Výsledková listina osmnáctého ročníku KSP po druhé sérii

		škola	*ročník*	*sérií*	*1821*	*1822*	*1823*	*1824*	*1825*	*1826*	*suma*	*celkem*
1.	Peter Perešíni	GJGTajov	4	10			15	9	11	10	45,0	90,2
2.	Josef Pihera	G Strakon	3	7			14	9	7	10	41,7	79,5
3.	Miroslav Klimoš	G Bílovec	1	11	6	8	4	9	10	9	36,0	74,0
4.	Roman Smrž	GOhradní	2	7	6		5	9	11	9	36,3	73,7
5.	Pavel Klavík	G Chrudim	3	11			5	9	11	6	31,0	70,1
6.	Jakub Kaplan	GJKTyla	2	7		6		9	11	6	33,2	63,9
7.	Michal Čudrnák	G Holešov	4	2		6	6		10		26,9	63,7
8.	Zbyněk Konečný	GKptJaroš	3	9		8	4	9	8	5	31,3	62,7
9.	Michal Pavelčík	G UBrod	3	5		8	5	7			23,8	62,5
10.	Lukáš Lánský	GJKTyla	2	7		8	4	9	5		29,2	58,7
11.	Petr Onderka	G VKlobou	3	2		4	5	7	3	1	28,1	58,4
12.	Jiří Maršík	GJKTyla	2	2		2	5	9			21,5	57,7
13.	Tomáš Zámečník	GJKeplera	3	2	6		5	9	3	2	30,0	51,6
14.	Michal Vaner	G Turnov	4	3	6			9			15,0	50,5
15.	Petr Kratochvíl	G SvětláNS	3	11	6	2	3				11,0	43,0
16. – 17.	Radim Pechal	SPŠ Rožnov	3	2	4	8	4				21,2	39,6
	Adam Zivner	G UBrod	4	7						1	1,6	39,6
18.	Daniel Marek	GZborov	4	6							0,0	38,0
19.	Pavel Veselý	G Strakon	1	2	5	3	4	7	3		27,9	37,2
20.	Kateřina Böhmová	G Rožnov	4	2	6	8	10	9			36,2	36,2
21.	Tomáš Herceg	G Třebíč	3	8		5	5				11,4	32,5
22.	Kristýna Krejčová	G Tišnov	3	1							0,0	32,1
23.	Tereza Klimošová	G Lanškr	4	3	6	5		9	11		31,0	31,0
24.	Cyril Hrubiš	G Bílovec	4	8	6	8		6	9		30,4	30,4
25.	Jan Kohout	G Roudnice	3	2	5		5		5	1	25,3	30,0
26.	Josef Špak	GJírovco	3	3							0,0	29,9
27.	Jiří Machálek	G Holešov	4	3			5				8,7	29,8
28.	Drahoslav Viktorýn	G UBrod	3	1							0,0	27,5
29.	Jan Hrnčíř	GFXŠaldy	4	11	6	2	5			1	14,0	27,0
30.	Vojtěch Molda	G Vsetín	4	1							0,0	26,9
31.	Richard Jedlička	G Vlašim	2	2			5				9,1	26,5
32.	Jakub Pavlík jn.	G Kladno	3	2		4		7			13,1	23,6
33.	Ján Mikuláš	G Lučenec	4	1							0,0	21,7
34.	Petr Trňák	G UHradi	3	2	3		3				10,9	19,3
35.	Ondřej Bílka	G Zlín	4	10							0,0	16,7
36.	Jiří Cabal	SPŠ DvKrál	3	2	6		5				15,1	15,1
37.	Rudolf Rosa	G Kladno	3	2		4					4,7	14,9
38.	Ondřej Bouda	GKptJaroš	3	3							0,0	13,1
39.	Lukáš Moravec	GSRandyJN	2	1							0,0	12,7
40.	David Škorvaga	G Kralupy	3	1							0,0	12,4
41.	Martin Kahoun	GJNerudy	3	3			3	3			11,2	11,2
42.	Matej Kollár	G PBystric	4	1							0,0	10,1
43.	Tomáš Ehrlich	G Holešov	3	3							0,0	9,8
44.	Marián Bazálik	G Košice	4	1							0,0	8,3
45.	Adam Ráž	GBudějo	3	4							0,0	7,9
46. – 47.	Ondrej Mikuláš	G Lučenec	3	1							0,0	7,3
	Jan Musílek	G NBydžov	2	1							0,0	7,3
48.	Jan Tichý	GDašická	1	1							0,0	6,6
49.	Vladimír Munzar	SPŠ Rožnov	1	1							0,0	5,8
50.	Radim Cajzl	G NMnMor	0	2		2					3,4	5,6
51. – 52.	Jakub Balhar	GJNerudy	3	1							0,0	4,7
	Martin Fojtík	GSRandyJN	2	1							0,0	4,7
53. – 55.	Miroslav Jančařík	G UBrod	2	1		2					3,5	3,5
	Jakub Loucký	G Písek	3	1							0,0	3,5
	Tomáš Sýkora	G VKlobou	2	1							0,0	3,5
56.	Jiří Václavík	G Dobříš	4	1							0,0	2,3
57.	Jiří Keresteš	ZŠKostelní	0	1	0						0,0	0,0

Milí řešitelé!

Vánoce jsou definitivně za námi. Stromeček dohořel, cukroví dojedli holubi a kapr zase přežil. Ale nezoufejte, blíží se Velikonoce. Navíc pro Vás máme přichystané (velikonoční) překvapení: první týden v dubnu se bude konat jarní soustředění! Podrobnosti a přihlášky se časem objeví na našem webu.

Termín odeslání čtvrté série je tentokrát dne 20. března 2006. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
Praha 1, 118 00

Aktuální informace o KSP naleznete na stránkách *http://ksp.mff.cuni.cz/*, dotazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz.*

Zadání čtvrté série osmnáctého ročníku KSP

18-4-1 HP — 5 bodů

Výpočetní centrum HP (Hippo Programmers) se obrátilo vzhůru nohama. Zmatení hrošíci pobíhají sem a tam a každou chvíli se některý z nich zastaví a usne. Není také divu. Hroši nejsou zvyklí pracovat, natož řešit složité výpočetní úlohy. Jedna taková úloha teď sužuje celou společnost HP, a protože si s tím hrošíci sami neporadí, budete jim muset poradit vy. Výpočetní centrum dostalo následující úkol.

Na vstupu máte libovolně dlouhé číslo ve dvojkovém zápisu a máte za úkol zjistit, kolik nul bude mít takové číslo na konci, když ho přepíšeme do desítkového zápisu. A protože hroši chtějí mít na všechno dost času (zejména na spánek a jídlo), měl by váš algoritmus pracovat co nejrychleji. Můžete předpokládat, že se dané číslo vejde do `integer`u.

Vaším cílem je vyprodukovat *algoritmus*, ne program, stačí tedy, když váš postup dostatečně podrobně popíšete. Na druhou stranu, za pěkný program vy vás mohla neminout i nějaká prémie :–)

Příklad: Číslo 11111010 končí v desítkovém zápisu na 1 nulu. Číslo 1010010000010000 končí na 3 nuly.

18-4-2 Elektronické hrátky — 10 bodů

Malý Martínek se jednoho dne velmi nudil. Nudil se tak moc, že už se víc ani nudit nemohl. Bloumal po bytě a hledal, čím by se zabavil. Televize byla rozbitá, hračky ho již omrzely a všechny knížky o diferenciálním počtu Martínek dávno přečetl. Při svém bloumání narazil na tatínkovu krabici plnou zvláštních broučků a při bližším prozkoumání zjistil, že se jedná o integrované elektronické obvody. Martínek si tedy vzal tatínkovo kontaktní pole a začal si hrát. Když už se mu na kontaktní pole nic nevešlo, rozhodl se, že vyzkouší, co vlastně postavil. Ale ouha. V té obrovské změti drátů a obvodů se skoro nevyznal, natož aby zjistil, jak vlastně jeho síť funguje. Už začínal natahovat moldánky, ale pak si vzpomněl, že má spoustu přátel, kteří řeší KSP, a ti mu jistě pomohou. Aby vám s tím Martínek alespoň trochu pomohl, přepsal schéma své sítě na papír a přitom si všiml, že mezi obvody není žádný cyklus. Všechny použité obvody jsou dvouvstupová logická hradla NAND. Hradlo NAND (negované AND) je hradlo s dvěma vstupy a jedním výstupem, které se chová dle tabulky. Síť má ještě několik nezapojených drátků, na které se zapojí vstup sítě, a několik drátků, které představují výstup sítě.

