KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1◇ poznámky k přednášce

7. Stromy

verze z 17. listopadu 2019

1 Speciální operátor `quote`

Operátor přijímá jeden argument, jako výsledek tento argument vrací, aniž by ho vyhodnotil:

```
CL-USER 1 > (quote (1 2 3))
(1 2 3)

CL-USER 2 > (quote a)
A
```

Zkratka pro totéž je

```
CL-USER 3 > '(1 2 3)
(1 2 3)

CL-USER 4 > 'a
A
```

Měli bychom si ale stále pamatovat, že je to jen zkratka pro seznam se symbolem `quote` jako operátorem.

Symboly jsou tedy hodnoty. Příklady:

```
CL-USER 5 > (setf a 'b)
B

CL-USER 6 > a
B

CL-USER 7 > (setf b 'a)
A

CL-USER 8 > (list a b)
(B A)
```

```
CL-USER 9 > (setf c 'c)
C

CL-USER 10 > (eql c 'c)
T

CL-USER 11 > (eql 'c 'c)
T
```

2 Stromy

Datová struktura *strom* se skládá

1. z konečné množiny, které říkáme *množina uzlů* a jejíž prvky nazýváme *uzly*,
2. z přiřazení, které každému uzlu přiřazuje množinu jeho *podstromů*, což jsou podmnožiny množiny uzlů.

Strom musí splňovat následující podmínky:

1. existuje právě jeden uzel, tzv. *kořen*, který neleží v žádném podstromu,
2. libovolné dva podstromy jednoho uzlu mají prázdný průnik (jsou tzv. disjunktní),
3. každý podstrom je opět strom, když pro něj uvažujeme stejné přiřazování podstromů, jako u celého stromu.

Uzly stromu mají obvykle přiřazenu jednu nebo víc hodnot.

Základní pojmy vztahující se ke stromům

Následník uzlu u je libovolný uzel v, který je kořenem podstromu uzlu u. Uzel u je pak *předchůdce* uzlu v Každý uzel kromě kořene má právě jednoho předchůdce, kořen nemá žádného.

Uzel, který má prázdnou množinu podstromů (a tedy prázdnou množinu následníků) se nazývá *list*.

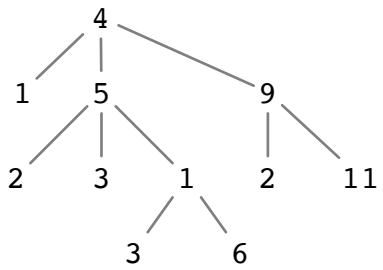
Cesta k uzlu u je posloupnost (seznam) uzlů s kořenem na prvním a uzlem u na posledním místě. Každý následující prvek v cestě je následníkem předchozího.

Maximální cesta je cesta k listu.

Délka cesty je počet uzlů v cestě minus 1.

Výška stromu je maximální délka cesty ve stromu.

Příklad. Stromy můžeme znázorňovat grafy jako například na tomto obrázku:



V tomto stromu je uzel s hodnotou 3 následníkem uzlu s hodnotou 5. Kořenem je uzel s hodnotou 4. Cesta k uzlu s hodnotou 6 obsahuje po řadě uzly s hodnotami 4, 5, 1, 6. Je to maximální cesta, protože uzel s hodnotou 6 je list.

Reprezentace stromů pomocí seznamů

Uzel stromu můžeme v Lispu reprezentovat takto:

```
(hodnota . seznam-následníků)
```

Reprezentaci uzlu můžeme chápat i jako reprezentaci celého (pod)stromu s tímto uzlem jako kořenem.

Příklad. Strom z předchozího příkladu můžeme tedy reprezentovat seznamem

```
(4 (1) (5 (2) (3) (1 (3) (6))) (9 (2) (11)))
```

Konstruktor a selektory pro novou datovou strukturu:

```
(defun tree-node (val children)
  (cons val children))

(defun node-value (node)
  (car node))

(defun node-children (node)
  (cdr node))
```

Za chvíli se nám bude hodit zobecnit selektory na seznam uzlů. Napíšeme si tedy funkci, které k seznamu uzlů vrátí seznam jejich hodnot, a funkci, která k seznamu uzlů vrátí seznam seznamů jejich následníků. V realitě bychom tyto funkce napsali až v momentě, kdy bychom zjistili, že je potřebujeme.

```
(defun node-value-multi (nodes)
  (if (null nodes)
      '()
    (cons (node-value (car nodes))
          (node-value-multi (cdr nodes)))))

(defun node-children-multi (nodes)
  (if (null nodes)
      '()
    (append (node-children (car nodes))
            (node-children-multi (cdr nodes)))))
```

Všimněte si podobnosti obou funkcí. V budoucnu ukážeme, jak lze takové podobné funkce sloučit do jedné.

