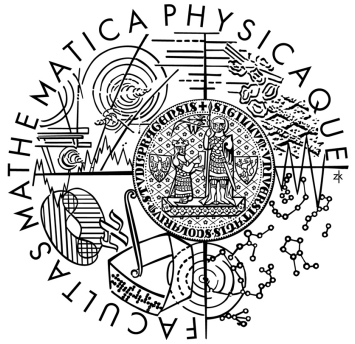
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jakub Hájek

Persistentní datové struktury v C

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Správa počítačových systémů

Praha 2011

Děkuji panu RNDr. Michalovi Kopeckému, Ph.D. za hodnotné rady, podnětné připomínky, odborné vedení a obrovskou ochotu s čímkoli pomoci během mé práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 4.8.2011 Jakub Hájek

Název práce: Persistentní datové struktury v C
Autor: Jakub Hájek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.
e-mail vedoucího: kopecky@ksi.mff.cuni.cz

Abstrakt: Tato práce popisuje návrh a implementaci frameworku pro vytváření persistentních stromových datových struktur a vzorovou implementaci těchto struktur. Rozebírá, jak efektivně takové struktury implementovat a zajistit uživateli pohodlný a transparentní přístup k datům. Popisuje, jaké služby by měla persistentní vrstva poskytovat a snaží se navrhnout takové rozhraní, které umožní snadnou změnu datového úložiště. Dále z výkonového hlediska srovnává tři různá datová úložiště, která byla využita pro zajištění stálosti dat. Dle provedeného měření určí, zda je tento přistup použitelný, případně pro který druh aplikací.

Klíčová slova: persistentní stromy, C, datové struktury

Title: Persistent data structures in C

Author: Jakub Hájek
Department: Department of software engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: kopecky@ksi.mff.cuni.cz

Abstract: This work describes the design and implementation of persistent data structures framework for C programming language. It discusses effective implementation of persistent structures and providing transparent manipulation of data. This work defines services which should be provided by persistent layer and tries to define some general persistent layer interface for good extensibility. In the end it analyses performance issues of various data storages and tries to specify class of applications, which could use persistent data structures framework.

Keywords: persistent tree structures, C, data structures

Obsah

1. Úvod

Persistentní datovou strukturou[1] myslíme takovou datovou strukturu, která je mimo své reprezentace v paměti uložena navíc v nějakém persistentním úložišti (typicky pevný disk), takže je možné strukturu, případně její část, obnovit i po jejím uvolnění z paměti nebo ukončení aplikace.

Využívaní persistentních struktur je pro řadu dnešních aplikací typické. Nejvýznamnějším příkladem jsou frameworky pro objektově relační mapovaní objektů do relačních databází[2]. Ty poskytují programátorovi možnost pracovat s objektem jako by byl pouze v paměti a jeho více či méně transparentní ukládání do databáze, typicky pomocí metod `load()`, `persist()` a `delete()`.

Důkazem oblíbenosti tohoto přístupu může být například jazyk Java a přidání persistentního API do standardní knihovny a dále rozšíření jazyka o konstrukty pro definování mapování, klíčových atributů a persistentních nebo transientních atributů.

Na rozdíl od objektově relačního mapování se práce zabývá transparentní persistencí rozsáhlejších dynamických datových vyhledávacích struktur, jakými jsou vyhledávací stromy, haldy a seznamy. Motivací je, mimo zřejmého uchování dat i po uvolnění z paměti, možnost existence stromové struktury o velikosti vetší, než je velikost dostupné paměti s tím, že v paměti se udržuje pouze část struktury, se kterou se právě pracuje.

Druhá kapitola rozebírá existující řešení persistence stromových struktur, jejich nevýhody a naopak výhody a funkčnosti, které je možné pro persistenci stromových struktur využít.

Ve třetí kapitole jsou rozebrány možnosti, kterými lze persistenci implementovat, argumenty pro vytvoření samostatné, od datových struktur oddělené persistenční vrstvy a popis rozhraní a služeb, které by měla persistenční vrstva pro vytváření persistentních stromových struktur poskytovat.

Ve čtvrté kapitole je popsaná technická realizace z pohledu programátora a popis jak knihovnu rozšířit o další datové struktury a/nebo podporu jiného persistentního úložiště.

Pátá kapitola popisuje vzorovou implementaci z pohledu uživatele - programátora - využívajícího služeb knihovny.

[1] označení persistentní datová struktura se také používá pro datovou strukturu, která udržuje nejen svůj aktuální stav, ale i historické verze. Takové struktury však v této práci neuvažujeme.

[2] exsitují i frameworky pro ukládání objektů do objektových databází, ty se však v komerční sféře příliš neujaly

V šesté kapitole je popsán postup testování výkonu při použití různých persistenčních úložišť a shrnutí výsledků.

Závěrem je uvedeno shrnutí, zhodnocení výsledků, případné možnosti zlepšení vzorové implementace a zhodnocení, pro kterou třídu aplikací by byla vzorová implementace vhodná k nasazení.

2. Existující řešení

Autorovi se nepodařilo nalézt žádné exitující řešení, které by pro jazyk C poskytovalo persistentní stromové struktury s rozhraním, které je od nich očekávané - např. s metodami pro vrácení kořene, levého syna a podobně.

Existují však embedded databáze poskytující C API, které uživateli nabízejí persistentní hash tabulky. Mezi zástupce takových databází patří například DBM, která se stala první rozšířenou on-disk hashtable databází. Již se však nevyvíjí, má pouze historický význam.

Dalším zástupcem je on-disk hashtable databáze GDBM[2], což je svobodná reimplementace NDBM - databáze, která vycházela z DBM, ale odstranila její nedostatky - především ošklivé API a nemožnost otevřít více než jednu databázi současně.

Stále aktivně vyvíjenou, velmi rozšířenou a dostatečně robustní on-disk databází je Oracle Berkeley DB[3].

Jejich hlavní nevýhodou je však to, že jedinou strukturou, kterou poskytují je hash tabulka - úložiště pro dvojici klíč-hodnota. Další nevýhodu je skutečnost, že data jsou pouze na disku a časté čtení jedné položky je zbytečně pomalé (pomineme-li možnost nastavení cacheování u Berkeley DB).

Na druhou stranu uložení a vyhledání dvojice klíč-hodnota implementují efektivně a mohou být použity pro trvalé uložení částečných dat stromových struktur. Vzorová implementace, která je součástí této práce, využívá z tohoto důvodu jako persistentní úložiště právě GDBM a Berkeley DB. Třetím implementovaným úložištěm je relační databáze Oracle.

3. Možnosti implementace persistentních struktur

3.1 Implementace přímo ve struktuře

Pravděpodobně nejpřímočařejším, avšak nejhorším přístupem, je implementace persistence přímo v implementaci datové struktury. To znamená vkládat volání API funkcí persistentního úložiště případně psát SQL příkazy v případě využití relační databáze přímo do kódu, který manipuluje s jednotlivými strukturami. To s sebou přináší zjevné nevýhody, jako je pevná vazba na konkrétní datové úložiště a nutnost implementovat persistenci s každou datovou strukturou znovu.

Snad jedinou výhodou může být rychlá implementace jedné konkrétní struktury s jedním konkrétním datovým úložištěm.

3.2 Vytvoření vrstvy uzlů

Pokud budeme chtít implementovat více různých stromových struktur, můžeme implementovat samostatnou dostatečně obecnou vrstvu pro práci se stromy, která bude poskytovat funkce, jako je přidání a odebrání uzlu, nastavení vlastních dat uzlu, vytvoření nového samostatného uzlu a podobně. Tato vrstva pak bude implementovat samotné ukládání a načítání z persistentního úložiště.

Výhodou potom je, že struktury, které lze reprezentovat jako stromy (byť degradované - např. spojové seznamy), lze implementovat nad touto vrstvou a není nutné persistenci implementovat pokaždé znovu. Nevýhoda svázání s konkrétním úložištěm však přetrvává.

3.3 Oddělení vrstvy uzlů a samostatné persistenční vrstvy

Nejlepším řešením je vytvoření samostatné persistenční vrstvy s jasně definovaným rozraním. Změna datového úložiště pak znamená pouze reimplementaci této vrstvy, která splňuje definované rozhraní a použití vrstvy ve vrstvě uzlů. Tento přístup je zvolen i v pilotní implementaci.

3.4 Požadavky na persistenční vrstvu

Generování identifikátorů - Každá samostatná položka datové struktury - v našem případě uzel stromu musí mít vazbu mezi svojí reprezentací v paměti a svojí persistentní reprezentací v úložišti. Toho dosáhneme přidělením jednoznačného identifikátoru oběma reprezentacím.

