# Hash tablice

**Mrak, Jakov**

**Master's thesis / Diplomski rad**

**2024**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Rijeka / Sveučilište u Rijeci**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:195:812803

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Sveučilišni diplomski studij Informatika

Jakov Mrak

# Hash tablice

Diplomski rad

Mentor: izv. prof. dr. sc. Marija Brkić Bakarić

Rijeka, rujan 2024.

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Rijeka, 3.6.2024.

# Zadatak za diplomski rad

Pristupnik: Jakov Mrak

Naziv diplomskog rada: Hash tablice

Naziv diplomskog rada na engleskom jeziku: Hash tables

Sadržaj zadatka:
Cilj rada je analizirati funkcionalnosti strukture podataka hash tablica, usporediti ih sa strukturama kao što su polja fiksne duljine, povezane liste i binarna stabla te opisati njihove prednosti i nedostatke. U radu će se opisati i utjecaj veličine polja tablice na distribuciju hash vrijednosti ključeva koji nisu slučajno odabrani. U praktičnom dijelu rada biti će implementirane sve opisane funkcionalnosti.

Mentorica
Izv. prof. dr. sc. Marija Brkić Bakarić

Voditeljica za diplomske radove
Doc. dr. sc. Lucia Načinović Prskalo

Zadatak preuzet: 3.6.2024.

(potpis pristupnika)

Radmile Matejčić 2,
51000 Rijeka, Hrvatska

T: +385 (0)51/584-700
E: ured@inf.uniri.hr
www.inf.uniri.hr

OIB: 64218323816
IBAN: HR1524020061400006966

## Sažetak

U ovom radu analizirane su funkcionalnosti strukture podataka hash tablica i opisane uz primjer kôda implementacije napisane u programskom jeziku C. Implementacija sadrži funkcionalnosti unosa, brisanja i pristupanja elementima tablice, stvaranja i brisanja cijele strukture tablice te promjene veličine njezinog polja s elementima. Hash tablice su uspoređene sa strukturama kao što su polja fiksne duljline, povezane liste i binarna stabla te su opisane njihove prednosti i nedostatci. Uspoređene su različite mogućnosti kod implementacije kao što su otvoreno adresiranje ili pokazivači na povezane liste ili na binarna stabla. Opisan je utjecaj veličine polja tablice na distribuciju hash vrijednosti ključeva koji nisu slučajno odabrani.

**Ključne riječi:** hash tablica; struktura podataka; ključ; polje; povezana lista; binarno stablo; otvoreno adresiranje; algoritam; vremenska kompleksnost

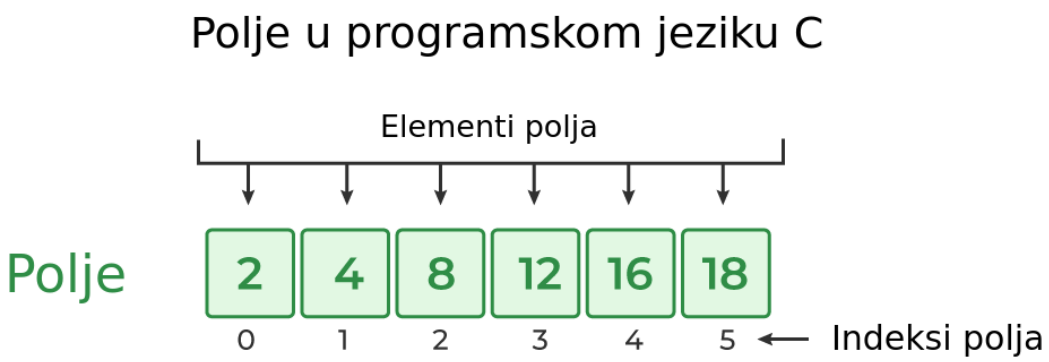
# SADRŽAJ

# 1. Uvod

Tema ovog rada je struktura podataka hash tablica (engl. *hash tables*). Bit će analizirane njezine prednosti i nedostatci u odnosu na druge strukture podataka, način na koji podatke pohranjuje u memoriji te algoritmi koji se koriste u implementaciji. U analizi će biti korišten primjer napisan u programskom jeziku C prema standardu C17 (službenog naziva ISO/IEC 9899:2018).

Hash tablice razlikuju se od osnovnih struktura podataka poput polja fiksne duljine (engl. *fixed-length array*), povezanih lista (engl. *linked list*) ili binarnih stabala (engl. *binary tree*) po tomu što omogućuju konstantnu prosječnu vremensku kompleksnost, odnosno O(1), za unos, pretraživanje i uklanjanje podataka.

## 1.1. Polja fiksne duljine

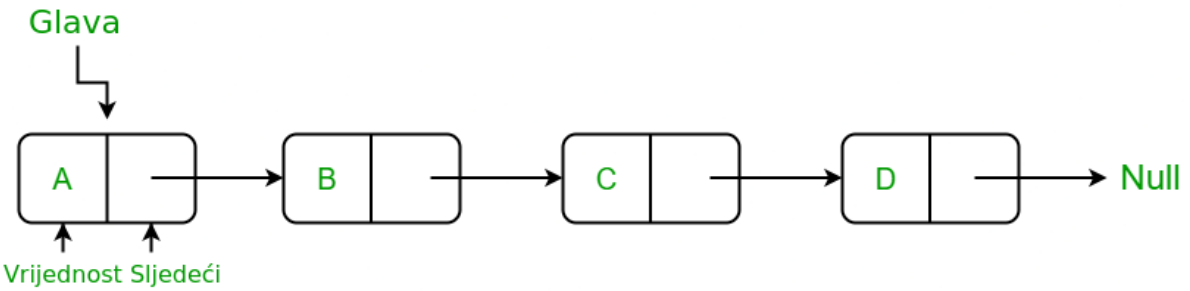



Slika 1. Polje fiksne duljine. Izvor: Prilagođeno iz [1]

Polja fiksne duljine su jedna od jednostavnijih struktura podataka. Implementirana su alociranjem bloka memorije prema unaprijed definiranom broju elemenata jednake veličine. Pošto su svi elementi jednako veliki, moguće je pristupanje svakom elementu putem njegove memorijske adrese koja je odmaknuta za definirani broj bajtova od prvog elementa (produkt razlike indeksa elemenata i duljina svakog elementa u bajtovima). Na Slici 1 prikazan je primjer polja od šest elemenata. Indeksiranjem elemenata od nule prikazana je udaljenost svakog elementa od prvog.

Glavni nedostatak ove strukture je to što umetanje elementa između dva postojeća zahtjeva kopiranje svih elemenata na višim indeksima na nove pozicije. Osim toga, ako polje nije dovoljno veliko za dodavanje novih elemenata, potrebna je alokacija novog bloka memorije zadovoljavajuće duljine i kopiranje cijelog polja u novi blok.

## 1.2. Povezane liste



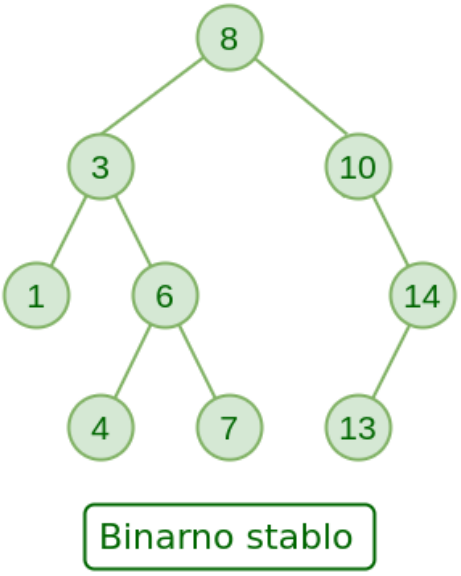

Slika 2. Povezana lista. Izvor: Prilagođeno iz [2]

Povezane liste ne koriste jedinstveni blok memorije za sve elemente, već pojedini blok za svaki od elemenata. Cijeloj strukturi pristupa se putem pokazivača na prvi element, kao i kod polja fiksne duljine, no nakon njega je pohranjen novi pokazivač za sljedeći element, i tako dalje. Zadnji element liste označen je pokazivačem `NULL` vrijednosti. Ovo vrijedi za jednostruko povezane liste, a ako je lista dvostruko povezana, uz svaki element bit će pohranjen i pokazivač na prethodni element.

Povezane liste olakšavaju umetanje elemenata tako što dodavanje novog elementa između dva postojeća ne zahtijeva pomicanje svih elemenata na višim pozicijama, već samo alokaciju bloka memorije za novi element i ažuriranje pokazivača prethodnog elementa, ili prethodnog i sljedećeg, ovisno o tomu je li lista jednostruko ili dvostruko povezana. Na primjer, na jednostruko povezanoj listi na Slici 2 umetanje elementa između elemenata B i C bilo bi obavljeno stvaranjem novog elementa čiji bi pokazivač bio usmjeren na element C (čija bi adresa bila dohvaćena iz vrijednosti pokazivača elementa B), a pokazivač elementa B bi se potom usmjerio na novi element.

Nedostatak povezanih lista je to što zbog korištenja odvojenih blokova memorije pristupanje *n*-tom elementu više ne zahtijeva konstantno vrijeme, već linearno raste s veličinom liste, kao kod linearnog pretraživanja polja fiksne duljine.

