# ВСТУП

Актуальність: Кожен користувач комп’ютера майже щоденно використовує програми роботи з текстом, такі як спеціалізовані текстові редактори, введення тексту в пошукових системах, в системах інтерактивного спілкування, тощо… І однією з найбільших відмінностей таких програмних засобів є інтерактивна перевірка правопису, зокрема правильність написання слів. Системи порівнюють веденні користувачем слова з словником виявляв та помічав незнайомі або не правильні слова. Довгий час такі функції були притаманні лише системам які використовували дуже потужні комп’ютери,. Але з розвитком телекомунікацій виявило потребу в таких функціях і в інших комп’ютерних системах, такі як мобільні телефони, інтернет браузери, сервіси голосового вводу тексту. Основним механізмом в реалізації такої функції в цих системах є словник, а ключовим єлементом єлектронного словника є механізм пошуку.

З оглу на це цілбю моєї роботи було розробка алгоритму швидкого пошуку в словнику, швидкість якого задовольнить потреби інтерактивного редагування текстів.

Для досягнення цієї цілі потрібно було вирішити такі задачі:

1. Дослідити методи формування словників, та механізми швидкого пошуку в них.
2. Порівняти різні види алгоритмів на предмет їх швидкодії, вимог до ресурсів пам’яті., тощо…
3. Вивчити особливості реалізації алгоритмів пошуку та структур словників різного виду.
4. Вибрати найбільш підходящій алгоритм реалізації словника що забезпечить потреби орфографічної перевірки в текстових редакторах.
5. Створити діючий прототип такого словника на мові програмування С.
6. Дослідити швидкодію розробленого програмного забезпечення.

## СУЧАСНІ МЕТОДИ СОРТУВАННЯ ТА ПОШУКУ

Механізми електронних словників базуються на двох взаємо пов’язаних алгоритмах. Це алгоритми сортування та алгоритми пошуку. Під сортуванням ми розуміємо спеціальну послідовність розміщення інформації, яка забезпечує виконання заданих операцій з цією інформацією. Під алгоритмом пошуку розуміють механізми знаходження деякої інформації в її великому обсязі. Сортування, та механізми пошуку взаємо пов’язані. Вдалий вибір сортування забезпечує швидкий пошук.

ПРИКЛАД

Відповідно до поставленої цілі ми хочемо створити електронний словник основною вимогою до якого є найшвидший механізм пошуку, тоді як інші вимоги є менш важливим, адже словник створюється лише один раз і поповнюється досить рідко.

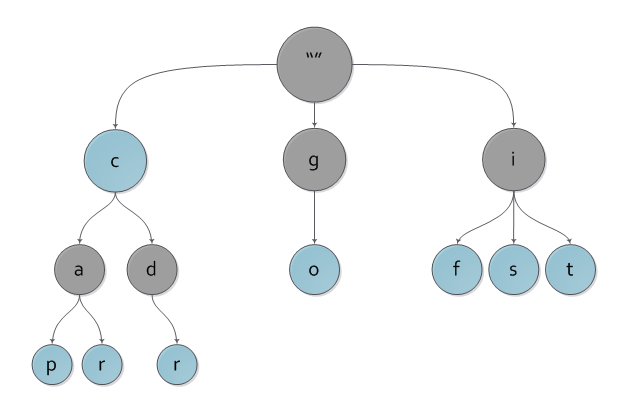
Були розлянуті найбільш популярні механізми пошуку, такі як:

1. Сортування вибором,
2. Сортування включенням
3. Сортування обміном
4. Пірамідальне сортування
5. Швидке сортування
6. Сортування злиттям
7. Сортування підрахунком
8. Сортування за розрядами
9. Сортування комірками
10. Сортування Ше́лла
11. Лінійний пошук
12. Двійковий пошук
13. Пошук в дереві