Hradlo NAND:	*vstup 0*	*vstup 1*	*výstup*
	0	0	1
	0	1	1
	1	0	1
	1	1	0

Napište program, který dostane na vstupu schéma sítě a pak bude schopen odpovídat na dotazy typu: „Když bude tohle na vstupu sítě, co bude na jejím výstupu?“ Svoji síť vám Martínek popsal takto:

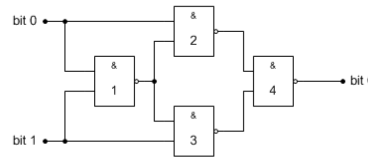
Máte celkem N hradel, k_i bitů na vstupu a k_o bitů na výstupu. Hradla jsou číslována od 1 do N. U každého hradla máte napsáno, kam jsou zapojeny jeho dva vstupy. Každý vstup může být zapojen buď na výstup jiného hradla, nebo na některý vstupní bit celé sítě. Dále máte seznam k_o hradel, jejichž výstupy jsou zapojeny na výstupní bity celé sítě.

Příklad: Máme hradlovou síť se čtyřmi hradly, 2 bity vstupu a 1 bitem výstupu.

Popis hradel:	*hradlo*	*vstup 1*	*vstup 2*
	1.	bit 0	bit 1
	2.	bit 0	hradlo 1
	3.	hradlo 1	bit 1
	4.	hradlo 2	hradlo 3

Popis výstupu:	*výstup*	*připojen na*
	bit 0	hradlo 4

Pro přehlednost ještě obrázek této sítě:



Program nyní spočítá ze zadaného vstupu výstup celé sítě:

vstup	*výstup*
00	0
01	1
10	1
11	0

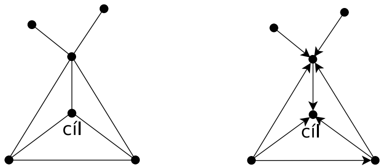
Pozn: Pořadí bitů vstupu a výstupu je stejné jako u klasického binárního zápisu. Tedy 0. bit představuje jednotky, 1. bit dvojky, 2. bit čtyřky atd.

18-4-3 Běh městem — 9 bodů

V Kocourkově se rozhodli uspořádat velkou oslavu na počest výročí města. Městská pokladna však zeje prázdnotou, a tak museli radní vymyslet finančně nenáročný způsob oslavy. Radní si dlouho lámali hlavu, až je napadlo uspořádat pro obyvatele města závod. Zpráva se rychle roznesla do širokého i dalekého okolí a na oslavy se začali sjíždět zvědavci i z jiných měst. Jak už to ale v Kocourkově bývá, radní zapomněli domyslet trať závodu. Ulice města jsou velice zamotané, takže i místní obyvatelé mají problémy trefit tam, kam zrovna chtějí. Tajemník městské rady si je tohoto problému vědom, ale má na práci spoustu jiných věcí (konekonců dělá většinu práce za celou městskou radu), a tak se obrátil s prosbou o pomoc na vás.

Máte danou mapu města jako neorientovaný graf, kde každá hrana představuje ulici a každý vrchol je křižovatka, do které ústí alespoň 3 ulice. Jeden z vrcholů je označen jako cíl celého závodu. Tajemník vás požádal, abyste z každé ulice udělali jednosměrku, a to tak, aby každý závodník, který bude dodržovat pravidla jednosměrek, doběhl po konečném počtu kroků do cíle. Závodníci mohou vybíhat z libovolného místa a na každé křižovatce se libovolně rozhodnout, kam poběží (samozřejmě jen pokud vede z dané křižovatky více ulic; pokud z křižovatky nevede ulice žádná, závodník zde musí čekat do skonání věků). Zároveň by bylo rozumné, aby závod někdy skončil, takže každý závodník se musí dostat do cíle v konečném počtu kroků, ať se na křižovatkách rozhodne pro libovolnou jednosměrku. Pokud existuje víc možných řešení, program vypíše libovolné z nich.

Příklad: Na obrázku vidíte mapu města s vyznačeným cílem a jednu z možných správných orientací:



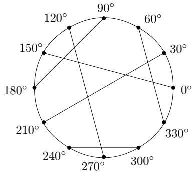
18-4-4 Metro pro krtky 10 bodů

Baron von Maulwurfshaufen měl okolo svého sídla překrásnou zahradu v raně euklidovském slohu, které vévodil obrovský kruhový záhon plný macešek a okrasných okurek. Leč staleté prokletí rodu von Maulwurfshaufenů na sebe nedalo dlouho čekat. Na okraji záhonu se začaly objevovat zlověstné krtiny: první, druhá, třetí, čtvrtá, … až N-tá. Nedosti na tom, krtci vedou velice bohatý společenský život a chtějí své příbuzné na druhém konci záhonu navštěvovat, a tak se rozhodli, že si mezi krtinami vyvrtají metro.

Všechny tunely metra vedou po úsečkách a v téže hloubce, proto se nejspíš mnohé z nich budou protínat a v místech průsečíků bude zapotřebí postavit výhybky a zaměstnat žížaly, které je budou obsluhovat. Krtci se ovšem obávají, že v okolí nebude dostatek žížal. Jelikož krtci jsou narozdíl od cholerického pana barona milá a přátelská zvířátka (černý sametový kožíšek, drápy jako vývrtky atd., však to znáte), jistě jim rádi pomůžete zjistit, kolik žížal budou potřebovat najmout.

Napište program, který dostane zadané polohy krtin (měřené ve stupních, podobně jako jsme popisovali svíčky v úloze 18-1-4) a dvojice krtin, mezi kterými povedou trasy metra, a odpoví počtem průsečíků tras. Můžete předpokládat, že v každé krtině končí právě jedna trasa.

Napište program, který dostane zadané polohy krtin (ve stupních, viz obrázek) a seznam dvojic krtin, mezi nimiž povedou trasy metra. Program pak odpoví počtem průsečíků tras (pro náš obrázek je to 8). Navíc můžete předpokládat, že v každé krtině končí právě jedna trasa a že se v žádném bodě neprotnou více než dvě trasy.



18-4-5 Datokopci 15 bodů

Moderní obor nazývaný Data Mining alias Datokopectví pokračuje ve svém vítězném tažení světem. Proniká už i do vydavatelského průmyslu. Známá tiskařská firma U-tisk se rozhodla jít s dobou a místo zaměstnávání spousty redaktorů shánějících v terénu potřebné informace se jala zprávy raději dolovat.

Za tímto účelem zakoupili dvojrozměrné pole, rozdělili jej na jednotková políčka a prozkoumali, kolik lze na kterém z nich vytěžit dobrých a špatných zpráv (obojí je dáno nějakým přirozeným číslem). Vytěžené zprávy chtějí dopravovat do redakcí, kde se budou zpracovávat: dobré zprávy do redakce pohádek ležící podél celého západního okraje pole, špatné do redakce zpravodajství ležící podél celého severního okraje.

Zprávy se mají dopravovat pomocí pásových dopravníků. Na každém políčku může stát buďto pás běžící z jihu na sever, nebo z východu na západ a naloží na něj vše, co je z příslušného políčka vytěženo. Každý pás musí vést buďto přímo do redakce (je-li políčko u okraje) nebo pokračovat pásem na sousedním políčku, který vede ve stejném směru, čímž se zprávy přidají ke zprávám ze sousedního políčka.

