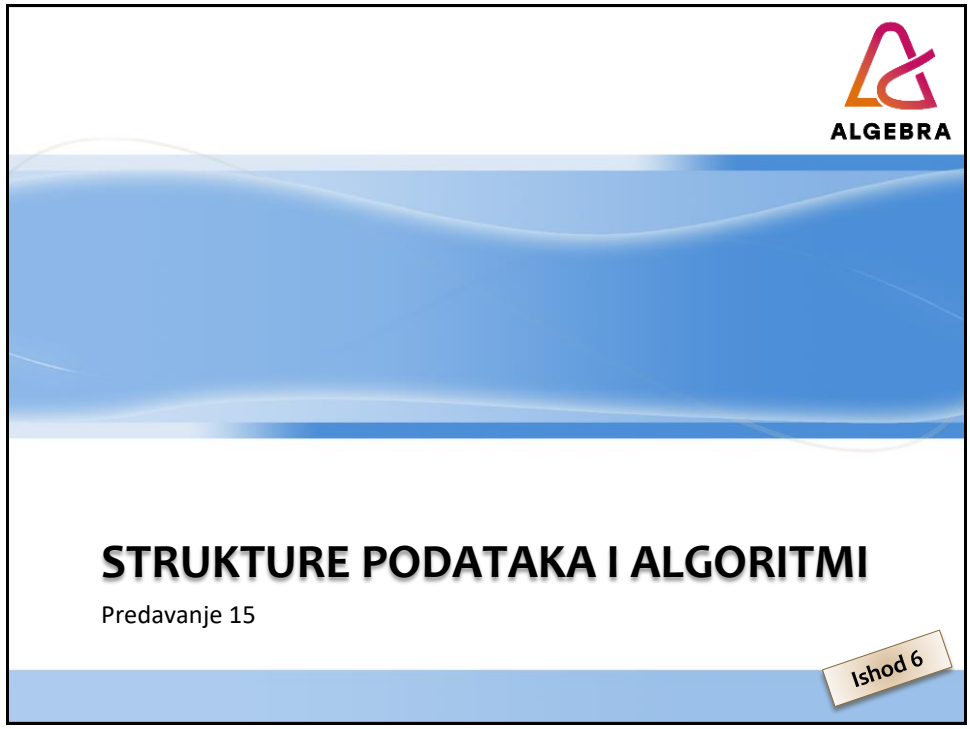
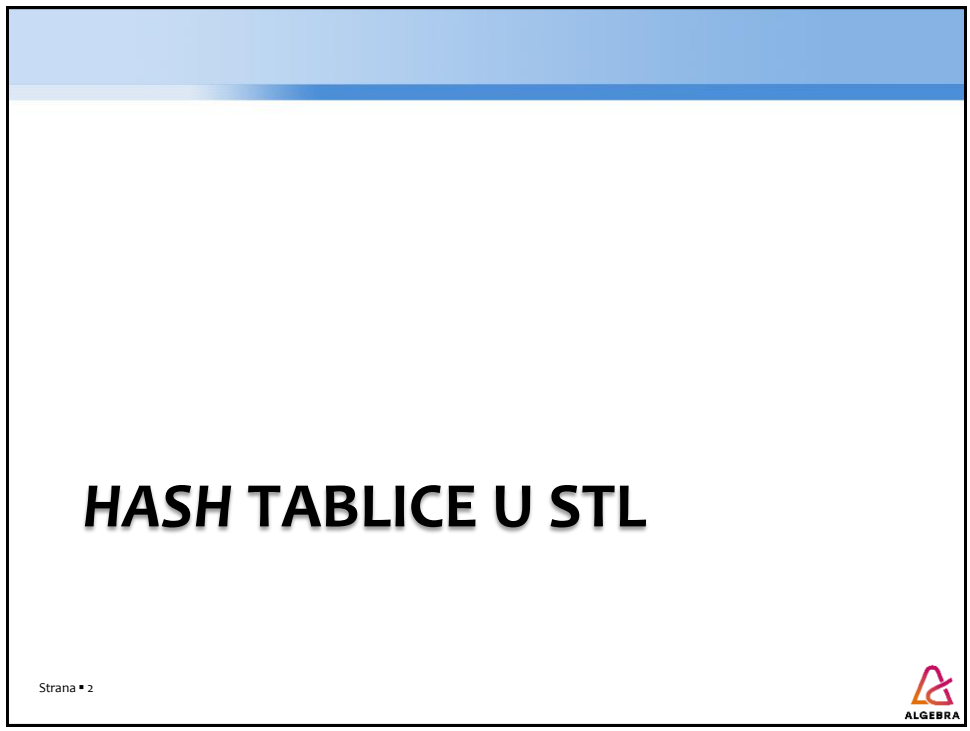

