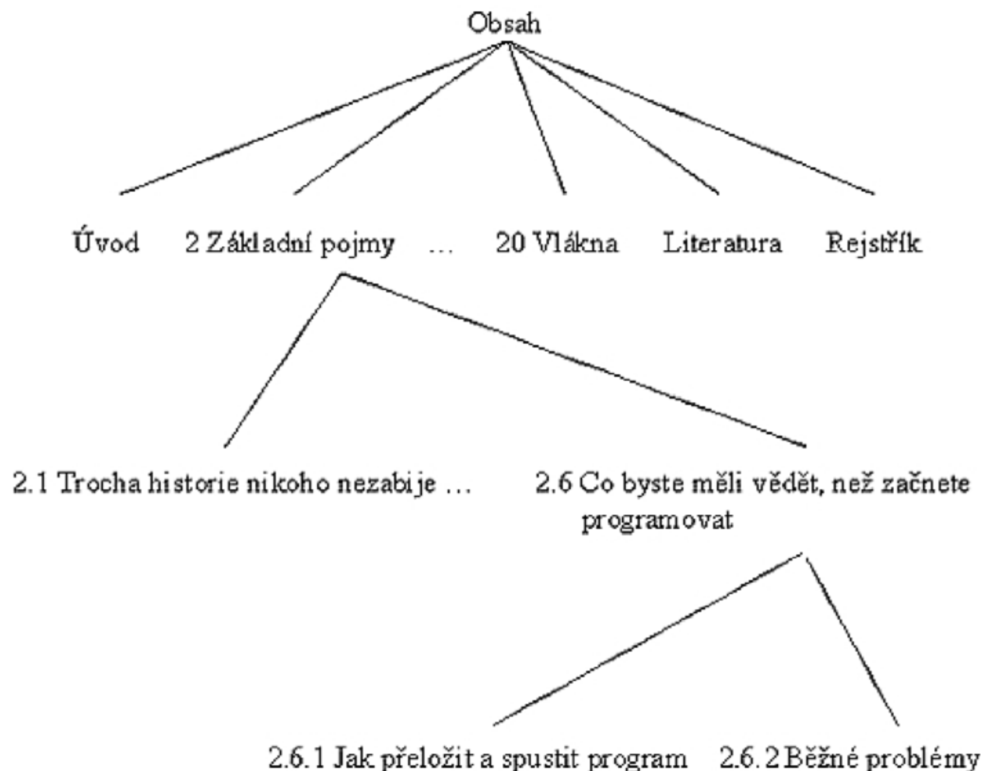
Základní pojmy

Strom je matematická abstrakce organizace dynamických množin, kterou v běžném životě používáme (možná nevědomě) velice často.

Příkladem může být rodokmen, tedy například struktura z druhé kapitoly, omezíme-li se na své rodiče, rodiče rodičů atd.. Podobně průběh sportovní soutěže založené na vylučovacím principu, kdy dál postupuje jeden ze soupeřů, lze vyjádřit stromem. Také organizaci knížky (obr. 6.1), která obsahuje kapitoly, ty podkapitoly atd. lze znázornit pomocí stromu. Na obr. je příklad pro knížku Herout P.: Učebnice jazyka Java.



Obr. 6.1 Stromová organizace knihy

V počítači může být ve formě stromu organizován systém souborů uložených v adresářích, které definujeme rekurzivně jako množinu souborů a adresářů.

Prvky v dynamické množině organizované jako strom nazýváme ***vrcholy***.
Některé vrcholy jsou spojeny a toto spojení nazýváme ***hranou***.

