

National University of Sciences & Technology School of Electrical Engineering and Computer Science

Department of Computing

**CS-250: Data Structures and Algorithms (3+1): BSCS-8AB Fall 2019**

**Semester Project**

**SUBMITTED TO: Dr. Qaiser Riaz SUBMITTED BY:**

1. **Name (CMS ID)\_ Danial Ahmad (249912)**
2. **Name (CMS ID)\_Faizan Qazi (263118)**
3. **Name (CMS ID)\_ Shariq Bin Rashid (242431)**

**Class (Section): BSCS-8B**

**Date of Submission: 22, December 2019**

**Problem Statement:**

Online companies require extra employee to reply their customers. Negotiating with them also take a lot of time. It cost them very much as they usually need a separate call center and employees for this purpose. The company need to hire double the employees as they need to be active 24/7.

**Solution:**

Here our chatbot will provide these companies with the perfect solution. It will reply customers in real time, help them find the best product. It will also convince them to buy our product and chat with them with a humanly manner. It will save them cost and time. This will increase their productivity and efficiency.

**Dataset:**

The dataset contains more than 60000 question and answers.

**Time complexity:**

The time complexity is O(1) in average case.

**Data Structures:**

The data structures used is

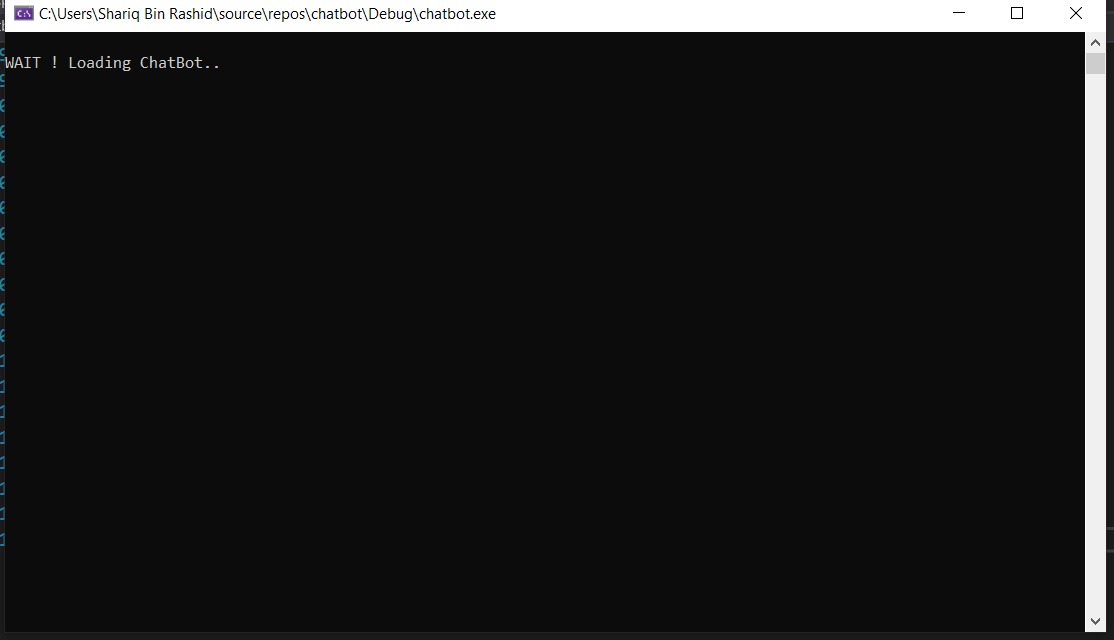
* *Dynamic Arrays*
* *Linked Lists*
* *Hash Tables*

