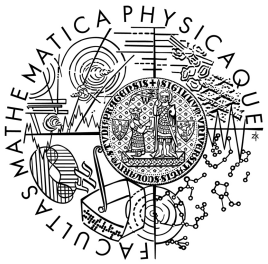


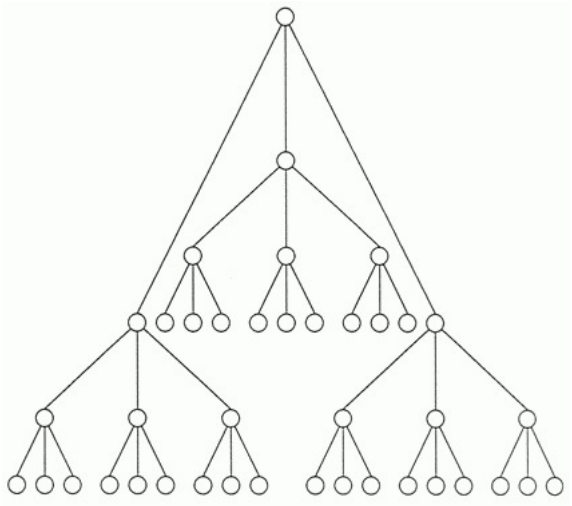
Matematicko-fyzikální fakulta
Univerzity Karlovy v Praze

Zápočtový program k NPRG030
ZS 2010/2011

Knihovna twhree

Autor:
Duc Trung Ha

Cvičící:
Bc. Jan Kohout



22. dubna 2011

Abstrakt

Dokumentace k zápočtovému programu. Program je implementací datové struktury binárního vyhledávacího 2-3 stromu a základních operací na něm.

Obsah

Část I
Úvodní slovo

Informační doba si žádá v čím dál větší míře lepší a lepší organizaci a zpracování dat. Gigantické databáze s astronomickým množstvím dat již dnes nejsou žádnou zvláštností. Vyvstává přirozeně otázka, po jakých datových strukturách pro tento účel sáhnout.

První, co většině lidí přijde na mysl, je využít pole či lineární spojové seznamy, jež jsou jednodušše implementovatelné. Tento přístup je ovšem poněkud naivní, neboť (ač vkládání lze zvládat v konstantním čase) operace vyhledávání obnáší časovou nářočnost $\mathcal{O}(n)$. Na tom není vůbec nic ošklivého, algoritmy polynomiální časové třídy jsou obecně považovány za efektivní, ovšem pro účely databázových systémů, jež mají často velikosti v řádech miliard, to není zrovna nejvhodnější návrh.

Na úkor vkládání lze však tento nesnadný úkol vyhledávání „jehly v kupce sena“ zvládnout pomocí sofistikovaných datových struktur v krásném čase $\mathcal{O}(\log n)$.

Jednou z takovýchto struktur je právě 2-3 strom, jehož standardní operace (Insert, Delete, Find, Find Max, Find Min, Previous, Next) právě knihovna *twhree* (jejíž název vznikl ze slov **T**wo-t**H**ree t**REE**) implementuje.

Část II
Programátorský manuál

1 Anotace

Implementace operací s vyváženými vyhledávacími stromy (např. AVL, červeno-černými nebo 2-3): insert, delete, find, findmin, findmax, next.

2 Přesné zadání

Implementujte datovou strukturu 2-3 stromu a jeho operací. Strom bude obsahovat uzly dvojího typu:

- ♠ **Vnitřní uzel** obsahující informace o svých podstromech a jejich příslušných maximálních klíčů
- ♠ **List** obsahující ukazatel na samotná data

Dále bude se stromem možné provádět následující operace:

- ♣ **Insert** vkládající nový uzel do stromu s příslušným ukazatelem na nově vytvořenou datovou položku
- ♣ **Delete** mazající uzel se zadaným klíčem ze stromu spolu s jeho příslušnou datovou položku
- ♣ **Find** vyhledávající uzel se zadaným klíčem ve stromu a vracející ukazatel na danou datovou položku
- ♣ **Findmin** vyhledávající uzel s nejnižším klíčem ve stromu a vracející ukazatel na danou datovou položku
- ♣ **Findmax** vyhledávající uzel s nejvyšším klíčem ve stromu a vracející ukazatel na danou datovou položku
- ♣ **Previous**[1] vyhledávající uzel s nejbližší nižší hodnotou klíče ve stromu a vracející ukazatel na danou datovou položku
- ♣ **Next** vyhledávající uzel s nejbližší vyšší hodnotou klíče ve stromu a vracející ukazatel na danou datovou položku

