consists of a term (which should be used as the key – assume that there won't be duplicate keys in the dataset), sentence and confidence score.

After loading the dataset, a second file (`GenericsKB-queries.txt`) should be read in. This file consists of a list of items (one item per line) only. The application should perform a search for each of the query items (one at a time). If the item (e.g. "tree") is found in the AVL tree, the data item should be printed out (e.g. "tree: Trees remove carbon dioxide from the atmosphere. (1.0)"). If there is no match, an appropriate message should be printed out (e.g. "Term not found: tree").

Your AVL Tree implementation can be created from scratch or re-used from anywhere. You may use the sample code provided. You may NOT replace the AVL Tree with a different data structure. Use output redirection in Unix to save the output of each run to a file.

Test that your application can load the dataset correctly, and manually construct a query file with 10 queries to test that the application can handle queries with both terms in the dataset and terms not in the dataset correctly.

Part 2: Experiment

Conduct an experiment with **`GenericsKbAVLApp`** to compare the performance obtained with the theoretical performance of the algorithms.

Instrumentation

Add additional code to your program from Part 1 to count the number of comparison operations (<, >, =) you are performing in the code. Only count where you are comparing the keys. This is called **instrumentation**. There are 3 basic steps:

- First, create a variable/object (e.g., `opCount=0`) somewhere in your code to track the counter; maybe use an instance variable in the data structure class.
- Secondly, wherever there is an operation you want to count, increment the counter (`opCount++`). For example:

```
opCount++;  // instrumentation

if (queryString == theKey)
  ...
```

- Finally, report the value of the counter before the program terminates. Maybe add a method to write the value to a file before the program terminates or print it to the screen. Note that you will probably need two variables to count the search and insert operations separately.

Experiment

Vary the size (*n*) of the dataset loaded and measure the number of comparison operations in the best/average/worst case for different values of *n*. Use 10 values of *n* (up to 100 000) that are spaced approximately equally apart on a logarithmic scale (e.g. 10, 100, 1 000, 10 000, 100 000). For each value of *n*:

- Create a randomised subset of *n* entries from the sample data.
- Run the instrumented application with this subset and the query file (the queries will remain fixed regardless of the value of *n*). Store all operation count values for both insert and search operations.
- Determine the minimum (best case), maximum (worst case) and average of these count values (separately across all the insert and search operations).

Use graphs to compare the experimental values obtained with the theoretical complexity analysis for insert and search operations.

It is recommended that you use Python/Java programs to automate this process.

Report

Write a report (of up to 6 pages) that includes the following:

- What your OO design is: what classes you created and how they interact.
- What the goal of the experiment is and how you executed the experiment.
- Which test query values you used in Part 1 and what the output was in each case. Only show the output printed for each of the 10 queries.
- What the results of your instrumentation experiments are, showing the best, average and worst cases for the application and the theoretical expectation. Use one or more graph. Discuss what the results mean.
- A statement of what you included in your application(s) that constitutes creativity - how you went beyond the basic requirements of the assignment.
- Summary statistics from your use of git to demonstrate usage. Print out the first 10 lines and last 10 lines from "git log", with line numbers added. You can use a Unix command such as:

```
git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail -10)
```

Dev requirements

As a software developer, you are required to make appropriate use of the following tools:

- `git`, for source code management
- `javadoc`, for documentation generation
- `make`, for automation of compilation and documentation generation

Submission requirements

Submit a compressed archive `STUNUM-CSC2001F-A2.tar.gz` containing:

- Makefile
- src/
  - all source code
- bin/
  - all class files
- doc/
  - javadoc output
- report.pdf

Your report must be in PDF format. Do not submit the git repository.

Marking Guidelines

Your assignment will be marked by tutors, using the following marking guide.

Artefact	*Aspect*	*Mark*
Report (19)	Appropriate design and implementation of OOP and data structures:	
	Overall OOP design	3
	AVL tree implementation	2
	Experimental tests:	
	Experiment description	3
	Trial test values and outputs (Part 1)	2
	Results - tables and/or graphs	3
	Discussion of results	3
	Description of creativity	3
Code (15)		
	OOP design	3
	Reading data from file	3
	Processing search queries	3
	AVL tree implementation	3
	Implementation of creativity	3

Dev (11)	Git usage log	2
	Documentation (javadoc)	6
	Makefile: make and clean targets	3

Total (45)

Additional resources for assignment

- `GenericsKB.txt`
- `GenericsKB-queries.txt`