VILNIAUS KOLEGIJA
ELEKTRONIKOS IR INFORMATIKOS FAKULTETAS

Hash algoritmai

DOKUMENTACIJA

653I10001 PIN13

STUDENTAS Darius Juodokas

(parašas)

2014-05

DĖSTYTOJAS Rimvydas Motiejūnas

(parašas)

2014-05

2014

Turinys

1. Įvadas

Visos kompiuterinės programos dirba naudodamos tuos pačius pagrindinius kompiuterio resursus: centrinį procesorių (CPU), operatyviąją atmintį (RAM) - ir pastovąją atmintį. Pastoviojoje atmintyje saugomas programos paleidžamasis failas. Šį paleidus procesorius apdoroja pradinius programos duomenis ir parengia kompiuterio komponentus programos vykdymui: rezervuojama operatyvioji atmintis, sukuriami proceso aprašai, įkeliami duomenys į operatyviąją atmintį, priskiriami įvesties-išvesties įrenginiai ir t.t. Aktyvus (vykdomas) procesas visada turi būti įkeltas į operatyviąją atmintį. Ko gero „sunkiausia“ – daugiausia vietos užimanti – proceso dalis yra kintamieji, t.y. jiems priskirtos reikšmės. Sukuriant kintamuosius (šiame darbe bus remiamasi C++ programavimo kalba) pirmiausia jie turi būti aprašomi: nurodomas tipas, kintamojo pavadinimas; tada – priskiriamos reikšmės. Toks kintamasis rezervuoja tam tikro dydžio atminties bloką operatyviojoje atmintyje. Rezervuoto bloko dydis priklauso nuo kintamojo tipo: pvz.: 32 bitų sistemoje *int* (integer – paprastas sveikasis skaičius) užima 4 baitus, *double* – 8 baitus, *char* – 1 baitą, rodykle (**pt*) – 4 baitus ir t.t. Jei formuojami kintamųjų masyvai – rezervuojami vientisi atminties blokai, kurių dydis tiesiogiai priklauso nuo masyvo dydžio ir tipo.

Šiame darbe bus aptariamos dvi problemos, glaudžiai susijusios su masyvais. Masyvas – puiki priemonė saugoti kintamuosius, be kurios būtų sunku išsiversti. Tačiau ši priemonė turi vieną didžiulį minusą – ji yra ribota. Hipotetinėje situacijoje suprogramuojamas *int* tipo masyvas su 4 elementais, kuriems priskiriamos atsitiktinės reikšmės. Programos paskirtis – patikrinti, ar tarp priskirtų reikšmių yra skaičius 13. Tai padaryti nesunku panaudojant *for* ar *while* ciklą ir patikrinant kiekvieną masyvo elementą, ar jis lygus 13.

Pateiktoje hipotetinėje programoje problemos nėra. Ji atsiranda tada, kai masyvas yra ne 4 elementų dydžio, o dešimčių ar šimtų tūkstančių elementų. Jei tokiame masyve skaičius 13 yra paskutinis elementas, programa turės sutikrinti visą masyvą, o tai reiškia, kad bus sugaišta nemažai laiko – CPU ciklų -, turint omeny, kad reikės nuodugniai patikrinti didžiulius klodus RAM atminties.

Kita problema atsiranda tada, kai norima sukurti masyvus didesnius, nei maksimalus leistinas dydis, t.y. didesnius, nei didžiausia skaitinio duomenų tipo reikšmė. Kitoje hipotetinėje programoje siekiama aprašyti masyvą ir jame talpinti visų pasaulio kalbų žodynus. Nėra tokio duomenų tipo, kuris leistų saugoti tokią didžiulę skaitinę reikšmę. O jei ir būtų, susidurtume su

didžiule problema randant norimą žodį, nes lyginant visas masyvo reikšmes su norimu žodžiu programa truktų, geriausiu atveju, kelias valandas.

Šios dvi problemos yra glaudžiai viena su kita susijusios ir riboja masyvų naudojamumą. Kuo „didesnio“ tipo (pvz.: *int* yra mažesnis už *string*) ir kuo daugiau elementų turi masyvas, tuo daugiau procesoriaus ciklų programa turės pereiti, kad patikrintų visą rezervuotą atmintį ir rastų ieškomą frazę. Šiame darbe aptarsime masyvų sudarymo būdą, iš dalies išsprendžiantį abi šias problemas – hash lentelių sudarymą.

2. Užduoties formulavimas

Šio darbo užduotis – rasti, išanalizuoti ir palyginti veiksmingumą įvairių algoritmų, leidžiančių apdoroti didelius duomenų kiekius, kuriems apdoroti nepakanka paprastų vienmačių masyvų. Konkrečiau – bus orientuojamasi į HASH sekų sudarymą ir jų panaudojimą kuriant HASH lenteles.

HASH – tai vienkrypčio kodavimo tipas, kuriuo užšifruojami duomenys negali būti iššifruoti. HASH sekos glaudžiai susijusios su kompiuterine sauga, nes jos naudojamos saugoti duomenims, kurių paskirtis – sulyginimas. Pavyzdžiui duomenų bazėse saugomi elektroninės sistemos slaptažodžiai yra užšifruoti tam tikrais destruktyviais algoritmais, todėl jų iššifruoti neįmanoma. Sistemos prisijungimo laukelyje įvedus slaptažodį šis yra taip pat užšifruojamas naudojant tą patį algoritmą ir sulyginamas su šifru, saugomu duomenų bazėje. Tik esant 100% abiejų šifrų sutapimui laikoma, kad slaptažodis teisingas.

HASH lentelės sudaromos iš dalies tuo pačiu principu. Naudojant pasirinktą algoritmą sugeneruojami HASH raktai, kurie turi būti kiek įmanoma atsitiktiniai. Šie raktai naudojami kaip masyvo indeksai. Įprastai naudojamas masyvo aprašas atrodo taip: *masyvas[indeksas] = reikšmė*, kur indeksai numeruojami iš eilės, paprastai nuo nulio iki reikšmės, lygios masyvo ilgiui. Naudojant HASH raktus masyvas atrodys šitaip: *masyvas[HASH_raktas] = reikšmė*. Iš pažiūros abi išraiškos skiriasi lyg ir labai nedaug, tačiau antrasis variantas – žymiai galingesnis įrankis siekiant saugoti indeksuotus duomenis atmintyje. Dažnai HASH lentelės būna sudarytos ne iš vienmačių, o iš daugiamačių masyvų. HASH raktą, priklausomai nuo jo ilgio, galima skaidyti į dalis, pvz.: 10 simbolių ilgio HASH raktą 0123456789 galima skaidyti į tris dalis: 0123, 456, 789 ir pasiekti trimačio masyvo elementą, esantį adresu *array[123][456][789]* . Naudojant HASH raktus masyvo elemento indeksas parenkamas atsitiktinai, tačiau yra nekintantis, t.y. keliskart koduojant tą pačią koduojamą eilutę gaunamas atsitiktinis, bet kiekvieną kartą tas pats šifras. Taip pat šifras turi būti sugeneruojamas ne tik atsitiktinis, bet ir kiek įmanoma unikalus bet kokiai koduojamai simbolių eilutei. Priešingu atveju pateikus programai dvi nieko bendro tarpusavyje neturinčias eilutes gali būti, kad gausime nuorodą į tą pačią atminties vietą, o tai jau yra blogo programos veikimo požymis. Kad taip nenutiktų, šifravimui reikia pasirinkti gerą algoritmą – gerą HASH funkciją.