Jako takový identifikátor dobře poslouží unikátní kladné celé číslo. Lze ho získat například tak, že najdeme položku s maximálním identifikátorem v persistentním úložišti a přičteme k němu jedničku. To však může být v případě některých persistenčních úložišť, jakou jsou třeba on-disk hash tabulky zbytečně pomalé, protože musíme projít všechny záznamy.

Další možností je mít v persistentním úložišti hodnotu maximálního identifikátoru uloženou (například jako unikátní záznam v hash-table databázi, nebo jako tabulka s jedním řádkem a sloupcem v relační databázi). Potom lze tuto hodnotu zvýšit o jedna a vrátit jako nový identifikátor.

Někdy persistentní úložiště poskytuje tuto funkcionalitu samo, například `sekvence` v databázi Oracle nebo `auto increment` v MySQL.

Vytvoření nového uzlu - persistenční vrstva vytvoří persistentní reprezentaci nového uzlu dle zadaných dat

Vrácení uzlu dle zadaného identifikátoru - persistenční vrstva musí být schopna rychle vrátit a v paměti vytvořit reprezentaci uzlu. Při použití relační databáze jako persistentního úložiště je nutné správné použití indexů. V případě persistentních hash tabulek je nutné hledat dle klíčových položek[3].

Změna dat uzlu - persistenční vrstva musí opět efektivně dle zadaného identifikátoru uzlu vyhledat a změnit data uzlu

Změna odkazů - pro stromové struktury je nezbytné mít možnost změnit odkazy na potomky, případně rodiče. Konkrétní možnosti, jak tyto informace uchovávat a aktualizovat, jsou popsány v následující kapitole.

Vrácení konkrétního potomka - persistenční vrstva musí umět vrátit konkrétního potomka (případně rodiče) uzlu dle daného identifikátoru.

3.5 Požadavky na vrstvu uzlů

Na vrstvu uzlů jsou kladeny s funkčního hlediska stejné požadavky jako na persistenční vrstvu. Využívá rozhraní persistenční vrstvy a zároveň provádí ekvivalentní

[3] konkrétní možnosti implementace jsou popsány ve čtvrté kapitole

operace na reprezentaci uzlu v paměti

Dále by neměla načítat uzly z persistentního úložiště zbytečně, pokud již existuje paměťová reprezentace. To lze vyřešit speciálním globálním ukazatelem, jehož hodnota není interpretována jako adresa v paměti, ale jako odkaz do persistentního úložiště. Další očekávanou funkcionalitou je uvolnění konkrétní paměťové reprezentace.

3.6 Požadavky na datové struktury

Kromě konkrétních operací, které jsou specifické pro konkrétní datové struktury, by měly struktury poskytovat i funkce pro uvolnění svých částí z paměti, především by pak měly rozlišovat mezi operacemi `free()` - uvolnění pouze paměťové reprezentace a `delete()` - uvolnění z paměti a současné vymazání z persistentního úložiště. To vše prostřednictvím vrstvy uzlů.

4. Technický koncept a popis vzorového řešení

4.1 Logické rozdělení do vrstev

Jak již bylo zmíněno v předchozí kapitole, implementace je rozdělena do tří vrstev - nejnižší vrstvou je persistenční vrstva, zprostředkovávající komunikaci s datovým úložištěm pomocí odpovídajícího jazyka a protokolu. Nad ní je vrstva uzlů, která nabízí své služby nezávisle na použitém úložišti vývojářům aplikací. Nad vrstvou uzlů jsou dále implementovány jednotlivé datové struktury - vrstva datových struktur. Ta nabízí již implementované prostředky pro práci s vybranými datovými strukturami. Aplikace by měla přímo využívat pouze vrstvu struktur. V případě potřeby pak může programátor doplnit tuto vrstvu o podporu dalších typů datových struktur. Služeb persistenční vrstvy by pak měla využívat pouze vrstva uzlů. Jedinou výjimku tvoří inicializace persistenční vrstvy, která musí být provedena z aplikace. Přes nižší vrstvy se pak z aplikace předává persistenční vrstvě databázový *handler*. Jeho konkrétní podoba závisí již na konkrétní persistenční vrstvě - může se jednat například pouze o textový řetězec s významem cesty k souboru, do kterého jsou data ukládána. Persistenční vrstva tedy musí poskytovat aplikaci funkci, která tento handler vytvoří a inicializuje a dále funkci, která tento handler zruší. Jsou to však jediné dvě funkce persistenční vrstvy, které jsou volané přímo z aplikace.

4.2 Vrstva uzlů

Poskytuje metody pro manipulaci s uzly stromových struktur a transparentně činí změny persistentním voláním funkcí persistenční vrstvy. Všechny veřejné funkce této vrstvy jsou definovány v hlavičkovém souboru `node.h`. Všechny funkce z tohoto modulu mají prefix `no_`.

4.2.1 Struktura node

Hlavní strukturou této vrstvy je struktura `node`. Ta reprezentuje jeden uzel datové struktury a je základní jednotkou, kterou je možné do persistentního úložiště ukládat a naopak z něj načítat. Struktura `node` je definována následovně:

```
typedef struct _node{
  struct _node ** childs;
  unsigned size_of_childs;
```

```
  void * data;
  size_t data_size;
  int  color;
  struct _node * parent;
  unsigned long node_id;
  unsigned long structure_id;
} node;
```

Persistentními atributy této struktury jsou `node_id`, což je unikátní identifikátor uzlu v rámci úložiště, dále identifikátor `structure_id` společný pro všechny uzly struktury a atribut `color`, který reprezentuje barvu uzlu v červeno-černých stromech a balanci v AVL stromech. Kromě nich uzel obsahuje atributy `data` a `data_size`. Atribut `data` představuje ukazatel na vlastní data uzlu[4]

a `data_size` jejich velikost.

Do persistentního úložiště jsou data ukládána jako prostý úsek paměti, proto v případě, že tato data obsahují ukazatele, budou po načtení z persistentního úložiště sice ukazovat na stejné místo, na kterém však už s největší pravděpodobností nebudou data jako v době ukládání ukazatele. Pokud je potřeba persistentně uchovávat data, která nejsou v paměti souvislá a jsou mezi nimi ukazatele, můžeme využít techniku tzv. *marshallingu*. Tato technika bude popsána později spolu s popisem implementace B-stromu, protože právě zde je využívána.

Dále jsou ve struktuře `node_id` definovány nepersistentní atributy - pole `childs` ukazatelů na potomky, velikost tohoto pole a ukazatel na předka. Pole potomků je indexováno, jak už je v C zvykem, od nuly. To znamená, že `N->childs[0]` je ukazatel na prvního (nejlevějšího) potomka uzlu `N`[5].

4.2.2 Makro in_persistence_layer

Další podstatnou definicí je makro `in_persistence_layer`, které představuje speciální ukazatel na uzel, který již není v paměti, ale existuje jeho reprezentace v persistentním úložišti. Pokud chceme „dereferencovat“ takovýto ukazatel nalezený v poli potomků, zavolá se funkce persistenční vrstvy, která potomka vyhledá dle identifikátoru rodiče a pozice potomka v poli potomků tohoto rodiče.

4.2.3 Funkce no_new_node()

Pro vytvoření nového uzlu je deklarována funkce

```
node * no_new_node(
```

[4]tyto data budeme označovat jako „ hodnota uzlu “

[5]v dalším textu bude pod pojmem „pozice potomka“ myšlen index v tomto poli

```
  void * data,
  size_t data_size,
  int color,
  unsigned child_number,
  unsigned long structure_id,
  void * db
),
```

která vrací ukazatel na nově vytvořený uzel a má tyto parametry:

- `data` - ukazatel na data, která budou v uzlu uložena
- `data_size` - velikost ukládaných dat
- `color` - hodnota atributu color ve struktuře `node`
- `child_number` - počáteční velikost pole potomků
- `structure_id` - identifikátor struktury, který je pro každou strukturu v úložišti identifikovaným handlerem `db` unikátní. Tato hodnota je stejná pro všechny uzly struktury
- `db` - ukazatel na databázový handler.

