SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Zlatko Pračić

BLOOM FILTER

SEMINARSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zlatko Pračić

Matični broj: 00161450975

Studij: Razlikovna godina diplomskog studija - IPS

BLOOM FILTER

SEMINARSKI RAD

Mentor :

Prof. dr. sc. Alen Lovrenčić

Varaždin, lipanj 2023.

Sadržaj

1. Uvod

1.1. Predmet i cilj rada

Predmet ovog seminarskog rada je apstraktni tip podataka (ATP) Bloom filter, koji je probabilistička struktura podataka korištena za učinkovitu provjeru prisustva elementa u skupu. Bloom filter nudi kompaktno rješenje na upit je li određeni element prisutan u skupu, bez potrebe za pohranjivanjem stvarnih elemenata skupa. To je ponekad vrlo korisna tehnika u situacijama kada su brzina i učinkovitost presudni, a ne postoji zahtjev za apsolutnom točnošću rezultata.

Cilj ovog rada je prikazati Bloom filter kao podatkovnu strukturu te istaći njegove karakteristike, prednosti i nedostatke. Kroz objašnjenje algoritma i matematičkih principa na kojima se zasniva, cilj je predočiti način na koji Bloom filter radi, kako se koristi i koje vrste problema može efikasno riješiti. Osim toga, kroz rad će se prikazati jedan od mogućih scenarija primjene Bloom filtra.

1.2. Izvori podataka i metode prikupljanja

Tijekom izrade seminarskog rada korišteni su različiti izvori informacija i to: relevantne knjige i znanstveni radovi koji opisuju općenito strukture podataka i algoritme te apstraktni tip podataka Bloom filter.

1.3. Sadržaj i struktura rada

Rad se sastoji od pet poglavlja od kojih je prvi uvod, a peti zaključak. Drugo poglavlje općenito opisuje Bloom filter, čemu isti služi i mogući način uporabe. U istom poglavlju je opisan način rada te vrste apstraktnog tipa podataka (ATP) i na koji način se određuju veličine vezane uz sam filter (veličina polja, broj unosa, broj funkcija sažimanja te kolika je mogućnost lažnih pozitivnih vrijednosti pri provjeri elemenata u polju). U trećem poglavlju se prikazuje kod sačinjen u C++ programskom jeziku pri čemu se pozornost daje načinu implementacije ATP-a i programu koji koristi ATP Bloom filter. U četvrtom se izračunala vremenska i amortizirana složenost Bloom filtra, a temeljem koda koji je dio rada.

2. Opis ATP-a Bloom filter

Bloom filter je vrsta podatkovne strukture nazvane po svom tvorcu, Burtonu Howardu Bloomu. Sličan je tablici sažetaka (eng. hash table), osim što umjesto pohranjivanja podataka, samo odgovara na upit je li podatak u skupu. Prednost Bloomovog filtra je što koristi manje memorije nego tablica sažetaka, no postoji kompromis između točnosti i iskorištenja memorije. Bloom filtri mogu proizvesti lažno pozitivne rezultate, a brisanje vrijednosti iz Bloom filtra nije izvedivo. Kako bi se dobio odnos između lažnih pozitivnih rezultata i prostora koji se zauzima postoji formula koja se može koristiti za izračunavanje količine memorije potrebne za održavanje stope lažno pozitivnih rezultata ispod određenog praga. Ista će se moći vidjeti dalje u radu [1].

La Rocca u svojoj knjizi opisuje implementaciju Bloom filtera, koji se sastoji od polja m veličine i zbirke k funkcija sažimanja (eng. *hash functions*). Svaki element polja početno je postavljen na 0, a svaka funkcija sažimanja vraća indeks između 0 i m-1. Kada se novi element doda filtru, indeksi niza se izračunavaju korištenjem funkcija sažimanja (ovisno o broju sažimanja koji se koristi) i postavljaju na 1. Da bi se pretražila stavka, izračunaju se sažimanja i ako su svi bitovi na tim lokacijama postavljeni na 1, smatra se da je unos u filtru (mora se paziti na lažne pozitivne rezultate) [1].

2.1. Namjena ATP Bloom filter

Bloom filtar je sažeto probabilističko predstavljanje skupa podataka i koristi se za ispitivanje nalazi li se određena stavka u skupu ili ne. Osnovni Bloom filtar sastoji se od polja bitova i određenog broja funkcija sažimanja. Funkcije sažimanja koriste se za mapiranje svake stavke u jednoj od pozicija polja. Nakon sažimanja ulaznog elementa, vrijednost koja se dobije kao sažetak služi kao broj elementa polja koji se iz 0 mijenja u 1 (iz „false“ u „true“). Ta akcija se obavlja na isti način onoliko puta koliko se puta sažima sadržaj pa na primjer ako se koriste tri funkcije sažimanja onda se ista radnja provodi tri puta. S druge strane, postojanje se provjerava tako što se nakon sažimanja ulazne vrijednosti provjerava jesu li polja niza postavljena na nula ili jedan. Bloom filtar ima sljedeće prednosti: ima mali memorijski otisak, ima brzu i dosljednu brzinu upita i ažuriranja te ne daje lažno negativne rezultate i ima nisku stopu lažno pozitivnih rezultata, ali ta stopa se da konfigurirati na prihvatljivu razinu. Zbog ovih prednosti, Bloom filtar i njegove varijante naširoko se koriste u širokom rasponu područja pa su neka od područja gdje se Bloom filter koristi: sustavi u stvarnom vremenu, računalne arhitekture, neuronske mreže, IP pretraživanja, P2P mreže podatkovnih centara, računalstvo u oblaku [2].

Međutim, Bloom filtar ima dva glavna nedostatka: unesene podatke nije moguće izbrisati niti je moguće listu proširiti. Konkretno, nakon što je stavka umetnuta u standardni Bloom filtar, stavka se ne može izravno izbrisati. Pored toga, nakon što je skup umetnut u Bloom filtar, nemoguće je konstruirati veći Bloom filtar koji predstavlja isti skup bez dodatnih informacija. Bez obzira na navedene nedostatke, Bloom filter je najbolji izbor u mnogim primjenama iako te aplikacije neizbježno pate od gore navedenih nedostataka pa se navedeni ATP koristi na

primjer za:

Crne liste - Bloom filter se koristi za pohranjivanje crne liste za sprječavanje prijetnji kao što su DDoS napadi i razne verzije te vrste napada. Nedostatak ovog pristupa može biti mogućnost postavljanja upita za IP adresu koja je legalna te rezultat može biti lažno pozitivan i IP adresa će se smatrati zlonamjernom. Kako bi se riješio taj problem, postavlja se bijela lista za pohranu prijateljskih adresa.

Traženje MAC adrese - Svaki preklopnik (eng. *switch*) ima tablicu MAC adresa. Tablica ima velik broj unosa i svaki se unos može smatrati parom ključ-vrijednost. Ključ je odredišna MAC adresa, a vrijednost je odlazni port. Jedan Bloom filtar izgrađen je za sve MAC adrese s istim odlaznim portom [2].

