CS 201 - Data Structures II:
Assignment #3

Muhammad Mobeen Movania (L1),
Syeda Saleha Raza (L2),
Faisal Alvi (L3, L4),
Abdullah Zafar (L5).

Due on March 30, 2024, 11.59pm

Student 1 Name, ID **Student 2 Name, ID**

Instructions

This assignment document consists of two problems.

- Problem 1 is a programming based question which requires implementation. It must be submitted by pushing all your code files to the Github repository. This problem is worth 40 points.
- Problem 2 is a theoretical question which requires analysis. It should be completed and submitted within this document as a pdf on Canvas. This problem is worth 20 points.

Problem 1

(40 points) [**Implementing File System Index using AVL and BST**]
You have learned about database indexes in CS 355 Database Systems. Indexes are used across various types of software, including the database management system (DBMS), operating system, and other applications to efficiently search and retrieve data. For example, an index in a database is a sorted list of key values with the storage locations of rows in the table that contain the key value. Similarly, the file system index in an operating system contains an entry for each file name and the starting location of the file on disk. Each key in the index is associated with a particular pointer (or a list of pointers) to a record in the data file, as shown in Figure 1.

	fileName	ext	size	...
1	abstract	doc	239	...
2	background	zip	750	...
3	chapter1	doc	2056	...
4	experiments	xls	4089	...
5	abstract	pdf	329	...
6	chapter1	Pdf	550	...
7	abstract	zip	192	...
8	Intro	doc	3089	...

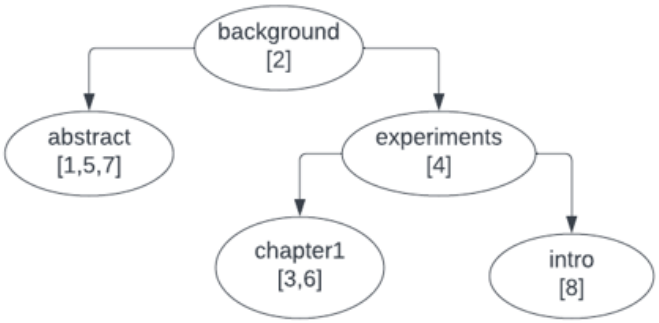

Figure 1: AVL index created on a list of files.

Self-balancing trees have been a popular choice to store indexes. Most commercial DBMSs use B-Tree/B+ Tree to maintain indexes. In this question, you will build your own BST and AVL tree based indexes to index files stored on your file system. These indexes will help you building a mini file explorer that can efficiently search for files on different criteria. The question involves:

- Loading file details (FileName, Size,accessedOn, modifiedOn,Type,path) from a given folder/CSV file on your file system
- Indexing these files using BST and AVL tree for efficient search
- Using these indexes in your mini file explorer to search for files on different criteria
- Performing a comparison of BST vs AVL indexes in terms of number of nodes that they traverse during different search operations. For this, we have used a dataset adapted from the Kaggle dataset on `https://www.kaggle.com/datasets/cogitoe/crab`.

This question provides you a minimal code structure and function interfaces that you have to maintain in your solution. Besides that, you are free to devise your own structure for the implementation of tree and its helper functions.

Dataset

You have been given three datasets and their details are as follows:

1. Small dataset which contains 1,000 records and is represented by the `Small.csv` file.
2. Medium dataset which contains 10,000 records and is represented by the `Medium.csv` file.
3. Large dataset which contains 50,000 records and is represented by the `Large.csv` file.

In addition, you have been given a sample dataset which contains 6 enteries for testing purposes.

Problem 1 continued on next page...

Filename	Type	Size	accessedOn	modifiedOn	path
15420-8	zip	99661	2016-10-14T21:21:17	2005-03-21T12:35:18	/gutenberg/www.gutenberg.lib.md.us/15420-8.zip
15420-8	zip	99761	2016-10-14T21:21:17	2005-03-25T12:35:18	/gutenberg1/www.gutenberg.lib.md.us/15420-8.zip
15420-9	zip	99631	2016-10-14T21:21:17	2005-03-22T12:35:18	/gutenberg/www.gutenberg.lib.md.us/15420-9.zip
15420-10	zip	99561	2016-10-14T21:21:17	2005-03-23T12:35:18	/gutenberg/www.gutenberg.lib.md.us/15420-10.zip
15420-11	zip	99961	2016-10-14T21:21:17	2005-03-24T12:35:18	/gutenberg/www.gutenberg.lib.md.us/15420-11.zip
15420-12	zip	99861	2016-10-14T21:21:17	2005-03-25T12:35:18	/gutenberg/www.gutenberg.lib.md.us/15420-12.zip

Table 1: Sample Dataset

Code Structure

The code base contains four main classes and their description are as follows:

1. The `FileSystem` class provides the functionality of loading file system data from a CSV file, and then storing it into a vector so that it can be later used to create File System Index.

