Abstraktni podatkovni tipi in podatkovne strukture

ADT, tabela (posplošena, skladovna, urejena, dinamična), tabela tabel, slovar, seznam, urejen seznam, preskočni seznam

Tomaž Dobravec, Algoritmi in podatkovne strukture 2

Abstraktni podatkovni tipi in podatkovne strukture

- **Abstraktni podatkovni tip** (angl. abstract data type, **ADT):** opis podatkov + operacije nad podatki.

- Posamezen podatek v ADT je lahko poljubnega tipa.
- V primeru množice lahko govorimo o ”elementih množice”
- Pri ADT ni govora o implementaciji, **ADT == vmesnik**

```
class ElementMnožice {
  int število;
}
```

```
interface ADTMnozica {
  void dodajElement(ElementMnožice elt);
  void brisiElement(ElementMnožice elt);
  boolean jeElement(ElementMnožice elt);
  boolean jePrazna();
}
```

ADT + implementacija = podatkovna struktura (PS)

- Pri implementaciji PS nas poleg **pravilnosti** operacij zanima tudi **učinkovitost**.
- **Želja:** ustvariti PS s čimbolj učinkovitimi operacijami!

Tabela

Tabela

- Najpreprostejša struktura za shranjevanje podatkov.
- Bistvena lastnost: shranjevanje in iskanje elementov po indeksu (položaju).
- `Tabela` je uporabna, kadar je položaj elementa v strukturi pomemben.
- Element tabele je lahko poljubnega tipa.

- ADT `Tabela` je podatkovni tip z naslednjimi operacijami:
  - `init(n)`
  - `put(i, x)`
  - `get(i)-> x`
  - `length()-> n`

- Možne implementacije:
  - povezan seznam elementov
  - povezan blok v pomnilniku

Posplošena tabela

- **Posplošena tabela:** pri dodajanju elementov v strukturo namesto `put(i,x)` (kot bi to naredili pri običajni tabeli) uporabimo operacijo `insert(x)`; na katero mesto se bo dodal element je odvisno od konkretne implementacije.

- Uporabimo, kadar nas pri vstavljanju ne zanima natančen položaj elementa; elemente bi radi le shranili za kasnejšo uporabo.

- Primer: iz vhoda beremo podatke o ocenah študentov; vsako prebrano oceno shranimo. Po koncu branja se ”sprehodimo” po prebranih ocenah in izračunamo povprečje.

- ADT `PosplošenaTabela` je podatkovni tip z naslednjimi operacijami:

  - `init(n)`
  - `insert(x)-> true/false`
  - `get(i)`
  - `find(x)-> i`
  - `delete(x)-> true/false`
  - `length()`
  - `size()`

Skladovna tabela

- Implementacija ADT `PosplošenaTabela`
- Nadgradnja navadne tabele: operacija `insert()` elemente doda na konec tabele.
- Potrebujemo dodaten atribut (`z`), ki ga na začetku (operacija `init()`) postavimo na `0`
  - Operacija `insert(x)`
  - Operacija `size()`
  - operacija `delete(x)`
- Časovna zahtevnost:
  - insert(), get() ... samo en dostop do polnilnika
  - find() ... pregledati je treba celotno tabelo
  - delete() ... v naslabšem primeru prestavim vse

Urejena tabela

- Še ena implementacija ADT `PosplošenaTabela`
- Elementi urejene tabele so urejeni po izbranem vrstnem redu.
- Najpomembnejša operacija: `insert(x)` … element vstavi na "pravo" mesto (elemente, ki so za njim, odrine)

- Časovna zahtevnost:
  - `get(i)` … en dostop do pomnilnika
  - `find(x)` … dvojiško iskanje
  - `delete(x)` … ni spremembe, še vedno slabo
  - `insert(x)` …

- **Kaj je boljše: skladovna tabela ali urejena tabela?**

Dinamična tabela

Dinamična tabela

- Pomanjkljivosti (posplošene) tabele je **omejitev števila elementov**: na začetku povemo, koliko elementov bomo imeli v tabeli, potem tega ne moremo več spreminjati.

- Težavo lahko rešimo z ”dinamično tabelo” – tabela z istimi operacijami kot posplošena tabela + operacija `insert()` nikoli ne odpove (pri posplošeni tabeli je v primeru zasedenosti tabele insert() vrnila false, tu pa operacija insert() polno tabelo ”razširi”).

- **Implementacija dinamične tabele:**

  - Imamo podatkovno strukturo T v kateri hranimo podatke
    - T.tabela
    - T.n
    - T.z

- Operacija T.insert(x)

- Definirajmo: `faktor_zasedenosti`

  - Želja: `faktor_zasedenosti` naj bo navzdol omejen z neko konstanto (recimo: ≥½); to pomeni, da nimamo preveč nezasedenega prostora.

