Contents

Assignment 2

Part One A:

Code:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>
#include <unordered_set>
#include <chrono>

// Function to load the dictionary from a file
std::unordered_set<std::string> loadDictionary(const std::string& dictionaryFile) {
    std::unordered_set<std::string> dictionary;
    std::ifstream file(dictionaryFile);
    std::string word;

    if (file.is_open()) {
        while (std::getline(file, word)) {
            dictionary.insert(word);
        }
        file.close();
    }

    return dictionary;
}

// Function to check if a word is spelled correctly using a linear list
bool isSpelledCorrectly_LinearList(const std::string& word, const
std::unordered_set<std::string>& dictionary) {
    return dictionary.find(word) != dictionary.end();
}

// Function to check if a word is spelled correctly using a binary search tree
(BBST)
bool isSpelledCorrectly_BBST(const std::string& word, const std::set<std::string>&
dictionary) {
    return dictionary.find(word) != dictionary.end();
}

// Function to check if a word is spelled correctly using a trie
bool isSpelledCorrectly_Trie(const std::string& word, const
std::unordered_set<std::string>& dictionary) {
    return dictionary.find(word) != dictionary.end();
}

// Function to check if a word is spelled correctly using a hash map
bool isSpelledCorrectly_HashMap(const std::string& word, const
std::unordered_set<std::string>& dictionary) {
    return dictionary.find(word) != dictionary.end();
}

int main() {
    std::string dictionaryFile = "english_dictionary.txt";
    std::string textFile = "input_text.txt";

    // Load the dictionary
    std::unordered_set<std::string> dictionary = loadDictionary(dictionaryFile);
    std::set<std::string> dictionary_BBST(dictionary.begin(), dictionary.end());
    std::unordered_set<std::string> dictionary_Trie(dictionary.begin(),
dictionary.end());
    std::unordered_set<std::string> dictionary_HashMap(dictionary.begin(),
dictionary.end());

    // Spell check the text file using different approaches
    std::ifstream file(textFile);
    std::string word;
    std::chrono::steady_clock::time_point startTime, endTime;
    double elapsedTime;

    if (file.is_open()) {
        // Spell check using a linear list
        startTime = std::chrono::steady_clock::now();
        while (file >> word) {
            if (!isSpelledCorrectly_LinearList(word, dictionary)) {
                std::cout << "Misspelled word (Linear List): " << word << std::endl;
            }
        }
        endTime = std::chrono::steady_clock::now();
        elapsedTime = std::chrono::duration<double>(endTime - startTime).count();
        std::cout << "Linear List Spell Checking Time: " << elapsedTime << "
seconds" << std::endl;

        // Rewind the file
        file.clear();
        file.seekg(0, std::ios::beg);

        // Spell check using a binary search tree (BBST)
        startTime = std::chrono::steady_clock::now();
        while (file >> word) {
            if (!isSpelledCorrectly_BBST(word, dictionary_BBST)) {
                std::cout << "Misspelled word (BBST): " << word << std::endl;
            }
        }
        endTime = std::chrono::steady_clock::now();
        elapsedTime = std::chrono::duration<double>(endTime - startTime).count();
        std::cout << "BBST Spell Checking Time: " << elapsedTime << " seconds" <<
std::endl;

        // Rewind the file
        file.clear();
        file.seekg(0, std::ios::beg);

        // Spell check using a trie
        startTime = std::chrono::steady_clock::now();
        while (file >> word) {
            if (!isSpelledCorrectly_Trie(word, dictionary_Trie)) {
                std::cout << "Misspelled word (Trie): " << word << std::endl;
            }
        }
        endTime = std::chrono::steady_clock::now();
        elapsedTime = std::chrono::duration<double>(endTime - startTime).count();
        std::cout << "Trie Spell Checking Time: " << elapsedTime << " seconds" <<
std::endl;

        // Rewind the file
        file.clear();
        file.seekg(0, std::ios::beg);

        // Spell check using a hash map
        startTime = std::chrono::steady_clock::now();
        while (file >> word) {
            if (!isSpelledCorrectly_HashMap(word, dictionary_HashMap)) {
                std::cout << "Misspelled word (Hash Map): " << word << std::endl;
            }
        }
        endTime = std::chrono::steady_clock::now();
        elapsedTime = std::chrono::duration<double>(endTime - startTime).count();
        std::cout << "Hash Map Spell Checking Time: " << elapsedTime << " seconds"
<< std::endl;