2. The `BSTIndex` class represents a binary search tree index structure designed to efficiently organize and search file system data. It utilizes a binary search tree composed of TreeNode structures, each containing a key for indexing purposes and a vector storing the indices of file system entries associated with that key. This class provides methods for creating the index, adding new entries, and searching a particular entry.

3. The `AVLIndex` class extends the functionality of BSTIndex to implement an AVL tree-based index structure, which automatically balances itself to maintain optimal performance during insertion operations. It inherits from BSTIndex and overrides the add methods to incorporate AVL tree balancing logic.

4. The `FileExplorer` class implements a File System Explorer. It maintains a index based on Filename and modifiedOn using either AVL or BST. It provides methods for searching files by name, date, name and date combination, name and size combination, and files created during a specified date range.

Required Tasks

- Implement BST and AVL trees to build and maintain indexes on given file data. The index will be created on a given key and each node of the index will store a list of file records having that key. To perform this task, you have to implement the following methods in the `BSTIndex` class:
  - `void CreateIndex(vector<FileSystemEntry> &Data, const string &IndexType)` – will create index on the given key in files data
  - `void Add(const int i, const string &Key)` – will add a new key to the index along with the details of files containing that key.
- Searching works the same way in both BST and AVL tree. Implement the following generic method in the BSTIndex class.
  - `std::pair<std::vector<int>, int> Search(const string &Key)` – traverses the index to return details of a given key. This method also returns number of nodes visited while searching for the given key.
- Now you will use these indexes to build your mini file explorer. Your file explorer will use either BST/AVLIndex to perform searches on various criteria. The explorer will create index on `FileName` and `lastModifiedOn` and provides following options to search for files:

- `FileExplorer(const string &type, const string &csv_file)` – the constructor that creates indexes of the given type on filename and lastModified date.
- `void FindByName(const string &filename, const string &output_path)`– Saves the file paths having the given name in a text file at the provided directory.
- `void FindByDate(const string &date, const string &output_path)` - Saves the file paths that were last modified on the given date in a text file at the provided directory
- `void FindByNameAndDate(const string &filename, const string &date, const string &output_path)` - This method first queries over both indexes separately for the given name and date and then saves the intersection of their results in a text file at the provided directory.
- `void FindByNameAndSize(const string &filename, const int size, const string &output_path)`– This method first finds the files of the given name and then further filters them on size. The results are then saved in a text file at the provided directory.
- `void FindFilesCreatedDuring(const string &date1, const string &date2, const string &output_path)` – This method finds the files last modified during the given data range (inclusive) and saves the file paths in a text file at the provided directory.
  Note: AVL trees are not ideal for range-based queries. However, you have no other choice. Let's not iterate over the whole tree and try to traverse as few nodes as possible to perform this task.

Testing

You can test the correctness of your implementation by executing the provided `Pytest` file `test_file_explorer.py`. The Pytest file compares the output files, which are generated by your program and stored in the output folder, with the expected output files stored in the `test-output` folder. You can execute the `Pytest` file by typing the following command in the terminal:

```
pytest test_file_explorer.py
```

Listing 1: Executing Pytest file

Note: For more guidance on how to execute Pytest files, please course staff during their office hours

Grading Criteria

The rubric for this homework is as follows:

- 10 points (BST) (Correctness + Implementation)
- 15 points (AVL) (Correctness + Implementation)
- 15 points (FileExplorer) (Correctness + Implementation)

Penalties

- (-5) Code compiles with warnings
- (-5) OOP Inheritance not applied correctly between AVL and BST
- (-5) Implementation does not match the provided function signature.

- (-5) GitHub repository does not follow appropriate structure. All the code files should be in the code folder and output files in the output folder.

Compilation Guidelines: Before proceeding with submission, kindly verify that your code compiles utilizing the **C++17** standard.

Problem 2

(20 points) [**Analysis**]Assume that your FileExplorer has maintained separate AVL tree-based indexes for C and D drives on your hard disk. Now you have decided to merge them and have a unified index for the whole file system. Rather than re-creating the index from scratch, you want to combine existing indexes in an efficient way. Write pseudocode to combine two indexes with the complexity of $O(m + n)$. where m and n are number of nodes in two indexes.

Due date: March 30, 2024, 11:59 PM