Praktické aspekty vývoje: Projekt 1 – Testování

Vysoké Učení Technické v Brně,
Fakulta Informačních Technologií

Filip Vaverka
ivaverka@fit.vutbr.cz

David Grochol
igrochol@fit.vutbr.cz

4. února 2017

1 Úvod

Cílem tohoto projektu je si prakticky vyzkoušet typické úlohy spojené s testováním software pomocí tzv. unit testů. Projekt je rozdělen do tří částí, jejichž účelem je otestovat neznámý kód (black box testy), otestovat známý kód (white box testy), ověřit pokrytí kódu testy (code coverage) a implementovat kód na základě testů (test driven development). Za každou z těchto úloh je možné získat až **6 bodů**, za celý projekt lze tedy získat až **18 bodů**.

Použité nástroje

Jako implementační jazyk byl zvolen `C/C++`, (základní znalost by měla být dostatečná). Jako testovací framework je využívána kombinace:

- `GoogleTest` (https://github.com/google/googletest/blob/master/googletest/docs/Primer.md)
- `CMake/CTest` (https://cmake.org/documentation/)
- `GCOV` (https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)

Hodnocení

Pro hodnocení bude použit server **ivs.fit.vutbr.cz** a vaše řešení na tomto serveru tedy **závazně musí** fungovat. Tento server také můžete použít pro vyhodnocení pokrytí kódu (viz dále).

2 Struktura projektu

- Systém překladu (CMake):
  Systém pro vygenerování skriptů pro překlad je tvořen soubory:
  "`CMakeLists.txt`, `CMakeLists.txt.in`, `CodeCoverage.cmake`, `GoogleTest.cmake`"
  a nevyžaduje žádné úpravy.
- Úloha black-box testing:
  Zdrojový soubor "`black_box_tests.cpp`", do kterého je třeba doplnit příslušné testy a knihovna "`black_box_lib.a/.lib`" s hlavičkovým souborem "`red_black_tree.h`", která obsahuje testovaný kód binárního stromu.
- Úloha white-box testing:
  Zdrojové soubory: "`white_box_code.h`" a "`white_box_code.cpp`", které obsahují testovaný kód a "`white_box_tests.cpp`", do kterého je třeba doplnit příslušné testy.
- Úloha test driven development:
  Zdrojový soubor: "`tdd_code.h`" obsahuje definici rozhraní prioritní fronty, kterou je třeba implementovat v "`tdd_code.cpp`". Soubor "`tdd_test.cpp`" obsahuje testy prioritní fronty, které musí vaše řešení splnit.

- **Odevzdávané soubory:**
  Odevzdejte `POUZE` následující soubory zabalené do `*.zip` archivu (lze provést přímo pomocí CMake – viz překlad a spuštění).
  - `black_box_tests.cpp` – Testy Red-Black Tree
  - `white_box_tests.cpp` – Testy maticových operací
  - `tdd_code.h`, `tdd_code.cpp` – Test Driven Development

3 Prerekvizity a překlad

Projekt je primárně určen pro platformu Linux/GCC, avšak lze jej přeložit a spustit i na Windows (Visual Studio/MSVC). Pod Windows však může být problém použít nástroje pro analýzu pokrytí kódu, které jsou založené na GCC.

Windows

Prerekvizity

- Microsoft Visual Studio 2015
- CMake 2.8.2+ (https://cmake.org)
- GoogleTest (https://github.com/google/googletest/archive/master.zip)
- Případně GIT client pro Windows (https://git-for-windows.github.io/)
  Pak si CMake skripty umí GoogleTest stáhnout automaticky z repozitáře.

