SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT

The Logarithmic Dynamic Cuckoo Filter

David Iveković, Ana Ujević

Voditelj: Mirjana Domazet-Lošo

Zagreb, svibanj, 2024.

**Sadržaj**

[1. Uvod 2](#_Toc454578710)

[2. Opis problema 3](#_Toc1554595940)

[2.1 Struktura LDCF 4](#_Toc598102305)

[2.2 Opis algoritma 4](#_Toc784229468)

[3. Objašnjenje algoritma kroz primjer + vizualizacija 5](#_Toc963511126)

[4. Rezultati mjerenja 6](#_Toc1403056809)

[4.1 Testiranje umjetno generiranih podataka 7](#_Toc1167555861)

[4.2. Testiranje na stvarnim podacima 10](#_Toc1163950110)

[5. Zaključak 11](#_Toc1695402467)

[6. Sažetak 12](#_Toc899134136)

[7. Literatura 13](#_Toc654191289)

# Uvod

Logaritamski dinamički Cuckoo filter (LDCF) je posebna vrsta strukture podataka koja se koristi za brzo provjeravanje prisustva ili odsustva elemenata u velikim skupovima podataka. On koristi kombinaciju logaritamske dinamike i Cuckoo filtra kako bi efikasno upravljao memorijom i brzo obavljao provjere.

Prije su se za filtriranje i provjeru prisutnosti elemenata u skupu podataka često koristile druge strukture podataka poput hash tablica ili Bloom filtara. Hash tablice su bile popularne zbog brzog pristupa elementima, ali su mogle zahtijevati puno memorije pogotovo kod velikih skupova podataka. Bloom filteri su smanjili potrebu za memorijom koristeći probabilistički pristup, ali su imali ograničenja u tome što su mogli dati lažno pozitivne rezultate (tj. reći da je element prisutan kada zapravo nije). [1] [2]

Taj problem je ublažen s dolaskom LDCF filtra jer je riješio problem prekomjerne potrošnje memorije uz istovremeno održavanje brze i pouzdane provjere prisutnosti elemenata. Danas se koristi u mnogim aplikacijama koje zahtijevaju efikasno upravljanje velikim skupovima podataka, poput baza podataka, sustava za pretraživanje i distribuiranih sustava pohrane podataka

Cilj našeg projekta je napraviti vlastitu LDCF implementaciju te napraviti analizu točnosti, analizu vremena izvođenja i analizu utroška memorije.

# Opis problema

Kao što je spomenuto u uvodu, s porastom velikih skupova podataka, javila se i potreba za boljim predstavljanjem i organiziranjem skupova podataka. Različite strukture podataka i tehnike, kao što su hash kodiranje, Bloom filter (BF), Cuckoo filter (CF) i Dynamic Cuckoo filter (DCF), korištene su u svrhu provjere članstva elemenata u skupovima podataka. Međutim, svaka od ovih tehnika ima svoje nedostatke, a potreba za još učinkovitijim strukturama podataka i dalje postoji.U tom kontekstu, Logarithmic Dynamic Cuckoo Filter (LDCF) predstavlja napredak koji kombinira prednosti prethodnih tehnika uz dodatna poboljšanja.[1]

LDCF koristi kombinaciju hash funkcija, cuckoo hashiranja i dinamičkog prilagođavanja veličine kako bi osigurao visoku učinkovitost u vremenu izvođenja i memoriji. Ovi koncepti omogućuju efikasno upravljanje memorijskim resursima i održava performanse čak i s promjenjivim brojem elemenata.

## Struktura LDCF

Osnovni dizajn Logaritamskog Dinamičkog Cuckoo Filtera (LDCF) koristi **višerazinsku strukturu stabla** za organizaciju blokova Cuckoo filtera (CF), za razliku od klasičnih lančanih struktura. Ova struktura omogućuje LDCF-u da bude prilagodljiv dinamičkim promjenama u skupovima podataka, smanjujući istovremeno računske troškove testiranja i memorijske zahtjeve na logaritamsku razinu.

Struktura LDCF-a sastoji se od **više razina**, gdje svaka razina predstavlja određenu dubinu u hijerarhijskoj organizaciji. Na vrhu strukture nalazi se korijenska razina, koja sadrži blokove Cuckoo filtera koji se koriste kao temelj za ostale razine. Svaka razina sadrži blokove CF-a, pri čemu svaki blok sadrži određeni broj otisaka ili hash vrijednosti.