Strom potom můžeme rekurzivně definovat následovně

> *a) Jeden vrchol je strom. Tento vrchol je také kořenem takovéhoto stromu.*
> *b) Nechť x je vrchol a T_1, T_2, ..., T_n jsou stromy. Strom je vrchol x spojený s kořeny stromů T_1, T_2, ..., T_n.*

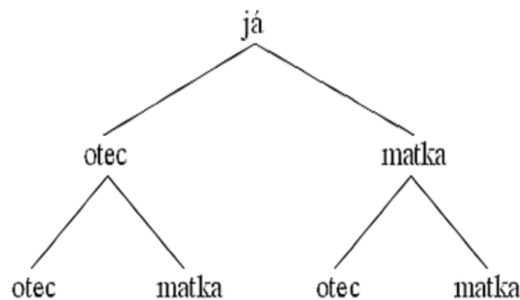
V tomto stromu je x ***kořenem stromu***. Stromy T_1, T_2, ..., T_n se nazývají ***podstromy***. Jejich kořeny jsou (*přímými*) ***následníky*** vrcholu *x* a vrchol *x* je jejich (*přímým*) ***předchůdcem***.
Někdy je vhodné zahrnout mezi stromy i prázdnou množinu vrcholů. Vrchol, který nemá následníky se nazývá ***listem***. Vrchol, který není listem se nazývá ***vnitřním vrcholem***. Kořen stromu nemá žádné předchůdce.

Cesta je posloupnost vrcholů, ve které po sobě jdoucí vrcholy jsou spojeny hranou.

Délka cesty je počet hran cesty. Délku cesty z vrcholu k sobě samému potom můžeme definovat jako nulovou. Ke každému vrcholu je z kořene právě jedna cesta.

Hloubka vrcholu ve stromě (úroveň, na které se nachází) je definována jako délka této cesty. Úroveň (hloubka) kořene stromu je tedy nulová.

Výška stromu je maximální hloubka vrcholu stromu.
Množina přímých následníků může být uspořádaná, například při grafickém zobrazení zleva doprava. V příkladu rodokmenu (obr. 6.2) pokud je vlevo otec a vpravo matka, je takovýto strom různý od stromu, ve kterém je vlevo matka a vpravo otec, i když oba stromy na jednotlivých úrovních mají tytéž prvky.



Obr. 6.2 Rodokmen

Důležitou třídou takových stromů jsou binární stromy, kterými se budeme zabývat v následujícím kapitole.

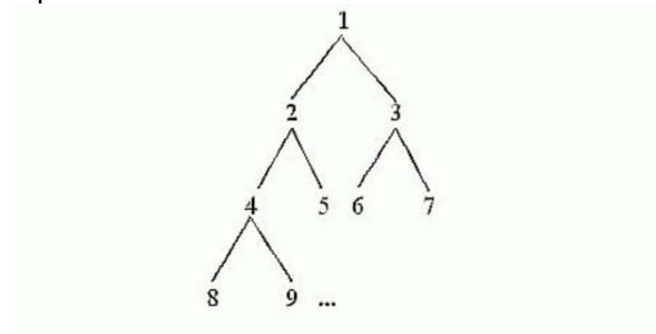
Binární stromy

Vyjdeme-li z obecné definice stromu a základních pojmů, můžeme binární stromy definovat následovně:

***Binární strom je prázdný strom** (vrchol), nemá-li levý a pravý podstrom, které jsou binární stromy.*

Modelem dat pro binární strom může být ***implementace polem***, jehož prvky mají typ klíče prvku.

Pozice vrcholů binárního stromu lze očíslovat následujícím způsobem:
Začneme kořenem, kterému přiřadíme 1, dále jeho levému následníku 2, pravému následníku 3, a v číslování pokračujeme na každé úrovni zleva až do jejího konce, kde přejdeme na další úroveň (obr. 6.3). Tato čísla pozic vrcholů jsou pak indexy odpovídajících prvků pole.

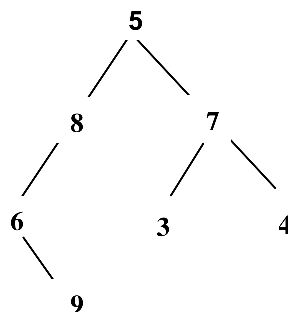


Obr. 6.3 Označení vrcholů binárního stromu

Obecně platí:

> *Má-li pozice vrcholu hodnotu indexu **i**, potom*
> *levý následník má index **2i***
> *pravý následník má index **2i + 1***
> *předchůdce, pokud existuje, má index **i/2** (celočíselně).*