Jako příklad je uvedeno vytvoření nového uzlu `uzel` s celočíselnou hodnotou 3, barvou 1 a identifikátorem struktury 5, u kterého očekáváme 2 potomky. Předpokládáme existenci handleru datového úložiště a ukazatele na něj `db`.

```
#include "node.h"
int hodnota = 3;
node * uzel;
uzel = no_new_node(&hodnota, sizeof(int), 1, 2, 5, db);
```

4.2.4 Funkce no_add_or_replace_child()

Pro vytvoření vztahu předek-potomek slouží funkce

```
node * no_add_or_replace_child(
  node * parent,
  node * child,
  unsigned position,
  void *db
)
```

Jejími parametry jsou ukazatel na předka, ukazatel na potomka, pozice potomka a ukazatel na databázový handler. Vrací ukazatel na přidávaného potomka.

Následuje příklad, který existujícímu uzlu, na nějž ukazuje ukazatel, `otec` přiřadí potomka `syn` na pozici 0 - tedy nejlevějšího potomka.

```
#include "node.h"
int hodnota = 3;
node * otec, * syn;
otec = no_new_node( .. );
syn  = no_new_node( .. );
no_add_or_replace_child(otec, syn, 0, db);
```

4.2.5 Funkce no_disinherit_child()

Naopak pro zrušení vztahu předek-potomek je zde funkce

```
node * no_disinherit_child(
  node * parent,
  unsigned position,
  void * db
),
```

která pro zadaného rodiče `parent` a pozici potomka `position` zruší tento vztah v paměťové reprezentaci, zavolá funkce persistenční vrstvy pro aplikování této změny do úložiště identifikovaného handlerem `db` a vrátí ukazatel na „vyděděného“ potomka.

Vztah rodič-potomek z předchozího příkladu zrušíme zavoláním

```
syn = no_disinherit_child(otec, 1, db);
```

4.2.6 Funkce no_delete_node()

Uzel se z paměti i persistentního úložiště vymaže voláním funkce

```
void * no_delete_node(
  node * del_node,
  void * db
)
```

Prvním parametrem je ukazatel na uzel, druhým je ukazatel na databázový handler. Funkce vrátí ukazatel na data vymazaného uzlu. Ta je potřeba explicitně uvolnit, pokud byla paměť alokována dynamicky na haldě.

Funkce nezajišťuje úpravu odkazů v případném rodiči ani odkazy na potomky. Předpokládá tedy jako vstup uzel, který nemá ani potomky ani rodiče. Před voláním `no_delete_node()` může být proto potřeba toto vyřešit pomocí funkce `no_disinherit_child()` .

Uvedeme příklad, který vymaže uzel s daty v dynamicky alokované paměti.

```
#include "node.h"
int * hodnota;
node * uzel;
void * data;
hodnota = malloc(sizeof(int);
*hodnota = 3;
uzel = no_new_node(hodnota, sizeof(int), 1, 2, 3, db);
data = no_delete_node(uzel,db);
free(data);
```

4.2.7 Funkce no_free_node()

Pro uvolnění paměťové reprezentace uzlu slouží funkce

```
void * no_free_node(
  node * del_node
),
```

která uvolní daný uzel z paměti a vrátí ukazatel na data vložená do uzlu. Tato data nelze implicitně uvolňovat, protože není jisté, zda byla dynamicky alokována na haldě. Na to je třeba dát pozor a případně paměť po zavolání funkce uvolnit. Příklad použití by byl analogický s příkladem u funkce `no_delete_node()` .

4.2.8 Funkce no_free_child()

Pro uvolnění paměťové reprezentace potomka slouží funkce

```
void * no_free_child(
  node * parent,
  unsigned position
),
```

která jako svůj první parametr bere ukazatel na rodiče a jako druhý parametr pozici potomka. Funkce nastaví v poli potomků ukazatel `in_persistence_layer` a vrátí ukazatel na data uvolněného uzlu.

Jako příklad uvedeme uvolnění pravého potomka uzlu `otec` v binárním stromě. Předpokládáme, že potomek má data - hodnotu uzlu - v dynamicky alokované paměti.

```
void *data;
data = free_child(otec, 1);
free(data);
```

4.2.9 Funkce no_restore_node_byid()

Ve chvíli, kdy je potřeba přistupovat k uzlu, který byl z paměti uvolněn, může být obnoven z persistentního úložiště podle svého id voláním funkce

```
node * no_restore_node_byid(
  unsigned long lid,
  void * db)
```

Prvním parametrem funkce je identifikátor uzlu v persistentním úložišti a druhým je databázový handler. Jak uzel, tak jeho data jsou v nově dynamicky alokované paměti. Tuto funkci je potřeba volat s rozmyslem, aby nedošlo k obnovení uzlu, který již svou paměťovou reprezentaci má. Pokud by se pak prováděly změny na dvou různých paměťových reprezentacích, které by byly transparentně aplikovány do persistentního úložiště, mohlo by docházet k nekonzistencím.

Uzel, který je identifikován číslem 2, načteme do paměti a ukazatel na načtený uzel získáme následovně:

```
  void *node;
  node = no_restore_node_byid(2,db);
```

4.2.10 Funkce no_get_child()

Pro transparentní získání potomka uzlu slouží funkce

```
node * no_get_child(
  node * parent,
  unsigned position,
  void * db
)
```

Funkce se volá se třemi parametry: `parent` je ukazatel na uzel, jehož potomka chceme získat, `position` je jeho pozice v poli potomků a `db` je ukazatel na handler persistentního úložiště. Funkce vrací ukazatel na paměťovou reprezentaci potomka. Pokud je v poli potomků nalezen ukazatel do persistentního úložiště, je potomek z persistentního úložiště obnoven voláním funkce persistenční vrstvy a je vytvořena jeho paměťová reprezentace. Pokud pro potomka neexistuje ani paměťová ani persistentní reprezentace, funkce vrátí `NULL`.

Následující příklad ukazuje jak získat pravého potomka uzlu `otec` v binárním stromě.

```
node * pravy_syn;
pravy_syn = no_get_child(otec, 1, db);
```

4.2.11 Funkce no_set_color()

```
int no_set_color(
  node * nod,
```

```
  int color,
  void * db
)
```

Tato funkce nastaví atribut `color` uzlu `nod` a zavolá funkci persistentní vrstvy pro aplikování změny do persistentního úložiště identifikovaného handlerem `db`.

Příkladem budiž nastavení červené barvy, která je reprezentována hodnotou 1 uzlu `rbnode`.

```
no_set_color(rbnode, 1, db);
```

4.2.12 Funkce no_get_newstructure_id()

Funkce `unsigned long no_get_newstructure_id (void * db)` vrátí nový unikátní identifikátor pro identifikaci datové struktury v úložišti identifikovaném handlerem `db`. Jedná se jen o wrapper funkce persistenční vrstvy, aby z vrstvy struktur nemusela být volána funkce persistenční vrstvy, což by bylo mírně proti návrhu oddělených vrstev.

4.2.13 Funkce no_update_pl_data()

Funkce

```
void no_update_pl_data(
  unsigned long id,
  void * data,
  size_t data_size,
  void * db)
```

změní data persistentní reprezentaci uzlu. Měněný uzel je v úložišti identifikovaném handlerem `db` identifikovaný identifikátorem `id` a jsou mu nastavena data o velikosti `data_size`, která jsou v paměti uložená na adrese `data`.

Následující příklad změní data uzlu `uzel` jak v paměti, tak v persistentním úložišti.

4.2.14 Příklad použití funkcí vrstvy uzlů

Následující příklad ukazuje jak vytvořit spojový seznam s uzly x a y, poté uvolnit y, následně obnovit a vymazat.

```
//include funkcí vrstvy uzlů
#include "node.h"
```

```
//include funkcí persistenční vrstvy
#include "dg_persistence.h"
void main(void){
  //deklarece proměnných
  node *x,*y;
  int * data;
  //inicializace persistentního úložiště a vytvoření handleru
  void * db  =  pl_init("connection_string");

  /* vytvoření dat pro uzel */
  val=(int*)malloc(ziseof(int);
  *data = 123;

  /* vytvoření uzlu x s daty na adrese val, atributem color 7,b  */
  /* velikostí pole pro potomky 1 a novým identifikátorem struktury */
  /* v úložišti s handlerem db */
  x = no_new_node(data, sizeof(int), 7, 1,
              no_get__newstructure_id(db),  db );

  /* podobně pro uzel y */
  data = (int*)malloc(ziseof(int);
  *data = 222;
  y = no_new_node(data, sizeof(int), 7, 1,
                  x->strucure_id,  db );

  /* přidání nejlevějšího (nultého) potomka y rodiči z*/
  no_add_or_replace_child(x,y,0,db);

  /* jeho uvolnění z paměti */
  data = no_free_child(x,0);
  free(data);

  /* jeho opětovné načtení do paměti */
  y = no_get_child(x,0,db);

  /* zrušeni vztahu x-y */
  no_disinherit_child(x,0,db);

  /* a vymazání */
  data = no_delete_node(y, db);
  free(data);

  /* uvolnění handleru */
  pl_close(db);
}
```

4.3 Persistenční vrstva

Rozhraní, které splňuje každá implementace persistenční vrstvy, je definované v hlavičkovém souboru `dg_persistence.h`. Všechny funkce persistentní vrstvy mají prefix `no_`. V následujících odstavcích je popsáno, jaké chování se od funkcí tohoto rozhraní očekává.

