1. Министерство высшего образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА №3**

«записная книжка»

1. по дисциплине «Структуры данных»
2. Выполнил
3. студент гр. Белоконь Д. А., Солуянов А. Д.
4. <*подпись*>

Проверил Вагисаров В. Б.

1. <*подпись*>

1. Санкт-Петербург
2. 2023
3. **Цель работы**

Необходимо написать программу, реализующую принцип записной книжки.

Формулировка задания

Необходимо разработать программу, реализующую функционал записной книжки дел:

1) хранение объектов типа "событие":

- год, месяц, число, час, минута события

- день недели

- описание события (текст)

- место события (текст)

2) возможность добавления новых событий

3) возможность удаления событий

4) вывод всех событий на экран:

- возможность сортировки событий по дате

- возможность фильтрации событий по месту

5) возможность сохранения и загрузки событий из файла

Требования:

- день недели должен высчитываться автоматически

- хранение событий должно быть реализовано на основе деревьев бинарного поиска

- сортировка должна быть реализована с помощью обхода дерева

- фильтр должен работать в том числе и по подстрокам

- структура хранения событий должна быть оптимальна по логике и памяти.

Результаты

Была реализована структура данных, хранящая заданные поля и детей(дочерние файлы) такая структура называется деревом. Такая структура данных является сложной в реализации, но крайне эффективной для поставленной задачи, так как при использовании записной книги, она позволяет быстро отыскать нужный раздел для прочтений. Недостатком данной структуры является затраты времени при вставку нового элемента(ветви), поскольку при каждом добавлении элемента происходит балансировка дерева.

Реализация программы началась написания структуры дерева функций добавления(addEvent), удаления(deleteEvent) и балансировки дерева(balanceTree). (Балансировка является самой применяемой функцией так как она применяется после каждого добавления\удаления) . Далее был реализован консольный интерфейс позволяющий взаимодействовать с программой(функция printHelp для вывода текста и printEvent для вывода записанных значений). После чего были реализованы функции сортировки по каждому из параметру данному в задании(compareEventsByDay…) и Алгоритмов поиска элементов дерева по всем указанным признакам(searchInDescription).

Последним были реализованны функции загрузки и выгрузки в файл значений сессии(loadFile для загрузки, writeEvent для выгрузки).

Вывод

В ходе выполнения лабораторной работы была реализована программа на языке Си, которая представляет собой записную книжку. В программе были реализованы следующие функции: добавление новой записи, удаление записи, поиск записи по указанным параметрам, вывод всех записей на экран, а так же загрузка и выгрузка.

Было использовано динамическое выделение памяти для хранения данных, что позволило эффективно управлять памятью и не тратить ее впустую. В результате выполнения лабораторной работы была получена полнофункциональная программа, способная эффективно управлять записями в записной книжке.

**Приложение**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct Event

{

char \*description;

char \*point;

int year;

int month;

int day;

int hour;

int minute;

int ID;

struct Event \*smaller;

struct Event \*larger;

struct Event \*parent;

};

struct Event \*events = NULL;

int globalEventsCounter = 0;

int globalIDCounter = 0;

struct Event \*createEvent(char \*description, char \*point, int year, int month, int day, int hour, int minute)

{

struct Event \*newEvent = (struct Event \*)malloc(sizeof(struct Event));

if (newEvent == NULL)

{

fprintf(stderr, "Could not allocate memory\n");

return NULL;

}

newEvent->description = (char \*)malloc(sizeof(char) \* (strlen(description) + 1));

newEvent->point = (char \*)malloc(sizeof(char) \* (strlen(point) + 1));

newEvent->description = description;

newEvent->point = point;

newEvent->year = year;

newEvent->month = month;

newEvent->day = day;

newEvent->hour = hour;

newEvent->minute = minute;

newEvent->ID = globalIDCounter++;

newEvent->smaller = NULL;

newEvent->larger = NULL;

newEvent->parent = 0x0;

return newEvent;