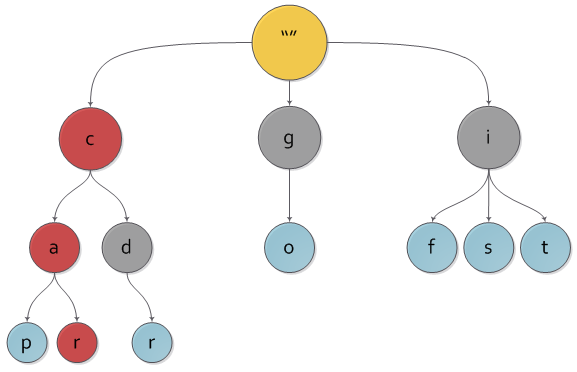
Огляд цих алгоритмів дає підстави говорити, що одним з найефективніших алгоритмів для інтерактивних словників є механізми дерева, зокрема так званого префіксного дерева.

Префіксне дерево – структура даних, що дозволяє зберігати асоціативний масив, ключами якого є рядок символів. Значення елемента можна знайти переглядом всіх батьківських вузлів, кожен з яких зберігає один або кілька символів алфавіту. Таким чином, нащадки вузла мають загальний префікс, звідки і пішла назва цієї структури даних.

Для електронного словника ми застосуємо так зване навантажене префіксне дерево. На малюнку ви можете спостерігати приклад навантаженого дерева з ключами *c, cap, car, cdr, go, if, is, it.*



І те ж саме дерево з виділеним на ньому ключем car.і

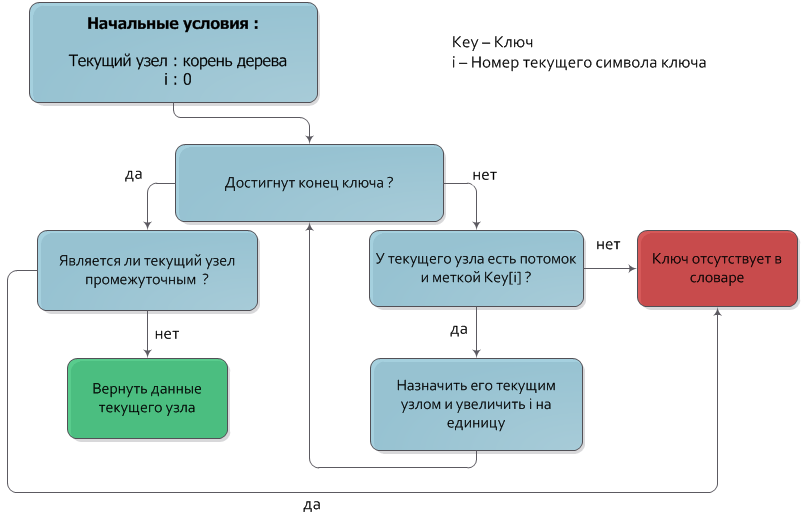
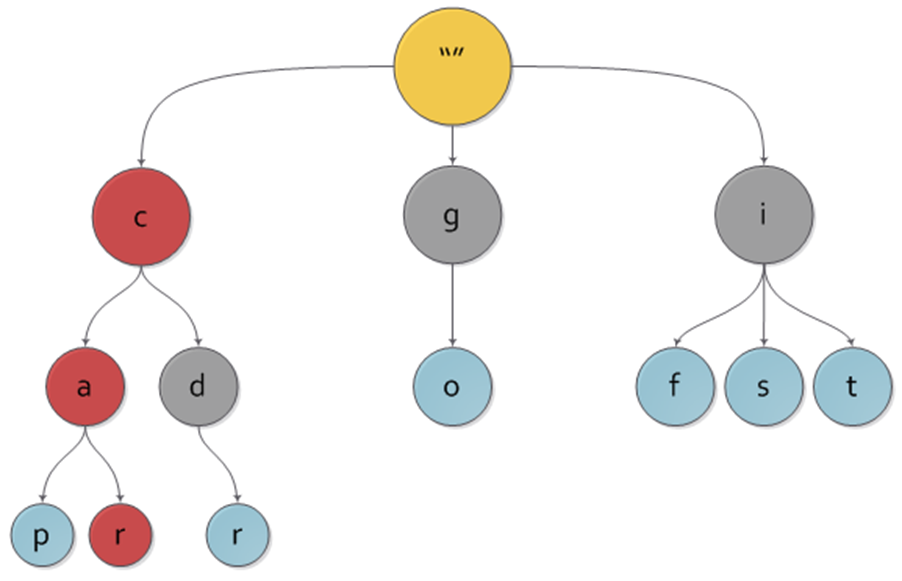


