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**

**Project: AS DICTIONARY**

**SUBMITTED TO: Dr. Qaiser Riaz**

**SUBMITTED BY:**

**1.** **\_Shehriyar Shariq (245009)\_\_\_**

**2.** **\_Ayesha Siddiqui (241615)\_\_\_**

**Class (Section): \_\_\_\_\_\_\_BSCS 8A\_\_\_\_\_**

**Date of Submission: \_\_\_\_\_\_\_21/12/2019\_\_\_\_\_\_\_**

**Description of the Project:**

In today’s digital era, using printed dictionaries to find meanings of words is very inefficient and time consuming. Using online dictionaries requires the user to have an internet connection in order to search for the word.

AS Dictionary is an offline dictionary, containing a word base of more than 10,800 words! It uses data structures and programming algorithms that allow the user to look for any word in a fraction of a second. Thus providing time efficient and accurate solution to our earlier problems.

The dictionary has a Graphical User Interface that makes it more user friendly, enhancing the experience of our program. The GUI features a log panel, where history of search words is displayed for the user to easily access frequently searched words. Words are stored as a file, and file lookup is used for further details of words. The search box field on the home page has an autocomplete functionality that displays a list of suggested words to the user while typing in the word. Furthermore, the Dictionary has Undo/Redo features to allow users to switch between past searches.

The main motivation behind this project was to learn and implement an in depth knowledge of various data structures including linked lists, stacks and trees.

This project idea can be further modified to make a search engine, which could one day beat google

**Solution:**

***Tools used:***

* Visual Studio
* Qt Creator for C++

***Data Structures and Algorithms used:***

* ***Ternary Search Trees***

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

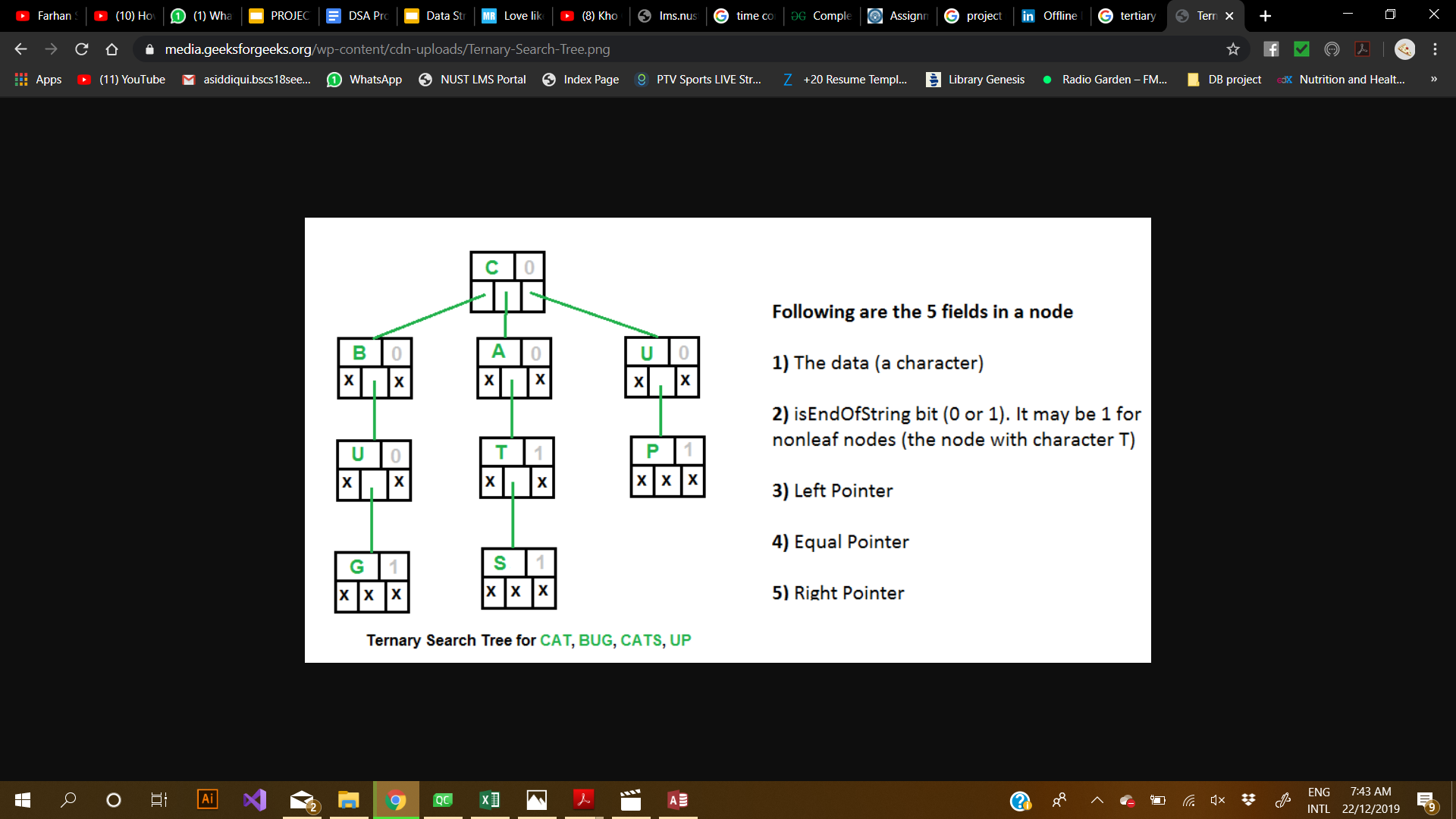
It is more or less similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes.