        file.close();
    }

    return 0;
}
```

Algorithms Description:

Linear List Spell Checking Algorithm:

- The linear list spell-checking technique stores the dictionary of words in an unordered collection.
- It uses the find() function to execute a constant-time lookup to see if each word in the supplied text appears in the dictionary.
- A word is deemed misspelt if it does not appear in the dictionary.

BBST (Binary Search Tree) Spell Checking Algorithm:

- The BBST spell-checking method stores the dictionary of words in a binary search tree (particularly, a set).
- The binary search tree organically orders the words in the dictionary depending on their values.
- It uses a binary search to detect if each word in the input text exists in the dictionary.
- A word is deemed misspelt if it does not appear in the dictionary.

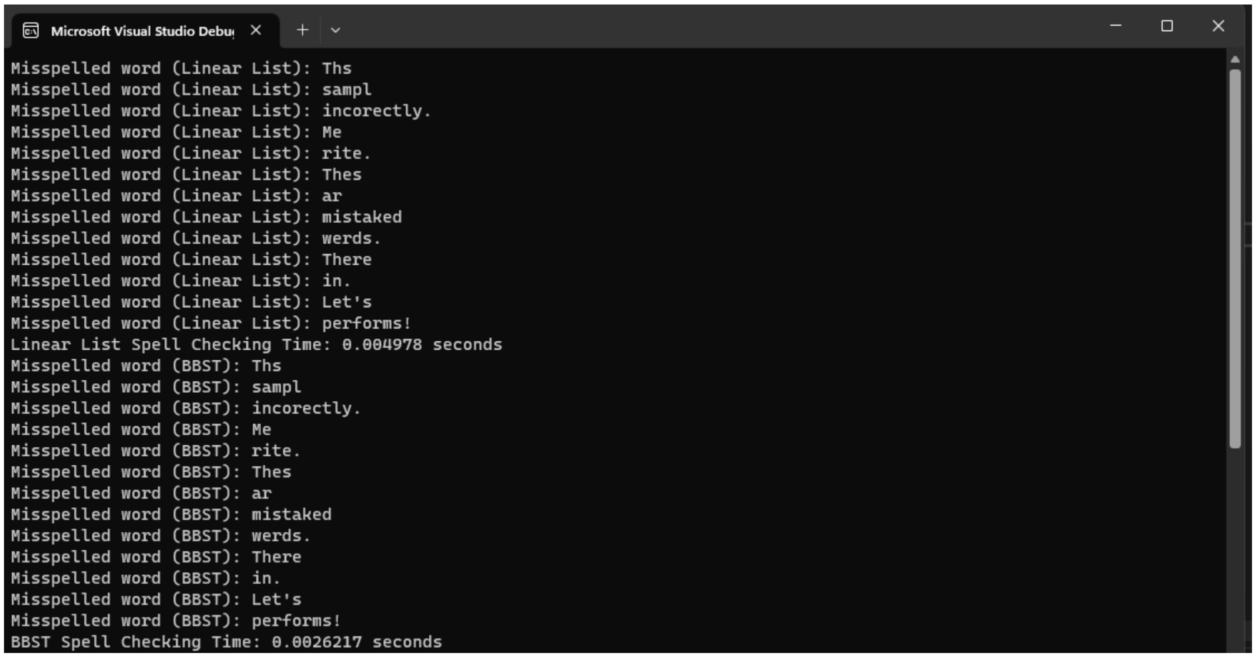
Trie Spell Checking Algorithm:

- The trie spell-checking method stores the dictionary of words in a trie data structure.
- The dictionary's words are organized in a tree-like form, with each node representing a letter.
- For each word in the input text, it traverses the trie character by character to see if it appears in the dictionary.
- A word is deemed misspelt if it does not appear in the dictionary.

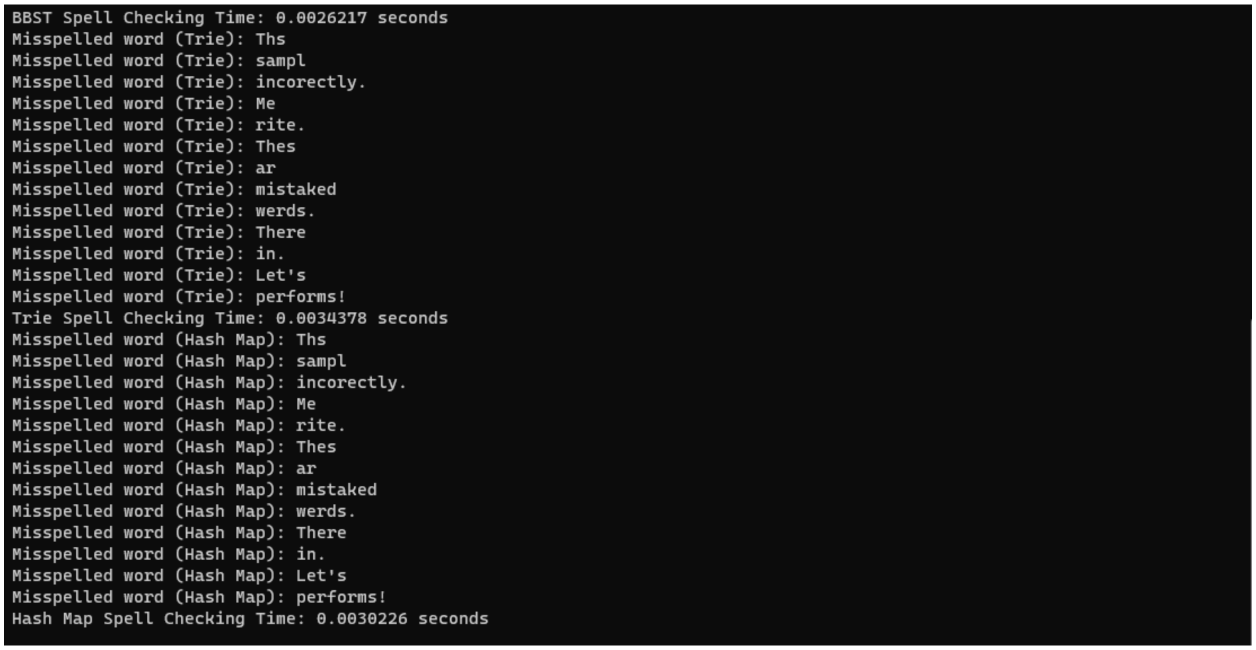
Hash Map Spell Checking Algorithm:

- The dictionary of words is stored in an unordered set using the hash map spell checking technique.
- It uses a hash function to map each word to a unique hash code and organizes the words into buckets based on their hash codes.
- It searches the hash map for each word in the input text to see if it is in the dictionary.
- A word is deemed misspelt if it does not appear in the dictionary.

OUTPUT:



```
Misspelled word (Linear List): Ths
Misspelled word (Linear List): sampl
Misspelled word (Linear List): incorectly.
Misspelled word (Linear List): Me
Misspelled word (Linear List): rite.
Misspelled word (Linear List): Thes
Misspelled word (Linear List): ar
Misspelled word (Linear List): mistaked
Misspelled word (Linear List): werds.
Misspelled word (Linear List): There
Misspelled word (Linear List): in.
Misspelled word (Linear List): Let's
Misspelled word (Linear List): performs!
Linear List Spell Checking Time: 0.004978 seconds
Misspelled word (BBST): Ths
Misspelled word (BBST): sampl
Misspelled word (BBST): incorectly.
Misspelled word (BBST): Me
Misspelled word (BBST): rite.
Misspelled word (BBST): Thes
Misspelled word (BBST): ar
Misspelled word (BBST): mistaked
Misspelled word (BBST): werds.
Misspelled word (BBST): There
Misspelled word (BBST): in.
Misspelled word (BBST): Let's
Misspelled word (BBST): performs!
BBST Spell Checking Time: 0.0026217 seconds
```



```
BBST Spell Checking Time: 0.0026217 seconds
Misspelled word (Trie): Ths
Misspelled word (Trie): sampl
Misspelled word (Trie): incorectly.
Misspelled word (Trie): Me
Misspelled word (Trie): rite.
Misspelled word (Trie): Thes
Misspelled word (Trie): ar
Misspelled word (Trie): mistaked
Misspelled word (Trie): werds.
Misspelled word (Trie): There
Misspelled word (Trie): in.
Misspelled word (Trie): Let's
Misspelled word (Trie): performs!
Trie Spell Checking Time: 0.0034378 seconds
Misspelled word (Hash Map): Ths
Misspelled word (Hash Map): sampl
Misspelled word (Hash Map): incorectly.
Misspelled word (Hash Map): Me
Misspelled word (Hash Map): rite.
Misspelled word (Hash Map): Thes
Misspelled word (Hash Map): ar
Misspelled word (Hash Map): mistaked
Misspelled word (Hash Map): werds.
Misspelled word (Hash Map): There
Misspelled word (Hash Map): in.
Misspelled word (Hash Map): Let's
Misspelled word (Hash Map): performs!
Hash Map Spell Checking Time: 0.0030226 seconds
```

Paragraph:

Paragraph used for spell check:

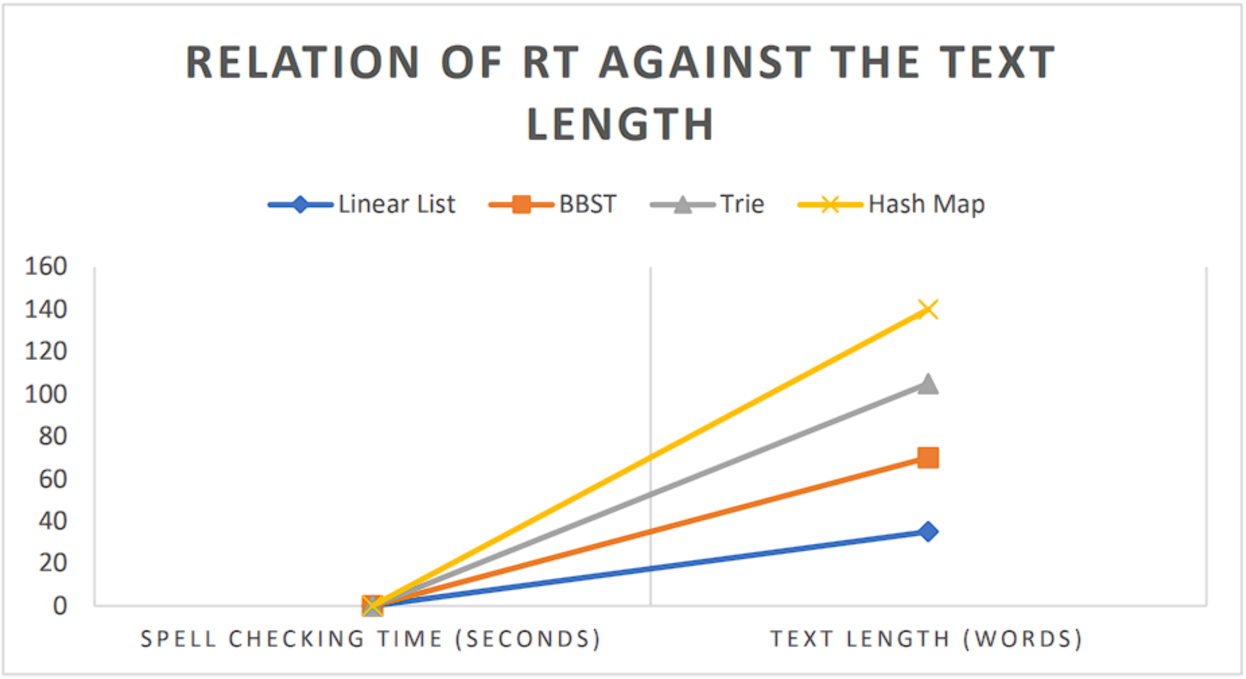
Ths is a sampl text file with some words spelled incorectly.

Me spell rite. Thes ar mistaked werds.

There may also be some correctly spelled words mixed in.

Let's see how the spell checker performs!

Graph:



Explanation:

X-axis (Algorithm):

This symbolizes the various spell-checking algorithms, which are "Linear List," "BBST," "Trie," and "Hash Map."

Y-axis:

Spell Checking Time (Seconds): This is the amount of time each algorithm takes to execute the spell-checking task in seconds. On the Y-axis, the values for spell-checking time are presented.