## 1.3. Binarna stabla

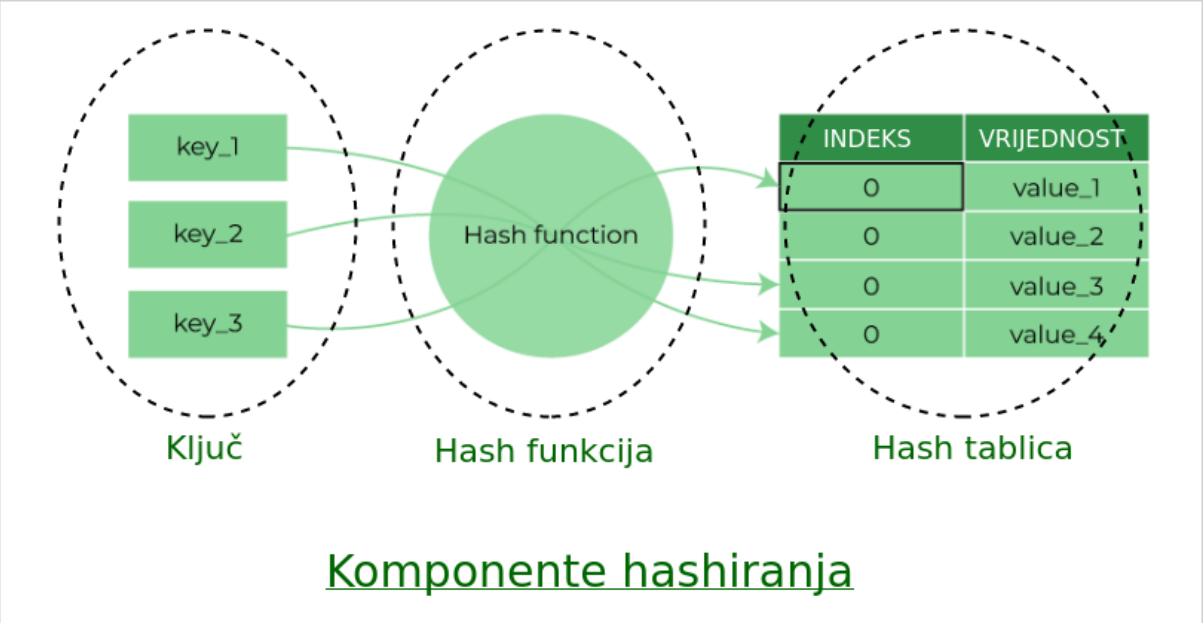



Slika 3. Binarno stablo. Izvor: Prilagođeno iz [3]

Binarno stablo je struktura podataka koja uz svaki element pohranjuje pokazivače na druga dva elementa, kao i kod dvostruko povezane liste, no razlikuje se po tomu što elementi nisu poredani u listu gdje svaki element ima dva susjeda, već se koristi za dijeljenje skupa elemenata na one koji su veći, odnosno manji, od onoga na trenutnom čvoru. Kao i kod povezanih lista, pokazivači koji ne pokazuju na drugi element imaju `NULL` vrijednost. Čvor koji ima dva pokazivača `NULL` vrijednosti naziva se list (engl. *leaf node*). To omogućava lakše umetanje novih elemenata, no i lakše pretraživanje (koje je opet potrebno za pronalaženje pozicije za umetanje elementa). Performanse pretraživanja stabla ovise o tomu koliko je balansirano, odnosno kolika je maksimalna razlika između visine podstabala na svakom čvoru.

Na primjer, na stablu na Slici 3 maksimalna razlika visina je 2 jer je lijevo podstablo čvora vrijednosti 10 visine nula, a desno podstablo visine 2. Da bi se stablo smatralo balansiranim maksimalna razlika visina mora biti 1. Kod balansiranog binarnog stabla vrijeme pretraživanja raste logaritamski jer se kod svakog grananja broj elemenata smanjuje na pola.

## 1.4. Hash tablice




Slika 4. Hash tablica. Izvor: Prilagođeno iz [4]

Ono što hash tablice omogućavaju, za razliku od dosad navedenih struktura, je pristupanje bilo kojem elementu putem ključa, njegovog jedinstvenog identifikatora u konstantnom vremenu, bez ograničenja polja fiksne duljine. Efektivno, to znači pristupanje svakom elementu kao putem indeksa u polju, gdje ključ za pristupanje može biti proizvoljna vrijednost.

To je ostvareno korištenjem hash funkcije koja prevodi ključ proizvoljne vrijednosti proizvoljne duljine u vrijednost iz unaprijed definiranog raspona koja se zatim koristi za indeksiranje elementa polja fiksne duljine. Maksimalan broj elemenata, za razliku od običnog polja, nije ograničen veličinom ovog polja jer se u svakom od indeksa može pohraniti više vrijednosti. To je nužno zbog svojstava hash funkcije, odnosno činjenice da je moguće dobiti jednak indeks u polju iz dva različita ključa. Zbog toga je potrebna mogućnost pohranjivanja više od jednog elementa na istom indeksu. Jedan od načina za to je korištenje povezane liste [5].

Zbog toga nije ispravno reći da je vremenska kompleksnost hash tablica uvijek O(1), već samo u prosječnom slučaju, dok je ona O($n$) u najgorem slučaju kada se svi elementi moraju pohraniti u povezanu listu na istom indeksu polja, odnosno kada ključevi svih elemenata dijele istu hash vrijednost za što je u praksi vjerojatnost iznimno malena ako se se ne koristi polje vrlo male veličine i hash funkcija loše distribucije.

Hash tablice su važna struktura podataka koja se koristi u brojne svrhe, kao na primjer u sustavima baza podataka, usmjeravanju paketa u mrežnim sustavima, pohrani podataka za autentifikaciju, pohrani metapodataka u datotečnim sustavima ili raznim sustavima za *caching* često korištenih podataka, na primjer kod web preglednika ili tražilica.

# 2. Implementacija

U nastavku ovog rada bit će opisana implementacija hash tablice u programskom jeziku C. Ona će uključivati sve osnovne funkcionalnosti strukture podataka i omogućiti pohranu podataka proizvoljne veličine, koristeći dinamičku alokaciju memorije umjesto polja predodređenih dimenzija.

Osnovne funkcionalnosti uključuju inicijalizaciju i brisanje hash tablice, pretraživanje, odnosno dohvaćanje vrijednosti putem ključa, te postavljanje i brisanje parova ključa i vrijednosti. One će biti dostupne putem funkcija kao što su:

```
ht = ht_create(TABLE_SIZE);
ht_set(ht, key, key_size, value);
value = ht_get(ht, key, key_size);
ht_remove(ht, key, key_size, FREE_VALUE);
ht_destroy(ht);
```

U *header* datoteci `hashtable.h` deklarirane su strukture podataka i funkcije koje će se koristiti u implementaciji.

## 2.1. Strukture podataka

U datoteci `hashtable.h` definirane su dvije osnovne strukture korištene u projektu. Prva je `struct entry_t` koja će predstavljati svaki od elemenata u hash tablici i sadržavati sve podatke o njemu kao što su ključ i vrijednost.

```
struct entry_t
{
    uint8_t *key;
    size_t key_size;
    void *value;
    struct entry_t *next;
};
```

Prva varijabla ove strukture je `key`, pokazivač na ključ elementa. Korišteni tip je pokazivač na 8-bitnu cjelobrojnu vrijednost bez predznaka jer ona predstavlja jedan bajt, a pošto ključ treba biti proizvoljne duljine, koristi se dinamička alokacija memorije umjesto unaprijed alociranog polja.

U varijabli key_size pohranjena je duljina alocirane memorije za ključ u prethodnoj varijabli. Korišteni tip je `size_t`, kao i za sve ostale varijable u projektu koje predstavljaju količinu alocirane memorije, jer standard garantira da je on dovoljno velik za pohranu veličine bilo kojeg objekta u programu.

Varijabla `value` je pokazivač na vrijednost pohranjenu u elementu tablice. Pošto ova implementacija ne ograničava tip podatka koji se može pohraniti u svakom od elemenata, korišteni tip je `void` pokazivač, odnosno pokazivač bez unaprijed zadanog tipa ciljne varijable. To znači da se bilo kojem tipu varijable može pristupiti ovim pokazivačem, ali

samo nakon *type casting*-a na ispravan tip varijable koja je bila definirana jer u suprotnom prema C standardu dolazi do nedefiniranog ponašanja programa. Moguće je, na primjer, i stvoriti hash tablicu čiji su elementi hash tablice, odnosno tablicu čijim bi se konačnim elementima ili elementima koji nisu hash tablice, pristupalo višestrukim indeksiranjem. Takva bi tablica na neki način bila analogna dvodimenzionalnim (ili višedimenzionalnim) poljima.

Ovo može izgledati problematično ako se tablica koristi za pohranu različitih vrsta podataka ili podataka različite duljine, tako da jedno od rješenja može biti korištenje strukture koja će uz varijablu sa samim podatcima sadržavati i varijablu koja sadrži veličinu tih podataka ili varijablu koja opisuje tip varijable s podatcim prema nekom dogovorenom kodiranju tipova u programu.

```
struct value_size
{
    size_t size;
    void *value;
};
```

Na primjer, struktura `value_size` može sadržavati varijablu `size` koja predstavlja veličinu podataka u bajtovima.

```
struct value_type
{
    int type;
    void *value;
};
```

Ako će program koristiti samo ograničeni broj unaprijed definiranih tipova podataka fiksne veličine koji se ne mogu koristiti kao polje elemenata veličine jednog bajta, može se definirati struktura poput `value_type` koja u varijabli `type` sadrži kodirani podatak o vrsti varijable koji će se moći koristiti za pristupanje tom podatku putem pokazivača.

Varijabla `next` je pokazivač na varijablu istog tipa kao što je struktura `entry_t`, odnosno to je varijabla koja niz elemenata na svakom indeksu tablice čini povezanom listom. Nakon što se iz dobivenog ključa odredi indeks u polju za pristupanje elementu, članovima ove povezane liste funkcije će prolaziti dok ne pronađu onaj koji zapravo sadrži identičan ključ. Zbog toga maksimalno vrijeme pristupanja raste linearno, a ne ostaje konstantno, pa će radi optimizacije cilj biti minimizirati broj elemenata u svakoj od ovih povezanih lista. Korištena lista je jednostruko povezana jer je obilaženje u jednom smjeru dovoljno i ne bi bilo nikakve dodatne koristi od korištenja dvostruko povezane liste.

Druga korištena struktura je `struct ht_t` i ona predstavlja cijelu hash tablicu.

```
struct ht_t
{
    struct entry_t **entries;
    size_t size;
    uint32_t (*hashfunction)(const uint8_t * const, const size_t);
};
```

Struktura sadrži varijablu `entries`, polje pokazivača, odnosno pokazivač na dinamički alocirane pokazivače pošto veličina polja nije konstanta. Ovi pokazivači pokazivat će na već opisani tip `struct entry_t` koji predstavlja jedan element u tablici. Dakle, svaki element alociranog polja sadrži adresu jedne povezane liste s elementima, ili `NULL` pokazivač ako je prazna.

Varijabla `size` sadrži veličinu dinamički alociranog polja entries i bit će korištena kod određivanja raspona indeksa iz dobivene hash vrijednosti. `hashfunction` je pokazivač na fukciju koja je korištena za dobivanje hash vrijednosti u odgovarajućoj tablici. Ta funkcija mora vraćati 32-bitnu cjelobrojnu vrijednost i primati kao argumente pokazivač na prvi bajt ključa te veličinu ključa u bajtovima.

## 2.2. Funkcije

Prva funkcija je funkcija `ht_create`.

```
struct ht_t *ht_create(size_t size);
```

Ona služi za stvaranje i inicijalizaciju hash tablice. Tablica je pohranjena putem pokazivača na strukturu `ht_t` kojeg ova funkcija vraća. Parametar joj je veličina polja za pohranu elemenata i on linearno utječe na veličinu prazne tablice u memoriji, još prije unosa podataka.

