Programování v příkladech

8. srpna 2007

Obsah

Seznam obrázků

Kapitola 1

Lineární spojové seznamy

1.1 Úlohy na zahřátí

Následuje několik jednoduchých úloh.

> Proč je při řešení úloh týkajících se spojových struktur důležitější tužka a guma než klávesnice a monitor?

U úloh tohoto typu si vždy kreslete *obrázky*, na kterých bude znázorněno, co je propojeno ukazately, odkud kam ukazatele vedou a jak se ukazatele mění, když provádíme nějakou operaci. Nejdůležitější pro řešení těchto úloh není klávesnice a monitor, ale tužka a guma (guma proto, že propojení ukazatele se při operacích často mění).

1.2 Jednoduché spojové seznamy

Příklady z této podkapitoly řešte tak, že do následujícího souboru doplníte implementace zadaných funkcí. Rozhodně si ale tento zdrojový kód nejprve projděte a ujistěte se, že mu zcela rozumíte.

```
type
    UkPrvek = ^Prvek;

    (*
     * Zakladni stavebni jednotka spojoveho seznamu.
     * Zapouzdruje: hodnotu jednoho prvku seznamu a ukazatel na dalsi prvek.
     *)
    Prvek =
        record
            info: Integer;
            dalsi: UkPrvek;
        end;


(*
 * Prida prvek na zacatek seznamu.
 * parametry:
 *     prvniPrvek - ukazatel na prvni prvek seznamu
 *     vkladanaHodnota - hodnota, ktera se ma do seznamu vlozit
 *)
procedure pridej(var prvniPrvek: UkPrvek; vkladanaHodnota: Integer);
var
    {Prvek, ktery bude zapouzdrovat nove pridavanou hodnotu}
    novyPrvek: UkPrvek;
begin
        {vytvorime novy prvek a pripojime ho na zacatek}
        new (novyPrvek);
        novyPrvek^.info := vkladanaHodnota;
        novyPrvek^.dalsi := prvniPrvek;
```

```
30          prvniPrvek := novyPrvek;
31  end;
32
33  (*
34   * Vypise vsechny prvky spojoveho seznamu, oddeluje je mezerou.
35   * parametry:
36   *      prvniPrvek - ukazatel na prvni prvek seznamu.
37   *)
38  procedure vypisSeznam(var prvniPrvek: UkPrvek);
39  var
40      {ukazatel na prvek, ktery zapouzdruje prave vypisovanou hodnotu}
41      aktualniPrvek: UkPrvek;
42  begin
43      aktualniPrvek := prvniPrvek;
44      while (aktualniPrvek <> nil) do begin
45          write(aktualniPrvek^.info, ' ');
46          aktualniPrvek := aktualniPrvek^.dalsi;
47      end;
48  end;
49
50
51  (*
52   * Test spojoveho seznamu. Seznam je naplnen cisly 1 az 50 a pote vypsan.
53   *)
54  var
55      seznam: UkPrvek;
56      vkladanaHodnota: Integer;
57  begin
58      seznam := nil;
59
60      for vkladanaHodnota := 1 to 50 do begin
61          writeln('Vkladam hodnotu ', vkladanaHodnota);
62          pridej(seznam, vkladanaHodnota);
63      end;
64
65      writeln('OBSAH SEZNAMU:');
66      vypisSeznam(seznam);
67      writeln;
68  end.
```

Příkaz `dispose` slouží k uvolnění bloku paměti. Co to přesně znamená a jak prvky přesně odstranit, abychom datovou strukturu udrželi v konzistentním stavu? Kam ukazuje ukazatel po provedení `dispose`? Co když na uvolňovaný blok ukazuje více ukazatelů? Je pak třeba `dispose` volat vícekrát?

Příkaz `dispose` slouží k uvolnění paměti, kterou si program alokoval (tj. ‚zamluvil') pomocí příkazu `new`. Alokovanou paměť je třeba uvolňovat, jinak by paměť dříve nebo později došla. Příkaz `dispose` se pro uvolnění daného bloku paměti volá vždy jen jedenkrát a jako parametr se mu předá ukazatel na blok, který se má uvolnit. Po provedení `dispose` ukazaje tento ukazatel na stejné místo v paměti jako před provedením. Protože uvolněný blok již program nebude používat, je většinou třeba všechny ukazatele, které na ten blok ukazují, nastavit na `nil`. Příklad:

```
1   var ukazatel1, ukazatel2, ukazatel3: UkPrvek;
2   begin
3           {tady neco alokujeme, budou na to ukazovat tri ukazatele}
4           new (ukazatel1);
5           ukazatel2 := ukazatel1;
6           ukazatel3 := ukazatel1;
7
8           {ted to zrusime - treba pres ukazatel2, ale to je jedno,
9            stejne tak by se mohlo napsat dispose (ukazatel1)
10           nebo dispose (ukazatel3), protoze rusime to, na co ten
11           ukazatel ukazuje, ne ten ukazatel samotny, a vsechny tri
12           ukazatele ukazuji na totez misto v pameti}
```

```
13          dispose(ukazatel2);
14
15          {a nakonec ukazatele nastavime na nil, aby neukazovaly
16           na uvolnenou pamet}
17          ukazatel1 := nil;
18          ukazatel2 := nil;
19          ukazatel3 := nil;
```

Implementujte funkci `velikostSeznamu(var prvniPrvek: UkPrvek): Integer`, která vrátí počet prvků seznamu.

```
1  function pocetPrvku(var prvniPrvek: UkPrvek): Integer;
2  var
3      pocet: Integer;
4      aktualniPrvek:UkPrvek;
5  begin
6      pocet := 0;
7      aktualniPrvek := prvniPrvek;
8      while (aktualniPrvek <> nil) do
9      begin
10         inc(pocet);
11         aktualniPrvek := aktualniPrvek^.dalsi;
12     end;
13     pocetPrvku := pocet;
14 end;
```

Implementujte funkci `vypisPosledni(var prvniPrvek: UkPrvek): Integer;`, která bere ukazatel na první prvek a vrací hodnotu posledního prvku.

```
1  function posledniHodnota(var prvniPrvek: UkPrvek): Integer;
2  var
3      aktualniPrvek:UkPrvek;
4  begin
5      aktualniPrvek := prvniPrvek;
6      if aktualniPrvek = nil then
7          posledniHodnota := 0
8      else begin
9          while (aktualniPrvek^.dalsi <> nil) do
10             aktualniPrvek := aktualniPrvek^.dalsi;
11         posledniHodnota := aktualniPrvek^.info;
12     end;
13 end;
```

Implementujte proceduru `odstranPosledni(var prvniPrvek: UkPrvek);`, která odstraní ze seznamu poslední prvek (tj. doskáče od prvního prvku až po předposlední a predposlednímu odtrhne nasledovníka). Technický problém: jednoprvkový seznam - je třeba zvlášť ošetřit případ, kdy má seznam právě jeden prvek.

```
1  procedure odstranPosledni(var prvniPrvek: UkPrvek);
2  var
3      predchudce, aktualni: UkPrvek;
4  begin
5      if prvniPrvek = nil then exit;
6      if prvniPrvek^.dalsi = nil then begin
7          dispose(prvniPrvek);
8          prvniPrvek := nil;
9      end else begin
10         aktualni := prvniPrvek;
11         while aktualni^.dalsi <> nil do begin
```

```
            predchudce := aktualni;
            aktualni := aktualni^.dalsi;
        end;
        dispose (predchudce^.dalsi);
        predchudce^.dalsi := nil;
    end;
end;
```

Implementujte funkci `otocNedestruktivne(var prvniPrvek: UkPrvek): UkPrvek`, která vytvoří nový seznam, který bude obsahovat prvky zadaného seznamu v obráceném pořadí.

```
function otocNedestruktivne(var prvniPrvek: UkPrvek): UkPrvek;
var
    aktualni, novy: UkPrvek;
    hodnota: Integer;
begin
    aktualni := prvniPrvek;
    novy := nil;
    while (aktualni <> nil) do begin
        hodnota := aktualni^.info;
        aktualni := aktualni^.dalsi;
        pridej(novy,hodnota);
    end;
    otocNedestruktivne := novy;
end;
```