Kursinio darbo pristatymo metu bus pristatoma programa, sugebanti užšifruoti norimą eilutę keliais skirtingais vienkrypčio šifravimo algoritmais. Programos tikslas – palyginti kelis skirtingu principu veikiančius ir skirtingas išvestis generuojančius HASH sekų sudarymo algoritmus, pademonstruoti jų teigiamas ir neigiamas savybes, įvertinti kiekvieno algoritmo spartą ir kitas savybes. Kursinio darbo tikslas – palyginti minėtus algoritmus, gyvai įliustruoti jų veikimą

specialiai tam sukurta programa ir įvertinti algoritmų tinkamumą įvairioms situacijoms. Programa rašoma C++ kalba naudojant ***Code::Blocks*** programinę įrangą ir GCC kompiliatorių. Dvejetainis programos failas pritaikytas veikti Linux operacinėje sistemoje, kurioje naudojamas procesoriaus žodžio ilgis yra 32 bitai. Programos kodas paliekamas laisvam redagavimui ir perkompiliavimui skirtingoms architektūroms.

3. Užduoties analizė

Užduočiai įgyvendinti buvo parašyta programa, įliustruojanti iškeltų problemų sprendimą. Programa sudaryta iš dviejų pagrindinių dalių. Pirmoji – vienkrypčio šifravimo algoritmai, surašyti antraštiniuose C++ failuose. Patys algoritmai buvo paimti iš įvairių internetinių šaltinių. Žemiau pateikiamas sąrašas naudotų algoritmų ir šaltinių, iš kurių jie paimti:

- CityHash32 - https://github.com/hrydgard/native/tree/master/ext/cityhash
- CityHash64 - https://github.com/hrydgard/native/tree/master/ext/cityhash
- CityHash128 - https://github.com/hrydgard/native/tree/master/ext/cityhash
- MurmurHash3_x86_32 - https://github.com/PeterScott/murmur3
- MurmurHash3_x86_128 - https://github.com/PeterScott/murmur3
- MurmurHash3_x64_128 - https://github.com/PeterScott/murmur3
- lookup3 - https://code.google.com/p/smhasher/source/browse/trunk/lookup3.cpp
- SuperFastHash - https://github.com/chrisvana/smhasher_copy/blob/master/SuperFastHash.cpp

Antroji programos dalis – main.cpp – apvalkalas, kuris apjungia visas antraštes, naudoja algoritmus ir atlieka pasirinktus testus. Main.cpp taip pat turi ir antraštinį failą, kuriame suvienodinamos visos šifruojančios funkcijos, t.y. surašytos funkcijos, atitinkančios algoritmų pavadinimus, paimančios tuos pačius parametrus. Main.cpp buvo sukurtas funkcijų rodyklių masyvas, saugantis adresus į visas prieš tai minėtas funkcijas. Toks programos kūrimo stilius leido šiek tiek optimizuoti kodą ir sumažinti vykdomojo failo dydį, nes nereikėjo aprašinėti kiekvieno algoritmo funkcijos iškvietimo atskirai. Funkcijos tiesiog iškviečiamos kiekvienos ciklo iteracijos, dedikuotos kiekvienam algoritmui, metu. Šis mechanizmas tam tikromis sąlygomis, kurios bus aptartos vėliau, yra neefektyvus ir netgi žalingas, todėl programoje buvo palikti kodo blokai, atskirai iškviečiantys kiekvieną algoritmą (neaudojant ciklo). Buvo padidintas programos vykdomojo failo dydis, tačiau padidėjo ir suderinamumas.

Pagrindinė programos dalis (main.cpp) sudaryta iš pagrindinės funkcijos (main ()) ir devynių papildomų funkcijų. Kai kurios iš jų pavadintos panašiais pavadinimais, nes atlieka labai panašius veiksmus, t.y. paima tuos pačius duomenis, juos apdoroja ir grąžina tuos pačius rezultatus. Šios funkcijos tarpusavyje skiriasi rezultato skaičiavimo algoritmu. Esant reikalui galima iškviesti tiek vieną, tiek kitą – jai analogišką – funkciją. Funkcijoms iškviesti buvo sudarytas elementarus meniu, iškviečiamas iš main() funkcijos, paleidus programą. MENU() išveda į ekraną elementarų meniu su pasirinkimo variantais: (1) nustatymai; (2) išvesti šifrą į ekraną; (3) atlikti greičio testą;

(4) atlikti žodyno testą. 2, 3 ir 4 meniu elementai turi atitikmenis: 22, 33 ir 44, kurie iškviečia analogiškas funkcijas, tik naudojančias skirtingus algoritmus.

Pirmasis meniu punktas, „Nustatymai“, leidžia programos veikimo metu koreguoti kai kuriuos programos parametrus. Dalis parametrų tiesiogiai įtakoja šifro generavimą, dalis – tik pagalbinės priemonės. Tiesiogiai šifro generavimą įtakoja eilutė, kuri bus šifruojama, ir „seed“ kintamasis, naudojamas algoritmuose. Su šifro sudarymu tiesiogiai nesusiję parametrai yra: „loopCount“ kintamasis, nurodantis, kiek ciklo iteracijų bus vykdomas eilutės šifravimas greičio testo metu; pasirinktis, ar žodymo testo metu užšifruotas eilutes saugoti į failus. Pastarasis paramatras buvo įdiegtas tam atvejui, jei bus poreikis analizuoti sudaytas eilutes toliau.

Antrasis meniu punktas išveda užšifruotas eilutes į ekraną – į standartinę išvestį. Naudojami visi algoritmai, aprašyti main.h faile. Su kiekvienu algoritmu užkoduojama testinė eilutė ir į ekraną išvedama:

- HASH algoritmo pavadinimas;
- Užšifruota testinė eilutė.