Funkcija za brisanje tablice je `ht_destroy`.

```
void ht_destroy(struct ht_t *ht, bool free_values);
```

Njezini argumenti su pokazivač na tablicu koji vraća funkcija `ht_create` i varijabla `free_values` tipa `bool` koja određuje hoće li funkcija dealocirati i vrijednosti elemenata na koje pokazuju pokazivači `value` ili samo ključeve i elemente tablice. Ona nema povratnu vrijednost.

Funkcija za unos para ključa i vrijednosti je `ht_set`.

```
bool ht_set(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size, void *value);
```

Ona također prima pokazivač na hash tablicu, ključ i veličinu ključa te pokazivač za pohranu u tablicu. Vraća `bool` vrijednost koja je `true` osim ako je došlo do problema pri alokaciji memorije za pohranu novog elementa ili ako je dobiveni pokazivač `NULL`.

Funkcija za pristup vrijednosti putem ključa je `ht_get`.

```
void *ht_get(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size);
```

Njezini parametri su isti kao kod funkcije `ht_set`, osim pokazivača na vrijednost, jer dobiveni ključ koristi za pristupanje vrijednosti, a ne za unos novog para. Vraća pokazivač na traženu vrijednost ili `NULL` pokazivač ako ona nije pronađena.

Funkcija `ht_remove` služi za uklanjanje para ključa i vrijednosti iz hash tablice.

```
bool ht_remove(struct ht_t *ht, const uint8_t * const key,
            const size_t key_size, bool free_value);
```

Prima iste parametre kao i funkcija `ht_get`, uz dodatni parametar `free_value` tipa `bool` koji određuje hoće li dealocirati vrijednosti uklonjenog elementa, slično kao i kod funkcije `ht_destroy`.

Tablici se može promijeniti veličina korištenog polja pokazivača pomoću funkcije `ht_resize`.

```
bool ht_resize(struct ht_t *ht, size_t new_size);
```

Njezini argumenti su pokazivač na hash tablicu i nova veličina za alokaciju polja. Ako funkcija uspješno promijeni veličinu vraća vrijednost `true`, u suprotnom vraća vrijednost `false`.

Dvije pomoćne funkcije koje su korištene primarno za otklanjanje pogrešaka, a ne za korištenje hash tablica su `ht_print_keys` i `ht_print_distribution`.

```
void ht_print_keys(const struct ht_t * const ht);

void ht_print_distribution(const struct ht_t * const ht);
```

Jedini parameter im je objema pokazivač na hash tablicu, a služe za ispis svih ključeva trenutno u tablici te ispis broja indeksa u polju prema odgovarajućem broju elemenata u povezanoj listi, odnosno za prikaz koliko dobro hash funkcija distribuira elemente u polju.

# 3. Opis funkcija

U datoteci `hashtable.c` definirane su sve funkcije potrebne za korištenje hash tablica.

## 3.1. Funkcija za stvaranje hash tablice

Kao što je već rečeno, za stvaranje hash tablice koristi se funkcija `ht_create`.

Prvi korak u stvaranju tablice je alokacija memorije za jednu strukturu tablice.

```
struct ht_t *ht = calloc(1, sizeof *ht);
if (ht == NULL)
{
    return NULL;
}
```

Alocira se funkcijom `calloc` kako bi svi bajtovi u alociranoj memoriji bili inicijalizirani na vrijednost nule. Argumenti te funkcije su broj članova za alokaciju te veličina svakog člana u bajtovima. U ovom slučaju potreban je jedan element veličine strukture hash tablice koja se može dobiti pomoću izraza `sizeof *ht`, odnosno operatorom `sizeof` i dereferenciranjem deklariranog pokazivača. Nakon alokacije memorije potrebno je provjeriti je li ona bila uspješna, odnosno je li funkcija vratila pokazivač koji nije vrijednosti `NULL`.

Nakon toga postavlja se veličina polja u strukturi tablice.

```
if (size == 0)
{
    ht->size = HT_SIZE;
}
else
{
    ht->size = size;
}
```

Veličina polja postavlja se na vrijednost argumenta funkcije, ili na unaprijed zadanu *macro* vrijednost `HT_SIZE`. U ovom slučaju ona je postavljena na 524243, najveći prosti broj manji od 524288, odnosno $2^{19}$.

### 3.1.1 Veličina polja hash tablice

Kod određivanja veličine polja koje će se koristiti za distribuciju elemenata hash tablice jedno od važnih obilježja je da veličina polja korištenog za pohranu elemenata bude prost broj. Razlog tomu je to da kod korištenja operacije modulo koja se često koristi kod određivanja indeksa iz danog ključa, svi ključevi čija je vrijednost višekratnik broja koji je djelitelj veličine polja bit će smješteni u indekse koji su također višekratnici tog broja [6].

Ako je odabir korištenih ključevi uistinu slučajan, veličina polja neće biti od značajne važnosti, no kod uporabe strukture podataka ključevi često nisu sasvim slučajni i ako zbog

nesavršenosti korištene hash funkcije dobivene hash vrijednosti nisu uniformne distribucije, moguće je velik broj ključeva bude smješten u ograničeni podskup indeksa. Na primjer, ako je veličina polja 12 i za dobivanje hash vrijednosti se koristi samo operacija modulo za dobivanje broja u rasponu od 0 do 11, a ključevi koji se koriste su višekratnici broja 3 koji je također i djelitelj veličine polja, dogodit će se sljedeće [7]:

- Ključevi {0, 12, 24, 36, ... } bit će smješteni u indeks 0
- Ključevi {3, 15, 27, 39, ... } bit će smješteni u indeks 3
- Ključevi {6, 18, 30, 42, ... } bit će smješteni u indeks 6
- Ključevi {9, 21, 33, 45, ... } bit će smješteni u indeks 9

Korištenje polja čija je veličina prost broj nema takav nedostatak jer bi tada korišteni indeksi trebali biti višekratnici broja jedan, odnosno cijeli raspon indeksa, ili višekratnici same veličine tablice za što je vjerojatnost dovoljno malena, pogotovo kod tablica velikih dimenzija, kao na primjer u ovom slučaju vrijednosti `HT_SIZE`.

Nakon postavljanja veličine polja, alocira se memorija za to polje funkcijom `calloc` kako bi svi bajtovi bili postavljeni na vrijednost nule, odnosno kako bi vrijednosti svih pokazivača u polju bile `NULL`.

```
ht->entries = calloc(ht->size, sizeof *ht->entries);
if (ht->entries == NULL)
{
    free(ht);
    return NULL;
}
```

Nakon alokacije provjerava se je li bila uspješna te se oslobađa alocirana memorija i izlazi iz funkcije s `NULL` pokazivačem ako je bila neuspješna.

Na kraju se postavlja i vrijednost pokazivača za hash funkciju te se izlazi iz funkcije vraćajući vrijednost pokazivača stvorene hash tablice.

```
ht->hashfunction = &testhash;

return ht;
```

## 3.2. Funkcija za uništavanje hash tablice

Funkcija za brisanje hash tablice iz memorije je `ht_destroy`. Prema vrijednosti argumenta `free_values`, najprije određuje hoće li dealocirati i memoriju vrijednosti pohranjenu u svakom paru ključ-vrijednost, ili samo memoriju za pohranu elemenata tablice, odnosno polje pokazivača na povezane liste s elementima i sve elemente tih lista.

Razlog iza takve odluke bio je da hash tablica može biti korištena za uređivanje i pristup već postojećim podatcima u programu te da brisanje tih podataka nije dužnost hash tablice. U tom slučaju izvođenje ovog koda moglo bi dovesti i do nedefiniranog ponašanja programa prema C standardu ako memorija vrijednosti elemenata nije dinamički alocirana, već se nalazi na stogu pošto funkcija `free` služi samo za oslobađanje memorije alocirane na gomili.

Prvi korak brisanja tablice je brisanje svih povezanih lista funkcijom `free_entries`.

```
free_entries(ht->entries, ht->size, free_values);
```

Brisanje ovih elemenata izdvojeno je u zasebnu funkciju jer će brisanje svih povezanih lista biti korišteno i kod promjene veličine hash tablice u funkciji `ht_resize`. Funkcija je statička jer je namijenjena samo korištenju unutar ove datoteke i kao argumente prima polje `entries`, veličinu tog polja i vrijednost `free_values` koja označava hoće li dealocirati i memoriju u kojoj su smješteni podatci svakog elementa. Koristi dva pokazivača na elemente, `p` i `p_next` za iteraciju kroz elemenente svih povezanih lista.

```
static void free_entries(struct entry_t **entries, const size_t size,
                         bool free_values)
{
    struct entry_t *p;
    struct entry_t *p_next;

    for (size_t i = 0; i < size; ++i)
    {
        p = entries[i];

        while (p != NULL)
        {
            p_next = p->next;

            if (free_values)
            {
                free(p->value);
            }
            free(p->key);
            free(p);

            p = p_next;
        }
    }
}
```

Dok glavna petlja prolazi kroz sve indekse polja tablice, druga petlja prolazi kroz elemente povezene liste liste oslobađajući memoriju strukture elementa, ključa (koja je također bila dinamički alocirana) i opcionalno memoriju vrijednosti. Dva pokazivača korištena su kako bi drugi sačuvao adresu idućeg elementa liste, dok se prvi koristi za brisanje trenutnog elementa.

Natrag u funkciji `ht_destroy` na kraju se još oslobađa i preostala memorija polja s pokazivačima te memorija same strukture.

```
free(ht->entries);
free(ht);
```

## 3.3. Funkcija za unos vrijednosti u hash tablicu

Funkcija `ht_set` služi za unos vrijednosti u hash tablicu. Na početku funkcije provjerava se je li vrijednost pokazivača na podatke jednaka `NULL`.

```
if (value == NULL)
{
    return false;
}
```

U ovoj implementaciji `NULL` vrijednost pokazivača na podatke nije dozvoljena jer se povratna vrijednost `NULL` u funkciji `ht_get` smatra nedostatkom traženog ključa u hash tablici.