Překlad a spuštění

1. Soubor se zadáním projektu rozbalte do libovolného adresáře.
2. Do libovolného adresáře rozbalte CMake (https://cmake.org/files/v3.7/cmake-3.7.2-win64-x64.zip), nebo nainstalujte (https://cmake.org/files/v3.7/cmake-3.7.2-win64-x64.msi)
3. Spusťte konzoli (zkratka Win+R a "cmd") a přesuňte se do adresáře "assignment/build". Pokud používáte "zip verzi" CMake je třeba aktualizovat proměnnou `PATH`, tak aby obsahovala cestu k "cmake.exe". To lze nastavit příkazem

   ```
   > set PATH=%PATH%;cesta\k\cmake.exe;
   ```

   Toto nastavení je dočasné (do zavření okna konzole).
4. Pokud není dostupný nástroj GIT, je nutné v aktuálním adresáři rozbalit archiv `googletest-master.zip`, tak aby jeho obsah byl v adresáři: `assignment/build/googletest-master`.
5. Soubory pro překlad je možné vygenerovat příkazem:

   ```
   > cmake ..
   ```

který vygeneruje Solution (případně stáhne GoogleTest) a měl by skončit výstupem: *"Build files have been written to: ..."*. **Tento krok je nutné opakovat po každém přidání nového testu!** Pokud není CMake nalezeno je zřejmě špatně nastavená proměnná `PATH`, nebo je třeba po jeho instalaci restartovat systém.

6. Tím by měl v aktuálním adresáři vzniknout projektový soubor pro Visual Studio: "ivs_proj_1.sln", který můžete otevřít a použít pro vypracování úloh.

7. Překlad je možné provést z konzole příkazem

   ```
   > cmake --build .
   ```

   nebo ve Visual Studiu.

8. Visual Studio by mělo být schopné spouštět jednotlivé testy přímo z textového editoru (tlačítko na řádku s testem v postranní liště vlevo). Jinak je možné testy spouštět z konzole přes CMake pomocí

   ```
   > ctest -C Debug
   ```

9. Odevzdávaný archiv je možné vytvořit příkazem

   ```
   > cmake --build . --target pack
   ```

   (stále v adresáři "assignment/build"), který vytvoří archiv "xlogin00.zip". Archiv zkontrolujte a přejmenujte dle svého loginu. **Vytvoření archivu je možné pouze s CMake 3.2 a novějším!**

Linux

Prerekvizity

- GCC 4.9.0+ (mělo by být dostupné v repozitářích pro danou distribuci)
- GCOV (mělo by být dostupné v repozitářích pro danou distribuci)
- LCOV (http://downloads.sourceforge.net/ltp/lcov-1.13.tar.gz, nebo v repozitáři)
- CMake 2.8.2+ (mělo by být dostupné v repozitářích pro danou distribuci)
- GoogleTest (https://github.com/google/googletest/archive/master.zip)
- Případně GIT client (opět by měl být v repozitářích)
  Pak si CMake skripty umí GoogleTest stáhnout automaticky z repozitáře.

Překlad a spuštění

1. Soubor se zadáním projektu rozbalte do libovolného adresáře (například příkazem:

   ```
   $ unzip ivs_project_1.zip
   ```

který archiv rozbalí v aktuálním adresáři).

2. Pokud nepoužíváte systémovou instalaci LCOV (například na serveru merlin) je nutné rozbalit staženou verzi do adresáře "`./assignment`" např. pomocí:

```
$ tar -xvf lcov-1.13.tar.gz
$ mv lcov-1.13 lcov
```

3. Přesuňte se do adresáře: `./assignment/build`, kde budou vytvořeny soubory překladu, výsledné spustitelné soubory a výstupy nástrojů pro analýzu pokrytí kódu (GCOV/LCOV).

4. Pokud není dostupný nástroj GIT, je nutné v aktuálním adresáři rozbalit archiv `googletest-master.zip`, tak aby jeho obsah byl v adresáři: `./assignment/build/googletest-master`.

5. Soubory pro překlad je možné vygenerovat příkazem:

```
$ cmake ..
```

   který vygeneruje Makefile (případně stáhne GoogleTest) a měl by skončit výstupem: *"Build files have been written to: ..."*. **Tento krok je nutné opakovat po každém přidání nového testu!**

6. Nyní by mělo být možné přeložit projekt příkazem:

```
$ make
```

   nebo

```
$ cmake --build .
```

   Tento krok je nutné opakovat při každé změně zdrojového kódu!