Trečiasis meniu punktas iškviečia funkciją, atliekančią kiekvieno algoritmo greičio testą. Greičio testas atliekamas leidžiant kiekvieną algoritmą *n* kartų ir užkoduojant tą pačią eilutę. Kintamojo *n* reikšmę galima keisti, nes programos veikimas priklauso nuo kompiuterio, kuriuo bus leidžiama programa, techninių savybių – čia daugiausia įtakos turi procesorius. Algoritmo greitį apskaičiuojanti funkcija paima šiuos kintamuosius:

- Koduosimą eilutę;
- Koduosimos eilutės ilgį;
- „*seed*“ reikšmę, nulemiančią algoritmo rezultatą.

Koduosimos eilutės ilgis funkcijai paduodamas atskirai, kad funkcijai reikėtų atlikti kiek įmanoma mažiau pašalinių skaičiavimų ir kuo daugiau procesoriaus ciklų būtų skiriama pačiam kodavimo algoritmo vykdymui.

Ketvirtasis meniu punktas iškviečia paskutinįjį testą – žodyno testą. Šio testo metu nuskaitomas žodyno failas, kuriame kiekvienoje eilutėje įrašytas vienas žodis. Žodyne neturi būti dviejų identiškų žodžių, priešingu atveju bus iškreipti testo rodmenys. Šis testas parodo algoritmo kokybiškumą, t.y. kaip labai skirtingų žodžių šifrai yra atsitiktiniai. Jei testo metu randami du vienodi šifrai, traktuojama, kad įvyko kolizija. Testo pabaigoje atliekamas kolizijų skaičiavimas. Kuo mažiau kolizijų – tuo šifravimo algoritmas yra kokybiškesnis. Verta paminėti šios funkcijos analogą – meniu pasirinktis 44. Pasirinkus šį punktą bus atliekamas to paties tipo testas, tačiau

visiškai kitu algoritmu, ir kolizijų skaičius į ekraną išvestas nebus. Kolizijų skaičiavimui šioje programoje naudojamas *std::map* įrankis, sukeliantis duomenis į laikinąją atmintį. Kadangi nuskaitomas žodyno failas yra neapibrėžto dydžio, laikinosios atminties (RAM) taupymo sumetimais kolizijų skaičiavimas automatiškai nėra atliekamas. Šį skaičiavimą galima atlikti rankiniu būdu naudojant UNIX ar į UNIX panašias operacines sistemas, pvz.: Linux. Naudojant įrankius *for,cat, sort, uniq* galima rasti kiekvieno algoritmo kolizijas.

Pavyzdys:

```
for i in *; do echo $i; cat $i|sort|uniq -d|wc -l; echo; done
```

Šią komandą reikia leisti kataloge, kuriame pograma sugeneravo algoritmų išvesitis (meniu pasirinktis 44 visada generuoja išvestis į failus), t.y. ***<kelias_iki_programos_vykdomojo_failo>/outp/*** . Čia bus saugomos sugeneruotos algoritmų išvestys failuose, kurių pavadinimai atitiks algoritmų pavadinimus. Failai bus be plėtinių, naudojamas LINUX sistemoms būdingas eilučių pabaigos žymėjimas: „\n“.

Kai kurie kintamieji, siekiant padidinti programos naudojimo patogumą ir stabilumą, turi iš anksto nustatytas pradines reikšmes. Visus šias reikšmes galima koreguoti programos veikimo metu pasirinkus meniu punktą 1. Numatytosios reikšmės:

Kintamasis	Paaiškinimas	Reikšmė
test	Testinė eilutė, kuri bus užkoduota	„Hello World!“
loopCount	Skaičius, nurodantis, kiek kartų bus vykdomas kiekvieno algoritmo iškvietimas algoritmo greičio testo metu	999 999
seed	Skaitinė reikšmė, galinti pakeisti šifravimo algoritmo sugeneruotą rezultatą	42
s2f	(angl. „save-to-file“) nurodo, ar pasirinkus meniu punktą „4“ duomenys bus saugomi į failą	true
DICT_FILE	Žodyno failas, naudosimas programos veikimo metu, atliekant žodyno testą	162282 žodžių

Programos veikimo pradžioje paleidžiama dar viena funkcija, kuri nebuvo paminėta anksčiau. Ši funkcija patikrina programos antračtinio failo turinio integralumą, t.y. ar duomenys įvesti teisingai ir programa galės juos naudoti be trikčių. Ši funkcija anksčiau nebuvo paminėta, nes ji neturi tiesioginio poveikio algoritmams ir jų testams. Yra vienintelis atvejis, kada ši funkcija neleis tęsti programos vykdymo – tai perkompiliuota programa paredagavus main.h antraštinį failą.

Visoms šioms funkcijoms veikiant darniai gaunama pageidaujama išvestis, padėsianti įvertinti pasirinktų algoritmų efektyvumą. Galimybė keisti šifruotiną eilutę leis patikrinti, kaip algoritmai elgiasi, kai šifruojami tik skaičiai; kaip jie elgiasi, kai šifruojamos trumpos eilutės; kai ilgos; kai šifruojami nestandartiniai simboliai ar ne ekrane nespausdinami simboliai, pvz.: „*n*“. Šių funkcijų vykdymo metu bus galima išsiaiškinti, kurie algoritmai efektyvesni naudojant vieno ar kito pobūdžio duomenis.

4. Programinės dalies aprašymas

Kaip jau minėra ankstesniame skyriuje, programa sudaryta iš dviejų esminių dalių: algoritmų ir apvalkalo. Algoritmų dalį sudaro HASH kodavimo algoritmai, surašyti atskiruose antraštiniuose failuose: *algorithms/<algoritmo_pavadinimas>.h*. Šiuose failuose surašytos funkcijos, atliekančios joms paduotos eilutės užšifravimą, ir grąžinančios šifrą. Visos funkcijos, t.y. visi algoritmai grąžina *unsigned int* (*uint*) tipo kintamąjį, tačiau skiriasi jo išraiška.

uint32_t tipo kintamąjį grąžina šie algoritmai:

- CityHash32
- lookup3
- SuperFastHash

uint64_t tipo kintamąjį grąžina CityHash64 algoritmas. Tuo tarpu Murmur algoritmai grąžina atitinkamai *uint32_t* ir *uint64_t* tipo masyvus, sudarytus iš 4 elementų. Verta paminėti, kad Murmur3_x86_32 algoritmas grąžina 4 narių *uint32_t* tipo masyvą, tačiau tik pirmas narys yra efektyvus, t.y. likę trys nariai beveik visuose šifruose lieka atitinkamai tapatūs.