4.3.1 Funkce pl_init()

Pro inicializaci datového úložiště slouží metoda
`void * pl_init(char* connection_string)`, která jako svůj jediný parametr bere inicializační řetězec a vrátí ukazatel na databázový handler. Jak inicializační řetězec, tak typ handleru závisí na konkrétním použitém úložišti.

4.3.2 Funkce pl_close()

Funkce `void pl_close(void * db)` zruší databázový handler `db` a korektně uzavře datové úložiště (např. potvrdí transakce, uzavře soubor - záleží na konkrétním úložišti.).

4.3.3 Funkce pl_get_newnode_id()

Funkce `unsigned long pl_get_newnode_id( void * db)` vrátí nový unikátní identifikátor pro identifikaci uzlu v úložišti identifikovaném handlerem `db`.

4.3.4 Funkce pl_get_newstructure_id()

Funkce `unsigned long pl_get_newstrucure_id( void * db)` vrátí nový unikátní identifikátor pro identifikaci struktury v úložišti identifikovaném handlerem `db`

4.3.5 Funkce void pl_create_node()

Pro vložení nového uzlu do persistentního úložiště slouží funkce

```
void pl_create_node(void * data, size_t data_size, int color,
      unsigned long  node_id,  unsigned long structure_id,
      unsigned long  parent_id, void * db
```

s těmito parametry:

- `data` - ukazatel na data, která budou v uzlu uložena
- `data_size` - velikost ukládaných dat
- `color` - hodnota atributu color
- `node_id` - unikátní identifikátor uzlu
- `structure_id` - identifikátor struktury
- `parent_id` - identifikátor rodiče - pokud uzel rodiče nemá, je jako identifikátor rodiče použita stejná hodnota jako pro identifikátor uzlu.
- `db` - ukazatel na databázový handler.

Vytvoření nového záznamu v úložišti reprezentujícího uzel shrnuje následující příklad. Uzel bude mít hodnotu 5 a barvu 0 a nebude mít žádného potomka ani předka. Předpokládá existenci databázového handleru `db`.

```
unsigned long int id;
int data = 5;
id = pl_get_newstructure_id(db);
void pl_create_node(&data, sizeof(int) 0, id,
                    pl_get_newstrucure_id(db), id, db);
```

4.3.6 Funkce pl_delete_node_by_id()

K vymazání uzlu z datového úložiště dle jeho unikátního identifikátoru slouží funkce.

```
void pl_delete_node_by_id(
  unsigned long id,
  void * db
)
```

Předpokládá uzel, který nemá potomky ani předka. Uzel s identifikátorem 7 se potom z datového úložiště vymaže voláním

```
void pl_delete_node_by_id(7, db)
```

4.3.7 Funkce pl_update_parent_and_position()

Pro změnu předka uzlu s identifikátorem `id` na uzel s identifikátorem parent slouží funkce.

```
void pl_update_parent_and_position(
  unsigned long id,
  unsigned long parent,
  unsigned position,
  void * db
)
```

Tato funkce zajišťuje i případné změny odkazů na potomka v původním rodiči.

Jako příklad poslouží nastavení rodiče s identifikátorem 1 potomkovi s identifikátorem 2 tak, že to bude druhý, v případě uzlu binárního stromu pravý, potomek.

```
pl_update_parent_and_position(2, 1, 1, db);
```

4.3.8 Funkce void pl_disinherit_child()

Funkce

```
void pl_disinherit_child(
  unsigned long parent,
  unsigned position,
  void * db
)
```

zruší vazbu mezi uzlem `parent` a jeho potomkem na pozici `position` v úložišti identifikovaném handlerem `db`.

Jako příklad uvedeme odebraní prvního potomka rodiče s identifikátorem 1.

```
void pl_disinherit_child(1,0,db);
```

4.3.9 Funkce pl_restore_node_byid

Funkce

```
node * pl_restore_node_byid(
  unsigned long id,
  void * db
)
```

vytvoří paměťovou reprezentaci uzlu s identifikátorem `id`, načteným z úložiště, identifikovaném handlerem `db` a vrátí ukazatel na tuto reprezentaci.

Pro obnovení uzlu s identifikátorem 1 zavoláme

```
node * obnoveny = pl_restore_node_byid(1,db);
```

4.3.10 Funkce pl_restore_child()

Funkce

```
node * pl_restore_child(
  unsigned long parent,
  unsigned position,
  void * db
)
```

vytvoří paměťovou reprezentaci potomka uzlu a vrátí ukazatel na tuto reprezentaci. Parametrem `parent` je identifikátor uzlu, jehož předek bude načten z úložiště identifikovaném handlerem `db`. Parametrem `position` je pořadí uzlu, který má být obnoven - číslováno od nuly.

Pro obnovení prvního potomka uzlu s identifikátorem 1 zavoláme

```
node * obnoveny = pl_restore_child(1, 0, db);
```

4.3.11 Funkce pl_update_color()

Změna atributu color v persistentní reprezentaci uzlu s identifikátorem `id` se provede voláním funkce

```
void pl_update_color(
  unsigned long id,
  int color,
  void * db
).
```

Parametr `db` je handler úložiště, v němž se reprezentace nachází a parametrem `color` je nová hodnota atributu color.

Barvu 1 uzlu s identifikátorem 4, který je uložen v úložišti s ukazatelem na handler, tedy nastavíme:

```
pl_update_color(4,1,db);
```

4.3.12 Funkce pl_update_data()

Funkce

```
void pl_update_data(
  unsigned long id,
  void * data,
  size_t data_size,
  void * db);
```

změní hodnotu uzlu v persistentní reprezentaci. Parametr `id` je identifikátor uzlu, `data` je ukazatel na nová data v paměti, `data_size` je velikost nových dat a `db` je handler úložiště.

Jako příklad uvedeme změnu dat uzlu s identifikátorem 5 na hodnotu 1234

```
int hodnota = 1234;
pl_update_data(5, &hodnota, sizeof(int), db);
```

4.4 Persistence prostřednictvím databáze Oracle

Persistenční vrstva pro relační databázi Oracle[1] je napsána v Oracle Embedded SQL - `Pro*C`. Jedná se o jazyk, který umožní přímo do kódu v jazyce C vkládat SQL příkazy. Výsledný kód se poté přeloží preprocesorem `Pro*C` do čistého C, který je již následně možné přeložit běžným kompilátorem jazyka C. Pro překlad a následné slinkování a spuštění je nutné mít nainstalované potřebné knihovny dodávané Oraclem[4].

Samotné uzly jsou poté ukládány do tabulky `node`, která je vytvořena následovně:

```
CREATE TABLE NODE
(
ID NUMBER(33,0) NOT NULL, -- id uzlu
PARENT NUMBER(33,0) NOT NULL, -- id rodiče
STRUCTURE_ID NUMBER(33,0) NOT NULL, --id datové struktury
CHILD_POSITION NUMBER(5,0) DEFAULT -1, -- pozice potomka v seznamu rodičů
COLOR NUMBER(5, 0)  DEFAULT 0 NOT NULL, -- hodnota atributu color
DATA BLOB, -- vlastní data uzlu
CONSTRAINT NODE_PK PRIMARY KEY(ID),
CONSTRAINT NODE_PARENT_FK FOREIGN KEY(PARENT ),
    REFERENCES NODE(ID),
CONSTRAINT NODE_PAR_POS_U UNIQUE (PARENT, CHILD_POSITION)
)
```

Integritní omezení na primární klíč a unikátní dvojici (id rodiče, pozice potomka) zajistí dodatečnou kontrolu (byť by se o to dobře napsaná aplikace neměla ani pokusit), zda se nesnažíme vložit podruhé uzel se stejným id, nebo jednomu rodiči přidat dva potomky na stejnou pozici. Mimo to taky vytvoří indexy, pro sloupec `ID` a složený index nad dvojicí `(PARENT, CHILD_POSITION)`. To umožní efektivně vyhledávat konkrétní uzly, případně potomky konkrétního uzlu. Dále se využívá dvou sekvencí pro generování identifikátorů uzlů a struktur.

```
CREATE SEQUENCE  NODE_SEQ
```

[4]Postačuje nainstalovat `Oracle Database Client` stáhnutelný z `http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html`

```
   INCREMENT BY 1 MAXVALUE 4294967295  MINVALUE 100;
CREATE SEQUENCE STRUCTURE_SEQ
   INCREMENT BY 1 MAXVALUE 4294967295  MINVALUE 100;
```

Funkce persistenční vrstvy jsou implementované v souboru `orcl_persistence.pc`.

