KATEDRA INFORMATIKY
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
UNIVERZITA KOMENSKÉHO, BRATISLAVA

IMPLEMENTÁCIA PREHĽADÁVANIA V DÚHOVÝCH TABUĽKÁCH

(bakalárska práca)

PETER GAZDÍK

Vedúci: RNDr. Martin Stanek, PhD.

Bratislava, 2007

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . . . .

Abstrakt

Dúhové tabuľky predstavujú kryptoanalytickú metódu nazvanú podľa rovnomennej dátovej štruktúry. Jej autorom je Philippe Oechslin, ktorý nadviazal na prácu Martina Hellmana. Tieto tabuľky sa dajú využiť napríklad na hľadanie kľúčov použitých pri šifrovaní, v prípade, že otvorený aj šifrový text sú známe. Hlavným cieľom mojej bakalárskej práce bolo implementovať knižnicu, ktorá umožní s týmito tabuľkami pracovať. Okrem vytvárania a prehľadávanie umožňuje meniť množstvo parametrov, vrátane použitej šifrovacej funkcie, a umožňuje sledovať výkonové parametre tabuliek. Na demonštráciu funkčnosti tejto knižnice som vytvoril jednoduchú aplikáciu, ktorá nad knižnicou pracuje. V úvode práce, pozostávajúcej z celkovo piatich kapitol, uvádzam popis dúhových tabuliek a možnosti ich využitia. Ďalšie kapitoly sú venované samotnej knižnici a obsahujú, okrem iného, vysokoúrovňový popis implementácie a špecifikáciu parametrov funkcií z tejto knižnice. Štvrtá kapitola pojednáva o spomínanej aplikácii. Posledná kapitola, ktorá slúži na ilustráciu výkonu knižnice, obsahuje údaje o čase výpočtu dôležitých častí funkcie vytvárania tabuliek.

Kľúčové slová: kryptoanalýza, dátová štruktúra, implementácia

Obsah

Kapitola 1

Dúhové tabuľky

1.1 Všeobecný úvod

Medzi základné úlohy kryptológie patrí šifrovanie textu a kontrola prístupových práv. V oboch týchto prípadoch ide o to, že ľudia, ktorí sa k dátam dostať nemajú, sa k nim ani nedostanú, ale súčasne tí, ktorí sa k nim dostať majú, sa aj dostanú. To sa dosahuje tak, že autorizovaní majú akúsi informáciu (kľúč, heslo), s ktorou môžu text dešifrovať, získať prístup do systému a podobne. O tejto informácii možno obvykle povedať, že je to prvok z nejakej konečnej množiny. A to je to miesto, na ktoré sa dá zaútočiť. Dve základné metódy ako na to sú postupné prehľadávanie priestoru kľúčov priamo pri útoku, alebo robenie výpočtov dopredu, a vypočítané dáta použiť na rýchlejšie riešenie konkrétnej inštancie daného problému. Prvá možnosť je náročná na výpočtovú silu, druhá zase na úložný priestor. Kryptografické systémy sú teda robené tak, aby prehľadávanie trvalo dlho, respektíve aby priestor pre vypočítané dáta bol veľmi veľký. Samozrejme, hovoríme o hodnotách času a priestoru, pre ktoré by pravdepodobnosť úspechu bola dostatočne veľká. Samotné dúhové tabuľky predstavujú kryptoanalytickú metódu, ktorá ponúka kompromis medzi veľkosťou uložených dát a časom, potrebným priamo na útok.

1.2 Použitie

V prípade, že hovoríme o útoku na šifrovaný text, ponúka sa otázka, aké výpočty môžeme robiť dopredu. Občas sa však stane, že otvorený text, respektíve časť z neho, poznáme. Napríklad možno predpokladať, že úvod ”C-čkovského” zdrojového súboru tvorí text ’#include <stdio.h>’, alebo niektoré formáty súborov (wordovský dokument) majú prvých niekoľko bajtov svojho obsahu vždy rovnakých. V prípade, že text bol šifrovaný pomocou blokovej šifry, je možné z týchto

blokov získať kľúč a potom dešifrovať celý text.

Ďalším príkladom je hashovanie hesiel. Niektoré bežne používané systémy hashujú heslá tak, že fixný otvorený text zašifrujú, pričom samotné heslo použijú ako kľúč. To je situácia, pre ktorú sú dúhové tabuľky vhodné. Vypočítané dáta sa dajú použiť na väčšie množstvo útokov a efektivita s použitím dúhových tabuliek je vysoká.

Ideálnym kandidátom na takýto druh útoku je napríklad "Windows NT LanManger". Ten zadané heslo doplní nulami na veľkosť 14 bajtov. Malé písmená zmení na veľké. Rozdelí tých 14 bajtov na dve 7 bajtové slová, a každé z týchto slov sa použije ako kľúč pri šifrovaní fixného otvoreného textu (vždy toho istého) pomocou DES. Takto vzniknú dve 8 bajtové slová, ktoré spolu tvoria hash hesla.

1.3 Pôvodná metóda

Metóda, na ktorej sú založené dúhové tabuľky, a ktorá umožnila robiť kompromis medzi množstvom uložených dát a časom útok, bola publikovaná už v roku 1980 Martinom Hellmanom [2]. Túto metódu opíšem na príklade šifrovania otvoreného textu. Chceme získať kľúč k s dĺžkou d, pričom otvorený text P_0 dĺžky m a šifrový text C dĺžky c sú známe. Šifrovacia funkcia S, taká že:

$$S : \{0,1\}^m \times \{0,1\}^d \rightarrow \{0,1\}^c$$

je takisto známa. Aby sme pri samotnom útoku ušetrili čas, snažíme sa uložiť si čo najviac dvojíc kľúč - šifrový text. Základná myšlienka, ktorá nám umožní šetriť miesto, spočíva v tom, že pomocou jednej dvojice kľúč - šifrový text reprezentujeme viacero takýchto dvojíc naraz v takzvaných reťaziach, teda postupnostiach tvorených kľúčmi a šifrovými textami. Ako na to?

Najprv si zvolíme jeden kľúč, ktorý bude predstavovať začiatok reťaze. Označme ho k_0. Aplikujeme šifrovaciu funkciu, teda známy otvorený text zašifrujeme s použitím nášho kľúča. Dostávame teda:

$$C_0 = S(P_0, k_0)$$

Namiesto toho, aby sme si uložili priamo túto dvojicu, na vypočítaný šifrový text aplikujeme redukčnú funkciu R, kde:

$$R : \{0,1\}^c \rightarrow \{0,1\}^d$$

Jedná sa teda o funkciu, ktorá nám zo šifrového textu vyrobí nejaký nový kľúč. Dostávame:

$$k_1 = R(C_0)$$

Takýmto spôsobom striedavo aplikujeme šifrovaciu a redukčnú funkciu l krát, pričom v poslednom kroku redukčnú funkciu vynecháme. Dostávame postupnosť $k_0, C_0, ..., k_l, C_l$, kde $C_i = S(P_0, k_i)$ a $k_{i+1} = R(C_i)$. Do našej tabuľky si ale uložíme len prvý a posledný element tejto postupnosti, teda k_0 a C_l. O dvojici k_i, C_i budeme hovoriť, že sa nachádza v i-tom stĺpci tabuľky. Dĺžka takejto reťaze je $l + 1$.

1.4 Dúhové tabuľky

V takomto vytváraní reťazí sa však skrýva nasledujúci problém. Občas je hodnota redukčnej funkcie rovnaká pre dva rôzne šifrové texty. Takúto situáciu budeme volať kolízia. Pre hore opísaný postup by kolízia v dvoch rôznych reťaziach znamenala, že od pozície, v ktorej kolízia nastala v jednej reťazi, by sme pokračovali rovnakou postupnosťou kľúčou a šifrových textov ako v druhej reťazi. Došlo by teda k splynutiu týchto dvoch reťazí (*merge*). To je situácia ktorej sa chceme vyhnúť, pretože to znamená, že naša tabuľka pokrýva menšie množstvo kľúčov, než by sme chceli. Podstatné zlepšenie v tomto smere predstavujú takzvané dúhové tabuľky, ktoré v [1] predstavil Philippe Oechslin. Tieto tabuľky pri vytváraní reťazí používajú v každom kroku inú redukčnú funkciu, a teda výrazne znižujú pravdepodobnosť, že takéto splynutie nastane. Na to by totiž kolízia musela nastať v oboch reťaziach v tom istom kroku (respektíve stĺpci). Redukčná funkcia je teda obohatená o ďalší parameter predstavujúci stĺpec tabuľky. Redukčnú funkciu, ktorú aplikujeme na šifrový text C_i aby sme dostali kľúč k_{i+1} označíme R_{i+1}.

1.5 Prehľadávanie

Ako teraz nájsť hľadaný kľúč v tabuľke? Predpokladajme, že máme daný šifrový text C a tabuľku s reťazami dĺžky $l + 1$. Pokúsme sa nájsť v tabuľke reťaz, ktorej koniec tvorí náš text. Samozrejme, tabuľka je utriedená podľa koncov reťazí, použijeme teda metódu binárneho vyhľadávanie. V prípade, že nájdeme reťaz, ktorej koniec sa s naším textom zhoduje, z kľúča, ktorý tvorí začiatok reťaze vieme celú reťaz zrekonštruovať a príslušný kľúč nájsť. Bude to totiž ten, ktorý v reťazi predchádza náš text. Ak sa šifrový text v tabuľke nenachádza, aplikujeme naň redukčnú funkciu R_l a následne spravíme šifrovanie. Tým sme sa v podstate posunuli v našej tabuľke o jeden stĺpec doľava. Získaný šifrový text sa snažíme nájsť v tabuľke, ak ho nájdeme, znova zrekonštruujeme reťaz, tentokrát sa ale zastavíme o jeden krok skôr, teda v stĺpci $l - 1$. Takto postupne prehľadávame všetky stĺpce od posledného k prvému.

