Přehled přednášky

Kapitola 4. Relace

- opakování
- skládání zobrazení
- princip inkluze a exkluze

Kapitola 5.

- programová implementace struktur
- implementace množin
- generování výběrů

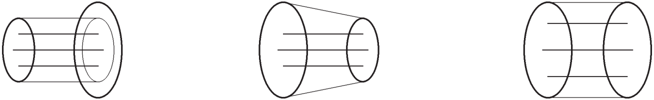
Opakování

Definice 4.7. *Zobrazení (funkce)* z množiny A do B
je podmnožina kartézského součinu $f \subset A \times B$. Pro každý prvek $x \in A$ *existuje právě jeden* prvek $y \in B$ takový, aby $(x, y) \in f$. Píšeme

$$f : A \to B.$$

Funkce

- *prostá (injektivní)* $x \neq y \Rightarrow f(x) \neq f(y)$, nebo $f(x) = f(y) \Rightarrow x = y$
- *na (surjektivní)* $\forall z \in B$ existuje $x \in A$ tak, že $f(x) = z$
- *vzájemně jednoznačná (bijektivní)* je-li f „prostá“ i „na“



4.4. Skládání zobrazení

Definice *skládání zobrazení*
Mějme zobrazení (funkce) $f : A \to B$ a $g : B \to C$.
Jejich *složením* rozumíme zobrazení $(g \circ f) : A \to C$ („g po f“) definované

$$(g \circ f)(x) = g(f(x)).$$

Permutace na množině A můžeme chápat jako bijektivní zobrazení $\pi : A \to A$.
Mějme $A = [1, n]$. Permutace na A určena pořadím $(p_1, p_2, \dots, p_n)$. Zobrazení π definujeme předpisem $\pi(i) = p_i$.

Příklady:
Permutace zapisujeme například

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 5 & 4 & 2 & 6 \end{pmatrix}, \quad \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 5 & 4 & 3 & 1 & 6 \end{pmatrix}.$$

Permutace můžeme skládat

$$\sigma \circ \pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 1 & 3 & 5 & 6 \end{pmatrix}, \quad \pi \circ \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 4 & 5 & 3 & 6 \end{pmatrix}.$$

Poznámky

- Všechny permutace množiny $[1, n]$ spolu s operací skládání tvoří grupu, které říkáme *symetrická grupa*, nebo *permutační grupa*. Značí se S_n.
- Každá grupa je isomorfní některé grupě (podgrupě) permutací.
- Pozor! Používají se i jiná značení pro skládání permutací.

Při zápisu permutace můžeme vynechat první řádek, je-li uspořádán $1, 2, \ldots, n$. Zavedeme si jiný v praxi používaný zápis permutací, pomocí jejich cyklů.

Definice

Nechť π je permutace na množině A. *Cyklem v π* rozumíme takovou posloupnost $(a_1, a_2, \ldots, a_k)$ různých prvků A, že

$$\pi(a_i) = a_{i+1} \text{ pro } i = 1, 2, \ldots, k-1 \text{ a } \pi(a_k) = a_1.$$

Příklady:

- $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 5 & 4 & 2 & 6 \end{pmatrix}$ zapíšeme jako $\pi = (1, 3, 5, 2)(4)(6)$
- $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 5 & 4 & 3 & 1 & 6 \end{pmatrix}$ zapíšeme jako $\sigma = (1, 2, 5)(3, 4)(6)$

Není důležité, kterým prvkem zápis permutace začínáme, avšak obvykle se snažíme začínat „od nejmenšího“ prvku.

Lemma 4.8. Každou permutaci na konečné množině A lze zapsat jako složení cyklů na disjunktních podmnožinách A.

Důkaz Vezmeme libovolný (např. nejmenší) prvek $a_1 \in A$ a iterujeme zobrazení $a_2 = \pi(a_1)$, $a_3 = \pi(a_3)$ atd., až dostaneme a_1 (postup je konečný, protože A je konečná). Tak získáme první cyklus $(a_1, \dots, a_k)$. Pokračujeme v sestavování cyklů ve zbylé množině $A \setminus \{a_1, \dots, a_k\}$ (např. od nejmenšího prvku), dokud prvky A nevyčerpáme. $\square$

- nevýhodou zápisu permutací pomocí cyklů je složitější skládání
- výhodou zápisu permutací pomocí cyklů je snadné určení řádu (délky) permutace, tj. počtu kolikrát složíme permutaci samu se sebou, abychom dostali identitu (existuje vždy po konečném počtu složení!)