4.5 Berkeley DB persistence

4.5.1 Uložení dat

Berkeley DB persistence je disková databáze umožňující vytváření persistentních dvojic klíč-hodnota. Využívá k tomu hash tabulek a umožňuje efektivní vyhledání dle klíčové položky.

Do této databáze jsou persistenční vrstvou vkládané tři typy záznamů, identifikované identifikátorem uzlu.

- záznam s vlastními daty, jehož klíčem je identifikátor uzlu a hodnotou jsou data uzlu (atribut `data` struktury `node`).
- záznam s polem potomků, který udržuje identifikátory potomků.
- záznam s dalšími atributy, který uchovává identifikátor rodiče, pořadí uzlu v poli rodičů a atribut `color` struktury `node`.

Každý uzel má potom v databázi právě jeden záznam každého typu.

Databáze chápe hodnotu záznamu jen jako úsek dat, není proto možné načíst pouze jejich část. To vede k tomu, že v případě potřeby změny byť jen jediného bytu v hodnotě je potřeba přečíst celý záznam, provést změny a zpět uložit.

Rozdělení na tři typy záznamů, vede k tomu, že není třeba načítat celá data uzlu při změnách ve vztazích předek-potomek nebo změně atributu `color`.

4.5.2 Typy klíčů

Klíče každého záznamu musí obsahovat identifikátor uzlu, ten však sám o sobě nestačí, protože je potřeba rozlišit, zda pro konkrétní uzel chceme záznam datový, záznam s atributy nebo záznam se seznamem potomků. Proto byla v hlavičkovém souboru `bkdb_persistence.h` vytvořena struktura `bkdb_data_key` definována následovně.

```
typedef struct _data_key{
int key_type;
unsigned long node_id;
} bkdb_data_key
```

Klíčovou položkou v databázi se potom stává dvojice typu záznamu a identifikátoru uzlu.

Pro typ klíče jsou potom v souboru `bkdb_persistence.c`, ve kterém jsou implementované funkce persistenční vrstvy, vytvořeny konstanty `DATA_KEY`, `CHILDREN_KEY` a `ATTR_KEY`, reprezentující klíče pro datové záznamy, záznamy s polem potomků a záznamy s atributy.

Dále je v `bkdb_persistence.h` definována struktura

```
typedef struct _util_key{
  char key_type;
  char key;
} bkdb_util_key,
```

která slouží jako klíč pro záznamy, které nejsou součástí uzlů, přesto však musí být v databázi přítomny. Je využívána pro uložení čítačů pro přidělování identifikátorů uzlů a struktur.

4.6 GDBM persistence

Vzhledem k tomu, že GDBM je stejně jako Berkeley DB on-disk hashtable databáze, je peristenční vrstva pro tuto databázi implementována stejně jako pro Berkeley DB databázi. Pouze je nahrazeno volání API funkcí pro Berkeley DB API funkcemi pro GDBM. Je implementována v souboru `gdbm_persistence.c`.

4.7 Nepersistentní persistenční vrstva

Pro porovnání změny časových nároků aplikace při zavedení persistence struktur byla implementována persistenční vrstva, která drží data výhradně v paměti. Tato vrstva je persistenční jen ve smyslu splnění rozhraní požadované od persistenční vrstvy. Její funkce mají prázdné tělo, případně vrací konstantu.

4.8 B-strom a data marshalling

Implemetace binárních stromů znamená implementaci běžných algoritmů pro binární stromy uložené pouze v paměti s tím, že pro vytváření uzlů a přiřazování a dereferencování ukazatelů na tyto uzly, jsou volány metody vrstvy uzlů.

Při implementaci B-stromu[6] bylo potřeba vyřešit to, že návrh persistenční vrstvy a vrstvy uzlů podporuje uložení pouze jedné hodnoty v uzlu. B-strom však

obecně v jednom uzlu uchovává více hodnot. Řešením by bylo rozšíření vrstvy uzlů a persistenční vrstvy tak, aby bylo možné ukládat atomicky více hodnot. Tím bychom však přišli o možnost získat uzel jedním přístupem do persistentního úložiště.

Místo toho byl zvolen přístup označovaný jako *data marshalling*. Nejedná se o nic jiného než serializaci dat do souvislého úseku v paměti.

4.8.1 Struktura n_datum

V souboru `node.h`[7] je definována struktura `n_datum`,

```
typedef struct n_datum{
size_t dsize;
void * dptr;
} n_datum,
```

která obsahuje ukazatel na data a velikost těchto dat. Ukazatel `data` struktury `node` v implementaci B-stromu potom ukazuje na pole těchto struktur.

Do persistentního úložiště se pak tato data při změně ukládají serializovaná ve formátu:

```
(velikost dat_1 , data_1, velikost dat_2, data_2, ... velikost dat_3, data_3)
```

Samotná serializace se pak provede voláním funkce

```
n_datum marshall_data(node * n),
```

která jako vstup bere uzel, jehož data serializuje a ve struktuře `n_datum` vrací ukazatel na serializovaná data a jejich velikost. Po uložení do persistentního úložiště je potřeba uvolnit paměť se serializovanými daty.

Pro získání potomka uzlu se zde na místo standardní funkce `no_get_child()` volá funkce

```
node * bt_get_child(node *n, unsigned int pos, btree *t),
```

která v případě existence potomka v paměti vrátí tohoto potomka, jinak ho načte z úložiště voláním `no_get_child()`, provede deserializaci dat, přiřadí je uzlu a načtená serializovaná data uvolní z paměti.

[7] Tato struktura je využívána pouze v implementaci B-stromu, proto by se nabízelo definovat ji v souboru s touto implementací, autor však věří, že může být v budoucnu využitá u dalších implementovaných datových struktur

5. Uživatelská dokumentace řešení

Knihovna byla vyvíjena a testován na operačním systému GNU Linux a také následující text přepokládá použití knihovny na tomto systému. Pravděpodobně by však měla jít přeložit i na jiných systémech.

5.1 Výběr persistenční vrstvy

Výběr persistenční vrstvy je velmi snadný, není třeba žádných změn v kódu aplikace, kromě parametru funkce `pl_init`. To, která persistenční vrstva bude použita, se určí až po kompilaci při linkování programu, dle toho, která implementace bude k programu slinkována. Tato implementace však musí splňovat rozhraní definované v souboru `dg_persistence.h` - což splňují všechny implementace vzorového řešení.

Jediné co je v kódu aplikace potřeba změnit dle použité persistenční vrstvy, je inicializace datového úložiště a vytvoření handleru.

5.1.1 Překlad knihovny

Knihovna je distribuována formou zdrojových kódů, je potřeba její překlad. Distribuce obsahuje i soubor `Makefile` pro kompilaci statických knihoven.

Překlad se provede zavoláním příkazu

```
make dg_orcl|dg_gdbm|dg_bkdb
```

v kořenovém adresáři s distribucí. Je třeba zvolit jeden ze tří cílů, dle toho, kterou persistenční vrstvu chceme využívat. Příkaz vytvoří v adresáři `lib` jeden ze souborů `dg_orcl.a`, `dg_gdbm.a` a `dg_bkdb.a` dle zvolené persistenční vrstvy.

V případě výběru cíle `dg_orcl` je před samtným spuštěním překladu potřeba nastavit v souboru Makefile cestu k preprocesoru Pro*C.

Berkeley DB a gdbm lze také používat v módu, kdy se změny dat cacheují v paměti, této funkcionality však není v knihovně standardně využíváno. Pokud bychom chtěli cacheování využít, provedeme překlad

```
make dg_gdbm_cache | dg_bkdb_cache
```

5.1.2 Překlad aplikace

K samotné aplikaci je poté potřeba slinkovat jeden ze souborů `dg_orcl.a`, `dg_gdbm.a` a `dg_bkdb.a` dle zvolené persistenční vrstvy a knihovny konkrétních úložišť

Dále je potřeba říci překladači, kde má hledat hlavičkové soubory knihovny. Ty jsou uloženy v adresáři `headers` adresáře s distribucí.