}

void normiliseRoot()

{

while (events->parent != NULL)

events = events->parent;

}

int compareEventsByDescription(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->year - second->year)

return first->year - second->year;

if (first->month - second->month)

return first->month - second->month;

if (first->day - second->day)

return first->day - second->day;

if (first->hour - second->hour)

return first->hour - second->hour;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByPoint(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (first->year - second->year)

return first->year - second->year;

if (first->month - second->month)

return first->month - second->month;

if (first->day - second->day)

return first->day - second->day;

if (first->hour - second->hour)

return first->hour - second->hour;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByYear(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (first->year - second->year)

return first->year - second->year;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->month - second->month)

return first->month - second->month;

if (first->day - second->day)

return first->day - second->day;

if (first->hour - second->hour)

return first->hour - second->hour;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByMonth(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (first->month - second->month)

return first->month - second->month;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->year - second->year)

return first->year - second->year;

if (first->day - second->day)

return first->day - second->day;

if (first->hour - second->hour)

return first->hour - second->hour;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByDay(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (first->day - second->day)

return first->day - second->day;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->year - second->year)

return first->year - second->year;

if (first->month - second->month)

return first->month - second->month;

if (first->hour - second->hour)

return first->hour - second->hour;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByHour(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (first->hour - second->hour)

return first->hour - second->hour;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->year - second->year)

return first->year - second->year;

if (first->month - second->month)

return first->month - second->month;

if (first->day - second->day)

return first->day - second->day;

if (first->minute - second->minute)

return first->minute - second->minute;

return 0;

}

int compareEventsByMinute(struct Event \*first, struct Event \*second)

{

if (first == NULL || second == NULL)

return 0;

if (first->minute - second->minute)

return first->minute - second->minute;

if (strcmp(first->description, second->description))

return strcmp(first->description, second->description);

if (strcmp(first->point, second->point))

return strcmp(first->point, second->point);

if (first->year - second->year)

return first->year - second->year;

if (first->month - second->month)

return first->month - second->month;

if (first->day - second->day)

return first->day - second->day;

if (first->hour - second->hour)

return first->hour - second->hour;

return 0;

}

int (\*comparisonFunction)(struct Event \*, struct Event \*) = compareEventsByDescription;

void printUsage()

{

return;

}

int checkInput(int argc, char \*argv[])

{

for (int i = 0; i < argc - 1; i++)

{

if (!strcmp(argv[i], "-i") ||

!strcmp(argv[i], "--input"))

return 1;

}

return 0;

}

void boundEvents(struct Event \*child, struct Event \*parent, int side)

{

if (parent != NULL)

{

if (side)

{

parent->larger = child;

}

else

{

parent->smaller = child;

}

}

if (child != NULL)

child->parent = parent;

}

int countLevel(struct Event \*event)

{

if (event == NULL)

return 0;

if (event->larger == NULL && event->smaller == NULL)

return 1;

int resultLarger = event->larger == NULL ? 0 : countLevel(event->larger);

int resultSmaller = event->smaller == NULL ? 0 : countLevel(event->smaller);

return resultLarger > resultSmaller ? resultLarger + 1 : resultSmaller + 1;

}

void balanceTree(struct Event \*event)

{ // A

int resultLarger = countLevel(event->larger);

int resultSmaller = countLevel(event->smaller);

if (abs(resultLarger - resultSmaller) < 2)

return;

struct Event \*largerTree = resultLarger > resultSmaller ? // B

event->larger

: event->smaller;

int countLeft = countLevel(largerTree->smaller);

int countRight = countLevel(largerTree->larger);

if (resultLarger > resultSmaller)

{

if (countLeft < countRight)

{

struct Event \*safeChild = largerTree->smaller;

boundEvents(largerTree, event->parent,

comparisonFunction(event, event->parent) < 0 ? 0 : 1);

boundEvents(safeChild, event, 1);

boundEvents(event, largerTree, 0);