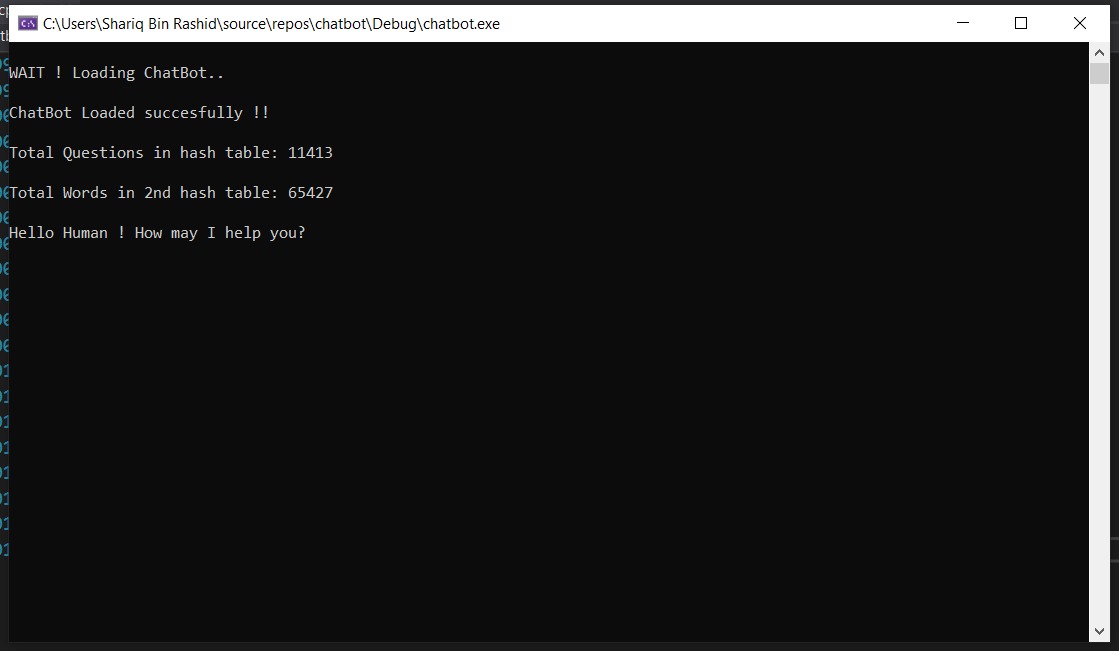
**Algorithms used:**

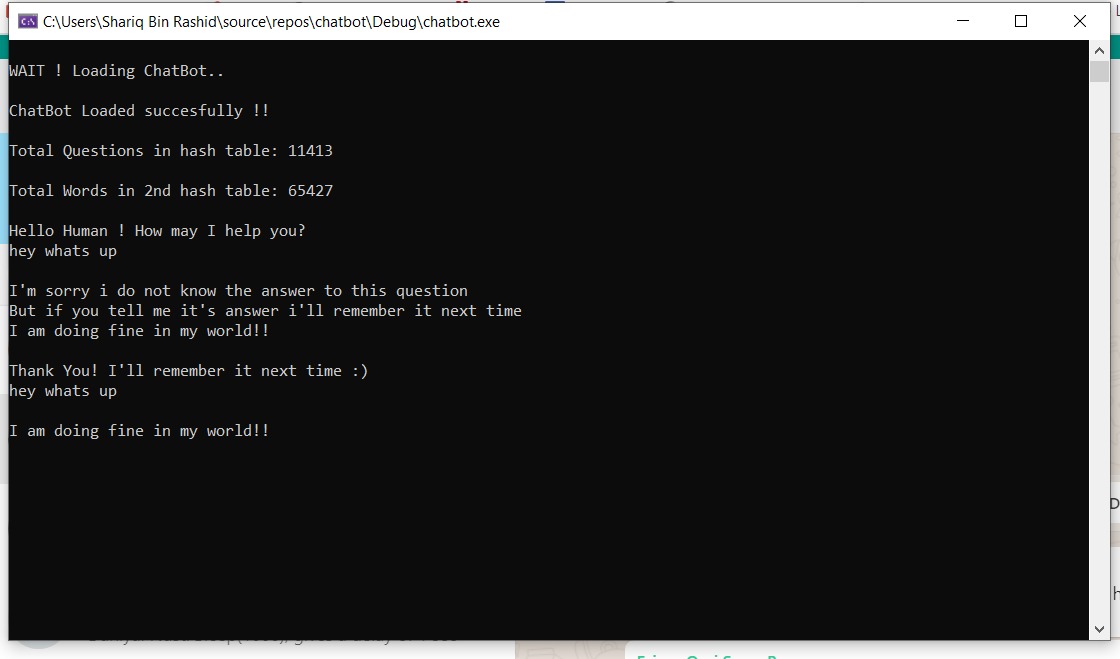
The algorithms used are