1.6 Falošný poplach

Prehľadávanie teda spočíva v tom, že postupne generujeme konce reťazí, v ktorých sa náš šifrový text nachádza postupne na poslednom, predposlednom, a tak ďalej, až prvom mieste. Treba si však dať pozor na jednu vec. To, že sme pri prehľadávaní niektorého stĺpca našli príslušný koniec reťaze v našej tabuľke ešte neznamená, že tabuľka náš šifrový text obsahuje. Našli sme len dve reťaze, ktoré majú rovnaký koniec. Problém je v tom, že niekde medzi našim šifrovým textom a koncom vygenerovanej reťaze mohlo dôjsť ku kolízii s reťazou z tabuľky a teda nami hľadaný kľúč v tej reťazi nenájdeme. Takejto situácii hovoríme falošný poplach.

Kapitola 2

Implementácia

V tejto kapitole si podrobnejšie predstavíme implementáciu knižnice ”rainbowtables”, presnejšie vysvetlíme význam niektorých parametrov a zamyslíme sa aké hodnoty je pre ne vhodné zvoliť. V prípade, že existuje viacero možných riešení daného problému sa budem snažiť zdôvodniť voľbu konkrétneho riešenia. V jednom prípade, kde sú implementované dve možnosti, si uvedieme analýzu problému a návrh, kedy si ktorú z možností vybrať.

2.1 Vytváranie tabuliek

Popíšme si postup, ako knižnica vytvára vlastné tabuľky. Poznamenajme, že počas vytvárania tabuľky budeme používať viacero súborov. Okrem súboru pre výslednú tabuľku s názvom ”<table_name>.rbt” aj niektoré ďalšie, ktorých názvy sú uvedené v prílohe. Je preto dôležité, aby adresár, v ktorom pracujeme, neobsahoval súbory s niektorým z týchto názvov, pretože ich obsah bude prepísaný.

Na vstupe dostaneme viacero vstupných parametrov, z ktorých sú pre nás v tejto chvíli dôležité najmä veľkosť tabuľky, označme ju n, a veľkosť operačnej pamäte, označme a. Ďalšou dôležitou premennou je ”strict_dupl_detection”. Ak je jej hodnota nenulová, dve reťaze sa považujú za rovnaké, ak majú rovnaký koniec. Teda na základe tohto parametra si zvolíme porovnávaciu funkciu, nazvime ju *compare*, ktorá nám povie, či sú dve reťaze rovnaké alebo nie. Podľa zvolenej hodnoty ”strict_dupl_detection” porovná buď začiatky[1], alebo konce reťazí. O reťaziach R_1 a R_2 budeme hovoriť ako o duplikátoch, ak ich funkcia *compare* vyhodnotí ako rovnaké. Ak teda chceme vytvoriť tabuľku o veľkosti n, znamená to okrem iného, že chceme n reťazí, z ktorých žiadne dve nie sú duplikáty. Pod zahadzovaním duplikátov rozumieme skutočnosť, že zahodíme práve jednu z reťazí R_1, R_2.

[1] Ak je hodnota ”strict_dupl_detection” rovná nule.

Vysokoúrovňový popis funkcie vytvárania tabuliek vyzerá asi takto. Na úvod vygenerujeme n reťazí. Samozrejme počítame s možnosťou, že sa nám nezmestia do operačnej pamäte všetky naraz (teda že $n > a$). Vytvoríme teda $\lceil \frac{n}{a} \rceil$ blokov reťazí o veľkosti a (v prípade, že n nie je násobok a bude jeden z nich menší). Jednotlivé bloky utriedime, a uložíme ich za sebou do pomocného súboru. Na triedenie používam štandardnú funkciu jazyka C "qsort()". Následne tieto bloky spájame tak, aby sme nakoniec dostali jeden veľký utriedený blok. Tento blok bude obsahovať všetky vygenerované reťaze, až na duplikáty, ktoré zahodíme. Chýbajúce reťaze sa pokúšame nahradiť a nakoniec zmažeme všetky nepotrebné pomocné súbory. Kľúčové časti postupu si teraz popíšeme podrobnejšie.

2.1.1 Vytváranie reťazí a používanie redukčnej funkcie

Jedným z dôležitých parametrov pri vytváraní reťazí je "fGenerateKey" - ukazovateľ na funkciu, ktorá na zadané miesto v pamäti zapíše nejaký kľúč. Tento parameter sa využíva hlavne ak sa chceme obmedziť na nejakú podmnožinu priestoru kľúčov. Napríklad ak chceme pomocou tabuľky "lámať" heslá, môžeme sa obmedziť na heslá alfanumerické. Samozrejme "tvaru" hesla treba prispôsobiť hlavne redukčnú funkciu, táto funkcia ovplyvňuje len nultý stĺpec tabuľky. V prípade, že užívateľ nemá záujem o využívanie tejto funkcie, je nutné zadať hodnotu tohto parametra rovnú "NULL". V takom prípade sa reťaze generujú s rovnomerným náhodným rozdelením na množine $\{0,1\}^k$, kde k je dĺžka kľúča. Nepoužívam však štandardnú funkciu jazyk C "rand()", ale vlastnú funkciu, ktorá je založená na iterovaní hashovacej funkcie SHA1. Autorom použitej implementácie tejto funkcie je Steve Reid[4]. Ešte by som poznamenal, že redukčná funkcia je postupne volaná s hodnotami 1, 2, ..., $l - 1$, kde l je dĺžka reťaze.

2.1.2 Spájanie dvoch blokov

Na vytváranie jedného súvislého bloku používame pomocnú funkcia, ktorá dokáže spojiť dva utriedené bloky reťazí do jedného utriedeného bloku. Jedná sa samozrejme o implementáciu klasického algoritmu "merge". Pretože reťaze tabuľky vytvárame s použitím kryptografických funkcií, a konce reťazí dostávame ako výstupy z týchto funkcií, je rozumné predpokladať, že tieto konce boli vytvorené s rovnakým náhodným rozdelením.[2] A pretože pri spájaní porovnávame práve konce reťazí, nemáme žiadne zvláštne predpoklady pri spájaní dvoch blokov. Respektíve pre dlhé bloky predpokladáme, že výsledný blok, vytvorený spojením

[2] Obvykle s rovnomerným rozdelením, za určitých okolností však aj s iným. Napríklad ak zvolíme funkciu pre vytváranie kľúčov, ktorá nemá rovnomerné rozdelenie.

týchto dvoch blokov, neobsahuje dlhé súvislé podpostupnosti skladajúce sa z prvkov len jedného z pôvodných blokov. Pre prípad, že veľkosť oboch blokov na vstupe je rovnaká, označme ju b, možno očakávať, že inštancia problému spájania, ktorú máme riešiť, sa podobá najhoršiemu možnému prípadu. To je taký, pre ktorý je výsledná postupnosť tvorená striedavo prvkami z prvého a druhého bloku. A keďže pre najhorší prípad je dokázané, že sa nedá zvládnuť pomocou menej ako $2b - 1$ porovnaní (nezávisle dokázali R. L. Graham a R. M. Karp), a my budeme väčšinou pracovať s blokmi približne rovnakej veľkosti, nemá zmysel pokúšať sa aplikovať nejakú heuristiku, ale uspokojíme sa so všeobecným algoritmom. Môžeme preto predpokladať, že počet porovnaní pri spájaní dvoch blokov je súčet dĺžok oboch blokov mínus jedna.

2.1.3 Vytváranie jedného utriedeného bloku

Na vytváranie utriedeného bloku teda používame funkciu, ktorá dokáže spojiť dva bloky. Väčší počet blokov potom spájame tak, že reťaze "presýpame" z jedného súboru do druhého, pričom vždy spájame dve susedné reťaze. Formálnejšie, predpokladajme, že máme vytvorených B blokov reťazí, označme ich S_1 až S_B. Poznamenajme, že o bloku S_i hovoríme, že sa nachádza na i-tom mieste. Ak B je párne číslo, pomocou spomenutej funkcie vytvoríme nový súbor, ktorý bude obsahovať $\frac{B}{2}$ reťazí, označme ich T_1 až $T_{\frac{B}{2}}$, pričom reťaz T_i vznikne spojením reťazí S_{2i-1} a S_{2i}. V prípade, že B nie je párne číslo, jeden z blokov len skopírujeme (teda vzťah pre postupnosti S_i a T_i bude iný). Tento krok, nazvime ho presypanie, opakujeme, až kým nedostaneme jeden utriedený blok.

Pozrime sa teraz na časovú zložitosť jedného presypania. Pre zjednodušenie uvažujme prípad, že počas presypania nezahadzujeme duplikáty. Nech počet všetkých reťazí je n. V každom prípade treba všetkých n reťazí prečítať z disku a potom ich aj na disk zapísať. Okrem toho robíme pri spájaní dvoch blokov porovnania, podľa ktorých určujeme poradie v akom budeme reťaze zapisovať. Ako sme uviedli v sekcii venovanej spájaniu dvoch blokov, pre dva bloky veľkosti x a y je počet porovnaní veľmi blízky hodnote $x + y - 1$, preto pre zjednodušenie našich úvah budeme túto hodnotu považovať za presný počet porovnaní. Ak máme párny počet blokov, spravíme $n - \frac{B}{2}$ porovnaní. Ak je ale počet blokov nepárny, je počet porovnaní menší. Ako teda vyberať bloky, ktoré nebudeme spájať, aby bol počet porovnaní počas celého vytvárania jedného bloku čo najmenší?