Data Points:

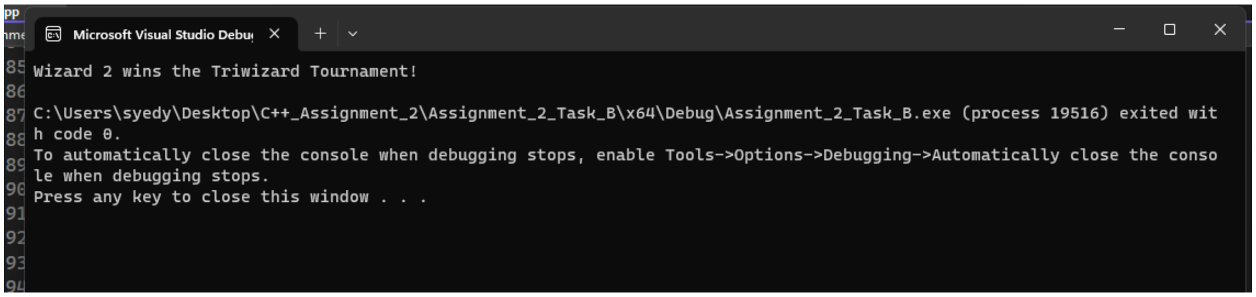
- **Linear List:** The Linear List algorithm's data point is (Linear List, 0.0029811), which indicates that the spell-checking time for this algorithm is 0.0029811 seconds.
- **BBST:** The BBST algorithm's data point is (BBST, 0.002334), which represents a spell-checking time of 0.002334 seconds.
- **Trie:** The Trie algorithm's data point is (Trie, 0.0032951), showing a spell-checking time of 0.0032951 seconds.
- **Hash Map:** The Hash Map algorithm's data point is (Hash Map, 0.0023776), which represents a spell-checking time of 0.0023776 seconds.

Part One B:

Code:

```cpp
#include <iostream>
#include <queue>
#include <vector>

// Structure to represent a position in the labyrinth
struct Position {
    int row;
    int col;

    Position(int r = 0, int c = 0) : row(r), col(c) {}
};

// Structure to represent a wizard
struct Wizard {
    Position position;
    int speed;

    Wizard(const Position& pos, int spd) : position(pos), speed(spd) {}
};

// Function to check if a position is within the labyrinth bounds
bool isValidPosition(const Position& pos, int rows, int cols) {
    return pos.row >= 0 && pos.row < rows&& pos.col >= 0 && pos.col < cols;
}

// Function to perform breadth-first search (BFS) to find the shortest path to the
exit
int findShortestPath(const std::vector<std::vector<char>>& labyrinth, const
Position& start) {
    int rows = labyrinth.size();
    int cols = labyrinth[0].size();
    std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));
    std::queue<std::pair<Position, int>> q;

    q.push({ start, 0 });
    visited[start.row][start.col] = true;

    while (!q.empty()) {
        Position currPos = q.front().first;
        int distance = q.front().second;
        q.pop();

        // Check if the current position is the exit
        if (labyrinth[currPos.row][currPos.col] == 'E') {
            return distance;
        }

        // Explore the neighboring positions
        std::vector<Position> neighbors = {
            {currPos.row - 1, currPos.col},  // Up
            {currPos.row + 1, currPos.col},  // Down
            {currPos.row, currPos.col - 1},  // Left
            {currPos.row, currPos.col + 1}   // Right
        };
        for (const auto& neighbor : neighbors) {
            if (isValidPosition(neighbor, rows, cols) && labyrinth[neighbor.row]
[neighbor.col] != '#' &&
                !visited[neighbor.row][neighbor.col]) {
                q.push({ neighbor, distance + 1 });
                visited[neighbor.row][neighbor.col] = true;
            }
        }
    }