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

By using only 3 pointers instead of 26, Ternary Search trees are considered more advantageous and efficient. Each node in a TEST also has a field to indicate data (character in case of dictionary) and another field to mark the end of a string.

Time complexity: log(n) ; data is divided and half of it is used in each step.



* ***Singly Linked Lists***

Singly linked lists are used in AS Dictionary for building the log panel that allows users to view recent searches.

Linked List is a linear data structure in which elements are not stored in a contiguous location; the elements are linked using pointers.

Linked list is used mainly because of the dynamic size of the data structure and the ease of insertion or deletion of data, in our case, words with their respective meaning.

Time Complexity: O(1), since no loops are run

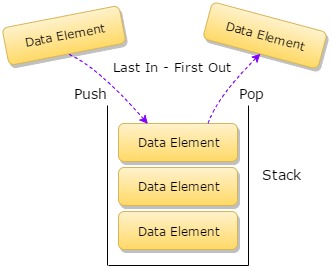


* ***Stacks***

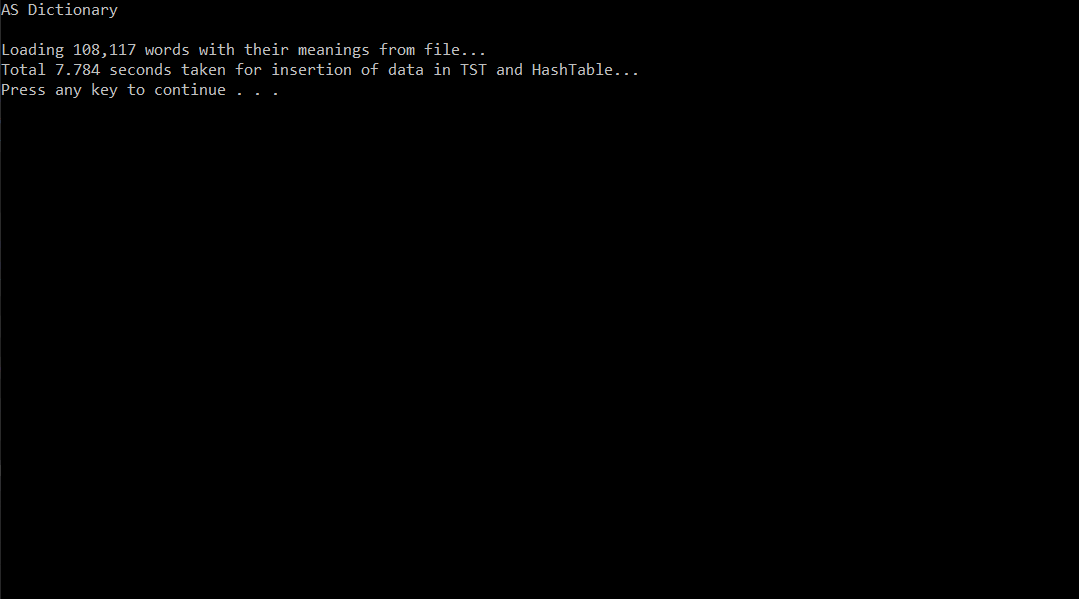
Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