Uvažujme všeobecný prípad. Máme bloky S_1 až S_B, pričom pre ich veľkosti nemáme žiadne obmedzenia, snáď len, že veľkosť každého z nich je väčšia ako nula. Predpokladajme, že tieto bloky chceme spojiť do jedného podobne ako v našom prípade, pričom cena spojenia bloku veľkosti x s blokom veľkosti y je presne $x + y - 1$. Nazvime takúto cenu štandardnou. Dá sa pomerne jednoducho

dokázať, že algoritmus, ktorý spája vždy dva aktuálne najmenšie bloky, je z hľadiska štandardnej ceny optimálny, t.j. žiaden iný algoritmus to nespraví lacnejšie. My ale máme obmedzenia na veľkosti blokov a síce, že nanajvýš jeden z blokov na vstupe má menšiu veľkosť ako niektorý z ostatných. Postupujme teda takto. Ak máme na začiatku nejaký kratší blok, dajme ho na koniec. Robme presypania tak, ako to bolo popísané hore, s tým, že v prípade nepárneho počtu blokov skopírujeme tretí blok od konca. Zaujímavé je, že takto dostaneme optimálny algoritmus pre štandardnú cenu spájania. Ukážme, že to tak naozaj je. Zaveďme najprv niektoré dôležité pojmy. Pretože na začiatku spájania máme väčšinu blokov rovnakej veľkosti, označme ju b, budú postupným presýpaním vznikať bloky s dĺžkami $2^i b$ a bloky s menšími dĺžkami.

Definícia Nech S je blok, ktorý sme dostali po i presypaniach. Blok S sa nazýva normálny, ak je jeho veľkosť $2^i b$. Blok, ktorý nie je normálny sa nazýva kratší.

Pre presýpanie potom platia nasledujúce dve tvrdenia.

Tvrdenie 2.1.1 *Po ľubovoľnom počte presypaní sa kratší blok môže nachádzať len na poslednom alebo predposlednom mieste.*

Dôkaz Tvrdenie dokážeme matematickou indukciou vzhľadom na počet presypaní. Pre situáciu na začiatku toto tvrdenie platí, lebo menší blok sme dali na koniec. Ďalej z toho, že kopírujeme tretí blok od konca vyplýva, že vždy spájame posledné dva bloky. Teda ak je jeden z nich kratší, bude aj posledný blok v novej postupnosti kratší. Predposledný blok novej postupnosti môže byť kratší, len ak je to práve tretí blok od konca pôvodnej postupnosti. Všetky ostatné bloky vznikli spojením dvoch normálnych blokov, teda sú tiež normálne.

Tvrdenie 2.1.2 *Po ľubovoľnom nenulovom počte presypaní má predposledný blok veľkosť aspoň polovice normálneho bloku a posledný blok má viac ako polovicu veľkosti normálneho bloku.*

Dôkaz Využime na dôkaz Tvrdenie 2.1.1. Označme bloky pred presypaním S_1 až S_B Nech normálne bloky majú pred presypaním veľkosť b, teda po presypaní to bude $2b$. Všimnime si najprv, ako mohol vzniknúť predposledný blok. Buď spojením dvoch normálnych blokov S_{B-3} a S_{B-2} a teda je tiež normálny, alebo skopírovaním bloku S_{B-2}, ktorý bol ale normálny, teda jeho veľkosť je presne b. Teda preň tvrdenie platí. Posledný blok musel vzniknúť spojením S_{B-1} a S_B. V prípade, že išlo o prvé presypanie, mal blok S_{B-1} veľkosť b a teda ich spojením vznikol blok, ktorého veľkosť je viac ako b. Ak nešlo o prvé presypanie, platí indukčný predpoklad, teda $S_{B-1} \geq \frac{b}{2}$ a $S_B > \frac{b}{2}$. Teda pre blok, ktorý vznikne ich spojením platí, že jeho veľkosť je väčšia ako b.

Povedzme si teraz v akom poradí by sme spájali bloky, ak by sme sa riadili heuristikou dvoch najmenších blokov, a ukážme, že presýpanie spája tie isté bloky. Na začiatku máme nejaké bloky. V prípade, že je jeden z nich kratší, musíme ho spojiť s niektorým z normálnych blokov. Tým ale dostaneme aktuálne najväčší blok, teda pokračujeme spájaním normálnych blokov, kým je to možné. Toto isté ale dosiahneme prvým presypaním. V prípade, že sme v situácii, keď je predposledný blok jediným kratším, postupujeme analogicky. Ak sme po nejakom počte presypaní v situácii, keď máme dva kratšie bloky, budeme ich samozrejme spájať spolu. Tým podľa druhého tvrdenia dostávame opäť aktuálne najväčší blok a teda podobne ako v prvej časti dôkazu postupujeme ďalej spájaním normálnych blokov, teda rovnako ako pri presýpaní.

Toto bola analýza prípadu, kedy počas spájania blokov nevyhadzujeme duplikáty. Tento postup by mal rovnako dobre fungovať aj v prípade, že duplikáty priebežne zahadzujeme. Je asi rozumné predpokladať, že ak je blok S_1 väčší ako S_2, bude rovnaký vzťah platiť aj pre bloky S_1' a S_2', ktoré vzniknú z S_1 a S_2 zahodením duplikátov. Preto môžeme predpokladať, že dva najkratšie bloku budú vždy na posledných dvoch miestach[3]. Samozrejme v tomto prípade sa už nedá hovoriť o normálnych blokoch, pretože vyhadzovaním duplikátov budú vznikať bloky rôznych veľkostí, ale ich veľkosti sa veľmi líšiť nebudú. Navyše jediný rozdiel v cene by bol v tom, že nekopírujeme tretí blok od konca, ale najdlhší blok. Teda počet ušetrených porovnaní by bol maximálne rozdiel medzi najdlhším a najkratším z prvých $B - 2$ blokov, aj to iba v prípade, že B je nepárne číslo. Dodatočná réžia by ale niečo stála a prípadný ušetrený čas by bol zanedbateľný v porovnaní s minimálnym potrebným časom porovnávania a časom diskových operácií pri jednom presypaní.

2.1.4 Nahrádzanie zahodených duplikátov

Teraz si opíšme postup, ako funkcia nahrádza zahodené duplikáty. Pre túto časť funkcie sú dôležité parametre ”limit1” a ”limit2”. Tieto parametre nepriamo súvisia s parametrom veľkosti tabuľky. Ak totiž chceme, aby sa v tabuľke nachádzal istý počet reťazí, tak samozrejme nechceme, aby to bola stále tá istá reťaz, alebo tam bol polovičný počet rôznych reťazí, z toho každá dva krát. Treba sa vysporiadať nie len s kolíziami vzniknutými pri aplikácii redukčnej funkcie, ale aj s možnosťou, že náhodou vygenerujeme viacero reťazí s rovnakým začiatkom. Vzhľadom na pravdepodobnostnú povahu celého problému asi nie je vhodné tvrdohlavo trvať na počte reťazí vo vzniknutej tabuľke. (V prípade nejakej chyby v šifrovacej či redukčnej

[3] Pre tieto bloky by náš predpoklad mal naozaj platiť, pretože rozdiel veľkostí týchto dvoch blokov od ostatných môže byť veľmi veľký.

funkcii, alebo pri zle zvolených parametroch, by to mohlo viesť k zacykleniu.) Neformálne by sa dalo povedať, že tieto parametre predstavujú trpezlivosť algoritmu, teda špecifikujú, ako dlho sa budeme pokúšať nájsť nové reťaze do tabuľky. Čo presne znamenajú ich hodnoty ukazuje nasledujúca schéma algoritmu nahrádzania duplikátov.

Algoritmus 1 Nahrádzanie duplikátov

```
1: // t_0 je požadovaná veľkosť tabuľky
2: // t je veľkosť vytvorenej tabuľky
3: // c je počet reťazí, ktoré sa podarilo funkcii UPDATE nahradiť
4: while ((limit1 > 0) and (t < t_0)) do
5:     limit1 ← limit1 − 1
6:     Update(t_0 − t, c)
7:     t ← t + c
8: end while
```

Pričom procedúra ”Update” vyzerá nasledovne:

Algoritmus 2 Update

```
1: // c_0 počet reťazí, ktoré chcem pridať
2: // c počet reťazí, ktoré sa podarilo pridať
3: // k počet reťazí, ktoré sa podarilo pridať
4: procedure UPDATE(c_0, var c)
5:     c ← 0
6:     while ( (limit2 > 0) and (c_0 > 0) ) do
7:         limit2 ← limit2 − 1
8:         vytvor c_0 reťazí
9:         utrieď tieto reťaze
10:        zahoď d duplikátov
11:        c_0 ← d                          ▷ d reťazí treba ešte pridať
12:        if ( (d = 0) or (limit2 = 0) ) then
13:            spoj pôvodnú tabuľku s novými reťazami
14:            zahoď d duplikátov
15:            c ← c + c_0 − d
16:            c_0 ← d
17:        end if
18:    end while
19: end procedure
```

Ako je vidieť, parameter ”limit1” slúži najmä ako prepínač, ktorý rozhoduje

či sa budeme snažiť zahodené duplikáty nahrádzať, alebo nie. Pomocou parametra ”limit2” potom bližšie špecifikujeme, ako veľmi sa budeme snažiť.