2.2. Problemski zadatak

U svojoj knjizi La Rocca opisuje hipotetsku situaciju u kojoj je zaposleniku povjerena izgradnja nove značajke pri redizajnu usluge e-pošte. Pri tome jedna od novijih opcija bi bila da se prije dodavanja nove email adrese provjeri je li ona već na popisu postojećih adresa u adresaru. Resursi projekta su ograničeni, a zaposlenik ne može pozivati bazu podataka svaki put kada treba usporediti adresu e-pošte s popisom kontakata (želi se vidjeti nalazi li se adresa u adresaru). U knjizi autor preporučuje tehniku u kojoj se popis kontakata dobiva asinkrono i sprema u prostor za pohranu sesije, omogućujući zaposleniku da provjeri lokalnu kopiju podataka svaki put kada traži postojeće kontakte. Nakon što je lista pohranjena na lokalnom računalu, potrebno je provjeriti nalazi li se adresa u listi ili ne. Time se dolazi do problema rada s rječnikom, što je čest računalni problem traženja određenog unosa na popisu [1].

Kako bi se traženo provelo na efikasan način postoji potreba za strukturom podataka koja može učinkovito upravljati operacijama poput preuzimanja, pohranjivanja i traženja kontakata. Obična polja nisu učinkovita za takve operacije jer im nedostaje metoda za brzo pronalaženje indeksa elementa ili određivanje je li element u polju. Autor predlaže asocijativno polje pošto isti ima izvornu metodu za učinkovit pristup pohranjenim unosima s pretraživanjem po vrijednosti. Redoslijed umetanja u asocijativno polje nije bitan i može pohranjivati parove (ključ, vrijednost). Korištenje apstrakcije rječnika omogućuje fokusiranje na zadatak rješavanja problema bez potrebe za rješavanjem detalja reprezentacije strukture podataka [1].

Nadalje, autor raspravlja o asocijativnom polju, koji su zbirke parova (ključ, vrijednost) gdje se svaki ključ pojavljuje najviše jednom i svaka se vrijednost može dohvatiti izravno preko odgovarajućeg ključa i predlaže korištenje asocijativnog polja za pohranjivanje korisničkog adresara, koji se prima od poslužitelja kada se prijavi. Rječnik se čuva u memoriji i ažurira pozivima za umetanje i uklanjanje. Kada korisnik napiše e-poštu i unese primatelja, rječnik se provjerava je li kontakt u adresaru. Ako nije, pojavit će se skočni prozor s pitanjem želi li korisnik spremiti novi kontakt. Ovaj pristup minimizira HTTP pozive poslužitelju i čita iz baze podataka samo jednom pri pokretanju [1].

2.3. Načini moguće implementacije zadatka

U nastavku autor govori o implementaciji asocijativnih polja u stvarnim sustavima. Ako je domena (skup mogućih ključeva) dovoljno mala, može se definirati ukupni poredak na ključevima i može se koristiti pravo polje, s nedostajućim vrijednostima postavljenim na *null*. Međutim, u većini slučajeva, skup mogućih vrijednosti za ključeve je prevelik da bi se koristilo polje s elementom za svaku moguću vrijednost ključa, jer bi to zahtijevalo previše memorije. Autor je opisao četiri često korištene alternative za prevladavanje ovog problema s memorijom. Nakon opisa svih opcija bit će prikazana i tablica s vrijednostima vremena izvođenja operacija [1]:

Nesortirano polje. Generalno za ovu vrstu polja autor kaže kako se u isto brzo umeće podatak, ali se jako sporo pretražuje i pronalazi. Tiskana knjiga bi mogla biti primjer gdje su riječi navedene redoslijedom kojim su ispisane. Ako je potrebno pronaći određenu riječ, kao što je na primjer "Bloom", mora se pretraživati knjiga riječ po riječ dok se ne pronađu sva pojavljivanja te riječi. Nerazvrstana polja ne zahtijevaju dodatni rad na stvaranju, a dodavanje novog unosa je jednostavno, ali pronalaženje određenih unosa može potrajati dugo [1].

Sortirano polje i binarno pretraživanje. Za ovu vrstu implementacije autor navodi kako je kod iste sporo umetanje podataka i brzo pretraživanje. Kod ove implementacije se koriste indeksi sortiranih popisa za učinkovito pretraživanje. Binarno pretraživanje može koristiti za traženje određene riječi na sortiranom popisu (kao na primjer telefonski imenik). Takva vrsta pretraživanja uključuje početak od otprilike polovice, provjeravanje nalazi li se riječ prije ili poslije tog mjesta. Ono što je specifično za ovu vrstu implementacije je da početno sortiranje i naknadno unošenje podataka (uz zahtjev da se uređenost zadrži) zahtjeva puno resursa [1].

Hash tablica. Hash tablice se koriste za implementaciju asocijativnih polja gdje moguće vrijednosti za pohranjivanje dolaze iz vrlo velikog skupa (na primjer, svi mogući nizovi ili svi cijeli brojevi), ali obično ih se treba pohraniti samo ograničen broj. Ako je to slučaj, tada se koristi funkcija sažimanja za mapiranje skupa mogućih vrijednosti (domena ili izvorni skup) u manji skup od M elemenata (kodomena ili ciljni skup). Uobičajeno se skup vrijednosti u domeni naziva ključevima, a vrijednosti u kodomeni indeksima od 0 do M-1. Budući da je ciljni skup funkcije sažimanja obično manji od izvornog skupa, doći će do sukoba: najmanje dvije vrijednosti bit će preslikane u isti indeks. Hash tablice koriste nekoliko strategija za rješavanje tog problema i to ulančavanje ili otvoreno adresiranje [1].

Binarno stablo. Sljedeća vrsta implementacije je binarno stablo. Ista koristi svojstva ukupnog uređenja u cilju osiguranja da se položaj ključa u stablu može odrediti gledanjem jedne staze od korijena do lista. Sve operacije na binarnom stablu zahtijevaju vrijeme proporcionalno visini stabla, što je $O(ln(n))$ za uravnoteženo binarno stablo. Iako je izvedba binarnog stabla na osnovnim operacijama sporija od hash tablica, ono ima prednost u pronalaženju prethodnika i sljedećeg ključa, te pronalaženju minimuma i maksimuma, pri čemu se sve te operacije izvode u $O(\ln(n))$ asimptotskom vremenu. Dodatno, binarno stablo može vratiti sve pohranjene ključeve, sortirane po ključu, u linearnom vremenu, dok hash tablice zahtijevaju sortiranje nakon dohvaćanja skupa ključeva.

Tablica 1 prikazuje vremena izvođenja glavnih operacija na navedenim implementacijama rječnika [1].

Tablica 1: Složenost operacija na rječnicima za različite implementacije

Operacija	Nesortirano polje	Sortirano polje	Binarno stablo	Hash tablica
Kreiranje strukture	$O(1)$	$O(n * \log(n))$	$O(n * \log(n))$	$O(n)$
Traženje unosa	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(n/M)$ [a]
Unos novog podatka	$O(1)$ [a]	$O(n)$	$O(\log(n))$	$O(n/M)$ [a]
Brisanje unosa	$O(1)$	$O(n)$	$O(\log(n))$	$O(n/M)$ [a]
Sortiranje	$O(n * \log(n))$	$O(n)$	$O(n)$	$O(M + n * log(n))$
Min/Max	$O(n)$	$O(1)$	$O(1)$ [b]	$O(M + n)$
Sljedeći/Prethodni	$O(n)$	$O(1)$	$O(\log(n))$	$O(M + n)$

[a] Amortizirana vrijednost

[b] Odvojenim pohranjivanjem max/min i amortiziranjem vremena za njihovu zamjenu pri umetanju/brisanju

Bloom filter. Za Bloom filter autor kaže kako je brz kao hash tablica, ali da pri tome štedi na memoriji računala. Postoje značajne razlike između hash tablica i Bloom filtara i one su:

- Osnovni Bloom filtri ne pohranjuju podatke, oni samo odgovaraju na pitanje je li podatak u skupu?
- Zahtjevi Bloom filtra za memorijom su manji u usporedbi s hash tablicama; to je glavni razlog njihove upotrebe.
- Iako je negativan odgovor 100% točan, može biti lažno pozitivnih odgovora.
- Nije moguće izbrisati vrijednost iz Bloomovog filtra [1].