Zatim se izračunava indeks za pohranu novog elementa u polje. Za to se koristi hash funkcija na koju je usmjeren pokazivač u strukturi hash tablice i funkcija `get_index` koja iz dobivene hash vrijednosti izračuna indeks za pristup u polju.

```
uint32_t key_hash = (*ht->hashfunction)(key, key_size);
size_t index = get_index(ht, key_hash);
```

Indeks se iz hash vrijednosti najčešće dobiva pomoću modula i veličine polja za pohranu, no dovoljna je bilo koje funkcija koja bilo kojoj hash vrijednosti deterministički pridružuje jedan indeks u polju tablice. Dani primjer `get_index` funkcije koristi operaciju modula.

```
static size_t get_index(const struct ht_t * const ht, const uint32_t key_hash)
{
    size_t index = key_hash % ht->size;
    return index;
}
```

U dvije nove varijable se potom pohranjuje veličina ključa u bajtovima za alokaciju potrebne memorije i kopiranje vrijednosti te pokazivač na element u povezanoj listi na indeksu `index` u polju koji ima istu vrijednost ključa kao i novi ključ za unos.

```
size_t key_mem_size = key_size * sizeof *key;

struct entry_t *match = find_entry(ht->entries[index], key, key_size);
```

Drugim rječima, nakon što je izračunat indeks koji odgovara novom ključu potrebno je provjeriti treba li alocirati novu memoriju za unos novog elementa u povezanu listu ili element toga ključa već postoji te mu samo treba ažurirati vrijednost. Za to se koristi funkcija `find_entry` koja prolazi kroz povezanu listu i traženi ključ uspoređuje sa svakim ključem u toj listi.

```
static struct entry_t *find_entry(struct entry_t * const entry_row,
                const uint8_t * const key, const size_t key_size)
{
    struct entry_t *match = entry_row;

    while (match != NULL)
    {
```

```
            if (key_size == match->key_size &&
                memcmp(match->key, key, key_size) == 0)
            {
                return match;
            }
            match = match->next;
        }

        return NULL;
    }
```

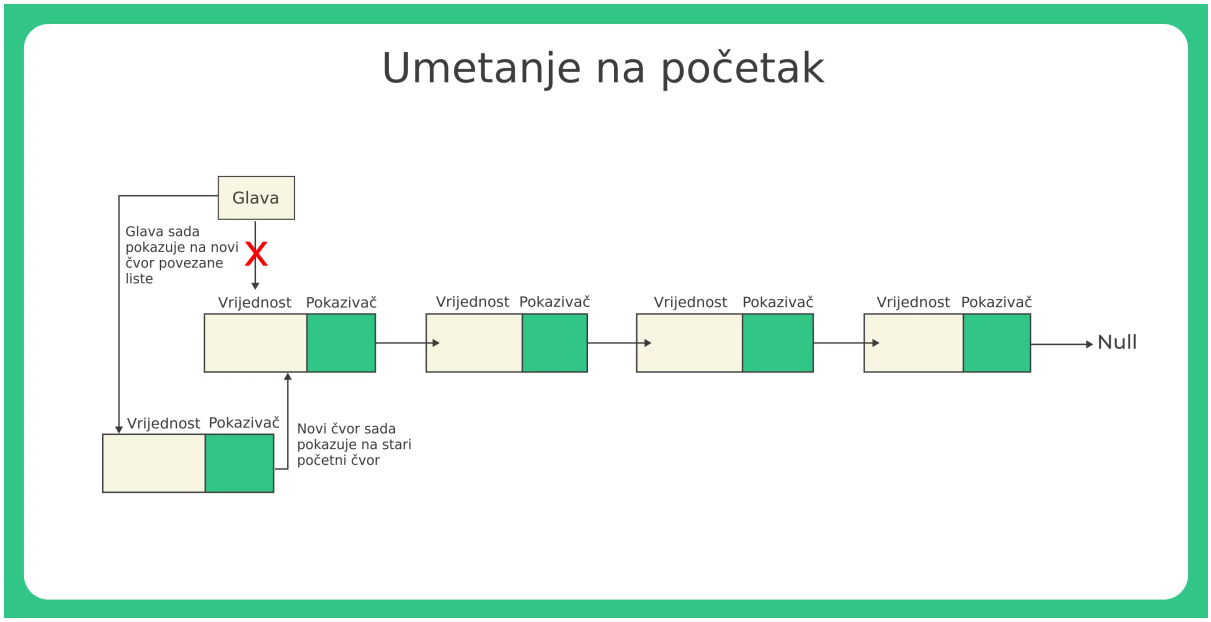
Ova funkcija definirana je kao statička funkcija jer nije namijenjena za uporabu izvan njezine datoteke, već samo unutar drugih funkcija koje su ovdje definirane. Služi za pretraživanje povezane liste prema danom ključu. Za to koristi petlju koja u svakoj iteraciji provjerava odgovara li vrijednost ključa ili je stigla do zadnjeg elementa kada vraća `NULL`. Kod usporedbe ključeva potrebno je prvo provjeriti jesu li jednake duljine kako kod usporedbe memorije funkcijom `memcmp` ne bi došlo do nedefiniranog ponašanja u slučaju pristupa memoriji za koju nema dozvolu. Ako je element pronađen funkcija vraća njegovu adresu koja će se potom koristiti za ažuriranje vrijednosti.

Vrijednost dobivenog pokazivača u funkciji `ht_set` se zatim provjerava i ako je `NULL`, potrebno je alocirati novi element u povezanoj listi.

```
        if (match == NULL)
        {
            struct entry_t *first = ht->entries[index];

            ht->entries[index] = calloc(1, sizeof *ht->entries[index]);
            if (ht->entries[index] == NULL)
            {
                ht->entries[index] = first;
                return false;
            }

            ht->entries[index]->key_size = key_size;
            ht->entries[index]->key = malloc(key_mem_size);
            if (ht->entries[index]->key == NULL)
            {
                free(ht->entries[index]);
                ht->entries[index] = first;
                return false;
            }

            memcpy(ht->entries[index]->key, key, key_mem_size);
            ht->entries[index]->value = value;
            ht->entries[index]->next = first;
        }
```

To se izvodi dodavanjem na početak liste jer je to najjednostavniji način pošto je za njega dovoljan pristup prvom elementu liste preko već dostupne adrese u polju `ht->entries`. Na početak liste postavlja se novi element alociran funkcijom `calloc` i zatim se postavljaju njegove nove vrijednosti (uz dodatnu alokaciju memorije za ključ). Na kraju se pokazivač

`next` na idući element liste usmjeruje na element koji je dosada bio prvi, a čija je adresa u međuvremenu bila pohranjena u pokazivaču `first`. Proces dodavanja elementa na početak liste prikazan je na Slici 5.




Slika 5. Dodavanje elementa na početak povezane liste. Izvor: Prilagođeno iz [8]

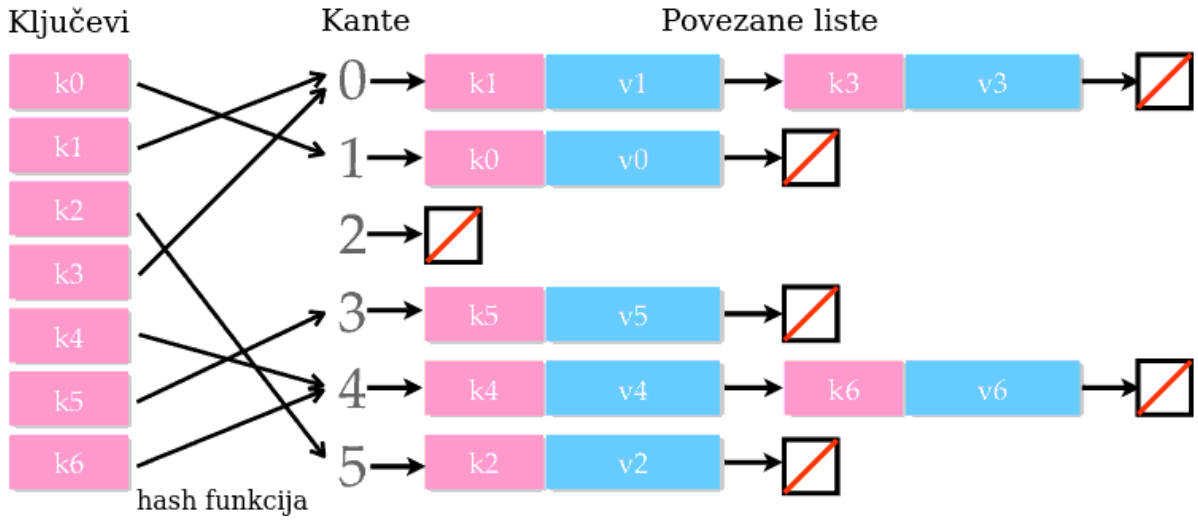
U slučaju da je element istog ključa pronađen u funkciji `find_entry`, postupak unošenja vrijednosti bit će nešto jednostavniji.

```
else
{
    if (key_size != match->key_size)
    {
        match->key_size = key_size;
        free(match->key);
        match->key = malloc(key_mem_size);
        if (match->key == NULL)
        {
            return false;
        }
    }

    memcpy(match->key, key, key_mem_size);
    match->value = value;
}

return true;
```

Tada je potrebno samo ažurirati pokazivač na vrijednost, te dealocirati i ponovno alocirati memoriju za ključ i ažurirati veličinu ključa prije kopiranja ako se ona promijenila. Ako je novi ključ iste veličine kao i stari, izvodi se samo kopiranje vrijednosti funkcijom `memcpy`.

Na kraju funkcije vraća se vrijednost `true` koja označava da se ažuriranje vrijednosti u tablici uspješno izvršilo. Kod svake alokacije nove memorije koja je potrebna provjerava se je li bila uspješna. Ako nije, dealocira se sva memorija koja je dotada uspješno alocirana u funkciji jer neće biti korištena i vraća se vrijednost `false` koja označava neuspješno izvođenje funkcije.

Na Slici 6 je prikazana vizualizacija stukture hash tablice u memoriji koja odgovara onoj korištenoj u ovom projektu. Indeksi u stupcu *Buckets* odgovaraju polju pokazivača entries u strukturi `ht_t`, dok su elementi povezane liste elementi tipa `entry_t`. Proces hashiranja ključeva predstavlja jednoznačno pridruživanje indeksa ključevima iz stupca *Keys* predstavljeno strelicama. Broj elemenata u svakoj povezanoj listi mora odgovarati broju ključeva čija hash vrijednost pokazuje na indeks te liste, kao što je vidljivo u ovom primjeru.