2.2 Zväčšovanie tabuliek

Zväčšovanie tabuliek vyzerá veľmi podobne ako nahrádzanie zahodených duplikátov pri vytváraní tabuliek. V prípade, že zadáme neplatnú hodnotu pre premennú ”limit1” (t.j. menšiu alebo rovnú ako nula), funkcia prebehne akoby sme jej hodnotu nastavili na 1. Okrem toho treba zdôrazniť, že funkcia spájania pôvodnej tabuľky s novými reťazami garantuje zachovanie všetkých reťazí z pôvodnej tabuľky. Teda sa netreba báť, že pri zvolení ”strict_dupl_detection” na nenulovú hodnotu sa vyhádžu nejaké reťaze z pôvodnej tabuľky.

Pre zväčšovanie tabuliek je však k dispozícii možnosť, ktorá pri vytváraní nie je. Ak si všimneme schému procedúry ”update” z predošlej sekcie, najprv vytvoríme nejaké reťaze, tie utriedime a nakoniec sa ich pokúsime pridať k existujúcej tabuľke. V prípade, že máme veľkú tabuľku sa môže stať, že pri spájaní s existujúcou tabuľkou veľké množstvo z týchto reťazí zahodíme. Čas, ktorý stratíme zbytočným triedením, nás moc trápiť nemusí. Triedenie je veľmi efektívne a jeho čas je v podstate zanedbateľný v porovnaní s vytváraním reťazí, ktoré pozostáva z opakovaného volania zložitých a časovo náročných funkcií, a spájaním blokov reťazí, ktoré vyžaduje množstvo diskových operácii. Sú to práve tie diskové operácie, pri ktorých by sme mohli nejaký čas ušetriť. Samozrejme počas generovania reťaze nemôžme vedieť, že reťaz sa už v tabuľke nachádza, pretože tabuľky sú zoradené podľa koncov reťazí. Funkcia preto ponúka možnosť vyhľadávania reťazí v tabuľke, ku ktorému dochádza bezprostredne po ich vygenerovaní. V prípad, že sa reťaz už v tabuľke nachádza, ihneď ju zahodíme. Na účely porovnávania reťazí používame funkciu *compare*, ktorú vyberáme rovnako ako v prípade vytvárania tabuliek. Ak si chceme zvoliť túto možnosť, je treba zadať nenulovú hodnotu parametra ”lookups_on”. Hodnota premennej ”lookups” predstavuje maximálny počet reťazí, ktoré pri takomto postupe zahodíme. Tým zabránime prípadnému zacykleniu. Podrobnejšia analýza problému, kedy je vhodné túto možnosť zvoliť, sa nachádza v samostatnej sekcii na konci tejto kapitoly.

2.3 Prehľadávanie tabuliek

Prehľadávanie je samozrejme realizované pomocou binárneho vyhľadávania. Aj keď algoritmus je trochu modifikovaný, pretože v tabuľke sa môže nachádzať viacero reťazí s rovnakým koncom. Okrem samotného prehľadávania tabuľky, sa počas behu tejto funkcie sleduje počet jednotlivých volaní funkcí šifrovania a redukcie

zvlášť počas falošných poplachov a zvlášť počas "normálneho" prehľadávania. Rozdiel medzi týmto dvomi prípadmi je nasledovný. Nech dĺžka reťazí v tabuľke je $l+1$. Ako som napísal v úvode tejto práce, na prehľadávanie i-teho stĺpca tabuľky[4] je potrebných $l-i$ volaní šifrovacej funkcie a rovnako veľa volaní redukčnej funkcie. Teda v prípade, že reťaz nájdeme v i-tom stĺpci bude počet volaní týchto funkcii $1+2+...+(l-i)$. O týchto volaniach hovoríme, že nastali počas normálneho prehľadávania a k prípadným falošným poplachom ich pripočítavať nebudeme. V prípade, že v i-tom stĺpci nájdeme hľadaný koniec, je treba zrekonštruovať prvých $i+1$ prvkov reťaze[5], na čo potrebujeme $i+1$ volaní funkcie šifrovania a i volaní redukčnej funkcie. Nakoniec robíme porovnanie, či sa jedná o falošný poplach. Ak nie, pripočítame tieto hodnoty k normálnemu hľadaniu a funkcia skončí. Ak áno, pripočítame ich k volaniam počas falošných poplachov.

Ďalšie atribúty, ktoré sa počas prehľadávania sledujú, sú počet porovnaní počas binárnych prehľadávaní a čas, ktorý prehľadávanie trvalo.

2.4 Bližší pohľad na parameter "lookups_on"

Pozrime sa teraz detailnejšie na dva možné spôsoby rozširovania tabuliek. Daná je teda tabuľka o veľkosti n. My sa ju pokúsime rozšíriť o ďalších k reťazí, teda aby veľkosť výslednej tabuľky bola $n+k$. O reťaziach, ktoré sa ešte v tabuľke nenachádzajú, budeme hovoriť ako o nových reťaziach. Uvažujme prípad, že algoritmus neskončí, kým sa mu nepodarí pridať všetkých k reťazí. Ako som už uviedol, naša knižnica ponúka dve možnosti ako, takéto rozširovanie robiť. *Algoritmus 1* rieši tento problém tak, že vygeneruje k reťazí, utriedi ich a následne spája takto získaný blok reťazí s pôvodnou tabuľkou. Tieto kroky bude opakovať, až kým sa pri spájaní nezahodí žiaden duplikát. *Algoritmus 2* spraví pre každú vygenerovanú reťaz kontrolu, či sa už v tabuľke nenachádza. Ak zistí, že áno, tak ju zahodí. Spájanie sa teda robí len raz, vo chvíli, keď máme k nových reťazí.

V prípade, že naša tabuľka je veľmi veľká z hľadiska priestoru všetkých možných kľúčov, mohli by sme medzi vygenerovanými reťazami mať viaceré, ktoré sa už v tabuľke nachádzajú. A pretože by takáto tabuľka bola pravdepodobne veľká aj z hľadiska diskového priestoru, samotné spájanie by mohlo trvať veľmi dlho. Ak by sme ho mali opakovať viac krát, aby sme nahradili zahodené duplikáty, stratili by sme veľa času zbytočne. Podobne by sme mohli uvažovať, keby bol počet pridávaných reťazí relatívne malý. Tu by sme zasa za určitých okolností vyhľadávaním v tabuľke stratili oveľa menej času, ako prípadným robením nejakého spájania navyše. Samozrejme, základným riešením je *algoritmus 1*. V nasledujúcom

[4] Pripomeňme si, že stĺpce sú číslované od nuly.

[5] Pretože číslujeme od nuly, tak nultý až i-ty, teda $i+1$.

texte sa budem snažiť nájsť vzťah, ktorý nám povie, kedy je výhodnejšie použiť druhú možnosť. Pre možné rôzne hardvérové parametre však nie je možné tento vzťah implementovať priamo v programe, a v konečnom dôsledku je na užívateľovi, ktorú možnosť zvolí. Uvažujme prípad, kedy je existujúca tabuľka veľká, a počet pridávaných reťazí relatívne malý. V takomto prípade môžeme prípadnú existenciu duplikátov medzi novovygenerovanými reťazami zanedbať.

Označme písmenom d veľkosť kľúča v bitoch. Počet všetkých možných kľúčov je teda 2^d. Nech p je pravdepodobnosť, že náhodne vygenerovaná reťaz sa už v našej tabuľke nachádza. Za predpokladu, že reťaze generujeme rovnomerne náhodne na množine $\{0,1\}^d$, p jednoducho vypočítame zo vzťahu:

$$p = \frac{n}{2^d}$$

Ďalej označme $q = 1 - p$. Potom pravdepodobnosť, že sa nám práve na m pokusov (kde $m \geq 1$) podarí vygenerovať novú reťaz je:

$$qp^{m-1}$$

Teda strednú hodnota počtu pokusov potrebných na vygenerovanie novej reťaze dostanem z výrazu:

$$\sum_{i=1}^{\infty} iqp^{i-1} = q\sum_{i=1}^{\infty} ip^{i-1}$$

Na výpočet sumy na pravej strane využijeme vzťah, ktorý platí pre $|x| < 1$:

$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$$

Po zderivovaní oboch strán rovnice dostávame vzťah:

$$\frac{1}{(1-x)^2} = \sum_{i=0}^{\infty} ix^{i-1} = \sum_{i=1}^{\infty} ix^{i-1}$$

Z toho teda dostávame strednú hodnotu rovnú:

$$\frac{q}{(1-p)^2} = \frac{1}{1-p}$$

Z linearity strednej hodnoty priamo vyplýva, že očakávaný počet reťazí, ktoré treba vygenerovať, aby sme dostali k nových reťazí, je

$$\frac{k}{1-p}$$

V ďalšom budem používať už popísaný algoritmus spájania dvoch blokov. Tento algoritmus má samozrejme lineárnu časovú zložitosť, v praxi je však závislý od viacerých hardvérových parametrov ako napríklad rýchlosť diskov, alebo veľkosť cache pamäte procesora. Čas, ktorý nás stojí spájanie dvoch blokov o veľkostiach x a y budeme teda vyjadrovať parametricky ako $C_1(x+y)$, kde C_1 je nejaké reálne číslo.