Так як навантажене дерево реалізує інтерфейс МНОЖИНИ (точніше асоціативного масиву), в ньому можна виділити три основні операції, а саме вставку, видалення і пошук ключа. Як і багато дерев, навантажене дерево має властивість самоподібності, тобто будь-яке його під-дерево також є повноцінним навантаженим деревом.

## 2.2. Пошук ключа

Алгортим пошуку в префіксному дереві показаний на малюнку:

Швидкодія такого механізму пропорційне кількості символів слову яке ми шукаємо

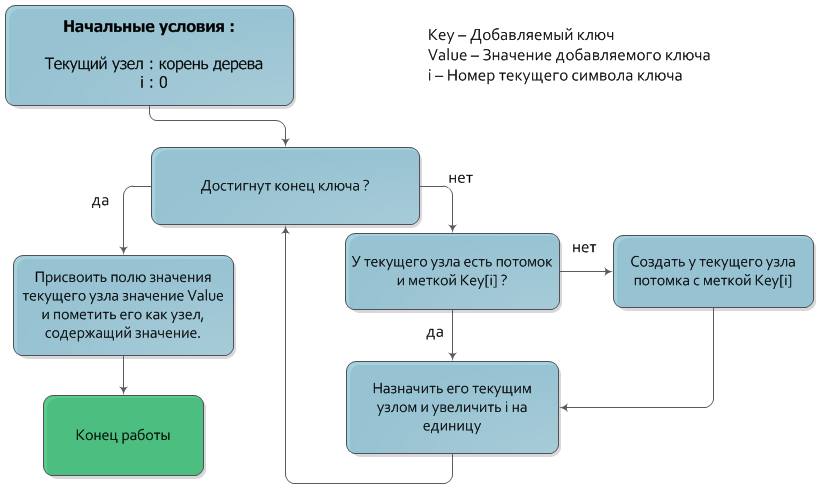
­

Як вже було сказано, ключ, відповідний вузлу - конкатенація міток вузлів, що містяться в шляху від кореня до даного вузла. З цієї властивості необхідно знайти в дереві. Будемо спускатися з кореня дерева на нижні рівні, кожен раз переходячи в вузол, чия мітка збігається з черговим символом ключа. Після того як оброблені всі символи ключа, вузол, в якому зупинився спуск і буде шуканим вузлом. Якщо в процесі спуску не знайшлося вузла з міткою, що відповідає черговому символу ключа, або спуск зупинився на проміжній вершині (вершині, яка не має значення), то шуканий ключ відсутній в дереві.

## 2.3. Вставка

Алгоритм додавання ключа в дерево дуже схожий на алгоритм пошуку.

Часова складність додавання ключа - О (| Key |). Ілюстрація алгоритму вставки на блок-схемі: Нехай дана пара з ключа Key і значення Value, яку потрібно додати. Як і в алгоритмі пошуку ключа, будемо спускатися з кореня дерева на нижні рівні, кожен раз переходячи в вузол, чия мітка збігається з черговим символом ключа. Після того як оброблені всі символи ключа, вузол, в якому зупинився спуск і буде вузлом, яким має бути присвоєно значення Value (також, зрозуміло, вузол повинен бути позначений як має значення). Якщо в процесі спуску відсутній вузол з міткою, що відповідає черговому символу ключа, то слід створити новий проміжний вузол з потрібною міткою і призначити його нащадком поточного. Часова складність додавання ключа - О (| Key |). Ілюстрація алгоритму вставки на блок-схемі:



Нехай дана пара з ключа Key і значення Value, яку потрібно додати. Як і в алгоритмі пошуку ключа, будемо спускатися з кореня дерева на нижні рівні, кожен раз переходячи в вузол, чия мітка збігається з черговим символом ключа. Після того як оброблені всі символи ключа, вузол, в якому зупинився спуск і буде вузлом, яким має бути присвоєно значення Value (також, зрозуміло, вузол повинен бути позначений як має значення). Якщо в процесі спуску відсутній вузол з міткою, що відповідає черговому символу ключа, то слід створити новий проміжний вузол з потрібною міткою і призначити його нащадком поточнаго.