ALGEBRA
STRUKTURE PODATAKA I ALGORITMI
Predavanje 15
Ishod 6

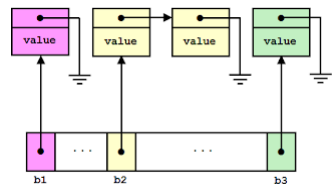
1


HASH TABLICE U STL
Strana ▪ 2
ALGEBRA

2

Uvod

- C++ standard dopušta bilo kakvu implementaciju *hash* tablica, sve dok vrijedi sljedeće:
  - *Defaultni* maksimalni faktor opterećenja mora biti 1.0
  - *Hash* tablica garantirano neće rasti sve dok faktor opterećenja ne probije maksimalni faktor opterećenja
- Rezultat je taj da sve implementacije koriste ulančavanje kao metodu rješavanja kolizija



Slika preuzeta s: bannalia.blogspot.hr

ALGEBRA

3

Problem s klasičnim ulančavanjem

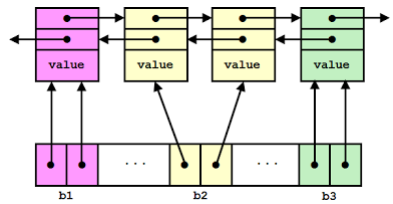
- Klasično ulančavanje je pristup u kojem svaki *slot* ima svoju vezanu listu
- Ovaj pristup ima problem:
  - Standard zahtijeva da *hash* tablica definira iterator
  - Standard zahtijeva da inkrement iteratora bude $O(1)$
- Gornji zahtjev se ne može postići s klasičnim ulančavanjem
  - Rješenje je da svi elementi iz svih *bucketa* budu međusobno povezani

ALGEBRA

4

Poboljšano ulančavanje

- Implementacija u Visual Studiju koristi jednu dvostruko povezanu listu za sve *buckete*
  - Svaki element zna za prethodni i sljedeći element
  - Svaki *bucket* čuva dva pokazivača: na prvi i na zadnji element u *bucketu*



Slika preuzeta s: bannalia.blogspot.hr

ALGEBRA

Umetanje

- Umetanje ima složenost $O(1)$:
  1. *Hashiraj* ključ da bi pronašao *bucket*
     - $O(1)$
  2. Ako ključ već postoji u *bucketu*, odustani
     - $O(k)$, gdje je k broj elemenata u *bucketu*
     - Dobra *hash* funkcija i dobar najveći faktor opterećenja garantiraju da će k biti oko 1
  3. Dodaj element na početak liste
     - $O(1)$
  4. Podesi pokazivače u *bucketu*
     - $O(1)$

ALGEBRA

Brisanje

- Brisanje ima složenost $O(1)$:
  1. *Hashiraj* ključ da bi pronašao *bucket*
     - $O(1)$
  2. Pronađi element u *bucketu*
     - $O(k)$, gdje je k broj elemenata u *bucketu*
  3. Ukloni element iz liste
     - $O(1)$
  4. Podesi pokazivače u *bucketu* ako treba
     - $O(1)$

Klasa unordered_map

- Glavna klasa koja implementira *hash* tablicu u STL-u je `unordered_map`
- Definicija glasi ovako:

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = equal_to<Key>,
          class Alloc = allocator< pair<const Key,T> >
          > class unordered_map;
```

- Obavezni parametri su Key i T
  - Key predstavlja tip ključa
  - T predstavlja tip vrijednosti

Osnovna izrada *hash* tablice

- Osnovni načini izrade *hash* tablice su:
  - `unordered_map<int, string> jedan;`
    - Kreira praznu *hash* tablicu s ključem `int` i vrijednosti `string`
  - `unordered_map<int, string> dva(n);`
    - Kreira *hash* tablicu s ključem `int` i vrijednosti `string` bez elemenata, ali s minimalno *n* praznih *bucketa*
  - ```
    unordered_map<int, string> tri = {
        { 1, "Miro" },
        { 2, "Ana"  }
    };
    ```
    - Kreira *hash* tablicu s ključem `int` i vrijednosti `string` te s dva elementa

Iteratori *hash* tablice

- Postoje dva moguća postupka iteriranja elemenata:
  - Iteriranje elemenata u *bucketu b*
  - Iteriranje svih elemenata u *hash* tablici
- Kod ovih postupaka koriste nam metode:
  - `ht.begin()` vraća iterator na prvi element u *hash* tablici
  - `ht.begin(b)` vraća iterator na prvi element u *bucketu b*
    - *Bucket b* je cijeli broj u rasponu [0, `bucket_count()`)
  - `ht.end()` vraća iterator na zadnji element u *hash* tablici
  - `ht.end(b)` vraća iterator na zadnji element u *bucketu b*
    - *Bucket b* je cijeli broj u rasponu [0, `bucket_count()`)