Implementujte funkci `otocSeznamDestrukt(var prvniPrvek: UkPrvek): UkPrvek`, ktará otočí seznam tak, že po otočení bude obsahovat stejné prvky jako před otočením, ale v obráceném pořadí. Nevytváří nový seznam, jen změní položku `dalsi` u vsech prvku tak, aby neukazovala na následníka v původním seznamu, ale na předchůdce, a vrátí ukazatel na prvek, který byl v původním seznamu poslední, tj. v novém seznamu bude první.

```
procedure otocDestruktivne(var prvniPrvek: UkPrvek);
var
    {prvni z dvojice prvku, mezi kterymi se zmeni "smer sipky"}
    prvniProhazovany,
    {druhy z dvojice prvku, mezi kterymi se zmeni "smer sipky"}
    druhyProhazovany,
    {pomocny ukazatel na naslednika druheho prohazovaneho,
     abychom na nej neztratili odkaz}
    naslednikDruhehoProhazovaneho:
        UkPrvek;
begin
    if (prvniPrvek = nil) then
        exit;
    prvniProhazovany := prvniPrvek;
    druhyProhazovany := prvniPrvek^.dalsi;
    prvniProhazovany^.dalsi := nil;
    while (druhyProhazovany <> nil) do begin
        naslednikDruhehoProhazovaneho := druhyProhazovany^.dalsi;
        druhyProhazovany^.dalsi := prvniProhazovany;
        prvniProhazovany := druhyProhazovany;
        druhyProhazovany := naslednikDruhehoProhazovaneho;
    end;
    prvniPrvek := prvniProhazovany;
end;
```

1.3 Seznamy s ukazatelem na konec

Příklady z této podkapitoly řešte tak, že do následujícího souboru doplníte implementace zadaných funkcí. Rozhodně si ale tento zdrojový kód nejprve projděte a ujistěte se, že mu zcela rozumíte.

```
type
    UKprvek = ^Prvek;

    (*
     * Zakladni stavebni jednotka spojoveho seznamu.
     * Zapouzdruje: hodnotu jednoho prvku seznamu a ukazatel na dalsi prvek.
     *)
    prvek =
        record
            info: Integer;
            dalsi: UkPrvek;
        end;

    (*
     * Seznam je tvoren ukazatelem na svuj prvni a posledni prvek.
     *)
    Seznam =
        record
            zacatek: UKprvek;
            konec: UKprvek;
        end;


(*
 * Inicializace seznamu.
 * Prazdny seznam neobsahuje zadne prvky,
 * takze ukazatele na prvni a posledni prvek budou nil.
 *)
procedure init(var s: Seznam);
begin
    s.zacatek := nil;
    s.konec := nil;
end;

(*
 * Prida prvek na konec seznamu.
 * parametry:
 *     s - seznam, do nejz chceme prvek vlozit
 *     hodnota - vkladana hodnota
 *)
procedure pridejNaKonec(var s: Seznam; hodnota: Integer);
var
    novy: UkPrvek;
begin
    new (novy);
    novy^.info := hodnota;
    novy^.dalsi := nil;
    if (s.konec <> nil) then
        s.konec^.dalsi := novy;
    if (s.zacatek = nil) then
        s.zacatek := novy;
    s.konec := novy;
end;

(*
 * Na zacetek seznamu vlozi zadany prvek.
 *)
procedure pridejNaZacatek(var s: Seznam; var prvek: UkPrvek);
begin
```

```
60      prvek^.dalsi := s.zacatek;
61      s.zacatek := prvek;
62      if (s.konec = nil) then
63          s.konec := s.zacatek;
64  end;
65
66  (*
67   * Na standardni vystup vypise vsechny prvky seznamu s od prvniho po posledni.
68   *)
69  procedure vypisSeznam (var s:seznam);
70  var
71      aktualni: UkPrvek;
72  begin
73      aktualni := s.zacatek;
74      while aktualni <> nil do begin
75          write(' ', aktualni^.info);
76          aktualni := aktualni^.dalsi;
77      end;
78  end;
79
80  (*
81   * Na konec seznamu pripoji usek z jineho seznamu.
82   * parametry:
83   *      kam - cilovy seznam (k nemu se bude pripojovat usek)
84   *      zacatek - prvni prvek useku ve zdrojovem seznamu
85   *      konec - prvni prvek ve zdrojovem seznamu, ktery jiz neni soucasti useku
86   *              je-li nil, pak se kopiruji prvky az po konec zdrojoveho seznamu
87   *)
88  procedure kopiruj(var kam: Seznam; zacatek, konec: UkPrvek);
89  var
90      akt: UkPrvek;
91  begin
92      akt := zacatek;
93      while (akt <> konec) do begin
94          pridejNaKonec(kam, akt^.info);
95          akt := akt^.dalsi;
96      end;
97  end;
98
99  (*
100  * Smaze vsechny prvky seznamu s.
101  *)
102 procedure zrus(var s: Seznam);
103 var
104     akt, tmp: UkPrvek;
105 begin
106     akt := s.zacatek;
107     while (akt <> nil) do begin
108         tmp := akt;
109         akt := akt^.dalsi;
110         dispose (tmp);
111     end;
112     s.zacatek := nil;
113     s.konec := nil;
114 end;
115
116
117 var
118     s, s_bez_kraju: Seznam;
119 begin
120     init(s);
121     pridejNaKonec(s, 8); pridejNaKonec(s, 5); pridejNaKonec(s, 7);
122     pridejNaKonec(s, 6); pridejNaKonec(s, 13); pridejNaKonec(s, 7);
123     pridejNaKonec(s, 21); pridejNaKonec(s, 1); pridejNaKonec(s, 4);
```

```
124     vypisSeznam(s);
125     writeln;
126
127     init(s_bez_kraju);
128     kopiruj(s_bez_kraju, s.zacatek^.dalsi, s.konec);
129     vypisSeznam(s_bez_kraju);
130     writeln;
131
132     zrus(s);
133     vypisSeznam(s);
134     writeln;
135
136     zrus(s_bez_kraju);
137     vypisSeznam(s_bez_kraju);
138     writeln;
139 end.
```

Napište proceduru `split`, která rozdělí spojový seznam $(a_1, a_2, \ldots, a_n)$ na dva seznamy $(a_1, a_2, \ldots, a_i)$ a $(a_{i+1}, \ldots, a_n)$, kde i je dolní celá část z $n/2$. Procedura nechť vytvoří nové senamy, původní seznam musí zůstat netknutý.

Základní myšlenka je jednoduchá. Procedura požívá dva ukazatele, `akt1` a `akt2`. Dokud ukazatel `akt2` není `nil`, platí, že ukazuje-li `akt1` na k-tý prvek seznamu, pak `akt2` ukazuje na $2k$-tý prvek seznamu. Toho je docíleno tak, že na začátku ukazuje `akt1` na první prvek seznamu a `akt2` na druhý prvek seznamu. Pak se v každém kroku posuneme ukazatelem `akt1` o jeden prvek vpřed, zatímco ukazatelem `akt2` se posumene o dva prvky vpřed (jde-li to). Tento krok opakujeme tak dlouho, dokud není `akt2` na konci seznamu, tj. dokud nemá hodnotu `nil`. Pak ukazatel `akt1` ukazuje na prostřední prvek seznamu. Prvky od začátku seznamu až po prostřední prvek zkopírujeme do seznamu `s1`, zbylé prvky do seznamu `s2`.

```
1  procedure split(var s, s1, s2: Seznam);
2  var
3      akt1, akt2: UkPrvek;
4  begin
5      if (s.zacatek = nil) then exit;
6      akt1 := s.zacatek;
7      akt2 := s.zacatek^.dalsi;
8      while akt2 <> nil do begin
9          akt2 := akt2^.dalsi;
10         if (akt2 <> nil) then begin
11             akt2 := akt2^.dalsi;
12             akt1 := akt1^.dalsi;
13         end;
14     end;
15     kopiruj(s1, s.zacatek, akt1^.dalsi);
16     kopiruj(s2, akt1^.dalsi, nil);
17 end;
```

Co se rozumí nedestruktivním a destruktivním a destruktivním sjednocením dvou uspořádaných seznamů?

Sjednocení uspořádaných seznamů s1 a s2 je uspořádaný seznam s takový, že množina všech jeho prvků je sjednocením množiny všech prvků prvního seznamu a všech prvků druhého seznamu. Destruktivním sjednocením se rozumí taková implementace, ve které se nikde neobjeví prikaz `new`, pouze se přepojují ukazatele. Naopak nedestruktivní je takové, které `new` hojně využívá, a co je důležité, sjednocované seznamy zůstanou nedotčeny. Destruktivní sjednocení by tedy mohlo vypadat třeba tak, že do prvního seznamu se vmíchají prvky z druhého seznamu. Druhý seznam bude tedy po provedení procedury prázdny, zatímco první bude obsahovat uspořádané sjednocení původnich seznamů.