7. Testy je nyní možné zpustit příkazem:

```
$ ctest -C Debug
```

   nebo každou část projektu odděleně přímo pomocí příslušných spustitelných souborů: "`./black_box_test`" pro "black-box testy", "`./white_box_test`" pro "white-box testy" a "`./tdd_test`" pro úlohu "test driven development".

8. Odevzdávaný archiv je možné vytvořit příkazem

```
$ cmake --build . --target pack
```

   (stále v adresáři "./assignment/build"), který vytvoří archiv "xlogin00.zip". Archiv zkontrolujte a přejmenujte dle svého loginu. **Vytvoření archivu je možné pouze s CMake 3.2 a novější!**

Ověření pokrytí kódu

Za předpokladu, že překlad a spuštění projektu proběhlo bez problémů, je nyní možné přistoupit k ověření pokrytí kódu testy pomocí nástrojů GCOV a LCOV. Pro tento účel vytváří CMake cíle "`white_box_test_coverage`" a "`tdd_test_coverage`". Analýzu pokrytí kódu je pak možné provést příkazem

```
$ make white_box_test_coverage
```

nebo

```
$ make tdd_test_coverage
```

Výsledky analýzy by měly být v adresářích "`white_box_test_coverage`", nebo "`tdd_test_coverage`" ve formátu HTML a lze je zobrazit pomocí `index.html` v příslušném adresáři.

4 Úlohy

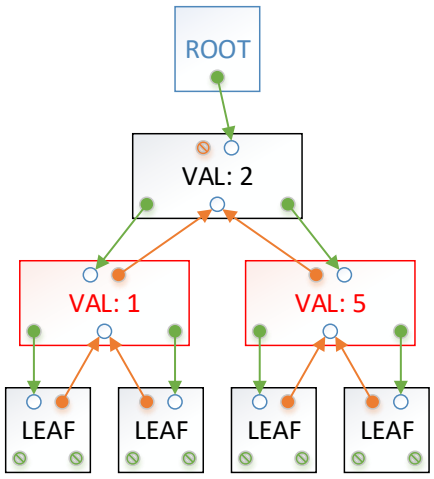
V rámci projektu je třeba vypracovat tři úlohy zaměřené na testování: testování neznámého kódu (tzv. black box), testování známého kódu (tzv. white box) a implementace dle testů (tzv. test driven development).

4.1 Black Box Testing (až 6 b.)

Cílem této úlohy je otestování základních operací nad neznámou implementací Red-Black tree (viz https://en.wikipedia.org/wiki/Red-black_tree) na základě jejich definice.

Red-Black Tree

Red-Black Tree je stromová struktura (binární strom), která umožňuje provedení základních operací (vkládání, mazání a vyhledávání) v logaritmickém čase $\mathcal{O}(\log n)$. Struktura stromu je tvořena kolekcí uzlů, které jsou propojeny tak, že každý má maximálně dva potomky (levý a pravý). Každý uzel pak tvoří další podstrom (který může být prázdný). Každý strom má pak právě jeden uzel, který nemá žádného rodiče a nazývá se "kořen/root" (v obrázku 1 má hodnotu 2).



Obrázek 1: Red-Black tree po vložení prvků 2,1 a 5.

Implementujte testy

Vaším úkolem je implementovat testy základních operací nad Red-Black Tree (Insert, Find a Delete) a trojici základních axiomů které v této struktuře musí platit. Rozhraní stromu je definováno v souboru: "red_black_tree.h" (třída "BinaryTree"). Vaše testy a inicializace testů doplňte do souboru: "black_box_tests.cpp", který je součástí odevzdávaného řešení.

Třída binárního stromu definuje následující veřejné rozhraní:

- `BinaryTree::BinaryTree()` – konstruktor
  Vytvoří prázdný strom ("m_pRoot" má hodnotu `NULL`).
- `BinaryTree::~BinaryTree()` – destruktor
  Odstraní všechny prvky stromu.
- `std::pair<bool, BinaryTree::Node_t *> BinaryTree::InsertNode(int key)` – vložit nový prvek
  Pokusí se do stromu vložit nový prvek s hodnotou "key" a vrací dvojici "(bool, pointer)". Tato dvojice bude obsahovat hodnotu "true" a ukazatel na vložený prvek, nebo "false" a ukazatel na již existující prvek (v případě, že strom již prvek s hodnotou "key" obsahoval).