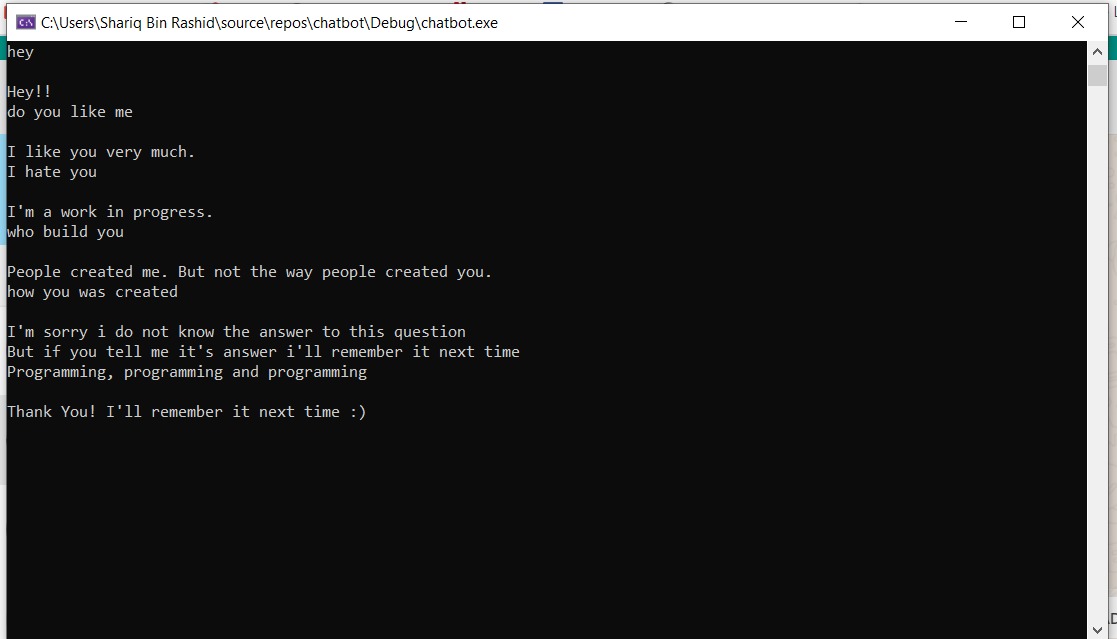
* *Hash functions*
* *String filtering*
* *Sorting*
* *Stemming*
* *lemmatization*