Slika 6. Vizualizacija strukture hash tablice. Izvor: Prilagođeno iz [9]

## 3.4. Funkcija za pristup elementu hash tablice

Funkcija za pristup elementima hash tablice vrlo je slična funkciji za unos elemenata. Odnosno, na isti način se određuje indeks za pristup polju i pretraživanje povezane liste putem indeksa.

```
uint32_t key_hash = (*ht->hashfunction)(key, key_size);
size_t index = get_index(ht, key_hash);

const struct entry_t *match = find_entry(ht->entries[index],
        key, key_size);

if (match == NULL)
{
    return NULL;
}

return match->value;
```

Jedina razlika je u tomu što je nakon određivanja pokazivača funkcijom `find_entry` potrebno samo provjeriti je li mu vrijednost `NULL` i tada vratiti `NULL`. U suprotnom, vraća se vrijednost pokazivača na vrijednost elementa, `match->value`.

Kao što je prije navedeno, ograničenje vrijednosti tog pokazivača u ovoj implementaciji je da ne smije biti `NULL`. U suprotnom ne bi bilo moguće odrediti nedostaje li traženi ključ u tablici ili postoji, no vrijednost njegovog `value` pokazivača je `NULL`.

## 3.5. Funkcija za uklanjanje vrijednosti iz hash tablice

Funkcija `ht_remove` služi za uklanjanje elementa hash tablice prema odgovarajućem ključu. Indeks za pristup elementu u polju dobiva se na isti način kao i u prethodne dvije funkcije.

```
uint32_t key_hash = (*ht->hashfunction)(key, key_size);
size_t index = get_index(ht, key_hash);
```

Potom se definiraju dva pokazivača na elemente povezane liste za pretraživanje.

```
struct entry_t *match = ht->entries[index];
struct entry_t *prev = NULL;
```

Prvi će se koristiti za traženje odgovarajućeg elementa za brisanje, a drugi će biti usmjeren na prethodni element kako bi se njegov pokazivač ispravno ažurirao nakon brisanja. Program zatim ulazi u petlju u kojoj uspoređuje elemente liste s odgovarajućim ključem.

```
while (match != NULL)
{
    if (key_size == match->key_size && memcmp(match->key, key, key_size) == 0)
    {
        break;
    }
    prev = match;
    match = match->next;
}
```

Iz petlje izlazi ako dođe do kraja liste ili nađe traženi ključ. Ključevi se uspoređuju na način opisan u funkciji `find_entry`. Nakon toga provjerava se razlog izlaska iz petlje. Ako je vrijednost pokazivača `match` jednaka `NULL`, brisanje elementa nije uspjelo i funkcija vraća vrijednost `false`. U suprotnom provjerava se je li vrijednost pokazivača `prev` jednaka `NULL`. To će se dogoditi samo ako program izađe iz petlje u prvoj iteraciji ili ju uopće ne počne izvoditi. Pošto će petlju preskočiti samo kada je početna vrijednost pokazivača `match` jednaka `NULL`, u kojem slučaju će već izaći iz funkcije u prijašnjem grananju, u ovom koraku može se sa sigurnošću reći da vrijednost `NULL` pokazivača `prev` znači da je prvi element povezane liste traženi element.

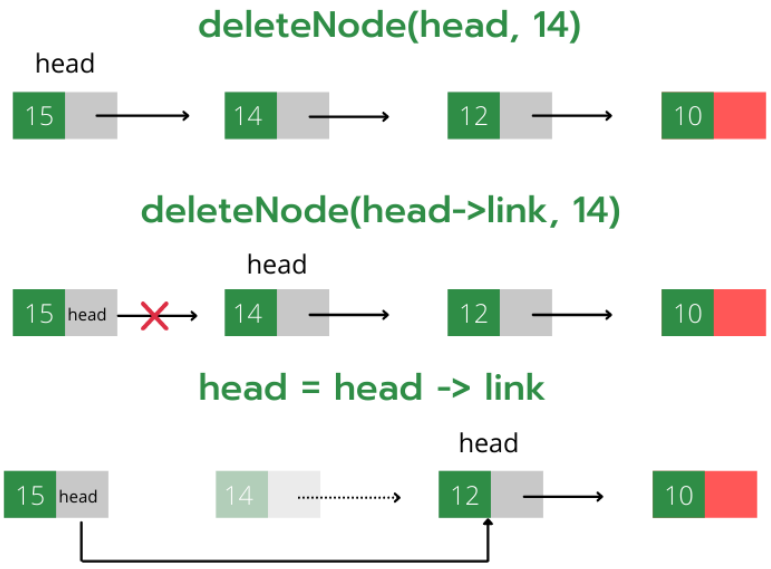
```
if (match == NULL)
{
    return false;
}

if (prev == NULL)
{
    ht->entries[index] = match->next;
}
else
{
    prev->next = match->next;
}
```

U tom slučaju početni pokazivač liste u polju tablice usmjeruje se na idući element liste. Ako je element za brisanje jedini element, njegova `next` vrijednost će ionako biti `NULL`, tako da će ispravna vrijednost svejedno biti unesena u polje. Ako se element ne nalazi na početku liste, koristi se pokazivač `prev` za ažuriranje pokazivača prethodnog elementa.

Nakon uređivanja pokazivača potrebno je još samo osloboditi memoriju brisanog elementa, uključujući i opcionalnu dealokaciju memorije vrijednosti elementa, i vraćanje vrijednosti `true` nakon uspješnog brisanja.

```
if (free_value)
{
    free(match->value);
}
free(match->key);
free(match);

return true;
```

Proces brisanja elementa iz povezane liste ilustriran je na Slici 7.




Slika 7. Proces brisanja elementa povezane liste. Izvor: [10]

## 3.6. Funkcija za promjenu veličine hash tablice

Funkcija `ht_resize` služi za promjenu veličine polja za distribuciju elemenata u povezane liste. To obavlja stvaranjem novog polja i novim unosom svih elemenata iz starog, pošto je potrebno odrediti novi indeks za svaki.

Na početku funkcije provjerava se je li tražena veličina jednaka trenutnoj. Ako jest, funkcija vraća vrijednost `true` i operacija se smatra uspješnom.

```
if (new_size == ht->size)
{
    return true;
}
```

U suprotnom potrebno je alocirati memoriju za novo polje veličine zadane parametrom `new_size`, uz provjeru je li alokacija bila uspješna.

```
struct entry_t **new_entries = calloc(new_size, sizeof *new_entries);
if (new_entries == NULL)
{
    return false;
}
```

Nakon toga, definiraju se varijable za privremenu pohranu trenutnih vrijednosti pokazivača hash tablice i veličine njezinog polja kako bi se one mogle ažurirati novim vrijednostima.

```
struct entry_t **old_entries = ht->entries;
size_t old_size = ht->size;

ht->entries = new_entries;
ht->size = new_size;
```

Vrijednosti u strukturi hash tablice se ažuriraju prije nego što se unesu svi ključevi i vrijednosti iz starog polja. Razlog tomu je mogućnost korištenja već gotove funkcije `ht_set` za unos vrijednosti u tablicu kojoj je kao argument potreban pokazivač na hash tablicu, a ne samo polje s vrijednostima. Ovako će biti potrebno ponovno ažurirati vrijednosti tablice na one stare veličine ako vrijednosti u novoj tablici ne budu uspješno postavljene, no sveukupni kôd funkcije će svejedeno biti jednostavniji.

U petlji za unos vrijednosti u novo polje koristit će se dvije nove varijable, pomoćni pokazivač `p` za iteraciju kroz povezane liste i varijabla `set_success` tipa `bool` za praćenje neuspjelih pokušaja unosa vrijednosti.

```
struct entry_t *p;
bool set_success = true;
```

Glavna petlja prolazi kroz sve pokazivače u polju `old_entries`, dok posebna petlja u svakoj iteraciji glavne prolazi kroz sve elemente povezanih lista. Podatci svakog elementa unose se u novo polje tablice `ht` pomoću gotove funkcije `ht_set`. Povratna vrijednost koristi se za praćenje eventualne greške kroz varijablu `set_success`. Ako do toga dođe, dolazi do izravnog izlaska iz obje petlje.

```
for (size_t i = 0; i < old_size; ++i)
{
    p = old_entries[i];

    while (p != NULL)
    {
        set_success = ht_set(ht, p->key, p->key_size, p->value);
        if (!set_success)
        {
            break;
        }
        p = p->next;
    }
    if (!set_success)
    {
        break;
    }
}
```

Nakon izlaska iz petlje vrijednost varijable `set_success` omogućit će provjeru jesu li svi elementi uspješno uneseni. Ako nisu, to će značiti da je došlo do neuspjele alokacije memorije za neki od elemenata. U tom slučaju u hash tablicu `ht` vraćaju se svi podatci u polju stare veličine kako ne bi došlo do gubitka podataka. Oslobađa se sva memoriju alocirana u funkciji u međuvremenu i vraća se povratna vrijednost `false` zbog neuspjele promjene veličine. Kod oslobađanja memorije povezanih lista, memorija s vrijednostima svakog elementa ne smije biti dealocirana jer će adrese istih memorijskih lokacija biti korištene u tablici bez obzira na to je li veličina polja uspješno promijenjena. Zbog toga je treći argument funkcije `free_entries` vrijednost `false`.

```
if (!set_success)
{
    free_entries(ht->entries, ht->size, false);
    free(ht->entries);
    ht->entries = old_entries;
    ht->size = old_size;

    return false;
}
```

Ako nije došlo do greške u petlji, oslobađa se memorija polja i povezanih lista stare veličine (bez dealokacije memorije vrijednosti elemenata, iz istog razloga kao i kod neuspjele promjene veličine) i funkcija vraća vrijednost `true`.

```
free_entries(old_entries, old_size, false);
free(old_entries);

return true;
```

## 3.7. Pomoćne funkcije

Dvije pomoćne funkcije koje nisu važne za osnovnu funkcionalnost hash tablice, već su napisane radi lakšeg testiranja i otklanjanja pogrešaka su `ht_print_keys` i `ht_print_distribution`. Funkcija `ht_print_keys` služi za ispis ključeva svih elemenata koji su trenutno u tablici. Za to koristi dvije petlje na sličan način kao u već nekoliko opisanih funkcija koje moraju obilaziti liste na svim indeksima polja u tablici.

```
void ht_print_keys(const struct ht_t * const ht)
{
    struct entry_t *p;
    for (size_t i = 0; i < ht->size; ++i)
    {
        p = ht->entries[i];
        while (p != NULL)
        {
            printf("0x");
            for (size_t j = 0; j < p->key_size; ++j)
            {
                printf("%hhx", p->key[j]);
            }
            printf("\n");
            p = p->next;
        }
    }
}
```

Ova funkcija koristi još jednu petlju unutar prethodne dvije koja služi za obilazak svih bajtova ključa. Razlog tomu je to što za testiranje implementacije hash tablice nisu bili korišteni samo ključevi čitljivih *string* vrijednosti, odnosno vrijednosti koje se mogu ispisati koristeći *string* format funkcije `printf`, već ključevi s bajtovima iz raspona svih mogućih vrijednosti (od 0 do 255). Zbog toga se za ispis koristi heksadekadski prikaz bajtova.