1. main.h antraštės paskirtis

Algoritmų antraštės yra įtraukiamos ne į main.cpp, o į main.h. Tai padaryta tikslingai, turint omeny, kad vartotojas gali pageidauti į testą įtraukti daugiau algoritmų. Tokiu atveju jam visiškai nereikės redaguoti pagrindinio programos failo main.cpp ir gilintis į programos veikimo mechanizmus. main.h antraštiniame faile kiekvienam algoritmui yra parašyti apvalkalai, suvienodinantys šifravimo funkcijas. Kadangi ne visi algoritmai paima tokį patį rinkinį kintamųjų, apvalkalas yra parašytas taip, kad programa galėtų iškviesti visus algoritmus, pateikdama jiems lygiai tuos pačius kintamuosius. Apvalkalas atrodo taip:

***std::string** AlgoritmoPavadinimas (**std::string** koduojama_eilutė, **int** eilutės_ilgis, **uint32_t** seed)*

Apvalkalo viduje atliekamas minimalus kiekis operacijų, kad kuo mažiau paveiktų skaičiavimų greitį. Ši funkcija turi atlikti visus reikalingus skaičiavimus ir duomenų keitimus, kad galėtų iškviesti HASH fukncją su pateiktais parametrais, ir pagrindinei programai grąžintų sugeneruotą *string* tipo atsakymą. Tam, kad duomenų pervertimas būtų paprastesnis, main.h yra įtrauktos dar kelios funkcijos ir *stringstream* tipo kintamasis:

```
std::string uint32ToString(uint32_t number);
std::string uint64ToString(uint64_t number);
std::stringstream strBuff;
```

uint32ToString ir uint64ToString atitinkamai verčia *uint32_t* ir *uint64_t* tipo skaitines reikšmes į *string* eilutę. strBuff kintamasis sukurtas kaip talpykla, palengvinanti operacijas su eilutėmis, pvz.: jas sujungianti - buvo panaudota Murmur3 apvalkaluose.

main.h yra du masyvai, kurie visada privalo turėti tiek pat narių. Jei keičiamas main.h failas ir pridedami ar šalinami šifravimo algoritmai, abu šie masyvai taip pat turi būti koreguojami atitinkamai:

*std::string (*str_Fpt[])(std::string what, int length, uint32_t seeding)*

const char ALGORITHMS[]*

Šie masyvai įgalina programą veikti dinamiškai, t.y. lengvai įtraukti naujus algoritmus ar pašalinti esamus. Į masyvą *ALGORITHMS* įvedami *string* tipo elementai, atspindintys algoritmo pavadinimą. Į **str_Fpt[]* masyvą tuo pačiu indeksu turi būti įvedamas atitinkamo algoritmo funkcijos-apvalkalo pavadinimas (be kabučių). Pastarasis masyvas yra sudarytas iš rodyklių į funkcijas, kuriomis bus užkoduojama simbolių eilutė. Šios funkcijos yra naudojamos kiekvieno testo metu: į ekraną išvedamas algoritmo pavadinimas ir iškviečiama jį atitinkanti funkcija. Algoritmo pavadinimas paimamas iš *ALGORITHMS[]* masyvo, todėl šie du masyvai turi būti tokio paties ilgio, o jų elementai – tarpusavyje atitikti vienas kitą, nes priešingu atveju programa nepasileis. Tam, kad būtų išvengta netyčinių klaidų ir programa pasileistų korektiškai, šių masyvų ilgiai yra sulyginami – tai atlieka funkcija *int checkIntegrity()*. Jei masyvai yra skirtingų ilgių, programa išves perspėjimą apie šią klaidą ir išsijungs grąžindama reikšmę „1“ (*exit(1)*).

2. main.cpp funkcijos ir kintamieji

main.cpp – tai pagrindinis programos failas, kuriame surašytos būtinos programos veikimui funkcijos, sukuriamas pagrindinis meniu, nustatymų meniu, aprašyti ir iškviečiami šifravimo algoritmų testai. Kodą sudaro 8 funkcijos, neįtraukiant į šį skaičių *int main()*, ir 8 globalūs kintamieji.

Du kintamieji – *timeBefore* ir *timeAfter* – yra *clock_t* tipo ir naudojami apskaičiuojant kiekvieno algoritmo greitį. Į *timeBefore* yra išsaugoma programos vykdymo laiko reikšmė prieš atliekant kurio nors algoritmo testą. Pasibaigus algoritmo testui išsaugomas laikas į *timeAfter*. Iš pastarojo reikšmės atimama *timeBefore* reikšmė ir apskaičiuojamas skirtumas – gaunama, kiek laiko buvo užtrukta, kol programa įvykdė kodo dalį, esančią tarp šių dviejų laiko vertės priskyrimų *clock_t* kintamiesiems.

Kintamasis *std::map<string, int> wordCounts* yra naudojamas vienoje iš šifravimo algoritmų greičių matavimo funkcijoje. Jo pagrindinė paskirtis – išsiaiškinti, kiek žodyne yra

neunikalių, t.y. pasikartojančių žodžių. Šiuo konkrečiu atveju žodynas yra užšifruotas kuriuo nors HASH algoritmu.

Du kintamieji – *uint32_t seed* ir *string test* yra naudojami visuose testuose. Jie, t.y. jų reikšmės, yra pateikiamos algoritmų funkcijų apvalkalams, aprašytiems main.h faile (apie apvalkalus buvo minima ankstesniame skyriuje). Šių kintamųjų pradinės reikšmės yra:

uint32_t seed = 42;

string test = "Hello world!";

seed reikšmė naudojama šifravimo algoritmuose šifrui pakeisti. Turint žinomą *seed* reikšmę galima tikslingai gauti identiškus šifrus naudojant skirtingas šifruotinas simbolių eilutes. Jei yra poreikis, sudarant HASH lenteles galima įterpti dar vieną matmenį, kurio elementų indeksai nulems *seed* reikšmę; taip bus praplėsta duomenų lentelė, duomenys bus kiek saugesni, nes turint tik šifrą be seed reikšmės nebus galima lengvai atspėti, kas kokia simboliu eilutė užkoduota, o duomenų pasiekimo greitis pasikeis tik nežymiai.

test kintamasis – tai simbolių eilutė, kurią programa užšifruos HASH algoritmais. Ši eilutė naudojama tiek paprastai išvedant šifrus į ekraną, tiek atliekant greičio testą. Kintamojo reikšmę galima keisti pagal naudotojo pageidavimus. Eilutę gali sudaryti bet kurie regimieji simboliai, įskaitant ir tarpo bei tabuliatoriaus simbolius. Taip pat palikta galimybė palikti eilutę tuščią.

