Algoritmi i strukture podataka

- predavanja -

12. Raspršeno adresiranje

Raspršeno adresiranje (eng. hashing)

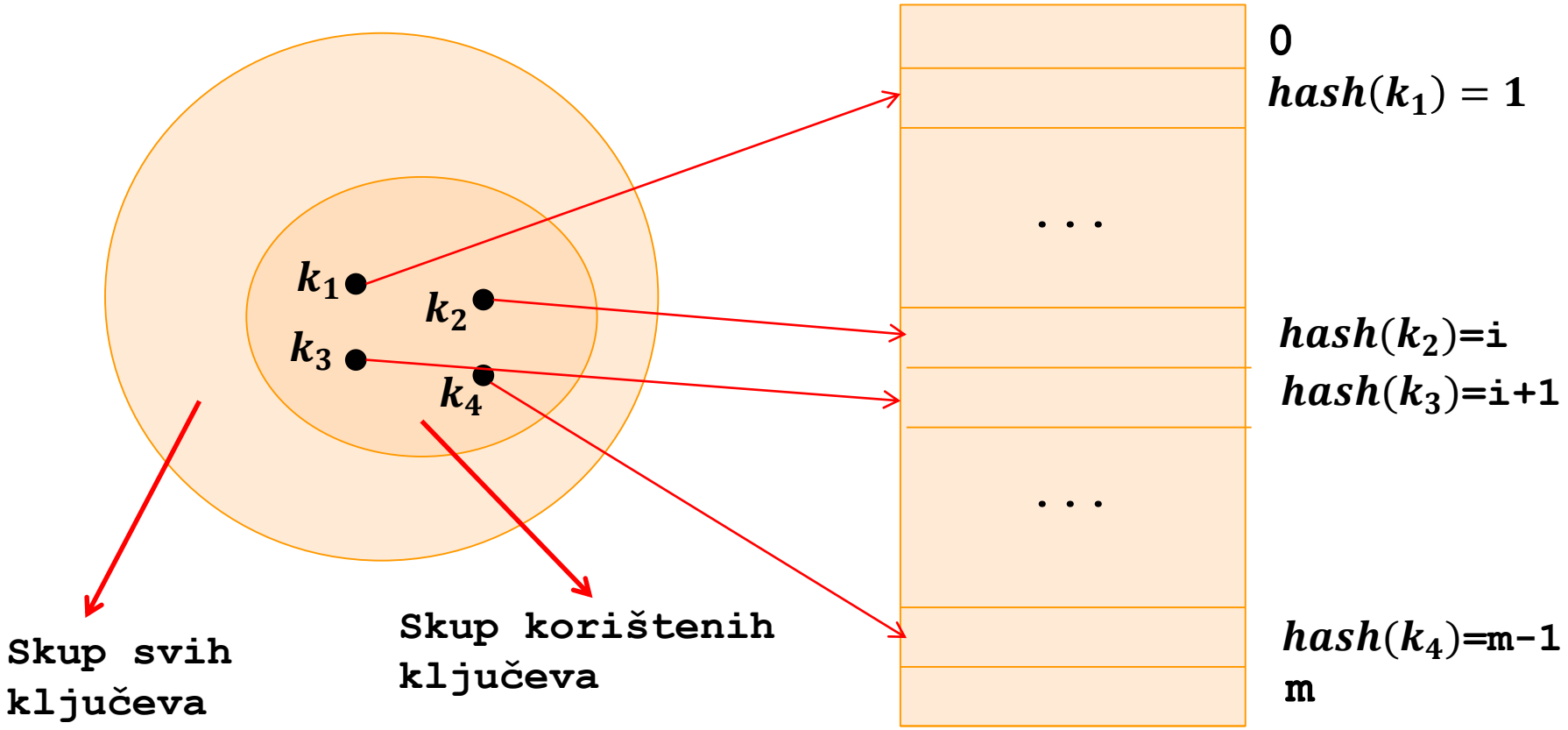
- osnovni pojmovi
- kolizija i rješavanje kolizije
- izrada hash-funkcije
- raspršeno adresiranje u datoteci

Raspršeno adresiranje (eng. hashing)

- Zadatak: pohraniti određeni skup ključeva, a da pohrana i pristup vrijednostima budu što brži
- Početna ideja: naizgled najjednostavnije je pohraniti ključeve u niz
  - Za nesortirani niz vremenska složenost pretrage u najgorem slučaju iznosi O(n)
  - Za soritrani niz vremenska složenost pretrage u najgorem slučaju iznosi O(log n)
- Problem: Što ako imamo 10 000 ključeva koje treba pohraniti?
  - Pretraživanje niza postaje neefikasno
  - Za veliki broj vrijednosti ovaj način pohrane nije dobar
- Može li bolje?

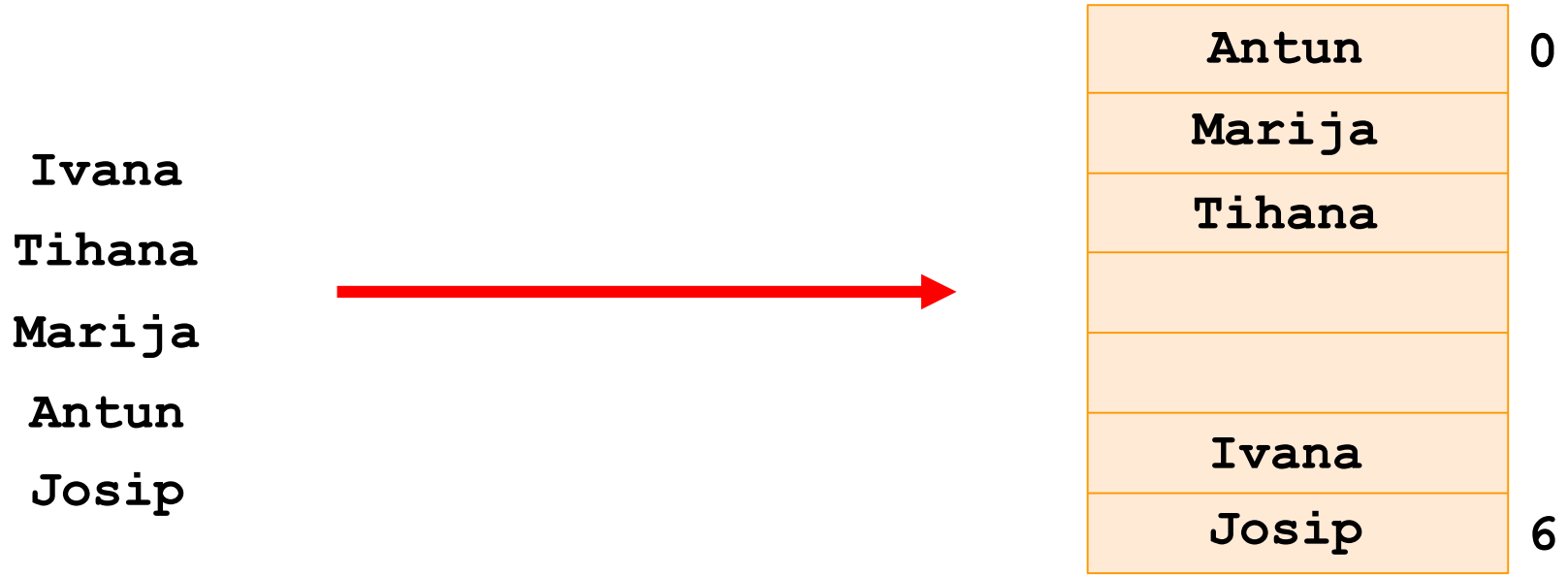
Raspršeno adresiranje (eng. hashing)

- Može, ako za ključ k uvedemo funkciju $hash(k)$ koja mapira vrijednost ključa u indeks pod kojim se taj ključ nalazi u hash tablici



Raspršeno adresiranje (eng. hashing)