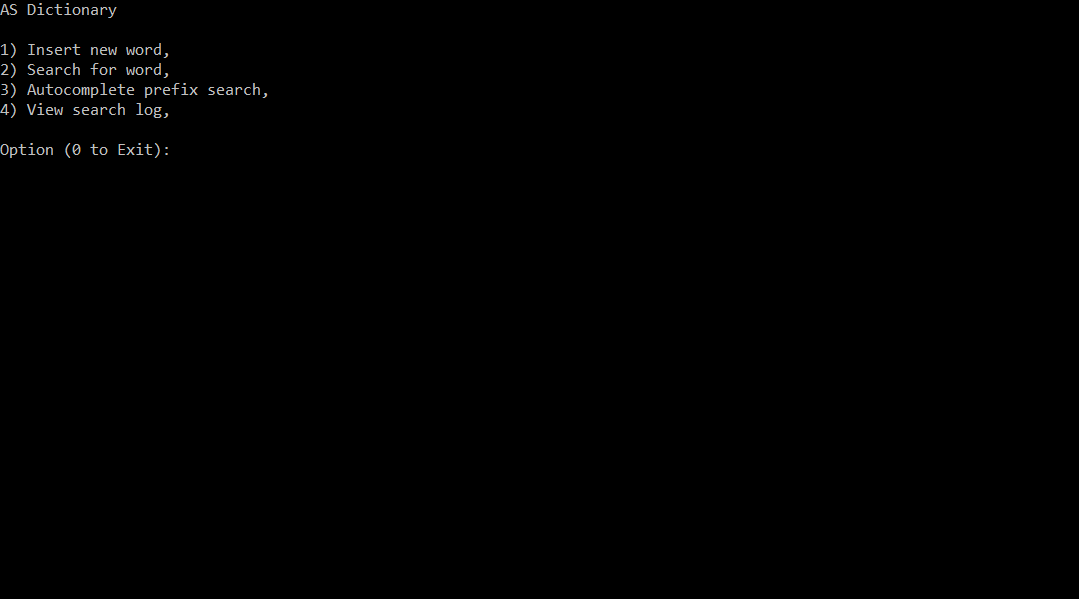
This Dictionary uses Linked List based implementation of Stacks because of the dynamic size and easy insertion/deletion. Word searches are stored in Stacks, in order to enable the user to Undo and Redo their searches.

