Rozhledy matematicko-fyzikální

Daniel Kráľ; Martin Mareš; Tomáš Valla

Recepty z programátorské kuchařky Korespondenčního semináře z programování, IV. část

Rozhledy matematicko-fyzikální, Vol. 82 (2007), No. 1, 22–35

Persistent URL: `http://dml.cz/dmlcz/146094`

Terms of use:

© Jednota českých matematiků a fyziků, 2007

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* `http://dml.cz`

INFORMATIKA

Recepty z programátorské kuchařky Korespondenčního semináře z programování, IV. část*)

Daniel Kráľ, Martin Mareš a Tomáš Valla, MFF UK Praha

Ve druhé části kuchařky jsme se zabývali tříděním dat. V dnešní části 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 máme dáno obrovské pole setříděných datových položek x_i, které mohou vypadat libovolně. To, že položky jsou setříděné, znamená jen a pouze to, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která určuje, která ze dvou položek je menší. V příkladech, které si v tomto povídání předvedeme, budou našimi daty celá čísla, ale vše, co si řekneme platí též pro jakákoliv jiná utříděná data.

Nejprve si popíšeme, jak v daném poli rychle najít nějakou konkrétní položku z. Při vyhledávání si budeme udržovat interval, ve kterém se může hledaná položka nacházet, tj., budeme mít k dispozici dva indexy l a p takové, že $x_l \leq z \leq x_p$. V každém kroku si nalistujeme položku x_m pro nějaký index m mezi l a p a porovnáme x_m a z. Pokud $x_m = z$, pak jsme nalezli hledanou položku. Pokud $x_m < z$, pak z musí následovat x_m v poli a položíme $l = m+1$. Pokud $x_m > z$, pak změníme p na $m-1$. Pokud $l = p$ a $x_l \neq z$, pak pole hledanou položku neobsahuje.

Pro libovolnou volbu m mezi l a p, se zmenší délka $p-l+1$ intervalu, kde se může hledaná položka z v poli nacházet, a algoritmus je tedy konečný. Nejčastěji se jako hodnota indexu m volí $(l+p)/2$. V takovém

*) Informace o Korespondečním semináři z programování pořádaného Matematicko-fyzikální fakultou Univerzity Karlovy v Praze lze nalézt na webové stránce `http://ksp.mff.cuni.cz`.

případě se totiž v každém kroku velikost intervalu, kde může být hledaná položka, zmenší na polovinu. Hledanou položku tedy v poli nalezneme (nebo zjistíme, že ji pole neobsahuje) v čase $O(\log N)$. Právě popsaný algoritmus se nazývá *binární vyhledávání* nebo také *hledání půlením intervalu*. Binární vyhledávání lze jednoduše naprogramovat rekurzivně nebo pomocí cyklu, jak můžeme vidět v následujícím příkladu:

```
function BinSearch(z : integer):integer;
var l,p,m : integer;
begin
  l := 1;     { meze intervalu, ve kterém hledáme }
  p := N;
  while l <= p do begin     { interval ještě není prázdný }
    m := (l+p) div 2;     { střed intervalu }
    if z < x[m] then
      p := m-1     { hleddaný prvek je menší }
    else if z > x[m] then
      l := m+1     { hledaný prvek je větší }
    else begin     { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;     { pole prvek z neobsahuje }
end;
```

Jinou možností jak volit m, používanou zejména, pokud lze předpokládat, že položky pole jsou čísla náhodně vybraná z nějakého intervalu, je volba

$$m = l + \frac{x_p - x_l}{p - l} z\,.$$

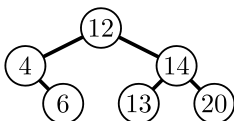
Důvodem takovéto volby je, že pokud jsou čísla v poli vybrána (rovnoměrně a nezávisle) náhodně z nějakého intervalu, pak hodnoty položek $x_1, \ldots, x_N$ jsou blízké aritmetické posloupnosti. Podrobnou analýzou, jež přesahuje rozsah tohoto článku, lze ukázat, že v takovémto případě je *očekávaný* počet kroků vyhledávání $O(\log \log N)$.