Ključna značajka ovog dizajna je sposobnost skaliranja i prilagođavanja strukture prema potrebama skupa podataka. Kada se skup podataka poveća ili smanji, LDCF može dinamički dodavati ili uklanjati blokove na odgovarajućim razinama, čime se održava učinkovitost operacija umetanja, provjere članstva i brisanja elemenata.

Dodatna prednost ove strukture je učinkovito upravljanje memorijom jer se memorijski resursi koriste samo onoliko koliko je potrebno za trenutni skup podataka.

## Opis algoritma

Počinje s jednim CF-om na najnižoj razini stabla. Kada se taj CF napuni, dodaju se dodatne razine stabla, a umetanje se nastavlja u nove CF-ove. Na svakoj razini, umetanje se odvija prema uzorku otisaka elemenata. Na primjer, na prvoj razini, elementi s otiscima koji počinju s '0' umetnut će se u jedan od CF-ova, dok će elementi s otiscima koji počinju s '1' biti umetnuti u drugi CF.

Ova struktura omogućuje učinkovito pretraživanje elemenata provjerom otisaka na svakoj razini stabla. Vrijeme potrebno za pretragu u LDCF-u s l razina je O(l), što je znatno manje od linearnog vremena potrebnog za pretragu u drugim strukturama podataka poput DCF filtra. Iako umetanje u LDCF može zahtijevati nešto više računalne obrade nego u DCF-u, to se kompenzira bržim pretraživanjem.

Algoritam LCDF koristi sljedeće funkcije: [1]

* **Funkcija “insert”**

Operacija umetanja prolazi kroz stablo CF-a kako bi pronašla odgovarajući listni CF za umetanje novog elementa. Nakon pronalaska odgovarajućeg CF-a, element se umetne u prazni redak (eng. bucket). Ako je bucket pun, vrši se nasumično premještanje elemenata kako bi se oslobodio prostor za novi element. Ako je cijeli CF pun, dodaju se novi CF-ovi na istoj razini stabla kako bi se proširila struktura.

* **Operacija “membership test”**

Membership test provjerava prisutnost određenog elementa unutar LDCF-a. Algoritam koristi hash funkciju za generiranje otiska (eng. fingerprint) elementa, te pretražuje kroz odgovarajuće CF-ove u stablu. Ako je element pronađen, vraća se pozitivni rezultat, inače negativni.

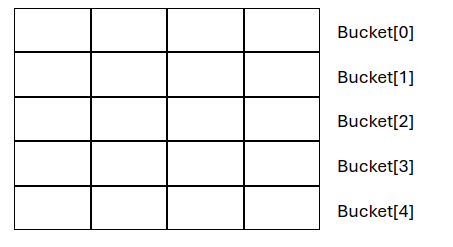
* **Funkcija “delete”**

Operacija brisanja uklanja određeni element iz LDCF-a. Algoritam pretražuje stablo CF-ova kako bi pronašao i uklonio element. Ako se element pronađe, uklanja se.

# Objašnjenje algoritma kroz primjer

Pokazat ćemo rad algoritma na jednostavnom primjeru tako što ćemo uzeti string “ATCGGGCCTC”.

Naša tablica se sastoji od 4 stupca i 5 bucketa (redaka).



1. **Izračunamo fingerprint tako da napravimo hash od zadanog stringa**

fingerprint = hash(“ATCGGGCCTC”) => 10001

1. **Računamo index1 – broj redka u koji se ubacuje u tablici**

index1 = hash(“ATCGGGCCTC”) % broj\_bucketa (redaka)

Index1 = 10001 % 5

Index1 = 1

3. **Ubacivanje fingerprint-a u bucket**

* **1. slučaj: Bucket[1] je prazan**

Provjera slobodnog mjesta u bucketu na index1 (bucket 1)

Pretpostavljamo da svaki bucket sadrži 4 slobodna mjesta (imamo 4 stupca)



S obzirom da ima slobodnog mjesta, ubacujemo fingerprint 10001 u prvi slobodan stupac.



* **2. slučaj: Bucket[1] je pun -> koristimo alternativni index**

Alternativni\_index = index1 xor hash(fingerprint) % broj\_redaka

Alternativni\_index = index1 xor hash(10001) % 5

Alternativni\_index = *broj (npr. 4)*

* Ubacivanje u alternativni bucket
* Provjera slobodnog mjesta u bucketu na alternativnom\_index (4):
* Ako ima slobodnog mjesta, ubacujemo fingerprint 10001.