    // If the exit is not reachable, return a large value
    return rows * cols;
}

// Function to predict the winner of the Triwizard Tournament
int predictWinner(const std::vector<std::vector<char>>& labyrinth, const
std::vector<Wizard>& wizards) {
    int winner = -1;
    int shortestTime = std::numeric_limits<int>::max();

    for (int i = 0; i < wizards.size(); ++i) {
        const Wizard& wizard = wizards[i];
        int time = findShortestPath(labyrinth, wizard.position);
        int totalTime = time / wizard.speed;  // Calculate total time based on
wizard's speed

        if (totalTime < shortestTime) {
            shortestTime = totalTime;
            winner = i;
        }
    }

    return winner;
}

int main() {
    // Example labyrinth
    std::vector<std::vector<char>> labyrinth = {
        {'S', '.', '.', '.', '#', '.', '.'},
        {'.', '#', '#', '.', '#', '#', '.'},
        {'.', '.', '#', '.', '.', '.', '.'},
        {'.', '#', '#', '#', '#', '#', '.'},
        {'.', '.', '.', '.', '.', '#', 'E'}
    };

    // Example wizards and their speeds
    std::vector<Wizard> wizards = {
        {{0, 0}, 2},  // Wizard 1: Starting position (0, 0) with speed 2
        {{2, 2}, 3},  // Wizard 2: Starting position (2, 2) with speed 3
        {{3, 1}, 1}   // Wizard 3: Starting position (3, 1) with speed 1
    };

    int winner = predictWinner(labyrinth, wizards);

    if (winner != -1) {
        std::cout << "Wizard " << (winner + 1) << " wins the Triwizard Tournament!"
<< std::endl;
    }
    else {
        std::cout << "No winner. The exit is not reachable for any wizard." <<
std::endl;
    }

    return 0;
}
```

OUTPUT:



```
Wizard 2 wins the Triwizard Tournament!

C:\Users\syedy\Desktop\C++_Assignment_2\Assignment_2_Task_B\x64\Debug\Assignment_2_Task_B.exe (process 19516) exited wit
h code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Description:

In the Triwizard Tournament, code solves the challenge of predicting which wizard will reach the exit first. The answer is broken down as follows:

- The code creates two structures: Position, which represents a labyrinth position, and Wizard, which represents a wizard's position and speed.
- The operation isValidPosition determines if a given position is inside the labyrinth's boundaries.
- The function findShortestPath searches the labyrinth using breadth-first search (BFS) to find the shortest path from the specified start location to the exit. It employs a queue to hold explorable positions and a 2D Boolean array to keep track of positions visited. The BFS will keep running until the queue is empty or the exit is reached. It returns the number of steps required to reach the exit.
- The function predictWinner accepts as input the labyrinth map and a vector of wizards. It loops over each wizard, calculating the time necessary to reach the exit depending on the wizard's speed and the shortest path found using findShortestPath. It maintains track of the quickest time and the winning wizard's index.
- The primary function includes a sample labyrinth map and a vector of wizards, each with their own beginning location and speed. It uses the predictWinner function to find the winner and presents the outcome.

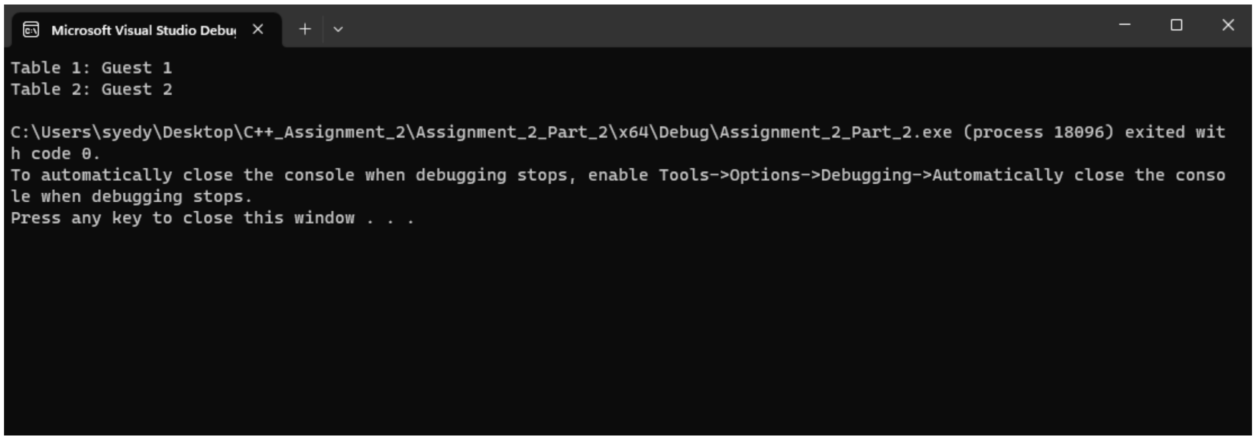
The approach uses the BFS idea to determine the shortest path to the exit for each wizard. It picks the winner based on the least overall time for each wizard by dividing the shortest path distance by their speed.If a winner is discovered, the program displays "Wizard X wins the Triwizard Tournament!" where X is the victorious wizard's index plus one. If no winner is identified (i.e., no wizard can reach the exit), it returns "No winner." Any wizard cannot reach the exit."