Hledání půlením intervalu je velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, jen s touto metodou si nevystačíme: 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 ve spojovém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni rychle najít prvek s indexem m. Způsobem, jak se vypořádat s těmito dvěma obtížemi, je udržovat si data ve vyhledávacím stromě.

Binární vyhledávací stromy

Zkusme provést následující myšlenkový pokus. 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 zakořeněného stromu.



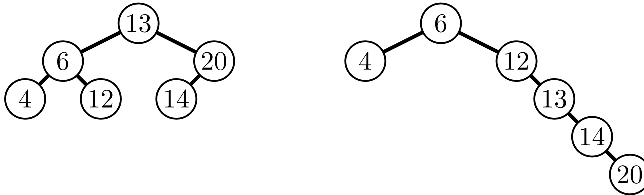
Obr. 1: Příklad binárního vyhledávacího stromu.

Kořen stromu odpovídá 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: začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále (obr. 1). Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu z.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout přesně 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 tedy popisovaly i stromy na obr. 2.

Vyhledání jednoho prvku ale v takovýchto jiných stromech už nemusí trvat jen $O(\log N)$, např. pokud je strom degenerovaný podobně jako

strom vyobrazený na obr. 2 vpravo, na vyhledání největšího prvku je potřeba lineární čas. Kromě rychlého vyhledávání ve stromech je ale též důležité, abychom takovéto stromy dokázali snadno modifikovat, tj. jednoduše přidat nebo odebrat prvek. V dnešním povídání si popíšeme AVL-stromy, které nemusí být dokonale vyvážené, ale jejich hloubka je přesto stále $O(\log N)$.



Obr. 2: Jiné vyhledávací stromy popisující stejnou množinu jako strom na obr. 1.

Zkusme si nyní pořádně nadefinovat pojmy, které budeme nadále používat. *Binární vyhledávací strom* je buď jednovrcholový strom nebo zakořeněný strom, jehož *kořen* má jeden nebo dva podstromy (*levý* a *pravý*), které jsou také binární vyhledávací stromy. Hodnoty uložené v levém podstromě jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Pokud x je prvek stromu, pak kořen levého podstromu x je *levý syn* vrcholu x a kořen pravého podstromu je jeho *pravý syn*. Vrcholu x se říká *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x odpovídajícího syna nemá. Vrchol, který nemá žádné syny, se nazývá *list*. 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.

Binární vyhledávací stromy si budeme v paměti počítače reprezentovat pomocí následující datové struktury:

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

Pokud některý ze synů neexistuje, má jemu odpovídající ukazatel hodnotu `nil`.

Vyhledávání prvku

Vyhledávání prvku v binárním vyhledávacím stromě jsme si již popsali, pro úplnost si nyní předvedeme jeho implementaci:

```
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^.p
  end;
  TreeFind := v;
end;
```

Všimněte si, že funkce `TreeFind` pracuje v čase $O(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Vkládání prvku