Dinamična tabela

- **Koliko moramo povečati tabelo?**

- **Implementacija operacij `init()` in `insert()`**

  `T.init():`

  `T.insert(x):`

- Dobili smo tabelo, v katero lahko vstavimo poljubno veliko elementov. Koliko pa nas to stane?

Amortizirana analiza operacije insert() za dinamično tabelo 1/2

- Operacija `T.insert(x)` je zelo draga, če je treba tabelo razširiti, sicer je poceni; kaj pa v povprečnem primeru – koliko v povprečju stane ena operacija ?

- Posamezna operacija `T.insert(x)` ima ceno `O(1)` (če gre za preprosto vstavljanje) in `O(T.n)` sicer.

Metoda vsote: seštejem vse prispevke

→ cena ene operacije je 3 oziroma o(1).

Metoda kopičenja: amortizirano ceno operacije `insert()` nastavimo na `3` (vsaka operacija plača za

a) vstavljanje v tabelo,

b) za premik, ko se tabela razširi in

c) za premik še enega elementa ob razširitvi)

in pokažemo, da bo vedno dovolj kredita za plačilo realne cene posamezne operacije.

Metoda potenciala: izberemo

$$\Phi(T) = 2 * T.z - T.n$$

in pokažemo, da je amortizirana cena operacije `insert()` v vsakem primeru enaka 3.

Amortizirana analiza operacije insert() za dinamično tabelo 2a/2

Tabela tabel

Tabela tabel

- Želja: podatkovna struktura s hitrim vstavljanjem in iskanjem
  - Urejena tabela: vstavljanje - O(n), iskanje - O(lg n)
  - Povezan seznam: vstavljanje - O(1), iskanje - O(n)

  Urejena tabela in seznam sta prepočasna (ne želimo imeti počasnih O(n) operacij)

- Tabela tabel (Cascading Arrays) – preprosta implementacija + obe operaciji izvede v času <O(n).

- Podatke hranimo v **tabeli urejenih tabel**, pri čemer ima `i`-ta tabela velikost 2^i.

- Nekatere od tabel so prazne (`0` elementov), druge pa polne (2^i elementov); to, katera tabela je prazna in katera polna, je odvisno od binarne reprezentacije števila elementov (`x`) v podatkovni strukturi.

- Med elementi tabel ni nobenih relacij (t.j. ne moremo vedeti, v kateri tabeli je posamezen element)

- Koliko tabel (praznih in polnih) potrebujemo za hranjenje n elementov?

- **Iskanje elementov:**
  - ker ne vemo, v kateri tabeli je element, iščemo po vrsti;
  - v najslabšem primeru bomo pregledali vseh $\log_2(n)$ tabel;
  - ker so tabele urejene, lahko iščemo z bisekcijo.
- Časovna zahtevnost za iskanje?

- **Vstavljanje v tabelo**: grem po nivojih (i=0, 1, ...): če je tabela A_i prazna, ji nastavim vrednost in končam, sicer jo zlijem s prejšnjo tabelo in nesem naprej, A_i pa postavim na [].

Tabela tabel - podrobneje

Tabele – povzetek časovnih zahtevnosti

	`find()`	`insert()`	`delete()`
skladovna tabela	O(n)	O(1)	O(n)
urejena tabela	O(lg n)	O(n)	O(n)
dinamična tabela	O(n)	O(1)	O(n)
tabela tabel	$O(lg^2\ n)$	O(lg n)	/

Slovar

Slovar

- Slovar je abstraktni podatkovni tip, ki shranjuje podatke, sestavljene iz para (key, value). (ključ, vrednost)
- Razlika med tabelo in slovarjem:
  - v tabeli so podatki ”enojni”, tu nastopajo v paru (key,value)
  - v tabeli je referenca na podatek njihov **indeks** (get(i)) v slovarju pa **ključ** (get(key)).
- Slovar (dictionary) deluje kot funkcija (ključ preslika v vrednost) → map
- Implementacija elementa slovarja z razredom.
- ADT `Slovar` je abstraktni podatkovni tip z (vsaj) naslednjimi operacijami:

Slovar

- Kaj se zgodi če dvakrat kličemo S=insert(S,e) z istim e-jem?

- Ali imamo lahko v slovarju le ključe?

- Slovar je torej na nek način le razširitev množice.