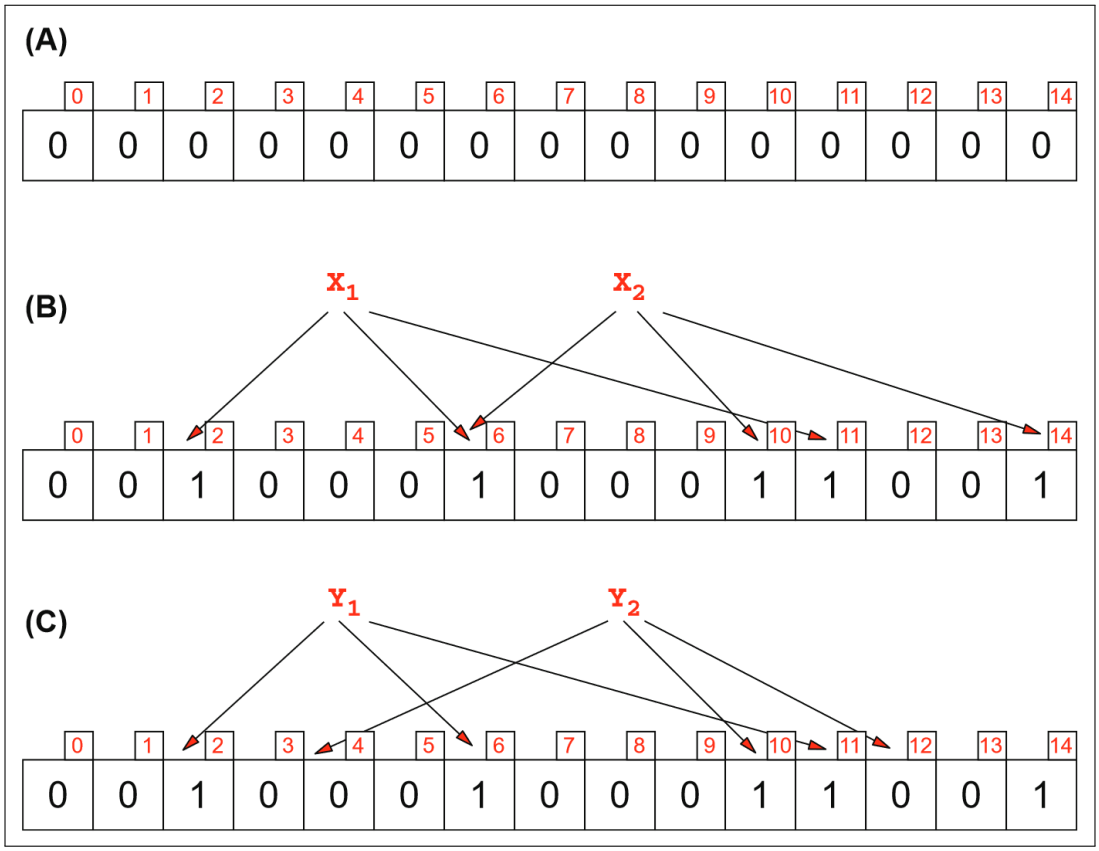
Postoji kompromis između točnosti Bloomovog filtra i memorije koju koristi. Što je manje memorije, vraća se više lažno pozitivnih rezultata. Međutim, postoji točna formula koja, s obzirom na broj vrijednosti koje se trebaju pohraniti, može izračunati količinu memorije potrebnu za održavanje stope lažno pozitivnih rezultata unutar određenog praga [1].

2.4. Kako radi Bloom filter

Bloom filter se sastoji se od dva elementa: polje od m elemenata i skup k funkcija sažimanja. Polje od m elemenata se inicijalno postavlja na 0. Svaka funkcija sažimanja daje indeks između 0 i $m - 1$. Korisniku je važna informacija kako ne postoji odnos 1-na-1 između elemenata polja i ključeva koji se dodaju Bloom filtru. Umjesto toga, koristi se k bitova (i stoga

k elemenata polja) za pohranu svakog unosa za Bloom filtar. U ovom slučaju k je ovdje obično mnogo manji od m. Potrebno je imati na umu da je k konstanta koja se bira pri stvaranju strukture podataka, tako da svaki unos koji se dodaje pohranjen koristeći istu količinu memorije, točno k bitova [1].

To znači da se mogu dodati proizvoljno dugačka polja filtru koristeći stalnu količinu memorije. Kada se umetne novi ključ elementa u filtar, izračunava se k indeksa za polje, danih vrijednostima h_0(ključ) do $h_{(k-1)}$(ključ) i ti se bitovi postavljaju na 1. Kada se traži unos, također treba izračunati k sažimanja za njega kao što je opisano za umetanje, ali ovaj put se provjerava k bitova na indeksima koje vraćaju funkcije sažimanja i vraća se $true$ akko su svi bitovi na tim pozicijama postavljeni na 1. Slika 1 prikazuje obje operacije [1].



Slika 1: Prikaz rada Bloom filtra [1]

U nastavku je objašnjenje rada Bloom filtra prikazanog na Slici 1.

A) Na početku je filtar niz nula.

B) Pohrana stavki x_i počinje tako što se svaki element sažima k puta (u primjeru se vidi kako je $k = 3$), pri čemu svakim sažimanjem indeksu pridružuje vrijednost 1. Kako je filterom određeno da će biti 3 funkcije sažimanja vrijednost x_1 mijenja vrijednost u polju na tri mjesta. Isto tako se može primijetiti da x_1 i x_2 imaju djelomično preklapanje (oba pokazuju na šesti element).

C) Kako bi se provjerilo je li element y_i u skupu, isti se sažima k puta dobivajući pri tom isto

toliko indeksa. Nakon toga se čitaju odgovarajući bitovi i akko su svi postavljeni na 1, vraćamo *"true"*. Na Slici 1 se vidi kako je element y_1 postavljen (međutim, ne može se isključiti mogućnost da filter vraća lažno pozitivan rezultat), dok element y_2 nije u skupu jer jedan od indeksa generiranih njegovim sažimanjem ima 0. Za taj element se može sa 100% sigurnošću reći kako nije u polju [1].

U idealnom slučaju, filtru je potrebno k različitih nezavisnih funkcija sažimanja kako bi se osiguralo da se indeksi ne dupliciraju za istu vrijednost. Iako je teško dizajnirati velik broj nezavisnih funkcija sažimanja, postoji nekoliko rješenja koje pružaju dobre aproksimacije. Jedno rješenje je korištenje parametarske funkcije sažimanja $H(i)$, koja generira k funkcija sažimanja pozivanjem generatora H na različite vrijednosti. Drugo rješenje je korištenje jedne funkcije sažimanja H, ali inicijalizacija liste L s k slučajnih i jedinstvenih vrijednosti te stvaranje k vrijednosti sažetka dodavanjem ili povezivanjem $L[i]$ s ključem. Treće rješenje je korištenje dvostrukog ili trostrukog sažimanja. U implementaciji se često koristi dvostruko sažimanje s dvije nezavisne funkcije sažimanja: *Murmur hashing* i *Fowler-Noll-Vo hashing* [1].