}

else if (countLeft > countRight)

{

// largerTree->smaller -- C

struct Event \*safeLeft = largerTree->smaller->smaller;

struct Event \*safeRight = largerTree->smaller->larger;

boundEvents(largerTree->smaller, event->parent,

comparisonFunction(event, event->parent) < 0 ? 0 : 1);

boundEvents(largerTree, largerTree->smaller, 1);

boundEvents(event, largerTree->smaller, 0);

boundEvents(safeLeft, event, 1);

boundEvents(safeRight, largerTree, 0);

}

}

else

{

if (countLeft > countRight)

{

struct Event \*safeChild = largerTree->larger;

boundEvents(largerTree, event->parent,

comparisonFunction(event, event->parent) < 0 ? 1 : 0);

boundEvents(event, largerTree, 0);

boundEvents(safeChild, event, 1);

}

else if (countLeft < countRight)

{

// largerTree->larger -- C

struct Event \*safeLeft = largerTree->larger->smaller;

struct Event \*safeRight = largerTree->larger->larger;

boundEvents(largerTree->larger, event->parent,

comparisonFunction(event, event->parent) < 0 ? 1 : 0);

boundEvents(largerTree, largerTree->larger, 0);

boundEvents(event, largerTree->larger, 1);

boundEvents(safeLeft, largerTree, 0);

boundEvents(safeRight, event, 1);

}

}

normiliseRoot();

}

void recursiveBalance(struct Event \*event)

{

if (event == NULL)

return;

balanceTree(event);

recursiveBalance(event->smaller);

recursiveBalance(event->larger);

}

void addEvent(struct Event \*event, struct Event \*\*eventPlace)

{

if (\*eventPlace == NULL)

{

\*eventPlace = event;

globalEventsCounter += 1;

return;

}

if (event == NULL)

return;

int result = comparisonFunction(event, \*eventPlace);

if (result > 0)

{

if ((\*eventPlace)->larger == NULL)

event->parent = \*eventPlace;

addEvent(event, &((\*eventPlace)->larger));

}

else if (result < 0)

{

if ((\*eventPlace)->smaller == NULL)

event->parent = \*eventPlace;

addEvent(event, &((\*eventPlace)->smaller));

}

else

{

printf("[Duplicate located: %s]\n", event->description);

}

balanceTree(\*eventPlace);

}

struct Event \*findEvent(struct Event \*event, struct Event \*node)

{

if (node == NULL)

return NULL;

int result = comparisonFunction(event, node);

if (result > 0)

return findEvent(event, node->larger);

else if (result < 0)

return findEvent(event, node->smaller);

else

return node;

}

void deleteEvent(struct Event \*event, struct Event \*node)

{

if (node == NULL)

return;

event = findEvent(event, node);

if (event == NULL)

return;

if (event->parent != NULL)

{

int result = comparisonFunction(event, event->parent);

if (result > 0)

event->parent->larger = NULL;

else

event->parent->smaller = NULL;

}

if (event->larger == NULL)

{

boundEvents(event->smaller, event->parent,

comparisonFunction(event->smaller, event->parent));

if (event == events)

events = event->smaller;

free(event);

globalEventsCounter -= 1;

return;

}

boundEvents(event->larger, event->parent,

comparisonFunction(event->larger, event->parent));

addEvent(event->larger->smaller, &(event->smaller));

boundEvents(event->smaller, event->larger, 0);

if (event == events)

events = event->larger;

if (node == events)

globalEventsCounter -= 1;

free(event);

recursiveBalance(events);

normiliseRoot();

}

void printHelp()

{

printf("\tadd\t\t\t\t\tStart adding an event\n");

printf("\tdelete ID\t\t\t\tDelete an event with the given ID\n");

printf("\tsearch\t\t\t\t\tSearch for an event by the given data\n");

printf("\tprint [sort field\_type]\t\t\tPrint all the events in given order\n");

printf("\tsave file\t\t\t\tSave all the data in a file\n");

printf("\tload file\t\t\t\tLoad all the data from a file\n");

return;