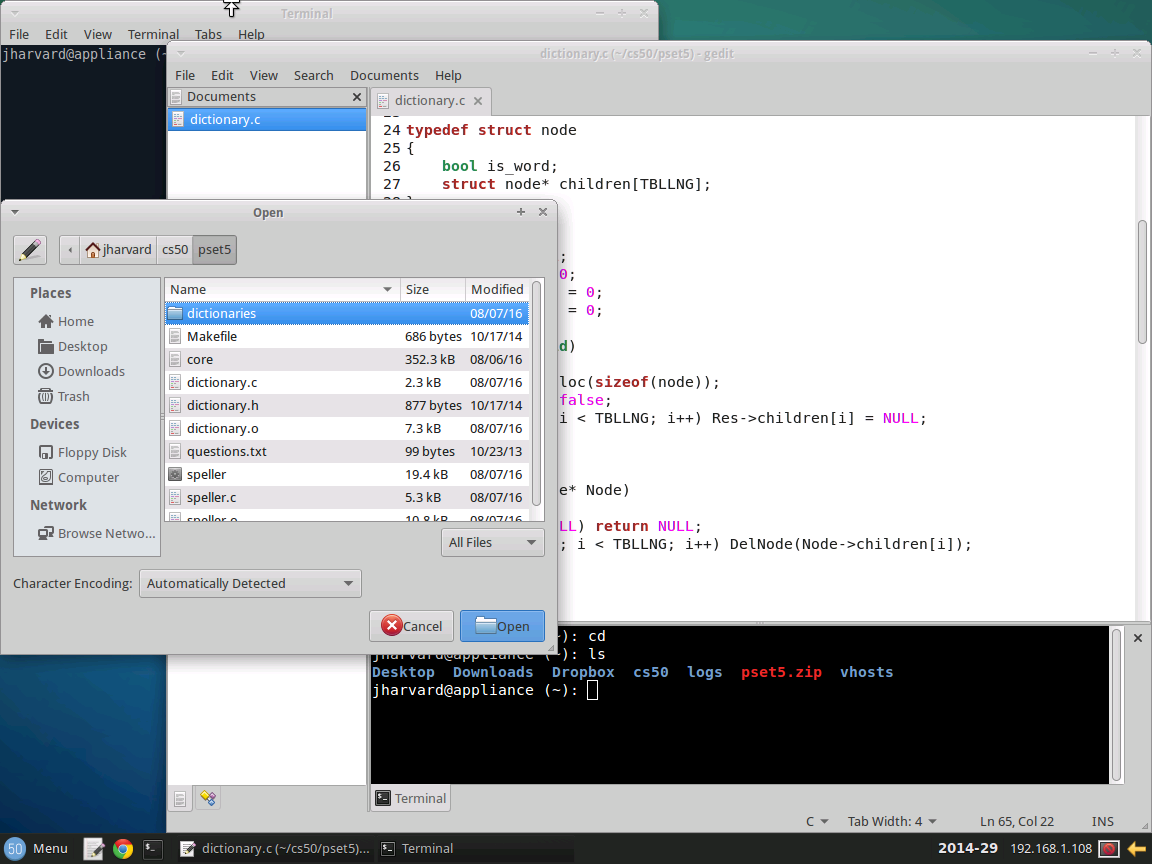
## 2.4. Видалення

Видалення ключа також реалізується дуже легко. Нехай дано ключ Key, який необхідно видалити з дерева. Проведемо пошук цього ключа. Якщо ключ існує в словнику, то знаючи вузол, якому він відповідає, можна просто позначити його як проміжний, зробивши його «невидимим» для подальших пошуків. Після цього можна піднятися від «відключеного» вузла до кореня, попутно видаляючи всі вузли які є листям, проте економія пам'яті в даному випадку є несуттєвою, а для ефективного визначення того, чи є вузол листом потрібно вводити додаткову характеристику вузла. Швидкодія алгоритму видалення пропорційне довжені ключа О (| Key |).