- Prilikom pristupa ključu izračunava se vrijednost $hash(k)$ koja odgovara indeksu mjesta na kojem se ključ nalazi
- Na taj način, u idealnom slučaju, mogu se postići pohrana i pristup ključu u konstantnom $O(1)$ vremenu
- Primjer: želimo pohraniti 5 ključeva u hash tablicu
  - hash(k) = ASCII suma slova % veličina tablice



Kolizija pri raspršenom adresiranju

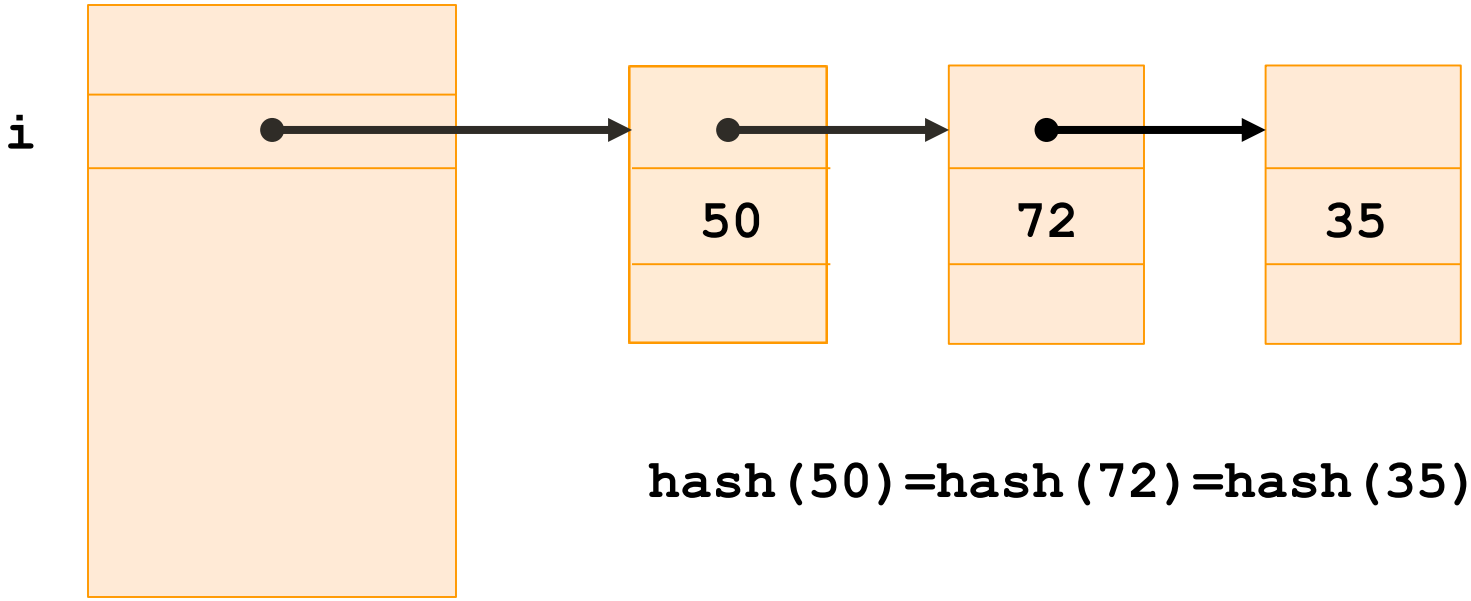
- Npr. za ključ „Antun” preslikavanje bi izgledalo:
  - $[ASCII(A) + ASCII(n) + ASCII(t) + ASCII(u) + ASCII(n)]$ % 7
  - $(65 + 110 + 116 + 117 + 110)$ % 7 = 0
  - Ključ „Antun” se preslikava u pretinac s indeksom 0 u hash tablici
- Što kada se više ključeva mapira na isto mjesto u hash tablici?
  - $hash(Iva) = hash(Marija)$
  - Dolazi do problema **kolizije**
- Za izbjeći koliziju možemo koristiti direktno adresiranje
  - Svaki ključ se mapira na jedinstveno mjesto u hash tablici
  - Npr. ključevi 30,127, 200 bi se mapirali redom na ta mjesta u hash tablici
  - Omogućuje ubacivanje i pretragu u konstantnom vremenu
  - Problem: za mali broj ključeva velika neiskorištenost memorije

Tehnike za rješavanje kolizije pri raspršenom adresiranju

- Drugi način pristupa je da konstruiramo savršenu hash funkciju
  - Napravljena da sve ključeve mapira na jedinstvena mjesta – nema kolizije
  - Konstantno vrijeme ubacivanja i pretrage
  - Problem: teška za konstruirati
- U praksi koristimo tehnike za rješavanje kolizije
  - ulančavanje
  - otvoreno adresiranje
    - Linearno ispitivanje
    - Kvadratno ispitivanje
    - Dvostruko raspršeno adresiranje

Rješavanje kolizije: Ulančavanje

- Elemente koje se preslikavaju na istu lokaciju u hash tablici dodajemo u listu



`hash(50)=hash(72)=hash(35)=i`

Rješavanje kolizije: Ulančavanje

- Pojedini pretinac hash tablice sadržava pokazivač na početak liste
- Omogućeno dodavanje elemenata u $O(1)$ vremenu
- Prilikom pretrage izračunamo $hash(k)$ kako bi saznali indeks te pretražujemo listu dok ne nađemo traženi element
  - Ne postoji mogućnost popunjavanja hash tablice, no povećanjem broja elemenata u listi pokazivača raste vrijeme pretrage
  - U praksi dobar način rješavanja kolizije ako osiguramo da liste pokazivača nisu prevelike

Klase za implementaciju hash-a - ulančavanje

```
template <typename T, typename K> class IHashableValue {
 public:
   virtual K GetKey() const = 0;
};

template <typename T, typename K> class HashElement {
 public:
   IHashableValue<T, K> *value;
   HashElement *next;
   HashElement(IHashableValue<T, K> *value) { this->value = value; }
};

template <typename T, typename K> class IHash {
 protected:
   size_t size;

 public:
   virtual void Add(IHashableValue<T, K> *element) const = 0;
   virtual IHashableValue<T, K> *Get(K key) const = 0;
   virtual void Print() const = 0;
};
```

IHashableValue će naslijediti klasa čiji će se objekti stavljati u hash

HashElement je klasa koja će služiti za ulančavanje elemenata. U njoj je učahuren objekt klase IHashableElement

IHash je klasa koju nasljeđuju konkretne implementacije hash-a. Ona definira tri apstraktne metode koje svaki hash za sebe implementira

Hash.h

Funkcije Add i Get - ulančavanje

```
virtual void Add(IHashableValue<T, K> *element) const {
      int i = HashStringToInt(element->GetKey(), this->size);
      HashElement<T, K> *el = new HashElement<T, K>(element);
      el->next = hash[i];
      hash[i] = el;
}

virtual IHashableValue<T, K> *Get(K key) const {
      int i = HashStringToInt(key, this->size);
      HashElement<T, K> *head;
      for (head = hash[i]; head && (head->value->GetKey() != key);
           head = head->next)
         ;
      return head->value;
}
```

HashChaining.cpp

Primjer klase za hash: `Person`- ulančavanje

```
class Person : public IHashableValue<Person, string> {
 public:
   string name;
   Person() {}
   Person(string name) { this->name = name; }
   virtual string GetKey() const { return name; }
};
```

HashChaining.cpp

Analiza ulančavanja

- Najgori slučaj: svaki ključ se mapira na isto mjesto
- Θ(n) vrijeme pretrage
- Prosječan slučaj: pretpostavimo jednostavno uniformno raspršeno adresiranje, odnosno da za svaki ključ vrijedi jednaka vjerojatnost preslikavanja na bilo koju lokaciju u hash tablici
- Nadalje, definiramo faktor opterećenja $\alpha = n/m$, pri čemu je n broj ključeva u hash tablici, a m ukupan broj lokacija u hash tablici
- Očekivano vrijeme pretrage iznosi $\Theta(1+\alpha)$