**3. Slučaj: nema slobodnog mjesta ni u jednom bucketu (index1 i alt\_index) te provodimo realokaciju**

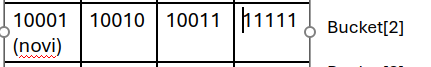
Odabir nasumičnog stupca za izbacivanje u index1 ili alt\_index:

Pretpostavimo da je bucket [1] pun i odabiremo nasumičan stupac, npr. drugi stupac.

* Izbacivanje fingerprint-a:

Izbacujemo fingerprint 11001 iz odabranog stupca i ubacujemo novi fingerprint 10001.

Y\_izbačeni = 11001



* Generiranje novog alt\_index-a za izbačeni fingerprint:

Alt\_index(y\_izbačeni) = index1 xor hash(11001) % 5

Alt\_index (y\_izbačeni) = *npr. 3*

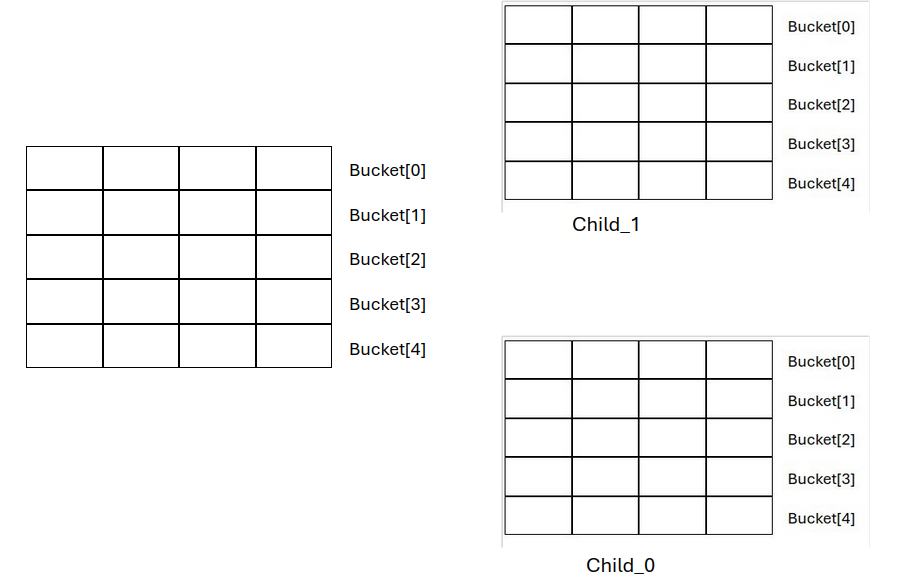
* Provjera slobodnog mjesta u novom alt\_index bucketu:

Ako ima slobodnog mjesta u bucket[3], ubacujemo fingerprint 11001.

**Kada broj relokacije dođe do maksimuma:**

* Kreiranje novih slojeva:

Dva nova childa: child\_0 i child\_1.



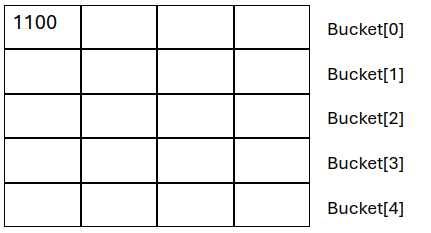
* Odabir childa za premještanje fingerprint-a:

Ako fingerprint završava sa 1, premještamo ga u child\_1, inače u child\_0.

Y = 11001

Završava sa 1 pa ga premještamo u child\_1, ali bez zadnjeg bita (1100)

Child\_1 - premještamo fingerprint `1100` bez zadnjeg bita



**OPERACIJA MEMBERSHIP CHECK – provjera prisutnosti**

1. Izračunavanje fingerprinta i index1

fingerprint = hash(“ATCGGGCCTC”) => 10001

index1 = hash(“ATCGGGCCTC”) % broj\_bucketa

alt\_index = hash(10001) xor index1 % broj\_bucketa

1. Provjeravamo postoji li na index1 ili alternativni index broj 1001
2. ako ne postoji:
   1. Idemo u child (u ovom slučaju child\_1)
      1. Unutar child\_1 tražimo 1000 -> nalazimo