Likę du kintamieji yra: *bool s2f* ir *long long loopCount*. Jų pradinės reikšmės yra atitinkamai *false* ir 999999. Kintamasis *s2f* yra naudojamas žodyno testą atliekančioje funkcijoje ir nulemia, ar užšifruotos žodyno eilutės bus išsaugotos į failus. Pagal nutylėjimą eilutės į failus neišvedamos, nes tai neigiamai įtakoja testo greitį. Įvesties-išvesties į kietajį diską operacijos trunka sąlyginai ilgai, todėl tikimasi, kad išvedimas į failą nebus būtinas. Toliau bus paaiškinama, kodėl paliekama ši pasirinkimo galimybė.

loopCount kintamasis nurodo, kiek kartų bus įvykdomi šifravimo algoritmai greičio testo metu. Kuo šio kintamojo reikšmė mažesnė, tuo greičiau bus baigtas žodyno testas. Kaip ir visi kompiuteriniai testai, ši programa yra priklausoma nuo fone veikiančių kitų procesų. Gali nutikti taip, kad nustačius *loopCount* reikšmę į mažesnę foninės programos pareikalaus daugiau procesoriaus ciklų, kas turės neigiamos įtakos vykdomam testui ir iškreips rodmenis. Dėl šios priežasties, norint gauti rodmensi su ko mažesne paklaida, reikėtų neapsiriboti mažomis *loopCount* reikšmėmis ir palikti numatytąją, ar net nustatyti didesnę.

Programą sudaro 9 funkcijos, iš kurių kelios atlieka tą patį darbą, tik kitais algoritmais. Pagrindinėje programos funkcijoje - *main()* – atliekamas programos nustatymų patikrinimas.

Iškviečiama funkcija *checkIntegrity()*, aprašyta main.h faile. Jei masyvų *ALGORITHMS[]* ir **str_Fpt[]* elementų skaičius nesutampa, laikoma, kad programos nustatymai yra neteisingi ir tolesnis jos vykdymas nutraukiamas įvykdant komandą *exit(1)*. Jei programos vykdymas nenutraukiamas, funkcija *main* priskiria ankščiau minėtiems kintamiesiems pradines reikšmes ir įeina į pagrindinį programos ciklą *while(loopIt)*, kuriame aprašyta *switch* konstrukciją, iškviečianti funkciją *MENU()* ir nuskaitanti jos grąžintą reikšmę – pasirinktą meniu punktą.

Pasirinkus meniu punktą „1“ patenkama į antrą – nustatymų – meniu. Čia galima pakeisti kai kurių globalių programos kintamųjų reikšmes, kurios paveiks testų rodmenis ar šifravimo rezultatus.

Meniu punktas „2“ turi alternatyvą – „22“. Abu šie pasirinkimai iškviečia funkcijas, kurios išveda į ekraną visais algoritmais užkoduotas *test* eilutes. Funkcijos tikslas – leisti vartotojui įvertinti, kaip kiekvienas algoritmas elgiasi jam pateikus įvairias koduotinas eilutes, įvertinti šifrus, jų ilgius ir palyginti juos tarpusavyje. Pirmuoju variantu – pasirinkus „2“ – bus iškviesta funkcija *showAll2* ir jai pateikti du parametrai: šifruotina eilutė *test* ir *seed* reikšmė. *showAll2* kiekvieną masyvo *ALGORITHMS[]* elementą traktuos kaip atskirą algoritmą ir įvykdys atitinkamą funkciją, kurios rodyklė įrašyta **str_Fpt[]* masyve atitinkamu indeksu, pateikdama jai *test* reikšmę, simbolių skaičių ir *seed* reikšmę. Ši funkcija visiškai pasikliauja vartotoju ir tikisi, kad minėtų masyvų atitinkami indeksai minės atitinkamus algoritmus ir nebus sumaišyti. Tuo tarpu antruoju variantu – „22“ – vartotoju nėra aklai pasikliaujama ir programa, nesikreipdama į masyvus, tiesiogiai iškviečia šifruojančių algoritmų funkcijų apvalkalus, aprašytus main.h faile. Į standartinę išvestį išvedami tie patys duomenys, kaip ir pirmuoju pasirinkimu, tačiau skiriasi mechanizmai, kaip tie duomenys gaunami.

Meniu punktas „3“ taip pat turi savo analogą – „33“. Abiem atvejais iškviečiama funkcija – atitinkamai *showSpeed2* ir *showSpeed* -, kuriai pateikiami trys parametrai: *test*, *seed* ir *loopCount*. Šios pasirinkties tikslas – apskaičiuoti, kokie yra santykiniai kiekvieno šifravimo algoritmo veikimo greičiai. Kiekvieno algoritmo apvalkalas bus iškviestas tiek kartų, kokia yra funkcijai pateikto kintamojo *loopCount* reikšmė. Prieš testuojant kiekvieną algoritmą *timeBefore* priskiriamas esamas programos veikimo laikas; tada atliekamas šifravimas tiek kartų, kiek buvo pageidauta; galiausiai pasibaigus šifravimui *timeAfter* vėl priskiriamas programos veikimo laikas. Skirtumas tar abiejų šių reikšmių – laikas, per kurį buvo užšifruota eilutė *loopCount* kartų. Apskaičiuotas skirtumas padalinamas iš *CLOCKS_PER_SEC* reikšmės ir į ekrana išvedamas rezultatas – laikas sekundėmis. Funkcijos *showSpeed* ir *showSpeed2* tarpusavyje skiriasi tais pačiais požymiais, kaip ir *showAll* nuo *showAll2*: pirmoji atlieka skaičiavimus nepriklausomai nuo anksčiau minėtų masyvų, antroji – naudodama tik masyvus. Tikėtina, kad „33“ (*showSpeed*) veiks kiek greičiau, nes bus atliekama

šiek tiek mažiau skaičiavimų, tačiau skirtumas neturėtų būti ryškus. Daugiausia pašalinės įtakos greičiui turės šie faktoriai:

- fone veikiančių procesų prioritetai;
- fone veikiančių procesų būsenos;
- kompiuterio procesoriaus:
    - architektūra;
    - taktinis dažnis;
    - branduolių skaičius;
    - branduolių savybės (virtualūs ar fiziniai; jei fiziniai – atstumas tarp branduolių);
- RAM atminties:
    - laisvos atminties kiekis absoliučiais vienetais;
    - laisvos atminties kiekis santykiniais vienetais (jei likę mažai atminties, OS branduolys gali iškviesti puslapiavimo servisą, kuris pradės laisvinti RAM atmintį ir ženkliai padidins sistemos apkrovą).

Dėl šių priežasčių negalima laikyti, kad greičio testo rezultatai yra absoliutūs. Gautus rezultatus reikia traktuoti kaip santykinius ir lyginti tarpusavyje, pvz.:: „vienas algoritmas lėtesnis už antrą, bet greitesnis už trečią du kartus“. Verta paminėti, kad reikia vengti mažų *loopCount* reikšmių. Kuo trumpiau bus tikrinamas kuris nors algoritmas, tuo didesnė tikimybė, kad tuo metu foninis procesas, sukėlęs sistemos apkrovą, iškreips testo rezultatus. Kuo ilgiau bus vykdomas testas, t.y. kuo didesnė bus *loopCount* reikšmė, tuo aiškesni ir tikslesni bus rezultatai.