Poznamenejme, že uvedené vztahy platí, má-li kořen stromu index *1*. Pokud bychom ho umístnili do prvku pole s indexem *0*, bylo by nutno tyto vztahy upravit.
Pokud se na pozici skutečně nachází vrchol, prvek pole obsahuje klíč. V opačném případě je prvek pole označen jako nepoužitý.
Například jsou-li hodnoty vrcholů nezáporná celá čísla, může být jeho hodnota *-1*.
Binární strom na obr. 6.4, kde ve vrcholech jsou hodnoty klíčů prvků, nemá obsazeny pozice s indexy *5, 8, 10* a všechny další.



Obr. 6.4 Příklad binárního stromu

Jeho reprezentace polem s klíči prvků by tedy byla

```
5, 8, 7, 6, -1, 3, 4, -1, 9, -1, ...
```

Obecně tato implementace není efektivní, protože nejenom musíme vytvořit pole pro předpokládanou maximální velikost stromu, ale navíc musí pole obsahovat i prvky pro pozice neobsazených vrcholů.

Vrchol binárního stromu můžeme ***implementovat*** obdobně jako ***prvek seznamu***, pouze místo položky `dalsi` budou v implementaci vrcholu polozky `levy` a `pravy` ukazující na kořen levého a pravého podstromu. Odpovídající třída `Vrchol` je na obr. 6.5. Prázdný strom bude reprezentován hodnotou `null`.

```
class Vrchol {
  int klic;
  Vrchol levy;
  Vrchol pravy;

  ...

  void tiskVrcholu() {
    System.out.print(klic+" ");
  }
}
```

Obr. 6.5 Implementace binárního stromu seznamem - třída `Vrchol`

Průchod stromem

V případě seznamu, když jsme potřebovali projít (navštívit) všechny jeho prvky, například k jejich vytištění, postupovali jsme od prvního k dalším pomocí ukazatele **`dalsi`**. Připomeňme si rekurzivní metody `pruchod()` a `pruchodR()` z kapitoly 4.2, v nichž jsme v průchodu pokračovali rekurzivním voláním `pruchod(x.dalsi)` resp. `pruchodR(x.dalsi)`.

V případě stromu začneme kořenem, ale pro další systematický postup máme tři možnosti:

Přímý průchod (preorder)- navštívíme vrchol, potom levý a pravý podstrom
Vnitřní průchod (inorder)- navštívíme levý podstrom, vrchol a pravý podstrom
Zpětný průchod (postorder)- navštívíme levý a pravý podstrom a potom vrchol

Vrcholy stromu, který je zobrazen na obr. 6.4, pak projdeme jednotlivými průchody v následujícím pořadí:

přímý průchod: **5, 8, 6, 9, 7, 3, 4**
vnitřní průchod: **6, 9, 8, 5, 3, 7, 4**
zpětný průchod: **9, 6, 8, 3, 4, 7, 5**

Průchody binárním stromem, vycházejíme-li z jeho rekurzivní definice, můžeme vyjádřit rekurzivní metodou `pruchodR()` na obr. 6.6.

```
void pruchodR(Vrchol v) {
  if (v == null)
    return;
  v.tiskVrcholu();
  pruchodR(v.levy);
  pruchodR(v.pravy);
}

Vrchol koren;
pruchodR (koren);
```

Obr. 6.6 Rekurzivní průchod stromem

Uvedená metoda implementuje průchod ***preorder***. Posunutím řádku s tiskem mezi její rekurzivní volání získáme implementaci průchodu ***inorder*** a jeho posunutím za obě rekurzivní volání získáme implementaci průchodu ***postorder***.

Čas průchodu stromem je pro prázdný strom dán vyhodnocením podmínky, tedy $T(0) = cpor$. Trvá-li tisk *ctisk* a má-li levý podstrom *m* vrcholů a pravý podstrom *n-m-1* vrcholů, potom $T(n) = T(m) + T(n-m-1) + cpor + ctisk$ pro $n > 0$

Postupem známým z kapitoly 4.2 můžeme ukázat, že její řešení je

$T(n) = (2cpor + ctisk)n + cpor$ a čas průchodu stromem je tedy $\Theta(n)$.