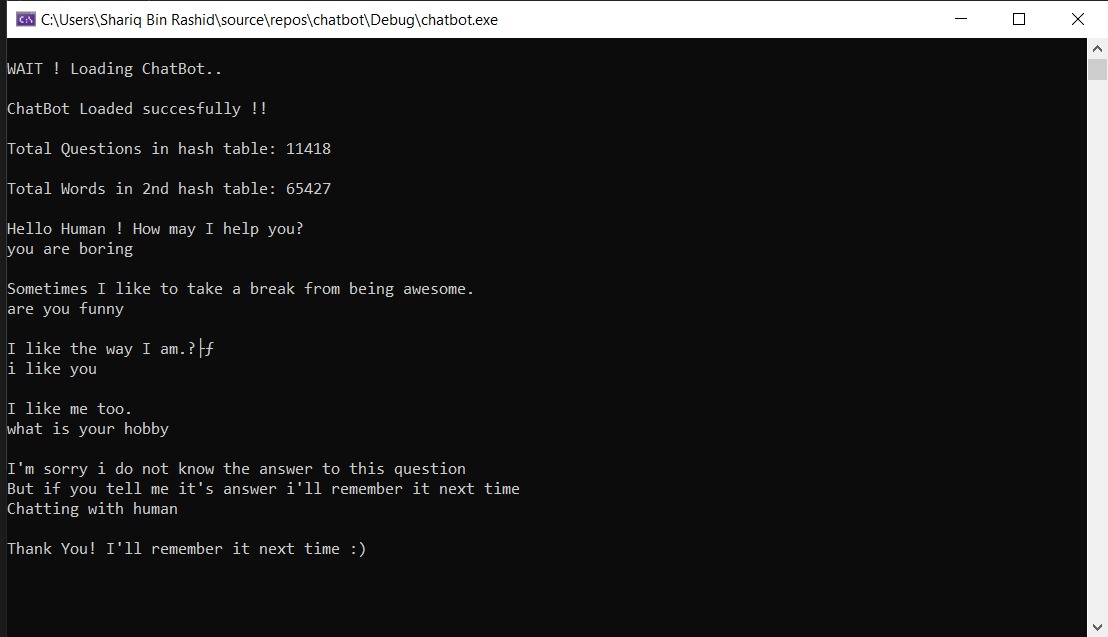
**Screenshots:**

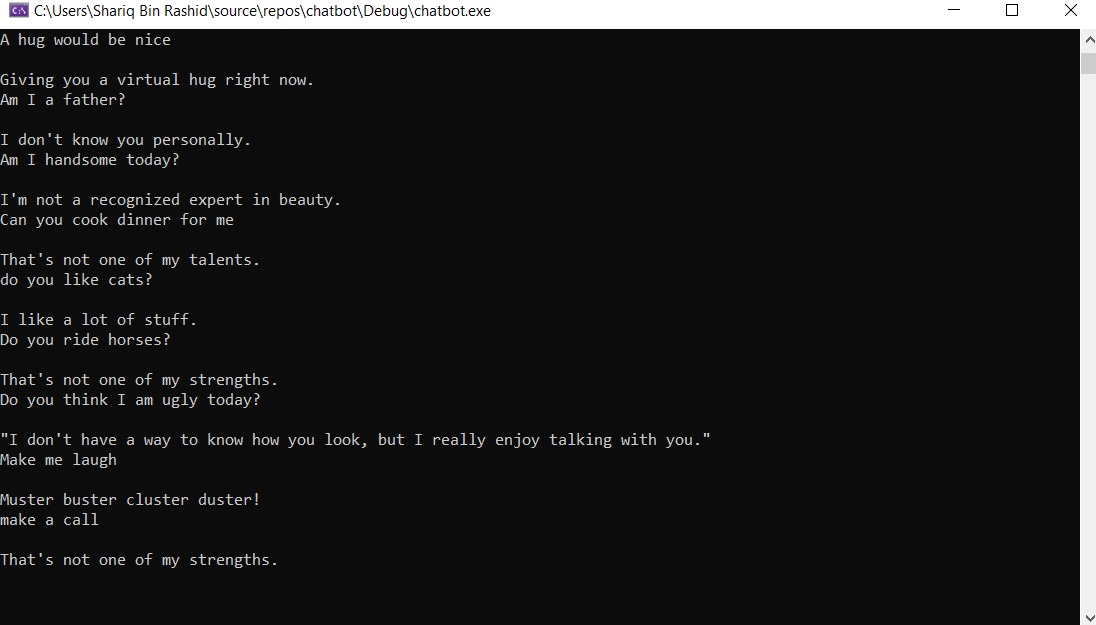


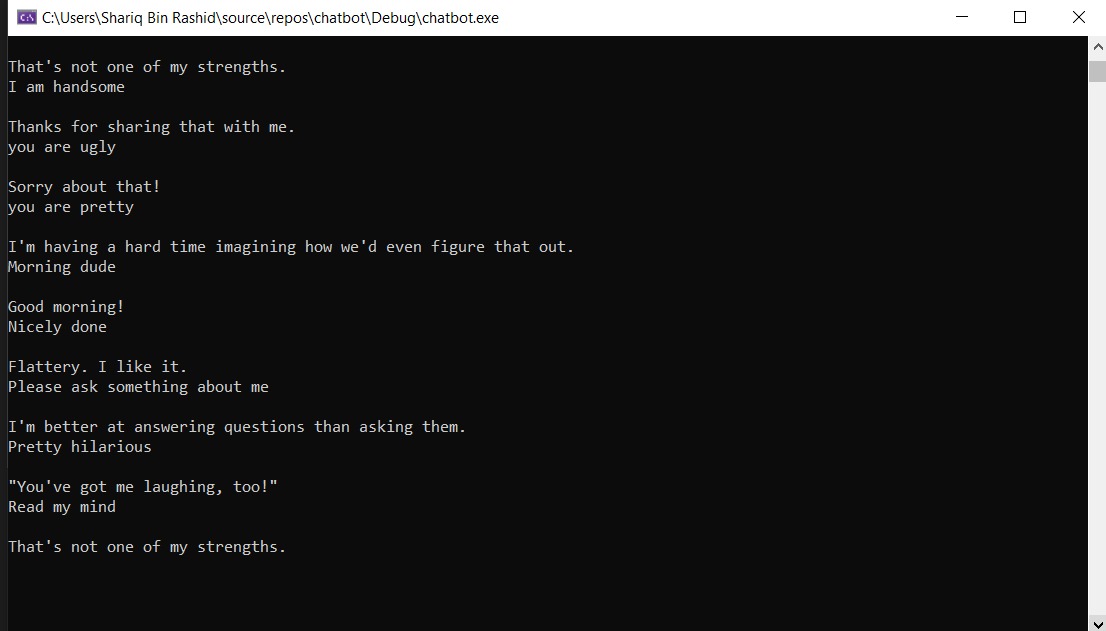












**Code:**

#include <iostream>

using namespace std;

#include <fstream>

#include <string>

#include <vector>

#include <algorithm>

#include <list>

#include <time.h>

#define hashSize 1000

#define hashTableSize 1000

string capitalize(string s)

{

string s2;

for (int i = 0; i < s.length(); i++)

{

s2.push\_back(toupper(s.at(i)));

}

return s2;

}

int hasFun2(string s, int maxVal)

{

s = capitalize(s);

int ascii;

int hash = 0;

for (int i = 0; i < s.length(); i++)

{

ascii = s.at(i);

hash = hash + ascii;

}

return (hash % maxVal);

}

int hasFun1(string s, int maxVal)

{

s = capitalize(s);

int ascii;

int hash = 0;

for (int i = 0; i < s.length(); i++)

{

ascii = s.at(i);

hash = hash + (ascii \* (i + 1));

}

return (hash % maxVal);

}

int hasFun3(string s, int maxVal)

{

s = capitalize(s);

int ascii;

int hash = 0;

for (int i = 0; i < s.length(); i++)

{

ascii = s.at(i);

//hash = hash + (ascii \* (i + 1));

hash = ascii + (hash << 6) + (hash << 16) - hash;

if (hash < 0)

{

hash = hash \* -1;

}

}

return (hash % maxVal);

}

class Word {

private:

string key;

string value;

public:

Word()

{

key = "defaultNull";

value = "defaultNull";

}

Word(string k, string val)

{

key = k;

value = val;

}

void setKey(string k)

{

key = k;

}

string getKey()

{

return key;

}

void setValue(string val)

{

value = val;

}

string getValue()

{

return value;

}

};

class HashTable {

private:

int totalElements = 0;

public:

int tableSize;

list<Word> hashTable[hashTableSize];

//CONSTRUCTORS

HashTable()

{

totalElements = 0;

tableSize = hashTableSize;