Nyní si popíšeme, jak do stromu vložit novou hodnotu, aniž bychom se teď starali o to, zda tím strom nemůže degenerovat. Nejprve zkusíme vkládanou hodnotu ve stromě najít a pokud ji strom neobsahuje, narazíme při hledání 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 by nová hodnota mohla být). 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);
```

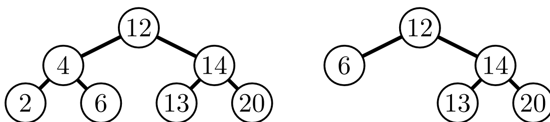
```
    v^.l := nil;
    v^.p := nil;
    v^.x := x;
  end else if x<v^.x then     { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then     { vkládáme vpravo }
    v^.p := TreeIns(v^.p, x);
  TreeIns := v;
end;
```

Odebírání prvku

Odebrání prvku ze stromu je o něco pracnější. Musíme totiž rozlišit tři případy: Pokud je odstraňovaný vrchol v list, stačí změnit ukazatel na něj na `nil`. Pokud má právě jednoho syna, vrchol v ze stromu odstraníme a jeho syna přepojíme k jeho otci. 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 odebíraného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Implementace 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^.p := TreeDel(v^.p, x)
  else begin     { našli jsme }
    if (v^.l=nil) and (v^.p=nil) then begin
      TreeDel := nil;     { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.p;     { jen pravý syn }
      dispose(v);
    end else if v^.p=nil then begin
      TreeDel := v^.l;     { jen levý }
```

```
      dispose(v);
    end else begin     { má oba syny }
      w := v^.l;     { hledáme max(L) }
      while w^.p<>nil do w := w^.p;
      v^.x := w^.x;     { prohazujeme }
    { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```



Obr. 3: Přidání prvku 2 a odebrání prvku 4 ze stromu z obr. 1.

Když do stromu z obr. 1 zkusíme nejdříve přidat prvek 10 a poté odebrat prvek 15, získáme stromy zobrazené na obr. 3. Na vkládání a odebírání prvku budeme potřebovat čas $O(h)$. Vzhledem k tomu, že jsme se prvky nesnažili přidávat a odebírat ze stromu tak, aby nedegeneroval, tak se může stát, že hloubka h stromu bude lineární v počtu jeho prvků.

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 v počtu prvků obsažených ve stromě, protože strávíme konstantní čas vypisováním každého prvku a těch je právě N. Implementace je opět přímočará:

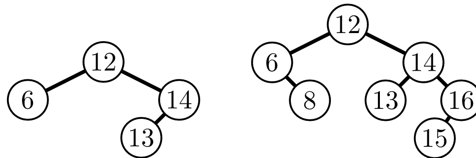
```
procedure TreeWrite(v:pvrchol);
begin
  if v=nil then exit;   { není co dělat }
  TreeWrite(v^.l);
  writeln(v^.x);
  TreeWrite(v^.p);
end;
```

Vyvážené stromy

Většina algoritmů pro binární vyhledávací stromy má časovou složitost, která je úměrná hloubce stromu. Jenže, jak jsme viděli, neopatrným vkládáním a odebíráním prvků mohou snadno vznikat stromy, jejichž hloubka je lineární v počtu prvků. Abychom tomu zabránili, musíme stromy *vyvažovat*. Definujeme si tedy vhodné omezení na tvar stromu tak, aby jeho hloubka byla vždy nejvýše $O(\log N)$.

Dokonale vyvážený vyhledávací strom je takový, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedna. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto 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 binární vyhledávací strom zkonstruovat v lineárním čase ze setříděného pole. Bohužel však vkládaní a odebírání prvků tak, aby výsledný strom byl opět dokonale vyvážený, nelze provést v logaritmickém čase.