Skúsme ďalej vyjadriť, ako dlho trvá utriedenie k reťazí, za predpokladu, že sme v operačnej pamäti vyhradili miesto pre a reťazí. Máme teda $\lceil \frac{k}{a} \rceil$ utriedených blokov, z ktorých každý, s možnou výnimkou posledného, obsahuje a reťazí. Následne robíme presypania, až kým nedostaneme len jeden utriedený blok. Pretože pri každom presypaní sa počet blokov zmenší na polovicu ($\lceil \frac{\#blokov}{2} \rceil$), potrebný počet prechodov poľom je možné aproximovať hodnotou $\lg \frac{k}{a}$. Teda odhadovaný čas potrebný na spojenie všetkých blokov do jedného utriedeného poľa je $C_1 k \lg \frac{k}{a}$. Prípadný čas ušetrený kopírovaním jedného z blokov počas presypania môžeme zanedbať, veď vzhľadom na čas diskových operácii je aj tak bezvýznamný. Prípadne sa tento problém dá ošetriť prispôsobením hodnoty konštanty C_1.

Teraz môžeme vyjadriť odhadovaný čas pre pridávanie reťazí jedným alebo druhým algoritmom. Pretože predpokladáme, že v oboch prípadoch treba vygenerovať rovnako veľa reťazí, nebudeme čas potrebný na ich generovanie rátať. Budeme uvažovať iba tie časti algoritmov, ktoré oba postupy odlišujú ich časová náročnosť je relatívne vysoká[6].

Pre *algoritmus 1*, keď nerobíme kontrolu, či sa nami vygenerovaná reťaz už v tabuľke nachádza, je odhadovaný čas strávený spájaním:

$$C_1[k \lg \frac{k}{a} + k + n]\frac{1}{1-p}$$

Túto hodnotu sme dostali ako súčet času potrebného na spojenie všetkých blokov a času potrebného na spojenie tabuľky a samotného poľa.[7] Pretože pri jednom takomto kroku vygenerujeme k reťazí, prenásobíme tento výraz hodnotou $\frac{1}{1-p}$, aby sme dostali očakávaný počet reťazí. Ako je to s presnosťou takéhoto odhadu? Na jednej strane sme nebrali do úvahy fakt, že pri každej iterácii *algoritmu 1* sa počet reťazí, ktoré ešte treba do tabuľky pridať, zmenšuje, a teda aj čas presýpaní sa zmenšuje. Rozdiel je však pre naše predpoklady zanedbateľný. Ak generujeme k reťazí, a p je pravdepodobnosť, že vyrobíme duplikát, je očakávaný počet duplikátov spomedzi všetkých k reťazí kp. Teda na začiatku i-teho spájania pôvodnej tabuľky a vygenerovaných reťazí očakávame počet týchto reťazí kp^{i-1}. Teda čas, ktorý

[6] V zmysle, že má významný vplyv na výsledný čas.

[7] Súčet veľkostí tabuľky a poľa je vždy $n+k$, pretože chceme výslednú tabuľku s touto veľkosťou.

trávime presýpaním je daný vzťahom:

$$\sum_{i=0}^{\infty} kp^i (\text{čas prsypania } kp^i \text{ reťazí})$$

Pritom čas presypania reťazí je $\lg \frac{kp^i}{a}$ ak $kp^i > a$ a nula v opačnom prípade. Pretože my uvažujeme prípad pre malé k, možno čas presypania nahradiť hodnotou $\lg \frac{k}{a}$. Nie len, že rozdiel týchto dvoch hodnôt je malý, ale ostáva malým aj po prenásobení hodnotou k[8]. Dostávame:

$$\sum_{i=0}^{\infty} kp^i \lg \frac{k}{a} = \frac{k}{1-p} \lg \frac{k}{a}$$

Okrem tohto sme zanedbali čas triedenia blokov, ktorí by túto hodnotu zasa zväčšil. To nám ale nevadí, pretože hľadáme hranicu od ktorej je *algoritmus 2* efektívnejší, teda nám pre prvý algoritmus stačí dolný odhad.

V prípade *algoritmu 2* budeme robiť kontrolu, či sa daná reťaz už v tabuľke nachádza. Samotné prehľadávanie tabuliek robíme algoritmom binárneho vyhľadávania, ktorý má logaritmickú časovú zložitosť. Ale podobne ako v prípade spájania tabuliek je presný čas potrebný na jeho vykonanie hardvérovo závislý. Nech teda hľadanie v tabuľke o veľkosti n trvá $C_2 \lg n$, kde C_2 je nejaké reálne číslo. Potom čas, ktorý trvá pridanie k nových reťazí do tabuľky je daný vzorcom:

$$\frac{k}{1-p} C_2 \lg n + C_1[k \lg \frac{k}{a} + k + n]$$

Teda o každej z vygenerovaných reťazí sa presvedčíme, či sa ešte v tabuľke nenachádza a na záver urobíme spájanie blokov ako v prvom prípade.

Vyjadrime teraz, kedy je výhodnejšie používať *algoritmus 2*:

$$\begin{array}{rcll} \text{Čas s kontrolou tabuľky} & \leq & \text{Čas bez kontroly} & \\ \frac{k}{1-p} C_2 \lg n + C_1[k \lg \frac{k}{a} + k + n] & \leq & C_1[k \lg \frac{k}{a} + k + n] \frac{1}{1-p} & \\ \frac{k}{1-p} C_2 \lg n & \leq & C_1[k \lg \frac{k}{a} + k + n] \frac{p}{1-p} & /.(1-p) \\ k\, C_2 \lg n & \leq & C_1 p[k \lg \frac{k}{a} + k + n] & \\ C_2 \lg n & \leq & C_1 p[\lg \frac{k}{a} + 1 + \frac{n}{k}] & \end{array}$$

Z poslednej nerovnosti je vidieť náš predpoklad, že pre veľké n respektíve malé hodnoty k je výhodnejšie robiť *algoritmus 2*. Pre špeciálny prípad, kedy máme k

[8] Teda malý v porovnaní s veľkým n

dispozícii dosť operačnej pamäte pre všetkých k reťazí, a teda žiadne presýpanie robiť nemusíme, je vzťah ešte jednoduchší:

$$C_2 \lg n \quad \leq \quad C_1 p[1 + \tfrac{n}{k}]$$

Toto bolo odvodenie pre prípad, kedy je tabuľka veľká a počet pridávaných reťazí relatívne malý. V prípade, že tabuľka nie je veľmi veľká, nemá zmysel robiť v nej kontrolu, pretože je malá pravdepodobnosť, že hľadanú reťaz nájdeme. V prípade, že chceme pridávať veľké množstvo reťazí do malej tabuľky je asi najvhodnejšie túto úlohu rozdeliť na dve. Najprv pridávať bez kontroly tabuľky, a keď už tabuľka bude veľká, začať ju priebežne prehľadávať.

Pre posledný prípad, kedy n aj k sú veľké, nemusí mať naša úloha riešenie, pretože vyžaduje vytvorenie tabuľky o veľkosti $n + k$. V podstate ani nie je možné určiť[9], či pri zvolených parametroch existuje nejaká reťaz, ktorá sa v tabuľke ešte nenachádza. Navyše implementácia obsahuje obmedzenie na počet zahodených duplikátov, takže predošlé úvahy pre tento prípad nie sú vhodné.

[9] Inak ako úplným prehľadávaním.

Kapitola 3

Popis rozhrania

3.1 Štruktúra na predávanie parametrov

Na predávanie vstupných a výstupných hodnôt medzi užívateľom a knižnicou slúži štruktúra menom ”TableInfo”. Táto kapitola je venovaná jej jednotlivým premenným a ich význame pre jednotlivé funkcie knižnice. Celkový prehľad ponúka nasledujúca tabuľka, bližšie sa na tieto premenné pozrieme pri popise samotných funkcií v druhej časti tejto kapitoly.

Názov premennej	**Typ**	**Popis**
key_size	int	dĺžka kľúča v bitoch
hash_size	int	dĺžka šifrového textu v bitoch
table_size	int	veľkosť tabuľky (pozri funkcie ”createTalbe” a ”upgradeTable”)
chain_length	int	dĺžka reťaze
array_size	int	veľkosť poľa alokovaného v pamäti (počet dvojíc kľúč - šifrový text)
remove_duplicates	int	špecifikuje, kedy počas spájania blokov zahadzujeme reťaze
strict_dupl_detection	int	špecifikuje, ktoré reťaze pri vytváraní zahodíme
lookups_on	int	špecifikuje, či pri rozširovaní tabuliek kontrolujeme kolízie s existujúcou tabuľkou
lookups	int	maximálny počet zahodených reťazí pri rozširovaní
limit1	int	
limit2	int	
table_name	char*	unikátny identifikátor tabuľky

Názov premennej	Typ	Popis
fEncode	pointer	void (*) (void* key, void* hash)
fReduce	pointer	void (*) (void* hash, int i, void* key)
fCompare	pointer	int (*) (const void* item1, const void* item2)
fGenerateKey	pointer	void (*) (void* key)
collisions	int	počet zahodených reťazí počas vytvárania tabuľky
chains	int	počet vytvorených\pridaných reťazí
row	int	riadok v ktorom bol nájdený kľúč v tabuľke; ak nebol nájdený, hodnota je rovná mínus jednej
column	int	podobne ako ”row”, len ide o stĺpec tabuľky
pKey	pointer	ukazuje na kľúč, ktorý bol v tabuľke nájdený
false_alarms	int	počet falošných poplachov počas prehľadávania
nencode	int	počet volaní ”fEncode” počas prehľadávania
falase_nencode	int	počet volaní ”fEncode” počas falošných poplachov
nreduce	int	počet volaní ”fReduce” počas prehľadávania
false_nreduce	int	počet volaní ”fReduce” počas falošných poplachov
ncompare	int	počet volaní ”fCompare” počas prehľadávania
create_time	my_time	celkový čas vytvárania tabuľky
create_chains_time	my_time	čas vytvárania reťazí počas vytvárania tabuľky
upgrade_time	my_time	celkový čas zväčšovania tabuľky
upgrade_chains_time	my_time	čas vytvárania reťazí počas zväčšovania tabuľky
search_time	my_time	čas prehľadávania tabuľky

Poznámka: Typ ”my_time” je štruktúra, pomocou ktorej sme schopní merať čas trvania jednotlivých častí programu s presnosťou na tisíciny sekundy (viď prílohu).