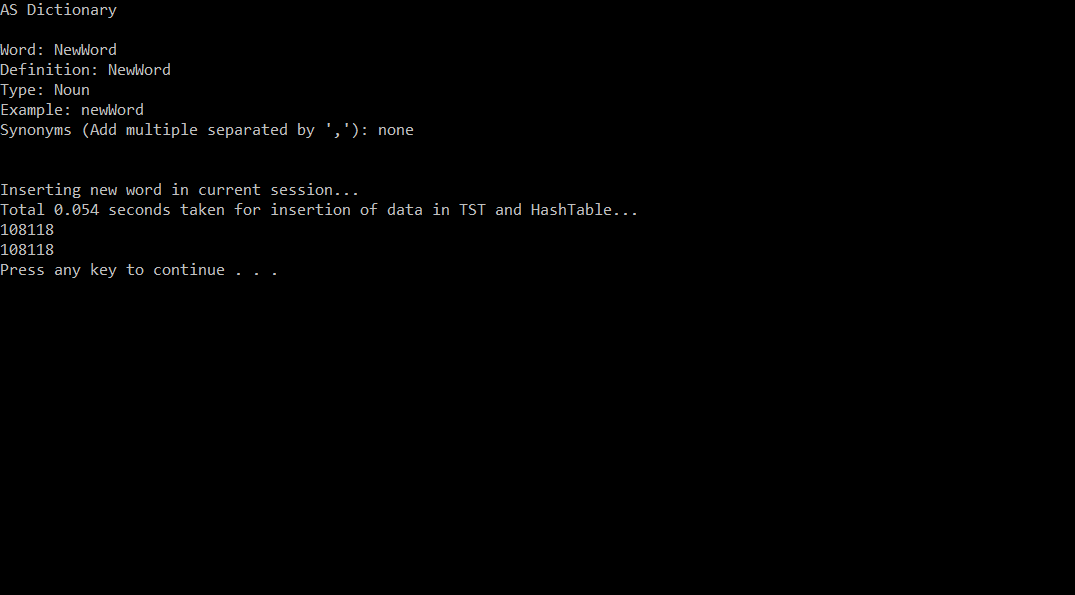
Time Complexity: all the functions of Push(), Pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