Implementujte nedestruktivní sjednocení dvou uspořádaných seznamů.

```
procedure sjednoceni(var s1, s2, s: Seznam);
var
    {ukazatel na aktualni prvek s1}
    prvni,
    {ukazatel na aktualni prvek s2}
    druhy: UkPrvek;
begin
    prvni := s1.zacatek;
    druhy := s2.zacatek;
    if (prvni=nil) and (druhy=nil) then exit;

    while (prvni<>nil) and (druhy<>nil) do begin
        if prvni^.info <= druhy^.info then begin
            pridejNaKonec(S, prvni^.info);
            prvni := prvni^.dalsi;
        end else begin
            pridejNaKonec(S, druhy^.info);
            druhy := druhy^.dalsi;
        end;
    end;

    if prvni <> nil then kopiruj(S, prvni, nil);
    if druhy <> nil then kopiruj(S, druhy, nil);
end;
```

Implementujte proceduru `mergesort`, která k zadanému spojovému seznamu vytvoří nový setříděný spojový seznam, který bude obsahovat stejné prvky jako původní seznam. Původní seznam zůstane nedotčen. Implementujte jako rekurzivní algoritmus. Co je to kontrakt funkce nebo procedury a proč je obzvláště důležitý při návrhu rekurzivních algoritmů?

Když navrhujete jakoukoliv rekurzivní proceduru, je vhodné rozmyslet si tzv. *kontrakt*. To slovo znamená doslova *závazek*. Tedy, co daná procedura musí dělat a co nesmí dělat. V případě mergesortu by kontrakt mohl znít takto:Procedura bere odkazy na dva inicializované seznamy. Druhý z těchto seznamů musí být prázdný. Procedura do druhého seznamu uloží všechny prvky prvního seznamu vzestupne setříděně, první seznam nechá nezměněný.

Skoro každá rekurzivní procedura vypadá takto:

```
if (vstup_maly) then
    jednoduse_spocitej_vystup
else begin
    uprav_vstup;
    zavolej_se_rekurzivne;
    uprav_vystup;
end;
```

Je zde velmi úzká souvislost s principem důkazu matematickou indukcí. Ostatně, korektnost rekurzivních algoritmů se obvykle indukcí dokazuje. Kontrakt je vlastně tvrzení, které je třeba dokázat.

Toto je kostra mergesortu:

```
if ("nesetrideno" je jednoprvkovy nebo prazdny) then begin
    kopiruj "nesetrideno" do "setrideno";
end else begin
    vytvor pomocne seznamy s1, s2, s_s1 a s_s2;
    inicializuj je;
    rozdel "nesetrideno" na 2 casti, vysledek do s1 a s2;
    setrid rekurzivne s1, vysledek do s_s1;
    setrid rekurzivne s2, vysledek do s_s2;
    sjednot s_s1 a s_s2, vysledek do "setrideno";
    zrus pomocne seznamy;
end;
```

Zkusme dokázat korektnost takto definované procedury.

Pro prázdný a jednoprvkový vstupní seznam procedura vrátí kopii vstupního seznamu a ta bude (triviálně) setříděná.

Nechť má seznam n prvků, kde $n > 1$. Rozdělíme-li seznam na dvě poloviny `s1` a `s2`, pak sjednocení kolekcí prvků `s1` a `s2` bude rovno kolekci prvků vstupního seznamu. Pokud už víme, že procedura správně třídí seznamy délky kratší než n, rekurzivním zavoláním procedury dostaneme uspořádané seznamy `s_s1` a `s_s2` a sjednocení kolekcí jejich prvků bude opět rovno kolekci všech prvků vstupního seznamu. Slitím `s_s1` a `s_s2` dostaneme uspořádaný seznam. Kolekce jeho prvků bude rovna kolekci prvků vstupního seznamu. Algoritmus je tedy korektní.

Pro úplnost uveďme ještě kompletní zdrojový kód procedury mergesort.

```
procedure mergesort (var nesetrideno, setrideno: Seznam);
var
    s1, s2, s_s1, s_s2: Seznam;
begin
    if (nesetrideno.zacatek = nesetrideno.konec) then begin
        kopiruj(setrideno, nesetrideno.zacatek, nil);
    end else begin
        init(s1);
        init(s2);
        init(s_s1);
        init(s_s2);

        split(nesetrideno, s1, s2);
        mergesort(s1, s_s1);
        mergesort(s2, s_s2);
        sjednoceni(s_s1, s_s2, setrideno);

        zrus(s_s2);
        zrus(s_s1);
        zrus(s2);
        zrus(s1);
    end;
end;
```

Implementujte destruktivní sjednocení dvou lineárních spojových seznamů.

```
(*
 * Provede destruktivni sjednoceni ponechavajici duplicitni klice.
 * parametry:
 *     s1 - ukazatel na zacatek prvniho seznamu a vysledneho sjednoceni
 *     s2 - ukazatel na zacatek druheho seznamu, po provedeni bude nil
 *)
procedure sjednoceni (var s1, s2: Seznam);
var
    {ukazatel na prvky prvniho seznamua jeho predchudce}
    akt1,pred,
    {ukazatel na prvky druheho seznamu}
    akt2,
    {pomocny ukazatel pri prepojovani}
    pom: UkPrvek;
begin
    akt1 := s1.zacatek;
    akt2 := s2.zacatek;
    if (akt1 = nil) and (akt2 = nil) then
        exit;
    if akt1 = nil then begin
        s1.zacatek := s2.zacatek;
        s1.konec := s2.konec;
        s2.zacatek := nil;
        s2.konec := nil;
        exit;
    end else if akt2 = nil then
```

```
27          exit;
28
29      while (akt1 <> nil) and (akt2 <> nil) do begin
30      {seznamy jsou neprazdne}
31          if akt1^.info <= akt2^.info then begin
32              {mensi hodnota v s1}
33              pred := akt1;
34              akt1 := akt1^.dalsi;
35          end else begin
36              {mensi hodnota v s2}
37              pom := akt2;
38              akt2 := akt2^.dalsi;
39              s2.zacatek := akt2;
40
41              if akt1 = s1.zacatek then begin
42                  {novy zacatek s1}
43                  pridejNaZacatek(s1, pom);
44                  pred := s1.zacatek;
45              end else begin
46                  {vkladame mezi pred a akt1}
47                  pred^.dalsi := pom;
48                  pom^.dalsi := akt1;
49                  pred := pom;
50              end;
51          end;
52      end;
53      if akt1 = nil then begin
54          {pripojeni zbytku s2 => s2 musime vyprazdnit}
55          pred^.dalsi:=akt2;
56          akt1 := akt2;
57          s1.konec := s2.konec;
58          s2.konec := nil;
59          s2.zacatek := s2.konec;
60      end else
61          s2.konec := akt2;
62  end;
```

1.4 Obousměrné seznamy

Nyní byste již měli mít dostatek zkušeností, abyste si sami mohli implementovat jednoduchou knihovnu pracující s obousměrnými spojovými seznamy. Připomeňme, že obousměrný spojový seznam je seznam, ve kterém každý prvek obsahuje nejen ukazatel na následníka, ale i na předchůdce. Ukazatel na předchůdce prvního prvku má hodnotu `nil`, stejně tak ukazatel na následníka posledního prvku. Řešte úkoly v následujícím pořadí:

1. Definujte nový datový typ pro prvek obousměrného spojového seznamu - záznam obsahující položky `info`, `dalsi` a `predchozi`.
2. Definujte datový typ pro obousmerný spojový seznam - záznam obsahující ukazatel na první a poslední prvek.
3. Napište proceduru `init`, které se předá reference na spojový seznam a ona ho inicaializuje: nastaví začátek i konec na `nil`.
4. Napište proceduru, která přidá na začátek zadaného seznamu prvek se zadanou hodnotou.
5. Napište proceduru, která přidá na konec zadaného seznamu prvek se zadanou hodnotou.
6. Napište proceduru, která smaže první prvek.
7. Napište proceduru, která smaže poslední prvek (zde se ukáže výhoda obousměrnosti).
8. Napište proceduru, která smaže ze seznamu prvek se zadanou hodnotou. Pokud tam takový prvek není, procedura bude požadavek ignorovat.

1.5 Seznamy s hlavou

Tato sekce pojednává o ne příliš používané, zato u zkoušky se občas vyskytující datové struktuře - lineárním spojovém seznamu s hlavou.

Co je to lineární spojový seznam s hlavou?