Vaším cílem je napsat program, který po zadání výtěžností jednotlivých políček navrhne rozmístění dopravníků, aby se do redakcí dostalo co nejvíce zpráv správného druhu (dobré zprávy se ve zpravodajství nehodí, podobně jako jsou nežádoucí špatné v pohádkách). Oba druhy zpráv jsou ceněny stejně.

Příklad: Pro pole 4×4 s dobrými zprávami rozmístěnými podle prvního obrázku a špatnými podle druhého je jedno ze správných řešení na obrázku třetím (vydoluje se 98 zpráv).

$$\begin{pmatrix} 0 & 0 & 10 & 9 \\ 1 & 3 & 10 & 0 \\ 4 & 2 & 1 & 3 \\ 1 & 1 & 20 & 0 \end{pmatrix} \quad \begin{pmatrix} 10 & 0 & 0 & 0 \\ 1 & 1 & 1 & 30 \\ 0 & 0 & 5 & 5 \\ 5 & 10 & 10 & 10 \end{pmatrix} \quad \begin{pmatrix} \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \end{pmatrix}$$

18-4-6 Komplikátorový φgl 11 bodů

V minulé sérii jsme si ukázali, jak se dělá globální propagace konstant pomocí dataflow. Nepříjemnou vlastností tohoto algoritmu je jeho kvadratická paměťová (a tedy i časová) složitost. Jejím důvodem je to, že si hodnotu každé proměnné určujeme na začátku a konci každého basic bloku. To ovšem vypadá jako bezdůvodné plýtvání – drtivá většina proměnných nemění své hodnoty příliš často (zejména pomocné proměnné, které si vytváří kompilátor, jsou často nastavovány jen na jednom místě v programu).

Úplně ideální by bylo, kdyby toto byla pravda pro všechny proměnné. Pokud se proměnná nastaví jen jednou a pak už se nemění, stačí si pro ni pamatovat jen jednu hodnotu, protože jestliže ji na tomto jediném místě nastavíme na konstantu, musí být této konstantě rovná úplně všude.

```
    scanf ("%d %d", &Vrcholu, &Hran);
    Hrany = malloc (Vrcholu * sizeof (struct Hrana *));
    for (i = 0; i < Vrcholu; i++) Hrany[i] = NULL;
    for (i = 0; i < Hran; i++) {
        scanf ("%d %d", &Z, &Do);
        Z--; Do--;                                        /* indexujeme 0..Vrcholu-1 */
        NovaHrana1 = malloc (sizeof (struct Hrana));
        NovaHrana2 = malloc (sizeof (struct Hrana));
        NovaHrana1->OpacnaHrana = NovaHrana2;
        NovaHrana2->OpacnaHrana = NovaHrana1;
        NovaHrana1->VedeDo = Do;
        NovaHrana1->Most = 0;
        NovaHrana1->DalsiHrana = Hrany[Z];
        Hrany[Z] = NovaHrana1;
        NovaHrana2->VedeDo = Z;
        NovaHrana2->Most = 0;
        NovaHrana2->DalsiHrana = Hrany[Do];
        Hrany[Do] = NovaHrana2;
    };

    Navstiveno = malloc (sizeof (int)*Vrcholu);
    for (i = 0; i < Vrcholu; i++) Navstiveno[i] = 0;
    for (i = 0; i < Vrcholu; i++) if (Navstiveno[i] == 0) HledejMosty (i, -1, 1);
    Stupen0 = 0; Listu = 0;
    for (i = 0; i < Vrcholu; i++) if (Navstiveno[i] != 0) {
        VidiMostu = SpocitejMosty (i);
        if (VidiMostu == 0) Stupen0++;
        if (VidiMostu == 1) Listu++;
    };
    printf ("%d\n", ((Stupen0 == 1) && (Listu == 0)) ? 0 : (Stupen0 + (Listu+1)/2));

    return 0;
};
```

```
      end;
      next[s] := 0; pred[s] := pred[0];          { a každopádně přidat na konec fronty }
      next[pred[0]] := s; pred[0] := s;
   end;
   s := next[0];                                  { vypíšeme koncový stav fronty }
   while s <> 0 do begin
      writeln(s);
      s := next[s];
   end;
end.
```

Úloha 18-2-5 – Krokoběh – program

```
#include <stdio.h>
#include <malloc.h>

int Vrcholu, Hran;

struct Hrana {
   int VedeDo;                              /* číslo vrcholu (jezírka), kam tahle hrana vede */
   int Most;                                /* 1, je-li to most, jinak 0 */
   struct Hrana *DalsiHrana;                /* další hrana vedoucí z daného vrcholu */
   struct Hrana *OpacnaHrana;               /* hrana opačného směru */
};

struct Hrana **Hrany;                       /* pole pointerů na seznamu hran z daného vrcholu */
int *Navstiveno;                            /* pro průchod do hloubky */

int HledejMosty (int Vrchol, int MinulyVrchol, int Hloubka) {
   /* průchod do hloubky z daného vrcholu, který v dané komponentě souvislosti hledá mosty */
   struct Hrana *AktualniHrana;
   int MinimalniHloubka;
   int HloubkaDaneVetve;

   Navstiveno[Vrchol] = Hloubka;
   MinimalniHloubka = Hloubka;
   for (AktualniHrana = Hrany[Vrchol]; AktualniHrana != NULL; AktualniHrana = AktualniHrana->DalsiHrana)
    if (AktualniHrana->VedeDo != MinulyVrchol) {
      if (Navstiveno[AktualniHrana->VedeDo] == 0) {      /* nový vrchol, voláme se rekurzivně */
         HloubkaDaneVetve = HledejMosty (AktualniHrana->VedeDo, Vrchol, Hloubka+1);
         if (HloubkaDaneVetve > Hloubka) {
            AktualniHrana->Most = 1;
            AktualniHrana->OpacnaHrana->Most = 1;
         };
      } else {
         HloubkaDaneVetve = Navstiveno[AktualniHrana->VedeDo];
      };
      if (HloubkaDaneVetve < MinimalniHloubka) MinimalniHloubka = HloubkaDaneVetve;
   };
   return MinimalniHloubka;
};

int SpocitejMosty (int Vrchol) {
   struct Hrana *AktualniHrana;
   int Mostu;

   Navstiveno[Vrchol] = 0;
   Mostu = 0;
   for (AktualniHrana = Hrany[Vrchol]; AktualniHrana != NULL; AktualniHrana = AktualniHrana->DalsiHrana)
     if (AktualniHrana->Most) Mostu++;
     else if (Navstiveno[AktualniHrana->VedeDo] != 0)
       Mostu+= SpocitejMosty (AktualniHrana->VedeDo);

   return Mostu;
};

int main () {
   int i, Z, Do;
   int Listu, Stupen0;
   int VidiMostu;
   struct Hrana *NovaHrana1, *NovaHrana2;
```


Samozřejmě, že ne všechny programy tuto podmínku splňují. Občas si dokážeme pomoct přejmenováním proměnných. Například kód

```
assign a 0
assign b (a + 1)
assign a b
assign c (a + 1)
```

ve kterém do a přiřazujeme dvakrát, lze přepsat na program

```
assign a_1 0
assign b (a_1 + 1)
assign a_2 b
assign c (a_2 + 1)
```

kde se do každé proměnné přiřazuje jen jednou. Ovšem existují programy, kde to udělat nelze. Třeba

```
if (a < b) 1 2

label 1
assign x 1
goto 3

label 2
assign x 2

label 3
assign i 0
assign s x

label 4
assign s (s + i)
assign i (i + 1)
if (i < 100) 4 5

label 5
assign result s
```

nejde přepsat tak, abychom do proměnných x, i a s nepřiřazovali dvakrát. Toto nastane tehdy, jestliže chceme někdy použít proměnnou, do které jsme předtím mohli dosadit různé hodnoty. Abychom se s tím vypořádali, přidáme si do programu takzvané φ funkce. Funkce φ se nacházejí vždy jen na začátcích basic bloků, a to v místech, kde se sbíhá víc různých definic jedné proměnné. Argumenty φ funkce odpovídají hranám, které do bloku vedou, a její výsledek je vždy roven tomu argumentu, z jehož hrany přijde provádění programu. Například výše uvedený kód vypadá po doplnění φ funkcí takto:

```
if (a < b) 1 2

label 1
assign x 1
goto 3

label 2
assign x 2

label 3
assign x φ(x,x)
assign i 0
assign s x

label 4
assign s φ(s,s)
assign i φ(i,i)
assign s (s + i)
assign i (i + 1)
if (i < 100) 4 5

label 5
assign result s
```