Samotný překlad jednoduchého programu `simple.c` potom může vypadat následovně:

```
gcc  -Iheaders lib/dg_gdbm.a simple.c -lgdbm -o runme
```

Nebo s použitím databáze Oracle a naistalovaným Oracle klientem ve verzi 11:

```
gcc  -Iheaders lib/dg_gdbm.a simple.c -lclntsh \
  -lsql11 -lnnz11 -o runme
```

5.1.3 Inicializace persistentního úložiště

Každá aplikace využívající knihovnu musí direktivou `include` vkládat hlavičkový soubor `dg_peristence.h`. Ta deklaruje funkci pro inicializaci persistentního úložiště.

```
void * pl_init(char* connection_string)
```

Jejím parametrem je řetězec, který závisí na použité persistenční vrstvě. Vrací handler datového úložiště.

Před ukončením aplikace je potřeba tento handler zrušit voláním funkce

```
void  pl_close(void * db);
```

5.1.4 Persistence prostřednictvím databáze Oracle

Pro Oracle persistenční vrstvu je parametrem funkce `pl_init()` přihlašovací řetězec k databázi. Při správně nastavené proměnné prostředí `SSID` postačuje řetězec ve formátu jméno/heslo, případně jméno/heslo@databáze. Databázové spojení k databázi `pes` pod uživatelem `jan` s heslem `honza` vytvoříme voláním:

```
pl_init("jan/honza@pes");
```

Vrstva vyžaduje, aby v databázi existovala příslušná tabulka a sekvence. Skript pro vytvoření je součástí distribuce knihovny. Pokud bude připojení aplikace k databázi provedeno pod uživatelem, který není vlastníkem schématu s těmito objekty, je třeba přidělit mu práva k zápisu do tabulky `schema.node` a vytvořit pro tuto tabulku synonymum `node`.

5.1.5 GDBM a Berkeley DB persistenční vrstvy

Funkce `void * pl_init(char* connection_string)` pro Berkeley DB persistenční vrstvu a GDBM persistenční vrstvu vyžaduje jako vstupní řetězec cestu k souboru kde je uložena databáze. Pokud soubor neexistuje, je databáze vytvořena a jsou do ní vloženy čítače identifikátorů uzlů a struktur.

Jako příklad uvedeme inicializaci databáze v souboru `/home/jan/database.db` a její následné uzavření.

```
void * db = pl_init("/home/jan/database.db");
pl_close(db);
```

5.2 Červeno-černý strom

Definice červeno-černého stromu[5] a deklarace funkcí pro operace na tomto stromě jsou v hlavičkovém souboru `node.h`, který je potřeba vložit direktivou `include` .

5.2.1 Vytvoření stromu

Z uživatelského hlediska je strom reprezentován strukturou `rbtree` , která je definována v hlavičkovém souboru `rbtree.h`. Její inicializace se provede funkcí

```
int rb_tree_init(
  rbtree * t,
  unsigned long int tree_id,
void * db,
  int(*compare)(const void *, const void *)
)
```

Parametr `t` je ukazatel na strukturu, kterou chceme inicializovat. Parametr `tree_id` je identifikátor struktury v persistentním úložišti. Pokud je rovný nule, vytvoří se nová struktura. Parametr `db` je ukazatel na handler datového úložiště a `compare` ukazatel na funkci, která porovnává data uzlu. Ta musí v případě, že data, na která ukazuje první parametr jsou menší než data, na která ukazuje druhý parametr, vracet záporné číslo. V případě rovnosti dat musí vracet nulu, jinak vrací kladné číslo.

Funkce `rb_tree_init` v případě vytváření nového stromu vrací nulu, pokud se nový strom podaří vytvořit. Jinak vrací 1. V případě načítání existujícího stromu - nenulového `tree_id` vrací nulu v případě, že strom s daným id existuje, jinak vrací 1.

5.2.2 Funkce rb_get_tree_id()

Pro získání identifikátoru stromu, aby mohl být později načten z datového úložiště, slouží funkce

```
unsigned long int get_tree_id(rbtree *t);
```

Jako parametr dostane inicializovaný strom a vrátí jeho identifikátor.

5.2.3 Funkce rb_finsh()

Po ukončení práce se stromem je na tento strom potřeba zavolat funkci

```
rb_finish(rbtree *t)
```

která uvolní data alokovaná při inicializaci a pokud je strom neprázdný, uvolní jeho uzly z paměti. Strukturu `rbtree`, na kterou ukazoval parametr `t` je před dalším použitím nutné znovu inicializovat voláním `rb_tree_init()`

5.2.4 Příklad užití funkcí rb_tree_init, rb_get_tree_id() a rb_finsh()

Následující příklad ukazuje vytvoření stromu, do kterého budou vkládány hodnoty typu `int`, které budou standardně porovnávány. Následně uvolní veškerá data stromu z paměti.

```
#include "stdlib.h"
#include "dg_persistence.h"
#include "rbtree.h"
#define CONN_STR "database.db"
int main(int argc, char ** argv){
  /* definice porovnávací funkce */
  int cmp(const void * a, const void * b){
    if( *(int*)a  < *(int*)b ){ return -1;}
    if( *(int*)a  > *(int*)b ){ return  1;}
    return 0;
  }
  /*inicializace úložiště*/
  void * DB;
  DB = pl_init(CONN_STR);
  unsigned long int id;
  rbtree t;
  /*incializace nového stromu*/
  if(rb_tree_init(&t,0,DB,cmp)) return -1;
  /*uložení id stromu pro načtení v dalším běhu aplikace*/
  id = get_tree_id(&t);
  rb_finsh(&t);
  return 0;
}
```

5.2.5 Funkce rb_insert()

Pro vložení hodnoty do stromu složí funkce

```
int rb_insert(
  rbtree* t,
  void* data,
  size_t data_size
)
```

která jako parametry dostane ukazatel na inicializovaný strom, ukazatel na data a velikost těchto dat. Přepokládá dynamicky alokovaná data. Vrací nulu v případě vložení hodnoty. Pokud dle porovnávací funkce shodná hodnota již ve stromu existuje, vrací nenulové číslo.

V následujícím příkladu je ukázáno, jak vložit do inicializovaného stromu `strom` hodnotu 8

```
int * hodnota = malloc(sizeof(int));
*hodnota = 8;
rb_insert(strom, hodnota, sizeof(int);
```

5.2.6 Funkce rb_find()

Uzel s konkrétní hodnotou se vyhledá zavoláním funkce

```
node * rb_find(
  rbtree *t,
  void * data
)
```

Prvním parametrem je ukazatel na inicializovaný strom, druhým ukazatel na data, která mají být dle porovnávací funkce `compare`, definované při inicializaci stromu, rovny datům v uzlu. Pokud strom data neobsahuje, vrací `NULL`.

5.2.7 Funkce rb_delete()

Vyhledaný uzel je z úložiště i paměti vymazán voláním

```
void rb_delete(
  rbtree * t,
  node * n
)
```

kde `t` je ukazatel na inicializovaný strom a `n` ukazatel na mazaný uzel.

5.2.8 Funkce rb_delete_key()

Dále je možné vymazat uzel, který se vyhledá dle zadaných dat voláním funkce

```
void rb_delete_key(
  rbtree* t,
  void *data
)
```

Jako parametry bere ukazatel na inicializovaný strom a ukazatel na data, která se dle porovnávací funkce rovnají hodnotě uzlu, který má být smazán.

5.2.9 Příklad užití funkcí rb_find(), rb_delete() a rb_delete$_key()$

Oba následují příklady jsou funkčně ekvivalentní. Vymažou z inicializovaného stromu `strom` uzel s hodnotou 8, pokud existuje.

```
int key = 8;
/* 1.zpusob */
rb_delete_key(strom, &key);
/* 2. zpusob*/
int key = 8;
node nalezeny = rb_find(strom, &key)
if(nalezeny)
   rb_delete(nalezeny);
```

5.2.10 Funkce rb_delete_tree()

Všechny uzly stromu lze vymazat funkcí

```
void rb_delete_tree(rbtree * t);
```

Jako parametr je jí předán ukazatel na inicializovaný strom. Prázdný strom však stále existuje v paměti i v datovém úložišti.