# ЗАСОБИ РЕАЛІЗАЦІЇ ТА ТЕСТУВАННЯ ПОШУКОВОГО МЕХАНІЗМУ ЕЛЕКТРОННОГО СЛОВНИКА

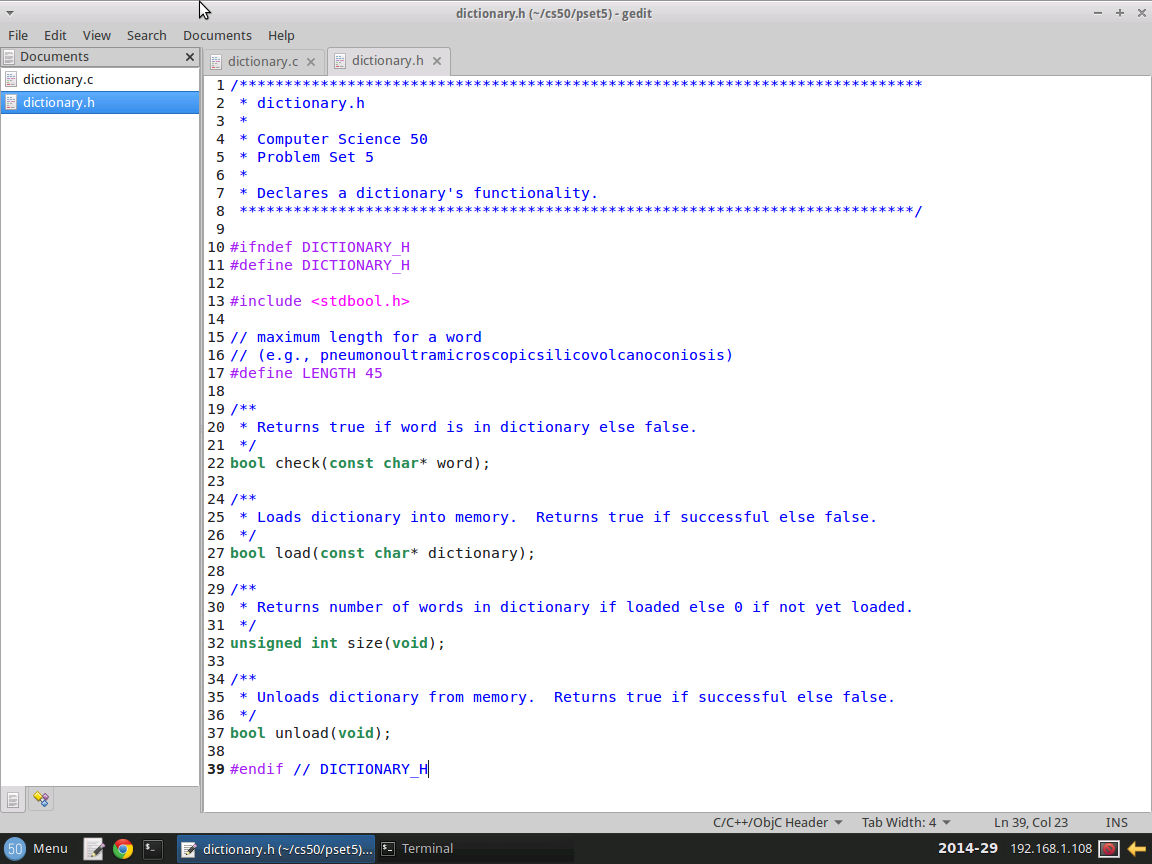
Для перевірки ефективності розробленого нами алгоритму було застосовано застосувати тестуюче програмне забезпечення яке використовують студенти Гарвардського університету для перевірки своїх алгоритмів.