Příklad:

Permutace $\tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 2 & 5 & 4 \end{pmatrix}$ má řád 6 ($\tau \circ \tau \circ \tau \circ \tau \circ \tau \circ \tau = \iota$)

Příklad:

Skládání permutací zadaných pomocí cyklů

Máme dány permutace

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 5 & 4 & 2 & 6 \end{pmatrix} = (1,3,5,2)(4)(6),$$

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 5 & 4 & 3 & 1 & 6 \end{pmatrix} = (1,2,5)(3,4)(6).$$

Víme

$$\sigma \circ \pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 1 & 3 & 5 & 6 \end{pmatrix}, \quad \pi \circ \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 4 & 5 & 3 & 6 \end{pmatrix}.$$

Složíme permutace jako cykly:

$$\sigma \circ \pi = (1,2,5)(3,4)(6) \circ (1,3,5,2)(4)(6) = (1,4,3)(2)(5)(6).$$

Podobně

$$\pi \circ \sigma = (1,3,5,2)(4)(6) \circ (1,2,5)(3,4)(6) = (1)(2)(3,4,5)(6).$$

Příklad:

Máme automat na míchání n karet.
Pokaždé provede stejnou permutaci množiny $\{1, 2, \ldots, n\}$.

- použijeme-li k-krát (k je řád permutace), bude jako před začátku
- lze ukázat, že jedním strojem nemůžeme pro $n > 2$ získat všechny permutace (všechna rozmíchání)

Otázky:

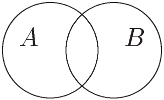
- Jaký je rozdíl mezi bijekcí a surjekcí?
- Jak ukážete že něčeho je stejně?
- Jak ukážete, že jsou množiny stejně velké?
- Jak ukážete, že jsou množiny s operací stejné?
- Jak porovnávat velikosti konečných množin pomocí zobrazení?
- Jak porovnávat velikosti nekonečných množin pomocí zobrazení?
- Kolik nejvíce různých rozmíchání umí automat pro 10 karet?
- Kolik nejvíce různých rozmíchání umí automat pro 32 karet?

4.6. Princip inkluze a exkluze

Bývá nazýván také „princip zapojení a vypojení“, nebo „zahrnutí a vyloučení“. Pro malá n jej často intuitivně používáme:

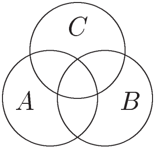
Věta 4.9.
Počet prvků ve sjednocení dvou množin je:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$



Počet prvků ve sjednocení tří je:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|.$$



Obecný tvar principu inkluze a exkluze

Počet prvků ve sjednocení n množin je:

$$\left|\bigcup_{j=1}^{n} A_j\right| = \sum_{\substack{I\subseteq\{1,\dots,n\}\\ I\neq\emptyset}} (-1)^{|I|-1}\cdot\left|\bigcap_{i\in I} A_i\right|.$$

Abychom zjistili, kolik prvků má sjednocení

- sečteme velikosti jednotlivých množin,
- odečteme velikosti průniků všech dvojic,
- přičteme velikosti průniků všech trojic,
- odečteme velikosti průniků všech čtveřic,
- ...

Jednodušší tvar, mají-li množiny (a průniky i množin) stejné velikosti:

$$\left|\bigcup_{j=1}^{n} A_j\right| = \sum_{i=1}^{n}(-1)^{i-1}\cdot\binom{n}{i}\cdot\left|\bigcap_{j=1}^{i} A_j\right|.$$

Příklad:

Ve třídě je 25 žáků. 17 z nich se učí anglicky a 10 německy. 4 se učí anglicky a německy, 4 anglicky a francouzsky, 2 německy a francouzsky a jeden studuje všechny tři jazyky. Kolik studentů se učí jen francouzsky?