Lineární spojový seznam s hlavou je, stejně jako základní varianta spojových seznamů, řetízek prvků, v němž každý prvek má dva atributy - hodnotu a ukazatel na další prvek. První prvek v tomto řetízku však nenese žádnou informaci, jeho hodnota může být libovolná. Reprezentuje-li seznam s hlavou posloupnost $[a_1, a_2, \ldots a_n]$, potom hodnoty prvků v řetízku jsou po řadě x, a_1, a_2,... a_n, kde x může být naprosto libovolné.

Vysvětlete, nejlépe na příkladě, výhodu použití lineárních spojových seznamů s hlavou.

Pro spojové seznamy platí jeden velmi důležitý invariant: první prvek vždy existuje (tj. ukazatel na první prvek nemá nikdy hodnotu `nil`). A navíc, žádná operace nemusí první prvek nijak využívat ani měnit. Proto spousta algoritmů pracujících se seznamy s hlavou nemusí rozlišovat, zda pracuje s prázdným nebo neprázdným seznamem. A navíc, nemusí rozlišovat ani to, zda požadovaná operace ovlivní první prvek seznamu nebo jiný než první prvek seznamu.

Ukažme si dvě varianty přidávání hodnoty do *uspořádaného* lineárního spojového seznamu. První varianta algoritmu bude přidávat prvek do obyčejného uspořádaného seznamu, druhá varianta do uspořádaného seznamu s hlavou.

```
procedure pridejDoUsporadaneho(var seznam: UkPrvek; hodnota: Integer);
var
    aktualni, novy: UkPrvek;
begin
    new (novy);
    novy^.info := hodnota;

    {budeme pridavat na zacatek seznamu}
    if (seznam = nil) or (seznam^.info > hodnota) then begin
        novy^.dalsi := seznam;
        seznam := novy;

    {budeme pridavat jinam nez na zacatek seznamu}
    end else begin
        aktualni := seznam;
        while (aktualni^.dalsi <> nil) and (aktualni^.dalsi^.info < hodnota) do
            aktualni := aktualni^.dalsi;

        novy^.dalsi := aktualni^.dalsi;
        aktualni^.dalsi := novy;
    end;
end;
```

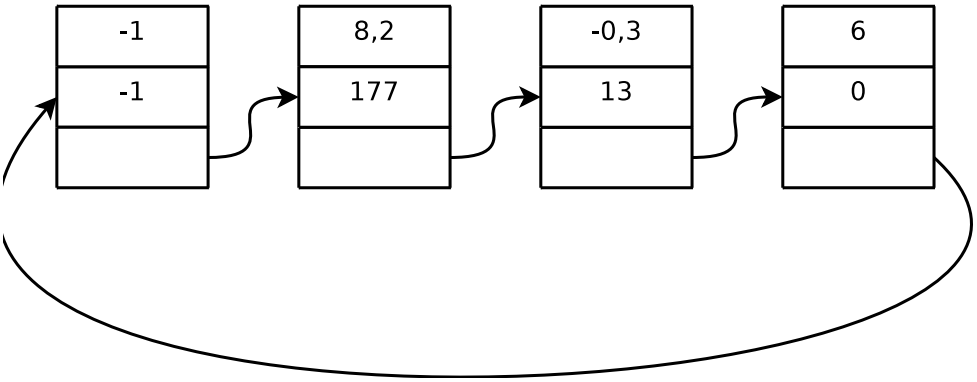
A nyní druhá varianta.

```
procedure pridejDoUsporadaneho(seznam: UkPrvek; hodnota: Integer);
var
    aktualni, novy: UkPrvek;
begin
    new (novy);
    novy^.info := hodnota;

    aktualni := seznam;
    while (aktualni^.dalsi <> nil) and (aktualni^.dalsi^.info < hodnota) do
        aktualni := aktualni^.dalsi;

    novy^.dalsi := aktualni^.dalsi;
    aktualni^.dalsi := novy;
end;
```

U druhé varianty jsme se nemuseli zabývat případem, kdy přidáváme prvek na začátek seznamu, a to z toho prostého důvodu, že na začátek seznamu prvek nikdy nepřidáváme. Na začátku seznamu se totiž vždy nachází hlava, jejíž hodnota nehraje žádnou roli (nevztahuje se na ni tedy pravidlo, že prvky seznamu jsou uspořádány). Hlava navíc vždy existuje, takže se nemusíme zabývat případem, kdy je seznam prázdný.

Nakreslete schematicky jednoduchý cyklický lineární spojový seznam s hlavou, který reprezentuje polynom $8{,}2x^{177}$ - $0{,}3x^{17}$ + 6.

Obrázek 1.1 Reprezentace polynomu pomocí cyklického seznamu s hlavou.



Jaké jsou výhody takovéto reprezentace polynomu?

Tato reprezentace se obzvlášť hodí pro řídké polynomy, tedy polynomy tvaru $a_0 + a_1x + a_2x^2 \ldots + a_nx^n$ takové, že pro většinu $m < n$ je $a_m = 0$. Reprezentace seznamem místo polem přinese efektivnější využití paměti Díky kruhovému seznamu s hlavou zase snáze detekujeme konec polynomu. Nulový polynom nebudeme reprezentovat ukazatelem s hodnotou `nil`, nýbrž seznamem obsahujícím jediný prvek - hlavu.

Na závěr si zkuste vyřešit několik příkladů, které se týkají seznamů s hlavou.

1. Implementujte několik základních operací pracujících nad lineárními spojovými seznamy s hlavou (inicializace, přidávání prvku, mazání zadaného prvku).
2. Implementujte několik základních operací pracujících nad kruhovými spojovými seznamy s hlavou (inicializace, přidávání prvku, mazání zadaného prvku).
3. Implementujte několik základních aritmetických operací (sčítání, násobení, negace) nad polynomy, které budete reprezentovat jako cyklické seznamy s hlavou.

1.6 Kde se stala chyba?

Následuje několik příkladů, ve kterých bude vaším úkolem odhalit chybu v uvedeném zdrojovém kódu, případně najít protipříklad, na kterém uvedená procedura nebo funkce selže.

Toto je pokus o proceduru, která má inicializovat obousměrný spojový seznam. Proč je špatně?

```
1 procedure inicializuj(var s: Seznam);
2 begin
3     s.zac^.pred := nil;
4     s.kon^.nasl := nil;
5     s.kon := s.zac;
6 end;
```

Procedura má datovou strukturu nastavit tak, aby reprezentovala prázdný spoják. Prázdný spoják = spoják bez žádných prvků = spoják, který nemá první ani poslední prvek = spoják, kde `s.zac = nil` a `s.kon = nil`. Tedy tělo procedury by mělo vypadat takto:

```
s.zac := nil;
s.kon := nil;
```

Procedura je chybná, neboť provádí chybnou dereferenci, a to na prvních dvou řádcích. Tak například u přikazu `s.zac^.pred := nil` je `s.zac` před provedením `init`u neinicializovaná, tj. míří na náhodné místo v paměti, takže dereference `s.zac^.pred` může způsobit zhroucení programu (ve FreePascalu se to tak stane takřka s jistotou, Borland Pascal si nechá líbit i chybnou dereferenci).

Následující procedura má přidat prvek do obousměrného spojového seznamu. Proč je špatně?

```
1  procedure pridejNaZacatek (var s: Seznam; hodnota: Integer);
2  var novy:
3      UkPrvek;
4  begin
5      new (novy);
6      novy^.info := hodnota;
7      novy^.pred := nil;
8      novy^.nasl := s.zac;
9      s.zac^.pred := novy;
10     s.zac := novy;
11     if s.kon = nil then
12         s.kon := novy;
13 end;
```

Pokud pomocí této procedury přidáváme prvek do prázdného seznamu, pak `s.zac` má hodnotu `nil` (za předpokladu, že se správně provede `init`, v opačném případě nějakou náhodnou hodnotu), takže dereference `s.zac^.pred := novy` v proceduře `pridejNaZacatek` je chybná.