Víme již, že rekurzi můžeme odstranit obecně pomocí zásobníku. Použitím abstraktního zásobníku, do kterého můžeme ukládat klíče i stromy, můžeme vytvořit iterační (nerekurzivní) implementace průchodů binárním stromem:

1. *Do zásobníku vložíme procházený strom*
2. *Dokud zásobník není prázdný, v cyklu vybereme prvek ze zásobníku a je-li to klíč, vytiskneme jej, jinak do zásobníku vložíme:*

*pro **preorder**: pravý podstrom, levý podstrom, klíč vrcholu*
*pro **inorder**: pravý podstrom, klíč vrcholu, levý podstrom*
*pro **postorder**: klíč vrcholu, pravý podstrom, levý podstrom*

Prázdné stromy do zásobníku neukládáme.

Pro ***preorder*** průchod je při vkládání jako poslední vložen klíč, je tedy vybrán na začátku uvedeného cyklu a vytisknut. Můžeme tedy vytisknout kořen každého stromu, který vybereme ze zásobníku a do zásobníku vložit pravý a levý podstrom. Zásobník bude tedy obsahovat pouze stromy, přesněji reference na jejich kořeny. Iterační *preorder* průchod je na obr. 6.7.

```
void pruchod(Vrchol v) {
  VZasobnik z = new VZasovnik();
  z.push(v);
  while (!z.jePrazdny()) {
    v = z.pop();
    v.tiskVrcholu();
    if (v.pravy != null) z.push(v.pravy);
    if (v.levy != null) z.push(v.levy);
  }
}
```

Obr. 6.7 Iterační průchod stromem

Uvedené průchody binárním stromem vycházely z jeho rekurzivní definice. Na druhé straně z pohledu našeho obvyklého procházení textu, tj. shora dolů a zleva doprava, dalším přirozeným průchodem je průchod stromem ***po úrovních*** a v jejich rámci zleva doprava. Průchod stromem po úrovních dosáhneme záměnou zásobníku v programu na obr. 6.7 frontou podstromů, které zbývá navštívit (obr. 6.8).

```
void pruchod(Vrchol v) {
  VFronta f = new VFronta();
  f.vloz(v);
  while (!f.jePrazdny()) {
    v = f.vyber();
    v.tiskVrcholu();
    if (v.levy != null) f.vloz(v.levy);
    if (v.pravy != null) f.vloz(v.pravy);
  }
}
```

Obr. 6.8 Průchod stromem po úrovních

Poznamenejme, že jsme ukázali zmíněné (v kapitole 3.2) využití zásobníku a fronty při konstrukci dalších algoritmů, v našem případě pro procházení binárním stromem.

Binární vyhledávací stromy (BVS)

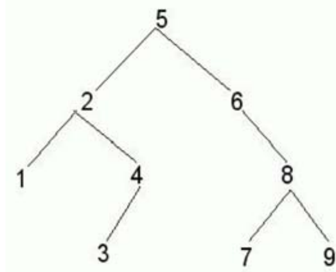
Vkládání prvků do uspořádaných množin implementovaných polem nebo seznamem je, jak jsme viděli, časově náročné, protože musíme udržet jejich uspořádání. Na druhé straně, pokud vkládáme to těchto implementací dynamických množin prvky bez ohledu na jejich uspořádání, operace vložení jsou rychlé, ale hledání prvků se

stane časově náročné. Pro BVS, jak uvidíme, jsou v průměrném případu obě tyto operace efektivní.

Prvky jsou ve vrcholech BVS uspořádány tak, že splňují následující **BVS vlastnost**:

> ***Nechť x je vrchol stromu.***
> ***Je-li y vrchol v levém podstromu, potom y.klíč < x.klíč.***
> ***Je-li y vrchol v pravém podstromu, potom y.klíč > x.klíč.***

Pro BVS budeme uvažovat, že klíče všech prvků jsou navzájem různé. Příklad BVS je na obr. 6.9.