3.2 Funkcie a procedúry

3.2.1 createTable

Základnou funkciou knižnice je možnosť vytvárania dúhových tabuliek pomocou funkcie ”createTable”. Táto funkcia má len jeden formálny parameter, a to ukazovateľ na štruktúru ”TableInfo”. Základné argumenty sú pointer na šifrovaciu funkciu, redukčnú funkciu a na porovnávaciu funkciu. Ak pre užívateľa nie je dôležité v akom poradí sú reťaze v tabuľke uložené, môže zadať hodnotu ukazovateľa na porovnávaciu funkciu rovnú ”NULL”. V takom prípade budú časti reťazí predstavujúce šifrový text chápané ako čísla v dvojkovej sústave a budú zoradené od najmenšieho po najväčšie. Ďalšie dôležité parametre sú veľkosť kľúča, veľkosť šifrového textu, dĺžka reťaze a veľkosť samotnej tabuľky. Na identifikáciu tabuliek

slúži parameter meno tabuľky.

Doteraz spomenuté parametre bližšie špecifikujú samotné tabuľky. Pre vytváranie tabuliek sú však k dispozícii aj parametre iného druhu, ktoré majú vplyv na použitý algoritmus. Najdôležitejší parameter z tejto kategórie je veľkosť operačnej pamäte. Nejde však o to, aká je celková veľkosť vašej pamäte, ale aký veľký súvislý úsek pamäte dáte tejto funkcii k dispozícii. Ako sme ukázali v kapitole venovanej implementácii, táto funkcia potrebuje jeden dlhý úsek pamäte, ktorého veľkosť v podstate určuje rýchlosť vytvárania tabuliek. Okrem tohto úseku navyše funkcia využíva len zanedbateľné miesto operačnej pamäte, preto je z hľadiska rýchlosti programu dobré voliť hodnotu tohto parametra na hranici dostupnej pamäte.

Výstup tejto funkcie tvoria dva súbory. Jeden, ktorý predstavuje tabuľku, je tvorený utriedenou postupnosťou reťazí. Druhý, ktorý budem nazývať popisným súborom pre danú tabuľku, obsahuje informácie o dĺžke kľúča, dĺžke šifrového textu a dĺžke reťazí uložených v tabuľke. Popisný súbor tvorí len akýsi doplnok k tabuľke a pre budúce pracovanie s tabuľkou, a správnu funkcionalitu ostatných funkcií z knižnice, nie je potrebný. Používanie tohto súboru sa však odporúča, pretože v ňom obsiahnuté informácie sa z tabuľky dajú zistiť len veľmi ťažko. Takisto si treba dať pozor na výstup z ostatných funkcií ktoré pracujú s už existujúcimi tabuľkami. Tie v prípade, že informácie o tabuľke neboli prečítané z popisného súboru, vrátia iné hodnoty.

Funkcia počas svojho behu sleduje počet zahodených duplikátov a čas, ktorý zaberajú jednotlivé podúlohy.

3.2.2 upgradeTable

Podobná funkcia ako vytváranie tabuliek je možnosť rozširovať už existujúce tabuľky, a to prostredníctvom funkcie ”upgradeTable”. Pre túto funkciu je význam vstupných parametrov rovnaký ako pri predošlej funkcii. Jediný rozdiel je v interpretácii parametra veľkosť tabuľky. Ten v tomto prípade predstavuje veľkosť o ktorú sa má tabuľka rozšíriť. Veľkosť výslednej tabuľky je teda väčšia. Názov parametra síce trochu mýli, dôvod pre takúto interpretáciu je ale taký, že ak chceme zväčšovať nejakú tabuľku, nepotrebujeme si zisťovať jej veľkosť.

Ďalšia vlastnosť, ktorú majú obe funkcie spoločnú je význam návratovej hodnoty. Ak je niektorá z funkcii vykonaná bez problémov, vráti hodnotu 0. Ak však počas behu funkcie nastane nejaká chyba, funkcia vráti kód danej chyby. Tabuľka chybových kódov je uvedená v prílohe.

V prípade, že sa nepodarí nájsť popisný súbor pre zvolenú tabuľku, a v priebehu vykonávania funkcie nedôjde k nejakej chybe, bude tento súbor automaticky vytvorený.

Na záver by som zdôraznil, že pre správne fungovanie funkcií ”createTable” a

”upgradeTable” je nutné mať na disku minimálne toľko voľného miesta, koľko by zaberal dvojnásobný počet pridávaných reťazí. Teda v prípade vytvárania tabuliek dvojnásobok veľkosti tabuľky. Táto skutočnosť vyplýva zo spôsobu akým spájame bloky reťazí. Výnimkou je prípad, kedy sa nám celá tabuľka zmestí naraz do operačnej pamäte a nechceme nahrádzať prípadné zahodené duplikáty. Je to ale jediný prípad, kedy vieme zaručiť, že sa nevyužíva funkcia na spájanie blokov.

3.2.3 searchTable

Na prehľadávanie tabuliek slúži funkcia ”searchTable”. Táto funkcia má dva formálne parametre, okrem štruktúry ”TableInfo” aj pointer na šifrový text, ku ktorému hľadáme kľúč. Podobne ako v predošlom prípade, väčšina vstupných argumentov má rovnaký význam. Parameter veľkosť tabuľky sa ignoruje, funkcia si ho dopočíta na základe veľkosti kľúča, šifrového textu a veľkosti súboru, ktorý obsahuje tabuľku. V prípade, že existuje popisný súbor pre danú tabuľku, hodnoty parametrov v ňom uložené sa uprednostnia pred tými, ktoré zvolil užívateľ.

V prípade, že sa hľadaný kľúč v tabuľke nepodarilo nájsť, funkcia vráti nenulovú hodnotu, presný význam tejto hodnoty opisuje tabuľka v prílohe. Ak sa však kľúč podarilo nájsť, a všetko prebehlo v poriadku, vráti hodnotu nula, a prostredníctvom našej štruktúry vráti riadok a stĺpec v ktorom sa nachádza. Spolu s tým aj pointer na hodnotu kľúča, ktorú zapíše do pamäte. Okrem toho sa počas prehľadávania sleduje počet jednotlivých volaní funkcí šifrovania a redukcie, a to zvlášť počas falošných poplachov, a zvlášť počas ”normálneho” prehľadávania. V premennej ”ncompare” sa počíta počet porovnaní, ktoré sa robia počas binárneho prehľadávania tabuľky. Celkový čas prehľadávania zistíme z premennej ”search_time”.

3.2.4 createDescriptionFile

Funkcia ”createDescriptionFile” vytvorí popisný súbor pre tabuľku s daným názvom. Táto funkcia nerobí žiadne kontroly čo sa týka korektnosti ukladaných parametrov ani existencie iného súboru s rovnakým názvom. Dokonca nemusí ani existovať tabuľka, ku ktorej popisný súbor vytvárame.

3.2.5 saveToTextfile, saveToTextfile2

Ďalej sú tu dve pomocné funkcie, slúžiace na zapisovanie parametrov. Funkcia ”saveToTextfile” dostane ako parametre ukazovateľ na štruktúru ”TableInfo” a názov textového súboru. Ona sama potom do zvoleného súboru zapíše všetky hodnoty zo štruktúry v prehľadnom formáte.

Druhá verzia tejto funkcie s názvom ”saveToTextfile2” je určená pre zápis informácii o prehľadávaní tabuľky. Oproti prvej funkcii má ďalšie dva parametre a to reťazec ”open_type” a ukazovateľ na šifrový text. Prvý parameter slúži na špecifikáciu, či zvolený súbor prepíšeme, alebo do neho budeme pripisovať, teda povolené hodnoty sú ”w” a ”a”. V prípade inej hodnoty, funkcia nevykoná žiadnu operáciu. Druhý parameter by mal ukazovať na šifrový text, ktorý sme sa práve pokúsili nájsť v tabuľke, pričom ak sme ho našli, príslušný kľúč bude prístupný pomocou premennej ”pKey” štruktúry ”TableInfo”. Funkcia potom do súboru zapíše všetky informácie súvisiace s prehľadávaním a v prípade, že je volaná s hodnotou ”w” pre parameter ”open_type”, pridá aj základné informácie o tabuľke. Funkcia je teda určená hlavne na zaznamenávanie priebehu viacnásobného prehľadávania tabuľky, kedy nie je vhodné robiť výpis na obrazovku.