Druga pomoćna funkcija je `ht_print_distribution` i služi za ispis distribucije indeksa u polju pokazivača prema duljini povezanih lista na koje oni pokazuju. Za to koristi dinamički alocirano polje `counts` iste veličine kao i polje hash tablice u koje će pohraniti brojnosti povezanih lista prema njihovoj duljini. Na primjer, broj povezanih lista duljine 0 bit će pohranjen u `counts[0]`, duljine 1 u `counts[1]`, itd.

```
void ht_print_distribution(const struct ht_t * const ht)
{
    size_t *counts = calloc(ht->size, sizeof *counts);
    struct entry_t *p;

    for (size_t i = 0; i < ht->size; ++i)
    {
        size_t count = 0;
        p = ht->entries[i];
        while (p != NULL)
        {
```

```
            ++count;
            p = p->next;
        }
        ++counts[count];
    }

    for (size_t i = 0; i < ht->size; ++i)
    {
        if (counts[i] > 0)
        {
            printf("%zu:\t%zu\n", i, counts[i]);
        }
    }

    free(counts);
}
```

Funkcija koristi dvije ugniježđene petlje za obilazak svih elemenata tablice i kod obilaska svake povezane liste broji njezine elemente varijablom `count`. Nakon toga povećava broj indeksa polja s listom dobivene duljine za jedan.

Nakon obilaska svih indeksa ispisuje se broj indeksa prema svim brojnostima elemenata povezane liste osim onih čija je vrijednost nula. U nastavku je prikazan primjer ispisa ove funkcije.

```
0:      24173
1:      12296
2:      2988
3:      484
4:      64
5:      4
```

# 4. Testiranje

Implementacija hash tablica testirana je na 20000 parova ključa i vrijednosti. U datoteci `test.c` nalazi se kod koji učitava vrijednosti iz datoteke `testdata.txt`. Vrijednosti su slučano generirani nizovi bajtova zapisani heksadekadskim zapisom i odvojeni u zasebne retke. Svaki par redaka predstavlja par ključa i vrijednosti i datoteka ima ukupno 40000 redaka. Unutar while petlje parovi se čitaju iz datoteke i provjerava se jesu li ispravne duljine za heksadekadski zapis broja te im se dodaje završni nul bajt kako bi bili ispravne string vrijednosti. Zatim se pretvaraju iz heksadekadskog zapisa u binarne vrijednosti funkcijom `strtol` i dodaju u tablicu veličine polja 30011 funkcijom `ht_set`. Nakon unosa podataka ispisuje se distribucija u tablici. Veličina polja tablice se zatim smanjuje na 10007 i ponovno se ispisuje distribucija. Na kraju se oslobađa sva memorija tablice i vrijednosti elemenata.

Program je bio izgrađen pomoću prevoditelja GCC, verzije 12.2.0. Za automatizaciju izgradnje korišten je alat GNU Make i priložena datoteka `Makefile`.

Kod tablice veličine 30011 iskorištenost polja bila je 48.7%, odnosno toliki udio polja je bio iskorišten za pohranu podataka veličine 66.7% veličine polja. 48.4% unesenih vrijednosti nalazi se u listi duljine veće od jedan. Najdulja lista bila je duljine šest.

```
0:      15383
1:      10315
2:      3394
3:      794
4:      111
5:      13
6:      1
```

Kod tablice veličine 10007 iskorištenost polja bila je 86.9% odnosno toliki udio polja je bio iskorišten za pohranu podataka veličine 199.9% veličine polja. 86.3% unesenih vrijednosti nalazi se u listi duljine veće od jedan. Najdulja lista bila je duljine deset.

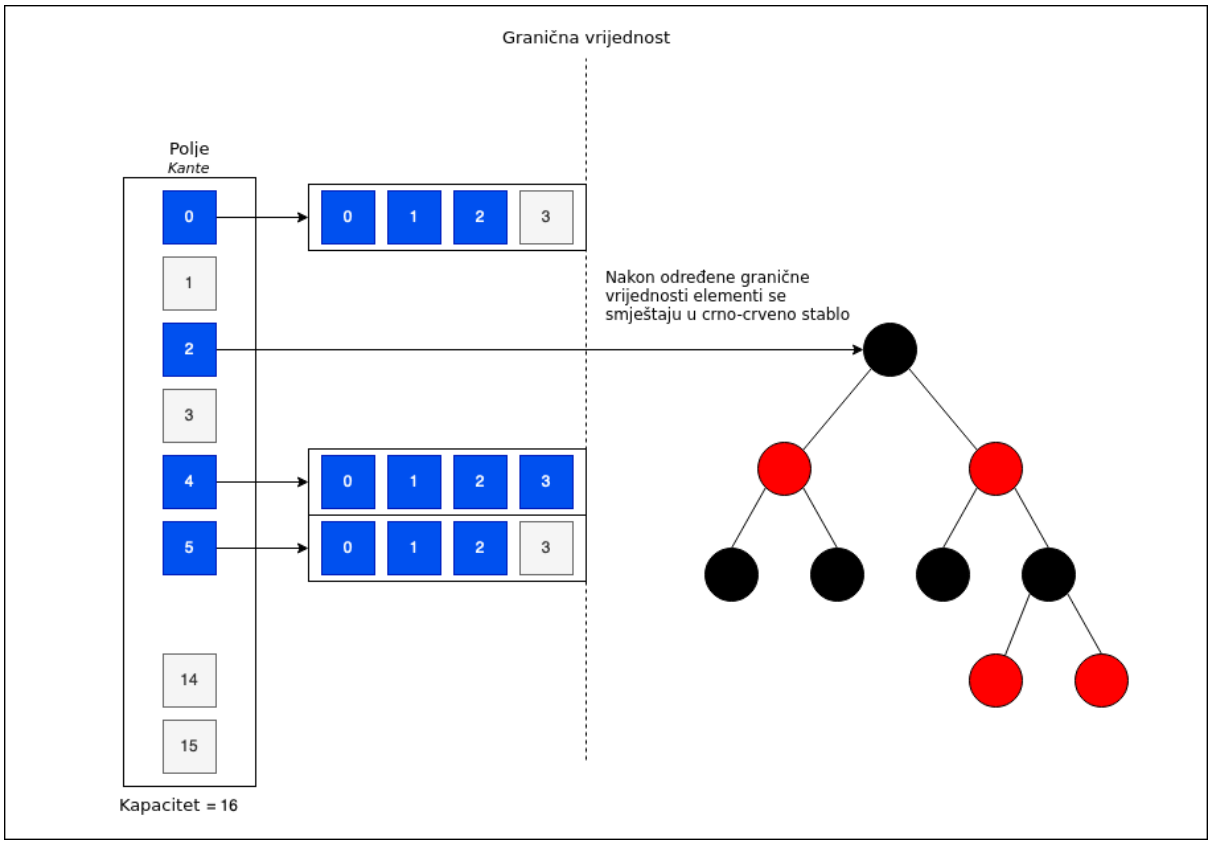
```
0:      1314
1:      2737
2:      2743
3:      1798
4:      919
5:      331
6:      118
7:      35
8:      10
9:      1
10:     1
```

# 5. Alternative

Kao što je već rečeno, u opisanoj implementaciji hash tablica koriste se povezane liste za pohranu elemenata na svakom indeksu polja. Zbog toga vremenska kompleksnost raste linearno s količinom podataka u najgorem slučaju kod unosa, brisanja i pristupa podatcima. Jedan od načina poboljšanja kompleksnosti u najgorem slučaju je korištenje druge strukture podataka kod dodavanja elemenata.

## 5.1. Binarno stablo

Na primjer, umjesto povezane liste za dodavanje elemenata u svaki od indeksa polja može se koristiti struktura balansiranog binarnog stabla. U tom slučaju vremenska kompleksnost će u najgorem slučaju rasti logaritamski, uz nešto dodatne kompleksnosti kod dodavanja i uklanjanja elemenata. Primjer takve strukture koji koristi strukturu crno-crvenog stabla (engl. *red-black tree*) prikazan je na Slici 8.



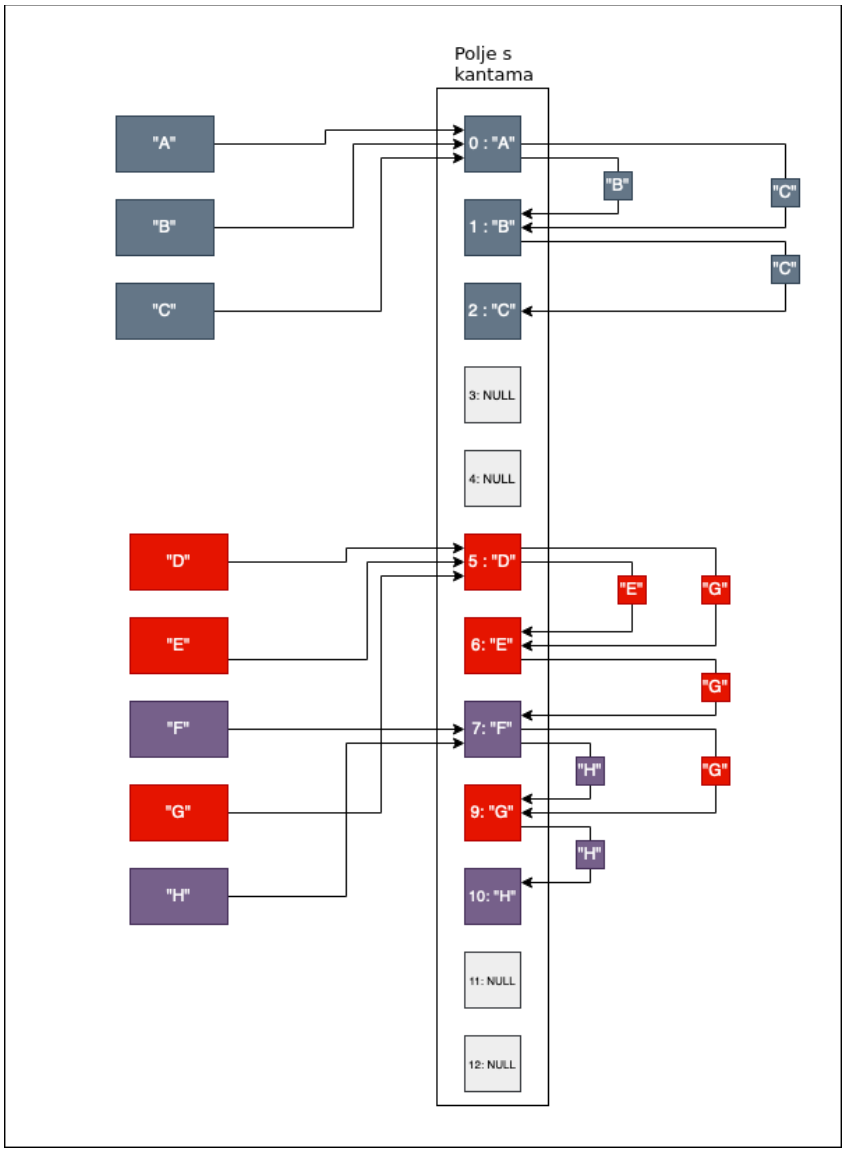

Slika 8. Hash tablica sa crno-crvenim stablom. Izvor: Prilagođeno iz [11]

U prikazanom primjeru podatci se pohranjuju u povezanu listu dok se ne dosegne granična vrijednost duljine liste, na primjer četiri. Tada se elementi liste sortiraju u crno-crveno stablo koje će ubrzati pretraživanje elemenata kako se njihov broj povećava.