**OPERACIJA DELETE**

1. Izračunavanje fingerprinta i index1

fingerprint = hash(“ATCGGGCCTC”) => 10001

index1 = hash(“ATCGGGCCTC”) % broj\_bucketa

alt\_index = hash(10001) xor index1 % broj\_bucketa

1. Provjeravamo postoji li na index1 ili alternativni index broj 1001
2. U slučaju da ne bi postojalo:
   1. Idemo u child (u ovom slučaju child\_1)
      1. Unutar child\_1 tražimo 1000 -> nalazimo -> označavamo kao izbrisano

# **Rezultati mjerenja**

## **Testiranje umjetno generiranih podataka**

* **Način generiranja podataka**

Napravili smo funkciju koja generira slova iz normalne distribucije u range-u od “a” do “z” te smo ta slova spojili u string veličine 10.

* **Specifikacije podataka (broj uzoraka, dimenzije, karakteristike)**
  + Podaci su stringovi, odnosno random generirane riječi.
  + Svako slovo je nasumično generiran znak (ASCII vrijednost od 97 do 122).
  + Broj uzoraka je 1000 do 10 milijuna.
  + False postive rate: 0.01
* **Procedura testiranja**

Probali smo za set sizeove od 103 na 107

**Analiza točnosti**

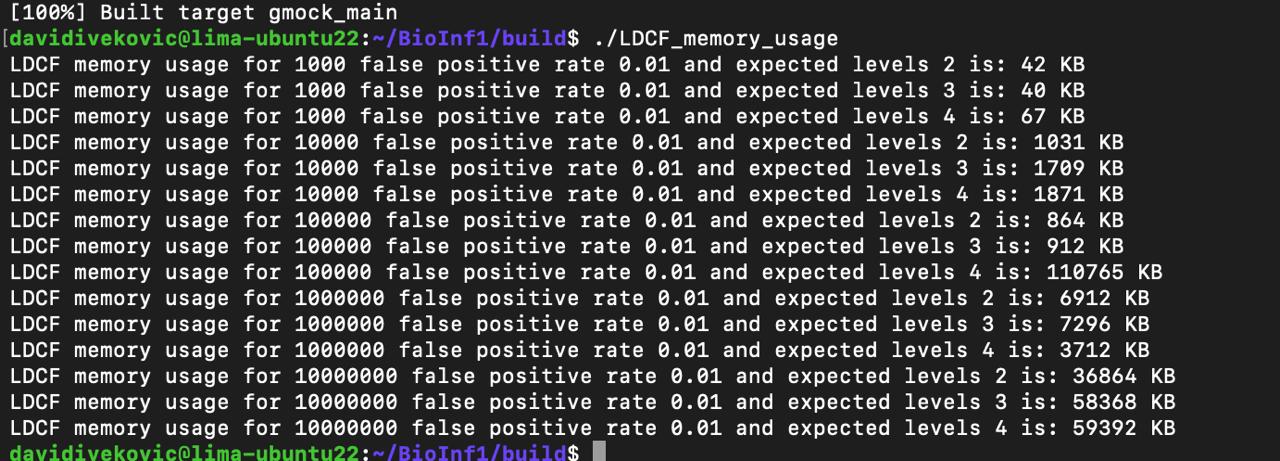
Pri mjerenju točnosti uspoređivali smo naš implementirani LDCF filtar sa Bloom Filtrom.