Mějme lineární spojový seznam, jehož prvky jsou znaky. Seznam reprezentuje řetězec (*k*-tý znak řetězce odpovídá *k*-tému prvku seznamu). Následující funkce má zjistit, jestli je řetězec reprezentovaný druhým parametrem podřetězcem řetězce, který je reprezentovaný prvním parametrem. Najděte protipříklad, u kterého funkce vrátí `false`, přestože by měla vrátit `true`.

```
1  function obsahujePodretezec(var ret, podRet: UkPrvek): boolean;
2  var
3      aktualni: UKprvek;
4      aktualni_vpodret: UKprvek;
5  begin
6      obsahujePodretezec := false;
7      aktualni := ret;
8      aktualni_vpodret := podRet;
9      while (aktualni <> nil) and (aktualni_vpodret <> nil) do begin
10         if aktualni^.info <> aktualni_vpodret^.info then begin
11             aktualni := aktualni^.dalsi;
12             aktualni_vpodret := podRet;
13         end else begin
14             aktualni := aktualni^.dalsi;
15             aktualni_vpodret := aktualni_vpodret^.dalsi;
16         end;
17     end;
18     if (aktualni = nil) and (podRet <> nil) then
19         obsahujePodretezec := false
20     else if aktualni_vpodret = nil then
21         obsahujePodretezec := true;
22 end;
```

Protipříkladem může být třeba řetězec *HAHHAHA* a hledaný podřetězec *HAHA*. Funkce srovnává řetězec a podřetězec znak po znaku. Jakmile zjistí, že se v nějakém znaku liší, vrátí se v podřetězci na první znak a v řetězci poskočí o znak vpřed. Jak se bude funkce chovat u našeho protipříkladu? Srovná první znak řetězce a první znak podřetězce. Rovnají se, takže přejde u řetězce i podřetězce na

druhý znak. Ty se taktéž rovnají. Přechází tedy na třetí znaky, i ty se rovnají. Ve čtvrtém znaku se však řetězec a podřetězec liší (v řetězci je čtvrtým znakem *H*, zatímco v podřetězci *A*). Proto se algoritmus posune v podřetězci na první znak a v řetězci na následující (tj. pátý znak) a začne zjišťovat, zda se podřetězec *HAHA* nenachází někde za pátým znakem v řetězci *HAHHAHA* . To je ale kámen úrazu, neboť jak je vidět, podřetězec *HAHA* se v řetězci *HAHHAHA* nachází již za čtvrtým znakem, takže funkce podřetězec v řetězci už nikdy neobjeví.

Zkusme nahradit posloupnost příkazů

```
        end else begin
            aktualni := aktualni^.dalsi;
            aktualni_vpodret := aktualni_vpodret^.dalsi;
        end;
```

posloupností příkazů

```
        end else begin
            aktualni := aktualni^.dalsi;
            aktualni_vpodret := aktualni_vpodret;
        end;
```

tak, že pokud se zjistí, že srovnávané znaky jsou v řetězci a podřetězci jiné, neposunujeme se v řetězci na další znak. Ani toto není správné řešení. Najdete protipříklad?

Nechť je řetězec *XYXYXW* a hledaný podřetězec *XYXW*. Zde nám nepomůže se po zjištění, že se čtvrté znaky liší, v řetězci neposunout na pátý znak a zůstat na čtvrtém, neboť podřetězec *XYXW* začíná na třetím znaku řetězce *XYXYXW*.

Toto je další z pokusů o nápravu funkce `obsahujePodretezec`. Myšlenka je již správná, ale procedura se přesto nechová vždy tak, jak má. Popišme nejprve její princip. Procedura pro každé k, pro které to má smysl, kontroluje, zda podřetězec nezačíná v řetězci na k-tém znaku (v cyklu `repeat-until`). Tato kontrola probíhá tak, že se postupně porovná první znak podřetězce s k-tým znakem řetězce, druhý znak podřetězce s $(k+1)$-ím znakem řetězce, atd., dokud se nezjistí, že se řetězce v některém znaku liší. Až se tak stane, znamená to, že buď

- jsme se ještě nedostali na konec podřetězce, a tedy podřetězec nezačíná na k-tém znaky řetězce, nebo
- už jsme na konci podřetězce, porovnali jsme tedy všechny jeho znaky s příslušnými znaky řetězce, a víme tedy, že podřetězec začíná na k-tém znaku řetězce.

Formulujte co nepřesněji, jaké chyby procedura obsahuje a kdy se projeví.

```
1  function obsahujePodretezec(var ret, podRet: UkPrvek): boolean;
2  var
3      aktualni: UkPrvek;
4      aktualni_vpodret: UkPrvek;
5      nalezeno: boolean;
6  begin
7      nalezeno := false;
8      aktualni := ret;
9      aktualni_vpodret := podRet;
10     repeat
11         while (aktualni^.info = aktualni_vpodret^.info)
12           and (aktualni <> nil) do begin
13               aktualni := aktualni^.dalsi;
14               aktualni_vpodret := aktualni_vpodret^.dalsi;
15           end;
16
17         if (aktualni_vpodret <> nil) then begin
18             nalezeno := false;
19             if (aktualni = nil) then
20                 exit
21             else begin
22                 ret := ret^.dalsi;
23                 aktualni := ret;
24                 aktualni_vpodret := podRet;
25             end;
26         end  else
27             nalezeno := true;
28     until (aktualni = nil)  or (nalezeno = true) ;
29     obsahujePodretezec := nalezeno;
30 end;
```

První chyba je v podmínce cyklu:

```
while (aktualni^.info = aktualni_vpodret^.info)
          and (aktualni <> nil) do begin
```

Tady autor (správně) předpokládá, že `aktualni` může mít hodnotu `nil` (například tehdy, když uživatel zadá prázdný řetězec). Potom však výraz `aktualni^.info` způsobí chybnou dereferenci!

Navíc, může se tady stát i to, že `aktualni_vpodret` bude mít hodnotu `nil` (například tehdy, když uživatel zadá prázdný podřetězec). A pak pro změnu výraz `aktualni_vpodret^.info` způsobí chybnou dereferenci.

Jak to opravit? Pokud vám něco říká termín ‚neúplné vyhodnocování logického výrazu', pak by to pro vás měla být hračka (v Borland Pascalu i Free Pascalu by mělo být implicitně zapnuté). Kdyby ne, prostě mi napište, vysvětlím, o co jde.

TIP

Když testujte svoje programy, je dobré myslet na tzv. ‚analýzu pokrytí testu'. Ten sprostý výraz znamená, že programu zadáte všechny takové kombinace vstupních dat, aby program při svém vykonávání prošel všemi větvemi ve zdrojovém kódu. Tj. zkuste mu zadat

1. prázdný řetězec, neprázdný podřetězec,
2. jednoznakový řetězec, neprázdný podřetězec,
3. neprázdný řetězec, prázdný podřetězec,
4. obojí neprázdné,

5. podřetězec, který se nachází hned na začátku řetězce,
6. podřetězec, který se nachází na konci řetězce,
7. podřetězec, který se v řetězci vůbec nenachází,
8. řetězec, který je delší než podřetězec,
9. podřetězec, který je delší než řetězec,
10. a další možné kombinace vstupů, které vás napadnou.

Zkrátka a dobře, největší problém působí tzv. okrajové případy. Heslovitě by se daly shrnout jako: nula, jednička, prázdný seznam, začátek seznamu/pole, konec seznamu/pole,...

Toto je procedura, která má za úkol vypustit z obousměrného spojového seznamu prvek se zadanou hodnotou. Kde je v ní chyba a jak se tato chyba projeví?

```
1  procedure odstranHodnotu(var s: Seznam; hodnota: Integer);
2
3  var
4      aktualni: UkPrvek;
5      hledane: Integer;
6      nalezeno: boolean;
7
8  begin
9      aktualni := s.zac;
10     hledane := hodnota;
11     nalezeno := false;
12
13     if aktualni = nil then exit;
14
15     {hledani hodnoty}
16     while (nalezeno = false) and (aktualni <> nil) do begin
17         if aktualni^.info = hodnota then begin
18             nalezeno := true;
19             break;
20         end else
21             aktualni := aktualni^.nasl;
22     end;
23
24     {mazani}
25     if nalezeno then begin
26         aktualni^.pred^.nasl := aktualni^.nasl;
27         aktualni^.nasl^.pred := aktualni^.pred;
28         aktualni^.nasl := nil;
29         aktualni^.pred := nil;
30         dispose (aktualni);
31         aktualni := nil;
32     end;
33 end;
```

Tento příklad testuje, jak se dokážete poučit z rad uvedených v této knížce. Před tímto příkladem se píše, že největší problém při testu správnosti procedury je umět odhalit, ověřit a otestovat okrajové podmínky. Zde mám konkrétně na mysli případ, kdy se odstraňovaný prvek nachází hned na začátku nebo na konci spojového seznamu. Pokud se odstraňovaný prvek nachází na začátku seznamu, pak příkaz `aktualni^.pred^.nasl` na řádku 26 způsobí chybnou dereferenci (neboť prvek, na nějž ukazuje ukazatel `aktualni` je prvním prvkem seznamu a nemá tedy předchůdce, tj. `aktualni^.pred` má hodnotu `nil`). Pokud se odstraňovaný prvek nachází na konci seznamu, dojde k obdobné situaci, a to sice na řádku 27.