2.5. Provjera i unos podatka uz pomoć Bloom filtra

U nastavku će se prikazati na koji način se provjerava postojanje neke vrijednosti uz pomoć Bloom filtra te način na koji se neka vrijednost unosi u isti.

2.5.1. Provjera vrijednosti

Kako se radi o hipotetskom zadatku provjere postojanja nečije e-mail adrese u imeniku, prvo će se prikazati provjera postojanja. Ta operacija se može opisati kroz sljedeće korake:

A) Počinje se s e-poštom koja se želi provjeriti nalazi li se u imeniku i to `duffy@acme.com`.

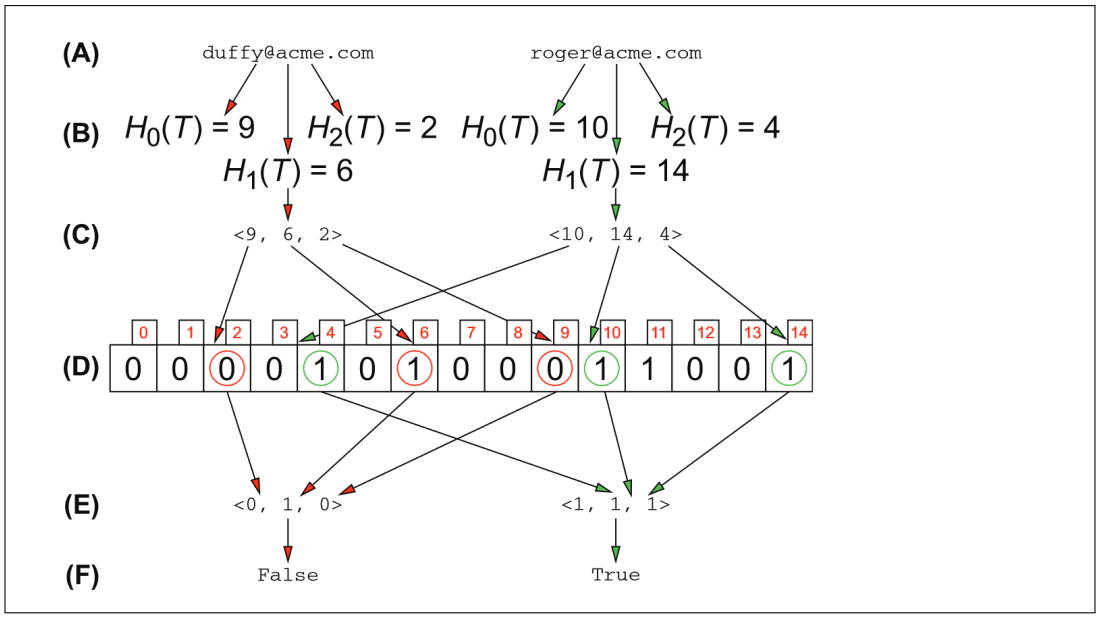
B) Ključ (e-pošta) se obrađuje kroz skup funkcija sažimanja. U konkretnom primjeru sa Slike 2 koriste se tri funkcije sažimanja i to H_0, H_1 i H_2.

C) Svaka hash funkcija proizvodi indeks za polje s binarnim vrijednostima. Konkretno u ovom slučaju se radi o indeksima $\langle 9, 6, 2 \rangle$.

D) Pristupa se elementima polja na tim indeksima.

E) Na indeksima 2 i 9 vrijednost je 0 (*false*), a na indeksu 6 je vrijednost 1 (*true*).

F) Kako svi elementi polja na traženim indeksima ne pokazuju vrijednost 1 (*true*) to može značiti samo jedno, a to je da tražena e-mail adresa `duffy@acme.com` nije pohranjen u Bloom filtru i vraća se vrijednost (*false*). Isti tijek koraka će se provesti i za e-mail adresu `roger@acme.com`. U tom slučaju su svi elementi polja na traženim indeksima postavljeni na 1 (*true*) što znači da je e-mail adresa `roger@acme.com` možda pohranjena u Bloom filteru i to s određenim stupnjem pouzdanosti [1].



Slika 2: Provjera prisutnosti podatka uz pomoć Bloom filtra [1]

2.5.2. Unos vrijednosti

Na Slici 3 se može vidjeti grafički prikaz rada Bloom filtra to jest konkretne operacije unošenja podatka u Bloom filter.

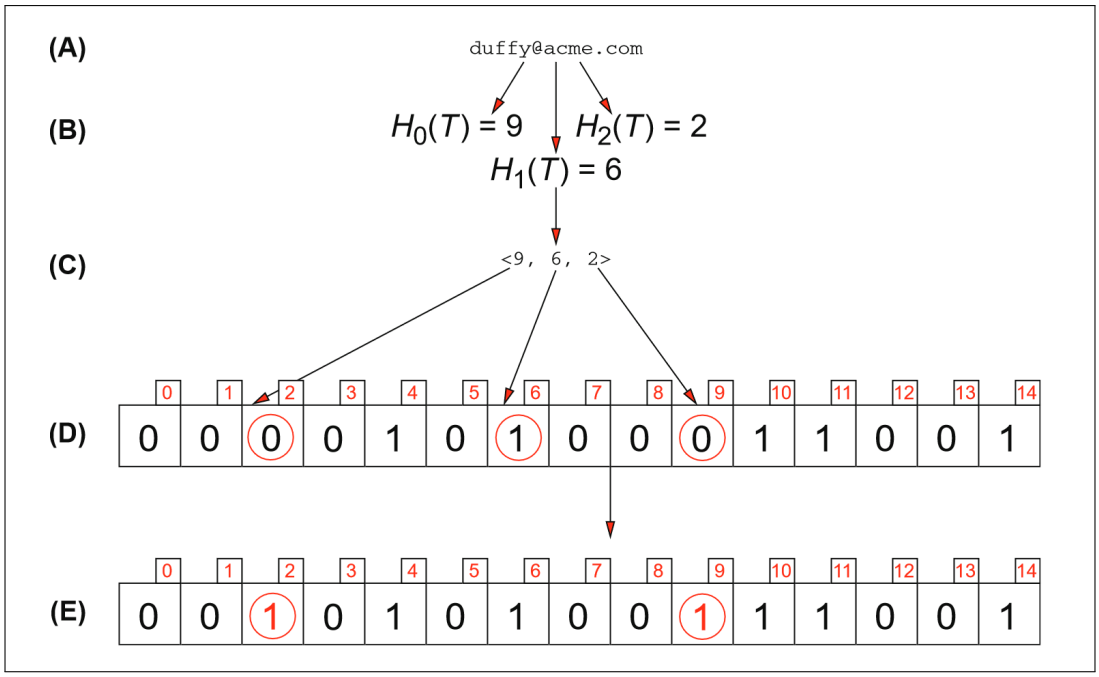
A) Počinje se s e-poštom koju se želi pohraniti, konkretno `duffy@acme.com`.

B) Ključ (e-pošta) obrađuje se kroz skup funkcija sažimanja: broj funkcija je postavljen $k = 3$, pa će biti tri različite funkcije sažimanja, H_0, H_1 i H_2.

C) Svaka funkcija sažimanja proizvodi indeks za binarno polje. U ovom slučaju su tri indeksa $\langle 9, 6, 2 \rangle$.

D) Pristupa se elementima binarnog polja kod tih indeksa. Na indeksima 2 i 9 je 0, a na indeksu 6 je 1.