Množiny označíme A, N a F. Zapíšeme si

$$|A| = 17,\ |N| = 10,\ |A \cap N| = |A \cap F| = 4,\ |N \cap F| = 2,\ |A \cap N \cap F| = 1$$

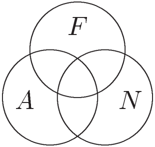
Z rovnice

$$|A \cup N \cup F| = |A| + |N| + |F| - |A \cap N| - |N \cap F| - |A \cap F| + |A \cap N \cap F|$$

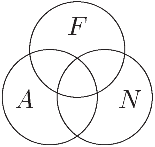
dostaneme

$$|F| = |A \cup N \cup F| - |A| - |N| + |A \cap N| + |N \cap F| + |A \cap F| - |A \cap N \cap F|$$

$$|F| = 25 - 17 - 10 + 4 + 4 + 2 - 1 = 7.$$



Ale někteří z těchto 7 sudentů se učí i jiné jazyky!



Jen francouzsky

$$x = |F| - |A \cap F| - |N \cap F| + |A \cap N \cap F|$$
$$x = 7 - 4 - 2 + 1 = 2 \text{ žáci.}$$

Jen francouzsky se učí 2 žáci.

5. Algoritmizace diskr. struktur

Závěrem prvního tématického celku o diskrétních strukturách si povíme něco o jejich implementaci do programovacích jazyků.

Některé probrané struktury a postupy lze naprogramovat snadno, jiné obtížně. Existuje řada přístupů, které se liší v nároku na pamět a/nebo strojový čas.

Často obecný přístup bývá na úkor rychlosti, ale ne vždy musí být obecný přístup řádově pomalejší.

Tato kapitola je věnována vybraným implementacím.

5.1. Programová implementace struktur

Posloupnost $(a_0, a_1, \ldots, a_n)$

implementujeme se jako jednorozměrné pole `a[ ]`, kde `a[i] =` a_i.

Zobrazení $f : A \to B$

Mějme konečné $A = \{a_1, \ldots, a_n\}$ a $B = \{b_1, \ldots, b_m\}$. Implementujeme jako posloupnost – pole `f[ ]`, ve kterém `f[i]=j` vyjadřuje $f(a_i) = b_j$.

Šikovné, jsou-li A a B celočíselné obory, pro jiné množiny musíme „překládat" prvky z A, B na jejich indexy (složité/náročné na strojový čas).

- pro prvky množiny A využijeme datové typy jako hašovací tabulky
- pro prvky množiny B využijeme pole strukturovaných proměnných či pointerů (ukazatelů)

Binární relace R na množině A

Pro konečné a *malé* $A = \{a_1, \ldots, a_n\}$ implementujeme relaci dvourozměrným polem (maticí) `r[ ][ ]`, ve kterém je
`r[i][j] = 0` pokud $(a_i, a_j) \notin R$ a
`r[i][j] = 1` pokud $(a_i, a_j) \in R$.

Příklady:

Implementace permutace (bijektivní zobrazení $p : [1, n] \to [1, n]$).

Test, je-li p[ ] permutace

Obvykle stačí ověřit zda je p na (surjektivní).

```
for (i=0; i<n; i++)  u[i] = 0;              // pomocné pole
for (i=0; i<n; i++)  u[p[i]] = 1;           // 1 = je použit
for (i=0; i<n; i++) if (u[i]!=1)
     printf("Není permutace!");
```

Složení permutací p[ ] a q[ ] je permutace r[ ]

```
for (i=0; i<n; i++)
     r[i] = q[p[i]];
```

Výpis všech cyklů n-prvkové permutace p[ ]

Výpis všech cyklů získáme například kódem:

```
for (i=0; i<n; i++)  u[i] = 0;                // pomocné pole
for (i=0; i<n; i++) if (u[i]==0) {            // ještě nepoužit
     printf("\ncyklus: %d",i);  u[i] = 1;
     for (j=p[i]; j!=i; j=p[j])  {            // další v cyklu
          printf(",%d",j);  u[j] = 1;
     }
}
```

Mějme binární relaci R na množině $\{0, 1, \ldots, n-1\}$ reprezentovanou dvou-rozměrným polem r[][].