Nyní již můžeme proměnné přejmenovat tak, aby se do každé z nich přiřazovalo jen jednou:

```
if (a_1 < b_2) 1 2

label 1
assign x_3 1
goto 3

label 2
assign x_4 2

label 3
assign x_5 φ(x_3,x_4)
assign i_6 0
assign s_7 x_5

label 4
assign s_8 φ(s_7,s_10)
assign i_9 φ(i_6,i_11)
assign s_10 (s_8 + i_9)
assign i_11 (i_9 + 1)
if (i_11 < 100) 4 5

label 5
assign result s_8
```

Výsledku této transformace se říká *SSA forma* (kde SSA znamená „Static Single Assignment"). Nyní vás možná napadlo několik otázek, na které je nutné si odpovědět:

- Proč jsme přejmenovali i proměnné a a b, a přiřadili i proměnným s různými jmény různá čísla? Důvod je ten, že nyní můžeme na původní jména proměnných zapomenout a pracovat jen s jejich novými *verzemi*. Je samozřejmě praktické, aby každá verze měla vlastní číslo, protože pak se jím dají indexovat tabulky, do nichž si optimalizace ukládají pomocné hodnoty vztahující se k dané verzi.
- Přestože jsme vám to v předchozím textu zatajili, nebylo by dobré, abychom používali neinicializované proměnné. SSA forma by tedy měla splňovat to, že každé použití verze proměnné je dominováno její definicí. Na první pohled by se mohlo zdát, že například proměnná s_{10} toto nesplňuje (má použití ve φ funkci, která je před její definicí). Vzpomeňme si ale, že tuto hodnotu použijeme pouze tehdy, pokud přijdeme z hrany z konce bloku s návěštím 4, a v tomto okamžiku je jistě hodnota s_{10} definována. Často je praktické se dívat na použití ve φ funkcích tak, jako by se ve skutečnosti nacházela na hranách, jimž odpovídají (tedy použití s_{10} a i_{11} ve φ funkcích se chovají tak, jako by byly na hraně z konce bloku s návěštím 4 na jeho začátek).
- Jak SSA formu vytvořit? Existuje několik různých algoritmů, všechny však jsou poměrně netriviální a nebudeme se jimi v tomto seriálu zabývat. Pouze poznamenejme, že časová složitost těchto algoritmů bývá v nejhorším případě kvadratická (převod do SSA formy může v nejhorším případě způsobit kvadratické prodloužení kódu programu), nicméně pro běžné programy, které neobsahují mnoho složitě se chovajících proměnných, je skoro lineární.
- Jak se SSA formy zbavit? Na konci kompilace se potřebujeme zbavit φ funkcí a verzí proměnných, abychom mohli program přeložit do assembleru. Jedna varianta, která vás asi napadne, je prostě zahodit čísla verzí a vrátit se k původním proměnným. To ovšem nemusí fungovat. Například kód

```
assign a x
assign x y
assign b a
```

po převedení do SSA formy vypadá takto:

```
assign a₁ x₂
assign x₃ y₄
assign b₅ a₁
```

Zatím je vše v pořádku, pokud zahodíme verze proměnných, dostaneme původní program. Může se ale stát, že optimalizace nazývaná propagace kopií změní tento kód na

```
assign a₁ x₂
assign x₃ y₄
assign b₅ x₂
```

Zahodíme-li nyní verze proměnných, dostáváme program

```
assign a x
assign x y
assign b x
```

v němž má proměnná b má na konci chybnou hodnotu. Další jednoduchá idea je prostě považovat verze proměnných za nové proměnné, a φ funkce nahradit přiřazeními, které přidáme na příslušné hrany. Toto řešení je korektní, ale není příliš vhodné – do programu typicky přidáme mnoho přiřazení, a navíc budeme mít podstatně víc proměnných. Proto se používá „něco mezi“ – verze proměnných, které spolu navzájem nekolidují (tj. není místo v programu, kde bychom zároveň potřebovali znát hodnotu obou) slepíme do jedné proměnné. Tím se zbavíme většiny přiřazení a zbylé kolidující verze proměnných pak prohlásíme za nové proměnné.

Na závěr si naznačme, jak se dá SSA forma využít. Typicky se podstatně zjednoduší úlohy, které se dříve řešily pomocí dataflow analýzy. Například v propagaci konstant si stačí hodnotu pamatovat pro každou verzi (není třeba rozlišovat, na kterém místě). Algoritmus pak vypadá takto:

1) Hodnoty všech verzí nastav na Top a vlož je do fronty.
2) Z fronty odeber verzi v. Projdi všechny přiřazení tvaru `assign x něco`, v nichž je v použito.
   a) pokud `něco` je φ funkce, slej hodnoty ze všech hran
   b) jinak vyhodnoť `něco`

   Pravidla slévání hodnot a vyhodnocování výrazů jsou stejná, jaká jsme si popsali v minulé sérii. Pokud je získaná hodnota jiná, než jakou jsme měli u x uloženou, nastavíme x novou hodnotu a přidáme x do fronty (pokud tam už není).
3) Opakujeme 2) dokud fronta není prázdná.
4) Verze, jejichž hodnota je nějaká konstanta, nahradíme v programu touto konstantou a přiřazení do nich smažeme.

Protože hodnota přiřazená proměnné se změní nanejvýš dvakrát (z Top na konstantu a z konstanty na Bottom), je časová složitost tohoto postupu lineární ve velikosti programu (v SSA formě), což většinou bývá podstatně lepší, než složitost, jíž jsme dosáhli klasickou dataflow analýzou. Také implementace bývá o dost jednodušší a snáze se přidávají různá vylepšení.

Úloha

Navrhněte algoritmus pro mazání mrtvého kódu (co to znamená viz zadání minulé série) pracující s programem v SSA formě.

Recepty z programátorské kuchařky

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

Binární vyhledávání. Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho x_m) a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;   { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil\log_2 N\rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být k nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $\mathcal{O}(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do O-čka.“]


```
  if a=0 then writeln('perioda delky 0')
  else begin
    z:=a;
    i:=0;
    repeat
      a:=(10*a) mod b;
      inc(i);
    until a=z;
    writeln('perioda delky ',i);
  end;
end.
```

Úloha 18-2-3 – Jeřábník Evžen – program

```
#include <stdio.h>
#include <math.h>

#define MAXN 16384                          /* musí být mocninou dvojky */
#define MAX  (2*MAXN+1)

static struct seg {
   double x, y;
   int a;
} seg[MAX];

static void merge (int i)
{
   struct seg *l = seg+2*i, *r = l+1;
   double a = 2*M_PI*l->a/360;
   seg[i].x = l->x + r->x*cos (a) - r->y*sin (a);
   seg[i].y = l->y + r->x*sin (a) + r->y*cos (a);
   seg[i].a = (l->a + r->a) % 360;
}

int main (void)
{
   int i, j, k, n0, N;

   scanf ("%d", &n0);
   for (N=1; N<n0; N*=2);
   for (i=0; i<n0; i++)
     scanf ("%lf", &seg[N+i].y);
   for (i=N-1; i>=1; i--)
     merge (i);
   while (scanf ("%d%d", &j, &k) == 2)
     {
       j = N+j-1;
       seg[j].a = (180+k) % 360;
       while (j != 1)
         merge (j /= 2);
       printf ("%.2f %.2f\n", seg[1].x, seg[1].y);
     }
   return 0;
}
```