    ```

Primjer

```
unordered_map<int, string> ht = {
   { 1, "Miro" },
   { 2, "Ana"  },
   { 3, "Petra" },
   { 4, "Janko" },
   { 5, "Branka" },
};

for (auto it = ht.begin(); it != ht.end(); ++it) {
   cout << it->second << endl;
}
cout << endl;

for (int i = 0; i < ht.bucket_count(); i++) {
   cout << "Bucket: " << i << ": ";
   for (auto it = ht.begin(i); it != ht.end(i); ++it) {
      cout << it->second << " ";
   }
   cout << endl;
}
```


Par ključ i vrijednost

- Iterator je pokazivač na par (ključ, vrijednost)
- Struktura `pair<T1,T2>` predstavlja par
  - T1 predstavlja tip ključa
  - T2 predstavlja tip vrijednosti
  - Objekt te strukture na sebi sadrži članove:
    - `first` predstavlja ključ
    - `second` predstavlja vrijednost
- Primjer:

```
pair<int, string> p(99, "Ivana");
cout << p.first << endl;        // Ispisuje 99
cout << p.second << endl;       // ispisuje Ivana
```


Izravno pristupanje elementima

- Osim pomoću iteratora, elementima možemo pristupiti i izravno na sljedeća dva načina:
  - `ht[key]`
    - Ako key postoji, vraća vrijednost
    - Ako key ne postoji, umeće praznu vrijednost s tim ključem i vraća je
  - `ht.at(key)`
    - Ako key postoji, vraća vrijednost
    - Ako key ne postoji, baca iznimku

Primjer

```
unordered_map<int, string> ht({
   { 1, "Ana"   },
   { 2, "Juro"  },
   { 3, "Marko" }
});

cout << ht[1] << endl;
cout << ht[2] << endl;
cout << ht[3] << endl;
cout << ht[4] << endl;

cout << ht.size() << endl;
```


Umetanje u *hash* tablicu

- Najvažniji načini umetanja su sljedeći:
  - `ht.insert(pair<int, string>(99, "Ivana"));`
    - Umeće zadani par i vraća objekt tipa `pair<iterator, bool>`
      - `first` pokazuje ili na friško umetnuti element ili na ekvivalentni element koji već postoji
      - `second` sadrži `true` (ako je umetanje uspjelo) ili `false` (ako je ekvivalentni element već postojao)
  - `ht.insert({ 99, "Ivana" });`
    - Jednako kao prethodno, ali s ljepšom sintaksom
  - `ht.insert({{ 99, "Ivana" }, { 118, "Jurica" }});`
    - Umeće zadane parove, ali ne vraća ništa

Primjer

```
unordered_map<int, string> ht;

auto it = ht.insert(pair<int, string>(99, "Ivana"));
cout << "Umetanje uspjelo: " << it.second << endl;

it = ht.insert(pair<int, string>(99, "Ivana"));
cout << "Umetanje uspjelo: " << it.second << endl;

cout << ht.size() << endl;

ht.insert({ { 99, "Marija" }, { 118, "Jurica" } });
cout << ht.size() << endl;

for (auto it = ht.begin(); it != ht.end(); ++it) {
   cout << it->first << " " << it->second << endl;
}
```


Brisanje iz *hash* tablice

- Brisanje možemo raditi na sljedeće načine:
  - `ht.erase(iterator)` briše element na zadanoj poziciji
    - Vraća iterator na prvi sljedeći element iza obrisanog
  - `ht.erase(key)` briše element sa zadanim ključem
    - Vraća broj obrisanih elemenata (dakle, 0 ili 1)
  - `ht.erase(begin, end)` briše sve elemente u rasponu
    - Vraća iterator na prvi sljedeći element iza zadnje obrisanog
  - `ht.clear()` uklanja i uništava sve elemente

Primjer

```
unordered_map<int, string> ht({
   { 1, "Ana" },
   { 2, "Juro" },
   { 3, "Marko" }
});

cout << ht.erase(2) << endl;
cout << ht.erase(2) << endl;

for (auto it = ht.begin(); it != ht.end(); ++it) {
   cout << it->first << " " << it->second << endl;
}
```


Ostale važnije metode

- `ht.find(key)`
  - Vraća iterator na element s ključem key
  - Vraća `ht.end()` ako ne postoji
- `ht.size()`
  - Vraća broj elemenata u *hash* tablici
- `ht.empty()`
  - Vraća je li *hash* tablica prazna

