Programming Assignment 2

PART 1: Version Control System v2

You are already familiar with a Version Control System from PA-1. In this section, we shall explore two more major functionalities of a VCS: branch and merge. We use a General Tree Structure (each node can have arbitrary number of children) to implement a VCS. However, for the sake of simplicity, we assume that a parent can have no more than 9 children (branches in our case). Each node in the tree basically contains a file along with all of its versions in a vector. Each commit on a node stores a newer version of the file in the vector *within* the node.

Whenever branch is called on a node `n`, the latest version of the file is used to create a new branch i.e. a child node of `n`.

Whenever merge is called from a source node `src` to a destination node `dst`, all the data from the latest version in `src` is appended at the end of data in the latest version of `dst`.

In the tree.h file, we have provided you with three structs:

- VCS_Node:
  - VCS_node is the individual node in your version control system. Data string is for any data in the file such as 'Changed Dijkstra algorithm with Bellman-Ford algorithm', whereas the time_stamp contains the information of when certain change was committed, in your case treat time_stamp as version of your file i.e., start with 0 and increase it discretely with every commit (0,1,2...).
- TreeItem:
  - It's an Individual node in your tree. It consists of a unique identifier (id), a vector of children nodes and a container containing different versions of a file. The string latest_data contains the data from the VCS_Node with the highest time_stamp.
- Merge_State:
  - This is used to store the source node id and the destination node id of the merge call for maintaining the history log.

Note: The Id of the TreeItem follows a strict convention. The root node has an Id of 1. The branches from this node with have id's 11, 12, 13, 14, 15...,19. Similarly, the branches from node of id 13 will have id's 131, 132, 133, 134, ..., 139. Since, our assumption states that we can have a maximum of 9 children, this convention will work for us.

Note: For the sake of simplicity, you don't need to write data to a text file in this part. You can just store it in the data string in the Node provided to you

Member Functions:

- `Tree (string data):`
  - Simple base constructor, data contains the data of original file (master file) you will be keeping track of.
- `~Tree ():`
  - Deallocate Memory
- `void branch (int id):`
  - This function is the core of this VCS implementation. When a branch is called on a given node, you need to create a child node with an appropriate ID. The child branch will have the same data as the latest version of the parent node.
- `void merge (int src_id, int dst_id):`
  - When a merge is called from a source node to a destination node, all the data from the latest version of source node is appended back to data of the latest version destination node. Don't add any spaces between the two strings while concatenation. Use the following convention for appending data:
  - dst_data = dst_data + src_data
- `void commit (int id, string new data):`
  - As implemented in PA-1 as well, when a commit is called on a given id, you have to store a new version of the file with the addition of the new_data that comes with the commit in the container along with its corresponding time_stamp and update the latest_data item in Tree node.
- `TreeItem* get_node (int id):`
  - This returns the node given the ID and returns NULL if the node with the given ID does not exist. This function should be able to search for the given node in steps equal to the number of the digits in the id. You can not scan the entire tree to find the node.
- `string get_file (int id, int t_stamp):`
  - Returns the data in the given version of a given id. Returns an empty string if the file does not exist
- `vector<merge_state> merge_history ():`
  - Returns a vector of the history of all the merges.
- `void level_traversal ():`
  - Performs and displays the level order traversal of the tree. Prints out the (id, latest_data) pair of each node.

PART 2: Binary Search Tree

In this part you'll implement a barebone BST. Please refer to the file **bst.h**, which provides the necessary class definitions and function declarations. You'll implement the following functions for BST:

Note: Some of the functions in **bst.h** are relevant to AVL Tree only (Part 3). For this part, you can simply comment them out.

Member Functions:

- `BST();`
  - initializes an empty BST
- `~BST();`
  - frees the memory occupied by each node in the tree
- `void insert(string val, T k);`
  - inserts the given key-value pair into the tree
- `node<T>* search(T k);`
  - takes key k as input and returns pointer to the node that has the matching key. Returns `NULL` if k does not exist in the tree
- `void delete_node(T k);`
  - deletes node with the given key k
- `node<T>* getRoot();`
  - returns root of the tree
- `node<T>* findmin(node<T> *p);`
  - finds node with the minimum key in the tree rooted at p
- `node<T>* removemin(node<T> *p);`
  - deletes node with the minimum key in the tree rooted at p
- `node<T>* insertHelper(string value,T k, node<T> *p);`
  - helper function need by `insert` for recursion
- `node<T>* remove(node<T> *p,T k);`
  - helper function need by `delete_node` for recursion
- `int height (node<T>* p);`
  - returns the height of the tree rooted at p

PART 3: AVL Tree

In this part, you'll implement an AVL tree. If you have done Part 2, then you just have to add balancing to BST in this part. In particular, you'll implement the following additional functions:

Member Functions:

- `node<T>* rotateleft(node<T>* p);`
  - p is root of the tree before rotation, the function should return pointer to the new root of the tree, around which the subtrees were rotated. Remember to fix the heights of the subtrees.
- `node<T>* rotateright(node<T>* p);`
  - same as above
- `node<T>* balance(node<T>* p);`
  - call appropriate rotations depending upon the four cases discussed in the class.
- `int balanceFactor(node<T>* p);`
  - calculates and returns the balance factor of the node p based upon the heights of left and right subtrees.
- `void fixHeight(node<T>* p);`
  - calculates new height of the tree rooted at p

PART 4: Dictionary & Word Search

In this part, you'll implement a dictionary using BST & AVL trees and compare the time taken by both to search words. We have provided you a file **words.txt** containing dictionary words in random order. All the words are in lower-case. You have to load (insert) these words in BST & AVL trees. The words themselves will act as keys. You'll have to implement the following functions:

Note: The skeleton code has been provided to you in the folder part-4. You may replace **bst.h** and **bst.cpp** in this folder with your implementation of BST/AVL trees.

Member Functions:

- Dictionary();
  - creates an empty (i.e. without words) Dictionary object
- ~Dictionary();
  - frees the memory occupied by nodes containing words
- void initialize(string wordsFile);
  - populates words read from `wordsFile` into `wordsTree`, `wordsFile` is the name of the file
- `node<string>* findWord(string word);`
  - corresponds to `node<T>* search(T k)`
- `bool deleteWord(string word);`
  - corresponds to `void delete_node(T k)`
- `bool editWord(string oldWord, string newWord);`
  - changes the key, Therefore, the tree may need to be adjusted to maintain BST property
- `bool insertWord(string word);`
  - corresponds to `void insert(string val, T k)`

Comparison

Select 26 words such that each one starts from a distinct alphabet. For report, please follow the following table format:

#	Word	Time (sec) taken by BST (t1)	Time (sec) taken by AVL (t2)	Difference dt = t1 - t2	Percentage decrease dt ÷ t1 × 100