ADT slovar lahko implementiramo na več načinov. V nadaljevanju si bomo ogledali naslednje izvedba:

  - izvedba s seznamom (neurejen seznam, urejen seznam);
  - izvedba s preskočnim seznamom,
  - izvedba z dvojiškimi drevesi (iskalna, urevnotežena, AVL, rdeče-črna);
  - izvedba z večsmernimi drevesi (B-drevo, 2-3 drevo);
  - izvedba z razpršenimi tabelami;
  - izvedba z disjunktnimi množicami;
  - izvedba z Bloomovim filtrom.

- Pri posamezni izvedbi nas bo zanimalo predvsem, **kako hitro** lahko v njih izvajamo operacije `insert()`, `find()` in `delete()` ter področja uporabe.

Seznam

- Seznam je implementacija podatkovnega tipa slovar.
- Seznam vsebuje elemente v določenem vrstnem redu.
- Rekurzivna definicija seznama:
  - prazen seznam označimo z []
  - neprazen seznam je sestavljen iz glave in repa (head, tail), pri čemer je rep seznam.

Seznam

- Seznam ima naslednje operacije: insert(), find(), delete()
- Psevdokoda:

Seznam

- Kaj se zgodi, če v seznam dodam dva enaka elementa (elementa z istim ključem)?

Namigi za implementacijo seznama v javi

- Seznam lahko implementiraš kot razred z dvema atributoma in konstruktorjem takole:

```
public class Seznam {
    Elt head;
    Seznam tail;

    Seznam(Elt elt, Seznam tail) {
        this.head = elt;
        this.tail = tail;
    }
}
```

Prazen seznam:
Seznam z enim elementom:
Seznam z dvema elementoma:

Metoda `insert()`:

Urejen seznam

- Iskanje v seznamu je počasno – v najslabšem primeru vedno pregledati celoten seznam
- Bi “urejenost elementov v seznamu” rešila problem?

Časovna zahtevnost

	`find()`	`insert()`	`delete()`
dinamična tabela	O(n)	O(1)	O(n)
urejena tabela	O(lg n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)

Preskočni seznam

Preskočni seznam - uvod

✧ Seznam elementov lahko shematsko predstavimo na več načinov:

- seznam elementov

- seznam z glavo in repom

- kazalčni seznam elementov

  - poznamo kazalec na začetek seznama,
  - vsak element seznama nosi referenco (kazalec) na naslednji element,
  - zadnji element “kaže” v prazno (vrednost null)

✧ Iskanje v kazalčnem seznamu je zamudno.

✧ Iskanje je počasno tudi v primeru, da so elementi v seznamu urejeni. Zakaj?

✧ Rešitev? Urejen kazalčni seznam dopolnimo z dodatnimi kazalci “za preskakovanje”.

Delno dopolnjen seznam

- ✧ Delno dopolnjen kazalčni seznam: osnovnim kazalcem (kazalci +1) dodamo kazalce na naslednika od naslednika (kazalci +2)

- ✧ Kaj smo s tem pridobili?
  - ko iščemo element, gremo lahko po dve mesti naprej
  - hitrost iskanja smo s tem povečali za faktor 2 → potrebujem le še $\left\lceil \frac{n}{2} \right\rceil + 1$ iskanj!

Bi lahko to iskanje še pohitrili?

- hitrost iskanja smo s tem dodatno povečali za faktor 2 → sedaj potrebujem le še $\left\lceil \frac{n}{4} \right\rceil + 2$ iskanj!

Implementacija delno dopolnjenega seznama

- Delno dopolnjen seznam s povezavamo +1 in +2 implementiramo, na primer, takole:

```
public class SeznamPlus2 {
   Elt glava;
   SeznamPlus2 rep1, rep2;
}
```

- Operacijo iskanja `find()` pa takole:

```
Elt find(SeznamPlus2 s, int key) {
   while(s != null && s.rep2.glava.key < key)
      s = s.rep2;
   while(s != null && s.rep1.glava.key < key)
      s = s.rep1;
   if (s!=null && s.glava.key==key )
      return s.glava;
   else
      return null;
}
```

Popolnoma dopolnjen seznam

- **Popolnoma dopolnjen seznam**: seznam z referencami +1, +2, +4, +8, +16, … $+2^i$ (pri i = $\lceil \lg n \rceil - 1$)
  - zakaj smo se ustavili pri lg(n) - 1?

- Število vseh povezav v PDS:
  - vsak element ima povezavo +1,
  - vsak 2. element ima povezavo +2,
  - vsak 4. element ima povezavo +4,
  - vsak 8. element ima povezavo +8,
  - …

Prostorsko taka predstavitev ne zasede veliko več kot osnovna.

Iskanje v PDS.