Zkusíme tedy méně přísnou vyvažovací podmínku a budeme vyžadovat pouze, aby se u každého vrcholu lišily o jedna nikoliv velikosti podstromů, nýbrž jejich hloubky. Takovým stromům se říká *AVL stromy*. Příklady AVL stromů lze nalézt na obr. 4.



Obr. 4: Dva příklady AVL stromů.

Každý dokonale vyvážený strom je také AVL stromem, ale opačně to platit nemusí, viz obr. 4. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a toto tvrzení si zaslouží samostatný důkaz.

Tvrzení: AVL strom, který obsahuje N prvků, má hloubku $O(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může mít AVL strom hloubky d. Snadno zjistíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příklady takových stromů hloubky 3 a 4 lze nalézt na obr. 4). 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 jinak by hloubka celého stromu nebyla d, a druhý hloubku $d-2$, neboť podle definice AVL stromu může mít hloubku $d-1$ nebo $d-2$ (a s menší hloubkou může mít evidentně méně vrcholů).

Spočítat, kolik přesně je A_d, není úplně snadné, ale nám bude stačit dokázat, že $A_d \geq 2^{n/2}$. To provedeme indukcí: Pro $d < 4$ tvrzení platí. Pro $d \geq 4$ je pak

$$A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2}(2^{-1/2} + 2^{-1}) > 2^{d/2}$$

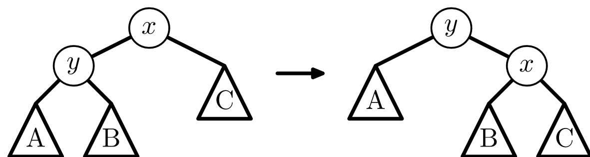
(součet čísel v závorce je ≈ 1.207).

Nyní už víme, že A_d roste s d alespoň exponenciálně. 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 $d \leq \log_{\sqrt{2}} N$, čili $d = O(\log N)$.

Rotace a dvojrotace

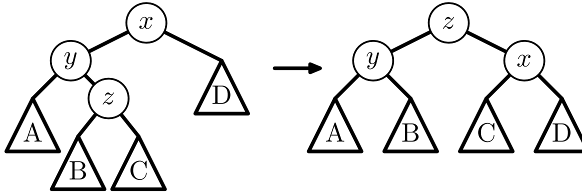
Po vložení či odebrání prvku si nemůžeme dovolit strukturu vyhledávacího stromu libovolně změnit, neboť musíme zachovat jak vyvážení stromu, tak i jeho uspořádání. Při práci se stromem se nám budou hodit dvě operace, které se nazývají rotace a dvojrotace.

Rotace binárního stromu (respektive nějakého podstromu) je jeho „překořenění“ za některého ze synů kořene. Místo formální definice odkážeme raději čtenáře na obr. 5. Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět správně uspořádané (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.



Obr. 5: Rotace v binárním vyhledávacím stromě.

Dvojrotace je posloupnost dvou rotací, které se liší svým směrem (tj. jedna rotace je doprava a druhá doleva). Výsledkem dvojrotace je překořenění podstromu za vnuka kořene a její příklad lze nalézt na obr. 6.



Obr. 6: Dvojrotace v binárním vyhledávacím stromě.

Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. To si budeme pamatovat pomocí *znaménka* vrcholu: znaménko vrcholu je $\odot$, jsou-li oba podstromy stejně hluboké, $\ominus$ pokud je levý podstrom hlubší, a $\oplus$ pokud je pravý podstrom hlubší.

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 jen jednu s tím, že druhá se v algoritmu zpracuje symetricky.

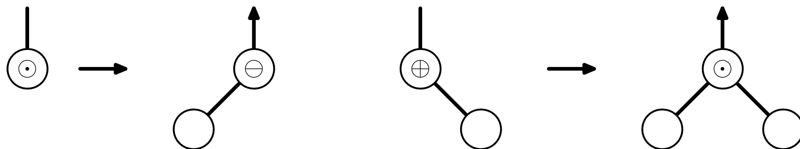
Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat. Anebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, celou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

AVL stromy – vyvažování po vložení prvku

Po vložení prvku do stromu vytvoříme nový list stromu. Pokud se tím vyváženost stromu neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže se porušila, musíme vyváženost stromu obnovit. Popíšeme si algoritmus, který bude postupovat od nového listu ke kořeni a vše potřebné zařídí.

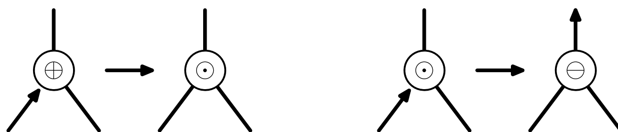
Samotné přidání listu do stromu je na obr. 7. Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci, že se hloubka podstromu zvýšila (to budeme značit šipkou). Pokud jsme

přidali list k vrcholu se znaménkem ⊕, změní se jeho znaménko na ⊙ a hloubka podstromu se nezmění. V takovém případě je strom vyvážený a můžeme skončit.

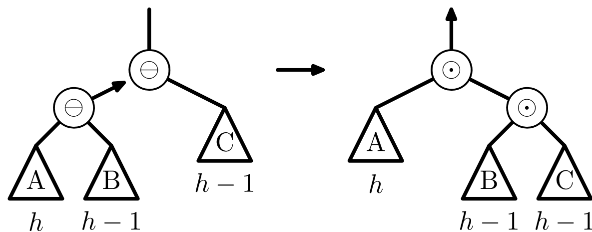


Obr. 7: Přidání listu do AVL stromu.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách. Vždy budeme (podle symetrie) předpokládat, že se hloubka levého podstromu zvětšila o jedna. Pokud měl vrchol znaménko ⊕ nebo ⊙, tak stačí změnit jeho znaménko a případně poslat o patro výše informaci, že se hloubka podstromu zvýšila o jedna (obr. 8).



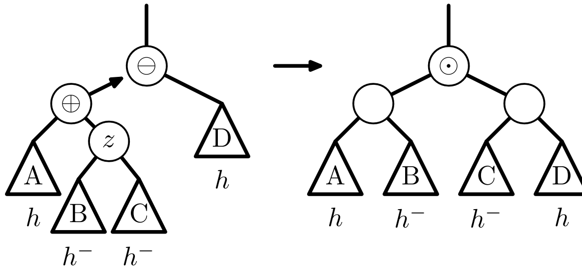
Obr. 8: Vyvažování po přidání u vrcholu se znaménkem ⊕ nebo ⊙.



Obr. 9: Vyvažování po přidání u vrcholu se znaménkem ⊖, jehož levý syn má též znaménko ⊖. Podstromy mají pod sebou uvedenu svoji hloubku.

Pokud ale má vrchol x znaménko ⊖, je situace složitější: levý podstrom má teď hloubku o dvě vyšší než pravý, takže musíme změnit strukturu stromu. Proto se podíváme o patro níž, jaké je znaménko levého syna vrcholu x (označme tohoto syna y). Pokud je znaménko y rovno ⊖, provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami lze vidět na obr. 9 – pokud si hloubku podstromu A označíme h, B musí mít

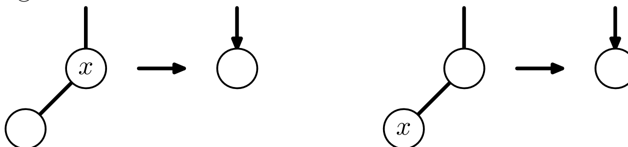
hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali, takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsou na obrázku v závorkách). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.



Obr. 10: Vyvažování po přidání u vrcholu se znaménkem $\ominus$, jehož levý syn má znaménko $\oplus$. Podstromy mají pod sebou uvedenu svoji hloubku; h^- značí hloubku $h-1$ nebo h.

Pokud y má znaménko $\oplus$, podíváme se ještě o hladinu níž a provedeme dvojrotaci (obr. 10). Všimněte si, že se nám nemůže stát, že by z neexistovalo, protože jinak by znaménko y nebylo $\oplus$. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^- (a podle těchto hloubek pak určíme znaménka vrcholů x a y po rotaci). Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže nemusíme ve vyvažování stromu dále pokračovat.

Poslední možnost, že by znaménko y bylo $\odot$, nemůže nastat. Kdykoliv se totiž změní hloubka nějakého podstromu, má jeho kořen znaménko různé od $\odot$.



Obr. 11: Odebrání vrcholu x s žádným nebo jedním synem.

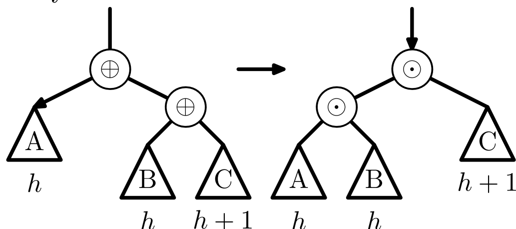
AVL stromy – vyvažování po odebrání prvku