Tyto dva příkazy se ve zdrojovém kódu nacházejí bezprostředně po sobě. Proč nedávají smysl?

```
new (uzel);
uzel := nil;
```

První příkaz alokuje blok paměti a adresu tohoto bloku uloží do proměnné `uzel`. Druhý příkaz hodnotu této proměnné nastaví na `nil`. Tím ztratíme odkaz na alokovaný blok paměti, takže jej už nikdy nebudeme moci použít. Nepůjde jej ani odalokovat pomocí `dispose`. Blok bude v paměi pouze zabírat místo, dokud náš program neskončí. Nejen, že je první příkaz zbytečný, ale po provedení této dvojice příkazů zůstane v paměti smetí.

Toto je rekurzivní procedura, která má za úkol vypustit ze zadaného seznamu všechny prvky s hodnotou `krit`. Procedura využívá pomocnou proceduru `smaz`, která smaže prvek, na nějž ukazuje její parametr, a tento parametr nastaví na následníka v seznamu. Je procedura `vypustHodnotu` korektní?

```
1  procedure vypustHodnotu(var PrvniPrvek: UkPrvek; krit: Integer);
2  var
3      aktualni,
4      pred: UkPrvek;
5  begin
6      if prvniPrvek <> nil then begin
7          if prvniPrvek^.info = krit then begin
8              smaz(prvniPrvek);
9              vypustHodnotu(prvniPrvek, krit);
10         end else begin
11             pred := PrvniPrvek;
12             aktualni := prvniPrvek^.dalsi;
13             while (aktualni^.info <> krit)
14                   and (aktualni^.dalsi <> nil) do begin
15                 pred := aktualni;
16                 aktualni := aktualni^.dalsi;
17             end;
18             vypustHodnotu(aktualni, krit);
19             pred^.dalsi := aktualni;
20         end;
21     end;
22 end;
```

Nikoliv. Podívejme se na řádky 11 až 13. Nechť seznam obsahuje jediný prvek a ten prvek má hodnotu 6. Nechť chceme z tohoto seznamu vymazat všechny prvky s hodnotou 2. Potom

```
pred := PrvniPrvek;              {pred bude ukazovat na prvek s hodnotou 6}
aktualni := prvniPrvek^.dalsi;   {aktualni bude ukazovat na nil}
while (aktualni^.info <> krit)   {...a tady to sebou rízne - dereference nil}
```

Navíc je procedura zbytečně složitá. Posloupnost příkazů na řádcích 11 až 19 by se totiž dala nahradit *jedním jediným* rekurzivním voláním. Zkuste přijít na to, jak bude toto volání vypadat.

Kapitola 2

Binární vyhledávací stromy

2.1 Rekurzivní operace

Příklady z této kapitoly řešte tak, že do následujícího zdrojového kódu doplníte implementace zadaných procedur. V této kostře je implementovaná i procedura, která strom vykreslí, aby se to dalo dobře ladit (odflákl jsem ji, není napsaná moc elegantně, tj. není to příklad hodný následování).

```
const
    {pocet radek platna}
    ROWS = 20;
    {pocet znaku na radku platna}
    COLUMNS = 80;


type
    {ukazatel na uzel stromu}
    UKUzel = ^Uzel;

    {usel stromu obsahuje zapouzdrenou hodnotu a ukazatele na prave a leve dite}
    Uzel =
        record
            info: Integer;
            left: UKUzel;
            right: UKUzel;
        end;

    {vyuzivano pri vykreslovani stromu}
    TPlatno = array[1..ROWS, 1..COLUMNS] of char;

(*
 * Vlozi do stromu novy prvek.
 * parametry:
 *     koren - ukazatel na koren stromu
 *     hodnota - hodnota, ktera se ma do stromu vlozit
 *)
procedure vloz(var koren: UKUzel; hodnota: Integer);
begin
    if (koren = nil) then begin
        new (koren);
        koren^.info := hodnota;
        koren^.left := nil;
        koren^.right := nil;
    end else if (hodnota < koren^.info) then begin
        vloz(koren^.left, hodnota);
    end else if (hodnota > koren^.info) then begin
        vloz(koren^.right, hodnota);
    end;
end;

```

```
43
44 (*
45  * Pomocna procedura, do platna zakresli na zadanou pozici schematicky strom.
46  * parametry:
47  *     koren - koren vykreslovaneho stromu
48  *     platno - platno, na nejz se bude kreslit
49  *     radek - radek platna, na nejz se zakresli koren;
50  *             ostatni casti stromu se zakresli niz
51  *     zacatek - index nejlevejsiho sloupce platna, kam muze byt zakreslen strom
52  *     sirka - maximalni pocet sloupcu,
53  *             ktere muze vykresleny strom na platne zabirat
54  *)
55 procedure nakresli(
56     var koren: UKUzel; var platno: TPlatno; radek, zacatek, sirka: Integer);
57 var
58     i: Integer;
59     novaSirka, novyZacatek, novyProstredek, prostredek: Integer;
60 begin
61     prostredek := zacatek + sirka div 2 + 1;
62     platno[radek, prostredek - 1] := chr(ord('0') + koren^.info div 100);
63     platno[radek, prostredek] := chr(ord('0') + (koren^.info mod 100) div 10);
64     platno[radek, prostredek + 1] := chr( ord('0') + koren^.info mod 10);
65     novaSirka := sirka div 2;
66     if (koren^.left <> nil) then begin
67         novyZacatek := zacatek;
68         novyProstredek := novyZacatek + novaSirka div 2 + 1;
69         for i := novyProstredek to prostredek do platno[radek + 1, i] := '-';
70         platno[radek + 2, novyProstredek] := '|';
71         nakresli(koren^.left, platno, radek + 3, novyZacatek, novaSirka)
72     end;
73     if (koren^.right <> nil) then begin
74         novyZacatek := zacatek + novaSirka;
75         novyProstredek := novyZacatek + novaSirka div 2 + 1;
76         for i := prostredek to novyProstredek do platno[radek + 1, i] := '-';
77         platno[radek + 2, novyProstredek] := '|';
78         nakresli(koren^.right, platno, radek + 3, novyZacatek, novaSirka)
79     end;
80 end;
81
82
83 (*
84  * Schematicky nakresli strom, kresli na standardni vystup.
85  * parametry:
86  *     koren - ukazatel na koren vykreslovaneho stromu
87  *)
88 procedure nakresliStrom(var koren: UKUzel);
89 var
90     platno: TPlatno;
91     i, j: Integer;
92 begin
93     for i := 1 to ROWS do begin
94         for j := 1 to COLUMNS do begin
95             platno[i, j] := ' ';
96         end;
97     end;
98     nakresli(koren, platno, 1, 0, COLUMNS);
99     for i := 1 to ROWS do begin
100        for j := 1 to COLUMNS do begin
101            write(platno[i, j]);
102        end;
103        writeln;
104    end;
105 end;
106
```

```
107
108 var
109     strom: UKUzel;
110 begin
111     strom := nil;
112     vloz(strom, 50);
113
114     vloz(strom, 25);
115     vloz(strom, 75);
116
117     vloz(strom, 12);
118     vloz(strom, 30);
119     vloz(strom, 60);
120     vloz(strom, 80);
121
122     vloz(strom, 11);
123     vloz(strom, 13);
124     vloz(strom, 26);
125     vloz(strom, 32);
126
127     vloz(strom, 51);
128     vloz(strom, 61);
129     vloz(strom, 79);
130     vloz(strom, 81);
131
132     nakresliStrom(strom);
133
134 end.
```

Implementujte proceduru provádějící levou, resp. pravou rotaci kořene.

Ukážeme si například pravou rotaci. Levá rotace by se provedla analogicky, lze též napsat obecnější funkci, které se směr předá jako parametr (Kryl ukazoval na přednášce).

```
1  procedure rotujDoprava(var koren: UkUzel);
2  var
3      {novy koren}
4      Nkoren: UkUzel;
5  begin
6      if koren = nil or koren^.left = nil then
7          exit;
8      Nkoren := koren^.left;
9      koren^.left := Nkoren^.right;
10     Nkoren^.right := koren;
11     koren := Nkoren;
12 end;
```

Implementujte funkci (vracející hodnotu typu `boolean`), která pro zadaný strom a zadanou hodnotu zjistí, zda se daná hodnota ve stromě nachází, či ne.

```
1  function jeTam(var koren: UkUzel; hodnota:Integer): boolean;
2  begin
3      if koren = nil then begin
4          jeTam := false;
5      end else if koren^.info > hodnota then begin
6          jeTam := jeTam(koren^.left, hodnota);
7      end else if koren^.info < hodnota then begin
8          jeTam := jeTam(koren^.right, hodnota);
9      end else begin
10         jeTam := true;
11     end;
12 end;
```

Implementujte funkci, která vypíše všechny prvky stromu ve vzestupném pořadí.

