Apstrakta struktura podataka - Skup

dr Davorka R. Jandrlić dr Goran Lazović

Mašinski fakultet, Univerziteta u Beogradu

ASP - Skup

Definicija

Skup je kolekcija različitih elemenata, od kojih je svaki ili skup ili primitivnog tipa (*atom*).

- Primitivni tipovi su obično neki od poznatih tipova: celi brojevi, realni brojevi, karakteri, itd. Pp. da su vrednosti primitivnog tipa uređeni.
- Skup se predstavlja na sledeći način: $\{a_1, a_2, \ldots, a_n\}$.
- Redosled elemenata u skupu nije bitan.
- Operacije: $\in, \cup, \cap, \subseteq, \setminus, \triangle$
- Prazan skup: $\emptyset$

ASP - Skup

ASP Skup predstavljanje

- **elementtype** bilo koji tip nad kojim je definisana relacija totalnog uređenja $\leqslant$.
- **Set** konačan skup međusobno različitih elemenata tipa **elementtype**

Operacije:

- MAKE_NULL(S) - pretvara skup u prazan
- INSERT(x, S) – ubacuje x u skup S ($S = S \cup x$), pod uslovom da takav element već ne pripada skupu S.
- DELETE(x, S) – izbacuje element x iz skupa S ($S = S \setminus x$), ako takav element postoji u skupu.
- MEMBER(x, S) – proverava da li je x element skupa S.
- MIN(S), MAX(S) – pronalazi najmanji odnosno najveći element skupa S u odnosu na definisanu relaciju $\leqslant$

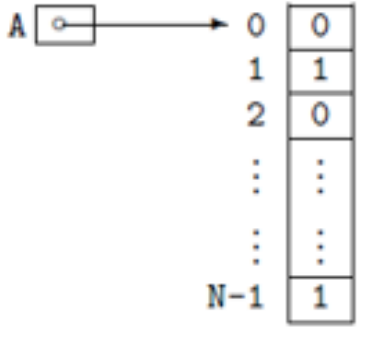
ASP - Skup

Operacije ...

- SUBSET(A, B) – proverava da li je skup A podskup skupa B.
- UNION(A, B)
- INTERSECTION(A, B)
- DIFFERENCE(A, B)

Implementacija Skupa pomoću bit vektora

- Izbor implementacije Skupa zavisi od operacija koje treba da se izvrše nad skupom, kao i od veličine skupa.
- Ako su vrednosti skupa iz nekog unapred zadatog univerzalnog skupa npr. celih brojeva od $1, ..., N$, za dato N, tada se za implementaciju skupa može koristiti **bit-vektor** (niz elemenata logičkog tipa (true-false, 1-0)).
- i-ti element vektora je "tačan" onda kada je i element skupa.



Predstavljanje Skupa bit vektorom

```
#define MAX_SET_ELEMENT 50
typedef int *Set; // Skup se definise kao niz
                  // celih brojeva ciji su
        // elementi 0 ili 1.

int member(int x, Set s) {
    return s[x];
}

void union(Set a, Set b, Set *rez) {
    int i;
    for(i = 0; i < N; i++) {
          (*rez)[i] = a[i] || b[i];
    }
}
```

Vremenska složenost

- Operacije MEMBER, INSERT i DELETE zahtevaju konstantno vreme, dok se operacije UNION, INTERSECTION, DIFFERENCE i SUBSET izvršavaju u vremenu $O(N)$, zato što je veličina svakog od skupova jednaka veličini univerzalnog skupa brojeva iz kojeg su elementi.
- Ova implementacija je neupotrebljiva za veliko N!

Implementacija Skupa pomoću uređene jednostruko povezane liste

- Očigledno je da se Skup može predstaviti kao povezana lista čiji su elementi elementi skupa.
- U ovakvoj implementaciji veličina liste jednaka je veličini skupa, a ne veličini univerzlalnog skupa brojeva kao što je slučaj sa implementacijom pomoću bit-vektora.
- Da bi se optimizovale operacije unije, preseka i razlike, ideja je da se lista "drži" sortiranom rastuće: npr. u slučaju preseka kada su obe liste neuređene, vreme potrebno za pronalaženje preseka je $O(n^2)$, dok je u slučaju uređenih listi vreme proporcionalno dužini jedne od lista (ZAŠTO!).