}

void resortEvents(char \*fieldName)

{

if (!strncmp(fieldName, "descriptio", 12))

{

comparisonFunction = compareEventsByDescription;

}

else if (!strncmp(fieldName, "point", 5))

{

comparisonFunction = compareEventsByPoint;

}

else if (!strncmp(fieldName, "year", 4))

{

comparisonFunction = compareEventsByYear;

}

else if (!strncmp(fieldName, "month", 5))

{

comparisonFunction = compareEventsByMonth;

}

else if (!strncmp(fieldName, "day", 3))

{

comparisonFunction = compareEventsByDay;

}

else if (!strncmp(fieldName, "hour", 4))

{

comparisonFunction = compareEventsByHour;

}

else if (!strncmp(fieldName, "minute", 6))

{

comparisonFunction = compareEventsByMinute;

}

else

{

return;

}

struct Event \*oldRoot = events;

events = NULL;

int idCounterSave = globalIDCounter;

while (oldRoot != NULL)

{

struct Event \*event = createEvent(oldRoot->description, oldRoot->point, oldRoot->year,

oldRoot->month, oldRoot->day, oldRoot->hour, oldRoot->minute);

addEvent(event, &events);

event->ID = oldRoot->ID;

struct Event \*save;

if (oldRoot->larger == NULL)

save = oldRoot->smaller;

else

save = oldRoot->larger;

deleteEvent(oldRoot, oldRoot);

oldRoot = save;

if (oldRoot == NULL)

continue;

while (oldRoot->parent != NULL)

oldRoot = oldRoot->parent;

recursiveBalance(oldRoot);

}

globalIDCounter = idCounterSave;

}

void printEvent(struct Event \*event, FILE \*stream)

{

fprintf(stream, "Description: %s\n", event->description);

fprintf(stream, "Point: %s\n", event->point);

fprintf(stream, "Year: %d\n", event->year);

fprintf(stream, "Month: %d\n", event->month);

fprintf(stream, "Day: %d\n", event->day);

fprintf(stream, "Hour: %d\n", event->hour);

fprintf(stream, "Minute: %d\n", event->minute);

if (stream == stdout)

fprintf(stream, "ID: %d\n\n", event->ID);

}

void searchInDescription(struct Event \*event, char \*data)

{

if (event == NULL)

return;

searchInDescription(event->smaller, data);

if (strstr(event->description, data) != NULL)

printEvent(event, stdout);

searchInDescription(event->larger, data);

}

void searchInPoint(struct Event \*event, char \*data)

{

if (event == NULL)

return;

searchInPoint(event->smaller, data);

if (strstr(event->point, data) != NULL)

printEvent(event, stdout);

searchInPoint(event->larger, data);

}

void searchInYear(struct Event \*event, int data)

{

if (event == NULL)

return;

searchInYear(event->smaller, data);

if (event->year == data)

printEvent(event, stdout);

searchInYear(event->larger, data);

}

void searchInMonth(struct Event \*event, int data)

{

if (event == NULL)

return;

searchInMonth(event->smaller, data);

if (event->month == data)

printEvent(event, stdout);

searchInMonth(event->larger, data);

}

void searchInDay(struct Event \*event, int data)

{

if (event == NULL)

return;

searchInDay(event->smaller, data);

if (event->day == data)

printEvent(event, stdout);

searchInDay(event->larger, data);

}

void searchInHour(struct Event \*event, int data)

{

if (event == NULL)

return;

searchInHour(event->smaller, data);

if (event->hour == data)

printEvent(event, stdout);

searchInHour(event->larger, data);

}

void searchInMinute(struct Event \*event, int data)

{

if (event == NULL)

return;

searchInMinute(event->smaller, data);

if (event->minute == data)

printEvent(event, stdout);

searchInMinute(event->larger, data);

}

void searchEvent()