[1] Lze si povšimnout, že oproti původnímu plánu byla přidána operace `Previous`, jež vyvstala jako logický důsledek implementace operace `Next`.

3 Algoritmy a datové struktury

3.1 Použité datové struktury

V této sekci si přiblížíme použité datové části v programu.

Hlavní struktura	**Datový typ položky**	**Název položky**	**Popis**
TItem	int	key	*klíč dat*
	char [10]	name	*samotná data - zde text*
subtree	struct node *	sub	*ukazatel na (pod)strom*
	int	max_key	*...a příslušný klíč maximální velikosti*
node	struct node *	parent	*ukazatel na otce/rodiče uzlu*
	subtree [3]	kidz	*pole ukazatelů na potomky uzlu a jejich maximálních klíčů*
	TItem *	pData	*ukazatel na samotná data (pouze v uzlech)*

Zřejmě nejzajímavější je položka `kidz`, která reprezentuje 2 až 3 potomky uzlu. Tato je řešena skrze pole z příčin ryze praktických ⇐ snadnější přerovnávání potomků pouhým jednoduchým zabublaním nového přidaného prvku z pravého konce pole na vhodné místo (přitom potomek s hodnotou `NULL` položky `sub` se vždy odsouvá napravo).

3.2 Použité algoritmy

V této sekci si popíšeme algoritmy použité na implementaci jednotlivých operací na 2-3 stromech. Půjde spíše o „nošení dříví do lesa“, autoři 2-3 stromu totiž tyto algoritmy navrhli s co možná největší efektivitou a proto není třeba vymýšlet něco nového či převratného.

3.2.1 Napojení knihovny na program

Knihovna `twhree` je psána v jazyku C a její integrování do programu je velice jednoduché. Soubory knihovny `twhree.h` a `twhree.c` stačí zkopírovat do složky s programem využívajích této knihovny a do programu vložit následující řádek:

```
#include "twhree.h"
```

To nalinkuje hlavičky funkcí a datové struktury knihovny. Dál je nutné při kompilaci zkompilovat soubory knihovny a připojit je ke kompilovanému programu.[2]

[2] Jako příklad tohoto může posloužit přiložený `Makefile`

3.2.2 Funkce insert

Funkce má následující hlavičku:

```
struct node*
 insert(struct TItem *pNewItem, struct node **pRoot);
```

Povšimněme si, že tato funkce přijímá jako argument ukazatel na ukazetel, tedy `struct node **`. To je z toho důvodu, že potřebujeme měnit přímo ukazatel na kořen stromu (bude-li na konci přidání jiný), a nestačí nám tedy měnit jen pouhá data samotná, páč ty už nemusí být kořenem.

Dále funkce `insert` vrací ukazatel na nově přidaný uzel, to je čistě pro účely obslužného programu, aby dokázal vypsat informace o novém uzlu. V případě potřeby lze přetypovat funkci na typ `void` a tím tuto drobnost vyřešit.

Jak ale samotný algoritmus pracuje?

Nejdřív jsou řešeny obskurní a málo časté speciální případy:

- Strom je **prázdný**. Pak se prostě vytvoří nový uzel s ukazatelem na daná data a označí se jako jediný uzel stromu (tj. jako kořen).
- Strom má **právě 1 uzel**. Pak se vytvoří nový uzel s ukazatelem na daná data. Dále se alokuje ještě společný rodič tohoto nového uzlu původního jediného uzlu, jenž bude sloužit jako nový kořen stromu se potřebnými 2 potomky.
- Jinak vznikají zajímavější případy. Nejprve se nalezne vhodný interval klíčů, do kterého nová data podle svého klíče zapadnou. Hledáme vlastně vhodného rodiče nějakých listů, jejichž hodnoty klíčů již určují, kam nový uzel zapadne.[3] Nyní se rozhoduje na základě toho, zda-li je pro nový uzel „místo“ v rodiči dle pravidel 2-3 stromu.
  - Rodič má **právě 2 potomky**. Pak se jednoduše na konec pole potomků přidá tento nový uzel a ten se posléze díky funkci `BubleKidz` zarovná (konkrétně „zabublá“) na správné místo.
  - Rodič má **právě 3 potomky**. Nejprve se stejnak vytvoří daný nový uzel. Pak se rozdělí dotyčný rodič na 2 uzly[4] a mezi ně se rozdělí tito 4 potomci v poměru 2 a 2, přičemž se zachová vzestupné pořadí klíčů.[5]

[3] Probíhá velmi podobně jako u funkce `find` - vyhledá se dle klíčů a rozhoduje se dle toho, kterou větví potomků se vydat

[4] Konkrétně se vytvoří nový pravý sourozenec tohoto rodiče

[5] Ty se seřadí v pomocném poli `tmp_kidz`

Pokud rodič těchto 2 rozdělených uzlů[6] má nyní nanejvýš 3 potomky je vše v pořádku a s `insert` může vrátit požadovaný ukazatel na uzel s nově vkládanými daty (viz výše).

Pokud rodič těchto 2 rozdělených uzlů má nyní 4, postupuje se pro něj analogicky rekursivně.[7]

Je ještě důležité nezapomenout aktualizovat hodnoty maximálních klíčů u předků nově vkládaných uzlů. Pokud se totiž uzlu přidá nový (úplně „nejpravější“) potomek, změní se tím samozřejmě i hodnota největšího klíče podstromu s kořenem v tomto uzlu.

3.2.3 Funkce delete

Funkce má následující hlavičku:

```
int delete(struct node **pRoot, int key);
```

V atributu `key` přijímá klíč k vyhledání mazaného listu.

Pokud list s klíčem není nalezen, funkce vrací hodnotu 1 či -1 (v případě prázdného stromu).

Pokud list s klíčem nalezen je, funkce vrací hodnotu 0. Algoritmus přitom maže list následujícím způsobem:

- Je-li uzel jediný ve stromu, tj. jedná se o samostatný **kořen**, jednoduše se dealokuje společně s daty, na něž ukazuje.
- Má-li rodič uzlu **3 potomky**, jednoduše se uzel smaže, v rodiči se patřičně setřídí vzestupně klíče a upraví se hodnoty maximálních klíčů pro další předky.[8]
- Má-li rodič uzlu **2 potomky**, rozlišují se následující 3 případy:
  - Je-li rodič *kořenem stromu*, uzel se smaže a za nový kořen stromu se určí jeho sourozenec.
  - Má-li rodič nějakého *sourozence se 3 potomky*, jednoho si „ukradne“ pro vyrovnání počtu vlastních dětí.[9]
  - Má-li rodič jen *sourozence se 2 potomky*, jednomu z nich přenechá svého jediného zbývajícího syna, čímž ze sourozence udělá uzel se 3 potomky. Ovšem nyní se musí eliminovat uzel s rodičem smazaného uzlu. Není nic jednoduššího-prostě se i na něj rekursivně

[6] tj. rodič nalezeného rodiče

[7] tj. vytvoří se jeho pravý sourozenec a jeho 4 potomci se mezi tyto 2 rodiče rozdělí po dvou

[8] V případě, že jsme smazali něčí maximální klíč.

[9] Tedy zabitý uzel je nahrazen tak, že jeho rodič místo něj adoptuje jeho bratrance z početnější rodinky `:-)`

zavolá funkce pro smazání uzlu, a pokud bychom takto rekursí náhodou došlu až ke kořeni, kořenový uzel se prostě smaže a kořen se přenastaví, jak je to popsáno o 2 body výše.