E) Mijenjaju se samo bitovi koji su bili postavljeni na 0 (u ovom primjeru, one na indeksima 2 i 9). Nakon toga su svi dijelovi kojemu su sažetci `duffy@acme.com` postavljeni na 1, tako da će svako buduće pretraživanje vratiti *"true"* [1].



Slika 3: Unos vrijednosti uz pomoć Bloom filtra [1]

2.6. Izračun lažno pozitivne vjerojatnost Bloom filtra

Lažno pozitivna stopa vjerojatnosti Bloomovog filtra može se izvesti na temelju pretpostavke kako je svaka pozicija niza odabrana s jednakom vjerojatnošću funkcijama sažimanja. Neka m označava broj bitova u Bloom filteru. Prilikom umetanja elementa u filtar, vjerojatnost da određeni bit nije postavljen na jedinicu funkcijom sažimanja dana je s formulom [3]:

$$1 - \frac{1}{m} \tag{1}$$

Uzimajući u obzir k funkcija sažimanja, vjerojatnost da bilo koja od njih nije postavila određeni bit na jedan je:

$$\left(1 - \frac{1}{m}\right)^k \tag{2}$$

Nakon umetanja n elemenata u filtar, vjerojatnost da je dati bit još uvijek nula je:

$$\left(1 - \frac{1}{m}\right)^{kn} \tag{3}$$

Posljedično, vjerojatnost da je bit jedan je:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \tag{4}$$

Za test članstva elementa, ako su svi k položaji niza izračunati funkcijama sažimanja postavljeni na jedan, Bloom filtar tvrdi da element pripada skupu. Vjerojatnost da se to dogodi kada element nije dio skupa aproksimira se kao:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \tag{5}$$

Autori rada napominju kako je $e^{-kn/m}$ vrlo bliska aproksimacija $\left(1 - \frac{1}{m}\right)^{kn}$. Lažno pozitivna vjerojatnost se smanjuje kako se veličina Bloomovog filtra, m, povećava. Vjerojatnost raste s n kako se dodaje više elemenata. Minimiziranjem se želi smanjiti vjerojatnost lažno pozitivnih rezultata $\left(1 - e^{-kn/m}\right)^k$ s obzirom na k. To se postiže uzimanjem derivacije i izjednačavanjem s nulom što daje optimalnu vrijednost k:

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \tag{6}$$

To rezultira lažno pozitivnom vjerojatnošću od približno:

$$\left(\frac{1}{2}\right)^k \approx 0.6185^{m/n} \tag{7}$$

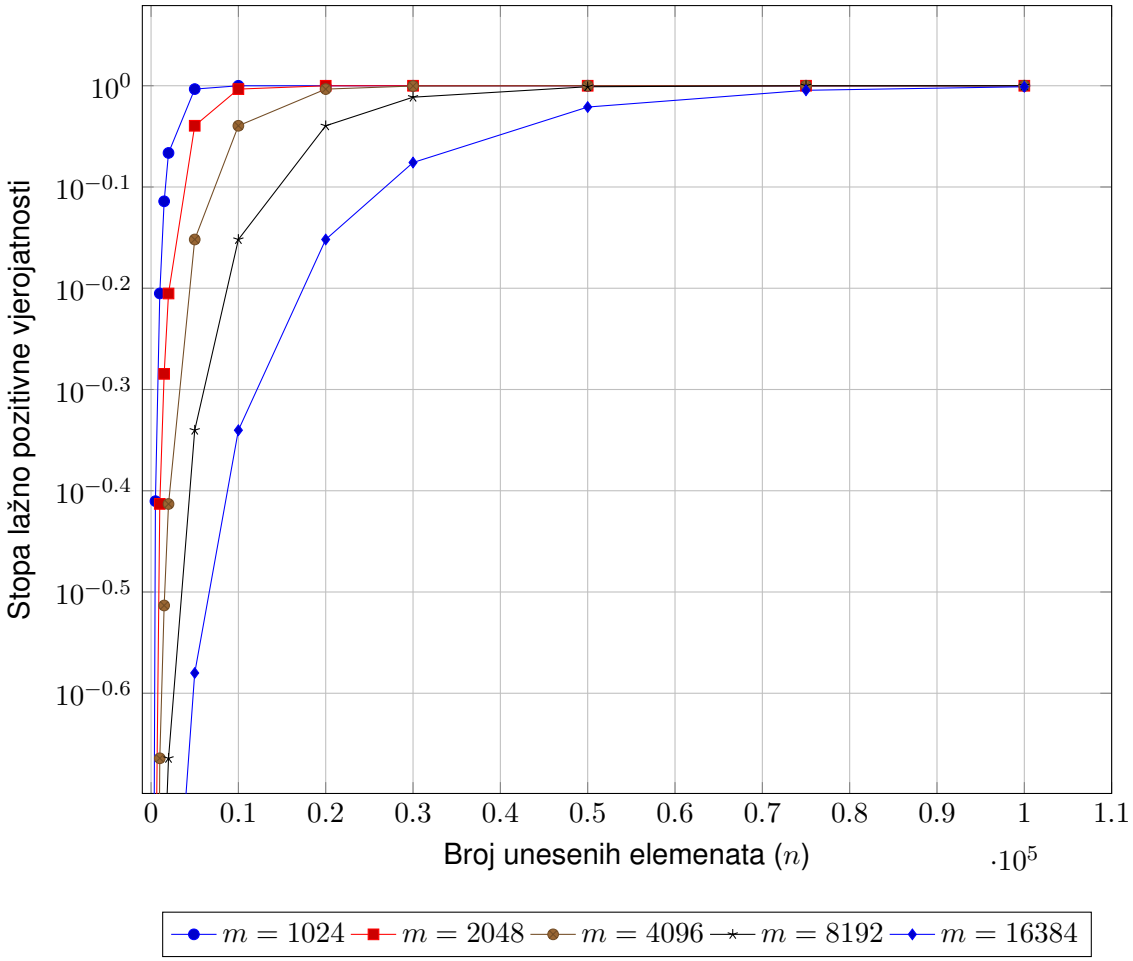
Upotrebom optimalnog broja hash funkcija k_{opt} preko lažno pozitivne vjerojatnosti se može pisati i ograničiti kao:

$$\frac{m}{n} \geqslant \frac{1}{\ln 2} \tag{8}$$

Da bi se održala fiksna lažno pozitivna vjerojatnost, duljina Bloomovog filtra mora rasti linearno s brojem umetnutih elemenata gdje je p željena lažno pozitivna stopa:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{9}$$

Slika 4 prikazuje lažno pozitivnu stopu vjerojatnosti p kao funkcija broja elemenata n u filtru i filtar veličina m. Optimalan broj funkcija sažimanja $k = \frac{m}{n} \ln 2$ je pretpostavljen. Za izračun se koristila formula $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$ [3].



Slika 4: Stopa vjerojatnosti lažno pozitivnih vrijednosti za Bloom filter

3. Implementacija Bloom filtra

Bloom Filter podržava dvije osnovne operacije, a to su dodavanje i upit. Da bi se dodao (tj. umetnuo) element u filtar, element se prosljeđuje svakoj od k funkcija sažimanja i generira se k indeksa polja Bloom filtra. Odgovarajući bitovi u nizu tada se postavljaju na 1. Za provjeru prisutnosti elementa u skupu, element se prosljeđuje svakoj od funkcija sažimanja kako bi se dobili indeksi polja. Ako se utvrdi da bilo koji od bitova na tim pozicijama polja 0, tada se potvrđuje da element nije u skupu. Ako su svi bitovi na tim pozicijama postavljeni na 1, tada je element ili doista prisutan u skupu ili su bitovi postavljeni na 1 slučajno kada su drugi elementi umetnuti, što uzrokuje lažni pozitivni rezultat [4].