Operacije nad *bucketima*

- Operacije specifične za *buckete* su:
  - `ht.bucket_count()`
    - Vraća broj *bucketa* u *hash* tablici
  - `ht.bucket_size(b)`
    - Vraća broj elemenata u *bucketu b*
  - `ht.bucket(key)`
    - Vraća broj *bucketa* u kojeg se *hashira* key
    - key može ili ne mora postojati

Zadatak

1. Datoteke mjestaRh_1.csv i mjestaRh_2.csv sadrže podatke o naseljima. Između tih datoteka postoje neka preklapanja (neka naselja se nalaze u obje datoteke). Što će se dogoditi kad obje datoteke učitamo u istu *hash* tablicu? Ispišimo sadržaj svih *bucketa*.

Rješenje – učitavanje

```
void napuni(ifstream& dat, unordered_map<string, string>& naselja) {
   string temp;
   getline(dat, temp);

   string key;
   string val;
   while (true) {
        if (!getline(dat, key, ';')) {
            return;
        }

        getline(dat, val);

        naselja.insert({ key, val });
   }
}
```


Rješenje – main

```
unordered_map<string, string> naselja;

ifstream dat1("mjestaRh_1.csv");
ifstream dat2("mjestaRh_2.csv");
if (!dat1 || !dat2) {
    cout << "Greska pri otvaranju datoteka" << endl;
    return 1;
}

napuni(dat1, naselja);
napuni(dat2, naselja);

for (unsigned i = 0; i < naselja.bucket_count(); i++) {
    cout << "Bucket " << i << ": ";
    for (auto it = naselja.begin(i); it != naselja.end(i); ++it) {
        cout << it->first << "-" << it->second << " ";
    }
    cout << endl;
}
```


Pravila oko *hashiranja* (1/2)

- `load_factor()`
  - Vraća trenutni faktor opterećenja
  - Faktor opterećenja je omjer broja elemenata i broja *bucketa*
- `max_load_factor()`
  - Vraća najveći faktor opterećenja (*defaultno* 1.0)
- Kad trenutni faktor pretekne najveći faktor, dešava se operacija preraspršivanja (engl. *rehash*):
  - Broj *bucketa* u *hash* tablici raste
  - Mijenja se *hash* funkcija kako bi uzela u obzir novi broj *bucketa*
  - Postojeći elementi se ponovno razmještaju po *bucketima*

Primjer

```
unordered_map<int, string> ht({
    { 1, "Ana" },
    { 2, "Juro" },
    { 3, "Marko" }
});

for (int i = 1; i <= 100; i++) {
    ht.insert({ i, "dummy" });
    cout << "n=" << ht.size();
    cout << ", m=" << ht.bucket_count();
    cout << ", a = " << ht.load_factor();
    cout << "/" << ht.max_load_factor() << endl;
}
```


Pravila oko *hashiranja* (2/2)

- `rehash(n)`
  - Postavlja broj *bucketa* na najmanje *n*:
    - Ako je *n* veći on trenutnog broja *bucketa*, slijedi *rehash*
    - Ako je *n* manji, vjerojatno neće napraviti ništa

Primjer

```
void print(unordered_map<int, string>& ht) {
    for (unsigned i = 0; i < ht.bucket_count(); i++) {
        cout << "Bucket " << i << ": ";
        for (auto it = ht.begin(i); it != ht.end(i); ++it) {
            cout << it->first << "-" << it->second << " ";
        }
        cout << endl;
    }
    cout << "---" << endl;
}
int main() {
    unordered_map<int, string> ht({{1,"Ana"},{2,"Juro"},{3,"Iva"}});
    print(ht);
    ht.rehash(8);
    print(ht);
    ht.rehash(5);
    print(ht);
    ht.rehash(100);
    print(ht);
    return 0;
}
```


VARIJACIJE *HASH* TABLICA

Uvod

- *Hash* tablice su implementirane sljedećim STL klasama:
  - `unordered_map`
  - `unordered_multimap`
  - `unordered_set`
  - `unordered_multiset`

`unordered_multimap`

- `unordered_multimap` je verzija `unordered_map` gdje ključevi ne moraju biti jedinstveni
- Sučelje za korištenje je slično, uz glavnu razliku:
  - Nema `operator[]`
  - Nema metodu `at()`

unordered_set

- unordered_set je verzija unordered_map gdje vrijedi:
  - Ključ = vrijednost
  - Ključevi moraju biti jedinstveni

31

unordered_multiset

- unordered_multiset je verzija unordered_set gdje vrijedi:
  - Ključ = vrijednost
  - Ključevi ne moraju biti jedinstveni

32