Meniu punktas „4“ arba „44“ paleis atitinkamą funkciją *dict2* arba *dict*, atliksiančią žodyno testą. Žodyno testo paskirtis – išsiaiškinti, kiek kokybiškas yra algoritmas, t.y. kiek atsitiktinių skirtingų simbolių sekų algoritmas gali užkoduoti sugeneruodamas skirtingus šifrus. Tai ypatingai svarbus algoritmo vertinimo kriterijus, nes kuo dažniau algoritmas skirtingiems žodžiams sugeneruos tapačius šifrus (toks reiškinys vadinamas kolizija), tuo lėtesnė bus HASH lentelė. Nėra algoritmų, kurie visiškai negeneruotų kolizijų, todėl kuriant HASH lenteles taikomi įvairūs mechanizmai, kaip elgtis esant kolizijai. Pavyzdžiui, atsitikus duomenų kolizijai galima tikrinti, ar sekantis masyvo elementas nėra užpildytas. Jei užpildytas – tikrinti sekantį ir t.t., kol bus rasta laisva vieta reikšmei įrašyti, tačiau atsiranda papildomų problemų, kai tas reikšmes reikia šalinti. Kitas mechanizmas – saugoti reikšmes šalia viena kitos, kaip tą patį lentelės elementą, o prireikus nuskaityti reikšmę – tikrinti visas elemente esančias reikšmes. Problema – reikšmių atskyrimas, ypač jei lentelėje saugomos sekos, sudarytos iš įvairių simbolių. Nepaisant paminėtų problemų, kiekviena kolizija reikalauja papildomų resursų mechanizmui iškviesti ir įvykdyti, todėl norimas

lentelės elementas ieškomas ilgiau. Laiko pokytis nebus labai pastebimas, jei lentelė yra nedidelė, tačiau jeigu HASH lentelė yra failų sistema (pvz.: reiserFS) ar duomenų bazė su daugeliu sesijų, uždelsimas gali būti juntamas ir sukelti nepatogumų. Taigi, žodyno testo metu rastas kolizijų skaičius yra kone svarbiausias HASH funkcijos vertinimo kriterijus.

Žodyno testui naudojamas žodynas – tai grynojo teksto byla, kurioje kiekvienoje eilutėje surašytos unikalios frazės. Žodyną galima nesunkiai pasidaryti UNIX operacinėse sistemose naudojant standartinius įrankius:

- susikurti naują grynojo teksto bylą;
- nukopijuoti į ją kiek įmanoma daugiau įvairių tekstų (iš knygų, straipsnių, naujienų portalų, kt.);
- išsaugoti bylą žinomu pavadinimu (pvz.: „kkk“);
- terminale paleisti komandą:
    - cat kkk |tr ' ' '\n' |sort|uniq >DICT_FILE

Žodynas turi būti įkeltos į programos paleidžiamojo failo katalogą ir pavadintas „DICT_FILE“. Koduotė ir eilučių pabaigos simbolis nėra svarbu.

Programa žodyno testą atliks nevienodai, priklausomai nuo to, kuri funkcija bus paleista: *dict* ar *dict2*. Pirmoji funkcija (iškviečiama meniu pasirinkus „44“) yra skirta itin dideliems žodynams, atliekant testą silpnesniuose kompiuteriuose. *dict* atidaro žodyno failą, nuskaito po vieną jo eilutę ir užkoduoja visais algoritmais paeiliui. Kiekvieno algoritmo šifras įrašomas į programos kataloge esantį pakatalogį „outp/“, kur sukuriamas grynojo teksto dokumentas algoritmo pavadinimu. Ši funkcija neskaičiuoja kolizijų ir nieko neišveda į standartinę išvestį, nes *std::map<>* iškvietimas, t.y. žodyno kėlimas į šią struktūrą, gali sunaudoti visą RAM atmintį ir padaryti operacinę sistemą neveiksnia - ši rizika kyla didėjant žodyno failo dydžui. Funkcijai atlikus skaičiavimus kolizijas galima suskaičiuoti Linux/UNIX terminale atsidarius „outp/“ direktoriją ir paleidus šią komandą:

*for i in *; do echo $i; cat $i|sort|uniq -d|wc -l; echo; done*

Funkcija *dict2* skirta mažesniems žodynams arba kompiuteriams, turintiems daugiau RAM atminties. Ši funkcija atidaro žodyną ir jį nuskaito tiek kartų, kiek yra elementų *ALGORITHMS[]* masyve. Kiekvieno nuskaitymo metu kiekviena failo eilutė yra užšifruojama vienu algoritmu ir įkeliama į *std::map<string, int>* struktūrą, kurios atitinkamo *int* nario reikšmė padidinama vienetu. Kitaip tariant, užšifruotas žodynas yra sukeliamas į RAM atmintį ir šalia kiekvieno šifro įrašoma, kelis kartus jis pasikartojo. Priklausomai nuo kintamojo *s2f* reikšmės, šifro kopija gali būti įrašoma

ir į failą, esantį *outp/* kataloge. Perskaičius ir užšifravus visą žodyną iš *std::map<>* struktūros išrenkami ir suskaičiuojami įrašai, kurie pasikartojo daugiau nei vieną kartą – įvyko kolizijos – ir į standartinę išvestį išvedamas tokių įrašų skaičius. Taip suskaičiuojamos kiekvieno algoritmo kolizijos.

5. Vartotojo instrukcija

Norint sėkmingai naudotis programa pirmiausia reikia patikrinti, ar išlaikyta programos failų ir katalogų išdėstymo struktūra. Nesukompiliuotos programos katalogo struktūra turi atrodyti taip:

[...] --- <main.h>
[...] --- <main.cpp>
[...] --- <DICT_FILE>
[...] --- [outp]
[...] --- [algorithms] --- <CityHash.h>
[...] --- [algorithms] --- <CityHash.cpp>
[...] --- [algorithms] --- <lookup3.h>
[...] --- [algorithms] --- <lookup3.cpp>
[...] --- [algorithms] --- <MurMur3.h>
[...] --- [algorithms] --- <MurMur3.cpp>
[...] --- [algorithms] --- <SuperFastHash.h>
[...] --- [algorithms] --- <SuperFastHash.cpp>
[...] --- [algorithms] --- <<naujasAlgoritmas.h>>
[...] --- [algorithms] --- <<naujasAlgoritmas.cpp>>

* [...] – programos katalogas; <abc> - failas; [abc] – katalogas; <<abc>> - galimas papildomas failas.