Primer

```
void presek(Skup s1, Skup s2, Skup *rez) {
     while(s1 != NULL && s2 != NULL) {
          if(s1 ->vrednost == s2 ->vrednost) {
             dodajNaKraj(rez, s1 ->vrednost);
             s1 = s1 ->sled;
             s2 = s2->sled;
          }
          else if(s1->vrednost < s2->vrednost) {
               s1 = s1->sled;
          }
          else {
               s2 = s2->sled;
          }
     }
}
```

ATP Rečnik

- ATP Skup nad kojim su definisane samo operacije INSERT, DELETE i MEMBER naziva se rečnik.
- Implementacija rečnika:
  - pomoću bit vektora
  - pomoću jednostruko povezane liste (ne mora biti uređena)
  - pomoću liste implementirane statičkim nizom. Niz je u tom slučaju najbolje urediti i čuvati ga u rastućem poretku. Tada je operaciju MEMBER moguće implementirati u vremenu $O(logn)$, korišćenjem binarne pretrage.
  - pomoću heš tabele,
  - pomoću binarnog stabla pretrage.

Heš funkcije i heš tabele

- Funkcija koja preslikava skup elemenata h :$\{0, 1, ..., M-1\}$ u skup $\{0, 1, ..., m-1\}$, takav da je $m \leqslant M$, naziva se **heš funkcija**.
- **Heš tabela** je struktura podataka sastavljena od m ćelija koje se nazivaju posude (kofe). Heš tabela koristi heš funkciju koja preslikava identifikatore (ključeva) u pridružene vrednosti. Heš tabele se ponašaju kao i asocijativni nizovi.

Primer

Treba sačuvati podatke o 250 studenata, pri čemu se svaki od njih identifikuje svojim matičnim brojem od 13 cifara. Struktura podataka u kojoj bi studenti mogli da se "drže" (čuvaju/skladište), a koja bi omogućila efikasno dohvatanje i pretragu, mogao bi da bude vektor (niz) od 1000 elemenata takav da se na poziciji i nalazi onaj student čiji se su poslednje tri cifre matičnog broja jednake broju i. Metod nije potpuno pouzdan jer je moguće da dva studenta imaju poslednje tri cifre matičnog broja jednake, pa će se desiti takozvana kolizija.

Primer

- U primeru heš funkciju predstavlja funkcija koja izdvaja broj sačinjen od poslednje tri cifre matičnog broja, dok posude heš tabele predstavljaju indeksi niza.
- Heš funkcija dakle preslikava matični broj u indeks niza koji određuje posudu u kojoj se element (student) nalazi.

- U opštem slučaju, ako je veličina heš tabele nedovoljno velika, desiće se da različitim ključevima odgovaraju iste lokacije (posude). Ovakav događaj naziva se **kolizija**.
- Potrebno je rešiti dva problema:
  - pronaći heš funkciju koja minimizuje verovatnoću pojave kolizija
  - definisati postupak za obradu kolizija kada do njih dođu.
- Dobra heš funkcija preslikava ključeve ravnomerno u skup slučajnih lokacija po tabeli.

Heš funkcija i obrada kolizija

Izbor heš funkcije

- Ako je veličina tabele m prost broj, a ključevi su celi brojevi, onda je prosta i dobra heš funkcija data izrazom:

$$h(x) = x \mod m$$

Obrada kolizija - odvojeno nizanje

Najjednostavniji način je **odvojeno nizanje** *(otvoreno heširanje)*:

- svaki element heš tabele je pokazivač na povezanu listu (koja sadrži sve ključeve smeštene na tu lokaciju u tabeli).
- traženje određenog elementa:
  - prvo se primeni heš funkcija da bi se pronašla posuda u kojoj se element nalazi,
  - zatim se linearno pretražuje lista.