Nemoguće je ukloniti element iz Bloom filtra. To je zato što ako se odgovarajući bitovi elementa koji se uklanja vraćaju na 0, nema jamstva da ti bitovi ne nose informacije za druge elemente. Ako se jedan od tih bitova poništi, drugi element koji je slučajno kodiran na isti bit dat će negativan rezultat kada se upita, otuda izraz lažno negativan. Stoga bi jedini način uklanjanja elementa iz Bloom filtra bio rekonstruirati sam filtar bez neželjenog elementa. Rekonstrukcijom filtra lažno negativni rezultati bi bili eliminirani [4].

Stoga, sljedeće operacije će se obraditi u ovom seminarskom radu:

- Konstruktor (ujedno i destruktor) $CreateBF()$
- Transformator $InsertBF(v)$
- Opservatori $IsElementBF(v)$

3.1. ATP Bloom filter pomoću polja

Na početku $bloom.h$ datoteke definiraju se osnovne postavke:

```
1  #ifndef BLOOM
2  #define BLOOM 1000
3  #endif
4
5  #include <iostream>
6  #include <fstream>
7  #include <bitset>
8  #include <functional>
9
10 template <typename elementtype, int velicina_polja = BLOOM>
11
12 class BloomFilter ...
```

Ovaj dio koda koristi pretprocesorske direktive kako bi se definirala veličina polja u kojeg se smješta Bloom filter (konkretno nazvan "BLOOM"). U ovom primjeru se određuje polje veličine 1000 elemenata. Korisnik može u svom programu definirati isto, ali ako zaboravi ili ne definira tu vrijednost, ista je određena ovom linijom koda. Ako korisnik definira veličinu, tada se taj dio koda preskoči. Nakon toga se definira klasa $BloomFilter$.

Potom se definira metoda $CreateBF()$ unutar klase $BloomFilter$:

```
1   void CreateBF()
2   {
3       filter.reset();
4       std::ofstream izlazni_dokument(ime_dokumenta);
5       if (izlazni_dokument.is_open())
6       {
7           izlazni_dokument << bloomfilter;
8           std::cout << "Bloom filter je kreiran." << std::endl;
9           izlazni_dokument.close();
10      }
11      else
12      {
13          std::cerr << "Nije moguce kreirati Bloom filter." << std::endl;
14      }
15  }
```

Ovom metodom se prvo resetiraju svi bitovi u filtru na vrijednost 0 što osigurava da je filter prazan prije nego što se bilo koji element dodaje u njega. Nakon otvaranja datoteke koja je u ovom slučaju $bloom.txt$ (imenovana je u private dijelu u nastavku koda) i provjere uspješnosti otvaranja datoteke zapisuje se trenutno stanje u otvorenu datoteku i daje se poruka kako je filter kreiran. Na kraju se zatvara otvorena datoteku. Ako otvaranje datoteke nije uspjelo, ispisuje se poruka "Nije moguce kreirati Bloom filter". Iz ovog se može zaključiti kako metoda $CreateBF$ može služiti i za destrukciju samog polja i postavljanja na početnu vrijednost bez i jednog elementa u filtru.

Zatim se definira metoda $InsertBF()$ unutar klase $BloomFilter$:

```
1   void InsertBF(const elementtype &vrijednost)
2   {
3       std::fstream file(ime_dokumenta, std::ios::out | std::ios::in);
4       if (file.is_open())
5       {
6           file >> bloomfilter;
7
8           std::hash<elementtype> funkcija_sazimanja;
9           std::size_t sazetak1=funkcija_sazimanja(vrijednost)%velicina_polja;
10          std::size_t sazetak2=(funkcija_sazimanja(vrijednost)>>16)%velicina_polja;
11          std::size_t sazetak3=(funkcija_sazimanja(vrijednost)>>32)%velicina_polja;
12
13          bloomfilter.set(sazetak1, true);
14          bloomfilter.set(sazetak2, true);
15          bloomfilter.set(sazetak3, true);
16
17          file.seekp(0);
18          file << bloomfilter;
19          file.close();
20
21          std::cout << "Vrijednost '" << vrijednost << "' je dodana u Bloom filter."
                << std::endl;
22      }
23      else
```

```
24      {
25          std::cerr << "Nije moguce azurirati Bloom filter." << std::endl;
26      }
27  }
```

Da bi se definirala metoda $InsertBF$ unutar klase $BloomFilter$ otvara se datoteka za čitanje i pisanje (datoteka je već kreirana). Provjerava se je li otvaranje datoteke uspjelo. Ako je otvaranje uspjelo, čita se trenutno stanje filtera iz datoteke. Nakon toga, stvara se objekt $funkcija\_sazimanja$ koji se koristi za generiranje sažetka vrijednosti. Tri različite vrijednosti sažetka se generiraju primjenom $funkcija\_sazimanja$. Ove vrijednosti sažetka se zatim koriste za postavljanje odgovarajućih bitova na 1. Postavlja se pokazivač za pisanje na početak datoteke kako bi se filter zapisao na početak datoteke te se zapisuje trenutno stanje filtera u datoteku. Na kraju se zatvara otvorena datoteka. Kao izlaznu informaciju o provedenom, ispisuje se poruka kako je vrijednost dodana u Bloom filter. Ako otvaranje datoteke nije uspjelo, ispisuje se poruka da nije moguće ažurirati Bloom filter.

U nastavku se definira $IsElementBF()$ metoda unutar klase $BloomFilter$:

```
1   bool IsElementBF(const elementtype &vrijednost)
2   {
3       std::ifstream file(ime_dokumenta);
4       if (file.is_open())
5       {
6           file >> bloomfilter;
7           file.close();
8
9           std::hash<elementtype> funkcija_sazimanja;
10          std::size_t sazetak1=funkcija_sazimanja(vrijednost)%velicina_polja;
11          std::size_t sazetak2=(funkcija_sazimanja(vrijednost)>>16)%velicina_polja;
12          std::size_t sazetak3=(funkcija_sazimanja(vrijednost)>>32)%velicina_polja;
13
14          return bloomfilter.test(sazetak1) && bloomfilter.test(sazetak2) &&
                bloomfilter.test(sazetak3);
15      }
16      else
17      {
18          std::cerr << "Nije moguce otvoriti Bloom filter za provjeru." << std::endl;
19          return false;
20      }
21  }
```

Metoda $IsElementBF()$ otvara datoteku za čitanje i provjerava je li otvaranje datoteke uspjelo. Ako je otvaranje uspjelo, čita se trenutno stanje filtera iz datoteke. Zatim se stvara objekt $funkcija\_sazimanja$ koji se koristi za generiranje vrijednosti sažetka nakon čega se generiraju tri različite vrijednosti sažetka. Ove vrijednosti sažetka se zatim koriste za provjeru postavljenosti bitova. Nakon toga se zatvara otvorena datoteka. Na kraju, metoda vraća logički rezultat koji provjerava jesu li svi provjereni bitovi u filteru postavljeni na 1. Ako jesu, vraća se $true$ inače se vraća $false$. Ako otvaranje datoteke nije uspjelo, ispisuje se poruka o nemogućnosti otvaranja filtra za provjeru.