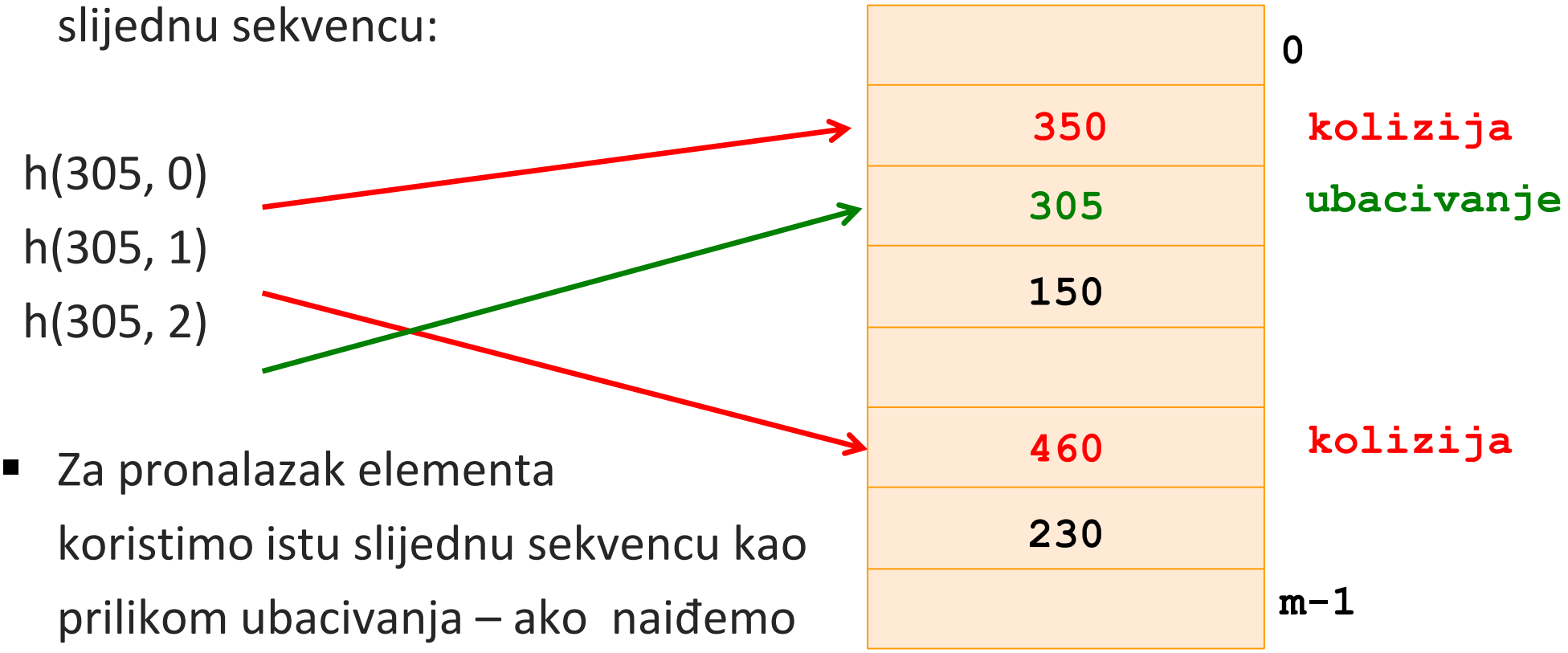
- Iz ovoga vidimo da vrijeme pretrage iznosi $\Theta(1)$ ako vrijedi α=O(1), a isto tako i ako vrijedi n=O(m)

Rješavanje kolizije: Otvoreno adresiranje

- Ne koristi se memorija izvan hash tablice kao kod ulančavanja, tj. svi elementi se nalaze unutar same hash tablice
- Prednost: imamo više memorije stoga možemo imati i veću hash tablicu
- Mana: postoji mogućnost da se hash tablica popuni i da ne možemo dodavati nove elemente (faktor opterećenja uvijek < 1)
- Prilikom ubacivanja elementa, slijedno ispitujemo hash tablicu dok ne nađemo prazno mjesto
  - Slijed lokacija koje se ispituju ovisi o ključu koji se ubacuje
  - Kako bi odredili koje lokacije ispitati, proširujemo hash funkciju slijednim brojem i zahtijevamo da slijedna sekvenca $h(k,0), h(k,1), .., h(k,m-1)$ naposljetku ispita sve lokacije u hash tablici prilikom ubacivanja novog elementa

Primjer otvorenog adresiranja

- Uzmimo funkciju koja prilikom ubacivanja ključa 305 ima sljedeću slijednu sekvencu:



- Za pronalazak elementa koristimo istu slijednu sekvencu kao prilikom ubacivanja – ako naiđemo na prazno mjesto, element se ne nalazi u hash tablici

Klase za implementaciju hash-a – otvoreno adresiranje

```
template <typename T, typename K> class IHashableValue {
 public:
   virtual K GetKey() const = 0;
};


template <typename T, typename K> class IHash {
 protected:
   size_t size;

 public:
   virtual void Add(IHashableValue<T, K> *element) const = 0;
   virtual IHashableValue<T, K> *Get(K key) const = 0;
   virtual void Print() const = 0;
};
```

IHashableValue će naslijediti klasa čiji će se objekti stavljati u hash

IHash je klasa koju nasljeđuju konkretne implementacije hash-a. Ona definira tri apstraktne metode koje svaki hash za sebe implementira

Hash.h

Pseudokôd funkcije `Upis` za otvoreno adresiranje (sve varijante)

```
Adresa = Izračunaj hash funkciju
Ponavljaj za brojač od 0 do veličina_tablice – 1
    Izračunaj indeks pretinca (Adresa)
    Ako je pretinac prazan
        Ubaci element u pretinac
        Izlaz iz petlje
```

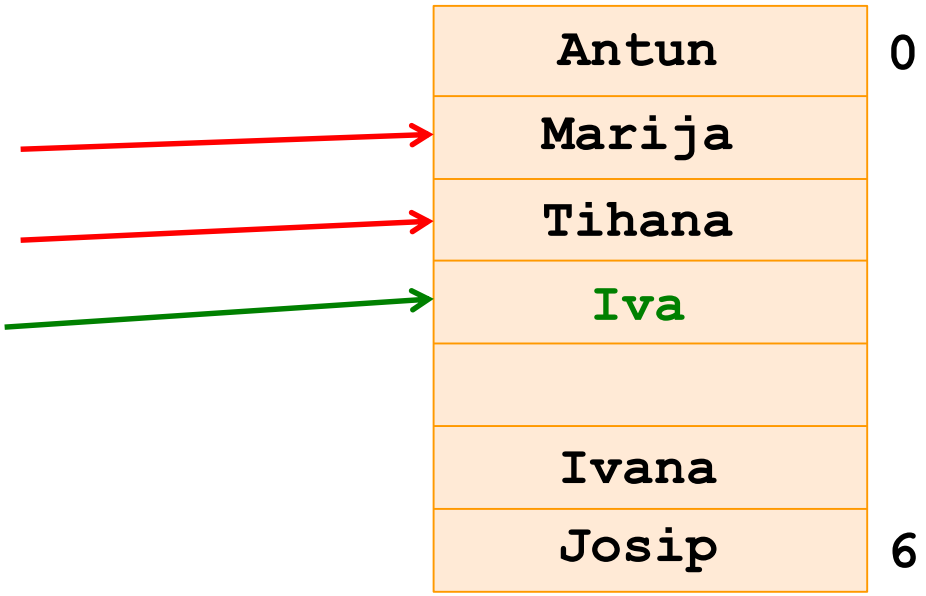
Tehnike otvorenog adresiranja – linearno ispitivanje