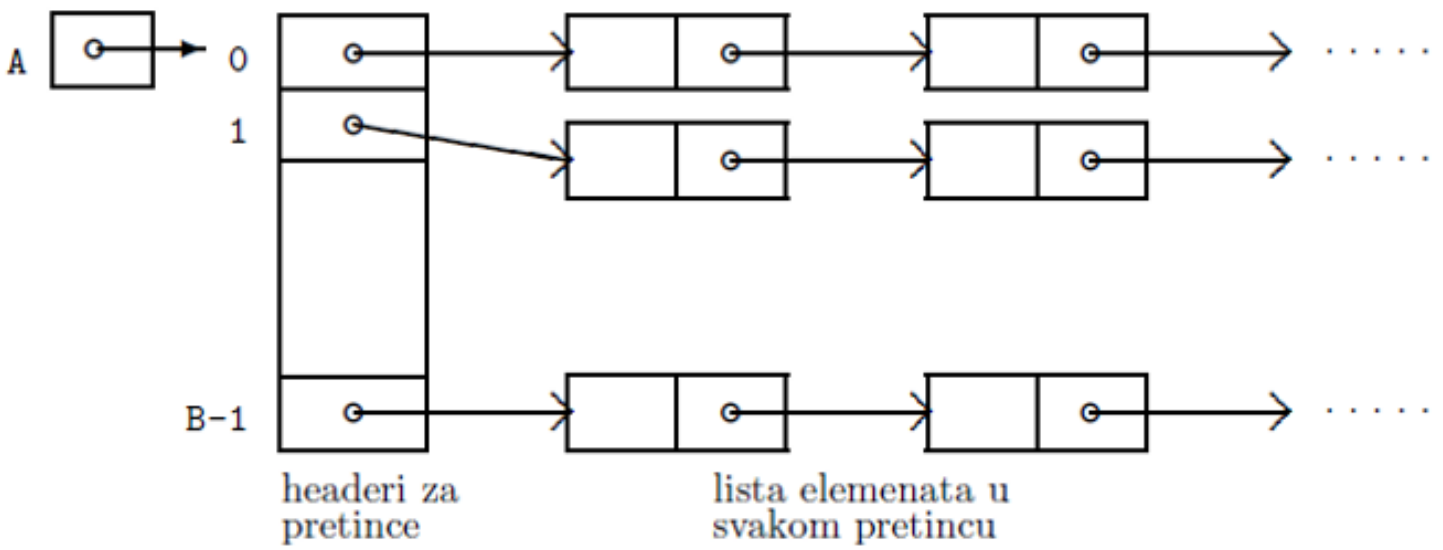
Obrada kolizija

Nedostaci odvojenog nizanja

Postupak je neefikasan ako su povezane liste dugačke!!

Do toga nužno dolazi kada je:

- tabela mala u odnosu na broj ključeva,
- izabrana je loša heš funkcija.



Obrada kolizija

Linearno popunjavanje

- Drugo jednostavno rešenje je **linearno popunjavanje** *(zatvoreno heširanje)*:
  - ako je lokacija sa adresom $h(x)$ već zauzeta, onda se novi element sa ključem x zapisuje na susednu lokaciju $h(x) + 1$ mod m (lokacija koja sledi iza m–1 je 0).
  - Prilikom traženja zadatog ključa x, neophodno je linearno pretraživanje počevši od lokacije $h(x)$ pa do prve nepopunjene lokacije.
- Rešenje je efikasno ako je tabela relativno retko popunjena.
- U protivnom će dolaziti do mnogo sekundarnih kolizija, odnosno kolizija prouzrokovanih ključevima sa različitim vrednostima heš funkcije.

Obrada kolizija

Heš tabele

- Kod obe varijante heš tabele važno je da tabela bude dobro dimenzionisana u odnosu na rečnik koji se u nju smešta.
- Neka je n broj elemenata u rečniku. Preporučuje se da
  - kod otvorenog heširanja bude $n \leqslant 2 \cdot B$, gde je B broj posuda u tabeli,
  - a kod zatvorenog heširanja $n \leqslant 0.9 \cdot B$.
- Ako je to ispunjeno, tada se može očekivati da bilo koja od operacija INSERT, DELETE i MEMBER zahteva svega nekoliko čitanja iz tabele i vreme $O(1)$.
- Ako se tabela previše napuni, treba je prepisati u novu, veću.