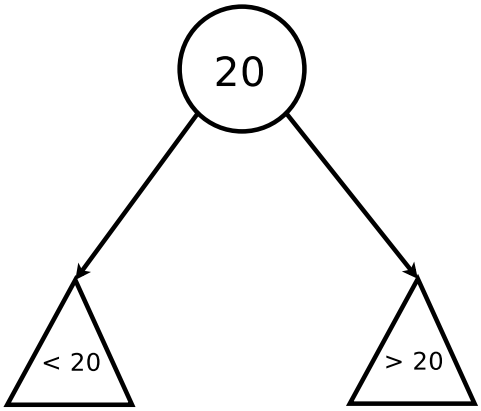
```
procedure vypisPrvkyVzestupne (var koren: UkUzel);
begin if koren <> nil then begin
    vypisPrvkyVzestupne(koren^.left);
    write(' ', koren^.info);
    vypisPrvkyVzestupne(koren^.right);
    end;
end;
```

Implementujte funkci, která přidá zadaný prvek do kořene stromu (dle hesla Rekurze je mocná čarodějka).

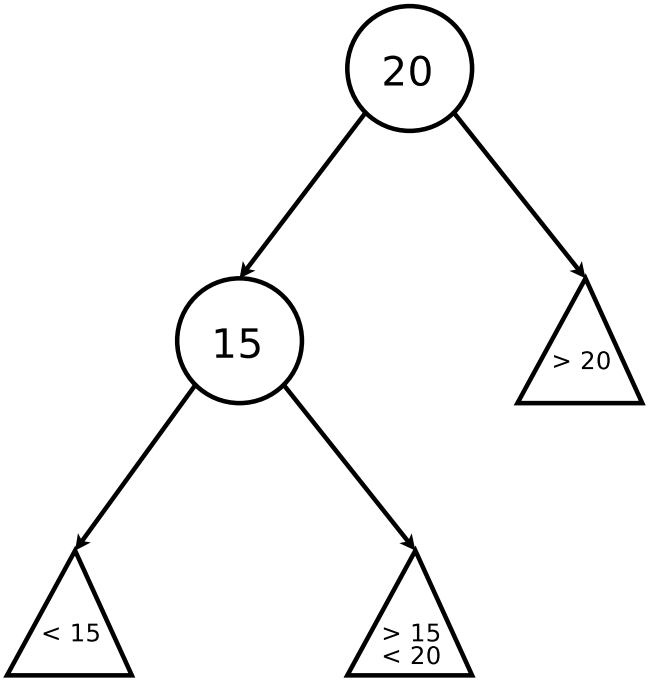
Zkuste nejprve sami přijít na nějaké řešení. Teprve kdybyste dlouhou dobu na nic nepřicházeli, podívejte se na toto řešení. Obrázky po zdrojovým kódem ilustrují vkládání hodnoty 15 do kořene stromu, v jehož kořeni je momentálně hodnota 20.

```
procedure vlozDoKorene(var koren: UkUzel; hodnota: Integer);
begin
    if koren = nil then
        vloz(koren, hodnota)
    else if hodnota < koren^.info then begin
        vlozDoKorene(koren^.left, hodnota);
        rotujDoprava(koren);
    end else if hodnota > koren^.infobegin
        vlozDoKorene(koren^.right, hodnota);
        rotujDoleva(koren);
    end;
end;
```

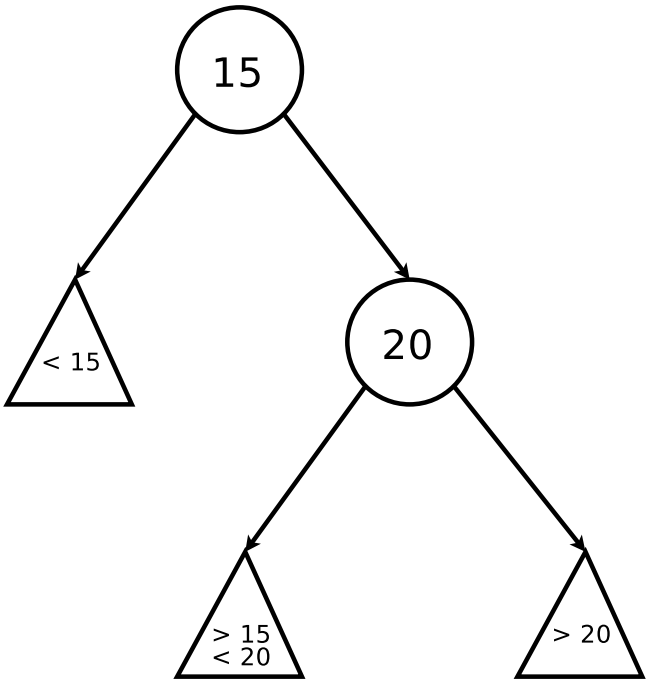
Obrázek 2.1 Před rekurzivním vložením čísla 15



Obrázek 2.2 Po rekurzivním vložení čísla 15



Obrázek 2.3 Po pravé rotaci kořeme



Implementujte funkci `odeberNejpravejsi(var koren: UkUzel): Integer;`, která odebere ze stromu prvek s nejvetší hodnotou (tj. prvek nacházející se co nejvíce vpravo) a jeho hodnotu vrátí. Parametr `koren` je nenilový ukazatel na kořen stromu, z nějž chceme odebrat prvek.

```
1 function odeberNejpravejsi(var koren: UkUzel): Integer;
2 var
3     tmp: UkUzel;
4 begin
5     if (koren^.right <> nil) then begin
```

```
        {existuji prvky vpravo od korene ==> rekurzivne volame na pravy podstrom}
        odeberNejpravejsi := odeberNejpravejsi(koren^.right);
    end else begin
        {vpravo od korene se uz zadny prvek nenachazi, proto vypustime koren}
        tmp := koren;
        koren := koren^.left;
        odeberNejpravejsi := tmp^.info;
        dispose (tmp);
    end;
end;
```

Všimněte si použití parametrů předávaných referencí. To, že je procedura takto napsaná, nám umožní ji zavolat například takto: `odeberNejpravejsi(uzel^.left)`. Ani v případě, že by nejpravějším prvem v levém podstromě uzlu `uzel` byl přímo prvek, na nějž ukazuje ukazatel `uzel^.left`, se volající nemusí starat o aktualizaci ukazatele `uzel^.left`, to zařídí přímo procedura `odeberNejpravejsi`.

Implementujte proceduru `odstranKoren(var koren: UkUzel);`, která odstraní kořen stromu. Parametr kořen je nenilový ukazatel na kořen stromu.

```
procedure odstranKoren(var koren: UkUzel);
var
    tmp: UkUzel;
begin
    if (koren^.left = nil) and (koren^.right = nil) then begin
        {koren nema deti}
        dispose (koren);
        koren := nil;
    end else if (koren^.right = nil) then begin
        {koren ma jen leveho syna, ten se stane novym korenem}
        tmp := koren;
        koren := koren^.left;
        dispose (tmp);
    end else if (koren^.left = nil) then begin
        {koren ma jen praveho syna, ten se stane novym korenem}
        tmp := koren;
        koren := koren^.right;
        dispose (tmp);
    end else begin
        {koren ma dve deti, hodnotu korene nahradime jeho naslednikem
         v in-order usporadani a naslednika vypustime}
        koren^.info := odeberNejpravejsi(koren^.left);
    end;
end;
```

Implementujte proceduru, která z binárního vyhledávacího stromu vypustí prvek se zadanou hodnotou.

```
procedure odstran(var koren: UkUzel; hodnota: Integer);
begin
    {pokud takova hodnota ve strome neni, pozadavek ignorujeme}
    if (koren = nil) then
        exit;

    {hondta je v koreni, vyuzijeme jiz existujici proceduru}
    if (hodnota = koren^.info) then begin
        odstranKoren(koren);
    {hodnota v levem podstrome, rekurzivne volame na levy podstrom}
    end else if (hodnota < koren^.info) then begin
        odstran(koren^.left, hodnota);
    {hodnota v pravem podstrome, rekurzivne volame na pravy podstrom}
    end else if (hodnota > koren^.info) then begin
        odstran(koren^.right, hodnota);
```

```
16      end;
17  end;
```

Kapitola 3

Těžké zkouškové úlohy

3.1 Asfaltování navazujících silnic

V Absurdistánu jsou řádově desítky tisíc měst. Mezi dvěma městy může vést libovolný počet přímých silnic. Všechny silnice jsou šotolinové. Konaly se volby, které vyhrála strana *A*, která slíbila vyasfaltovat všechny silnice. Načež strana *B* se rozhodla, že jí úkol ztíží, a prosadila následující legislativní úpravu:

- v každém dni se mohou vyasfaltovat právě dvě na sebe navazující silnice (tj. silnice z města x do města y a silnice z města y do města z, pro x, y, z různá),
- žádná silnice se nesmí asfaltovat dvakrát.