//hashTable = new list<Word>[tableSize];

}

HashTable(int size)

{

totalElements = 0;

tableSize = size;

//hashTable = new list<Word>[tableSize];

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int getTotalElements()

{

return totalElements;

}

void insertElement(Word word)

{

// Compute the index using Hash Function

int index = hasFun1(word.getKey(), tableSize);

// Insert the element in the linked list at the particular index

hashTable[index].push\_back(word);

totalElements++;

}

int searchForIndex(string key)

{

int index = hasFun1(key, hashSize);

list<Word>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getKey() == key)

{

return index;

}

p++;

}

return -1;

}

bool wordExsists(string key)

{

int index = hasFun1(key, hashSize);

list<Word>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getKey() == key)

{

return true;

}

p++;

}

return false;

}

string getVal(string key)

{

int index = hasFun1(key, hashSize);

list<Word>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getKey() == key)

{

return p->getValue();

}

p++;

}

return "NOT FOUND";

}

Word getWord(string key)

{

int index = hasFun1(key, hashSize);

list<Word>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getKey() == key)

{

return \*p;

}

p++;

}

}

};

HashTable mappedStemming;

vector<string> tokenize(string sentence)

{

vector<string> tokens;

string word;

for (int i = 0; i < sentence.length(); i++)

{

if (sentence[i] != ' ')

{

word.push\_back(sentence[i]);

}

else if (sentence[i] == ' ')

{

tokens.push\_back(word);

word.clear();

}

}

tokens.push\_back(word);

return tokens;

}

string filterString(string s)

{

string filteredString;

int ch;

int spaceCounter = 0;

int wordFound = 0;

for (int i = 0; i < s.length(); i++)