Úloha 18-2-4 – Stavbyvedoucí – program

```
const MaxN = 1000;                          { maximální počet sloupečků }
var M, N : integer;                         { počet požadavků a sloupečků }
    pred, next : array [0..MaxN] of integer; { předchůdci a následníci sloupečků ve frontě }
    p, s : integer;                         { právě zpracovávaný požadavek a sloupeček }
begin
   read(N, M);
   pred[0] := 0; next[0] := 0;              { nula je sama sobě předchůdcem i následníkem }
   for s := 1 to N do pred[s] := -1;        { ostatní sloupečky jsme ještě neviděli }
   for p := 1 to M do begin
      read(s);                              { další požadavek }
      if pred[s] >= 0 then begin            { pokud už jsme ho viděli, pryč s ním z fronty }
         next[pred[s]] := next[s];
         pred[next[s]] := pred[s];
```

γ) Připojujeme-li ho hranou za vrchol stupně alespoň 2, pak se nám zvýší B o 1, A zůstane stejné. Pokud B bylo sudé, není třeba nic dělat. Po tomto přidání bude B liché a vrchol X bude konec nezesouvislněné cesty. Vzorec b) bude zřejmě platit. Nyní pokud je B liché, označíme si list na konci cesty Y. Pokud vrchol X napojujeme za vrchol, který nebyl součástí cesty, pak stačí přidat hranu $X \leftrightarrow Y$. Pokud napojujeme X na cestu, pak vezmeme libovolnou přidanou hranu $I \leftrightarrow J$, tu z grafu odstraníme a přidáme 2 nové $I \to X$ a $J \to Y$. V obou případech stoupne počet přidaných hran v do lesa o 1, což je v souladu s a).

A je to. Pro sudé B jsme dostali rovnou 2-souvislý graf, pro liché musíme ještě konec cesty napojit na libovolný vrchol, který do téhle cesty nepatří, abychom dostali 2-souvislý graf. Tím se ale dostaneme na $A+(B-1)/2+1 = A+\lceil B/2 \rceil$ hran.

Program je implementací výše uvedeného. Pomocí algoritmu popsaného v kuchařce 2. série najde v zadaném grafu mosty a pak v každé komponentě 2-souvislosti spočítá, kolik mostů z ní vede. Nakonec spočte hrany, které je třeba přidat, pomocí ($*$). Časová i paměťová náročnost programu je $\mathcal{O}(M+N)$ (při každém průchodu do hloubky se algoritmus zřejmě na každou hranu podívá dvakrát).

Pavel Čížek

18-2-6 Dominující komplikátory

Někteří řešitelé si prostě uložili matici relace dominance (tedy pole indexované čísly bloků, v níž na pozici $[A, B]$ je 1 pokud basic blok A dominuje basic blok B, a 0 jinak). Toto řešení je nepraktické – na dotaz, zda jeden blok dominuje druhý, jsme sice schopni odpovědět v konstantním čase, ale paměťová složitost je $\mathcal{O}(N^2)$, kde N je počet bloků v programu, což často bude příliš mnoho. Navíc se s touto reprezentací špatně pracuje, pokud optimalizace změní CFG, často je jediná možnost celou matici přepočítat. Dominance je ovšem velmi speciální relace, kterou lze reprezentovat mnohem efektivněji.

Začneme tím, že si zavedeme následující značení: budeme psát $A \le B$, pokud basic blok A dominuje basic blok B. Samozřejmě se tím snažíme naznačit, že relace dominance by se mohla chovat jako uspořádání. Je asi vhodné si to zdůvodnit:

- Pro každý blok A platí, že $A \le A$, tedy že dominuje sám sebe.
- Jestliže $A \le B$ a zároveň $B \le A$, pak $A = B$: pokud by A a B byly různé bloky, pak si vezměme cestu $sv_1v_2 \ldots v_kA$ z počátku do A, která neprochází A (tj. $v_i \ne A$ pro všechna $1 \le i \le k$). Protože $B \le A$, někde na této cestě musí být blok B, tedy $v_t = B$ pro nějaké t. Pak ale $sv_1v_2 \ldots v_t$ je cesta z počátku do B, která neobsahuje A, což je ve sporu s tím, že $A \le B$.
- Jestliže $A \le B$ a zároveň $B \le C$, pak $A \le C$: pokud každá cesta z počátku do C obsahuje B a každá cesta z počátku do B obsahuje A, pak každá cesta z počátku do C také musí obsahovat A.

Dominance tedy opravdu je uspořádání, ale nemusí to být uspořádání úplné – většinou existují bloky, které jsou v ní neporovnatelné, tj. $A \not\le B$ a $B \not\le A$. Tím jsme si tedy příliš nepomohli, protože obecné uspořádání v lepším než kvadratickém prostoru reprezentovat nelze. Povšimneme-li si ovšem ještě následující vlastnosti, situace se značně zjednoduší: Pokud se omezíme na bloky, které dominují nějaký pevně zvolený blok C, pak je uspořádání dominancí úplné, tedy pokud $A \le C$ a $B \le C$, pak buď $A \le B$ nebo $B \le A$. Dokažme si toto tvrzení. Předpokládejme, že $A \le C$, $B \le C$ a A a B jsou přitom neporovnatelné. Zvolme si libovolnou cestu $p = sv_1 \ldots v_kC$ z počátku do C. Na p se musí vyskytovat jak A, tak B. Nechť bez újmy na obecnosti p projde jako poslední A, tj. existuje t tak, že $v_t = A$ a $v_i \ne B$ pro $i > t$. Protože $B \not\le A$, existuje cesta q z počátku do A, která neprochází přes B. Pak ale cesta q, za niž připojíme $v_{t+1} \ldots v_kC$, jde z počátku do C, aniž by prošla přes B, což je spor s tím, že $B \le C$.

Pro každý blok C kromě počátku tedy existuje „nejpozdější“ blok, který ho dominuje (budeme mu říkat *přímý dominátor* bloku C a značit ho $d(C)$), tedy takový, že pokud $A \le C$ a $A \ne C$, pak $A \le d(C)$. Zřejmě $A \le C$ právě tehdy, pokud $A = d(d(\ldots(d(C))\ldots))$. Jestliže si nakreslíme orientovaný graf, jehož hrany jsou dvojice $(C, d(C))$ pro všechny bloky C, dostaneme strom, orientovaný směrem ke kořeni, jímž je počátek. Platí, že $A \le B$, pokud vrchol B patří do podstromu, jehož kořen je A. Další možnost, jak si relaci dominance reprezentovat, je tedy uložit si tento strom a pak při dotazu procházet buď podstrom A, nebo následníky vrcholu B. Paměťová složitost je $\mathcal{O}(N)$ – stačí mít uložen tento strom. Oba postupy mají ovšem v nejhorším případě lineární časovou složitost.

Tento strom si proto ještě dále zpracujeme. Provedeme jeho prohledání do hloubky a budeme si počítat čísla kroků (tedy budeme mít čítač, který si zvýšíme o 1 pokaždé, když vstoupíme do vrcholu nebo se z něj vracíme). Pro každý vrchol C si zapamatujeme čísla $i(C)$ a $o(C)$ – čísla kroků, kdy jsme vstoupili do C a kdy jsme se z něj vrátili. Zřejmě $A \le B$ právě tehdy, když do B vstoupíme až po A, ale vrátíme se z něj předtím, než se vrátíme z A, tedy pokud $i(A) \le i(B)$ a zároveň $o(B) \le o(A)$. Paměťová složitost zůstane lineární, neboť pro každý vrchol si potřebujeme pamatovat pouze dvě čísla. Složitost dotazu bude konstantní, stačí provést dvě porovnání.

Na závěr poznamenejme, že většina algoritmů pro určení dominance přímo konstruuje strom přímých dominátorů, takže nikdy není nutné si pamatovat celou matici dominance. Zajímavá otázka je, jak popsanou datovou strukturu opravit, pokud některá optimalizace změní CFG. Strom přímých dominátorů se změní snadno, nicméně popsané očíslování se nám tím pokazí, proto je v praxi nutné použít složitější datové struktury.