Vládní strana *A* proto zadala softwarové firmě, aby problém vyřešila. SW firma jste vy. Pro která vstupní data má úloha řešení? Jde-li to, vytvořte harmonogram prací.

Jakou matematickou abstrakci použijete při řešení tohoto problému?

Nabízí se použití grafů. Vrcholy grafu budou města, hranami pak přímé silnice vedoucí mezi dvěma městy. Jediná potíž je v tom, že mezi dvěma městy může vést více než jedna přímá silnice. V grafu se pak mezi dvěma vrcholy může objevit více než jedna hrana. I s takovým grafem se dá pracovat. Odborně se mu říká *multigraf*.

Je nutné se při řešení této úlohy zabývat zvlášť grafy souvislými a nesouvilslými?

Zkusme otázku přeformulovat. Stačí nám umět řešit tuto úlohy pro souvislé grafy, abychom ji jednoduše vyřešili i pro nesouvislé grafy? Odpověď zni *ano*. Uvědomte si totiž jednoduchou vlastnost: *graf lze vyasfaltovat, právě když lze vyasfaltovat každou jeho komponentu souvislosti*. Důkaz je triviální. Pokud umíme vyasfaltovat celý graf, pak snadno vyasfaltujeme i každou jeho komponentu souvislosti. Naopak, jestliže umíme vyasfaltovat každou komponentu souvislosti, vyasfaltujeme celý graf komponentu po komponentě.

Proto nás stačí, když umíme úlohu vyřešit jen pro souvislé grafy. Nesouvislý graf vyasfaltujeme tak, že algoritmus asfaltování rekurzivně pustíme na všechny komponenty souvislosti.

Kdy graf vyasfaltovat určitě nepůjde? Pokuste se o co nejobecnější charakteristiku.

Z pravidel uvedených v legislativní úpravě plyně, že každý den lze vyasfaltovat právě dvě nevyasfaltované silnice. Celkový počet asfaltovaných silnic tedy musí být sudé číslo. Kdyby měl graf lichý počet hran, pak by jej určitě nebylo možné vyasfaltovat. A co víc, v předchozím úkolu jsme si řekli, že graf je vyasfaltovatelný, právě když je každá jeho komponenta souvislosti vyasfaltovatelná. Obecná charakteristika grafů, které nelze vyasfaltovat, by tedy mohla znít třeba takto: ‚Graf nelze vyasfaltovat, jestliže některá z jeho komponent souvislosti má lichý počet hran.'

> Dále se pokusíme předešlé tvrzení zesílit na: ‚Graf lze vyasfaltovat, *právě když* každá z jeho komponent souvislosti má sudý počet hran.'

Jaký postup navrhujete pro důkaz tohoto tvrzení?

Obvykle se v důkazech podobných tvrzení postupuje tak, že se nejprve problém rozdělí na několik možných případů. A rozumné bývá rozlišit dva případy:

- graf je acyklický,
- graf není acyklický.

Toho se budeme držet i v našem případě.

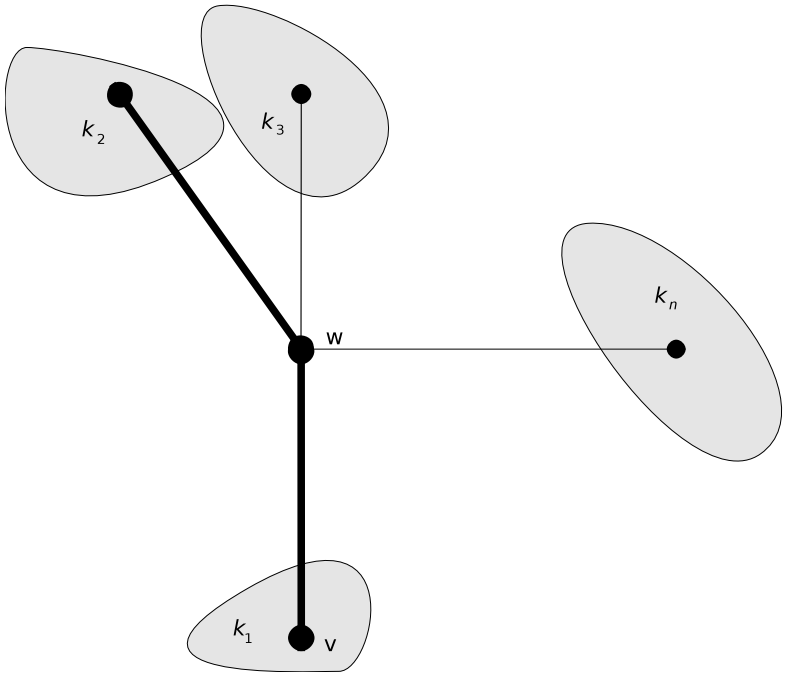
Dále, uvědomte si, že tvrzení se nám bude krásně dokazovat pomocí matematické indukce. Je-li totiž dán souvislý graf se sudým počtem hran, pak se můžeme pokusit z něj dvě na sebe navazující hrany odtrhnout (tj. označit za vyasfaltované) a dostaneme opět graf se sudým počtem hran. Jen se při odebrání těchto dvou hran graf nesmí rozpadnout na více komponent, z nichž aspoň jedna by měla lichý počet hran.

Nechť je dán strom se sudým nenulovým počtem hran. Napadá vás, jak z něj odtrhnout dvě navazující hrany tak, aby se strom *nerozpadl* na více komponent souvislosti, z nichž aspoň jedna by měla lichý počet hran?

Strom má sudý nenulový počet hran, má tedy aspoň dvě hrany. Z toho plyne, že musí mít aspoň tři vrcholy. A známé trvrzení říká, že každý strom s alepoň dvěma vrcholy má alespoň dva listy. Ve stromě tedy musí existovat list.

Označme libovolný list stromu písmenem v. Označme jeho bezprostředního souseda písmenem w. Označme k_1, k_2,... k_n komponenty, na které by se graf rozpadl, kdybychom odstranili vrchol w. Nechť bez újmy na obecnosti je k_1 komponenta, kterou tvoří jediný vrchol, a to vrchol v. Kdyby měly všechny komponenty k_2,... k_n lichý počet hran, pak by měl i celý graf lichý počet hran (pokud ke každé z těchto komponent přidáme i hranu, která ji spojuje s vrcholem w, bude mít každá z těchto komponent sudý počet hran, a protože v grafu se nachází ještě hrana (v, w), bude mít graf celkově lichý počet hran). Proto aspoň jedna z komponent k_2,... k_n má sudý počet hran, nechť je to bez újmy na obecnosti k_2. Odstraňme z grafu hranu (v, w) a hranu spojující vrchol w s komponentou k_2. Tím jsme odstranili dvě na sebe navazující hrany, aniž by se nám graf rozpadl na více komponent, z nichž by některá měla lichý počet hran.

Obrázek 3.1 Odtrhávání dvojice hran ze stromu



3.1.1 Úkoly pro vás

Toto byl úvod, který vám měl dát základní vodítko pro řešení úlohy. Úloha však zatím zdaleka není vyřešená. Minimálně je třeba vyřešit tyto problémy.

1. Proveďte podobný rozbor, jako byl předveden pro stromy, i pro cyklické grafy.
2. Dokažte tvrzení: ‚Graf lze vyasfaltovat, *právě když* každá z jeho komponent souvislosti má sudý počet hran.‘
3. Navrhněte algoritmus pro řešení naší úlohy. Jakou roli v něm bude hrát rekurze a jak rekurze souvisí s důkazem tvrzení?
4. Jakým způsobem budete reprezentovat graf?
5. Jakým způsobem budete graf dělit na komponenty?
6. Jak budete z grafu odebírat dvojici hran?
7. Napište kostru datových struktur, kterými budete reprezentovat graf.
8. Napište kostru základních procedur.
9. Pokuste se spočítat časovou a prostorovou složitost algoritmu.