Obr. 6.9 Binární vyhledávací strom

Vlastnost BVS umožňuje získat ***seřazené klíče*** ve vrcholech stromu jednoduše ***inorder průchodem***. Inorder průchod vytiskne klíče stromu na obr. S.9 v pořadí:

1, 2, 3, 4, 5, 6, 7, 8, 9.

Pro doplnění uvedeme i zbylé dva průchody – *preorder:* 5, 2, 1, 4, 3, 6, 8, 7, 9 a *postorder:* 1, 3, 4, 2, 7, 9, 8, 6, 5.

Základní operací nad BVS, je hledání hodnoty uložené ve vrcholu podle klíče. Třída `DVrchol` tedy bude obsahovat navíc člen
`data`, například typu `String` (obr. 6.10).

```
private class DVrchol {
  int klic;
  String data;
  DVrchol levy;
  DVrchol pravy;

  DVrchol (int klic, String data) {
    this.klic = klic;
    this.data = data;
  }

  void tiskVrcholu() {
    System.out.print(data+" ");
  }
}
```

Obr. 6.10 Třída `DVrchol`

BVS strom je reprezentován svým kořenem

```
private DVrchol koren;
```

Pro prázdný strom je

```
koren == null;
```

Hledání prvku v BVS

V našem případě je signatura metody `hledej()` podle klíče

```
String hledej(int)
```

Z definice BVS přímo plyne její rekurzivní implementace na obr. 6.11.

```
private String hledejR(DVrchol v, int klic) {
  if (v == null)
    return null;
  if (klic == v.klic)
    return v.data;
  if (klic < v.klic)
    return hledejR(v.levy, klic);
  else
    return hledejR(v.pravy, klic);
}

String hledej(int klic) {
  return hledejR(koren, klic);
}
```

Obr. 6.11 Rekurzivní hledání

Pokud hledaný prvek není kořenem, rozdělíme množinu prvků vyjádřenou BVS na dvě podmnožiny, tj. na levý a pravý podstrom a podle toho, je-li klíč hledaného prvku menší nebo větší než kořen určíme, ve kterém podstromu budeme pokračovat. Hledání v BVS je tedy z hlediska časové složitosti stejně efektivní jako binární hledání v uspořádané množině vyjádřené pomocí pole, kde jsme další hledání určovali podle prostředního prvku.

Program na obr. S.11 obsahuje koncovou rekurzi, můžeme ho tedy známým způsobem přepsat použitím příkazu cyklu (obr. 6.12).

```
String hledej(int klic) {
  DVrchol x = koren;
  while (x != null && klic != x.klic)
    if (klic < x.klic )
      x = x.levy;
    else
      x = x.pravy;
  return x == null ? null : x.data;
}
```

Obr. 6.12 Rekurzivní hledání

Tento program bude na většině počítačů rychlejší.

Minimum a maximum

Kromě hledání prvku podle klíče, jsme například v kapitole 4.2 uvedli řešení hledání nejmenšího prvku v poli. Obdobná úloha je nalézt prvek s nejmenším klíčem v BVS. Pro neprázdný strom to znamená sledovat ukazatel na levý podstrom, ve kterém jsou všechny prvky stromu menší než jeho kořen. Dále postupujeme stejně, až levý strom je prázdný. Odpovídající metoda `minKlic()` je na obr. S.13. Metoda pro nalezení maximálního prvku `maxKlic()` je symetrická a je uvedena také na obr. 6.13.

```
int minKlic() {
  DVrchol x = koren;
    while (x.levy != null)
      x = x.levy;
  return x.klic;
}

int maxKlic() {
  DVrchol x = koren;
  while (x.pravy != null)
    x = x.pravy;
  return x.klic;
}
```