{

ch = s.at(i);

if ((ch >= 48 && ch <= 57) || (ch >= 65 && ch <= 90) || (ch >= 97 && ch <= 122) || (ch == 32))

{

if (ch == 32)

{

if (i == s.length() - 1 || i == 0 || wordFound == 0)

{

spaceCounter++;

continue;

}

if (spaceCounter < 1)

{

filteredString.push\_back(s.at(i));

spaceCounter++;

}

else

{

continue;

}

}

else

{

filteredString.push\_back(s.at(i));

spaceCounter = 0;

wordFound = 1;

}

}

else

{

//spaceCounter = 0;

}

}

if (filteredString.back() == 32)

{

filteredString.pop\_back();

}

return capitalize(filteredString);;

}

string sortString(string s)

{

string sortedString;

vector<string> tokens;

tokens = tokenize(s);

sort(tokens.begin(), tokens.end());

vector<string>::iterator p = tokens.begin();

while (p != tokens.end())

{

sortedString.append(\*p);

sortedString.append(" ");

p++;

}

sortedString.pop\_back();

return sortedString;

}

string stemmenizeString(string s)

{

string stemmedString;

vector<string> tokens;

tokens = tokenize(s);

vector<string>::iterator p = tokens.begin();

while (p != tokens.end())

{

if (mappedStemming.wordExsists(\*p))

{

stemmedString.append(mappedStemming.getVal(\*p));

}

else

{

stemmedString.append(\*p);

}

stemmedString.append(" ");

p++;

}

stemmedString.pop\_back();

if (stemmedString != s)

{

stemmedString = stemmenizeString(stemmedString);

}

return stemmedString;

}

string filterSortString(string s)

{

s = filterString(s);

s = stemmenizeString(s);

s = sortString(s);

return s;

}

class Question {

string question;

string ansFriendly = "defaultNullVal";

string ansCaring = "defaultNullVal";

string ansProfessional = "defaultNullVal";

string ansEnthausiastic = "defaultNullVal";

string ansWitty = "defaultNullVal";

string ansException = "defaultNullVal";

/\*Question()

{

question = "default";

ansFriendly = "default";

ansCaring = "default";

ansProfessional = "default";

ansEnthausiastic = "default";

ansWitty = "default";

}\*/

public:

void setQuestion(string question1)

{

question = question1;

}

string getQuestion()

{

return question;

}

void setAnsFriendly(string ansFriendly1)

{

ansFriendly = ansFriendly1;

}

string getAnsFriendly()

{

return ansFriendly;

}

void setAnsCaring(string ansCaring1)

{

ansCaring = ansCaring1;

}

string getAnsCaring()

{

return ansCaring;

}

void setAnsProfessional(string ansProfessional1)

{

ansProfessional = ansProfessional1;

}

string getAnsProfessional()

{

return ansProfessional;

}

void setAnsEnthausiastic(string ansEnthausiastic1)

{

ansEnthausiastic = ansEnthausiastic1;

}

string getAnsEnthausiastic()

{

return ansEnthausiastic;

}

void setAnsWitty(string ansWitty1)

{

ansWitty = ansWitty1;

}

string getAnsWitty()

{

return ansWitty;

}

void setAnsException(string ex)

{

ansException = ex;

}

string getAnsException()

{

return ansException;

}

};

class HashMap {

private:

int totalElements = 0;

public:

list<Question> hashTable[hashSize];

//CONSTRUCTORS

/\*HashMap()

{

list<Question> hashTable[hashSize];

}

HashMap(int size)

{

list<Question> \*hashTable = new list<Question>[size];

}\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int getTotalElements()

{

return totalElements;

}

void insertVal(Question q)

{

// Compute the index using Hash Function

int index = hasFun1(q.getQuestion(), hashSize);

// Insert the element in the linked list at the particular index

hashTable[index].push\_back(q);

totalElements++;

}

int searchForIndex(Question q)

{

int index = hasFun1(q.getQuestion(), hashSize);

list<Question>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getQuestion() == q.getQuestion())

{

return index;

break;

}

p++;

}

return -1;

}

bool questionExsists(Question q)

{

int index = hasFun1(q.getQuestion(), hashSize);

list<Question>::iterator p = hashTable[index].begin();

while (p != hashTable[index].end())