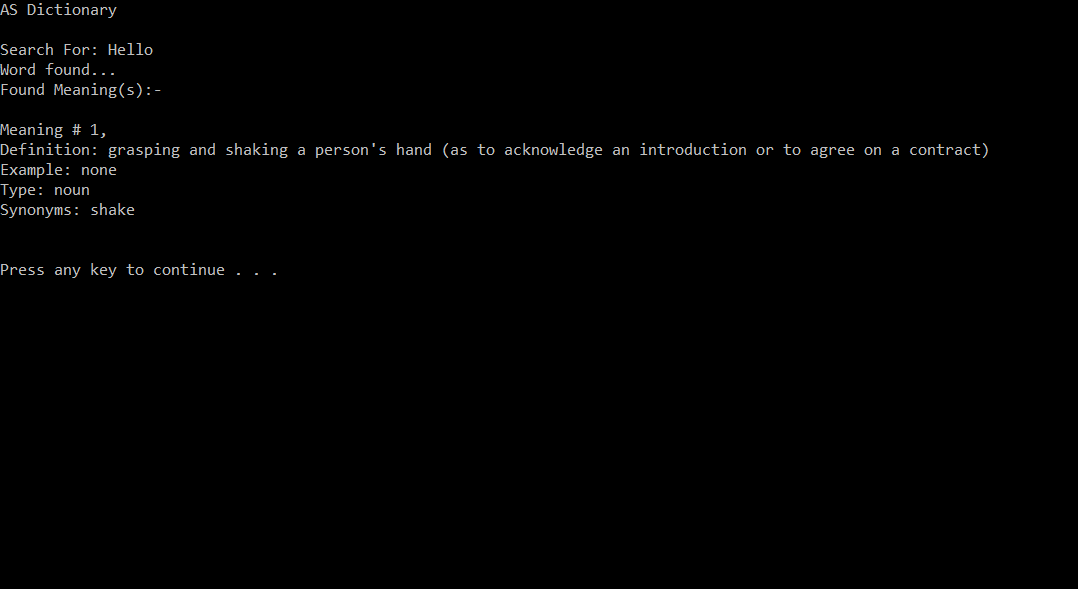


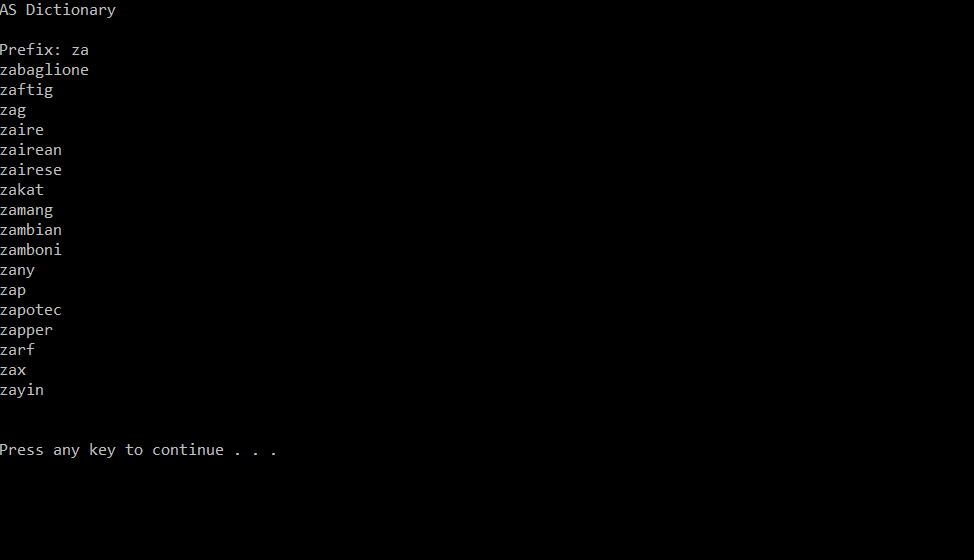
**Screenshots of the Output:**











**Source Code:**

**#include "pch.h"**

**#include <iostream>**

**#include <algorithm>**

**#include <cctype>**

**#include <string>**

**#include <fstream>**

**#include <vector>**

**#include <time.h>**

**#include <iomanip>**

**#define BACK\_LIMIT 4**

**using namespace std;**

**class Meaning {**

**private:**

**string def, example, type;**

**vector<string> synonyms;**

**public:**

**Meaning(string def, string example, string type, vector<string> synonyms) {**

**this->def = def;**

**this->type = type;**

**this->synonyms = synonyms;**

**this->example = example;**

**}**

**void display() {**

**cout << "Definition: " << def << endl;**

**cout << "Example: " << example << endl;**

**cout << "Type: " << type << endl;**

**cout << "Synonyms: ";**

**for (int i = 0; i < synonyms.size(); i++) {**

**cout << synonyms.at(i) << " ";**

**}**

**cout << endl;**

**}**

**};**

**class Meanings {**

**public:**

**vector<Meaning> meanings;**

**Meanings(vector<Meaning> meanings) {**

**this->meanings = meanings;**

**}**

**};**

**class HashTable {**

**private:**

**vector<Meanings> allWordsMeanings;**

**public:**

**void insert(Meanings meanings) {**

**allWordsMeanings.push\_back(meanings);**

**}**

**Meanings get(int pos) {**

**return allWordsMeanings.at(pos);**

**}**

**void display(int pos) {**

**vector<Meaning> meanings = (get(pos)).meanings;**

**for (int i = 0; i < meanings.size(); i++) {**

**Meaning meaning = meanings.at(i);**

**cout << endl << "Meaning # " << (i + 1) << ", " << endl;**

**meaning.display();**

**}**

**}**

**int getSize() {**

**return allWordsMeanings.size();**

**}**

**};**

**class LinkedList {**

**private:**

**class ListNode {**

**public:**

**string val;**

**ListNode \*next;**

**ListNode(string val, ListNode \*next = NULL) {**

**this->val = val;**

**this->next = next;**

**}**

**};**

**ListNode \*head = NULL;**

**int len = 0;**

**public:**

**// Insert at Head**

**void insertHead(string val) {**

**ListNode\* newHead = new ListNode(val, head); // Make new head**

**head = newHead; // Reassign Head**

**len++; // Increase length**

**}**

**void insertAtEnd(string val) {**

**if (head) {**

**ListNode\* temp = head;**

**while (temp->next != NULL) {**

**temp = temp->next;**

**}**

**temp->next = new ListNode(val);**

**len++;**

**} else {**

**insertHead(val);**

**}**

**}**

**void displayList() {**

**// Display list until last node**

**if (head != NULL) {**

**ListNode\* temp = head;**

**while (temp) {**

**cout << temp->val << " ";**

**temp = temp->next;**

**}**

**}**

**}**

**};**

**class Queue {**

**private:**

**vector<string> elems;**

**public:**

**void pushBack(string element) {**

**if (elems.size() >= BACK\_LIMIT) {**

**elems.pop\_back();**

**}**

**elems.push\_back(element);**