Vyvažování po odebrání prvku je poněkud obtížnější, ale lze jej též snadno popsat pomocí několika obrázků. Při odebírání prvku odstraňujeme ze stromu vždy vrchol x, který je list nebo má jednoho syna.

Ze symetrie můžeme předpokládat, že pokud x je list, je levým synem svého otce, a pokud má jednoho syna, má levého syna. V obou případech vrchol x ze stromu odstraníme a pošleme informaci, že se hloubka podstromu zmenšila o jedna (obr. 11). Toto naznačíme šipkou dolů.

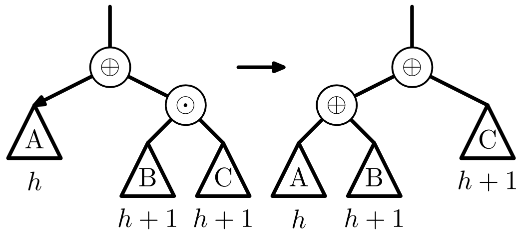
Obr. 12: Vyvažování po zmenšení hloubky levého podstromu vrcholu se znaménkem $\ominus$ nebo $\odot$.

Pokud se zmenšila hloubka levého podstromu vrcholu, který má znaménko $\ominus$ nebo $\odot$, stačí změnit znaménko tohoto vrcholu, jak je znázorněno na obr. 12. Obtížnější jsou případy, kdy se zmenší hloubka levého podstromu vrcholu se znaménkem $\oplus$. Tehdy se musíme podívat na znaménko jeho *pravého* syna a podle toho zvolit vhodnou (dvoj)rotaci. Pokud má pravý syn znaménko $\oplus$ (obr. 13), provedeme rotaci vlevo. Znaménka obou rotovaných vrcholů se změní na $\odot$ a celková hloubka stromu se sníží o jednu hladinu. Musíme tedy ve vyvažování pokračovat o jednu hladinu výše.

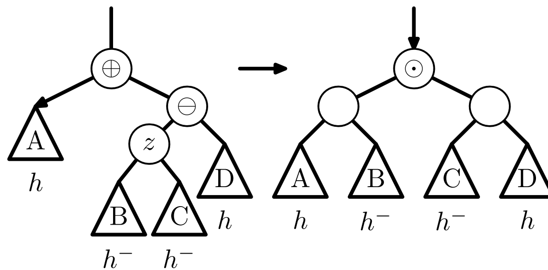


Obr. 13: Vyvažování po zmenšení hloubky levého podstromu vrcholu se znaménkem $\oplus$, jehož pravý syn má též znaménko $\oplus$. Podstromy mají pod sebou uvedenu svoji hloubku.

Pokud je znaménko pravého syna $\odot$, provedeme též rotaci vlevo (obr. 14), ale protože se tentokráte hloubka podstromu nezměnila, skončíme. Nejkomplikovanější případ je, když je znaménko pravého syna $\ominus$ (obr. 15). V tomto případě provedeme dvojrotaci (vrchol z existuje, protože jeho otec má znaménko $\ominus$). Vrcholy obdrží znaménka v závislosti na původním znaménku vrcholu z a protože se hloubka podstromu snížila o jedna, musíme ve vyvažování pokračovat ve stromě o patro výš.



Obr. 14: Vyvažování po zmenšení hloubky levého podstromu vrcholu se znaménkem ⊕, jehož pravý syn má znaménko ⊙. Podstromy mají pod sebou uvedenu svoji hloubku.



Obr. 15: Vyvažování po zmenšení hloubky levého podstromu vrcholu se znaménkem ⊕, jehož pravý syn má znaménko ⊖. Podstromy mají pod sebou uvedenu svoji hloubku; h^- značí hloubku $h-1$ nebo h.

AVL stromy – shrnutí

Předvedli jsme si, jak do AVL stromu vložit a jak z něj odebrat prvek tak, aby výsledný strom byl opět AVL stromem. V obou případech jsme postupovali při vyvažování od listu ke kořeni, a proto je časová složitost vložení či odebrání jednoho prvku lineární v hloubce stromu, a tedy logaritmická v počtu jeho prvků.

AVL stromy nejsou jedinými vyhledávacími stromy, u kterých umíme udržet logaritmickou hloubku. V některém z následujících dílů kuchařky si předvedeme *červeno-černé stromy*, kde si u každého vrcholu pamatujeme jeho barvu (červenou nebo černou) a platí, že počet černých vrcholů na cestě z kořene do libovolného listu je stejný. Jiným typem vyhledávacích stromů jsou *2-3-stromy*, kde každý vrchol má dva nebo tři syny. Další typy vyhledávacích struktur jsou např. *splay stromy*, *treapy*, *BB-α stromy* – jejich detailní popis však přesahuje rozsah našeho povídání.