  K prvku objektu "std::pair<bool, void *ptr> x(b, p);" je možné přistupovat pomocí: "x.first" – vrátí bool hodnotu "b" a "x.second" – vrátí ukazatel "p".
- `bool BinaryTree::DeleteNode(int key)` – odstranit prvek
  Pokusí se odstranit prvek s hodnotou "key", pokud takový nalezne a odstraní vrací "true", jinak "false".
- `BinaryTree::Node_t *BinaryTree::FindNode(int key) const` – vyhledat prvek
  Pokusí se nalézt prvek s hodnotou "key" a vrátí ukazatel na tento prvek. Pokud takový prvek není nalezen, vrací hodnotu `NULL`.
- `void BinaryTree::GetLeafNodes(std::vector<BinaryTree::Node_t *> &outLeafNodes)` – získat listové uzly
  Vyplní pole "outLeafNodes" ukazateli na listové uzly (tedy bez potomků) ve stromu. Pole je nejdříve vyprázdněno.
- `void BinaryTree::GetAllNodes(std::vector<BinaryTree::Node_t *> &outAllNodes)` – získat všechny uzly
  Vyplní pole "outAllNodes" ukazateli na všechny uzly ve stromu. Pole je nejdříve vyprázdněno.
- `void BinaryTree::GetNonLeafNodes(std::vector<BinaryTree::Node_t *> &outNonLeafNodes)` – získat NE-listové uzly
  Vyplní pole "outNonLeafNodes" ukazateli na všechny NE-listové uzly ve stromu. Pole je nejdříve vyprázdněno.
- `Node_t *BinaryTree::GetRoot()` – získat ukazatel na kořen stromu
  Vrací ukazatel na kořen stromu, nebo `NULL` je-li strom prázdný.

Základní axiomy Red-Black Tree:

- Všechny listové uzly (tedy uzly bez potomků) jsou "černé".
- Pokud je uzel "červený", pak jsou jeho oba potomci "černé".
- Každá cesta od každého listového uzlu ke kořeni obsahuje stejný počet "černých" uzlů (viz oranžové cesty v obr. 1).

Hodnocení

- `1,5 b` – Testy funkčnosti operací: “InsertNode”, “DeleteNode” a “FindNode” nad prázdným stromem.
- `1,5 b` – Testy funkčnosti operací: “InsertNode”, “DeleteNode” a “FindNode” nad NE-prázdným stromem.
- `3 b` – Testy 3 základních axiomů Red-Black Tree (viz 4.1).

4.2 White Box Testing (až 6 b.)

Cílem této úlohy je otestování definovaných operací nad maticemi na základě jejich známé implementace.

Maticové operace

Operace s maticemi vychází ze studijní opory předmětu IDA (viz http://www.umat.feec.vutbr.cz/~kovar/webs/personal/MMAT.pdf). Všechny operace implementují standardní chování. Pro řešení soustavy lineárních rovnic je použito Cramerovo pravidlo.

Implementujte testy

Vaším úkolem je implementovat testy základních operací nad maticemi. Rozhraní maticových operací je definováno v souboru: "white_box_code.h" (třída Matrix). Implementace operací je v souboru: "white_box_code.cpp". Vaše testy a inicializace testů doplňte do souboru: "white_box_tests.cpp", který je součástí odevzdaného řešení.

Třída Matrix definuje následující veřejné rozhraní:

- `Matrix::Matrix()` – konstruktor
  Vytvoří nulovou matici velikosti 1x1.
- `Matrix::Matrix(size_t R,size_t C)` – konstruktor
  Vytvoří nulovou matici velikosti R x C.
- `bool Matrix::set(size_t R,size_t C, double value)` – nastaví hodnotu v matici
  Nastaví prvek matice na pozici (R,C) na hodnoty value. Pokud je (R,C) v matici, vrátí metoda true, jinak false.
- `bool Matrix::set(std::vector<std::vector< double > > values)` – nastaví matici
  Nastaví matici hodnotami z pole values. Pokud operace proběhla úspěšně, vrátí true, jinak false.
- `double Matrix::get(size_t R,size_t C)` – vrátí hodnotu v matici
  Vrátí hodnotu prvku na pozici (R,C). Pokud je (R,C) mimo matici, vyvolá výjimku.
- `bool Matrix::operator==(const Matrix m)` – porovná matice
  Porovná matici m s matici definovanou objektem. Pokud jsou shodné, vrátí true, jinak false. Pokud mají matice rozdílnou velikost, vyvolá výjimku.
- `Matrix Matrix::operator+(const Matrix m)` – součet matic
  Přičte matici m k matici definované objektem a vrátí součet matic. Pokud mají matice rozdílnou velikost, vyvolá výjimku.
- `Matrix Matrix::operator*(const Matrix m)` – součin matic
  Vynásobí matici m s matici v objektu a vrátí součin matic. Pokud matice nesplňují podmínky pro násobení, vyvolá výjimku.

- `Matrix Matrix::operator*(const double value)` – násobení matice skalárem
  Vynásobí matici definovanou objektem a hodnotu value a vrátí tento součin.
- `Matrix Matrix::solveEquation(std::vector<double> b)` – řešení soustavy lineárních rovnic
  Metoda řeší soustavu lineárních rovnic pomocí Cramerova pravidla. Parametr b přestavuje pravou stranu rovnice. Pokud lze řešit soustavu, metoda vrátí řešení v pořadí $x_1, x_2, ...$ Jinak vyvolá výjimku.

Hodnocení

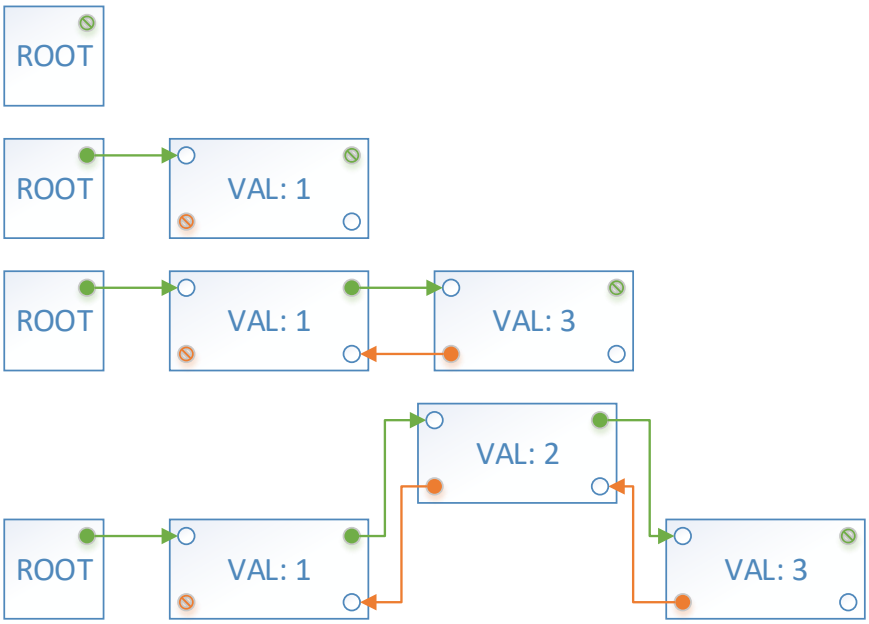
- `až 4 b.` – Rozsah pokrytí kódu pomocí testů.
- `2 b.` – Testy s různými velikostmi a tvary matic.).

4.3 Test Driven Development (až 6b)

Cílem této úlohy je vytvořit jednoduchou implementaci prioritní fronty založenou na obousměrně vázaném seznamu (double-linked list https://en.wikipedia.org/wiki/Doubly_linked_list), která umožní vložit novou položku (na libovolnou pozici) bez přesouvání ostatních položek. Vytvořená implementace pak musí odpovídat specifikaci a musí také splňovat předem definovanou sadu testů.