**};**

**string popFront() {**

**string elem = elems.front();**

**elems.erase(elems.begin());**

**return elem; // Return top element and then decrement top index**

**};**

**bool isEmpty() {**

**return elems.empty(); // No Elements**

**};**

**void clear() {**

**elems.clear();**

**};**

**string front() {**

**return elems.front();**

**};**

**};**

**class Stack {**

**private:**

**vector<string> elems;**

**public:**

**void push(string element) {**

**if (elems.size() >= BACK\_LIMIT) {**

**elems.erase(elems.begin());**

**}**

**elems.push\_back(element);**

**};**

**string pop() {**

**string elem = elems.back();**

**elems.pop\_back();**

**return elem; // Return top element and then decrement top index**

**};**

**bool isEmpty() {**

**return elems.empty(); // No Elements**

**};**

**void clear() {**

**elems.clear();**

**};**

**string top() {**

**return elems.back();**

**};**

**};**

**class Node {**

**public:**

**char key;**

**int ind = -1;**

**Node \*left, \*mid, \*right;**

**Node(char key, Node \*left = NULL, Node \*mid = NULL, Node \*right = NULL, int ing = -1) {**

**this->key = key;**

**this->ind = ind;**

**this->left = left;**

**this->mid = mid;**

**this->right = right;**

**}**

**};**

**class TernarySearchTree {**

**private:**

**vector<string> searchResults;**

**int totalWords = 0;**

**public:**

**Node\* insert(Node \*root, const char\* word, int ind) {**

**if (root == NULL) {**

**root = new Node(word[0]);**

**}**

**if (\*word < root->key) {**

**root->left = insert(root->left, word, ind);**

**} else if (\*word == root->key) {**

**if (\*(word + 1)) {**

**root->mid = insert(root->mid, word + 1, ind);**

**} else {**

**root->ind = ind;**

**totalWords++;**

**}**

**} else {**

**root->right = insert(root->right, word, ind);**

**}**

**return root;**

**}**

**void display(Node \*root, char\* word, int level, bool withPrefix, const char \*prefix) {**

**if (root) {**

**display(root->left, word, level, withPrefix, prefix);**

**word[level] = root->key;**

**if (root->ind != -1) {**

**word[level + 1] = '\0';**

**if (withPrefix) {**

**searchResults.push\_back(string(prefix) + string(word));**

**cout << string(prefix) + string(word) << endl;**

**} else {**

**cout << string(word) << endl;**

**searchResults.push\_back(string(word));**

**}**

**}**

**display(root->mid, word, level + 1, withPrefix, prefix);**

**display(root->right, word, level, withPrefix, prefix);**

**}**

**}**

**int search(Node \*root, const char \*word) {**

**while (root) {**

**if (\*word < root->key) {**

**root = root->left;**

**} else if (\*word == root->key) {**

**if ((root->ind != -1) && \*(word + 1) == '\0') {**

**return root->ind;**

**}**

**word++;**

**root = root->mid;**

**} else {**

**root = root->right;**

**}**

**}**

**return -1;**

**}**

**vector<string> prefixSearch(Node \*root, const char \*prefix, const