Obr. 6.13 Minimum a maximum

Vkládání a výběr

Pro BVS je významné, že stejně efektivně jako hledání prvků lze implementovat i jejich vkládání. Obdobně jako u hledání prvku, projdeme strom od kořene k místu, kam má být prvek vložen. Pro vybírání prvku je opět vhodné uvědomit si analogii práce se seznamem, kde jsme si řekli, že je užitečné, aby prvek obsahoval ukazatel na předchůdce. Rozšíříme tedy i prvek BVS o ukazatel na předchůdce. Modifikovaná třída DVrchol je na obr. 6.14.

```
private class DVrchol {
  int klic;
  String data;
  DVrchol levy;
  DVrchol pravy;
  DVrchol predch;

  DVrchol (int klic, String data) {
    this.klic = klic;
    this.data = data;
  }

  void tiskVrcholu() {
    System.out.print(data+" ");
  }
}
```

Obr. 6.14 Třída `DVrchol` s předchůdcem

Signatura metody `vloz` je v našem případě

`void vloz(int, String)`

Její implementace je na obr. 6.15. Pro vložení prvku s klíčem `klic` ukazatel `x` prochází strom levým nebo pravým podstromem podle porovnání hodnot `klic` a `x.klic` až dosáhne hodnotu `null`, kam patří vkládaný prvek. Proměnná `predch` obdobně jako u seznamu udržuje jeho předchůdce, který je uložen do položky `predch` vkládaného prvku. Nakonec je nastavena hodnota `predch.levy` nebo `predch.pravy` na vkládaný prvek.

Pro vybrání prvku ze stromu musíme rozlišovat tři případy.

Za prvé, odebíraný prvek je listem, kdy odpovídající ukazatel v jeho přímém předchůdci nastavíme na `null`, čímž je odpojen od stromu. Příkladem je vrchol s klíčem *1* na obr. 6.16.

```
void vloz(int klic, String data) {
  DVrchol x = koren, predch = null;
  while (x != null ) {
    predch = x;
    if (klic < x.klic)
      x = x.levy;
    else
      x = x.pravy;
  }
  DVrchol z = new DVrchol(klic, data);
  z.predch = predch;
  if (predch == null)
    koren = z;
  else if (klic < predch.klic)
         predch.levy = z;
       else
         predch.pravy = z;
}
```

Obr. 6.15 Vložení prvku

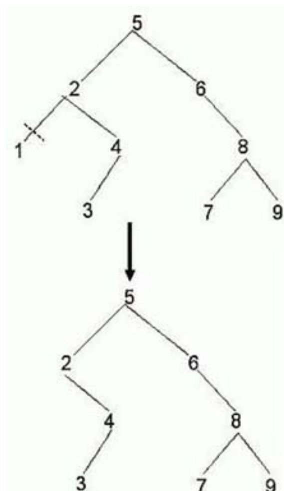
Za druhé, odebíraný prvek má právě jednoho následníka a je vybrán, stejně jako tomu bylo u seznamu jeho odpojením, změnou odpovídajících ukazatelů. Příkladem je vrchol s klíčem *6* na obr. 6.17. Jedním z těchto ukazatelů může být ukazatel na kořen stromu, je-li vybíraným prvkem kořen s právě jedním potomkem.

Třetí případ je, má-li vybíraný prvek dva následníky, kdy pozici tohoto vrcholu musíme zachovat obsazenou, abychom udrželi strukturu BVS. Jeho odebrání provedeme tak, že jeho klíč a data přepíšeme hodnotami jiného prvku a tento odpojíme. Přitom musíme zachovat vlastnost BVS. Uvědomíme-li si, že pravý podstrom odebíraného prvku obsahuje prvky s klíči většími, než je klíč tohoto prvku, můžeme ho nahradit minimem pravého stromu, čímž bude zachována vlastnost BVS a tento prvek odpojit. Příklad je na obr. 6.18, kde je ze stromu vybrán prvek s klíčem *2* tak, že se do něj zkopírují hodnoty prvku s klíčem *3* a tento se odpojí. Zde si třeba uvědomit, že minimum uvedeného podstromu může být i jeho kořen.