{

if (p->getQuestion() == q.getQuestion())

{

return true;

break;

}

p++;

}

return false;

}

string getAns(string q)

{

int index = hasFun1(q, hashSize);

int r = rand() % 5;

list<Question>::iterator p = hashTable[index].begin();

int reRandCounter = 0;

int friendlyNullCounter = 0;

while (p != hashTable[index].end())

{

if (p->getQuestion() == q)

{

if (reRandCounter < 5000)

{

if (r == 0)

{

if (p->getAnsFriendly() == "defaultNullVal")

{

r = rand() % 5;

reRandCounter++;

continue;

}

return p->getAnsFriendly();

}

else if (r == 1)

{

if (p->getAnsCaring() == "defaultNullVal")

{

if (friendlyNullCounter == 0)

{

r = 0;

reRandCounter++;

friendlyNullCounter++;

continue;

}

r = rand() % 5;

reRandCounter++;

continue;

}

return p->getAnsCaring();

}

else if (r == 2)

{

if (p->getAnsEnthausiastic() == "defaultNullVal")

{

if (friendlyNullCounter == 0)

{

r = 0;

reRandCounter++;

friendlyNullCounter++;

continue;

}

r = rand() % 5;

reRandCounter++;

continue;

}

return p->getAnsEnthausiastic();

}

else if (r == 3)

{

if (p->getAnsProfessional() == "defaultNullVal")

{

if (friendlyNullCounter == 0)

{

r = 0;

reRandCounter++;

friendlyNullCounter++;

continue;

}

r = rand() % 5;

reRandCounter++;

continue;

}

return p->getAnsProfessional();

}

else if (r == 4)

{

if (p->getAnsWitty() == "defaultNullVal")

{

if (friendlyNullCounter == 0)

{

r = 0;

reRandCounter++;

friendlyNullCounter++;

continue;

}

r = rand() % 5;

reRandCounter++;

continue;

}

return p->getAnsWitty();

}

}

else {

return "No Answer :(";

}

}

p++;

}

return "NOT FOUND";

}

};

void loadStemmingWords(string sourceFile, char usedDelimiter)

{

string line;

string key;

string value;

char ch;

int delimiterCounter = 0;

ifstream srcFile(sourceFile);

if (srcFile.is\_open())

{

while (getline(srcFile, line))

{

delimiterCounter = 0;

for (int i = 0; i < line.length(); i++)

{

ch = line.at(i);

if (ch == usedDelimiter)

{

delimiterCounter++;

continue;

}

if (delimiterCounter == 0)

{

value.push\_back(ch);

}

else if (delimiterCounter == 1)

{

key.push\_back(ch);

}

}

value = filterString(value);

key = filterString(key);

Word\* w = new Word(value, value);

Word\* word = new Word(key, value);

if (!mappedStemming.wordExsists(value))

{

mappedStemming.insertElement(\*w);

}

if (!mappedStemming.wordExsists(key))

{

mappedStemming.insertElement(\*word);

}

key.clear();

value.clear();

}

}

srcFile.close();

}

HashMap h1;

void loadQuestionsData(string sourceFile, char usedDelimiter)

{

string line;

string question;

string answer;

string ansFriendly;

string ansCaring;

string ansProfessional;

string ansEnthausiastic;

string ansWitty;

string ansException;

char ch;

int delimiterCounter = 0;

Question q;

ifstream srcFile(sourceFile);

if (srcFile.is\_open())

{

while (getline(srcFile, line))

{

delimiterCounter = 0;

for (int i = 0; i < line.length(); i++)

{

ch = line.at(i);

if (ch == usedDelimiter)

{

delimiterCounter++;

continue;

}

if (delimiterCounter == 0)

{

question.push\_back(ch);

}

else if (delimiterCounter == 1)

{

ansFriendly.push\_back(ch);

}

else if (delimiterCounter == 2)

{

ansCaring.push\_back(ch);

}

else if (delimiterCounter == 3)

{

ansProfessional.push\_back(ch);

}

else if (delimiterCounter == 4)

{

ansEnthausiastic.push\_back(ch);

}

else if (delimiterCounter == 5)

{

ansWitty.push\_back(ch);

}

else

{

ansException.push\_back(ch);

cout << endl << "Delimeter Exceeded !" << endl;

