Първи проект по ООП

Тема 5 – Личен календар

Стоян Димитров Малинин, 8-ма група, 0MI0800080

Връзка с GitHub - https://github.com/StoyanMalinin/PersonalCalendar

**Описание на задачата**

Целта на този проект е да се създаде работещо приложение, което да работи като календар, който следи ангажиментите на потребителите. Също така трябва да се реализира и записване във файлове с цел информация да остава и между пусканията на програмите. Функционалността трябва да съдържа – добавяне и премахване на ангажименти, търсене на свободно място, търсене на ангажименти по някакви критерии, както и изкарване на някаква статистика за натовареността.

**Подход за решение**

В общи линии, функционалността на програмата е имплементирана, следвайки описанието на задачата. Техническа особеност е, че за удобство и производителност е взето решение срещите винаги да се пазят в сортиран ред спрямо началните им дати. Това много улеснява работата с тях, но поражда проблеми при поддържането на това свойство. По тази причина, за да се избегне постоянното презаписване на файла данните се пазят, грубо казано, на две места – голяма и малка част. В голямата част са записани срещи в сортиран ред, а в малката – срещи, които трябва да бъдат добавяни или премахвани, също в сортиран ред. При натрупване на твърде много елементи в „малката част“, промените се опресняват. Повече информация ще има в секцията за имплементация.

**Архитектура на приложението**

**Обекти**

Приложението е изградено, стриктно спазвайки ООП парадигмата. Това значи, че функционалността е разделена между отделни обекти.

* String – отговаря за работата със стрингове като абстрахира работата с динамична памет
* Time – поддържа информация за момент от времето с точност до минута
* Meeting – съдържа информация за една среща. По-конкретно: начално време, продължителност, заглавие и описание. Сравнява се спрямо началното си време. Също така поддържа стандарт за четене и записване в двоичен файл, който да бъде удобен и да не използва прекалено много памет.
* CalendarDbFileManager – изпълнява ролята на контролер на ниско ниво за „базата данни“. Поддържа основните операции: добавяне, премахване, четене в различни форми и проверка дали среща е премахната. Тук се съдържа цялата работа с файлове и респективно „опасните“ операции.
* CalendarDatabase – разширява функционалността на CalendarDbFileManager. Основно се използва за различни видове търсене, повечето от които се възползват от факта, че срещите са записани в сортиран вид. Тук с имплементирани и основните заявки, дадени от условието – извеждане на дневна програма, промяна на среща, търсене на среща по низ, извеждане на статистика за натовареност и намиране на свободен час.
* UIHandler – поддържа потребителския интерфейс и дава възможност на потребителя да контролира приложението. Също така се грижи и за обработката на грешки.

**Конвенция за exception-и**

Повечето грешки в приложението се обработват чрез exception-и. Като конвенция се приема, че грешките, които не са фатални за програмата и са просто резултат от некоректни подадени данни, се индикират с std::logic\_error. От друга страна, грешките, след които програмата не може да продължи да работи коректно (предимно проблемите с файлове) се индикират с std::exception. Всяка хваната друга грешкасе счита за фатална.

**Конвенция за работа с Meeting обекти**

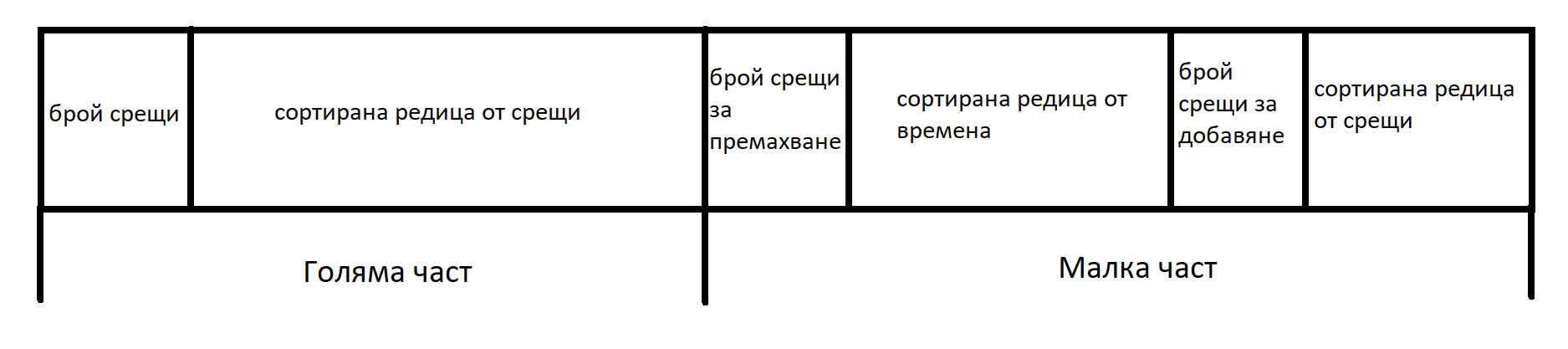
Тъй като препокриването на срещи е забранено, то сравненията на срещи се базират изцяло на тяхното начално време. Важно е да се отбележи, че се допуска, че две срещи са равни, ако имат едно и също начално време. Това се прави с цел улеснение работата, понеже създаването на отделна система за раздаване на id-та би усложнило проекта.