Na kraju je deklariran privatan dio klase $BloomFilter$ i sadrži dvije privatne varijable.

```
1  private:
2    const std::string ime_dokumenta = "bloom.txt";
3    std::bitset<velicina_polja> bloomfilter;
```

Deklarirano je ime datoteke i to $bloom.txt$ te objekt tipa $bitset$ koji se koristi za pohranjivanje bitova Bloom filtera. Veličina bitseta je određena parametrom $velicina\_polja$, koji je postavljen na vrijednost $BLOOM$ ili 1000 (prema definiciji iznad). Ovaj bitset predstavlja sam Bloom filter i koristi se za postavljanje ili provjeru postavljenosti pojedinih bitova u filteru. Ključna riječ $private$ označava da su ove varijable privatne, što znači da im se može pristupiti samo unutar klase $BloomFilter$ i nisu dostupne izvan nje. Takav način rada osigurava enkapsulaciju, ograničavajući izravan pristup tim varijablama izvan klase.

3.2. Korištenje ATP Bloom filter

U nastavku će se opisati program koji koristi ATP Bloom filter. Datoteka je nazvana $bloom.cpp$ i ista u svojim prvim redovima započinje uključivanjem potrebne biblioteke i $bloom.h$ datoteke. Deklariraju se tipovi podataka i konstanta $velicina\_polja$, koja predstavlja veličinu Bloom filtera. Da se veličina filtra nije odredila u korisničkom programu, ista bi ipak bila određena pošto se veličina deklarira i u $bloom.h$ datoteci te ako nije deklarirano ranije, bilo bi deklarirano kroz $bloom.h$ datoteku.

```
1  #include <iostream>
2  #include "bloom.h"
3
4  typedef std::string elementtype;
5
6  const int velicina_polja = BLOOM;
7  BloomFilter<elementtype, velicina_polja> bloomFilter;
```

U funkciji $main()$ se koristi beskonačna petlja $while(true)$ kako bi se omogućilo korisniku da izvršava različite operacije sve dok ne odabere izlazak iz programa. Unutar petlje se ispisuje izbornik s različitim opcijama za manipulaciju Bloom filtrom. Kada korisnik odabere neku operaciju, unosi se odgovarajući broj izbornika (varijabla izbor) putem standardnog ulaza. Zatim se koristi $switch$ izraz za odabir odgovarajuće akcije na temelju unesenog broja.

Ako je izbor 1, poziva se metoda $CreateBF()$ na $bloomFilter$ objektu, koja stvara novi Bloom filter. Kod izbora broja 2 korisnika se traži unos vrijednosti, a zatim se poziva metoda $InsertBF()$ na $bloomFilter$ objektu, koja dodaje vrijednost u Bloom filter. Nakon izbora 3, korisniku se također traži unos vrijednosti, a zatim se poziva metoda $IsElementBF()$ na $bloomFilter$ objektu. Metoda provjerava je li vrijednost prisutna u Bloom filtru i vraća bool vrijednost ($true$ ako je vjerojatno prisutna, $false$ ako nije). Rezultat se ispisuje na standardnom izlazu. Po izboru broja 4, ispisuje se poruka o izlasku iz programa i petlja se prekida.

Ako korisnik unese neispravan broj (izbor koji nije 1, 2, 3 ili 4), ispisuje se poruka o neispravnom unosu i petlja se nastavlja. Nakon svake izvršene operacije, ispisuje se prazna linija radi razdvajanja ispisnih redaka. Kod završava s izrazom $return0$, što označava uspješan završetak programa.

```
1  int main()
2  {
3      int izbor;
4      while (true)
5      {
6          std::cout << "       Bloom filter - Izbornik       " << std::endl;
7          std::cout << "-------------------------------------" << std::endl;
8          std::cout << "1. Stvori Bloom filter               " << std::endl;
9          std::cout << "2. Dodaj vrijednost u Bloom filter   " << std::endl;
10         std::cout << "3. Provjeri vrijednost u Bloom filtru" << std::endl;
11         std::cout << "4. Izlazak iz programa               " << std::endl;
12         std::cout << "-------------------------------------" << std::endl;
13         std::cout << "Unesi broj zeljene operacije: ";
14         std::cin >> izbor;
15
16         switch (izbor)
17         {
18         case 1:
19             bloomFilter.CreateBF();
20             break;
21         case 2:
22         {
23             elementtype vrijednost;
24             std::cout << "Unesi vrijednost: ";
25             std::cin >> vrijednost;
26             bloomFilter.InsertBF(vrijednost);
27             break;
28         }
29         case 3:
30         {
31             elementtype vrijednost;
32             std::cout << "Unesi vrijednost: ";
33             std::cin >> vrijednost;
34             bool exists = bloomFilter.IsElementBF(vrijednost);
35             std::cout << "Vrijednost '" << vrijednost << (exists ? "' je mozda u" :
                   "' nije u") << " Bloom filtru." << std::endl;
36             break;
37         }
38         case 4:
39             std::cout << "Izlazak iz programa" << std::endl;
40             return 0;
41         default:
42             std::cout << "Neispravan unos. Pokusajte opet." << std::endl;
43         }
44         std::cout << std::endl;
45     }
46     return 0;
47 }
```

4. Izračun složenosti i amortizirane složenosti

Vremenska složenost algoritma odnosi se na procjenu koliko vremena je potrebno za izvršavanje algoritma u ovisnosti o veličini ulaza. Obično se izražava kroz notaciju $BigO$, koja pruža gornju granicu rasta vremena izvršavanja algoritma s povećanjem ulaza. Vremenska složenost je ključna za razumijevanje performansi algoritma i donošenje odluka o odabiru najboljeg algoritma za određeni problem [5].

Amortizirana složenost je koncept koji se koristi za analizu algoritama koji povremeno izvršavaju operacije s velikom složenošću, ali se u prosjeku izvršavaju brzo. Ova analiza uzima u obzir ukupno vrijeme izvršavanja niza operacija umjesto samo pojedinačnih operacija. To omogućuje da se visoka složenost pojedinačnih operacija "amortizira" kroz vrijeme u kojem se izvršavaju brze operacije [5].

4.1. Vremenska složenost Bloom filtra

Prvo će se prikazati vremenska složenost (za najgori slučaj) metode $CreateBF$.

```
1   void CreateBF()
2   {
3       bloomfilter.reset();                                    // O(c1)
4       std::ofstream izlazni_dokument(ime_dokumenta);          // O(c2)
5       if (izlazni_dokument.is_open())                         // O(c3)
6       {
7           izlazni_dokument << bloomfilter;                    // O(c4)
8           std::cout << "Bloom filter je kreiran." << std::endl;  // O(b1)
9           izlazni_dokument.close();                           // O(c5)
10      }
11      else
12      {
13          std::cerr << "Nije moguce kreirati Bloom filter." << std::endl;
                                                                // O(b2)
14      }
15  }
```

Kako se radi o jednostavnoj operaciji inicijalizacije polja s unaprijed određenim brojem elemenata koji se postavljaju na vrijednost 0 vremenska kompleksnost je (ako je $c_6 = max\{b_1, b_2\}$):

$$\begin{aligned} T_{max}^{CreateBF} &\leq c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \\ &= d_1 \end{aligned}$$