Zdeněk Dvořák

Úloha 18-2-2 – Kvakulátor – program

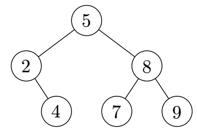
```
var a,b,z,i:integer;

begin
  readln(a,b);
  for i:=1 to b do
    a:=(10*a) mod b;
```


Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

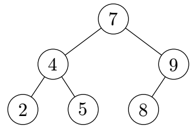
Zkusme ale provést jednoduchý myšlenkový pokus:

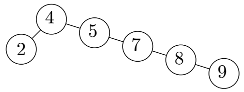
Vyhledávací stromy. Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se budto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):





Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání a, jak za chvilku uvidíme, mnohých dalších operací.

Definice. Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
     vrchol = record
       l, r : pvrchol;  { levý a pravý syn }
       x    : integer;  { hodnota }
     end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil*.

Find. V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
  end;
  TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $\mathcal{O}(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Insert. Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil*. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

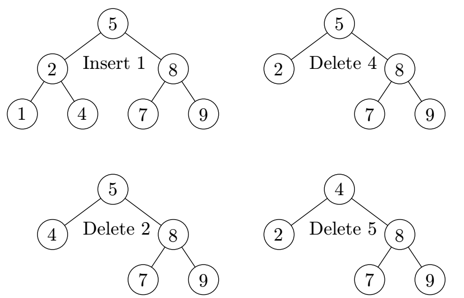
```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení,
  vrátí nový kořen. }
begin
  if v=nil then begin
    { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then  { vkládáme vlevo }
```

```
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then      { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

Delete. Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit          { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin                  { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin
      TreeDel := nil;         { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.r;        { jen pravý syn }
      dispose(v);
    end else if v^.r=nil then begin
      TreeDel := v^.l;        { jen levý }
      dispose(v);
    end else begin            { má oba syny }
      w := v^.l;              { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
      v^.x := w^.x;           { prohazujeme }
      { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $\mathcal{O}(h)$. Ale pozor, jejich používáním může h nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy h dosáhne až N.

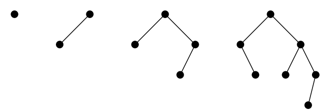
Procházení stromu. Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

```
procedure TreeShow(v:pvrchol);
begin
  if v=nil then exit;  { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

Vyvážené stromy. S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat*. To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy. Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $\mathcal{O}(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože

třetího. Totéž samozřejmě platí i pro třídění tabulek podle sloupečků podle Potrhlíkových požadavků: ponejprv řádky třídíme podle sloupečku daného posledním požadavkem, skupiny se stejnou hodnotou tohoto sloupečku pak podle předchozího požadavku a tak dále, až se propracujeme k začátku seznamu požadavků nebo skončíme se skupinkami o jednom řádku. Z toho ovšem ihned plyne, že zabývat se tímtéž sloupečkem vícekrát je zhola zbytečné: pokud po prvním porovnání podle nějakého sloupečku zůstaly nějaké dva řádky v téže skupině, měly v příslušném sloupečku stejnou hodnotu, a proto nám je další porovnání musí ponechat ve stejném pořadí.

Pokud se tedy nějaký sloupeček v posloupnosti požadavků vyskytuje vícekrát, stačí ponechat jen jeho poslední výskyt. Tím určitě dostaneme posloupnost ekvivalentní se zadanou (takové budeme říkat *řešení*). Zbývá nám ještě dokázat, že žádné kratší řešení nemůže existovat. Kdyby existovalo, vezmeme si nejkratší takové. Určitě se v něm nebudou opakovat sloupečky (jinak by se naším algoritmem dalo ještě zkrátit) a ani v něm nebude žádný sloupeček navíc (to by byl sloupeček, podle kterého se netřídilo ani v zadané posloupnosti, takže bychom ho mohli škrtnout). Tudíž v ní musí nějaký sloupeček z našeho řešení chybět. Pak stačí vytvořit dva řádky, které se budou lišit pouze v chybějícím sloupečku, a takové musí obě řešení setřídit různě, což je evidentní podvod, totiž spor. Podobně můžeme dokázat i to, že naše řešení je nejen nejkratší, ale také jediné s touto délkou: jiné by se nutně lišilo pořadím nějakých dvou sloupečků i, j a mohli bychom sestrojit dva řádky podle i uspořádané opačně než podle j a jinak stejné a opět dojít ke sporu.

Zbývá si rozmyslet, jak naše řešení naprogramovat. Znalci Unixového shellu mohou navrhnout třeba toto:

```
nl -s:|tac|sort -t: -suk2|sort -n|cut -d: -f2
```

My si předvedeme jednoduchý (a přiznejme, že daleko efektivnější) program v Pascalu. Bude číst vstupní posloupnost po jednotlivých prvcích a ve frontě si udržovat řešení pro zatím přečtenou část vstupu. Přijde-li požadavek na třídění podle nějakého sloupečku, přidáme tento sloupeček na konec fronty a pokud se již ve frontě vyskytoval, předchozí výskyt odstraníme. Abychom to zvládli rychle, budeme si frontu pamatovat jako obousměrný spojový seznam, tj. pro každý sloupeček si uložíme jeho předchůdce a následníka. Tak nám celá fronta zabere paměť lineární s počtem sloupečků a na každou operaci si vystačíme s konstantním časem, celkově tedy s časem $\mathcal{O}(M+N)$ (požadavků + sloupečků).

Ještě přidáme malý trik pro zkrácení programu: Abychom si nemuseli dávat pozor na případy, kdy je fronta prázdná, a udržovat si ukazatel na konec fronty, přidáme do fronty ještě sloupeček 0, který bude stále na začátku i na konci (fronta tedy bude zacyklená) a který pak nevypíšeme.

Tomáš Gavenčiak & Martin Mareš

18-2-5 Krokoběh

Krokodýli šli dneska spát nalačno prakticky u všech došlých řešení a překvapivě u většiny došlých řešení byl spokojen i Potrhlík, který stavbu přežil bez bankrotu. No a nyní, jak se úloha měla řešit. Většina řešitelů (možná pod vlivem kuchařky) používala terminologii teorie grafů, proto jí i v tomto řešení použijeme.

O co tedy šlo. Zjistit, kolik nejméně hran je třeba přidat do grafu, aby se stal 2-souvislý. Hned na začátku si všimneme, že pokud najdeme v grafu komponentu, která je 2-souvislá, tak jí můžeme zkontrahovat (scvrknout) do jednoho vrcholu, aniž by se změnil počet potřebných hran. Takhle můžeme pokračovat tak dlouho, dokud se v grafu budou vyskytovat kružnice. Snadno se dá nahlédnout, že hrany tohoto zkontrahovaného grafu budou mosty v původním grafu (most není součástí žádné kružnice, proto nebude v žádném kroku zkontrahován, na druhou stranu pokud hrana není most, pak je součástí nějaké kružnice a proto bude dříve či později zkontrahována). Je také vidět, že takto zkontrahovaný graf bude les, jelikož neobsahuje kružnice. Dále budeme uvažovat tento les.

Nyní mohou nastat 2 situace. První, kterou dost řešitelů zapomnělo ve svých řešení ošetřit, je ta, že graf byl na počátku 2-souvislý, tj. že se zkontrahoval do bodu. Pak není třeba nic přidávat.

Druhá je zbytek. Kolik bude třeba hran dodat? Jelikož v 2-souvislém grafu má každý vrchol stupeň (tj. kolik hran do něj vede) alespoň 2 a hrana spojuje právě 2 vrcholy, musíme přidat alespoň $A + \lceil B/2 \rceil$ $(*)$ hran, kde A je počet vrcholů stupně 0, B počet vrcholů stupně 1 (tj. listů) a zaokrouhluje se nahoru, jelikož v případě lichého B musíme tento lichý list také zapojit do nějaké kružnice, tedy tento lichý list zapojíme na libovolný vrchol.