- Linearno ispitivanje (eng. linear probing): $h(k,i) = (h'(k) + i) mod\ m$, i = 0,…,m-1
  - pri čemu je h'(k) neka pomoćna hash funkcija
- U slijednoj sekvenci, nakon svakog ispitivanja povećavamo i za 1 te se na taj način zapravo pomičemo po susjednim mjestima u hash tablici dok ne nađemo prazno mjesto na koje možemo ubaciti ključ
- Problem je kad omogućimo brisanje elemenata iz hash tablice
  - Ako prilikom pretrage naiđemo na prazno mjesto u hash tablici, više ne možemo zaključiti da element ne postoji u tablici
  - Međutim, možemo voditi računa o najdužoj sekvenci prilikom ubacivanja te proglasiti da elementa nema tek kad obavimo onoliko ispitivanja koliko odgovara broju ispitivanja najduže sekvence
- Kod linearnog ispitivanja postoji i problem linearnog grupiranja
  - Ključevi se nakupljaju u pojedinim dijelovima tablice
  - Povećano vrijeme ubacivanja i pretrage

Primjer linearnog ispitivanja

- Uzmimo već poznati primjer kad se ključ „Iva” mapira na isto mjesto kao i ključ „Marija” u hash tablici
  - Dakle, za početni slučaj, odnosno i=0 vrijedi: hash(Iva,0)=hash(Marija,0)
- Uzmimo da je pomoćna funkcija $h'(k) =$ ASCII suma slova
  - $h'(Iva) = 288$

$$h(Iva, 0) = (288 + 0) mod\ 7 = 1$$
$$h(Iva, 1) = (288 + 1) mod\ 7 = 2$$
$$h(Iva, 2) = (288 + 2) mod\ 7 = 3$$



Funkcije Add – linearno ispitivanje

```
virtual void Add(IHashableValue<T, K> *element) const {
      int address = HashIntOverlap(element->GetKey());
      int index;
      for (int i = 0; i < this->size; i++) {
         index = (address + i) % this->size;
         if (hash[index] == nullptr) {
            hash[index] = element;
            break;
         }
      }
}
```

HashLinearProbing.cpp

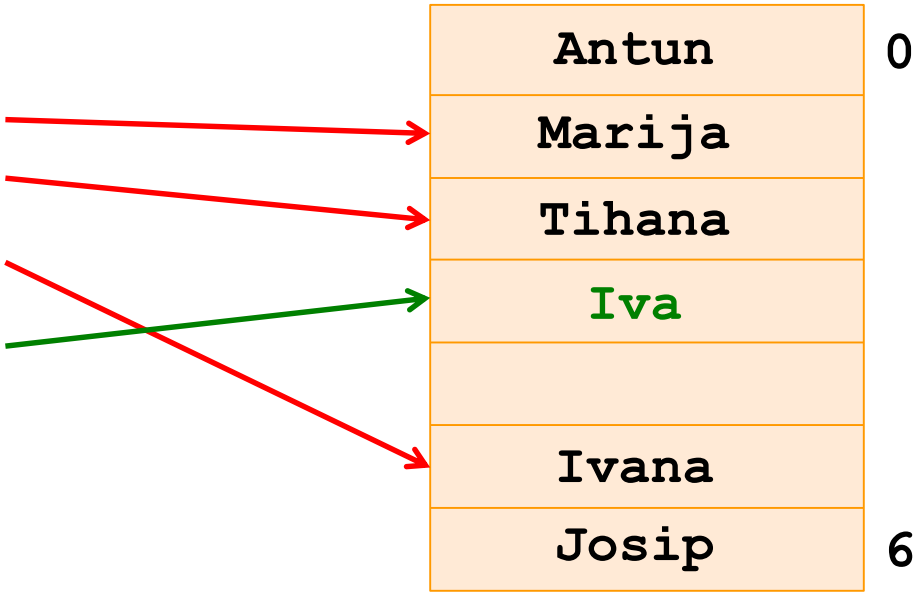
Tehnike otvorenog adresiranja – kvadratno ispitivanje

- Kvadratno ispitivanje (eng. quadratic probing): $h(k,i) = (h'(k) + c_1 i + c_2 i^2) mod\ m$
  - c_1 i $c_2 \neq 0$ su pomoćne konstante, a i = 0,...,m-1
  - Korištenjem kvadratnog ispitivanja rješava se problem linearnog grupiranja
- No, postoji problem kvadratnog grupiranja
  - Ako dva ključa imaju istu početnu slijednu poziciju
  - Tad zbog $h(k_1, 0) = h(k_2, 0)$ slijedi $h(k_1, i) = h(k_2, i)$
  - Tj., imaju istu slijednu sekvencu
  - Ipak, manji problem od linearnog grupiranja

Primjer kvadratnog ispitivanja

- Uzmimo opet da je pomoćna funkcija $h'(k) =$ ASCII suma slova
  - $h'(Iva) = 288$
- Također, radi jednostavnosti uzmimo da je $c_1 = 0$ i $c_2 = 1$
  - Opet vrijedi hash(Iva,0)=hash(Marija,0)

$$h(Iva, 0) = (288 + 0) mod\ 7 = 1$$
$$h(Iva, 1) = (288 + 1) mod\ 7 = 2$$
$$h(Iva, 2) = (288 + 4) mod\ 7 = 5$$
$$h(Iva, 3) = (288 + 9) mod\ 7 = 3$$



Funkcije Add – kvadratno ispitivanje

```
virtual void Add(IHashableValue<T, K> *element) const {
     int address = HashMultiplicationMethod(element->GetKey());
     int index;
     for (int i = 0; i < this->size; i++) {
        index = (int)fmod((address + c1 * i + c2 * i * i), this->size);
        if (hash[index] == nullptr) {
           hash[index] = element;
           break;
        }
     }
  }
```

HashQuadraticProbing.cpp

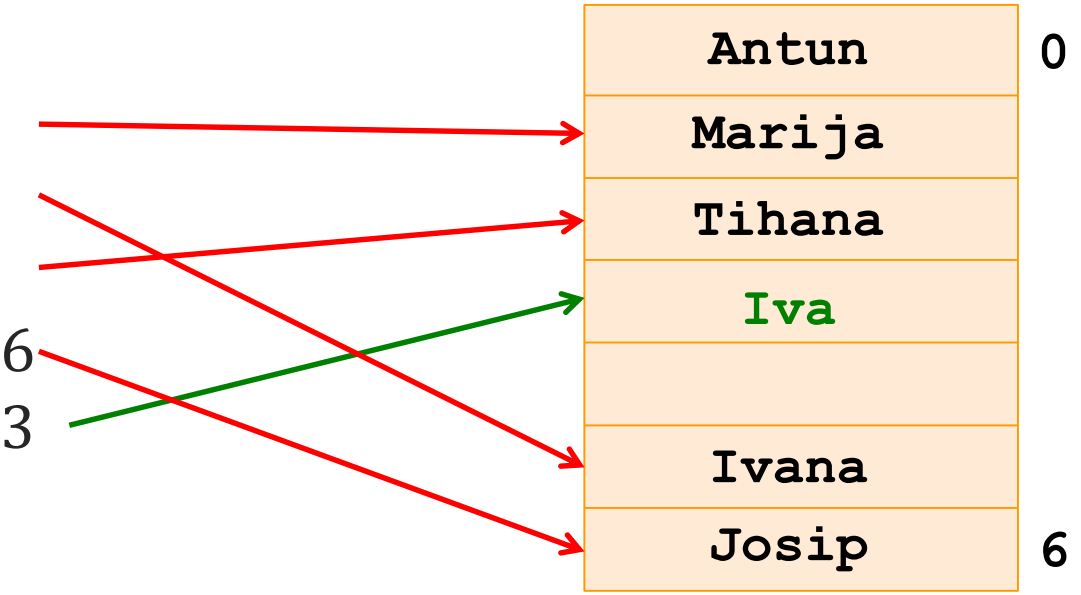
Tehnike otvorenog adresiranja – dvostruko raspršeno adresiranje

- Dvostruko raspršeno adresiranje (eng. double hashing):
  - $h(k,i) = \big(h_1(k) + ih_2(k)\big) mod\ m$, i=0,…,m-1
  - $h_1(k)$, $h_2(k)$ su neovisne pomoćne funkcije
- Budući da ovisi o dvije funkcije, izbjegava se problem grupiranja
  - Vrlo blizu idealnoj funkciji raspršenog adresiranja
  - Permutacije dobivene dvostrukim raspršenim adresiranjem imaju karakteristike nasumičnosti
- Vrijednost $h_2(k)$ mora biti relativno prost broj u odnosu na veličinu hash tablice m (nemaju zajedničkog djelitelja osim 1) kako bi cijela hash tablica mogla biti pretražena
  - To možemo postići npr. ako stavimo da je m potencija broja dva, a dizajn funkcije h_2 takav da uvijek vraća neparne brojeve