Sukompiliavus programą, jos paleidžiamasis failas turi būti tame pačiame kataloge, kaip ir „DICT_FILE“ žodynas bei „outp“ katalogas. Programa neturi grafinės sąsajos, todėl ją leisti reikia iš komandų eilutės (*cmd.exe* – Windows OS; terminalo – UNIX/Linux OS). Paleidus programą į standartinę išvestį bus išvestas pagrindinis meniu su 5 pasirinkimo variantais:

1. Nustatymai
2(22). Parodyti HASH'us
3(33). Greičio testas
4(44). Žodyno testas (kolizijos)
0. Išeiti

Pasirinkus pirmą pasirinktį į standartinę išvestį bus išvestas antras meniu, leidžiantis pasirinkti, kurį nustatymą norima redaguoti, taip pat atvaizduojantis esamas keistinų kintamųjų reikšmes:

1.	*Ciklų skaičius:*	*999999*
2.	*seed:*	*42*
3.	*HASHuojama eilutė:*	*Hello world!*
4.	*Saugoti hashus į failą:*	*0*

Visuose programos meniu pasirinktis „0“ yra traktuojama kaip „Grįžti“ (nustatymų meniu) arba „Išeiti“ (pagrindiniame meniu). Nustatymų meniu pakeitus kurią nors reikšmę pakeitimai pritaikomi iškart.

Dirbant su programa ir atliekant testus reikėtų vengti fone leisti programas, reikalaujančias ženklaus kiekio kompiuterio resursų. Įvairios interneto naršyklės, pašto programos,

vaizdo ir garso medžiagos apdorojimo programos reikalauja didelių kiekių RAM atminties, o jei jos atlieka kokį nors darbą, pvz.: vyksta projekto išsaugojimas, tai pareikalaujama ir didelė dalis procesoriaus laiko. Geriausia šiuos testus leisti vos paleidus operacinę sistemą, nepaleidus jokių papildomų taikomųjų programų. Tokiu atveju testų rezultatai bus tikslesni ir greitesni.

Jei pagrindiniame meniu pasirenkamas 2, 22, 3 ar 33 punktai, programa vykdys šifravimo algoritmus naudodama numatytąsias kintamųjų *test*, *seed* ir *loopCount* reikšmes. Jei norima užšifruoti kitą simbolių eilutę, pakeisti greičio matavimo testo trukmę ar pakeisti *seed*, šių kintamųjų reikšmes galima pakeisti nustatymų meniu.

Programa paleidimo metu netikrina, ar jei buvo pateikti kokie nors parametrai. Visas darbas su programa atliekamas ją paleidus o jos parametrus keisti galima tik programai veikiant. Kita galimybė pakeisti parametrus – redaguoti programos kodą. Tokiu atveju programą reikės perkompiliuoti. Jei redaguojant programos kodą buvo pakeistos HASH funkcijos – pridėtos naujos ar pašalintos senos – būtina atitinkamai pakeisti ir masyvų *ALGORITHMS[]* ir **str_Fpt[]* elementus, kitaip pakeitimai bus neįgyvendinti arba kritiškai paveiks programos veikimą. Pridedant naujus HASH algoritmus, pačios algoritmų funkcijos turi būti *algorithms/* kataloge. Tinkamai aprašius funkcijas ir sukūrus jų antraštinius failus, šie turi būti įtraukti į main.h failą. Ten pat turi būti ir algoritmų funkcijų apvalkalai, priimantys tris parametrus: koduojamą eilutę *string*, jos ilgį *int* ir *seed* reikšmę – *uint32_t*, ir grąžinantys šifrą *string* formatu. Apvalkale galima aprašyti visus reikalingus veiksmus duomenims paruošti, pateikti ir apdoroti, kad atitiktu minėtus apvalkalo kriterijus.

6. Išvados ir pasiūlymai

Atlikus testus su turimomis HASH funkcijomis ir įvertinus gautus rezultatus galima padaryti išvadas apie jų patikimumą, kokyiškumą, spartą ir sritis, kur jas galima taikyti. Nei vienos HASH funkcijos neverta taikyti paprastiems masyvams sudarinėti, nes tai būtų nereikalingas resursų švaistymas. Nuskaityti rakto – šifruotinos simbolių sekos – reikšmė, ją pateikti šifruojančiai funkcijai, užšifruoti, šifrą pateikti masyvui kaip indeksą ir iškviesti mechanizmus įvykus galimai kolizijai reikalauja nemažai resursų, o ypač, jei tokių veiksmų per sekundę įvykdoma daug. Kuriant nedidelius masyvus, jei įmanoma, reikėtų naudoti paprastus *int* ar *double* tipo indeksus, nes toks mechanizmas veikia žymiai sparčiau.

Įvairios HASH funkcijos generuoja įvairaus ilgio šifrus. Paprastai šifro ilgis nepriklauso nuo šifruojamos simbolių sekos ilgio. Mažoms HASH lentelėms sudaryti rekomenduotina rinktis algoritmus, generuojančius trumpesnius šifrus, nes paprastai jie greitesni. Pavyzdžiui SuperFastHash algoritmas yra vienas greitesnių, tačiau žodyno testo metu stebimas didelis kolizijų skaičius, vadinasi tokį algoritmą galime naudoti tik tokiose HASH lentelėse, kuriose bus nedaug įrašų. SuperFastHash žodyno testo metu užšifravus 162292 žodžius buvo aptikta 51 kolizija, vadinasi tikėtina, kad sudarant HASH lentelę atsitiks vidutiniškai bent viena kolizija kas 3183 įrašus. Tuo tarpu tik nežymiai greičiu nuo SuperFastHash besiskiriantys algoritmai Murmur3_x86_32, lookup3 ir kiek lėtesnis CityHash32 kolizijų atžvilgiu davė dešimtis kartų geresnius rezultatus: tame pačiame žodyne buvo aptiktos atitinkamai 2, 1 ir 1 kolizija. Šie algoritmai sugeneruoja 10 simbolių ilgio šifrus, tad žodyno testo rezultatai gana neblogi.

Tuo tarpu CityHash64 greičiu labai nedaug nusileidžia minėtiems _x86_ algoritmams ir sugeneruoja dvigubai ilgesnius – 20-ties simbolių – šifrus. Žodyno testo metu nebuvo rasta nė vienos kolizijos, tad galima daryti išvadą, kad didesnėse duomenų bazėse ar žodynuose šis algoritmas yra taikytinas. Itin didelėse duomenų lentelėse, priklausomai nuo jų dydžio, galima taikyti Murmur3_x32_128 ir Murmur3_x64_128 algoritmus. Jie eilutę užšifruoja ~5 kartus ilgiau, tačiau šifro ilgis siekia atitinkamai 40 ir 80 simbolių, taigi teoriškai galima naudoti sudarinėjant duomenų bazes, kurias sudaro iki 10^{80} unikalių elementų. Tiesa, tokiose duomenų bazėse jau reikėtų naudoti krūvio paskirstymo sistemas (angl. *load balancer*), nes šie šifravimo algoritmai veikia sąlyginai lėtai, o paprastai į dideles duomenų bazes daromas didelis kiekis užklausų.