**Реализация**

**Записване във файл**

Тъй като от приложението се изисква да записва информацията, която получава във файлове, то трябва да се вземе решение за това какъв да бъде форматът. Взето е решение да се работи с двоични файлове, заради лесния достъп до данните, които се намират далеч един от друг. Както бе споменато по-горе, срещите се поддържат записани в сортиран вид. Тъй като основна част от приложението е добавянето и премахването на срещи, то това създава проблеми с бързодействието на програмата, тъй като поддържането на сортирана последователност от елементи във файл е до голяма степен еквивалентна на тази за поддържане на сортирана последователност от елементи в RAM. Тук най-популярния подход за справяне с този проблем (балансирани двоични дървета) е много трудно приложим, понеже трябва да се направи някаква имплементация на работа с динамична памет във файлове. По тази причина е взето решението да се използва по прост, но също доста ефективен вариант. Той е описан грубо в секцията за **Подход за решение**, но тук ще бъде описан в повече детайли.

Структурата на файла е описана на високо ниво от следното изображение:



Ключово е да се отбележи, че малката част е поставена в края на файла, за да може лесно да се моделира и презаписва. Както вече казахме, работата с този вид файлове се извършва от обектите от класа CalendarDbFileManager. За повече яснота по имплементацията му следва списък със семантиката на член-данните му.

* const size\_t MAX\_POSPONED – константа, указваща максималния брой на отложени промени (брой на елементи в малката част)
* std::fstream f – връзка с файла
* size\_t meetingCnt – брой срещи от голямата част
* size\_t\* meetingPtrs – указатели към началата на секторите, в които се намира всяка една среща от голямата част. Това нещо е нужно, понеже обект от вид Meeting е с неопределена големина, понеже няма горна граница за размерите на заглавието и описанието.
* size\_t postponedStartPtr – начало на малката част или с други думи: на частта, в която започват отложените промени
* size\_t toRemCnt – брой на срещи за премахване
* Time\* toRem[MAX\_POSPONED + 1] – сортирана редица от времена, които представляват срещите, които трябва да бъдат премахнати.
* size\_t toAddCnt – брой срещи за добавяне
* Meeting\* toAdd[MAX\_POSPONED + 1] – сортирана редица от срещи, които трябва да бъдат добавени

При създаване на обект от вида CalendarDbFileManager данните от файла, към който се насочва не се зареждат, понеже това е функция, която може да хвърли exception, което не е добре да се случва в конструктор. Затова е взето решение тази част да се остави на потребителя. Той може да използва функцията load. Тя ходи по файла и се възползва от структурата му, за да зареди нужните данни в оперативната памет, а също така и следи да не би файлът да е проблемен. Обектите от класа CalendarDatabase работят на същия принцип – отново ръчно трябва да бъде извикана функцията load, за да се заредят данните.

При добавяне или премахване на среща (чрез функциите addMeeting и remMeeting) промените се случват само и единствено в оперативната памет. Тези промени могат да бъдат записани във файла чрез функцията saveChanges, която се вика автоматично при разрушаване на обект от вида CalendarDbFileManager, но също може да бъде извикана и ръчно. При индикация, че са се натрупали твърде много промени малката част се изчиства и промените влизат в сила. За това се грижи функцията updatePostponedChanges.