char \*original, int len) {**

**while (\*prefix != '\0') {**

**if (\*prefix < root->key) {**

**root = root->left;**

**} else if (\*prefix == root->key) {**

**prefix++;**

**root = root->mid;**

**} else {**

**root = root->right;**

**}**

**}**

**searchResults.clear();**

**char word[50];**

**display(root, word, 0, true, original);**

**return searchResults;**

**}**

**int getCurrentTotalWords() {**

**return totalWords;**

**}**

**};**

**void showAutoCompletePrefixMenu(TernarySearchTree tst, Node \*root) {**

**system("clear");**

**cout << "AS Dictionary" << endl << endl;**

**string word;**

**cout << "Prefix: ";**

**cin >> word;**

**std::transform(word.begin(), word.end(), word.begin(),**

**[](unsigned char c) { return std::tolower(c); });**

**vector<string> result = tst.prefixSearch(root, word.c\_str(), word.c\_str(), word.length());**

**cout << endl << endl;**

**system("pause");**

**}**

**void showSearchMenu(TernarySearchTree tst, Node \*root, HashTable ht) {**

**system("clear");**

**cout << "AS Dictionary" << endl << endl;**

**string word;**

**cout << "Search For: ";**

**cin >> word;**

**std::transform(word.begin(), word.end(), word.begin(),**

**[](unsigned char c) { return std::tolower(c); });**

**int res = tst.search(root, word.c\_str());**

**if (res != -1) {**

**cout << "Word found..." << endl;**

**cout << "Found Meaning(s):- " << endl;**

**ht.display(res);**

**} else {**

**cout << "Word not found...";**

**}**

**cout << endl << endl;**

**system("pause");**

**}**

**void showInsertWordMenu(TernarySearchTree tst, Node \*root, HashTable ht) {**

**system("clear");**

**cout << "AS Dictionary" << endl << endl;**

**string word, definition, type, example, synonyms;**

**cout << "Word: ";**

**cin >> word;**

**cout << "Definition: ";**

**cin >> definition;**

**cout << "Type: ";**

**cin >> type;**

**cout << "Example: ";**

**cin >> example;**

**cout << "Synonyms (Add multiple separated by ','): ";**

**cin >> synonyms;**

**string synonymsDelimiter = ",";**

**vector<string> synonymsArr;**

**size\_t pos = 0;**

**string token;**

**while ((pos = synonyms.find(synonymsDelimiter)) != string::npos) {**

**token = synonyms.substr(0, pos);**

**synonymsArr.push\_back(token);**

**synonyms.erase(0, pos + synonymsDelimiter.length());**

**}**

**synonymsArr.push\_back(synonyms);**

**Meaning meaning(definition, example, type, synonymsArr);**

**vector<Meaning> meaningsArr;**

**meaningsArr.push\_back(meaning);**

**Meanings meanings(meaningsArr);**

**cout << endl << endl << "Inserting new word in current session..." << endl;**

**clock\_t startTime = clock();**

**tst.insert(root, word.c\_str(), tst.getCurrentTotalWords() + 1);**

**ht.insert(meanings);**

**clock\_t endTime = clock();**

**cout << "Total " << ((endTime - startTime) / (double)CLOCKS\_PER\_SEC) << " seconds taken for insertion of data in TST and HashTable..." << endl;**