Následující funkce `tree-values-dfs` projde celý strom a vrátí seznam jeho hodnot. V zadání neurčujeme, v jakém pořadí mají hodnoty v seznamu být. To bude dáno tím, jak je funkce naprogramována: pro každý uzel rekurzivně zjistí seznam hodnot jeho podstromů a přidá k němu hodnotu uzlu.

```
(defun tree-values-dfs (root)
  (cons (node-value root)
        (tree-values-dfs-multi (node-children root))))

(defun tree-values-dfs-multi (roots)
  (if (null roots)
      '()
    (append (tree-values-dfs (car roots))
            (tree-values-dfs-multi (cdr roots)))))
```

Funkci můžeme otestovat na našem stromu:

```
CL-USER 1 > (tree-values-dfs '(4 (1) (5 (2) (3) (1 (3) (6))) (9 (2)
(11))))
(4 1 5 2 3 1 3 6 9 2 11)
```

Na příkladu vidíme, proč se tomuto způsobu prohledávání stromu říká *prohledávání do hloubky* (*depth-first search*).

Nyní napíšeme funkci, která vrátí seznam všech hodnot ve stromu *prohledáním do šířky* (*breadth-first search*). Podstatnou je tady pomocná funkce `tree-values-bfs-multi`, která k dané „vrstvě“ ve stromu (pojem vrstvy se dá přesně definovat) zadané seznamem uzlů zjistí nejprve seznam hodnot uzlů a k němu připojí seznam dalších hodnot zjištěný rekurzivním voláním pro seznam všech následníků uzlů vrstvy.

Test, opět na našem stromu:

```
CL-USER 2 > (tree-values-bfs '(4 (1) (5 (2) (3) (1 (3) (6))) (9 (2)
(11))))
(4 1 5 9 2 3 1 2 11 3 6)
```

```
(defun tree-values-bfs (root)
  (tree-values-bfs-multi (list root)))

(defun tree-values-bfs-multi (roots)
  (if (null roots)
      '()
    (append (node-value-multi roots)
            (tree-values-bfs-multi (node-children-multi roots)))))
```

3 Binární stromy

Binární vyhledávací strom

Uvažujeme stromy s číselnými hodnotami uzlů. Binární vyhledávací stromy slouží k rychlému nalezení uzlu podle hodnoty.

Binární strom

Strom se nazývá *binární*, pokud má každý jeho uzel nejvýše dva následníky. Ty jsou označeny jako *levý* a *pravý*. Levý následník uzlu určuje jeho *levý podstrom*, pravý následník určuje *pravý podstrom*.

Uzel nemusí mít žádného následníka, nebo jen levého nebo pravého, nebo oba.

(Zjednodušený) binární vyhledávací strom

Pro každý uzel platí, že hodnoty všech uzlů v jeho levém podstromu jsou menší než hodnota uzlu, hodnoty všech uzlů v jeho pravém podstromu jsou větší.

Implementace

Změna: abychom jednoznačně určili, který následník je levý a který pravý, změníme reprezentaci uzlu na

```
(hodnota levý-následník pravý-následník)
```

Pokud následník neexistuje, bude na jeho pozici `nil` (dá se chápat jako *prázdný strom*).

Implementace

Konstruktor:

```
(defun binary-tree-node (val left-child right-child)
  (list val left-child right-child))
```

Nové selektory:

```
(defun left-child (node)
  (cadr node))
(defun right-child (node)
  (caddr node))
```

Úprava funkce `node-children`:

```
(defun bt-node-children (node)
  (remove nil (cdr node)))
```

4 Využití symbolů jako identifikátorů typu

Vidíme, že funkci `node-children` jsme museli napsat jinak, než byla napsaná pro obecné stromy. Proto jsme ji také museli přejmenovat. To nám bude působit potíže u jiných funkcí: například k funkci `tree-values-dfs` bychom také museli zbytečně dělat novou verzi pro binární stromy, protože funkci `node-children` používá. Totéž se týká funkce `tree-values-dfs-multi`.