3.2.4 Funkce find

Tato funkce má následující podobu:

```
struct TItem * find(struct node *root, int sKey);
```

Rozhoduje se podle těchto 3 situací:

- ∇ V případě **prázdného stromu** vrací ukazatel `NULL`.
- ∇ V případě **listu** obsahující hledaný klíč vrací ukazatel na svá data.
- ∇ V případě **listu** NEobsahující hledaný klíč vrací ukazatel `NULL` (tj. položka s daným klíčem nebyla nalezena).
- ∇ V případě **vnitřního uzlu** se rekursivně předá hledání do 1 z potomků. Přitom se rozhoduje dle hodnot maximálních klíčů podstromů. Ty vlastně rozdělují hodnoty klíčů na intervaly a podle toho, kam hledaný klíč spadne, rozhodne se o cestě dál.

3.2.5 Funkce findMin

Tato funkce má tuto hlavičku:

```
struct TItem * findMin(struct node *root);
```

Její algoritmus je vskutku jednoduchý: z vrcholu se vydává vždy do 1. potomku (nejvíc vlevo), dokud nedorazí do listu (ty jsou všechny ve stejné výšce, a jelikož jsou na této úrovni již seřazeni, nalezne se skutečně nejmenší).

3.2.6 Funkce findMax

Tato funkce má takovouto hlavičku:

```
struct TItem * findMax(struct node *root);
```

Její algoritmus je velice podobný funkci `findMin`: z vrcholu se vydává vždy do 3. potomku či (pokud ho nemá) do 2. potomku (tj. nejvíc napravo), dokud nedorazí do listu.

3.2.7 Funkce previous

Funkce s následující hlavičkou:

```
struct TItem * previous(struct node *root, int sKey);
```

Algoritmus nejprve nalezne uzel s daným klíčem (pokud neexistuje, vratí `NULL`), potom se snaží najít jeho nejbližšího levého sourozence.

Pokud ho nenalezne, postoupí ke svému rodiči „nahoru“ a tam se rekursivně opět pokusí najít nejbližšího levého sourozence.

Pokud ho nalezne, vrací maximum podstromu tohoto sourozence.[10] To odpovídá tomu, že chceme najít největší klíč, který je menší než ten zadaný.

Pokud dorazí až do kořene stromu, nemůže dál a znamená to, že zadaný klíč byl ve stromu nejmenší.

3.2.8 Funkce next

Funkce s následující hlavičkou:

```
struct TItem * next(struct node *root, int sKey);
```

Opět zcela analogicky algoritmus nejprve nalezne uzel s daným klíčem (pokud neexistuje, vratí `NULL`), potom se snaží najít jeho nejbližšího pravého sourozence. Vše je opravdu podobné funkci `previous` jen s tím rozdílem, že se hledá pravý sourozenec a pak se spustí `findMin`.

[10] s využitím funkce `findMax` - viz výše

Část III
Uživatelský manuál

Pro účely názorné demonstrace knihovny *twhree* byl sepsán obslužný program umístěný v souboru `main.c`. Ten je zcela separován od knihovních funkcí, tudíž ho lze použít na demonstrace jiných stromů. Naopak i knihovnu *twhree* lze univerzálně užívat v jiných programech. Pro ukázku práce s touto obsluhou zde bude zobrazeno několik výstupů z obrazovky a způsobu zadávání příslušných vstupních dat.

4 Menu

Při spuštění obsluhy nás uvítá zpráva o aktuální verzi ovládacího programu:

```
Welcome to twhree v2.3!
This is a demonstration program of twhree [T(wo)-(t)H(ree)
(t)REE] library.
```

Následuje nabídka příkazů:

```
----------------------------------------------------------------
Choose option (enter the part in brackets)
(I)nsert         - insert a new node
(D)elete         - delete the node containing specified value
delete (A)ll     - delete the node containing specified value
(F)ind           - ascertain the presence of the value in the tree
find(M)in        - show the minimum of the tree
findma(X)        - show the maximum of the tree
(P)revious       - show the previous item according to the key
(N)ext           - show the next item according to the key
displa(Y)        - display tree (depth-first, preorder)
c(L)ear          - clear screen (only for in *nix like OS)
(H)elp           - displays this menu:)
(Q)uit           - quit the program
----------------------------------------------------------------
>>
```

Jak vidno, tuto nabídku lze kdykolit znova vyvolat pomocí příkazu[11] `h`:

```
>> h
```

[11] Příkazy lze zadávat velkými i malými písmeny abecedy. Dokonce ani nevadí, když jsou za prvním písmenem další znaky, ty se do konce řádku ignorují.