7. Anotacija lietuvių ir anglų kalbomis

Vilniaus kolegija
Elektronikos ir informatikos fakultetas
Programinės įrangos katedra

Vilnius College of Higher Education
Faculty of Electronics and Informatics
Department of Software

Specialybės kodas 653I10001
Data 2014-05-01

Kursinio darbo tema: **Hash algoritmai**
Studentas: **Darius Juodokas**
Darbo vadovas: **Rimvydas Motiejūnas**

The Theme of the Project: **Hashing algorithms**
Student: **Darius Juodokas**
Adviser: **Rimvydas Motiejūnas**

Lietuvių kalba

Darbo tikslai ir uždaviniai:

► Parašyti programą, testuojančią HASH funkcijas
► Įvertinti atliktų testų rezultatus
► Įvertinti HASH funkcijų pritaikomumą duomenų struktūose

Taikyti metodai:

► Naudojamos 7 HASH funkcijos
► Atlikti greičio ir žodyno testai

Darbo rezultatai:

► Gauti skirtingi funkcijų greičiai
► Gauti skirtingi kolizijų dažniai

Išvados:

► HASH šifrų ilgiai skiriasi
► Šifravimo greitis priklauso nuo šifro ilgio
► Šifravimo algoritmas turi būti parinktas atsižvelgiant į duomenų struktūros dydį

Foreign language (English)

Objectives:

► To develop a program able to test different hashing functions
► Evaluate results of the tests performed
► Summarise on which hashing functions would be the best choice for various data structures

Applied methods:

► 7 hashing algorithms were used
► Speed and dictionary tests were performed

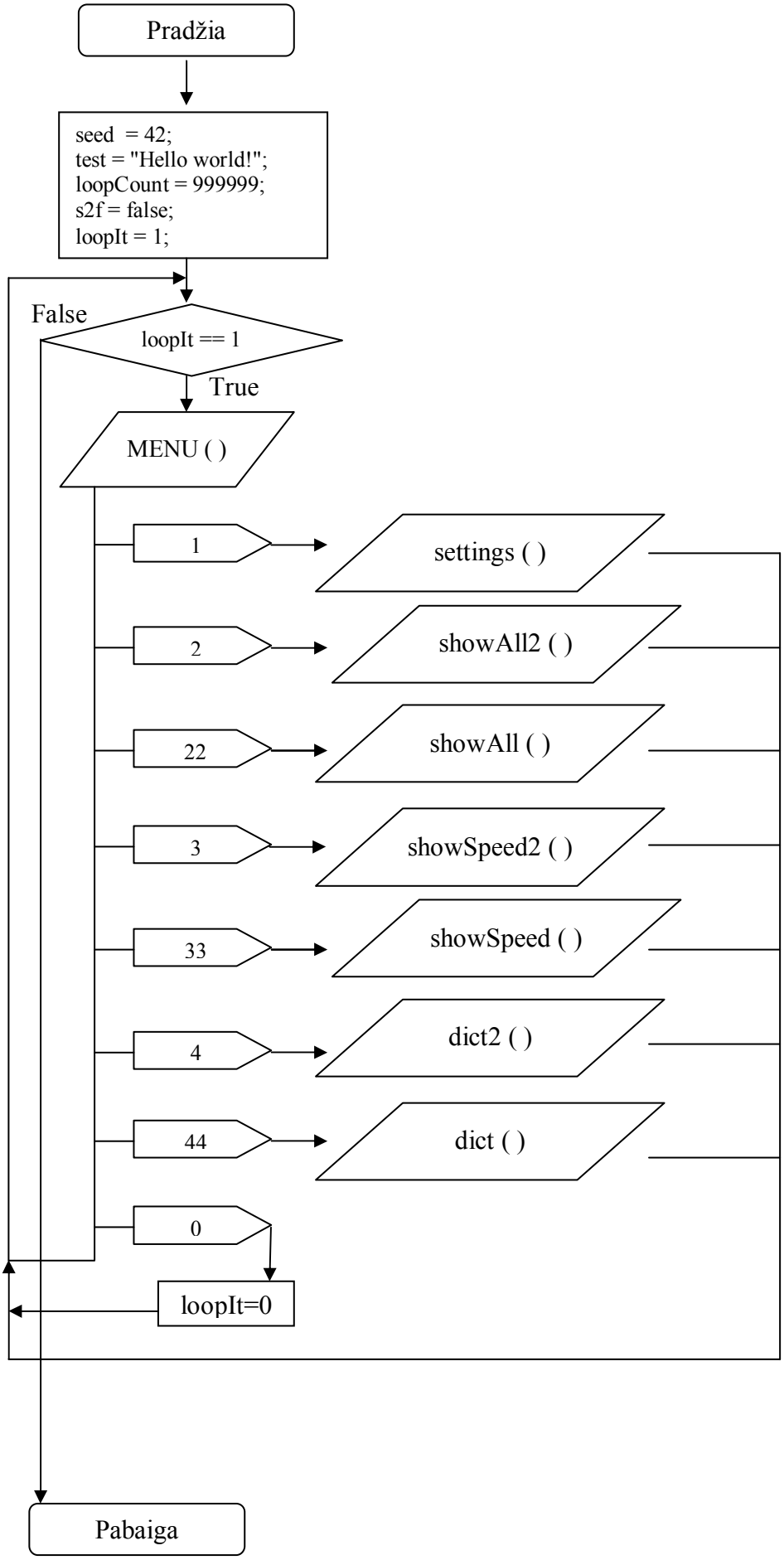
Results:

► Different hashing speeds observed
► Different collision rates observed

Conclusions:

► Hashes differ in length
► Hashing speed depends on hash length
► Hashing algorithm must be chosed depending on the size of data structure it‘s going to be used for.

8. Programos algoritmo blokinė schema



9. Literatūros sąrašas

- https://github.com/hrydgard/native/tree/master/ext/cityhash
- https://github.com/PeterScott/murmur3
- https://code.google.com/p/smhasher/source/browse/trunk/lookup3.cpp
- https://github.com/chrisvana/smhasher_copy/blob/master/SuperFastHash.cpp
- http://www.cplusplus.com/forum/general/49980/
- http://www.cprogramming.com/tutorial/function-pointers.html
- http://www.cprogramming.com/tutorial/computersciencetheory/hash-table.html
- http://www.cs.uregina.ca/Links/class-info/210/Hash/
- http://stackoverflow.com/questions/2179965/whats-the-point-of-a-hash-table
- http://stackoverflow.com/questions/282712/the-fundamentals-of-hash-tables
- http://en.wikipedia.org/wiki/Hash_function