Test zda je relace r je symetrická

```
for (i=0; i<n; i++)
     for (j=i+1; j<n; j++)
          if (r[i][j]!=r[j][i]) {     // je symetrická matice?
               printf("Není symetrická!");
               return -1;
          }
```

Test zda je relace `r` tranzitivní

$$\forall i, j, k : \; r[i][j] \;\wedge\; r[j][k] \;\Rightarrow\; r[i][k]$$

```
for (i=0; i<n; i++)
     for (j=0; j<n; j++) {
          if (!r[i][j]) continue;               // netřeba
          for (k=0; k<n; k++) {
               if (!r[j][k]) continue;          // netřeba
               if (r[i][k]) continue;           // musí být!
               printf("Není tranzitivní!");
               return -1;
          }
     }
```

5.2. Implementace množin

Množiny se implementují obtížně. Problémem je

- jejich neuspořádanost (není stanoveno, kam který prvek uložit)
- hlídat, že prvky se nesmí opakovat.

Charakteristická funkce podmnožiny

Musíme znát celé *univerzum* $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$, ze kterého vybíráme prvky. Podmnožinu $X \subseteq \mathcal{U}$ implementujeme jako pole `x[ ]`, kde

$$\mathtt{x[i]} = \begin{cases} 1 & \text{pro } u_i \in X \\ 0 & \text{jinak.} \end{cases}$$

Výhody: snadno se hledají prvky množiny, sjednocení je funkcí OR, průnik funkcí AND...

Nevýhoda: použitelné jen pro malá univerza $\mathcal{U}$!

Seznam prvků množiny

Množinu X implementujeme seznamem všech prvků. Seznam prvků X uložen v poli `x[ ]` a můžeme psát

$$X = \{\mathtt{x[1]}, \mathtt{x[2]}, \ldots, \mathtt{x[k]}\} \text{ pro pole } \mathtt{x[\ ]} \text{ délky } k.$$

Výhoda: vhodné i pro velmi velká a předem neurčená univerza.

Místo pole je lepší použít *dynamický spojový seznam prvků*, pak lze snadno přidávat a vynechávat prvky v seznamu.

Nevýhoda: vyhledání prvku množiny (nejčastější požadovaná operace) je zdlouhavé – musí se projít celý seznam.

Uspořádaný seznam prvků množiny

Obměna předchozí implementace.

Prvky v seznamu jsou navíc lineárně uspořádány podle předem daného klíče (dle velikosti v případě čísel, dle abecedy u jmen, atd.)

Výhoda: možnost *binárního vyhledávání* prvků v množině metodou půlení intervalu v seznamu (viz příklad 5.4.).

Sjednocení dvou množin daných a[ ],b[ ] do pole c[ ]

```
for (i=0; i<n; i++)
     c[i] = a[i];                   // z a[ ] všech n prvků
for (i=0,k=n; i<m; i++) {           // b[ ] dalších m prvků
     for (j=0; j<n; j++)
          if (b[i]==a[j]) break;    // je-li b[i] v a[ ]
     if (j<n) continue;             // přeskoč ho
     c[k++] = b[i];                 // jinak ho přidej
}
```

Binární vyhledávání čísla k v *uspořádaném* poli p[ ] délky n

```
a = 0;   b = n-1;
while (a<b && p[a]!=k) {            // našli?
     c = (a+b)/2;
     if (p[c]<k)  a = c+1;          // ne, bude větší
     else         b = c;            // ne, bude menší
}
if (p[a]!=k)  printf("Číslo k není v seznamu.");
```

Jen $\lceil \log_2 n \rceil$ vyhledávacích kroků.

Sjednocení dvou seřazených množin (polí) `a[ ]`, `b[ ]` velikosti `m`, `n` do seřazeného pole `c[ ]` velikosti `l`

```
int i=0, j=0, k, l=0;
for (k=0; k < m+n; k++) {
     if (i >= m) {                 // je-li a[ ] vyčerpáno
          c[l++] = b[j++];
          continue;
     }
     if (j >= n) {                 // je-li b[ ] vyčerpáno
          c[l++] = a[i++];
          continue;
     }
     if (a[i] == b[j]) {           // průnik jen jednou
          j++;
          continue;
     }
     c[l++] = (a[i] < b[j]) ? a[i++] : b[j++];
}
```

Příště

Kapitola 5. Algoritmizace diskrétních struktur

- generování výběrů
- generátory náhodných čísel
- kombinatorická exploze

Část II Úvod do Teorie Grafů

Kapitola 6. Pojem grafu

- motivace
- definice grafu
- stupeň vrcholu v grafu