Nyní, indukcí podle počtu vrcholů dokážeme, že tolik i stačí. Pro 2 vrcholy může les vypadat buď jako 2 vrcholy a pak je třeba přidat ještě 2 hrany (což splňuje vzorec $(*)$), nebo jsou tyto 2 vrcholy spojené hranou, a pak stačí přidat jednu (opět v souladu s $(*)$). Všimněme si také, že jsme v obou případech alespoň jednu hranu přidali.

Teď si uvědomíme, že libovolný les jde vyrobit z jednoho vrcholu pomocí operací:

1) přidej vrchol (a s ničím ho nespojuj)
2) přidej vrchol a spoj ho hranou s nějakým vrcholem, který už v lese je.

Nyní uvažujme, že pro N vrcholů máme již 2-souvislý les pomocí

a) $A + B/2$ hran (pro sudé B)
b) $A + (B-1)/2$ hran (pro liché B; 1 cesta nezesouvislena)

Přidejme vrchol X pomocí pravidla:

1) Vezmeme nějakou přidanou hranu (vedoucí $I \leftrightarrow J$), tu odstraníme a přidáme místo ní hrany $I \leftrightarrow X$ a $X \leftrightarrow J$. Tím nám stoupl počet přidaných hran do grafu o 1. Také A se zvětšilo o jedna, takže a), resp. b) stále platí.
2) Při připojování X mohou nastat tři situace:
   α) Připojujeme ho hranou pod vrchol Y stupně 0. Pak ale od tohoto vrcholu vedou 2 přidané hrany. Vezmeme libovolnou z nich (nechť vede z I) a zrušíme ji. Místo ní zavedeme novou hranu $I \leftrightarrow X$. Touto operací se nám snížil počet vrcholů stupně 0 v grafu o 1, nicméně z X i Y se staly listy a proto je B o 2 vetší. Tedy a) příp. b) je stále splněno.
   β) Připojujeme ho hranou za list L. Pokud je B liché a list L je konec naší volné cesty, není třeba nic dělat a indukční předpoklady máme splněny. Jinak do tohoto listu vede nějaké přidaná hrana (z nějakého vrcholu I). Pak ale stačí zrušit hranu $I \leftrightarrow L$ a zavést novou hranu $I \leftrightarrow X$. Tím zůstane počet přidaných hran zachován. L přestal být po tomto kroku listem, nicméně objevil se nový list X, tudíž A i B zůstalo a tedy a), resp. b) stále platí.

tím, aby $a < b$ tak, že provedu $a' = a \bmod b$, kde mod je zbytek po dělení. Počítáním s a' místo a se mi perioda určitě nezmění. První zbytek si tedy nastavím rovnou jako $z_1 = a \bmod b$ (zbavím se tak všech cifer a najednou a potom při dělení pod sebou "připisuji" už jen 0). Další zbytky získám postupně jako $z_{i+1} = (10z_i) \bmod b$. Všimněte si, že cifry výsledku nás vůbec nezajímají, po nalezení opakování zbytků se budou opakovat i cifry, neboť cifry závisí jen na předchozím zbytku a b, které je pevné. Délka periody je nanejvýš $b-1$, protože po více než $b-1$ číslech $0..b-1$ potkám buď 0 nebo nějaký zbytek dvakrát.

Jednodušší cesta, jak najít periodu, je pamatovat si zbytky z_i v poli a pokaždé pole prohledat, zda už zbytek máme. Toto má časovou složitost $\mathcal{O}(b^2)$ a paměťovou $\mathcal{O}(b)$.

Trochu chytřejší je nepamatovat si pole z_i, ale zda a kde jsem již potkal zbytek z v poli a_z. Potom zjišťuji, zda jsem již zbytek potkal v konstantním čase a dostanu časovou i paměťovou složitost $\mathcal{O}(b)$.

Situaci mi komplikuje jen možná před-perioda, bez ní bych si mohl zapamatovat první zbytek a prostě si na něj počkat až ho potkám znovu. Před-perioda má ale délku nanejvýš b, stačí tedy provést prvních b dělení a z_b buď bude 0 (pak je perioda 0) nebo si ho zapamatuji a až ho potkám na pozici z_{b+p}, našel jsem periodu délky p. Časová složitost je tak pořád $\mathcal{O}(n)$, paměťová $\mathcal{O}(1)$.

Existuje i rychlejší řešení pracující v čase $\mathcal{O}(\sqrt{b})$ nebo i lepším (hlavně podle zběsilosti použité faktorizace), ale to je příliš složité na to, abych ho zde uspokojivě popsal.

Tomáš Gavenčiak

18-2-3 Jeřábník Evžen

Z počtu došlých řešení je jasné, že vás demolice bažiny zaujala, ale poradit Evženovi, jak ovládat jeřáb nejen spolehlivě, ale také rychle, byl pro vás oříšek. Pojďme ho tedy rozlousknout.

Jednoduché řešení problému je zapamatovat si úhel natočení každého segmentu a po každé změně výslednou pozici demoliční koule spočítat. Pokud víme počáteční bod i-tého segmentu a jeho absolutní úhel α v rovině, spočítáme z něj pozici $(i+1)$-ního segmentu posunutím o délku segmentu ve směru α. Takto postupným počítáním koncové pozice segmentů dojdeme až k pozici koule. Pokud máme n segmentů a dostaneme m dotazů, tento algoritmus poběží v čase $\mathcal{O}(m \cdot n)$. Podobný algoritmus poslala většina z vás.

My se ovšem nespokojíme s jednoduchým řešením. Potřebujeme si zapamatovat některé pozice, abychom je nemuseli pokaždé počítat znovu. To uděláme celkem klasickou metodou — postavíme nad segmenty *intervalový strom*.

Předpokládejme nyní, že n je mocninou dvojky. Potom si segmenty představíme jako listy dokonale vyváženého binárního stromu. Protože počet vrcholů se na každé další hladině zdvojnásobí, je hloubka našeho stromu $\log_2 n$, neboli $\mathcal{O}(\log n)$. Pokud spočteme tuto geometrickou posloupnost velikosti hladin, dostaneme $2n-1$, a tedy náš strom má celkem $\mathcal{O}(n)$ vrcholů.

Každý vnitřní uzel stromu u bude reprezentovat skupinu segmentů, které jsou listy podstromu s kořenem u. Tuto skupinu po sobě jdoucích segmentů si můžeme představit jako jeden dlouhý segment, který začíná na počátku prvního segmentu a končí na konci posledního segmentu.

Každý segment můžeme charakterizovat jeho délkou a relativním natočením vůči předchozímu segmentu. Pokud chceme podobně popsat i složený segment, musíme ještě přidat natočení posledního segmentu ze skupiny, protože právě ten určuje, jak bude natočen další segment. Jednoduchý segment pak má v tomto popisu oba úhly stejné.

Protože bychom ale museli pro složené segmenty složitě počítat jejich délku a vstupní úhel, zapamatujeme si rovnou souřadnice bodu, kde by tento segment končil, kdyby byl umístěn do počátku a předchozí segment mířil do kladného směru osy y. Díky tomu jsou data segmentu nezávislá na ostatních segmentech a nebudeme jich muset tolik měnit při otočení jeřábu.

Jak tedy vytvoříme složený segment? Souřadnice segmentu v označme $x(v)$ a $y(v)$, výstupní úhel $\alpha(v)$. Vezměme vnitřní uzel u, který má dva syny s_1 a s_2. Pak souřadnice u spočítáme jako otočení souřadnic s_2 o úhel $\alpha(s_1)$ a přičtení k souřadnicím s_1. Výstupní úhel u je součtem výstupních úhlů jeho synů.

$$\begin{aligned}x(u) &= x(s_1) + x(s_2)\cdot\cos\alpha(s_1) + y(s_2)\cdot\sin\alpha(s_1)\\ y(u) &= y(s_1) + x(s_2)\cdot\sin\alpha(s_1) + y(s_2)\cdot\cos\alpha(s_1)\\ \alpha(u) &= \alpha(s_1) + \alpha(s_2)\end{aligned}$$

Již umíme skládat segmenty a můžeme tedy ze segmentů v listech vygenerovat složené segmenty tak, že půjdeme po hladinách stromu od nejnižší k nejvyšší a budeme je popsaným způsobem plnit. Všimněte si, že odpověď na dotaz, kde je koule jeřábu, se schovává již v kořeni stromu, protože ten je složením všech dílčích segmentů.