{

char \*fieldName = (char \*)malloc(sizeof(char) \* 13);

printf("Enter field name: ");

fflush(stdout);

fgets(fieldName, 12, stdin);

fieldName[strlen(fieldName) - 1] = 0;

printf("Enter data: ");

fflush(stdout);

if (!strncmp(fieldName, "descriptio", 12))

{

char \*data = (char \*)malloc(sizeof(char) \* (64 \* 1024 \* 1024 + 1));

fgets(data, 64 \* 1024 \* 1024, stdin);

data[strlen(data) - 1] = 0;

searchInDescription(events, data);

free(data);

}

else if (!strncmp(fieldName, "point", 5))

{

char \*data = (char \*)malloc(sizeof(char) \* (64 \* 1024 \* 1024 + 1));

fgets(data, 64 \* 1024 \* 1024, stdin);

data[strlen(data) - 1] = 0;

searchInPoint(events, data);

free(data);

}

else if (!strncmp(fieldName, "year", 4))

{

int data;

scanf("%d", &data);

searchInYear(events, data);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

}

else if (!strncmp(fieldName, "month", 5))

{

int data;

scanf("%d", &data);

searchInMonth(events, data);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

}

else if (!strncmp(fieldName, "day", 3))

{

int data;

scanf("%d", &data);

searchInDay(events, data);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

}

else if (!strncmp(fieldName, "hour", 4))

{

int data;

scanf("%d", &data);

searchInHour(events, data);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

}

else if (!strncmp(fieldName, "minute", 6))

{

int data;

scanf("%d", &data);

searchInMinute(events, data);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

}

free(fieldName);

}

struct Event \*searchID(struct Event \*event, int ID)

{

if (event == NULL)

return NULL;

if (event->ID == ID)

return event;

struct Event \*resultSmaller = searchID(event->smaller, ID);

if (resultSmaller != NULL)

return resultSmaller;

return searchID(event->larger, ID);

}

void deleteID()

{

printf("Print ID: ");

fflush(stdout);

int ID;

scanf("%d", &ID);

char c;

while ((c = fgetc(stdin)) != EOF && (c != '\n'))

;

struct Event \*event = searchID(events, ID);

if (event == NULL)

printf("Could not find the event\n");

else

deleteEvent(event, events);

}

void loadFile(char \*fileName)

{

FILE \*inputFile = fopen(fileName, "r");

if (inputFile == NULL)

{

fprintf(stderr, "Could not open input file\n");

return;

}

char \*buffer = (char \*)malloc(sizeof(char) \* (64 \* 1024 \* 1024 + 1));

while (!feof(inputFile))

{

char \*description = NULL, \*point = NULL;

int month = 0, year = 2022, day = 0, hour = -1, minute = -1;

for (int i = 0; i < 7; i++)

{

fgets(buffer, 64 \* 1024 \* 1024, inputFile);

buffer[strlen(buffer) - 1] = 0;

if (feof(inputFile))

break;

switch (i)

{

case 0:

description = (char \*)malloc(sizeof(char) \* strlen(buffer));

strncpy(description, buffer, strlen(buffer));

break;

case 1:

point = (char \*)malloc(sizeof(char) \* strlen(buffer));

strncpy(point, buffer, strlen(buffer));

break;

case 2:

year = atoi(buffer);

break;

case 3:

month = atoi(buffer);

break;

case 4:

day = atoi(buffer);

break;

case 5:

hour = atoi(buffer);

break;

case 6:

minute = atoi(buffer);

break;

}

}

if (description == NULL || point == NULL || year == 2022 || month == -1 || day == -1 || hour == -1 || minute == -1)

break;

struct Event \*newEvent = createEvent(description, point, year, month, day, hour, minute);

addEvent(newEvent, &events);

printEvent(newEvent, stdout);

}

fclose(inputFile);

free(buffer);

return;

}

void writeEvent(struct Event \*event, FILE \*outputFile)