## 5.2. Otvoreno adresiranje

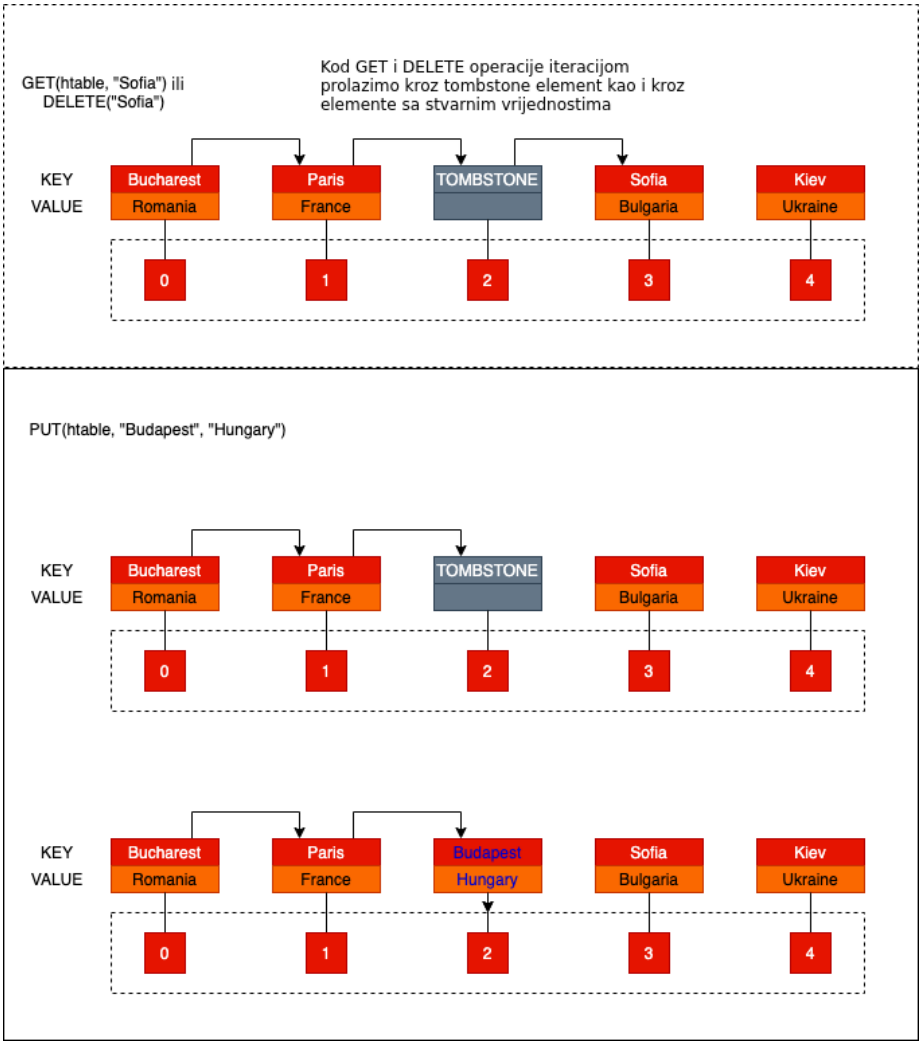
Otvoreno adresiranje (engl. *open addressing*) je drugi način za pohranu elemenata hash tablica. Od prijašnjih načina se razlikuje po tomu što ne koristi polje pokazivača na strukture koje sadrže elemente tablice, već se sami elementi nalaze u polju tablice. Zbog toga, za razliku od ostalih načina, njegov nedostatak je to što se pri stvaranju polja hash tablice odredi granica broja elemenata koja ona može sadržavati, odnosno veličina njezinog polja.

Način na koji otvoreno adresiranje rješava problem kolizija hash vrijednosti je da se kod unosa elemenata provjerava postoji li već element tablice na zadanom indeksu. Ako postoji, polje tablice se linearno pretražuje po indeksima dok se ne pronađe slobodno mjesto za unos. Proces unosa elemenata kod otvorenog adresiranja prikazan je na Slici 9. Kada se tablica popuni, petlja će kod pokušaja novog unosa pretražiti cijelo polje bez slobodnog indeksa. Zbog toga bi trebalo i provjeriti je li algoritam prošao kroz sve elemente kako ne bi došlo do beskonačne petlje.




Slika 9. Hash tablica s otvorenim adresiranjem. Izvor: Prilagođeno iz [11]

Jedan problem kod otvorenog adresiranja je da je kod brisanja jednog od elemenata u nizu jednakih hash vrijednosti moguće presjeći niz koji je potreban za pretraživanje kod pristupanja elementima. Na primjer, neka su u tablicu unesena tri elementa jednake hash vrijednosti i nalaze se na tri susjedna indeksa (gdje je indeks prvog elementa hash vrijednost preko koje im se pristupa). Ako se iz tablice izbriše drugi element, kod pristupanja trećem unesenom elementu, koji je sada drugi od ukupno dva elementa, niz za linearno pretraživanje bit će prekinut kada se s prvog elementa prijeđe na drugi element jer je on trenutno prazan. To označava da su pretraženi svi elementi zadane hash vrijednosti što nije točno. Zbog toga se umjesto označavanja izbrisanog elementa kao praznog elementa polja on označava posebnom vrijednošću, tzv. *tombstone*. Ta vrijednost označava da se na tom indeksu nekada nalazio element iste hash vrijednosti kao i onaj na prethodnom indeksu i algoritam ga tretira kao nastavak niza elemenata jednakih hash vrijednosti pri pretraživanju. Kod unosa novog elementa jednake hash vrijednosti tombstone se može tretirati kao slobodan element u polju i može se zamijeniti podatcima novog elementa koji će svejedno imati istu hash vrijednost kao i prethodni tako da niz za pretraživanje neće biti prekinut. Primjer pristupa i unosa podataka s *tombstone* elementom prikazan je na Slici 10.



Slika 10. Tombstone element hash tablice. Izvor: Prilagođeno iz [11]

# 6. Zaključak

Hash tablice su vrlo važna struktura podataka koja se zbog svoje efikasnosti koristi u brojnim softverskim rješenjima. Glavna prednost u odnosu na ostale strukture joj je prosječna vremenska kompleksnost O(1), odnosno konstantno vrijeme za unos, brisanje i pristupanje podatcima. Njezina struktura se temelji na polju fiksne duljine koje se koristi za pohranjivanje elemenata tablice ili pokazivače na druge strukture koje sadrže njezine elemente poput povezanih lista ili binarnih stabla, ovisno o potrebama. U svojoj srži hash tablica je polje fiksne duljine tako da ima sve prednosti korištenja fiksnog bloka memorije kao što je pristup bilo kojem elementu u konstantnom vremenu, no uz korištenje pametnijeg načina određivanja indeksa za pristup svakom elementu na temelju njegovog ključa tako da zaobilazi glavne nedostatke polja fiksne duljine. Za određivanje indeksa polja iz ključa elementa tablice koriste se hash algoritmi, odnosno algoritmi koji na deterministički način iz dobivenog ključa određuju vrijednost koja se koristi kao indeks u polju tablice. Cilj ovih algoritama je uniformna distribucija povratnih vrijednosti, odnosno da je kod proizvoljne vrijednosti ključa svaka moguća izlazna vrijednost jednake vjerojatnosti. To optimizira korištenje polja za pohranu elemenata i smanjuje broj kolizija, različitih ključeva u tablici koji svejedno dobivaju istu hash vrijednost, pa time i indeks, i usporavaju pretraživanje tablice.

# 7. Literatura


[1] *C Arrays*, GeeksforGeeks, https://www.geeksforgeeks.org/c-arrays/ (pristupljeno 15.6.2024.)

[2] *Applications of linked list data structure*, GeeksforGeeks, https://www.geeksforgeeks.org/applications-of-linked-list-data-structure/ (pristupljeno 15.6.2024.)

[3] *Binary Search Tree*, GeeksforGeeks, https://www.geeksforgeeks.org/binary-search-tree-data-structure/ (pristupljeno 15.6.2024.)

[4] *Implementation of Hash Table in Python using Separate Chaining*, GeeksforGeeks, https://www.geeksforgeeks.org/implementation-of-hash-table-in-python-using-separate-chaining/ (pristupljeno 15.6.2024.)

[5] Cormen, Leiserson, Rivest, Stein, Introduction to Algorithms, Second Edition, MIT Press, 2001.

[6] Mailund Thomas, The Joys of Hashing: Hash Table Programming with C, Apress, 2019.

[7] *Why is it best to use a prime number as a mod in a hashing function?*, Computer Science Stack Exchange, https://cs.stackexchange.com/questions/11029/why-is-it-best-to-use-a-prime-number-as-a-mod-in-a-hashing-function (pristupljeno 19.6.2024.)

[8] *Insertion in Linked List in C*, PrepInsta, https://prepinsta.com/c-program/insertion-in-linked-list/ (pristupljeno 20.6.2024.)

[9] *Illustration of a classical hash map using separate chaining*, ResearchGate, https://www.researchgate.net/figure/Illustration-of-a-classical-hash-map-using-separate-chaining-Keys-left-are-put-into_fig1_355070749 (pristupljeno 20.6.2024.)