Nové verze by vypadaly takto:

```
(defun tree-values-dfs-bt (root)
  (cons (node-value root)
        (tree-values-dfs-bt-multi (node-children-bt root))))

(defun tree-values-dfs-bt-multi (roots)
  (if (null roots)
      '()
    (append (tree-values-dfs-bt (car roots))
            (tree-values-dfs-bt-multi (cdr roots)))))
```

(Porovnejte si je s původními funkcemi `tree-values-bfs` a `tree-values-bfs-multi`.

Takové řešení je ovšem nevhodné, protože není správné programovat víc funkcí na v podstatě tutéž práci.

Problém vyřešíme tak, že obohatíme reprezentaci uzlů stromu o informaci, zda jde o obecný uzel, nebo o uzel binárního stromu. Informace bude dána symbolem na prvním místě seznamu.

Nová reprezentace obecného uzlu

Symbol `tree` na prvním místě určuje, že jde o uzel obecného stromu.

```
(tree hodnota . seznam-následníků)
```

Konstruktor:

```
(defun tree-node (val children)
  (cons 'tree (cons val children)))
```

Selektor `node-value`:

```
(defun node-value (node)
  (cadr node))
```

Nová reprezentace uzlu binárního stromu

Symbol `binary-tree` na prvním místě určuje, že jde o uzel binárního stromu.

```
(binary-tree hodnota levý-následník pravý-následník)
```

Konstruktor:

```
(defun binary-tree-node (val left-child right-child)
  (list 'binary-tree val left-child right-child))
```

Selektory `left-child` a `right-child`:

```
(defun left-child (node)
  (caddr node))

(defun right-child (node)
  (cadddr node))
```

Detekce typu stromu

```
(defun tree-type (node)
  (car node))

(defun treep (node)
  (eql (tree-type tree) 'tree))

(defun binary-tree-p (node)
  (eql (tree-type tree) 'binary-tree))
```

Funkce `node-children`

Funkce je tzv. *polymorfní*, umí pracovat s hodnotami více typů současně.

```
(defun node-children (node)
  (cond ((treep node) (cddr node))
        ((binary-tree-p node) (remove nil (cddr node)))
        (t (error "Unknown tree type."))))
```

Všechny funkce, které jsme napsali pro obecné stromy, nyní budou fungovat na oba typy stromů.

Poznámka. Použitá funkce `error` způsobí zastavení programu a vyvolání chyby:

```
CL-USER 3 > (error "Unknown tree type.")

Error: Unknown tree type.
  1 (abort) Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for
other options.

CL-USER 4 : 1 >
```

V tomto případě jsme ji použili, protože jiný typ stromu než naše dva není definován a pokud se program dostane do třetí větve makra `cond`, znamená to chybu a program by neměl pokračovat v práci.

Jako argument funkce `error` jsme použili tzv. *textový řetězec*. Co to je, si vysvětlíme příště.

5 Reprezentace číselných množin binárními vyhledávacími stromy

Pokud je množina čísel reprezentována binárním vyhledávacím stromem s minimální možnou výškou (tzv. vyvážený binární vyhledávací strom), je hledání prvků v množině mnohem rychlejší než u reprezentace seznamem, kterou jsme probírali na minulé přednášce (místo lineární složitosti jde o složitost logaritmickou).

Následující dvě funkce implementují dvě základní akce pro množiny reprezentované binárním vyhledávacím stromem: testování, zda prvek leží v množině, a přidání prvku do množiny (přesněji řečeno vytvoření množiny, která obsahuje všechny prvky zadané množiny a jeden zadaný prvek navíc).

Nová verze funkce `elementp` zjišťuje, zda dané číslo leží v množině zadané binárním vyhledávacím stromem. Jako strom (parametr `tree`) přijímá i symbol `nil` (prázdný strom).