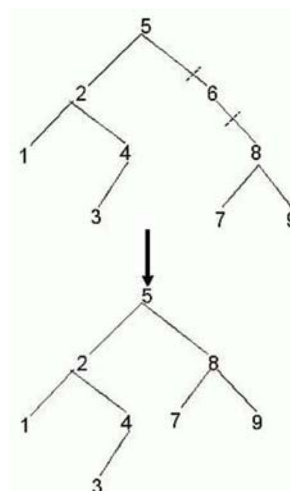
Signatura metody `vyber()` je v našem případě

```
void vyber(int)
```

Její implementace je na obr. 6.19.

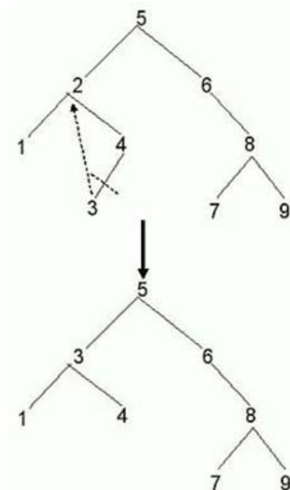


Obr. 6.16 Výběr prvku - 1. případ



Obr. 6.17 Výběr prvku - 2. případ

Na obr. S.19 nejprve nalezneme vrchol se zadanou hodnotou klíče. Postup je stejný jako v iterační metodě `hledej()`, přičemž po jeho nalezení na něj ukazuje proměnná `z`. Dále určíme, zda při jeho výběru bude odpojen tento vrchol nebo vrchol s nejmenší hodnotou klíče jeho pravého podstromu. Ukazatel na odpojovaný uzel je uložen v proměnné `y`. Pro jeho odpojení uložíme na `x` ukazatel na jeho následníka anebo `null`, pokud ho nemá. Potom následuje odpojení určeného prvku, přičemž jsou zohledněny situace, na které jsme upozornili při popisu algoritmu, což možná činí kód trochu složitější. Pokud odpojený vrchol nebyl prvkem, který máme ze stromu vybrat, zkopírujeme jeho hodnoty do odebíraného vrcholu (třetí případ). Nakonec uvolníme referenci na odpojený vrchol pro garbage collector.



Obr. 6.18 Výběr prvku – 3. případ

```
void vyber(int klic) {
  // najdeme vrchol z na vybrani ze stromu
  DVrchol z = koren;
  while (z != null && klic != z.klic)
    if (klic < z.klic )
      z = z.levy;
    else
      z = z.pravy;

  // urciem vrchol y na odpojeni
  DVrchol y = z;
  if (z.levy != null && z.pravy != null) {
    y = z.pravy;
    while (y.levy != null)
      y = y.levy;
  }

  // x ukazuje na potomka y anebo je null, kdyz nema
  // zadneho potomka
  DVrchol x;
  if (y.levy != null)
    x = y.levy;
  else
    x = y.pravy;
```

```
    // modifikaci y.predch a x odpojime y
    if (x != null)
      x.predch = y.predch;
    if (y.predch == null)
      koren = x;
    else
      if (y == y.predch.levy)
        y.predch.levy = x;
      else
        y.predch.pravy = x;

    // nebyl-li odpojen z, skopirujeme klic a data
    if (y != z) {
      z.klic = y.klic;
      z.data = y.data;
    }

    // uvolnime y
    y = null;
  }
```

Obr. 6.19 Výběr prvku – zdrojový kód

Uvedený způsob výběru prvku ze stromu s modifikací stromu není jediný. Jiným přístupem, nevyžadujícím modifikaci stromu, je označit vybraný prvek jako neplatný, například použitím logické členské proměnné v třídě `DVrchol`. Často totiž v aplikaci nemáme mnoho prvků, které jsou ze stromu vybírány, a dokonce můžou být užitečné, pokud by jich bylo později opět potřeba.

Další technikou je vytvoření nového stromu pro platné prvky, dosáhl-li počet neplatných vrcholů nějakou mez, což je také častý obecný způsob práce s datovými strukturami.

Vlastnosti BVS

Všechny operace nad BVS uvedené v tomto článku procházely stromem od kořene nejvíce k jeho listům. Z toho plyne, že čas jejich výpočtu je *O(h),* kde *h* je hloubka stromu. Jaká je tedy výška stromu vytvořeného z náhodné permutace *n* různých prvků?