[10] *Deletion in Linked List*, GeeksforGeeks, https://www.geeksforgeeks.org/deletion-in-linked-list/ (pristupljeno 20.6.2024.)

[11] Andrei Ciobanu, *Implementing Hash Tables in C*, andreinc, https://www.andreinc.net/2021/10/02/implementing-hash-tables-in-c-part-1 (pristupljeno 1.7.2024.)

# 8. Popis slika

# 9. Popis priloga

Prilog 1: Kôd datoteke hashtable.h

Prilog 2: Kôd datoteke hashtable.c

Prilog 3: Kôd datoteke test.c

# 10. Kôd datoteke hashtable.h

```
#ifndef _HASHTABLE_H
#define _HASHTABLE_H

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

struct entry_t
{
    uint8_t *key;
    size_t key_size;
    void *value;
    struct entry_t *next;
};

struct ht_t
{
    struct entry_t **entries;
    size_t size;
    uint32_t (*hashfunction)(const uint8_t * const, const size_t);
};

struct ht_t *ht_create(size_t size);

void ht_destroy(struct ht_t *ht, bool free_values);

bool ht_set(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size, void *value);

void *ht_get(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size);

bool ht_remove(struct ht_t *ht, const uint8_t * const key,
            const size_t key_size, bool free_value);

void ht_print_keys(const struct ht_t * const ht);

void ht_print_distribution(const struct ht_t * const ht);

bool ht_resize(struct ht_t *ht, size_t new_size);

#endif
```

# 11. Kôd datoteke hashtable.c

```
#include "hashtable.h"
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdio.h>

#define HT_SIZE 524243


uint32_t testhash(const uint8_t * const input, const size_t input_size)
{
    uint32_t sum = 0;
    for (size_t i = 0; i < input_size; ++i)
    {
        sum = sum * 31 + input[i];
    }
    return sum;
}

static size_t get_index(const struct ht_t * const ht, const uint32_t key_hash)
{
    size_t index = key_hash % ht->size;
    return index;
}

struct ht_t *ht_create(size_t size)
{
    struct ht_t *ht = calloc(1, sizeof *ht);
    if (ht == NULL)
    {
        return NULL;
    }

    if (size == 0)
    {
        ht->size = HT_SIZE;
    }
    else
    {
        ht->size = size;
    }

    ht->entries = calloc(ht->size, sizeof *ht->entries);
    if (ht->entries == NULL)
    {
        free(ht);
        return NULL;
    }

    ht->hashfunction = &testhash;

    return ht;
}
```

```
static void free_entries(struct entry_t **entries, const size_t size,
            bool free_values)
{
    struct entry_t *p;
    struct entry_t *p_next;

    for (size_t i = 0; i < size; ++i)
    {
        p = entries[i];

        while (p != NULL)
        {
            p_next = p->next;

            if (free_values)
            {
                free(p->value);
            }
            free(p->key);
            free(p);

            p = p_next;
        }
    }
}

void ht_destroy(struct ht_t *ht, bool free_values)
{
    free_entries(ht->entries, ht->size, free_values);

    free(ht->entries);
    free(ht);
}

static struct entry_t *find_entry(struct entry_t * const entry_row,
            const uint8_t * const key, const size_t key_size)
{
    struct entry_t *match = entry_row;
    while (match != NULL)
    {
        if (key_size == match->key_size &&
            memcmp(match->key, key, key_size) == 0)
        {
            return match;
        }

        match = match->next;
    }

    return NULL;
}

bool ht_set(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size, void *value)
{
```

```
    if (value == NULL)
    {
        return false;
    }

    uint32_t key_hash = (*ht->hashfunction)(key, key_size);
    size_t index = get_index(ht, key_hash);

    size_t key_mem_size = key_size * sizeof *key;

    struct entry_t *match = find_entry(ht->entries[index], key, key_size);

    if (match == NULL)
    {
        struct entry_t *first = ht->entries[index];

        ht->entries[index] = calloc(1, sizeof *ht->entries[index]);
        if (ht->entries[index] == NULL)
        {
            ht->entries[index] = first;
            return false;
        }

        ht->entries[index]->key_size = key_size;
        ht->entries[index]->key = malloc(key_mem_size);
        if (ht->entries[index]->key == NULL)
        {
            free(ht->entries[index]);
            ht->entries[index] = first;
            return false;
        }

        memcpy(ht->entries[index]->key, key, key_mem_size);
        ht->entries[index]->value = value;
        ht->entries[index]->next = first;
    }
    else
    {
        if (key_size != match->key_size)
        {
            match->key_size = key_size;
            free(match->key);
            match->key = malloc(key_mem_size);
            if (match->key == NULL)
            {
                return false;
            }
        }

        memcpy(match->key, key, key_mem_size);
        match->value = value;
    }

    return true;
}
```

```
void *ht_get(struct ht_t * const ht, const uint8_t * const key,
            const size_t key_size)
{
    uint32_t key_hash = (*ht->hashfunction)(key, key_size);
    size_t index = get_index(ht, key_hash);

    const struct entry_t *match = find_entry(ht->entries[index], key,
            key_size);

    if (match == NULL)
    {
        return NULL;
    }

    return match->value;
}

bool ht_remove(struct ht_t *ht, const uint8_t * const key,
            const size_t key_size, bool free_value)
{
    uint32_t key_hash = (*ht->hashfunction)(key, key_size);
    size_t index = get_index(ht, key_hash);

    struct entry_t *match = ht->entries[index];
    struct entry_t *prev = NULL;

    while (match != NULL)
    {
        if (key_size == match->key_size &&
            memcmp(match->key, key, key_size) == 0)
        {
            break;
        }
        prev = match;
        match = match->next;
    }

    if (match == NULL)
    {
        return false;
    }

    if (prev == NULL)
    {
        ht->entries[index] = match->next;
    }
    else
    {
        prev->next = match->next;
    }

    if (free_value)
    {
        free(match->value);
    }
    free(match->key);
```

```
    free(match);

    return true;
}

void ht_print_keys(const struct ht_t * const ht)
{
    struct entry_t *p;
    for (size_t i = 0; i < ht->size; ++i)
    {
        p = ht->entries[i];
        while (p != NULL)
        {
            printf("0x");
            for (size_t j = 0; j < p->key_size; ++j)
            {
                printf("%hhx", p->key[j]);
            }
            printf("\n");
            p = p->next;
        }
    }
}

void ht_print_distribution(const struct ht_t * const ht)
{
    size_t *counts = calloc(ht->size, sizeof *counts);
    struct entry_t *p;

    for (size_t i = 0; i < ht->size; ++i)
    {
        size_t count = 0;
        p = ht->entries[i];
        while (p != NULL)
        {
            ++count;
            p = p->next;
        }
        ++counts[count];
    }

    for (size_t i = 0; i < ht->size; ++i)
    {
        if (counts[i] > 0)
        {
            printf("%zu:\t%zu\n", i, counts[i]);
        }
    }

    free(counts);
}

bool ht_resize(struct ht_t *ht, const size_t new_size)
{
    if (new_size == ht->size)
    {
```

```
        return true;
    }

    struct entry_t **new_entries = calloc(new_size, sizeof *new_entries);
    if (new_entries == NULL)
    {
        return false;
    }

    struct entry_t **old_entries = ht->entries;
    size_t old_size = ht->size;

    ht->entries = new_entries;
    ht->size = new_size;

    struct entry_t *p;
    bool set_success = true;

    for (size_t i = 0; i < old_size; ++i)
    {
        p = old_entries[i];

        while (p != NULL)
        {
            set_success = ht_set(ht, p->key, p->key_size, p->value);
            if (!set_success)
            {
                break;
            }
            p = p->next;
        }
        if (!set_success)
        {
            break;
        }
    }

    if (!set_success)
    {
        free_entries(ht->entries, ht->size, false);
        free(ht->entries);
        ht->entries = old_entries;
        ht->size = old_size;

        return false;
    }

    free_entries(old_entries, old_size, false);
    free(old_entries);

    return true;
}
```

# 12. Kôd datoteke test.c

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "hashtable.h"

#define LINE_MAX 4096

struct data_t {
    size_t size;
    uint8_t *p;
};

int main(void)
{
    struct ht_t *table = ht_create(30011);

    char *filename = "testdata.txt";
    FILE *fp = fopen(filename, "r");
    if (fp == NULL)
    {
        fprintf(stderr, "error opening %s\n", filename);
        return 1;
    }

    char key_line[LINE_MAX] = "";
    char value_line[LINE_MAX] = "";
    size_t key_line_size;
    size_t value_line_size;

    while (true)
    {
        if (fgets(key_line, LINE_MAX, fp) == NULL)
        {
            break;
        }
        if (fgets(value_line, LINE_MAX, fp) == NULL)
        {
            fprintf(stderr, "odd number of lines in file\n");
            break;
        }

        key_line_size = strlen(key_line);
        if (key_line[key_line_size - 1] == '\n')
        {
            key_line[key_line_size - 1] = '\0';
            --key_line_size;
        }
        else
        {
            fprintf(stderr, "hex key line missing newline\n");
            return 1;
        }
        if (key_line_size % 2 == 1)
```

```
{
    fprintf(stderr, "invalid hex key line length\n");
    return 1;
}

value_line_size = strlen(value_line);
if (value_line[value_line_size - 1] == '\n')
{
    value_line[value_line_size - 1] = '\0';
    --value_line_size;
}
else
{
    fprintf(stderr, "hex value line missing newline\n");
    return 1;
}
if (key_line_size % 2 == 1)
{
    fprintf(stderr, "invalid hex value line length\n");
    return 1;
}

uint8_t key[LINE_MAX / 2];
uint8_t value[LINE_MAX / 2];
size_t key_size = key_line_size / 2;
size_t value_size = value_line_size / 2;

for (size_t i = 0; i < key_line_size / 2; ++i)
{
    char buf[3] = {key_line[i * 2], key_line[i * 2 + 1], '\0'};
    key[i] = strtol(buf, NULL, 16);
}

for (size_t i = 0; i < value_line_size / 2; ++i)
{
    char buf[3] = {value_line[i * 2], value_line[i * 2 + 1], '\0'};
    value[i] = strtol(buf, NULL, 16);
}

struct data_t *data = malloc(sizeof *data);
if (data == NULL)
{
    return 1;
}

data->size = value_size;
data->p = malloc(data->size);
if (data->p == NULL)
{
    free(data);
    return 1;
}

memcpy(data->p, value, data->size);

ht_set(table, key, key_size, data);
```

```
    }

    ht_print_distribution(table);

    ht_resize(table, 10007);

    ht_print_distribution(table);

    ht_destroy(table, true);

    return 0;
}
```