Obousměrně vázaný seznam je datová struktura používaná pro ukládání dynamického počtu prvků (seznam/list), kde každý prvek je nezávisle dynamicky alokovaný (new/malloc). Aby bylo možné s takovouto strukturou pracovat, každý prvek obsahuje kromě "užitečných dat" také ukazatele na svého předchůdce a následovníka (tedy na prvek, který je v seznamu před ním a za ním). Samotný seznam je pak reprezentován jediným ukazatelem na první prvek seznamu (tzv. kořen/root).

Prioritní fronta je pak z takovéto struktury vytvořena pouze tím, že nový prvek není připojen na konec seznamu, ale je vložen na patřičné místo dle své hodnoty a hodnot prvků, které již ve frontě jsou. Postupné vkládání nových hodnot do prázdné fronty znázorňuje obrázek 2, kde vyplněná kolečka znázorňují ukazatele (oranžový – předchůdce, zelený – následník), přeškrtnuté kolečko má stejný význam jako plné, ale znázorňuje, že daný ukazatel má hodnotu `NULL`.



Obrázek 2: Postupné vložení prvků 1, 3 a 2 do prioritní fronty.

Vlastnosti prioritní fronty

- Prvky ve frontě jsou stále seřazeny podle jejich hodnoty, tak že: $p_{\text{head}} \leq p_1 \leq \ldots \leq p_{n-1} \leq p_{\text{tail}}$.
- Fronta může obsahovat libovolné množství prvků se stejnou hodnotou.

- V případě prázdné fronty má ukazatel na prvek p_{head} (`m_pHead`) hodnotu `NULL`.
- Odkaz prvního prvku na předchůdce a odkaz na následující prvek posledního mají hodnotu `NULL`.

Implementujte

Vaším úkolem je doplnit implementace následujících metod prioritní fronty definované jako třída "PriorityQueue" v souboru "tdd_code.h". Váš kód doplňte do připravených definic metod v souboru: "tdd_code.cpp", který je součástí odevzdávaného řešení.

- `PriorityQueue::PriorityQueue()` – konstruktor
  Zde doplňte inicializaci prázdné prioritní fronty (pokud je nějaká nutná).
- `PriorityQueue::~PriorityQueue()` – destruktor
  Úkolem této metody je odstranění všech prvků fronty (jsou dynamicky alokované) před odstraněním samotné fronty.
- `PriorityQueue::Insert(int value)` – vložit prvek
  Metoda vloží nový prvek s hodnotou "`value`" do fronty na místo dané jeho hodnotou (viz definice prioritní fronty). Je tedy třeba nalézt patřičné místo a prvek zařadit do obousměrně vázaného seznamu tvořícího frontu. Pokud prvek s danou hodnotou "`value`" již existuje, je nový prvek zařazen před nebo za něj.
- `PriorityQueue::Remove(int value)` – odstranit prvek
  Metoda nalezne `PRVNÍ` prvek s danou hodnotou "`value`" a odstraní ho z prioritní fronty (výsledná fronta musí stále obsahovat všechny ostatní prvky ve správném pořadí). Pokud není daný prvek nalezen, vrací metoda hodnotu "`false`", v opačném případě vrátí "`true`".
- `PriorityQueue::Element_t *PriorityQueue::Find(int value)` – vyhledat prvek
  Metoda nalezne `PRVNÍ` prvek s danou hodnotou "`value`" a vrátí ukazatel na tento prvek. Pokud není prvek nalezen, vrací hodnotu `NULL`.
- `PriorityQueue::Element_t *PriorityQueue::GetHead()` – první prvek
  Metoda vždy vrací ukazatel na první (tedy nejmenší) prvek ve frontě (případně hodnotu `NULL`, pokud je fronta prázdná).

Vaše implementace těchto metod musí splnit výše uvedené definice a projít sadou testů (viz soubor "`tdd_tests.cpp`").

Hodnocení

- `1 b.` – Splnění testů nad prázdnou frontou.
- `2 b.` – Splnění testů nad neprázdnou frontou (4 testy po 0.5b).
- `2 b.` – Implementace metod "Insert", "Find", "Remove" a destruktoru (4 metody po 0.5b).
- `1 b.` – Správnost implementace.