Nejhorší případ je zřejmě $h = n - 1$, kdy strom degeneruje na seznam, protože prvky v permutaci byly uspořádané. Časová složitost uvedených operací tedy je ***O(n)***.

Nejlepší případ nastane, když prvky budou vloženy tak, že kromě poslední úrovně, jsou zaplněny všechny vyšší úrovně. Pro *h* potom platí

$$2h \le N < 2h+1$$

a $h \approx \log_2 n$. Čas výpočtu uvedených operací tedy je ***O(log2 n)***.

Jaký je čas výpočtu těchto operací pro obecný BVS s *n* vrcholy? Čas uvedených operací je dán hloubkou pozice vkládání a hledání. Zavedeme-li celkovou délku cest *P(S)* binárního stromu *S* jakou součet hloubek všech vrcholů, průměrná hloubka vrcholu ve stromu *S* je *P(S)/n.* Pro *n* vrcholů máme *n!* permutací, z kterých můžeme vytvořit stromy. Obecně v nich bude průměrná hloubka vrcholu různá, můžeme ovšem určit, jaká bude pro průměrný případ.

Je-li kořenem BVS *i*-tý největší prvek, potom v levém podstromě je *i-1* vrcholů a v pravém podstromě *n-i* vrcholů. Označme P_n průměrnou celkovou délku cest BVS s *n*

vrcholy. Vznikl-li vkládáním posloupnosti *n* prvků, přičemž všechny jejich permutace jsou stejně pravděpodobné, může být kořenem každý z *n* prvků se stejnou pravděpodobností. Potom platí

$$P_n = \frac{1}{n} \sum_{1 \le i \le n} (P_{i-1} + i - 1 + P_{n-i} + n - i), \qquad P_0 = 0,\ P_1 = 1$$

kde členy *i - 1* a *n - i* jsou příspěvky kořene k délce cest v levém a pravém podstromu. Její úpravou je

$$P_n = n - 1 + \frac{2}{n} \sum_{0 \le k \le n-1} P_k$$

Řešením dostaneme

Pn ≈ 2n ln n = 1,39n log2 n

V průměrném případu je tak průměrná hloubka vrcholu *1,39 log2 n*. Čas výpočtu základních operací nad BVS v průměrném případu je tudíž ***O(log2 n)***. Poznamenejme, že vybrat prvek ze stromu lze několika způsoby. Obecně vedou k tomu, že strom po odstranění vrcholu nezůstává náhodným, Někdy se průměrná hloubka změní na $\sqrt{n}$.
V předcházejících úvahách jsme předpokládali, že všechny permutace mají stejnou pravděpodobnost a každé odpovídá nějaký BVS. Odpovídají však každé permutaci různé BVS?
Uložme prvky posloupnosti do mřížky tak, že řádek odpovídá hodnotě a sloupec poloze.

Pro posloupnost *3,5,2,4,1* má tato mřížka tvar

x x x x 1
x x 2 x x
3 x x x x
x x x 4 x
x 5 x x x

Získali jsme mřížkovou reprezentaci BVS. V prvním sloupci je jeho kořen, kořen levého podstromu je nejbližší sloupec s prvkem v řádku nad ním a kořen pravého podstromu je nejbližší sloupec s prvkem v řádku pod ním, atd.

Výměnou sloupců odpovídajících vrcholům nad a pod libovolným vrcholem se vyhledávací strom nezmění, zatímco posloupnost, která ho tvoří ano.

Vyměníme-li poslední dva sloupce, získáme mřížku

x x x 1 x
x x 2 x x
3 x x x x
x x x x 4
x 5 x x x

a odpovídající posloupnost je *3,5,2,1,4.* Konkrétní BVS je tedy tvořen více posloupnostmi.Počet všech posloupností z *n* prvků je *n!*. Lze ukázat, že počet různých binárních stromů s *n* vrcholy je ~ $4n/\sqrt{(\pi n)}$, obecně tedy veliký počet posloupností odpovídá každému BVS.