- naredimo kvečjemu po en skok vsake dolžine
  - z najdaljšim skokom (ki ga naredim ali pa ne) določim, v kateri polovici bo iskani element
  - z drugim najdaljšim skokom določim, v kateri od preostalih četrtin bo iskani element
  - s tretjim najdaljšim skokom določim, v kateri od preostalih osmin bo iskani element
  - …

Popolnoma dopolnjen seznam

✧ **Iskanje v PDS.**
- skupno število primerjav (in s tem skokov) je torej v najslabšem primeru enako
- **To je pa super**: s povečanjem velikosti strukture za faktor **2** (n -> 2n) smo čas iskanja zmanjšali za cel **velikostni razred** (n -> lg n)!

- Imamo prostorsko nepotratno strukturo s hitrim iskanjem. Kaj nam še manjka do "popolne strukture"?

- **Vstavljanje v PDS**
  - recimo, da že imamo PDS in bi radi vanj vstavili nov element. Koliko nas bo to stalo?

  - V najslabšem primeru moramo popraviti O(n) povezav!

✧ **PDS: iskanje → `O(lgn)`, vstavljenje → `O(n)`, brisanje → `O(n)`.**

✧ **Bi lahko tudi preostali operaciji spremenili v O(lg n)?**

Preskočni seznam

- Namesto toge strukture PDS, v kateri poznamo natančno število referenc posamezne dolžine, definirajmo strukturo, v kateri poznamo le "približno število" referenc posamezne dolžine.

Definicija: Element seznama, ki ima `l` referenc na druge elemente seznama, imenujem element nivoja `l`.

Definicija: dopolnjen seznam z n elementi, v katerem je za vsak $l=0, 1, \ldots \lfloor \lg n \rfloor$ **približno** $\left\lfloor \frac{n}{2^l} \right\rfloor$ elementov nivoja $l+1$, ki so enakomerno porazdeljeni po seznamu, se imenuje **preskočni seznam**.

Preskočni seznam - implementacija

- Preskočni seznam implementiramo, na primer, takole:

```
public class SkipList {
  Elt head;
  int l; // nivo
  SkipList[] tail; // teh je l
}
```

- Operacijo iskanja `find()` pa rekurzivno takole:

```
public static Elt find(SkipList s, int key, int nivo) {
  // izpis poti iskanja:
  System.out.println("Searching in " + s.glava);

  // morda sem že našel (previdno: ker začnem na začetku seznama,
  // je treba preveriti, če je glava null (indicator za začetek)
  if (s.head != null && s.head.key == key) return s.head;

  // … "skačem" po nivojih, dokler ne pridem blizu iskanega elta
  while (nivo >= 0 && s.tail[nivo].head.key > key) {
    nivo--;
  }

  // če sem prišel pod prvi nivo, pomeni, da iskanega elta ni!
  if (nivo < 0) return null;

  // sicer: skočim na naslednji element tega nivoja
  return find(s.tail[nivo], key, nivo);
}
```

Iskanje v preskočnem seznamu - primer

Preskočni seznam – operacija insert()

- **Operacija `insert()`**
  1. določimo nivo l novega elementa
  2. poiščemo mesto, kamor bomo nov element vstavili
  3. element vstavimo in popravimo strukturo

Preskočni seznam – operacija insert()

K1) Kako določimo nivo elementa?

Preskočni seznam – operacija insert()

K2) Iskanje mesta, kamor bomo nov element vstavili

K3) Kako popravimo strukturo?

Preskočni seznam – operacija delete()

- Operacija `delete()`
  - klasično brisanje iz seznama, le da moram po brisanju ustrezno popraviti vse reference;
  - podobno kot pri vstavljanju, si pri brisanju med sprehodom (ko iščem pravo mesto), zapomnim vse reference, ki jih je treba popraviti.

Časovna zahtevnost brisanja: iskanje (lg*(n)*) + popravljanje največ lg*(n)* referenc → skupaj: **O(lg(n))**

Preskočni seznam - uporaba

- Uporaba preskočnega seznama:
  - za implementacijo baz podatkov;
  - v podatkovnih strukturah v različnih programskih jezikih;
  - za implementacijo urejene množice;
  - v algoritmih.

- **Kako bi z uporabo preskočnega seznama iskali k-ti element?**

Časovna zahtevnost

	`find()`	`insert()`	`delete()`
dinamična tabela	O(n)	O(1)	O(n)
urejena tabela	O(log n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)
preskočni seznam	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas

- ker je preskočni seznam verjetnostna podatkovna struktura, govorimo le o pričakovanih časih
  - Če bi šlo vse narobe (zelo "zloben" generator naključnih števil), bi operacije v preskočnem seznamu potekale v linearnem času.

  - Obstajajo tudi boljše implementacije preskočnega seznama z zagotovljeno logaritmično časovno zahtevnostjo.
    (glej: Munro, Papadakis, Sedgewick: Deterministic skip lists)