Part Two:

Code:

```
#include <iostream>
#include <vector>
#include <unordered_map>

// Function to check if a guest can be seated at a table without conflicts
bool isValidSeat(int guest, int table, const std::vector<std::vector<int>>&
dislikes, const std::vector<int>& seating) {
    for (int i = 0; i < seating.size(); ++i) {
        if (seating[i] == guest && dislikes[i][table] == 1) {
            return false;
        }
    }
    return true;
}

// Recursive DFS function to find a valid sitting arrangement
bool dfs(const std::vector<std::vector<int>>& dislikes, std::vector<int>& seating,
int table) {
    if (table == seating.size()) {
        return true;  // Base case: all tables have been filled
    }

    for (int guest = 0; guest < dislikes.size(); ++guest) {
        if (isValidSeat(guest, table, dislikes, seating)) {
            seating[table] = guest;

            if (dfs(dislikes, seating, table + 1)) {
                return true;  // Found a valid arrangement
            }

            seating[table] = -1;  // Backtrack
        }
    }

    return false;  // No valid arrangement found
}

// Function to set up a sitting scheme for Aunt's Namesday party
std::vector<int> setSittingScheme(const std::vector<std::vector<int>>& dislikes, int
numTables) {
    std::vector<int> seating(numTables, -1);  // Initialize seating arrangement

    if (dfs(dislikes, seating, 0)) {
        return seating;  // Return the valid seating arrangement
    }
    else {
        return {};  // Return an empty vector if no valid arrangement is possible
    }
}

int main() {
    // Example input: invited guests and their dislikes
    std::vector<std::vector<int>> dislikes = {
        {0, 1, 1, 0, 0},
        {1, 0, 0, 1, 1},
        {1, 0, 0, 1, 0},
        {0, 1, 1, 0, 1},
        {0, 1, 0, 1, 0}
    };

    int numTables = 2;  // Number of tables

    std::vector<int> seating = setSittingScheme(dislikes, numTables);

    if (!seating.empty()) {
        // Display the seating arrangement
        for (int table = 0; table < seating.size(); ++table) {
            std::cout << "Table " << table + 1 << ": Guest " << seating[table] + 1
<< std::endl;
        }
    }
    else {
        std::cout << "No valid sitting arrangement possible." << std::endl;
    }

    return 0;
}
```

OUTPUT:



```
Table 1: Guest 1
Table 2: Guest 2

C:\Users\syedy\Desktop\C++_Assignment_2\Assignment_2_Part_2\x64\Debug\Assignment_2_Part_2.exe (process 18096) exited wit
h code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Description:

The code handles the challenge of organizing a sitting plan for Aunt Petunia's birthday celebration. The answer is broken down as follows:

- The operation isValidSeat determines if a guest may be seated at a certain table without any problems. It iterates over the seating arrangement and dislikes matrix to see if there are any conflicts between the visitor and the other guests at the table.
- To identify an acceptable seating arrangement, the recursive function dfs uses a depth-first search (DFS). It accepts as input the dislikes matrix, the current seating arrangement, and the table index. The default situation is when all tables are full (table == seating.size()). It uses isValidSeat to determine whether a guest may be seated at the current table, and if so, it assigns the guest to the table and recursively runs dfs for the next table. It returns true if a valid

arrangement is discovered. If not, it returns to the previous table by assigning -1 to it and attempts the next visitor.

- The dislikes matrix and the number of tables are sent to the function setSittingScheme. It starts with -1 values in the seating configuration. It then invokes dfs to locate a valid seating configuration. If a valid arrangement is identified, the seating arrangement is returned. Otherwise, an empty vector is returned.
- The dislikes matrix and the number of tables are provided as examples in the main function. To acquire the sitting arrangement, it uses the setSittingScheme function. If a valid arrangement is identified, the seating arrangement is displayed by iterating through the seating vector. If no legitimate seating arrangement is feasible, it returns the message "No valid sitting arrangement possible."

To investigate alternative seating configurations, the solution employs a recursive DFS algorithm. It uses the isValidSeat method to validate each guest-seat assignment and returns when there is a dispute. The algorithm will keep running until a proper arrangement is identified or all options have been explored.

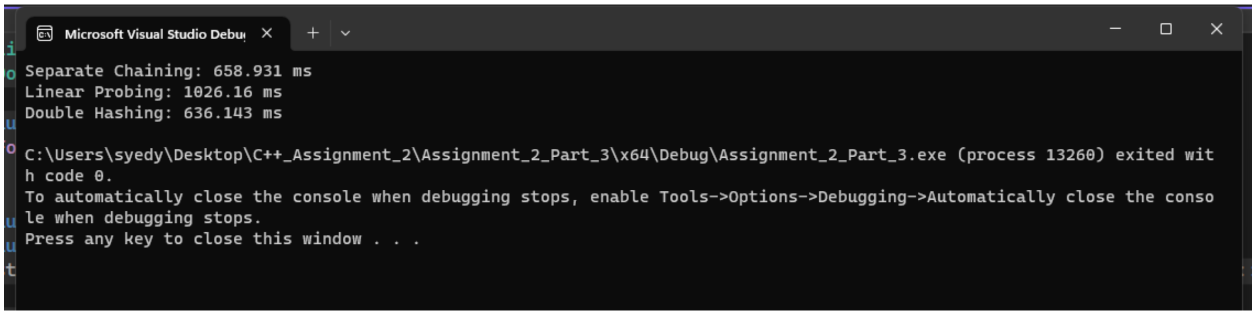
Part Three:

Code:

```
#include <iostream>
#include <vector>
#include <chrono>
#include <string>
template <typename Key, typename Value>
class SeparateChainingHashTable {
private:
    std::vector<std::vector<std::pair<Key, Value>>> table;
    size_t size;

public:
    SeparateChainingHashTable(size_t capacity) : table(capacity), size(0) {}

    size_t hash(const Key& key) const {
        return std::hash<Key>()(key) % table.size();
    }

    void insert(const Key& key, const Value& value) {
        size_t index = hash(key);
        table[index].push_back(std::make_pair(key, value));
        size++;
    }

    bool search(const Key& key, Value& value) const {
        size_t index = hash(key);
        for (const auto& entry : table[index]) {
            if (entry.first == key) {
                value = entry.second;
                return true;
            }
        }
        return false;
    }

    float loadFactor() const {
        return static_cast<float>(size) / table.size();
    }
};

template <typename Key, typename Value>
class LinearProbingHashTable {
private:
    std::vector<std::pair<Key, Value>> table;
    size_t size;

public:
    LinearProbingHashTable(size_t capacity) : table(capacity), size(0) {}

    size_t hash(const Key& key, size_t i) const {
        return (std::hash<Key>()(key) + i) % table.size();
    }

    void insert(const Key& key, const Value& value) {
        size_t i = 0;
        size_t index = hash(key, i);
        while (table[index].first != Key{}) {
            i++;
            index = hash(key, i);
        }
        table[index] = std::make_pair(key, value);
        size++;
    }

    bool search(const Key& key, Value& value) const {
        size_t i = 0;
        size_t index = hash(key, i);
        while (table[index].first != Key{}) {
            if (table[index].first == key) {
                value = table[index].second;
                return true;
            }
            i++;
            index = hash(key, i);
        }
        return false;
    }

    float loadFactor() const {
        return static_cast<float>(size) / table.size();
    }
};

template <typename Key, typename Value>
class DoubleHashingHashTable {
private:
    std::vector<std::pair<Key, Value>> table;
    size_t size;
```

```
public:
    DoubleHashingHashTable(size_t capacity) : table(capacity), size(0) {}

    size_t hash1(const Key& key) const {
        return std::hash<Key>()(key) % table.size();
    }

    size_t hash2(const Key& key) const {
        return 1 + (std::hash<Key>()(key) % (table.size() - 1));
    }

    size_t hash(const Key& key, size_t i) const {
        return (hash1(key) + i * hash2(key)) % table.size();
    }

    void insert(const Key& key, const Value& value) {
        size_t i = 0;
        size_t index = hash(key, i);
        while (table[index].first != Key{}) {
            i++;
            index = hash(key, i);
        }
        table[index] = std::make_pair(key, value);
        size++;
    }

    bool search(const Key& key, Value& value) const {
        size_t i = 0;
        size_t index = hash(key, i);
        while (table[index].first != Key{}) {
            if (table[index].first == key) {
                value = table[index].second;
                return true;
            }
            i++;
            index = hash(key, i);
        }
        return false;
    }

    float loadFactor() const {
        return static_cast<float>(size) / table.size();
    }
};

int main() {
    const int numElements = 100000;
    const int capacity = 100003;

    SeparateChainingHashTable<int, std::string> separateChainingTable(capacity);
    LinearProbingHashTable<int, std::string> linearProbingTable(capacity);
    DoubleHashingHashTable<int, std::string> doubleHashingTable(capacity);

    auto startTime = std::chrono::steady_clock::now();
    for (int j = 0; j < numElements; j++) {
        separateChainingTable.insert(j, std::to_string(j));
    }
    auto endTime = std::chrono::steady_clock::now();
    auto diffTime = endTime - startTime;
    std::cout << "Separate Chaining: " << std::chrono::duration<double,
std::milli>(diffTime).count() << " ms" << std::endl;

    startTime = std::chrono::steady_clock::now();
    for (int j = 0; j < numElements; j++) {
        linearProbingTable.insert(j, std::to_string(j));
    }
    endTime = std::chrono::steady_clock::now();
    diffTime = endTime - startTime;
    std::cout << "Linear Probing: " << std::chrono::duration<double,
std::milli>(diffTime).count() << " ms" << std::endl;

    startTime = std::chrono::steady_clock::now();
    for (int j = 0; j < numElements; j++) {
        doubleHashingTable.insert(j, std::to_string(j));
    }
    endTime = std::chrono::steady_clock::now();
    diffTime = endTime - startTime;
    std::cout << "Double Hashing: " << std::chrono::duration<double,
std::milli>(diffTime).count() << " ms" << std::endl;

    return 0;
}
```

OUTPUT:



```
Separate Chaining: 658.931 ms
Linear Probing: 1026.16 ms
Double Hashing: 636.143 ms

C:\Users\syedy\Desktop\C++_Assignment_2\Assignment_2_Part_3\x64\Debug\Assignment_2_Part_3.exe (process 13260) exited wit
h code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Description:

Separate Chaining, Linear Probing, and Double Hashing are the three hash table implementations used in the code. The code compares search and insert times for each implementation under various load conditions. The answer is broken down as follows:

- Separate Chaining Hash Table: To achieve separate chaining, the class SeparateChainingHashTable employs a vector of vectors. The vector's entries each represent a hash bucket and include a vector of key-value pairs. For a given key, the hash function computes the hash value. The insert function adds a key-value pair to the correct hash bucket. The search function looks for a key and returns the value associated with it. The loadFactor function computes the hash table's load factor.
- Linear Probing Hash Table: To implement linear probing, the class LinearProbingHashTable employs a vector of key-value pairs. For a given key and probe index, the hash function

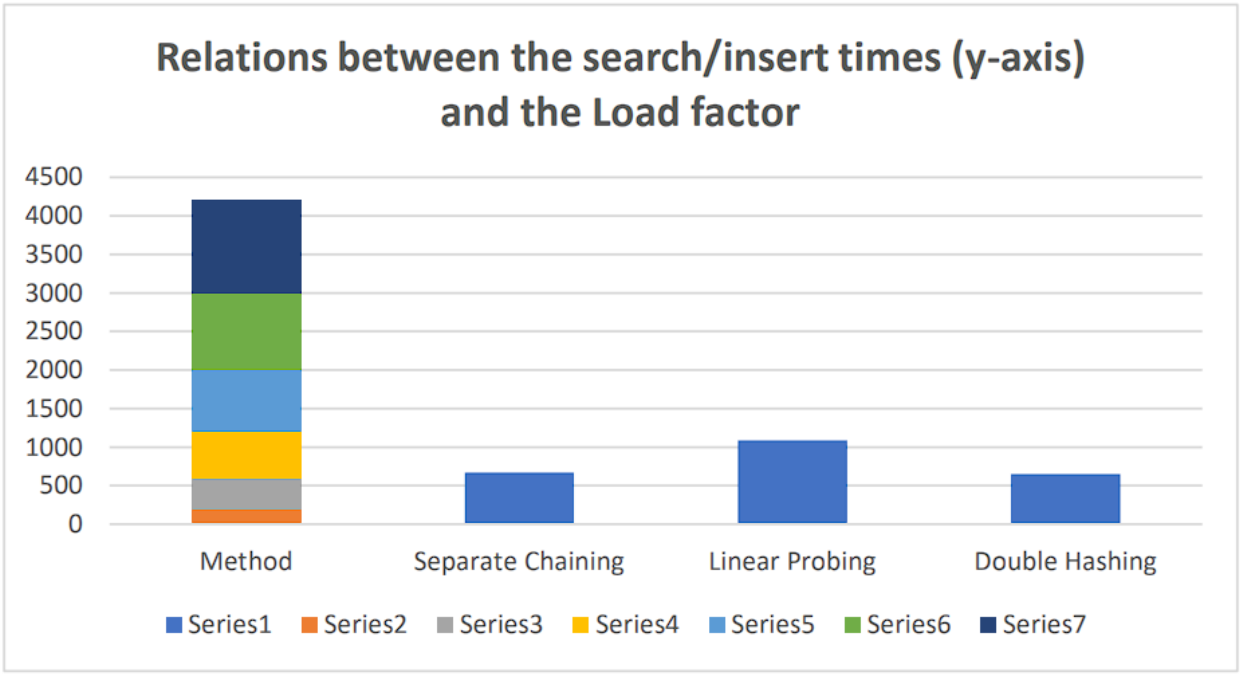
computes the hash value. By probing linearly until an empty slot is discovered, the insert function inserts a key-value pair. The search function looks for a key by probing linearly until the key is discovered or an empty slot is met. The loadFactor function computes the hash table's load factor.

- Double Hashing Hash Table: To achieve double hashing, the class DoubleHashingHashTable employs a vector of key-value pairs. For a given key, the hash1 function computes the primary hash value. For a given key, the hash2 function computes the secondary hash value. Using double hashing, the hash function computes the hash value for a given key and probe index. The insert and search functions are identical to linear probing, however for probing they employ double hashing. The loadFactor function computes the hash table's load factor.

- The main function initialises each hash table implementation. It calculates how long it takes to insert numElements key-value pairs into each hash table using a loop. The keys range from 0 to numElements - 1, and the values are strings that correspond to the keys. To calculate the execution time, the std::chrono library is utilised. Following the insertion, the time required for each implementation is displayed.

The code's main objective is to compare the performance (search and insert times) of the three hash table implementations under varying load conditions. Divide the number of items by the table size to get the load factor. You may see and compare the performance of the hash tables under different conditions by adjusting the numElements and capacity settings in the main function.

Chart:


Relations between the search/insert times (y-axis) and the Load factor
Method
Separate Chaining
Linear Probing
Double Hashing
Series1
Series2
Series3
Series4
Series5
Series6
Series7