Parametri: false\_positive\_rate, set\_size, expected\_levels

|  |  |  |
| --- | --- | --- |
| Redni broj | Testni slučaj: false\_positive\_rate, set\_size, expected\_levels | Rezultati |
| 1. | Testni slučaj: 0.001, 1 000 000, 3 | number of elements in map 1000000  number of false strings 10000  LDCF Insert Time: 284 ms  Bloom Filter Insert Time: 421 ms  LDCF Check Time: 93 ms  Bloom Filter Check Time: 412 ms  LDCF False Positive Rate: 0.009998  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 9e-06  Elements not present in Bloom Filter: 50069 |
| 2. | Testni slučaj: 0.001, 100 000, 3 | number of elements in map 10000  number of false strings 100  LDCF Insert Time: 9 ms  Bloom Filter Insert Time: 4 ms  LDCF Check Time: 6 ms  Bloom Filter Check Time: 4 ms  LDCF False Positive Rate: 0.0007  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0  Elements not present in Bloom Filter: 0 |
| 3. | Testni slučaj: 0.001, 1 000, 3 | number of elements in map 1000  number of false strings 10  LDCF Insert Time: 1 ms  Bloom Filter Insert Time: 0 ms  LDCF Check Time: 0 ms  Bloom Filter Check Time: 0 ms  LDCF False Positive Rate: 0.001  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0  Elements not present in Bloom Filter: 0 |
| 4. | Testni slučaj: 0.001, 10 000, 2 | number of elements in map 10000  number of false strings 100  LDCF Insert Time: 6 ms  Bloom Filter Insert Time: 3 ms  LDCF Check Time: 4 ms  Bloom Filter Check Time: 3 ms  LDCF False Positive Rate: 0.0011  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.0001  Elements not present in Bloom Filter: 0 |
|  | Testni slučaj: 0.001, 500 000, 3 | number of elements in map 50000  number of false strings 500  LDCF Insert Time: 70 ms  Bloom Filter Insert Time: 18 ms  LDCF Check Time: 36 ms  Bloom Filter Check Time: 18 ms  LDCF False Positive Rate: 0.00314  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0  Elements not present in Bloom Filter: 0 |
|  | Testni slučaj: 0.001, 200 000, 2 | number of elements in map 200000  number of false strings 2000  LDCF Insert Time: 4733 ms  Bloom Filter Insert Time: 72 ms  LDCF Check Time: 357 ms  Bloom Filter Check Time: 112 ms  LDCF False Positive Rate: 0.00956  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.000165  Elements not present in Bloom Filter: 0 |
|  | Testni slučaj: 0.01, 50 000, 3 | number of elements in map 500000  number of false strings 5000  LDCF Insert Time: 151 ms  Bloom Filter Insert Time: 166 ms  LDCF Check Time: 36 ms  Bloom Filter Check Time: 169 ms  LDCF False Positive Rate: 0.009996  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.000144  Elements not present in Bloom Filter: 1038 |
| 8. | Testni slučaj: 0.01, 750 000, 1 | number of elements in map 750000  number of false strings 7500  LDCF Insert Time: 262 ms  Bloom Filter Insert Time: 248 ms  LDCF Check Time: 95 ms  Bloom Filter Check Time: 226 ms  LDCF False Positive Rate: 0.00945467  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.000126667  Elements not present in Bloom Filter: 31935 |
| 9. | Testni slučaj: 0.01, 300 000, 2 | number of elements in map 300000  number of false strings 3000  LDCF Insert Time: 76 ms  Bloom Filter Insert Time: 98 ms  LDCF Check Time: 23 ms  Bloom Filter Check Time: 98 ms  LDCF False Positive Rate: 0.00990667  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.000153333  Elements not present in Bloom Filter: 144 |
| 10. | Testni slučaj: 0.01, 1 000 0000, 4 | number of elements in map 9999999  number of false strings 100000  LDCF Insert Time: 4456 ms  Bloom Filter Insert Time: 5161 ms  LDCF Check Time: 3038 ms  Bloom Filter Check Time: 10278 ms  LDCF False Positive Rate: 0.0099995  Elements not present in LDCF: 0  Bloom Filter False Positive Rate: 0.0001357  Elements not present in Bloom Filter: 606223 |

**Analiza vremena izvođenja**

**Analiza utroška memorije**

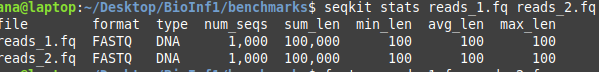


## 4.2. **Testiranje na stvarnim podacima**

* Način generiranja podataka

Preuzeli smo genom bakterije Escherichia Coli sa stranice [4] i pomoću wgsim alata [5] generirali dvije datoteke reads\_1.pq i reads\_2.fq koji sadrže genom bakterije.