Це програмне забезпечення доступне в віртуальній лабораторії CS50. Віртуальна лабораторія представляє собою віддалений термінал комп’ютер з Linux подібною операційною системою, або образ віртуальної машини яку можно встановити на власному комп’ютері



В своєму розпорядженню командний інтерпретатор команд файлової системи. Універсальний текстовий редактор для написання програмного коду на декількох мовах програмування та компілятори з декількох мов програмування, зокрема С, за допомогою яких програмний код написаний користувачем перетворюється в двійникові файл програми яка може виконуватись на комп’ютері.

Як зазначалося вище програмне забезпечення лабораторії CS50 вимагає підключення спеціалізованої бібліотеки dictionari.c з структурою або з програмним кодом електронного словника. Електронний словник повинен мати стандартні інтерфейси для можливості тестування алгоритмів розроблена бібліотека повинна відповідати стандарту інтерфейсу який задекларований в файлі dictionary.h



Програмний код розробленого електронного словника для інтерактивної перевірки правопису подано нижче:

# РЕАЛІЗАЦІЯ ПОШУКОВОГО МЕХАНІЗМУ ЕЛЕКТРОННОГО СЛОВНИКА НА МОВІ С

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

\* dictionary.c

\*

\* Computer Science 50

\* Problem Set 5

\*

\* Implements a dictionary's functionality.

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

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include "dictionary.h"

#include <ctype.h>

#include <cs50.h>

#include <string.h>

#define TBLLNG 27

/\*\*

\* Returns true if word is in dictionary else false.

\*/

typedef struct node

{

bool is\_word;

struct node\* children[TBLLNG];

}

node;

node\* Root = NULL;

long ListCount = 0;

long NewNodeCount = 0;

long DelNodeCount = 0;

node\* NewNode(void)

{

node\* Res = malloc(sizeof(node));

Res->is\_word = false;

for(int i = 0; i < TBLLNG; i++) Res->children[i] = NULL;

return Res;

}

node\* DelNode(node\* Node)

{

if(Node == NULL) return NULL;

for(int i = 0; i < TBLLNG; i++) DelNode(Node->children[i]);

free(Node);

return NULL;

}

node\* AddChar(node\* Node, const char\* s)

{

// printf("%c", \*s);

int c = TBLLNG-1;

if (Node == NULL) Node = NewNode();

if(\*s != '\'') c = \*s-'a';

if(\*s++ == '\n') Node->is\_word = true;

else Node->children[c]=AddChar(Node->children[c], s++);

return Node;

}

bool NodeCheck(node\* Node, const char\* s)

{

int c = TBLLNG-1;

if (Node == NULL) return false;

if(\*s != '\'') c = tolower(\*s)-'a';

if(\*s++ == '\0') return Node->is\_word;

else return NodeCheck(Node->children[c], s++);

}

bool check(const char\* word)

{

return NodeCheck(Root, word);

}

/\*\*

\* Loads dictionary into memory. Returns true if successful else false.

\*/

bool load(const char\* dictionary)

{

Root = NewNode();

FILE \*dist = fopen(dictionary, "r");

char s[LENGTH+2];

while(fgets(s, LENGTH+2, dist) != NULL)

{

ListCount++;

// printf("%li ", ListCount);

AddChar(Root, s);

}

fclose(dist);

return true;

}

/\*\*

\* Returns number of words in dictionary if loaded else 0 if not yet loaded.

\*/

unsigned int size(void)

{

return ListCount;

}

/\*\*

\* Unloads dictionary from memory. Returns true if successful else false.