5.2.11 Funkce rb_destroy()

Pro vymazání stromu a zrušení struktury `rbtree` slouží funkce

```
void rb_destroy(rbtree * t)
```

které je předán ukazatel na inicializovaný strom. Takový strom je poté kompletně zrušen jak v paměti tak v datovém úložišti. Po zavolaní této funkce již není nutné volat `rb_finish()` Strukturu `rbtree`, na kterou ukazoval parametr `t` je před dalším použitím nutné znovu inicializovat voláním `rb_tree_init()`

5.2.12 Funkce rb_free_tree()

Všechny uzly stromu jsou uvolněny z paměti voláním funkce

```
void rb_free_tree(rbtree *t)
```

které je předán ukazatel na inicializovaný strom.

5.2.13 Funkce rb_restore_tree()

Na strom, který byl právě inicializován funkcí `rb_tree_init()` nebo uvolněn funkcí `rb_free_tree()` lze zavolat funkci

```
void rb_restore_tree(rbtree *t)
```

která načte celý strom do paměti.

Jako parametr dostane ukazatel na inicializovaný strom. Tento strom je poté před dalším použitím nutné znovu inicializovat.

5.2.14 Příklad užití funkcí rb_delete_tree(), rb_free_tree(), rb_restore_tree() a rb_destroy()

```
#include "stdlib.h"
#include "dg_persistence.h"
#include "rbtree.h"
#define CONN_STR "database.db"
int main(int argc, char ** argv){
  /* definice porovnávací funkce */
  int cmp(const void * a, const void * b){
    if( *(int*)a  < *(int*)b ){ return -1;}
    if( *(int*)a  > *(int*)b ){ return  1;}
    return 0;
  }
  /*incializace úložiště*/
  void * DB;
  DB = pl_init(CONN_STR);
  unsigned long int id;
  rbtree t;
  /*inicializace nového stromu*/
  if(rb_tree_init(&t,0,DB,cmp)) return -1;
  /*uložení id stromu pro načtení v dalším běhu aplikace*/
  id = get_tree_id(&t);
  /* zde vložíme nějaká data funcki rb_insert() */
    .
    .
  /* uvolníme data z paměti*/
  rb_free_tree(&t);
  /* opět je načteme do paměti */
  rb_resotore_tree(&t);
  /* a vymažeme */
  rb_delete_tree(&t);
  /* ukončíme práci se stromem */
  rb_finish(&t);
  /* znovu inicializujeme */
  /* použijeme existující strom */
  rb_tree_init(&t,id,DB,cmp)
  /* načteme uzly stromu do paměti */
  rb_resotore_tree(&t);
  /* a všechna data stromu navždy vymažeme */
  rb_destroy(&t);
  /* tady již */
  /* rb_tree_init(&t,id,DB,cmp) != 0 */
  return 0;
}
```

5.3 Avl strom

Z uživatelského hlediska je AVL strom[4] reprzentován strukturou `avltree` , která je definovana v hlavičkovém souboru `avltree.h`.

Rozhraní k operacím na stromě je shodné s červeno-černým stromem, jediný rozdíl je, že prefix `no_` je nahrazen prefixem `avl_` a tam kde funkce pro červeno-černý strom berou jako parametr ukazatel na strukturu `rbtree`, berou funkce pro AVL strom ukazatel na strukturu `avltree`.

5.4 B-strom

Z uživatelského hlediska je B-strom reprezentován strukturou `btree` , která je definovaná v hlavičkovém souboru `btree.h`.

Rozhraní k operacím na stromě je shodné s červeno-černým stromem s rozdílem, že prefix `no_` je nahrazen prefixem `bt_` a tam kde funkce pro červeno-černý strom berou jako parametr ukazatel na strukturu `rbtree`, berou funkce pro B-strom ukazatel na strukturu `btree`.

Dále je rozdílná funkce pro inicializaci stromu

```
int bt_tree_init(
 btree * t,
 unsigned long int tree_id,
 unsigned int order,
 void * db,
 int(*compare)(const void *, const void*)
)
```

která má navíc parametr `order`, který udává řád stromu, neboli maximální počet počet potomků uzlů. Musí mít hodnotu minimálně 3.

Posledním rozdílem je funkce pro vyhledání hodnoty

```
n_datum bt_find(
  btree *t,
  void * data
)
```

která místo ukazatele na uzel s hodnotou vrací ve struktuře `n_datum` ukazatel na nalezená data ve stromě a jejich velikost. V případě, že strom hodnotu neobsahuje, bude ukazatel i velikost dat 0.

6. Testování

Pro testování dopadu zavedení persistence datových struktur na dopad časové náročnosti aplikace byly vytvořeny celkem tři sady testů:

- První test vložil do datové struktury 1000 náhodných hodnot z intervalu 0-800 s tím, že po každém stém vložení byla struktura uvolněna z paměti. Následně bylo 1000x vyhledána náhodná hodnota z intervalu 0-800. Po každém stém vyhledání byla struktura uvolněna z paměti.
- V druhém testu se opět vložilo 1000 náhodných hodnot z intervalu 0-800 s tím, že struktura se z paměti uvolnila až po vložení všech hodnot. Následně byla 100 000x vyhledána náhodná hodnota z intervalu 0-800.
- V posledním testu bylo vloženo 1000 náhodných hodnot z intervalu 0-800 bez uvolňování struktury z paměti a následně byla hodnota z tohoto intervalu vyhledána 10 000 000x

Tyto testy byly spuštěny desetkrát pro každou strukturu a každou persistenční vrstvu. Jako referenční byla využita „nepersistentní“ persistenční vrstva, která všechny data drží výhradně v paměti. Pro B-strom byly testy prováděny pro řád 3, 10 a 50.

Byla měřena celková doba běhu programu pomocí nástroje `time`.

Testy byly provedeny na běžném notebooku s procesorem Core2 Duo T7100 na frekvenci 1,8 GHz, s 2 MB L2 cache a 800 MHz FSB. Testovací notebook byl vybaven 3GB paměti, která během každého testu postačovala a operační systém nikdy nemusel stránkovat paměť na disk.

Všechny testy probíhaly v distribuci OpenSuSE s jádrem GNU Linux verze 2.6.31.12. Zdrojové kódy byly překládány kompilátorem gcc verze 4.4.1.

Testy jsou uložené na přiloženém CD v adresáři `tests` jako `testavl1.c`, `testrb1.c`, `testbt1.c`, `testavl2.c`, `testrb2.c`, `testbt2.c`, `testavl3.c`, `testrb3.c` a `testbt3.c`, kde číslo v názvu souboru označuje pořadí testu.

Testy lze spustit z kořenového adresáře distribuce příkazem `make run_tests` Před voláním tohoto příkazu musí být ze zdrojových souborů spouštěného textu odstraněno pořadové číslo testu a v souboru Makefile nastavena proměnná `DBNAME` značící cestu k souboru, kam budou uloženy GDBM a BerkeleyDB databáze. Dále je na témže místě třeba nastavit proměnnou `ORCLCN` na přihlašovací řetězec k databázi Oracle.

6.1 Výsledky testů

Výsledky testů shrnují následující tabulky a graf. Jako výsledek je uveden aritmetický průměr doby běhu deseti opakování testu. Doba běhu je uvedena v milisekundách. Sloupce jsou označeny zkratkou datové struktury - avl pro AVL strom, rb pro červeno-černý strom, bt3 pro B-strom řádu 3, bt10 pro B-strom řádu 10 a bt50 pro B-strom řádu 50. Řádky jou označeny zkratkou persistenční vrstvy - none pro nepersistentní persistenční vrstvu, bkdb pro vrstvu využívající Berkeley DB, gdbm pro vrstvu využívající GDBM a orcl pro vrstvu, která ukládá data do databáze Oracle.

Pro každý test jsou uvedeny dvě tabulky. V první jsou uvedeny absolutní doby běhu a v druhé je uvedený poměr zpomalení oproti nepersistentní implementaci. Dále je pro každý test uveden graf s dobami běhů.