**cout << tst.getCurrentTotalWords() << endl;**

**cout << ht.getSize() << endl;**

**system("pause");**

**}**

**void showMainMenu(TernarySearchTree tst, Node \*root, HashTable ht) {**

**while (true) {**

**system("clear");**

**cout << "AS Dictionary" << endl << endl;**

**cout << "1) Insert new word," << endl;**

**cout << "2) Search for word," << endl;**

**cout << "3) Autocomplete prefix search," << endl;**

**cout << "4) View search log," << endl << endl;**

**int inp;**

**cout << "Option (0 to Exit): ";**

**cin >> inp;**

**switch (inp) {**

**case 1:**

**showInsertWordMenu(tst, root, ht);**

**break;**

**case 2:**

**showSearchMenu(tst, root, ht);**

**break;**

**case 3:**

**showAutoCompletePrefixMenu(tst, root);**

**break;**

**case 4:**

**break;**

**default:**

**exit(0);**

**}**

**}**

**}**

**int main() {**

**TernarySearchTree tst;**

**Node \*root = NULL;**

**HashTable wordMeanings;**

**string meaningsDelimiter = "=\_=";**

**string meaningInfoDelimiter = "=>";**

**string synonymsDelimiter = ",";**

**ifstream wordInpFile;**

**ifstream meaningsInpFile;**

**wordInpFile.open("words.txt");**

**meaningsInpFile.open("meanings.txt");**

**char\* words[108117];**

**string tempFileInp;**

**string tempFileMeaningsInp;**

**int i = 0;**

**cout << "AS Dictionary" << endl << endl;**

**cout << "Loading 108,117 words with their meanings from file..." << endl;**

**clock\_t startTime = clock();**

**if (wordInpFile.is\_open() && meaningsInpFile.is\_open()) {**

**while (!wordInpFile.eof() && !meaningsInpFile.eof()) {**

**getline(wordInpFile, tempFileInp);**

**getline(meaningsInpFile, tempFileMeaningsInp);**

**std::transform(tempFileInp.begin(), tempFileInp.end(), tempFileInp.begin(),**

**[](unsigned char c) { return std::tolower(c); });**

**root = tst.insert(root, tempFileInp.c\_str(), i);**

**string s = tempFileMeaningsInp;**

**size\_t pos = 0;**

**string token;**

**vector<string> meaningsStrs;**

**while ((pos = s.find(meaningsDelimiter)) != string::npos) {**

**token = s.substr(0, pos);**

**meaningsStrs.push\_back(token);**

**s.erase(0, pos + meaningsDelimiter.length());**

**}**

**meaningsStrs.push\_back(s);**

**vector<Meaning> meaningsArr;**

**for (int i = 0; i < meaningsStrs.size(); i++) {**

**vector<string> meaningInfoStrs;**

**string meaningStr = meaningsStrs.at(i);**

**pos = 0;**

**while ((pos = meaningStr.find(meaningInfoDelimiter)) != string::npos) {**

**token = meaningStr.substr(0, pos);**

**meaningInfoStrs.push\_back(token);**

**meaningStr.erase(0, pos + meaningInfoDelimiter.length());**

**}**

**meaningInfoStrs.push\_back(meaningStr);**

**string synonymsStr = meaningInfoStrs.at(3);**

**vector<string> synonyms;**

**pos = 0;**

**while ((pos = synonymsStr.find(synonymsDelimiter)) != string::npos) {**

**token = synonymsStr.substr(0, pos);**

**synonyms.push\_back(token);**

**synonymsStr.erase(0, pos + synonymsDelimiter.length());**

**}**

**synonyms.push\_back(synonymsStr);**

**Meaning meaning(meaningInfoStrs.at(0), meaningInfoStrs.at(1), meaningInfoStrs.at(2), synonyms);**

**meaningsArr.push\_back(meaning);**

**}**

**Meanings meanings(meaningsArr);**

**wordMeanings.insert(meanings);**

**i++;**

**}**

**}**

**wordInpFile.close();**

**meaningsInpFile.close();**

**clock\_t endTime = clock();**

**cout << "Total " << ((endTime - startTime) / (double)CLOCKS\_PER\_SEC) << " seconds taken for insertion of data in TST and HashTable..." << endl;**

**system("pause");**

**showMainMenu(tst, root, wordMeanings);**

**system("pause");**

**return 0;**

**}**