Primjer dvostrukog raspršenog adresiranja

- Uzmimo sad da je pomoćna funkcija h_1 = ASCII suma slova%7, a h_2 = 1 + ASCII suma slova%5
  - Dakle, za primjer $h_1(Iva) = 288\%7 = 1$
  - $h_2(Iva) = 1 + 288\%5 = 1 + 3 = 4$

$$h(Iva, 0) = (1 + 0) mod\ 7 = 1$$
$$h(Iva, 1) = (1 + 4) mod\ 7 = 5$$
$$h(Iva, 2) = (1 + 8) mod\ 7 = 2$$
$$h(Iva, 3) = (1 + 12) mod\ 7 = 6$$
$$h(Iva, 4) = (1 + 16) mod\ 7 = 3$$



Funkcije Add – dvostruko raspršeno ispitivanje

```
virtual void Add(IHashableValue<T, K> *element) const {
     int h1 = HashDoubleHashing1(element->GetKey());
     int h2 = HashDoubleHashing2(element->GetKey());
     int index;
     for (int i = 0; i < this->size; i++) {
        index = (h1 + i * h2) % this->size;
        if (hash[index] == nullptr) {
           hash[index] = element;
           break;
        }
     }
  }
```

HashDoubleHashing.cpp

Analiza otvorenog adresiranja

- Za razliku od ulančavanja, možemo imati samo jedan element po mjestu u hash tablici
  - Stoga vrijedi $n \leq m$ pa je $\alpha \leq 1$
  - Štoviše, prilikom ubacivanja treba vrijediti $\alpha < 1$ jer je za $\alpha = 1$ (zbog n=m) hash tablica već popunjena
- Pretpostavljamo uniformno raspršeno adresiranje
  - Svaka sekvenca ispitivanja $h(k,0), h(k,1), \ldots, h(k,m-1)$ je jednako vjerojatna za svaki ključ
- Ako imamo zadanu hash tablicu s faktorom opterećenja $\alpha = n/m$, očekivani broj ispitivanja prilikom neuspješne pretrage iznosi najviše $1/(1-\alpha)$

Analiza otvorenog adresiranja

- Dokaz:
  - Barem jedno ispitivanje je uvijek potrebno
  - Vjerojatnost da prilikom prvog ispitivanja naiđemo na koliziju iznosi n/m
  - Vjerojatnost da u drugom ispitivanju naiđemo na koliziju iznosi $(n-1)/(m-1)$, prilikom trećeg $(n-2)/(m-2)$ itd.
- Valja primjetiti da $\frac{n-i}{m-i} < \frac{n}{m}$, za $i = 0, \ldots, m-1$
  - stoga je očekivani broj ispitivanja

$$1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(\ldots\left(1 + \frac{1}{m-n+1}\right)\ldots\right)\right)\right)$$

$$\leq 1 + \alpha\left(1 + \alpha\left(1 + \alpha(\ldots(1+\alpha)\ldots)\right)\right)$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Analiza otvorenog adresiranja

- Za mali α, npr. 0.1
  - Očekivani broj ispitivanja je $1/(1-0.1) = 1/0.9 \approx 1$
  - Jedno ispitivanje potrebno prilikom ubacivanja ili pretrage
- Za $\alpha = 0.5$, dakle kad je hash tablica polupopunjena
  - Očekivani broj ispitivanja je $1/(1-0.5) = 2$
  - Dva ispitivanja potrebna prilikom ubacivanja ili pretrage
- Za $\alpha = 0.9$, tj. za 90% popunjenu hash tablicu
  - Očekivani broj ispitivanja je $1/(1-0.9) = 10$
  - Deset ispitivanja potrebnih prilikom ubacivanja ili pretrage
- Vidljivo je da rezultati ovise o α
  - Dobra praksa održavati vrijednost faktora opterećenja na 0.5 - 0.6

Karakteristike dobre hash-funkcije

- izlazna vrijednost ovisi isključivo o ulaznom podatku
  - kad bi ovisila i o nekoj drugoj varijabli, pri pretraživanju bi trebalo poznavati i tu varijablu
- funkcija koristi sve ulazne podatke
  - kad ne bi koristila sve podatke, uz male varijacije ulaznih podataka bio bi preveliki broj istih izlaznih vrijednosti - narušava se razdioba
- jednoliko raspoređuje izlazne vrijednosti
  - u suprotnom se smanjuje učinkovitost
- za slične ulazne podatke daje vrlo različite izlazne vrijednosti
  - u realnosti su ulazni podaci često vrlo slični, želimo ih ravnomjerno raspodijeliti

Transformacija ključa u adresu

- općenito se ključ transformira u adresu pretinca u 3 koraka:
  - ako ključ nije numerički, treba ga transformirati u broj i to bez gubitka informacije
  - nad ključem se upotrijebi algoritam koji ga transformira, što je moguće ravnomjernije, u pseudo-slučajni broj reda veličine broja pretinaca
  - rezultat se množi s odgovarajućom konstantom ≤ 1 zbog transformacije u interval relativnih adresa koji je jednak broju pretinaca
    - relativne adrese se konvertiraju u apsolutne na konkretnoj fizičkoj jedinici i to je u pravilu zadatak sistemskih programa
- idealna transformacija: vjerojatnost da 2 različita ključa u tablici veličine M daju istu adresu je 1/M