3.2.6 loadMainParameters

Poslednou z pomocných funkcií je ”loadMainParameters”. Táto funkcia dostane takisto ukazovateľ na štruktúru ”TableInfo”, k tomu ešte jeden parameter typu int. Pokúsi sa zistiť, či existuje tabuľka so zvoleným názvom, a či existuje popisný súbor pre túto tabuľku. Môžu totiž nastať viaceré kombinácie, pokiaľ ide o existenciu jedného či druhého súboru. Ak však popisný súbor existuje, tak funkcia do našej štruktúry v každom prípade zapíše v súbore uložené informácie, teda veľkosť kľúča, šifrového textu a dĺžku reťazí v tabuľke. V prípade, že druhý parameter je nenulový, a tabuľka je k dispozícii, zapíše aj veľkosť tabuľky. (Tú vypočíta z veľkosti kľúča, šifrového textu a súboru obsahujúceho tabuľku.) Možnosť nenačítať veľkosť tabuľky je z dôvodu používania tejto funkcie aj v rámci samotnej knižnice. Ak všetko prebehne bez problémov, funkcia vráti hodnotu 0. Pre nenulové hodnoty je uvedená tabuľka chybových kódov v prílohe.

Kapitola 4

Príklad použitia vytvorenej knižnice

4.1 Zvolené parametre a funkcie

Ako som spomínal už v úvode tejto práce, priestor kľúčov šifrovacích funkcií býva veľmi veľký. Ale na to, aby sme ukázali, že knižnica naozaj funguje by bolo dobré, aby sme okrem vytvárania a rozširovania tabuliek boli schopní tieto tabuľky aj úspešne prehľadávať. Úspešne znamená, že k hľadanému hashu nájdeme príslušný vzor. Aby nám vytváranie tabuliek, ktoré nám zaručia dobrú pravdepodobnosť úspešného hľadania, netrvalo príliš dlho, obmedzíme veľkosť vzoru aj veľkosť hashu na 24 bitov.

Hashovacia funkcia vyzerá tak, že 24 bitová hodnota sa zahashuje pomocou funkcie MD5 a z výsledného 128 bitového hashu sa zoberie iba prvých 24 bitov zľava. Autorom implementácie funkcie MD5 je L. Peter Deutsch[3]. Redukčná funkcia vyzerá nasledovne:

Algoritmus 3 Redukčná funkcia

1: **function** REDUCE($i, hash$)
2: $key[0] \leftarrow (hash[0] + i) \bmod 256$
3: $i \leftarrow i/256$;
4: $key[1] \leftarrow (hash[1] + i) \bmod 256$
5: $i \leftarrow i/256$;
6: $key[2] \leftarrow (hash[2] + i) \bmod 256$
7: **return** key
8: **end function**

Táto funkcia je veľmi jednoduchá, ale je ľahko vidieť, že pre rovnaké hodnoty parametra *hash* a rôzne hodnoty parametra i je návratová hodnota rôzna (pre $0 \leq i < 2^{24}$).

Na vytváranie reťazí a porovnávanie pri triedení reťazí sa používajú funkcie implementované v knižnici, teda hodnota príslušných parametrov sa nastaví na ”NULL”.

4.2 Orientácia v menu

Naša aplikácia pracuje s tromi súbormi, jeden slúži na prácu so vstupnými hodnotami premennej typu ”TableInfo”, do druhého sa zapisujú všetky hodnoty tejto premennej (rozdiel je vysvetlený neskôr) a tretí slúži na zaznamenávanie priebehu prehľadávania, teda sa posiela ako parameter pre funkciu ”saveToTextfile2”. Samotný program pracuje s konštantami ”FILE_NAME[1, 2, 3]” definovanými pomocou direktívy ”#define”. Ja ich mám definované ako ”nastavenia.txt”, ”parametre.txt” a ”prehladavanie.txt”, a tieto názvy budem v ďalšom texte používať.

Rozhranie programu je robené tak, že si užívateľ vyberá jednotlivé položky z menu. Tie sú porade očíslované, obvykle od nuly ďalej. Na výber danej položky je treba zadať príslušné číslo a potvrdiť klávesou ”enter”. V prípade, že užívateľ zadá hodnotu, ktorá v ponuke nie je, nič sa nedeje. Program čaká, až kým nie je zadaná platná hodnota. Treba si však dať pozor, aby zadaná hodnota neobsahovala iné znaky ako čísla. V takom prípade to vedie k problémom.

Ďalej by som dodal, že v prípade, že je to aktuálne, je vždy nad menu, v ktorom sa práve nachádzame, zobrazený výpis dôležitých parametrov s aktuálne nastavenými hodnotami.

Na záver by som uviedol malú poznámku k názvosloviu. Na prácu s parametrami tabuľky sa používa premenná ”ti” typu ”TableInfo”. V ďalšom texte budem o premenných obsiahnutých v tejto štruktúre hovoriť ako o parametroch potrebných pre prácu s tabuľkami respektíve aktuálnych parametroch.

4.2.1 Hlavné menu

Hlavné menu aplikácie vyzerá takto:

0 - Ukončiť program
1 - Načítanie vstupných parametrov zo súboru ”nastavenia.txt”
2 - Zapísanie vstupných parametrov do súboru ”nastavenia.txt”
3 - Zadanie vstupných parametrov ručne
4 - Vytvoriť tabuľku
5 - Uložiť parametre do súboru ”parametre.txt”
6 - Zväčšiť existujúcu tabuľku
7 - Prehľadávať tabuľku
8 - Prečítaj parametre tabuľky

Pritom možnosti 5 - 7 budú k dispozícii len ak sa podarí nájsť tabuľku s aktuálne zvoleným názvom. Vysvetlime si teraz význam jednotlivých položiek.

Načítanie vstupných parametrov zo súboru ”nastavenia.txt” - parametre potrebné pre prácu s tabuľkami sa prečítajú z daného súboru. Formát tohto súboru však musí spĺňať dosť prísne kritéria. Hodnota každého parametra musí byť uvedená v samostatnom riadku. Každý riadok musí byť v tvare:

<názov premennej> = <hodnota>

Teda súbor nesmie obsahovať ani prázdny riadok, ani medzery pred názvom premennej. Na poradí parametrov nezáleží. Parametre, ktoré je možno takto zadať, tvoria riadky 3 až 12 v tabuľke v kapitole venovanej rozhraniu. Sú to teda parametre od veľkosti tabuľky po názov tabuľky.

Zapísanie vstupných parametrov do súboru ”nastavenia.txt” - vstupné premenné budú zapísané v správnom formáte do daného súboru.

Zadanie vstupných parametrov ručne - presunutie sa do *Menu pre zadávanie parametrov*, ktoré je popísané neskôr.

Vytvoriť tabuľku - najprv sa skontrolujú najdôležitejšie parametre. V prípade, že nie je detekovaný problém, zavolá sa funkcia ”createTable” s aktuálne zvolenými parametrami. Možné problémy s parametrami sú:

1 - Nie je zadaný názov tabuľky
2 - Veľkosť tabuľky nie je kladné číslo
3 - Dĺžka reťaze nie je kladné číslo
4 - Veľkosť poľa je menšia ako 10
5 - Tabuľka so zvoleným názvom už existuje

V prípade, že nastane niektorý z problémov 1-4, aplikácia upozorní na problém aj s informáciou, ktorý parameter je zle zvolený. V prípade 5 sa opýta, či

sa má táto tabuľka prepísať, alebo chceme zrušiť akciu. Pokiaľ ide o problém 4, tak minimálna hodnota dĺžky poľa, pre ktorú knižnica funguje správne, je dva. Ako som spomínal, funkcie ”createTable” a ”upgradeTable” majú vlastnú kontrolu veľkosti poľa a táto hodnota je nastavená na 10, čo sa ale dá na úrovni kódu zmeniť. Autor tohto textu však neverí, že v tomto konkrétnom prípade by bolo potrebné ísť pod hodnotu 10, pretože to predstavuje len 60 bajtov operačnej pamäte. A preto aj keď by bola hodnota tej hranice v knižnici zmenená, v tomto prípade je hodnotu pod 10 lepšie považovať za preklep.

Uložiť parametre do súboru ”parametre.txt” - volanie funkcie ”saveToTextfile” s aktuálnymi parametrami.

Zväčšiť existujúcu tabuľku - vykonanie kontroly parametrov podobne ako pri vytváraní tabuľky. Ak je všetko v poriadku, zavolá sa funkcia ”upgradeTable” s aktuálnymi parametrami. Ako už bolo spomenuté pri popise funkcií knižnice, v prípade že existuje popisný súbor tabuľky, nie je nutné mať parameter dĺžky reťaze zvolený správne.

Prehľadávať tabuľku - presunutie sa do *Menu pre prehľadávanie*.

Prečítaj parametre tabuľky - načítanie veľkosti tabuľky a dĺžky reťaze z popisného súboru aktuálne zvolenej tabuľky.

4.2.2 Menu pre zadávanie parametrov

V tomto menu sú k dispozícii nasledujúce možnosti:

Ako chcete zadať parametre?
0 - Späť
1 - Všetky naraz
2 - Po jednom

Všetky naraz - aplikácia sa bude porade pýtať na hodnoty jednotlivých parametrov, pričom sa opýta na každý parameter. Táto možnosť je vhodná, ak chceme meniť väčšinu z dostupných parametrov.

Po jednom - prepnutie do menu, v ktorom budeme vidieť aktuálne hodnoty všetkých parametrov a my si volíme, ktorý z týchto parametrov chceme meniť.