Složitější je otázka, zda dokážeme také tuto strukturu aktualizovat po otočení nějakého segmentu. Podívejme se, kolik složených segmentů se změní při otočení jednoho segmentu. Jsou to pouze ty, které tento segment obsahují, a ty leží na cestě od listu ke kořeni. Stačí tedy při změně segmentu projít po cestě od segmentu ke kořeni a upravit všechny složené segmenty. To uděláme stejně, jako bychom je vytvářeli znova.

Jak dlouho nám to bude trvat? Již jsme si ukázali, že náš strom má hloubku $\mathcal{O}(\log n)$ a tedy celková časová složitost je $\mathcal{O}(m \cdot \log n)$. Paměťová složitost je $\mathcal{O}(n)$, protože strom má také lineární velikost.

Ještě se podíváme na implementaci našeho algoritmu. Pokud vytváříme vyvážený strom, do kterého nechceme přidávat nebo z něj odebírat prvky, často se na to hodí použít implementaci haldy v poli. Ta má kořen na pozici s indexem 1 a synové vrcholu s indexem i jsou (v případě binárního stromu) na pozici $2i$ a $2i+1$. Tím se často zjednoduší veškeré cestování po stromě.

Jako třešničkou na dortu se můžete inspirovat Martinovým (MJ) vzorovým řešením.

Petr Škoda

18-2-4 Stavbyvedoucí

Ah, bezdůvodně čekáš, čtenáři drahý, ďábelské fígle, i když na obyčejný počátek prachobyčejného řešení stavbyvedoucího strastí tato věta vyznívá značně zvláštně. Demonstruje totiž jeden velice důležitý fakt: setřídit slova podle třetího písmenka, pak podle druhého (stabilně, tj. se zachováním původního pořadí, pokud jsou druhá písmenka nějakých dvou slov stejná) a nakonec podle prvního, dopadne stejně jako nejdříve je setřídit podle prvního, pak zvlášť každou skupinu začínající stejným písmenem setřídit podle druhého a skupinky mající stejné i druhé písmeno ještě podle

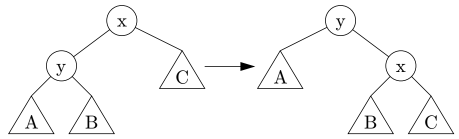
jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2}.(2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = \mathcal{O}(\log N)$. *Q.E.D.*

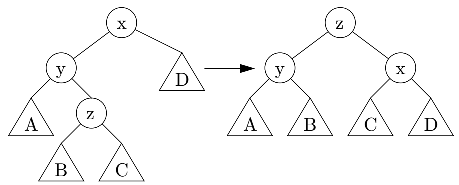
AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

Rotace. Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění" za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo" po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace. Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak". Raději opět předvedeme na obrázku:



Znaménka. Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, $-$ pro levý podstrom hlubší a $+$ pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

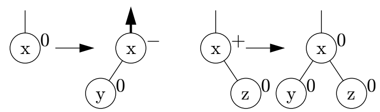
Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

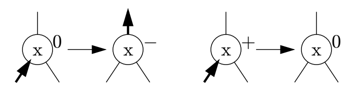
Vyvažování po Insertu. Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

Nejprve přidání listu samotné:

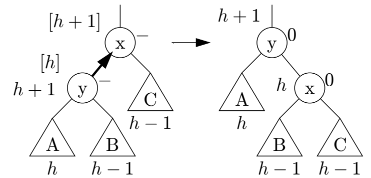


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

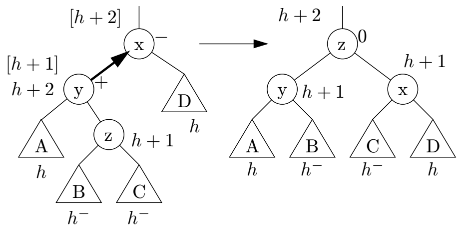


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

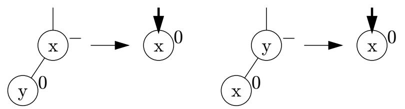
Další možnost je y jako $\oplus$:



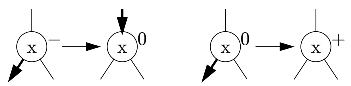
Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

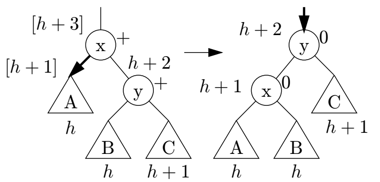
Vyvažování po Deletu. Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

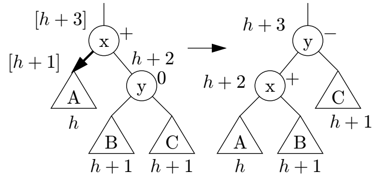


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



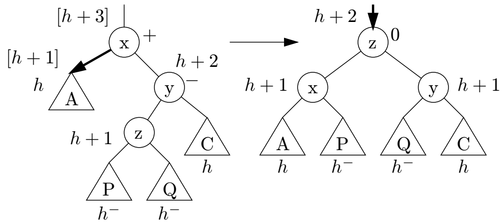
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end. Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů. AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-Černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.)
- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název). Hloubka opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

  Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $\mathcal{O}(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $\mathcal{O}(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných). To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromad – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.
- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0,1 \rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $\mathcal{O}(\log N)$.
- *BB-α stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $\mathcal{O}(\log N)$ na operaci.

Cvičení. Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-α stromy
3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $\mathcal{O}(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a, b \rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase…

Několik poznámek na závěr.

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP. Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adeľsona-Veľského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Dnešní menu vám servírovali
Martin Mareš a Tomáš Valla

Vzorová řešení druhé série osmnáctého ročníku KSP

18-2-1 Potrhlý syslík

Většina z vás si s tímto úkolem hravě poradila. Je však nutné dodat, že samotný postup není dostačující řešení. Vždy je třeba říci, proč zvolený postup funguje. Ale i to se většině řešitelů povedlo, a tak se v rychlosti koukněme, jak by mohlo vypadat vzorové řešení.

Máme 50 mincí, 18 je otočeno lícem, zbylé jsou otočeny rubem. Oddělme libovolných 18 mincí a označme je jako první hromádku. Ostatní mince tvoří hromádku druhou. Všechny mince v první hromádce otočíme a nahlédneme, že tím je úkol splněn. Z původních 18 mincí otočených lícem se do první hromádky dostalo právě k z nich. A tedy $18 - k$ je počet rubem otočených mincí v první hromádce. Po otočení všech mincí v této hromádce se počty rubem a lícem otočených prohodí. V první hromádce tak bude $18-k$ lícem otočených mincí. Ve druhé hromádce je ale již od začátku $18 - k$ lícem otočených mincí, protože celkem jich bylo 18 a k z nich připadlo do první hromádky. Počty v obou jsou tak shodné a úloha je vyřešena.

Jak mnozí z vás také poznali, princip řešení není dán tím, že mincí bylo právě 50 a 18 bylo lícem otočených a že řešení se dá spolu se zadáním úlohy zobecnit a stále bude fungovat. Dodejme ještě, že k takovýmto úlohám není třeba psát zdrojový kód. V některých došlých řešeních se zbytečně otáčely mince tam a zase zpět, za což byl strháván jeden bodík, protože taková řešení jsou vlastně pomalejší.

David Matoušek

18-2-2 Kvakulátor

Vzpomenu si na ZŠ a písemné dělení "pod sebou". Tam vše záleželo na zbytcích – vyšel-li nám někdy po vyčerpání cifer čitatele zbytek 0, nemělo číslo periodu, pokud se nějaký zbytek opakoval potom, co nám došly cifry čitatele, vznikla tak automaticky perioda. Ještě předtím ale zajis-