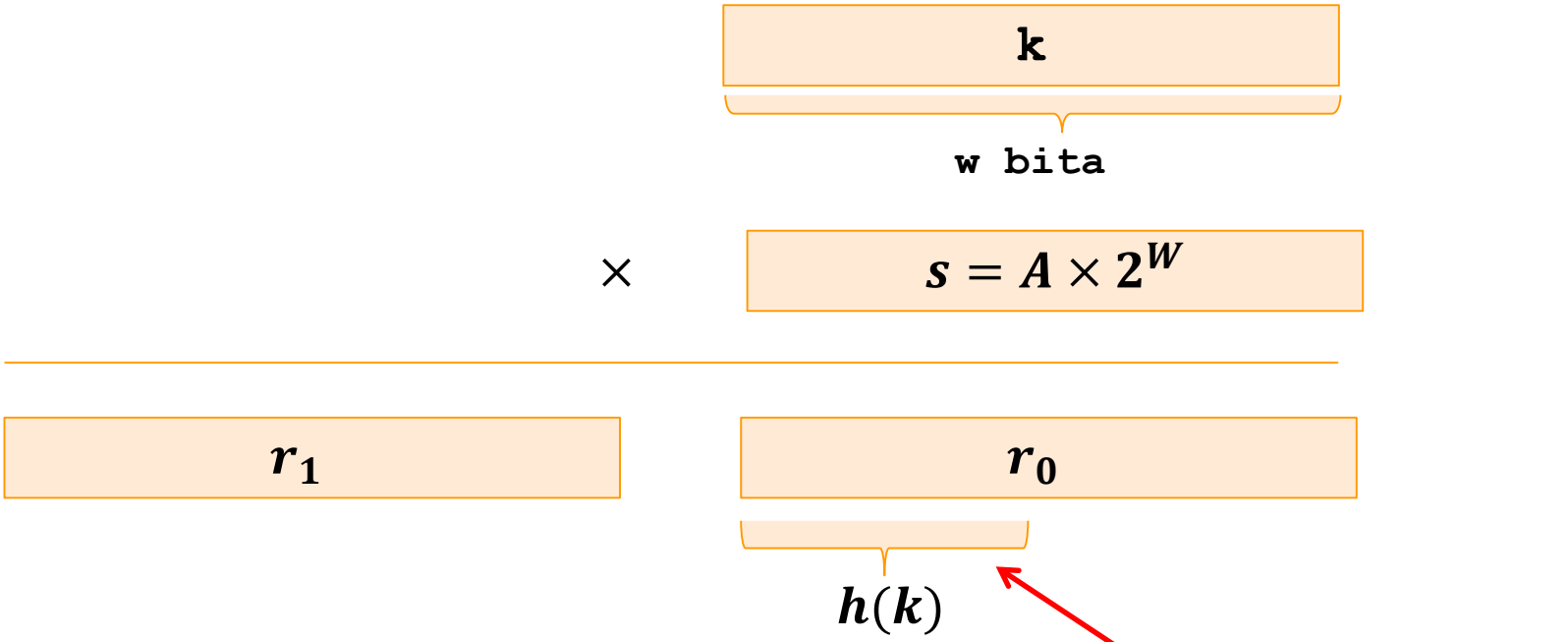
Osmišljavanje hash-funkcija

- U praksi se koriste razne metode prilikom kreiranja hash funkcije
- Metoda dijeljenja: $h(k) = k \; mod \; m$
  - Indeks ključa je jednak ostatku dijeljenja ključa veličinom tablice
  - Biramo m takav da ne bude potencija broja dva
  - Naime, ako je $m = 2^r$, tada hash funkcija ovisi samo o r bitova ključa
  - Npr. za $k = 11010111101101\underline{00}_2$ i $r = 4 \;\; h(k) = 0100_2$
  - Izabiremo m takav da je prost broj i da nije blizak potenciji broja 2 ili 10

Osmišljavanje hash-funkcije

- Metoda množenja: $h(k) = \lfloor m(kA \bmod 1) \rfloor$
  - množimo k s konstantom $0 < A < 1$ i uzimamo ostatak dijeljenja kA s 1 (ovo se zapravo odnosi na dio iza decimalne točke)
  - Tu vrijednost množimo s m i uzimamo najveće cijelo dobivenog rezultata
  - Prednost što ne trebamo paziti da m bude prost broj – obično biramo m takav da je potencija broja dva
  - Pretpostavimo da naše računalo radi s riječima veličine w bita
  - Ograničavamo A da bude razlomak veličine $s/2^w$, gdje je s cijeli broj raspona $0 < s < 2^w$ te množimo ključ k sa s
  - Produkt je 2w-bitna riječ $r_1 \times 2^w + r_0$, gdje se r_1 odnosi na prvih w znamenki riječi, a r_0 na drugih w znamenki
  - Željena p-bitna hash vrijednost sadrži p vodećih bita r_0 dijela produkta

Osmišljavanje hash-funkcija




```
      10101    k
×     11001    s
----------------
1000001101
```

r_1 r_0

Primjer funkcije raspršenog adresiranja za metode dijeljenja i množenja

```
//metoda dijeljenja
int HashDivisionMethod(int key) {
   return key % M;
}

//metoda množenja
int HashMultiplicationMethod(int key) {
    unsigned int A = 2654435769; // Knuth
    unsigned int shift= 29;
   return (A * key) >> shift;
}
```

HashQuadraticProbing.cpp

HashDoubleHashing.cpp

Primjer transformacije ključa u adresu

- 6 znamenkasti ključ, 7000 pretinaca; ključ: 172148
- metoda: središnje znamenke kvadrata ključa
  - kvadrat ključa daje 12 znamenkasti broj. Koriste se 5. do 8. znamenka

    $172148^2 = 029634933904$
  - središnje 4 znamenke treba transformirati u interval [0, 6999]
    - budući da pseudo-slučajni broj poprima vrijednosti iz intervala [0, 9999], a adrese pretinaca su iz intervala [0, 6999], faktor kojim ga se množi je $6999/9999 \approx 0.7$
  - adresa pretinca = 3493 * 0.7 = 2445
  - rezultati odgovaraju onima za *roulette*

Metode transformacije ključa u adresu

- korijen iz središnjih znamenki kvadrata ključa
  - za prethodni primjer nakon kvadriranja se izvadi korijen iz 8 središnjih znamenki, odbace se decimale da bi se dobio četveroznamenkasti broj:
  - sqrt(96349339) = 9815
- dijeljenje
  - ključ se dijeli s prim brojem približno jednakim broju pretinaca (npr. 6997)
  - ostatak dijeljenja je adresa pretinca
    - adresa pretinca = 172148 mod (6997) = 4220
  - dobro se raspoređuju ključevi koji su u nizu
- posmak znamenki i zbrajanje
  - npr. ključ = 1720
    - 1720 + 7359 = 9079

Metode transformacije ključa u adresu

- preklapanje
  - Preklapanje je slično posmaku, ali je prikladnije za dugačke ključeve
  - npr. ključ = 172407359
    - **407 + 953 + 271 = 1631**
- izmjena baze brojanja
  - broj se izračuna kao da ima drugu bazu brojanja B
  - npr. B = 11, ključ = 172148
    - $1*11^5 + 7*11^4 + 2*11^3 + 1*11^2 + 4*11^1 + 8*11^0 = 266373$
  - odabere se potreban broj najmanje značajnih znamenki i transformira u raspon adresa: adresa pretinca = 6373 * 0.7 = 4461
- najbolji postupak se postiže simulacijom za konkretnu primjenu
  - dijeljenje je općenito najbolje

Primjer funkcije raspršenog adresiranja s metodom preklapanja

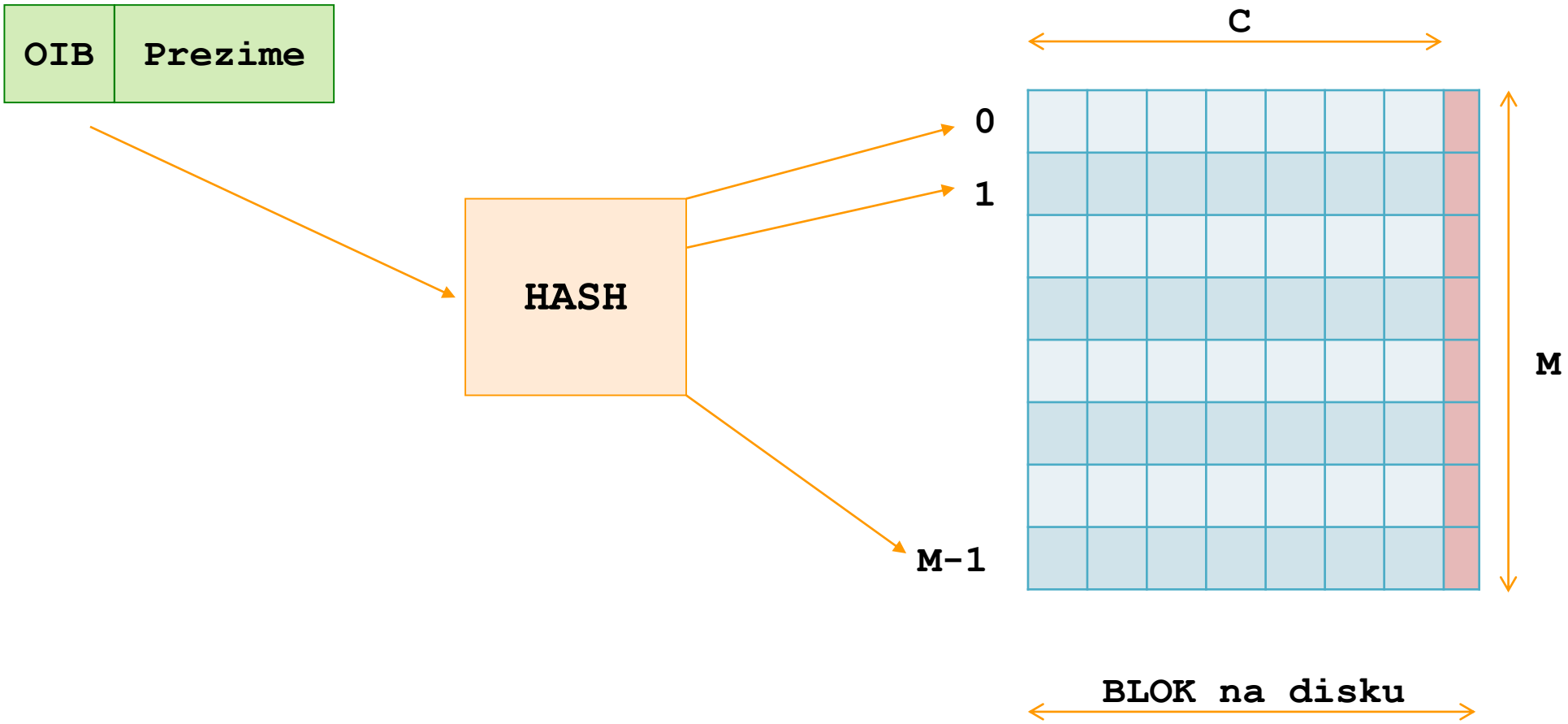
```
int HashIntOverlap(int key) {
   int sum = 0;
   char result[2] = {0};
   for (int i = 0; i < 3; i++) { // split 6-digit number into 3 parts
      if (i != 1) {
         result[0] = key % 10;
         key = key / 10;
         result[1] = key % 10;
         key = key / 10;
         sum += result[0] * 10 + result[1];
      } else {
         result[0] = key % 10;
         key = key / 10;
         result[1] = key % 10;
         key = key / 10;
         sum += result[1] * 10 + result[0];
      }
   }
   return sum;
}
```