}

}

question = filterSortString(question);

q.setQuestion(question);

if (!h1.questionExsists(q))

{

h1.insertVal(q);

int index = h1.searchForIndex(q);

if (index != -1)

{

list<Question>::iterator p = h1.hashTable[index].begin();

while (p != h1.hashTable[index].end())

{

if (p->getQuestion() == q.getQuestion())

{

if (delimiterCounter == 1)

{

p->setAnsFriendly(ansFriendly);

}

else if (delimiterCounter == 2)

{

p->setAnsFriendly(ansFriendly);

p->setAnsCaring(ansCaring);

}

else if (delimiterCounter == 3)

{

p->setAnsFriendly(ansFriendly);

p->setAnsCaring(ansCaring);

p->setAnsProfessional(ansProfessional);

}

else if (delimiterCounter == 4)

{

p->setAnsFriendly(ansFriendly);

p->setAnsCaring(ansCaring);

p->setAnsProfessional(ansProfessional);

p->setAnsEnthausiastic(ansEnthausiastic);

}

else if (delimiterCounter == 5)

{

p->setAnsFriendly(ansFriendly);

p->setAnsCaring(ansCaring);

p->setAnsProfessional(ansProfessional);

p->setAnsEnthausiastic(ansEnthausiastic);

p->setAnsWitty(ansWitty);

}

else if (delimiterCounter > 5)

{

p->setAnsFriendly(ansFriendly);

p->setAnsCaring(ansCaring);

p->setAnsProfessional(ansProfessional);

p->setAnsEnthausiastic(ansEnthausiastic);

p->setAnsWitty(ansWitty);

p->setAnsException(ansException);

}

break;

}

p++;

}

}

}

question.clear();

answer.clear();

ansFriendly.clear();

ansCaring.clear();

ansEnthausiastic.clear();

ansProfessional.clear();

ansWitty.clear();

ansException.clear();

}

}

srcFile.close();

}

void insertQuestion(string sourceFile, char usedDelimiter, string que, string ans)

{

char ch;

int delimiterCounter = 0;

string question = que;

string answer = ans;

Question q;

q.setQuestion(question);

q.setAnsFriendly(answer);

ofstream srcFile;

srcFile.open(sourceFile, std::ios\_base::app); // append instead of overwrite

srcFile << question << usedDelimiter << ans << endl;

srcFile.close();

}

int main()

{

srand((unsigned)time(0));

string questionAnswersFilePath = "D:/questionAnswers.txt";

string lemmatizationMappingFilePath = "D:/lemmatizationData.txt";

cout << endl << "WAIT ! Loading ChatBot.." << endl;

loadStemmingWords(lemmatizationMappingFilePath, '\t');

loadQuestionsData(questionAnswersFilePath, '\t');

cout << endl << "ChatBot Loaded succesfully !!" << endl;

cout << endl << "Total Questions in hash table: " << h1.getTotalElements() << endl;

cout << endl << "Total Words in 2nd hash table: " << mappedStemming.getTotalElements() << endl;

string askedQuestion = "1";

string filteredQuestion;

string suggestedAnswer;

cout << endl << "Hello Human ! How may I help you?" << endl;

Question qUser;

Question q;

getline(cin, askedQuestion);

while (askedQuestion != "0")

{

filteredQuestion = filterSortString(askedQuestion);

q.setQuestion(filteredQuestion);

if (h1.getAns(filteredQuestion) == "NOT FOUND")

{

cout << endl << "I'm sorry i do not know the answer to this question" << endl;

cout << "But if you tell me it's answer i'll remember it next time" << endl;

getline(cin, suggestedAnswer);

insertQuestion(questionAnswersFilePath, '\t', askedQuestion, suggestedAnswer);

if (!h1.questionExsists(q))

{

h1.insertVal(q);

int index = h1.searchForIndex(q);

if (index != -1)

{

list<Question>::iterator p = h1.hashTable[index].begin();

while (p != h1.hashTable[index].end())

{

if (p->getQuestion() == q.getQuestion())

{

p->setAnsFriendly(suggestedAnswer);

break;

}

p++;

}

}

}

cout << endl << "Thank You! I'll remember it next time :)" << endl;

}

else

{

cout << endl << h1.getAns(filteredQuestion) << endl;

}

getline(cin, askedQuestion);

}

cout << endl;

}