\*/

bool unload(void)

{

Root = DelNode(Root);

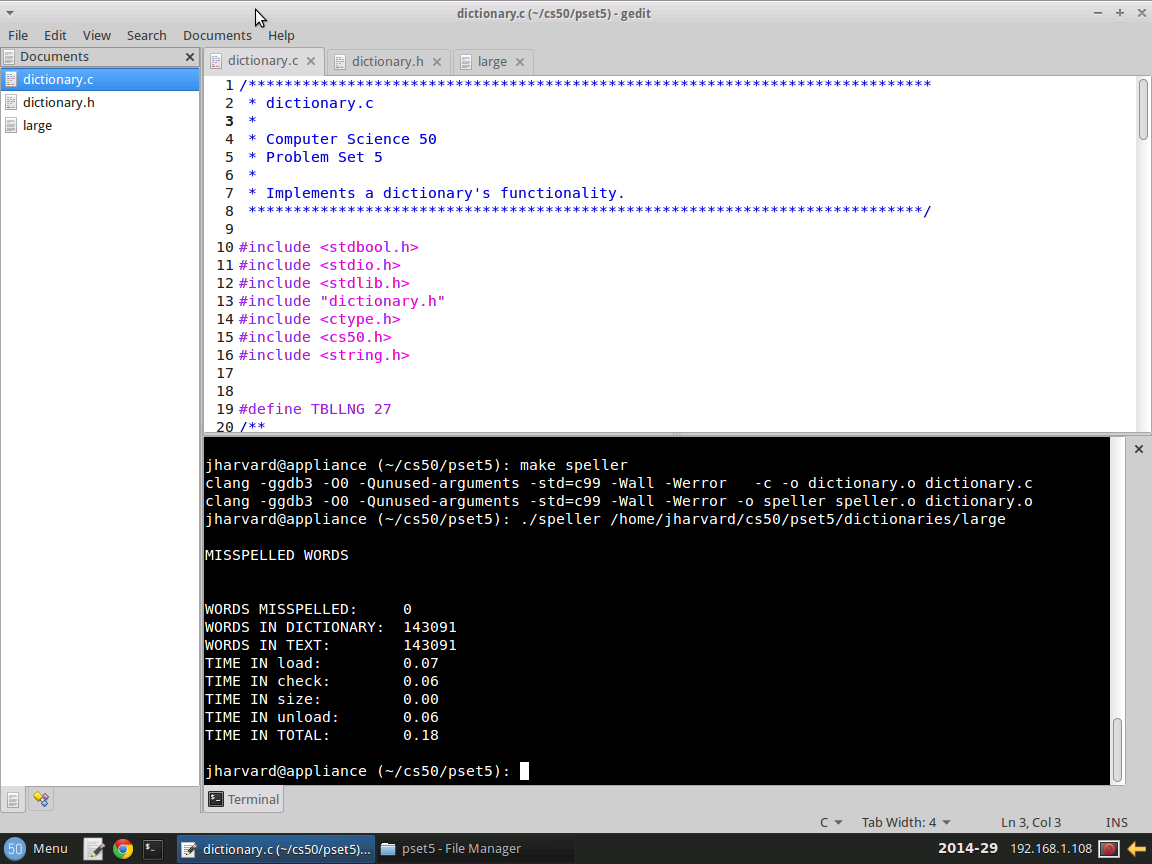
ListCount = 0;

return true;

}

# АПРОБАЦІЯ ПОШУКОВОГО МЕХАНІЗМУ ЕЛЕКТРОННОГО СЛОВНИКА

Для того, щоб перевірити нашу реалізацію пошукового механізму електронного словника, лабораторія CS50 застосовувала декілька текстів, серед яких сценарій фільму "Остін Паверс: International Man of Mystery", звуковий фрагмент Ральфа Віггама, три мільйони байтів Толстого, деякі уривки з творів Макіавеллі і Шекспіра, повна версія Біблії короля Якова та багато інших.



Розроблений програмний засіб пройшов тестування на швидкодію та валідність функціювання в лабораторії Гарвардського університету. Префіксне дерево показало найкращій результат швидкодії

# ВИСНОВОК

У даній роботі була розглянуто створення пошукового механізму електронного словника . Розкриті поняття та особливості , переваги і недоліки різних видів пошукового механізму. Також вивчені основні алгоритми сортування та пошуку інформації.

Були проведені тестування швидкодії, та валідності роботи алгоритмів за допомогою засобів в лаболаторії CS50 Гарвардського університету