Rečnik

Implementacija rečnika pomoću binarnog stabla

- Korisna implementacija onda kada je skup mogućih vrednosti u čvorovima stabla veliki.
- U proseku u ovoj implementaciji brzine operacija INSERT, DELETE i MEMBER je $O(\log n)$, gde je n broj elemenata rečnika.

```
typedef int tipelementa;
typedef struct _cvor {
        tipelementa element;
        struct _cvor *levo, *desno;
} cvor;
typedef cvor *recnik;
```

* Kompletan source je dat uz prezentaciju

Primer

Operaciju MEMBER je lako implementirati, zahvaljujući svojstvima binarnog stabla pretrage.

```
int member(tipelementa x, recnik r) {
    if(!r) {
        return 0;
    }

    if(r->element == x) {
        return 1;
    }
    else if(r->element < x) {
        return member(x, r->desno);
    }
    else {
        return member(x, r->levo);
    }
}
```

Primer

Operacija INSERT se implementira na sličan način:

```
void insert(tipelementa x, recnik *r) {
     if(*r == NULL) {
          *r = (cvor*)malloc(sizeof(cvor));
          (*r)->element = x;
          (*r)->levo = NULL;
          (*r)->desno = NULL;
     }
     else if((*r)->element == x) {
          // Ne radi nista, element je vec u recniku.
          return;
     }
     else if((*r)->element < x) {
          insert(x, &((*r)->desno));
     }
     else {
          insert(x, &((*r)->levo));
     }
```

Rečnik

Delete() - operacija nad rečnikom

U slučaju brisanja čvora razlikujemo tri slučaja:

1.) element x je u listu – čvor se tada prosto izbaci iz stabla (fizički ukloni)

2.) element x je u čvoru koji ima samo jedno dete – tada se umesto čvora sa elementom x postavlja njegovo dete

3.) element x je u čvoru koji ima oba deteta. Tada se pronalazi najmanji element y u desnom podstablu čvora x, uklanja se, i postavlja na mesto čvora x.

Za detalje videti lekciju BSP!!!

Rečnik

Za implementaciju operacije DELETE, korisno je napisati funkciju tipelementa *delete_min*(*recnik**), koja uklanja čvor sa najmanjom vrednošću iz stabla i kao rezultat vraća tu vrednost.

```
tipelementa delete_min(recnik *r) {
    if ((*r)->levo == NULL) {
        // *r je krajnji levi -> najmanji
        tipelementa min_element = (*r)->element;
        cvor *temp = *r;
        *r = (*r)->desno;
        free(temp);
        return min_element;
    }
    else {
     // Idemo stalno ulevo
        return delete_min(&((*r)->levo));
    }
}
```

```
void delete(tipelementa x, recnik *r) {
     if (*r != NULL) {
        if(x < (*r)->element) {
            delete( x, &((*r)->levo) );
        }
        else if (x > (*r)->element) {
            delete( x, &((*r)->desno) );
        }
        else {
             // Trazeni cvor je pronadjen, sledi brisanje.
             if ( ((*r)->levo == NULL) && ((*r)->desno ==
                 NULL)) {
                // list je, treba ga samo obrisati
                 free(*r);
                 *r = NULL;
             }
             else if ((*r)->levo == NULL) {
                 // Ima samo desno dete, zameniti ga sa njim
                 cvor *temp = *r;
                 *r = (*r)->desno;
```

Rečnik

Vremenska analiza operacija u binarnom stablu pretrage

- Ako je binarno stablo pretrage kompletno (svi čvorovi, osim onih na najnižem nivou, imaju dvoje dece), onda ne postoji put dužine veće od $1 + \log n$ čvorova, gde je n ukupan broj čvorova u stablu. U tom slučaju sve operacije zahtevaju vreme $O(\log n)$.
- U najgorem slučaju, kada je stablo potpuno opruženo, sve operacije zahtevaju linearno vreme $O(n)$.

Teorema

Neka je binarno stablo pretrage sastavljeno od n čvorova, n-strukom primenom operacije INSERT(), pri čemu je svaki redosled ubacivanja elemenata jednako verovatan. Tada je očekivano vreme izvršavanja operacija INSERT(), DELETE() i MEMBER() $O(\log n)$.