{

if (outputFile == NULL || event == NULL)

return;

fprintf(outputFile, "%s\n", event->description);

fprintf(outputFile, "%s\n", event->point);

fprintf(outputFile, "%d\n", event->year);

fprintf(outputFile, "%d\n", event->month);

fprintf(outputFile, "%d\n", event->day);

fprintf(outputFile, "%d\n", event->hour);

fprintf(outputFile, "%d\n", event->minute);

}

void wanderTreeLUR(struct Event \*event, FILE \*stream, void (\*func)(struct Event \*, FILE \*))

{

if (event == NULL)

return;

wanderTreeLUR(event->smaller, stream, func);

func(event, stream);

wanderTreeLUR(event->larger, stream, func);

}

void writeFile(char \*fileName)

{

FILE \*outputFile = fopen(fileName, "w");

if (outputFile == NULL)

{

fprintf(stderr, "Could not open output file\n");

return;

}

wanderTreeLUR(events, outputFile, writeEvent);

fclose(outputFile);

}

void parseEvent()

{

char \*buffer = (char \*)malloc(sizeof(char) \* (64 \* 1024 \* 1024 + 1));

char \*description, \*point;

int month, day, year, hour, minute;

for (int i = 0; i < 7; i++)

{

switch (i)

{

case 0:

printf("Description: ");

break;

case 1:

printf("Point: ");

break;

case 2:

printf("Year: ");

break;

case 3:

printf("Month: ");

break;

case 4:

printf("Day: ");

break;

case 5:

printf("Hour: ");

break;

case 6:

printf("Minute: ");

break;

}

fflush(stdout);

fgets(buffer, 64 \* 1024 \* 1024, stdin);

buffer[strlen(buffer) - 1] = 0;

switch (i)

{

case 0:

description = (char \*)malloc(sizeof(char) \* strlen(buffer));

strncpy(description, buffer, strlen(buffer));

break;

case 1:

point = (char \*)malloc(sizeof(char) \* strlen(buffer));

strncpy(point, buffer, strlen(buffer));

break;

case 2:

year = atoi(buffer);

break;

case 3:

month = atoi(buffer);

break;

case 4:

day = atoi(buffer);

break;

case 5:

hour = atoi(buffer);

break;

case 6:

minute = atoi(buffer);

break;

}

}

if (description == NULL || point == NULL || month == -1 || day == -1 || year == -300 || hour == -1

|| minute == -1)

return;

struct Event \*newEvent = createEvent(description, point, year, month, day, hour, minute);

addEvent(newEvent, &events);

printf("\nNEW ID IS -> %d\n", newEvent->ID);

}

void enterInterectiveMode()

{

printHelp();

char command[65];

while (1)

{

printf("\n> ");

fflush(stdout);

fgets(command, 65, stdin);

command[strlen(command) - 1] = 0;

if (!strncmp(command, "load", 4))

{

if (strlen(command) < 6)

{

printHelp();

continue;

}

char fileName[60];

strncpy(fileName, command + 5, strlen(command) - 5);

loadFile(fileName);

}

else if (!strncmp(command, "add", 3))

{

parseEvent();

}

else if (!strncmp(command, "print", 5))

{

if (strlen(command) > 6)

{

char fieldName[12];

strncpy(fieldName, command + 6, strlen(command) - 6);

resortEvents(fieldName);

}

wanderTreeLUR(events, stdout, printEvent);

}

else if (!strncmp(command, "save", 4))

{

if (strlen(command) < 6)

{

printHelp();

continue;

}

char fileName[60];

strncpy(fileName, command + 5, strlen(command) - 5);

writeFile(fileName);

}

else if (!strncmp(command, "search", 6))

{

searchEvent();

}

else if (!strncmp(command, "delete", 6))

{

deleteID();

}

else

{

printHelp();

}

}

}

int main(int argc, char \*argv[])

{

if (argc < 2)

{

enterInterectiveMode();

return 1;

}

return 0;

}