Тази функция целта на тази функция е да преработи файла така, че всички срещи да се намират в голямата част, а малката част да бъде празна. За целта тя използва помощен файл, за да сортира всички срещи. Това се прави с цел да не се зареждат всички срещи в RAM, понеже се очаква, че те могат да са много. Цялото нещо представлява сливане на две сортирани редици (тази на срещите от голямата част и тази на срещите за добавяне от малката част), за което се използва greedy алгоритъма от merge sort, като се следи да не би някоя среща да е вече премахната, за което се използва функцията checkIfRemoved. Тя е базирана на обикновено двоично търсене. Така сливането става линейно по броя на всички срещи, ако броим проверката за това дали среща е премахната като константна, понеже тя зависи от стойността на MAX\_POSTPONED.

Също така е добре да се отбележи какъв е формата на записване на останалите обекти от програмата в двоични файлове. Обектите от тип Time се записват точно по начина, по който седят в RAM, понеже те са статични и нямат връзка с външни ресурси. Обектите от тип Meeting се записват по по-специфичен начин не само защото имат вътре в себе си динамична памет (дори напротив тя е „скрита“ в класа String), а и защото трябва да бъдат така направени, че лесно да се „прескачат“, тоест за много малко операции да може да се измести get-указателя във файла, сочещ към началото на обект от тип Meeting, към първия байт след края на обекта. По тази причина е избрана следната структура:



\*стринговете се пазят без терминиращата нула, понеже това не е нужно

**Имплементация на заявките**

* Запазване и отменяне на час – поддържат масивите toRem, toAdd като следят за колизии
* Промяна на среща – изпълнява се като се опитва дали може да се премахне старата версия на срещата и да се добави новата без да се получават колизии
* Извеждане на дневна програма – използва се функцията, която печата всички срещи в даден времеви диапазон - printRangeReport
* Търсене на среща – минава се по всички срещи и се използва функция findSubstr за проверка дали търсения низ е открит
* Натовареност – използва се getRangeReport за намиране на всички срещи в интервал и след това се прави статистика по дните от седмицата
* Намиране на свободен час – отновно чрез getRangeReport се извличат всички срещи и се търсят свободни места между тях.

**Имплементация на помощните функции**

* CalendarDatabase::getRangeReport – с двоично търсене определя откъде трябва да почне да търси в голямата и малката част и по подобен начин на функцията updatePostponedChanges генерира списък от сортирани срещи, които не са премахнати.
* CalendarDatabase::printRangeReport – използва CalendarDatabase::getRangeReport.
* String::findSubstr – използва алгоритъма на Knuth–Morris–Pratt, за провери дали даден шаблон е подстринг

**Тестване**

По-малките и независими компоненти от програмата могат да се тестват чрез предварително направените unit тестове. Това са класовете Time, String и Meeting. Нещата, които работят с външни ресурси като контролерите за бази данни и нещата за потребителски интерфейс се тестват само с предварително зададени данни, които на ръка трябва да се проверяват за коректност.

**Демонстрация**

Файлът **demo.db** е предварително попълнена „база“ данни. В нея се съдържат следните срещи

|  |  |  |  |
| --- | --- | --- | --- |
| **StartTime** | **Duration** | **Title** | **Description** |
| 09:00 25.05.2022 | 120 mins | oop test | second oop test for the semester |
| 12:35 16.05.202 | 35 mins | walk in the park | go to the park for a walk |
| 09:00 14.05.2022 | 240 mins | calculus test | second calculus test for the semester |
| 08:00 26.05.2022 | 30 mins | wake up and prepare | wake up and pack my luggage so I can catch the train and go home |

**Подобрения**

Както бе споменато по-рано, за съхраняването на данните не се използва много класически подход, което значи, че в някакъв смисъл сме ограничени от това да добавяме нова функционалност към проекта. Би било добре да се използва нещо, което наподобява повече бази данни за по-добра абстракция и производителност. Друго нещо, което би било хубаво да се добави като опция е някакъв вид компресия, понеже всички тези оптимизации предполагат работа с големи ресурси от данни. Също така за бъдеща разработка ще е по-добре, ако може да се промени тестването. В този си вид проекта може да се каже, че е оптимизиран за заявките добавяне, премахване и търсене по време, но не е напълно оптимален за търсене по стринг. Ако се окаже, че този тип заявки ще са преобладаващите, могат да се използват по-добри алгоритми за стрингове, които да постигнат по-добра производителност в този аспект за сметка на останалите.

**Заключение**

Проекта изпълнява успешно функциите, които са изискани от заданието. Той съдържа в себе си и допълнителни оптимизации, които биха били полезни в някои специфични случаи. Поведението му може до голяма степен да се контролира от константата MAX\_POSTPONED, разбира се с цената на това, че трябва да се прекомпилира.