5 Vstup a příkazy programu

5.1 Insert

Příkaz vložení nové položky se zadaným klíčem a textem (reprezentující data). Na tu pak ve stromu ukazuje nově vkládaný uzel.

```
>> i
Enter new item...
 New key: 1
 New name: aaa
Adding "aaa" with key 1......OK
```

5.2 Delete

Příkaz smazání položky se zadaným klíčem včetně k ní příslušného uzlu ve stromu.

```
>> d
 Which key: 1
Deleting node with key 1......OK
```

V případě nenalezení zadaného klíče:

```
>> d
 Which key: -1
Deleting node with key -1......KEY NOT FOUND!
```

Je-li dokonce strom prázdný:

```
>> d
 Which key: -1
Deleting node with key -1......EMPTY TREE!
```

5.3 Delete all

Příkaz smazání všech položek a tím i celého stromu.[12]

```
>> a
```

[12] Mimochodem tento příkaz se defaultně volá při ukončení programu, neb data nejsou ukládána externě do souboru na disku, nýbrž jsou zachovávána interně v hlavní paměti počítače a tam po skončení práce pak jen zabírají zbytečně místo.

5.4 Find

Příkaz vyhledání položky se zadaným klíčem.

```
>> f
 Search key: 1
Serching key 1..... (1,"aaa")
```

V případě nenalezení zadaného klíče:

```
>> f
 Search key: -1
Serching key -1......NOT FOUND!
```

5.5 Find min

Příkaz vyhledání položky se nejmenším klíčem.[4]

```
>> m
Searching minimal key..... (1,"1")
```

5.6 Find max

Příkaz vyhledání položky se největším klíčem.[13]

```
>> x
Searching maximal key..... (1,"1")
```

5.7 Previous

Příkaz vyhledání položky s nejbližší nižší hodnotou klíče.

```
>> p
 Previous of which key? 2
Searching previous item of item with 2 key..... (1,"aaa")
```

5.8 Next

Příkaz vyhledání položky s nejbližší vyšší hodnotou klíče.

```
>> n
 Next of which key? 0
Searching next item of item with 0 key..... (1,"aaa")
```

[13] V případě existence více takových položek nalezne tu, jež byla přidána první. To by se nemělo stávat, neboť klíče by měly být v databázích unikátní - možná bude předmětem zájmu v dalších verzích.

5.9 Display

Příkaz zobrazení 2-3 stromu. Každá hlubší úroveň stromu je o 1 mezeru více odsazená než předchozí, bezprostředně vždy následují potomci aktuálního uzlu (a popř. jejich potomci s příslušným odsazením). Vnitřní uzly jsou v [hranatých závorkách] spolu s maximálními klíčemi levého popř. prostředního stromu, listy zas v (kulatých závorkách) s klíčem a hodnotou dat příslušné položky:

```
>> y
[4]
 [2]
  [1]
   (1,"a")
   (2,"b")
  [3]
   (3,"c")
   (4,"d")
 [6|8]
  [5]
   (5,"e")
   (6,"g")
  [7]
   (7,"f")
   (8,"h")
  [9|10]
   (9,"k")
   (10,"l")
   (11,"j")
```

5.10 Clear

Příkaz na vyčištění obrazovky (v abstraktním slova smyslu, samozřejmě, jinak použijte suchý hadřík či látku `:-)` Funguje převážně na systémech *nixového typu, páč využívá systémové procedury `clear`.

```
>> l
```

5.11 Quit

A tady končí naše cesta...

```
>> q
Bye bye...
```

Reference

[1] http://en.wikipedia.org/wiki/2-3_tree

[2] http://pages.cs.wisc.edu/~vernon/cs367/notes/10.23TREE.html

[3] Doc. RNDr. Töpfer, Pavel. *Algoritmy a programovací techniky*, 2. vydání, Prometheus, Praha (2007). ISBN 978-80-7196-350-9.