# Project 3
# COP4530- Data Structures

November 14, 2024

## Instructions

In this project, you will implement a rudimentary Database Management System (`https://en.wikipedia.org/wiki/Database`) to facilitate INSERT, UPDATE, SEARCH, and DELETE functionalities (utilizing an underlying AVL tree structure).

## Motivation

AVL trees guarantee a $O(\log_2 n)$ time complexity for expensive operations such as INSERT, UPDATE, and DELETE. This is possible because they are *self-balancing*; they maintain a balance factor for each node, ensuring that the height difference between the left and right subtrees does not exceed 1.

## Implementation details

### `Record` class

All `Record` objects will consist of a string and an integer. You can think of it as a key-value pair.

### `AVLNode` class

Building blocks in AVL trees. It stores key-value pairs and maintains links to child nodes. The height attribute maintains the balance of the tree during insertions and deletions.

### `AVLTree` class

#### Private Member functions

- `height:` Returns the height of the given node.
- `balance:` Computes the balance factor of the given node, which is the difference in heights between the left and right subtrees.
- `rotateLeft` and `rotateRight`: Perform left and right rotations, respectively, to balance the AVL tree.
  *Hint: Don't forget to update the node heights after rotation.*

#### Public Member functions

- `AVLTree()`: Constructor that initializes an empty AVL tree by setting `root` to `nullptr`.
- `insert(const string& key, int value)`: Public interface to insert a new `record` with the given key-value pair into the AVL tree.
- `void deleteNode(const string& key, int value)`: Public interface to remove a `record` with the specified key-value pair from the AVL tree.
- `Record* search(const string& key, int value)`: Public interface to search for a `record` with the specified key-value pair from the AVL tree.

### `IndexedDatabase` Class

The `IndexedDatabase` class manages a collection of records using an AVL tree (`AVLTree`) for efficient insertion, searching, and manipulation operations.

#### Private Members:

- `AVLTree index`: An instance of the `AVLTree` class that serves as the primary data structure for indexing and organizing records based on their keys.

#### Public Member Functions:

- `void insert(Record* record)`: Inserts a new `record` into the database. The record is passed as a pointer (`Record*`) and is inserted into the AVL tree (`index`).
  *Hints:*
  *Insertion Conditionals: Determine where to insert the new record based on its value compared to the current node's record's value.*
  *Update Node Height: After inserting the record into the correct position, update the height of the current node to reflect any changes caused by the insertion.*
  *Check Balance Factor: Calculate the balance factor of the node to determine if rotations are necessary*
  *Rotation Cases: Recall various cases for balancing an AVL tree. (Left-Left, Left-Right, Right-Left, Right-Right. Refer to algorithm 1)*
- `Record* search(const string& key, int value)`: Searches for a `record` with the specified key-value pair in the database. Returns a pointer to the found `record`. If the `record` with the given key-value pair is not found, return a new `record` object with `key=""` and `value=0`
- `void deleteRecord(const string& key, int value)`: Deletes a `record` from the database based on the provided key-value pair. This operation removes the corresponding `record` from the AVL tree (`index`), maintaining the AVL tree's balance after deletion.
  *Hint: Like the `insert` function, update node heights, check balance factors, and rotate accordingly after deleting a node.*
- `vector<Record*> rangeQuery(int start, int end)`: Retrieves records whose keys fall within the specified range $[start, end]$. Returns a vector of `record`s that match the criteria.
  *Hint: Consider using an inorder traversal with a conditional check at each node to determine if its key falls within the specified range. Think recursively.*
- `int countRecords()`: Returns the total number of records currently stored in the database. This function provides a count of records by traversing the AVL tree (`index`) and counting the nodes.

## AVL tree rotation logic

**Algorithm 1** AVL Tree Rotation Logic

**if** $bal > 1$ **and** $record.value < node.left.record.value$ **then**
    **return** rotateRight($node$)
**end if**
**if** $bal < -1$ **and** $record.value > node.right.record.value$ **then**
    **return** rotateLeft($node$)
**end if**
**if** $bal > 1$ **and** $record.value > node.left.record.value$ **then**
    $node.left \leftarrow$ rotateLeft($node.left$)
    **return** rotateRight($node$)
**end if**
**if** $bal < -1$ **and** $record.value < node.right.record.value$ **then**
    $node.right \leftarrow$ rotateRight($node.right$)
    **return** rotateLeft($node$)
**end if**

## Deliverables

- `AVL_Database.hpp`
- `AVL_Database.cpp`
- Makefile

and any other files necessary to successfully compile your code. No need to include the `db_driver.cpp` file in your submission.

## Suggestions/Assumptions

- You may not use any other header files besides what is already included.
- You can add any number of helper functions and modify the provided .hpp and .cpp files according to your needs.
- All the record objects will have different values i.e. the attribute `record->key` is unique for every `record` object.

## Rubric

14 test cases: 14x6.5= 91
Comments/Documentations :9

---

Total:100
**Any code that does not compile will receive a zero for this project.**