Test č.1	avl	rb	bt3	bt10	bt50
none	3,3	7,1	10,5	9	16,4
bkdb	26424,4	7292,3	4599,5	1992,4	923,5
gdbm	10241,7	11627,3	5945,3	9007	5010,5
orcl	35553,5	7935,8	6384	3161,8	1797,3

Tabulka 6.1: Absolutní výsledky prvního testu

Test č.1	avl	rb	bt3	bt10	bt50
none	1	1	1	1	1
bkdb	8007,39	1027,08	438,05	221,38	56,31
gdbm	3103,55	1637,65	566,22	1000,78	305,52
orcl	10773,79	1117,72	608	351,31	109,59

Tabulka 6.2: Relativní výsledky prvního testu

Test č.2	avl	rb	bt3	bt10	bt50
none	45	13,3	67,1	59,9	86,8
bkdb	26618,8	7220,2	4623,9	1919,6	960,1
gdbm	10241,9	11752,4	5941,7	9075,3	4877,3
orcl	35335,9	7699,2	6540,7	3119,2	1814,5

Tabulka 6.3: Absolutní výsledky druhého testu

Test č.2	avl	rb	bt3	bt10	bt50
none	1	1	1	1	1
bkdb	591,53	542,87	68,91	32,05	11,06
gdbm	227,6	883,64	88,55	151,51	56,19
orcl	785,24	578,89	97,48	52,07	20,9

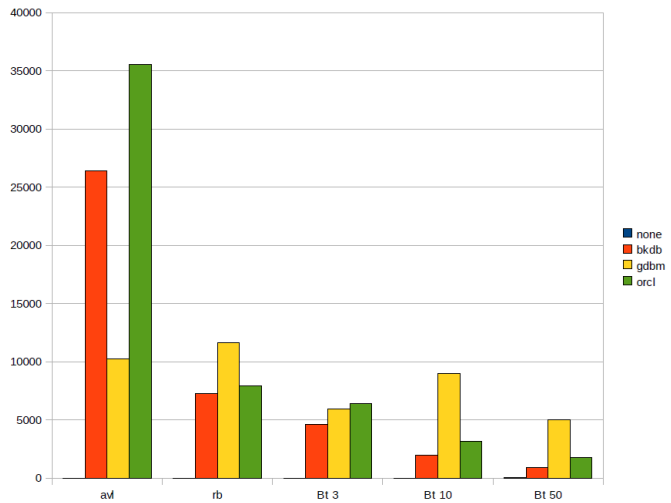
Tabulka 6.4: Relativní výsledky druhého testu

Test č.3	avl	rb	bt3	bt10	bt50
none	3892,1	4007,4	5396,6	4839,6	6586,8
bkdb	30413,9	11240,1	10348,2	6748,5	7457,6
gdbm	14238,3	16335,6	11707,7	14100,8	11673,6
orcl	38809,1	11524,7	11924,1	7944,4	8814,8

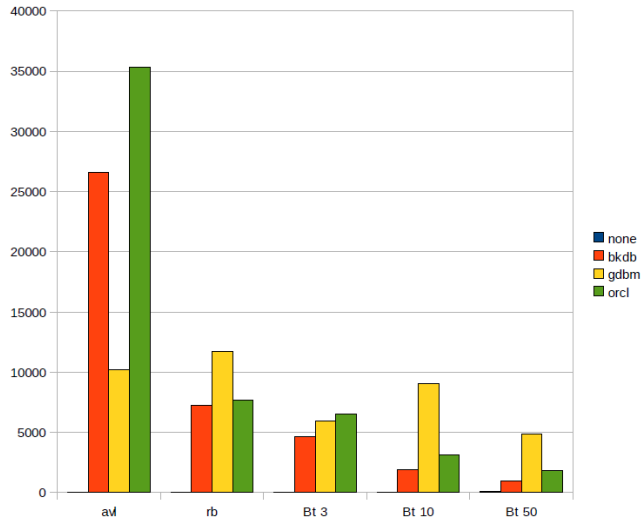
Tabulka 6.5: Absolutní výsledky třetího testu

Test č.3	avl	rb	bt3	bt10	bt50
none	1	1	1	1	1
bkdb	7,81	2,8	1,92	1,39	1,13
gdbm	3,66	4,08	2,17	2,91	1,77
orcl	9,97	2,88	2,21	1,64	1,34

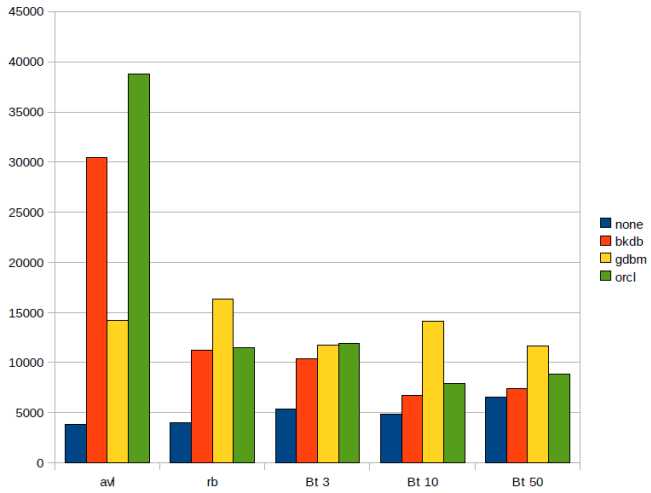
Tabulka 6.6: Relativní výsledky třetího testu



Obrázek 6.1: Výsledky prvního testu



Obrázek 6.2: Výsledky druhého testu



Obrázek 6.3: Výsledky třetího testu

6.2 Shrnutí výsledků

Výsledky prvního testu ukazují, že pokud je potřeba datové struktury nebo jejich části často obnovovat z persistentního úložiště, nebo je do persistentního úložiště ukládat je zpomalení oproti nepersistentní implementaci značné. Naopak druhý a třetí test ukazuje, že pokud je již potřebná část struktury v paměti a převažuje čtení, začne se časová náročnost přibližovat nepersistentní implementaci.

Jako nejvýhodnější z implementovaných struktur se z hlediska časové náročnosti ukazuje B-strom s velkým řádem. Jeho nevýhodou je však vyšší paměťová náročnost, protože se vždy načítají všechna data uzlu, byť nemusí být aktuálně nezbytné.

7. Závěr

Podařilo se vytvořit knihovnu, která umožňuje vytváření stromových struktur, které jsou persistentně ukládány do datového úložiště a následně jsou z něj načítány. Ukládání a načítání je naprosto transparentní a nezatěžuje uživatele knihovny potřebnou znalostí implementace.

Modulární návrh umožňuje snadnou změnu persistentního úložiště a přidání podpory pro další úložiště. Zároveň umožňuje snadné vytváření nových persistentních datových struktur bez znalosti implementace persistenční vrstvy.

Zavedení persistence s sebou však přináší i zvýšení časové náročnosti. Knihovna může být použita pro všechny typy aplikací, které potřebují datové struktury i mezi jednotlivými běhy aplikace, ale z důvodu zvýšené časové náročnosti se však příliš nehodí například pro interaktivní aplikace, ve kterých se datové struktury často mění. V takovém případě je lepší udržovat změny pouze v paměti a celou strukturu uložit až při ukončovaní aplikace, protože uživateli bude pravděpodobně méně vadit čekání na ukončení aplikace než zpomalení během práce s aplikací. Některá datová úložiště, například Berkeley DB, toto chování sice umí simulovat cacheováním. Protože to však není funkcionalita poskytovaná všemi typy úložišť, je knihovna standardně kompilována bez podpory cacheování.

Knihovna je naopak vhodná pro aplikace, kde se datové struktury vejdou do paměti a převažuje čtení těchto struktur nad jejich změnami. Dále je vhodná pro neinteraktivní aplikace, které pracují s velkými daty, která se nemusí vejít do paměti a nevadí jim průběžné zpomalení způsobené přístupem do persistentního úložiště. Takovým příkladem může být aplikace pro periodické ukládání výsledků fyzikálních měření.

Literatura

[1] Oracle: *Oracle Documentation*
http://www.oracle.com/technetwork/indexes/documentation/index.html

[2] Pierre Gaumond et al.: *GNU gdm Documentation*
http://www.gnu.org.ua/software/gdbm/manual/gdbm.html

[3] Oracle: *Oracle Oracle Berkeley DB 11g Release 2 Documentation*
http://download.oracle.com/docs/cd/E17076_02/html/toc.htm

[4] Přispěvatelé Wikipedie: *AVL tree*
http://en.wikipedia.org/wiki/AVL_tree

[5] Přispěvatelé Wikipedie: *Red–black tree*
http://en.wikipedia.org/wiki/Red-black_tree

[6] Přispěvatelé Wikipedie: *B-tree*
http://en.wikipedia.org/wiki/B-tree

Seznam příloh

Na CD, které je přiloženo k této práci, se nachází:

- Text této práce
- archiv perslib.gz, který obsahuje adresář `dataguard` s distribucí zdrojových kódů knihovny.