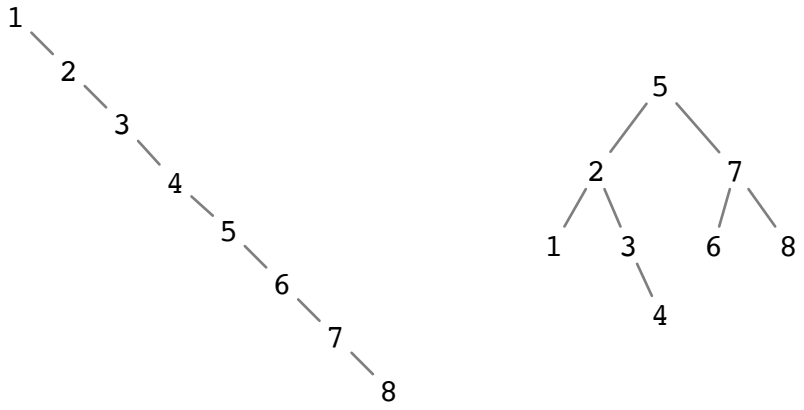
```
(defun elementp (el tree)
  (if (null tree)
      nil
    (let ((val (node-value tree)))
      (or (= el (node-value tree))
          (and (< el val) (elementp el (left-child tree)))
          (and (> el val) (elementp el (right-child tree)))))))
```

Výpočet bude tím rychlejší, čím bude mít strom menší výšku (počet rekurzivních volání je roven nejvýše délce cesty k nějakému listu).

Přidání prvku k množině reprezentované binárním vyhledávacím stromem realizuje funkce `my-adjoin`. Pokud prvek v množině už je, prvek podruhé nepřidává. Funkce se stará o to, aby nový prvek do stromu správně zařadila, tj. aby nový strom byl stále binární vyhledávací.

```
(defun my-adjoin (elem tree)
  (if (null tree)
      (binary-tree-node elem nil nil)
    (let ((val (tree-node-value tree))
          (left (left-child tree))
          (right (right-child tree)))
      (cond ((= elem val) tree)
            ((< elem val) (binary-tree-node val
                                            (my-adjoin elem left)
                                            right))
            (t (binary-tree-node val
                                 left
                                 (my-adjoin elem right)))))))
```

Jak už bylo řečeno, *vyvážený binární vyhledávací strom* (*balanced binary search tree*) je binární vyhledávací strom s minimální výškou. Platí, že výšky levého a pravého podstromu každého uzlu vyváženého binárního vyhledávacího stromu se liší nejvýše o 1.



nevyvážený **vyvážený**

Vyvážených binárních vyhledávacích stromů se týkají některé úkoly k této přednášce.

Otázky a úkoly na cvičení

1. Napište výraz, který vytvoří reprezentaci vyváženého binárního stromu z přednášky. Uzly vytvářejte výhradně pomocí konstruktoru `binary-tree-node`.
2. Vytvořte stejný binární strom z prázdného stromu opakovaným použitím funkce `my-adjoin`. Zkuste změnou pořadí přidávaných uzlů vytvořit ze stejných hodnot jiný strom.
3. Napište funkci, která k danému seznamu čísel vytvoří binární vyhledávací strom.
4. Napište predikát `balancedp`, který zjistí, zda je daný binární vyhledávací strom vyvážený.
5. Napište funkci `bt-swap`, která k danému binárnímu stromu vrátí strom s vyměněným levým a pravým následníkem všech uzlů.
6. Napište funkci, která k danému číslu vrátí jeho uzel v binárním vyhledávacím stromu.
7. Napište funkci `tree-sum`, která k danému stromu s číselnými hodnotami vrátí jejich součin.

8. Je možné napsat funkce na průnik a sjednocení množin čísel reprezentovaných binárními vyhledávacími stromy stejně, jako příslušné funkce pro množiny reprezentované seznamy z minulé přednášky?

9. Napište funkci `tree-maximal-paths`, která k danému stromu vrátí seznam všech cest k listům.

10. Napište funkci `tree-height`, která vrátí výšku daného stromu, aniž by zjišťovala cesty (tedy bez pomoci funkce `tree-maximal-paths`).

11. Strom je zadán seznamem hodnot a symbolů `/` a `//`, kde `/` znamená přechod k dalšímu předchůdci a `//` přechod na další vrstvu. První strom z přednášky lze tedy zadat seznamem `(4 // 1 5 9 // / 2 3 1 / 2 11 // / / 3 6)`. Napište funkci, která z takového seznamu udělá strom.

12. Upravte funkci `my-adjoin`, aby zachovala vyváženost stromu, ke kterému přidává nový prvek.