HashLinearProbing.cpp

Raspršeno adresiranje u datoteci

- Problem: tvrtka ima 100 tisuća zaposlenih, svaki ima jedinstveni matični broj koji se generira iz intervala [0, 1 000 000]. Zapise treba brzo pohraniti i brzo im pristupiti. Kako organizirati datoteku?
  - direktna datoteka gdje je ključ jednak matičnom broju?
    - 1 000 000 x 4 okteta ~ 4MB, 90% je neiskorišteno!
- moguće je propisati postupke za transformaciju ključa u adresu, ili, što je još bolje, u neki redni broj
  - pod tim rednim brojem nalazi se zapisan položaj zapisa
  - ova modifikacija poboljšava fleksibilnost
    - gornji primjer pokazuje da dio prostora za zapise ostaje neiskorišten
- raspršeno adresiranje (*hashing*)

Upisivanje zapisa u pretince tablice raspršenih adresa

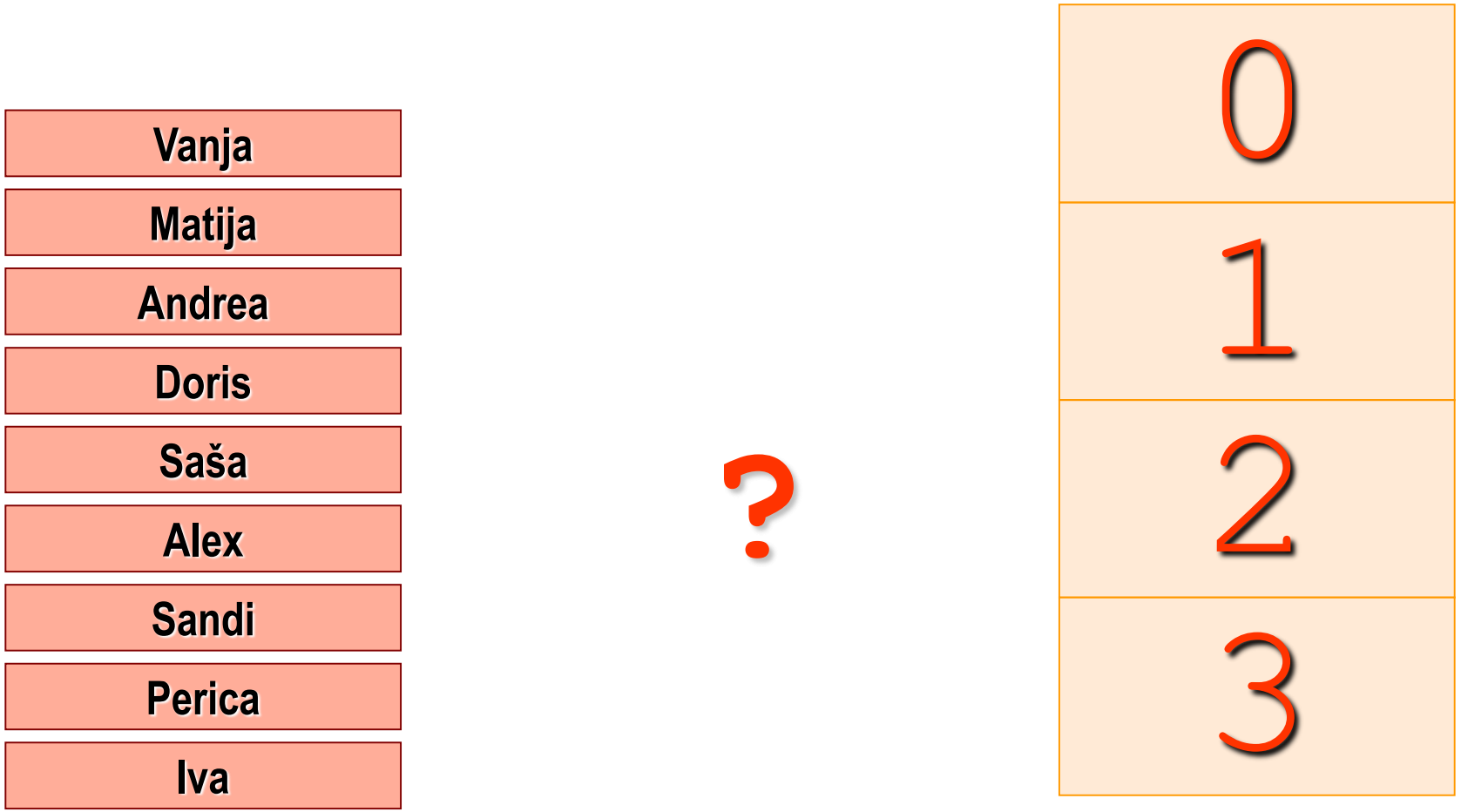


Raspršeno adresiranje u datoteci

- neka nam je na raspolaganju M pretinaca
- iz vrijednosti ključa pomoću hash-funkcije izračunava se pseudo-slučajni broj iz intervala od 0 do M-1
- taj broj je adresa grupe podataka (pretinca) koji svi daju isti pseudo-slučajni broj
  - kolizijom nazivamo slučaj kad se dva različita ključa transformiraju u istu adresu
  - ako se neki pretinac popuni, može se u njega upisati pokazivač na preljevno područje ili se prelazi na susjedni pretinac - preljev
- kod primjene raspršenog adresiranja variranju su podložni sljedeći elementi:
  - kapacitet pretinca
  - gustoća pakiranja

Primjer

- imena pohraniti u hash-tablicu raspršenim adresiranjem
  - hash-fukcija - suma ASCII kodova slova % veličina tablice



Kapacitet pretinca

- transformacijom ključa nastaje pseudo-slučajni broj koji daje adresu pretinca
  - ako je kapacitet pretinca jednak 1, čest je slučaj preljeva
  - što je veći kapacitet pretinca, preljev je manje vjerojatan, ali je čitanje pojedinog pretinca dulje i raste potreba za slijednim pretraživanjem unutar pretinca
- povoljno je veličinu pretinca uskladiti s fizičkom veličinom zapisa na vanjskoj memoriji (blok)

Gustoća pakiranja

- nakon što je odabrana veličina pretinca, može se odabrati gustoća pakiranja, tj. broj takvih pretinaca za pohranjivanje predviđenog broja zapisa
  - da bi se smanjio broj preljeva, izabire se veći kapacitet
  - gustoća pakiranja = broj zapisa / ukupni kapacitet
    - N = broj zapisa koje treba pohraniti
    - M = broj pretinaca
    - C = broj zapisa u jednom pretincu

- gustoća pakiranja = N / (M *C)