4.2.3 Menu pre prehľadávanie

Menu pre prehľadávanie vyzerá nasledovne:
Zadajte spôsob prehľadávania
0 - Späť
1 - Jeden hash (Momentálna hodnota je XX XX XX)
2 - N náhodných hashov
3 - Zmeniť hľadaný hash

Jeden hash (...) - tu je vhodné povedať, že aplikácia používa globálnu premennú pre uloženie jednej hodnoty o veľkosti 24 bitov. Hodnota tejto premennej v šestnástkovej sústave bude zobrazená v tejto položke menu namiesto tých šiestich "X". Zvolením tejto položky sa zavolá funkcia "searchTable", pričom hodnota tejto premennej sa pošle ako hash, ku ktorému chceme v tabuľke nájsť vzor. Táto hodnota sa dá zmeniť zvolením možnosti **Zmeniť hľadaný hash**.

N náhodných hashov - treba najprv zadať celočíselnú hodnotu N. Následne sa postupne generuje N náhodných 24-bitových hodnôt, ktoré sa hľadajú v tabuľke. Priebeh prehľadávania sa zaznamenáva prostredníctvom funkcie "saveToTextfile2" do súboru "prehladavanie.txt", pričom po skončení prehľadávania sa nám zobrazí štatistika úspešnosti hľadania.
Pokiaľ ide o generovanie náhodných reťazí, to vyzerá takto. Vygeneruje sa náhodných 24 bitov a tie sa zahashujú. To, čo funkcia vráti, je potom výsledok tohto hashovania. Dôvod pre takýto postup je ten, že ak by sme zahashovali postupne všetky hodnoty z množiny $\{0,1\}^{24}$, dostaneme len 10 607 959 rôznych hodnôt, čo predstavuje len približne 63% z hodnoty 2^{24}. To teda znamená, že konce reťazí negenerujeme rovnomerne náhodne, a z tohto príkladu nie je možné robiť zovšeobecnenia pre vhodnú voľbu parametrov.

Kapitola 5

Merania

Táto kapitola obsahuje výsledky meraní, ktoré ilustrujú výkon knižnice pri vytváraní tabuliek. Na základe týchto hodnôt by mal byť čitateľ schopný odhadnúť náročnosť operácií počas vytvárania a efekt parametra veľkosti operačnej pamäte. Všetky testy boli robené pre vytváranie tabuliek, pričom parametre "strict_dupl_detection" a "remove_duplicates" boli nastavené na nulu. Prípadné zahodené duplikáty sa nenahrádzali. V tabuľkách sú uvedené len hodnoty vytvárania reťazí a celkové trvanie vytvárania tabuliek (a rozdiel týchto dvoch hodnôt). Čo sa za týmito pojmami skrýva bolo vysvetlené v predchádzajúcej časti tejto práce, a na základe toho si môže čitateľ tieto údaje interpretovať. Počítač, na ktorom boli testy robené, používa procesor Intel Celeron 2.66GHz. Časy sú uvádzané v sekundách.

Ako prvé uvádzam meranie vytvárania tabuľky použitím aplikácie popísanej v predchádzajúcej kapitole. V prípade, že má čitateľ zájem, môže tento test vyskúšať, a porovnať s nameranými údajmi. Použité parametre pre tabuľky boli:

veľkosť tabuľky	=	50000
veľkosť pamäte	=	20000
dĺžka reťaze	=	200

Výsledné merania teda boli:

Číslo	Celkovo	Reťaze	Rozdiel
1	13.641	13.594	0.047
2	13.672	13.625	0.047
3	13.641	13.609	0.032
4	13.625	13.578	0.047
5	13.625	13.579	0.046
Priemer:	13.6408	13.597	0.0438

Druhý test bol robený so 128-bitovou veľkosťou kľúča a šifrového textu a na šifrovanie sa použila funkcia MD5, pričom sa použila rovnaká implementácia ako v našej aplikácii s predchádzajúcej kapitoly. Dĺžka reťaze je jedna, teda sa nerobí žiadna redukcia. Na generovanie reťazí sa používa funkcia z knižnice, teda vytvorenie každej reťaze znamená jedno hashovanie pomocou SHA1 a jedno pomocou MD5. Veľkosť tabuľky je: 3276800 reťazí, teda presne 100MB. Veľkosť poľa je K násobok hodnoty 327680.

K:	**1**		**2**		**10**	
Číslo	**Celkovo**	**Reťaze**	**Celkovo**	**Reťaze**	**Celkovo**	**Reťaze**
1	54.360	26.891	41.218	25.953	31.484	26.937
2	43.547	25.829	40.203	26.250	31.125	26.859
3	46.656	25.890	46.656	31.891	31.437	26.906
4	41.203	25.500	41.265	26.000	31.434	26.906
5	43.766	25.714	41.453	26.000	32.531	26.937
Priemer:	45.9054	25.9648	42.159	26.2188	31.6022	26.909

Literatúra

[1] Philippe Oechslin, Making a Faster Cryptanalytic Time-Memory Trade-Off, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science 2729 Springer 2003, ISBN 3-540-40674-3

[2] M. E. Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT-26:401-406, 1980

[3] L. Peter Deutsch, Independent implementation of MD5 (RFC 1321), v 1.6, http://kent.dl.sourceforge.net/sourceforge/libmd5-rfc/md5.tar.gz

[4] Steve Reid, SHA1 in C, http://www.mirrors.wiretapped.net/security/cryptography/hashes/sha1/sha1.c

Dodatok A

Návratové hodnoty funkcií a používané súbory

A.1 Funkcie vytvárania a rozširovania

Hodnota	Popis
0	Všetko v poriadku.
1	Hodnota premennej *array_size* je pod minimálnou povolenou hodnotou.
2	Nepodarila sa alokácia miesta v operačnej pamäti.
3	Problém pri otváraní niektorého súboru.
4	Niektorý z parametrov bol zle zvolený.
5	Tabuľka neexistuje[10].
6	Nebol nájdený popisný súbor pre danú tabuľku[10].

Ak funkcia vráti hodnotu 4, nastal niektorý z nasledujúcich prípadov:

- Názov tabuľky je prázdny reťazec.
- Veľkosť tabuľky nie je kladné číslo.
- Dĺžka reťaze nie je kladné číslo.
- Veľkosť kľúča alebo šifrového textu nie je kladné číslo.
- Veľkosť kľúča alebo šifrového textu nie je deliteľná ôsmimi.
- Ukazovateľ na šifrovaciu alebo redukčnú funkciu má hodnotu ”NULL”.

[10] Iba v prípade zväčšovania tabuľky.

A.2 Prehľadávanie

Hodnota	Popis
0	Kľúč bol nájdený a všetko bolo v poriadku.
1	Kľúč bol v tabuľke nájdený, ale nepodarilo sa ho zapísať do pamäte.
2	Súbor obsahujúci tabuľku sa nepodarilo otvoriť.
3	Kľúč sa nepodarilo nájsť, ale popisný súbor tabuľky bol k dispozícii.
4	Kľúč, ani popisný súbor tabuľky sa nepodarilo nájsť.

A.3 Čítanie popisného súboru

Hodnota	T	P	Popis
0	A	A	Všetko v poriadku.
1	N	A	Prečítajú sa informácie zo súboru ale nenastaví sa veľkosť kľúča.
2	A	N	Veľkosť tabuľky sa vypočíta na základe poslaných informácií o veľkosti kľúča a šifrového textu.
3	N	N	Nevykoná sa žiadna akcia.

Stĺpce T a P hovoria o tom, či sa podarilo nájsť súbor obsahujúci tabuľku, respektíve popisný súbor. Hodnota A znamená áno, N znamená nie.

A.4 Súbory používané pri vytváraní a rozširovaní tabuliek

<**table_name**>**.rbt** Súbor, ktorý obsahuje utriedené reťaze tvoriace tabuľku.

<**table_name**>**2.rbt** Popisný súbor tabuľky.

<**table_name**>**1.tmp** Súbor, ktorý obsahuje utriedené bloky reťazí.

<**table_name**>**1a.tmp** Súbor, ktorý obsahuje veľkosti jednotlivých blokov zo súboru <table_name>1.tmp.

<**table_name**>**2.tmp** Pri presýpaní čítame reťaze zo súboru <table_name>1.tmp a zapisujeme ich do tohto súboru.

<**table_name**>**2a.tmp** Súbor, ktorý obsahuje veľkosti jednotlivých blokov zo súboru <table_name>2.tmp.

Dodatok B

Meranie času

Na meranie času častí niektorých funkcií sa používa štruktúra my_time. Jej dve premenné sú ”sec” a ”millisec”, ktorých význam je zrejmý. Táto štruktúra je definovaná v knižnici ”my_timer”, ktorá navyše ponúka niekoľko funkcií ktoré, s touto štruktúrou pracujú. Tu uvádzam stručný prehľad týchto funkcií:

void now(my_time *m) Do premennej ”m” uloží aktuálny čas.

void resetTime(my_time *m) Obom premenným ”m” nastaví hodnotu nula.

void timeDifference(my_time m1, my_time m2, my_time *m3) Do ”m3” sa zapíše čas, ktorý uplynul od ”ml” po ”m2”.

void timeSum(my_time *m1, my_time m2) Do ”m1” sa pripočíta čas ”m2”.

void timeToString(const my_time* m, char* buffer) Na adresu ktorú predstavuje premenná ”buffer” sa zapíše čas ”m” ako text.