* Specifikacije podataka (broj uzoraka, dimenzije, karakteristike)

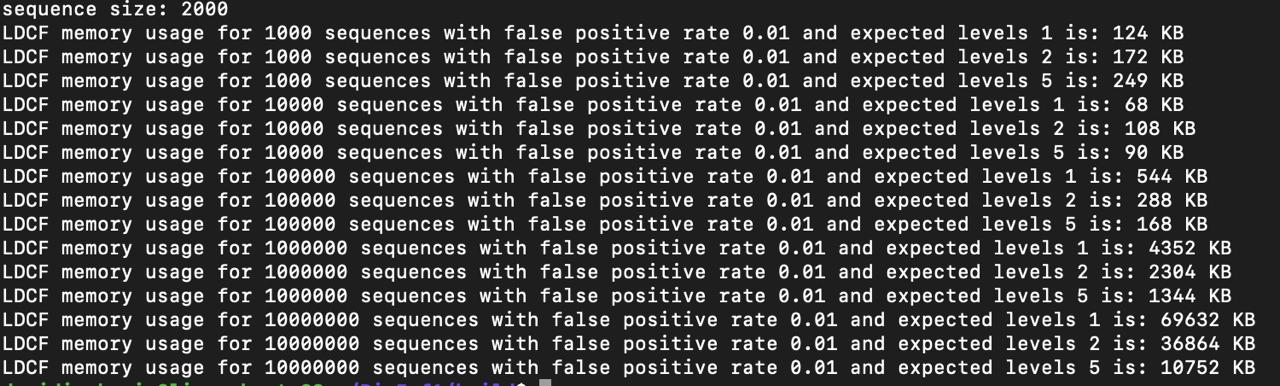


Analiza točnosti

Pokretali smo datoteku ./main <reads\_1.fq> <reads\_2.fq> <num\_strings> <string\_length> <false\_positive\_rate> <expected\_levels> gdje su reads\_1.fq i reads\_2.fq imena datoteka iz kojih čitamo genom, a ostalo su parametri koje mijenjamo pri mjerenju točnosti

Analiza vremena izvođenja

Analiza utroška memorije



# Zaključak

Iz rezultata testiranja našeg (LDCF) na možemo izvući nekoliko značajnih zaključaka. Primjećujemo da je LDCF pružio impresivne rezultate u smislu vremena umetanja i provjere elemenata. U većini testnih slučajeva, LDCF je demonstrirao značajno manja vremena umetanja i provjere u odnosu na tradicionalne Bloom filtre, čak i pri radu s velikim skupovima podataka. Drugo, razina lažno pozitivnih rezultata varira ovisno o veličini skupa podataka i parametrima testiranja. U većini slučajeva, LDCF je uspio održati nisku stopu lažno pozitivnih rezultata, što je ključno za pouzdanost filtriranja podataka. Treće, vrijeme umetanja i provjere elemenata u LDCF-u raste s povećanjem veličine skupa podataka i broja lažno pozitivnih rezultata. Ovo sugerira da je važno pažljivo prilagoditi parametre LDCF-a kako bi se postigla optimalna ravnoteža između brzine i točnosti.

# Sažetak

U našem projektu istraživali smo Logaritamski dinamički Cuckoo filter (LDCF), naprednu strukturu podataka koja omogućuje brzu provjeru prisutnosti ili odsutnosti elemenata u velikim skupovima podataka. LDCF kombinira logaritamsku dinamiku i Cuckoo filter kako bi efikasno upravljao memorijom i brzo obavljao provjere. Tradicionalne strukture poput hash tablica i Bloom filtara imale su ograničenja u potrošnji memorije i točnosti provjere, što je potaknulo razvoj LDCF-a. Cilj našeg projekta bio je implementirati vlastitu LDCF implementaciju te provesti analizu točnosti, vremena izvođenja i utroška memorije. Analiza rezultata testiranja pokazala je da LDCF pruža impresivne performanse u smislu vremena umetanja i provjere elemenata. Također, LDCF je uspio održati nisku stopu lažno pozitivnih rezultata u većini testnih slučajeva. Međutim, važno je prilagoditi parametre LDCF-a kako bi se postigla optimalna ravnoteža između brzine i točnosti, što sugerira potrebu za daljnjim istraživanjem i optimizacijom.

# Literatura

[1] Zhang et al. The Logarithmic Dynamic Cuckoo Filter

[2] Fan et al. 2014. Cuckoo Filter: Practically Better Than Bloom;

<http://www.cs.cmu.edu/%7Ebinfan/papers/conext14_cuckoofilter.pdf>

[3] Chen et al. 2017. The dynamic cuckoo filter; <https://ieeexplore.ieee.org/abstract/document/8117563>

[4] <http://bacteria.ensembl.org/Multi/Search/Results?species=all;idx=;q=coli;site=ensemblunit> (zadnji put posjećeno: 30.5.2024.)

[5] <https://github.com/lh3/wgsim> (zadnji put posjećeno: 30.5.2024.)