Stoga, za operaciju kreiranja Bloom filtra se može reći kako iznosi

$$T_{max}^{CreateBF}(n) = O(1)$$

```
1     void InsertBF(const elementtype &vrijednost)
2 {
3     std::fstream file(ime_dokumenta, std::ios::out | std::ios::in);          // O(c1)
4     if (file.is_open())                                                      // O(c2)
5     {
6     file >> bloomfilter;                                                     // O(c3)
7
8     std::hash<elementtype> funkcija_sazimanja;                               // O(c4)
9     std::size_t sazetak1=funkcija_sazimanja(vrijednost)%velicina_polja;      // O(c5)
10    std::size_t sazetak2=(funkcija_sazimanja(vrijednost)>>16)%velicina_polja// O(c6)
11    std::size_t sazetak3=(funkcija_sazimanja(vrijednost)>>32)%velicina_polja// O(c7)
12
13    filter.set(sazetak1, true);                                              // O(c8)
14    filter.set(sazetak2, true);                                              // O(c9)
15    filter.set(sazetak3, true);                                             // O(c10)
16
17    file.seekp(0);                                                           // O(c11)
18    file << bloomfilter;                                                     // O(c12)
19    file.close();                                                            // O(c13)
20
21      std::cout << "Vrijednost '" << vrijednost << "' je dodana u Bloom filter." <<
            std::endl;                                                        // O(b1)
22    }
23    else
24    {
25        std::cerr << "Nije moguce azurirati Bloom filter." << std::endl;    // O(b2)
26    }
27 }
```

Operacija umetanja vrijednosti u polje ovisi o broju funkcija sažimanja koji je unaprijed određen. Vremenska kompleksnost je (ako je $c_{14} = max\{b_1, b_2\}$):

$$\begin{aligned} T_{max}^{InsertBF} &\leq c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} \\ &= d_1 \end{aligned}$$

Stoga, za operaciju umetanja vrijednost u Bloom filtar se može reći kako iznosi

$$T_{max}^{InsertBF}(k) = O(1)$$

Ostalo je još za odrediti složenost provjere operacije kojom se provjerava je li uneseni element u Bloom filtru.

```
1     bool IsElementBF(const elementtype &vrijednost)
2     {
3         std::ifstream file(ime_dokumenta);                                   // O(c1)
4         if (file.is_open())                                                  // O(c2)
5         {
6             file >> bloomfilter;                                             // O(c3)
7             file.close();                                                    // O(c4)
8             std::hash<elementtype> funkcija_sazimanja;                       // O(c5)
```

```
9     std::size_t sazetak1=funkcija_sazimanja(vrijednost)%velicina_polja;      // O(c6)
10    std::size_t sazetak2=(funkcija_sazimanja(vrijednost)>>16)%velicina_polja // O(c7)
11    std::size_t sazetak3=(funkcija_sazimanja(vrijednost)>>32)%velicina_polja // O(c8)
12
13    return bloomfilter.test(sazetak1) && bloomfilter.test(sazetak2) && bloomfilter.
          test(sazetak3);                                                      // O(b1)
14    }
15    else
16    {
17    std::cerr <<"Nije moguce otvoriti Bloom filter za provjeru."<<std::endl;// O(b2)
18    return false;                                                            // O(b3)
19    }
20 }
```

Operacija provjere vrijednosti, slično operaciji umetanja vrijednosti u polje, ovisi o broju funkcija sažimanja. S unaprijed određenim brojem funkcija (u ovom slučaju 3) sažimanja vremenska kompleksnost je (ako je $c_9 = max\{b_1, b_2 + b_3\}$):

$$\begin{aligned} T_{max}^{IsElementBF} &\leq c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 \\ &= d_1 \end{aligned}$$

Stoga, za operaciju provjere vrijednosti u Bloom filtar se može reći kako je

$$T_{max}^{IsElementBF}(k) = O(1)$$

4.2. Amortizirana složenost Bloom filtra

Nakon izračuna vremenske složenosti Bloom filtra može se pristupiti izračunu amortizirane složenosti. Koristit će se metoda energetskih potencijala. Izračunat će se vrijednosti za operacije umetanja $InsertBF$ i provjere $IsElementBF$ pošto Bloom filter nema operaciju brisanja vrijednosti. Stvarne cijene operacija su 1 i to je prikazano u Tablici 2.

Tablica 2: Stvarne cijene operacija

Operacija	Cijena
$InsertBF$	1
$IsElementBF$	1

Potom se može prijeći na izračun amortizirane vrijednosti. Samo operacija $InsertBF$ se povećava potencijal strukture podataka za 1 tj. $\Phi(D_i) = \Phi(D_{i-1}) + 1$ odnosno $\Phi(D_i) - \Phi(D_{i-1}) = 1$. Stoga je

$$\begin{aligned} \hat{c_i}(InsertBF) &= c(InsertBF) + 1 \\ &= 1 + 1 = 2 \end{aligned}$$

Iz istog proizilazi Tablica 3 :

Tablica 3: Amortizirane cijene operacija

Operacija	Cijena
$InsertBF$	2
$IsElementBF$	1

Najgori slučaj je kada se izvede N uzastopnih operacija $InsertBF$ i tada je

$$\begin{aligned}\sum_{i=1}^{N} c_i\left(O_i\right) &\leq \sum_{i=1}^{N} \hat{c}_i\left(O_i\right) \\ &\leq \sum_{i=1}^{N} \hat{c}_i\left(InsertBF\right) \\ &= \sum_{i=1}^{N} 2 \\ &= 2N\end{aligned}$$

Pa je:

$$T^{BloomFilter}\left(N\right) = O\left(N\right)$$

5. Zaključak

Ovaj rad je pružio pregled Bloom filtera kao probabilističke strukture podataka za efikasnu provjeru prisustva elemenata u skupu. Prikazani su osnovni principi Bloom filtra, njegova implementaciju i karakteristike. Također su razmotrene prednosti i nedostaci ove strukture podataka, kao i neke od njegovih primjena u stvarnim situacijama.

Bloom filter je posebno koristan u situacijama gdje je brzina ključna, a mala vjerojatnost greške pri dobijanju lažno pozitivnih rezultata je prihvatljiva. On pruža kompaktno rješenje za provjeru prisustva elemenata u velikim skupovima podataka bez potrebe za čuvanjem samih elemenata. Međutim, mora se uzeti u obzir da Bloom filter može dati lažno pozitivne rezultate, ali nikada lažno negativne.

Iako Bloom filter ima svoje prednosti, kao što su niski memorijski zahtjevi i brze operacije provjere prisustva, treba biti oprezan prilikom odabira ove strukture podataka. Njegova efikasnost ovisi od odgovarajućeg odabira parametara, poput veličine filtera i broja funkcija sažimanja te od pravilne procjene potreba za točnošću rezultata. U nekim slučajevima, gdje je potrebna apsolutna točnost, bilo bi potrebno razmotriti alternative.

Popis slika

Popis tablica

Popis literature

[1] M. La Rocca, *Advanced Algorithms and Data Structures*. Simon i Schuster, 2021.

[2] Y. Wu, J. He, S. Yan i dr., „Elastic bloom filter: deletable and expandable filter using elastic fingerprints," *IEEE Transactions on Computers*, sv. 71, br. 4, str. 984–991, 2021.

[3] S. Tarkoma, C. E. Rothenberg i E. Lagerspetz, „Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, sv. 14, br. 1, str. 131–155, 2011.

[4] O. MEI, „String Algorithm on GPGPU," 2013.

[5] A. Levitin, *Introduction to the Design and Analysis of Algorithms: International Edition*. Pearson Education, 2014., ISBN: 9781292014111. adresa: `https://books.google.hr/books?id=8KapBwAAQBAJ`.