Postupak s preljevom

- korištenje primarnog područja
  - ako je neki pretinac popunjen, koristi se sljedeći itd.
  - iza zadnjega ciklički dolazi prvi
  - postupak je efikasan kod veličina pretinca iznad 10
- ulančavanje
  - pretinci su organizirani kao linearne liste

Statistika kod raspršenog adresiranja

- Neka je M broj pretinaca, a N broj ulaznih podataka. Vjerojatnost da će u neki pretinac biti upućeno x zapisa ravna se po binomnoj razdiobi:

$$P(x) = \frac{N!}{x!\cdot(N-x)!} \cdot \left(\frac{1}{M}\right)^x \left(1-\frac{1}{M}\right)^{N-x}$$

  - vjerojatnost da će biti Y preljeva: P(C + Y)
  - očekivani broj preljeva iz zadanog pretinca:

$$s = \sum_{Y=1}^{\infty} \left(P(C+Y)\,Y\right)$$

- ukupni očekivani postotak preljeva: 100∗s ∗M/N
- prosječni broj zapisa koji će biti upisan u hash-tablicu prije nego dođe do kolizije je ~ 1.25 √M
- prosječni broj ukupno upisanih zapisa prije nego što svaki pretinac sadrži bar 1 upisani zapis je M ln M

Određivanje parametara

- primjer:
  - Na Fakultetu ima oko 350 studenata. Treba pohraniti njihov OIB (11 znakova) i prezime (14 znakova), s tim da ih se može pronalaziti brzo po OIB-u.
- napomena:
  - OIB se sastoji od 11 znamenaka
    - zadnja znamenka je kontrolni broj i može se, ali i ne mora pohranjivati ako se zna pravilo po kojem je izračunata

Rješenje

- jedan zapis sadrži 11+1 + 14+1 = 27 okteta
- fizički blok na disku neka je veličine 512 okteta
  - veličina pretinca trebala bi biti manja ili jednaka tom iznosu
    - 512/27 = 18,963
  - slijedi da će pretinac sadržavati podatke o 18 studenata i 26 okteta neiskorištenog prostora
  - predvidjet će se nešto veći kapacitet tablice kako bismo smanjili broj očekivanih preljeva, npr. za 30%
    - to znači da ima $\lfloor 350/18 \rfloor$ *1.3 = 25 pretinaca
    - OIB treba transformirati u adresu pretinca iz intervala [0, 24]
  - OIB je vrlo dugačak pa bi došla u obzir tehnika preklapanja
    - postupci se mogu i kombinirati - nakon preklapanja obaviti dijeljenje
      - adresa će se računati dijeljenjem s prim brojem bliskim broju pretinaca, npr. 23

Primjeri transformacije ključa

```
OIB = 5702926036x
         2926
          630
          075
        3631 mod (23) = 20
```

```
OIB = 6702926036x
         2926
          630
          076
        3632 mod (23) = 21
```

```
OIB = 6702926037x
         2926
          730
          076
        3732 mod (23) = 6
```

```
OIB = 5702926037x
         2926
          730
          075
        3731 mod (23) = 5
```

- ako pretinac bude popunjen, prelazi se na susjedni pretinac, ciklički (*bad neighbour policy*)

Programsko rješenje

```
Kreiraj na disku praznu tablicu
Čitaj slijedno OIB i prezime, dok ima podataka
  Ako je kontrolna znamenka nije ispravna
    "Neispravan OIB"
  Inače
    Stavi oznaku da zapis nije upisan
    Izračunaj adresu pretinca
    Upamti izračunatu adresu kao početnu
    Ponavljaj
      Čitaj iz pretinca upisane zapise
      Ponavljaj za sve zapise iz pretinca
        Ako zapis nije prazan
          Ako je upisani OIB identičan ulaznom
            "Zapis vec postoji"
            Stavi oznaku da je zapis upisan
            Skok iz petlje
        Inače
```

Programsko rješenje

```
        Upiši ulazni zapis
        Stavi oznaku da je zapis upisan
        Skok iz petlje
    Ako zapis nije upisan
        Povećaj adresu pretinca za 1 i izračunaj mod
            (broja pretinaca)
        Ako je dobivena adresa jednaka početnoj adresi
            Tablica je puna
Dok ne bude zapis upisan ili tablica puna
Kraj
```

HashInFile.cpp

Korištenje raspršenog adresiranja

- kad je prikladno?
  - prevoditelji ga koriste za evidenciju deklariranih varijabli
  - za provjeru teksta (*spelling checker*) i rječnike
  - npr. u igrama za pohranu položaja igrača
  - za provjeru jednakosti
    - ako dva elementa daju različite *hash* vrijednosti, sigurno su različiti
  - kad postoji potreba za brzim, a čestim pretraživanjem
- kad nije prikladno?
  - kad se podatke pretražuje po vrijednosti pojma koji nije ključ
  - kad se traži da podaci budu sortirani
    - npr. kad treba pronaći najmanji ključ

Zadaci za vježbu

- Napisati funkciju koja će u neformatiranoj datoteci artikli organiziranoj po načelu raspršenog adresiranja prebrojiti koliko ima upisanih zapisa o artiklima. Jedan zapis sadrži šifru (cijeli broj), naziv (50+1 znakova), količinu (cijeli broj) i cijenu (realni broj). Zapis je prazan ako je na mjestu šifre vrijednost nula. Veličina fizičkog bloka na disku je BLOK, a očekivani maksimalni broj zapisa je MAXZAP. Ovi su parametri upisani u parametri.h.

- Napisati funkciju koja će u neformatiranoj datoteci organiziranoj po načelu raspršenog adresiranja odrediti gustoću pakiranja. Jedan zapis sadrži naziv (50+1 znakova), količinu (cijeli broj) i cijenu (realni broj). Zapis je prazan ako je na mjestu količine vrijednost nula. Veličina fizičkog bloka na disku je BLOK, što je definirano u parametri.h. Prototip funkcije je:

```
float gustoca (const char *ime_datoteke);
```

Zadaci za vježbu

- Napisati funkciju za upis šifre (cijeli broj) i naziva (20+1) u memorijski rezidentnu tablicu raspršenih adresa s 500 pretinaca. Pretinac sadrži jedan zapis. Ako je pretinac popunjen, prelazi se ciklički na susjedni. Ulazni argumenti su već izračunata adresa pretinca, šifra i naziv. Funkcija vraća vrijednost 1 ako je upis obavljen, 0 ako podatak već postoji, a -1 ako je tablica popunjena pa se podatak nije mogao upisati.

- Napisati funkciju za pronalaženje šifre (cijeli broj) i naziva poduzeća (30+1) iz memorijski rezidentne tablice raspršenih adresa s 200 pretinaca. Pretinac sadrži jedan zapis. Ako je pretinac popunjen, a ne sadrži traženu vrijednost ključa, prelazi se ciklički na susjedni. Ulazni argumenti su već izračunata adresa pretinca i šifra. Izlazni argument je naziv poduzeća. Funkcija vraća vrijednost 1 ako je zapis pronađen, a 0 ako nije.

Zadaci za vježbu

- Napisati funkciju za transformaciju ključa koji je telefonski broj od 7 znamenki u raspršene adrese. Tablica raspršenih adresa sadrži M pretinaca. Koristiti postupak dijeljenja. Prototip funkcije je:

```
int adresa (int m, long telef);
```

- Napisati funkciju za pražnjenje neformatirane datoteke artikli organizirane po načelu raspršenog adresiranja. Jedan zapis sadrži šifru (četveroznamenkasti cijeli broj), naziv (do 30 znakova) i cijenu (realni broj). Zapis je prazan ako je na mjestu šifre vrijednost nula. Veličina fizičkog bloka na disku je BLOK, a očekivani maksimalni broj zapisa je MAXZAP. Ovi su parametri upisani u parametri.h.

- Napisati funkciju za izračun adrese pretinca u tablici s 500 pretinaca. Ključ je šifra od 4 znamenke, a metoda je korijen iz srednjih znamenki kvadrata.