## FIIT STU

## Dátové štruktúry a algoritmy

## Zadanie 3 – Binárne rozhodovacie diagramy

## Adam Miškove

## ID: 103056

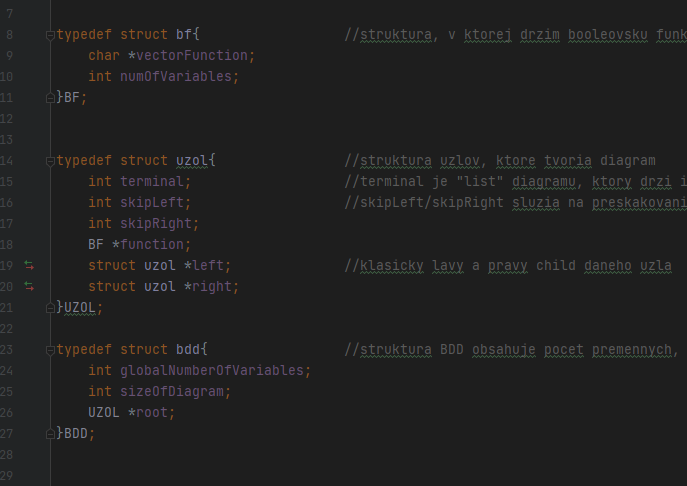
## 2020/2021

Binárne rozhodovacie diagramy sú dátová štruktúra, ktorá slúži na zjednodušenie práce s booleovskými funkciami v programovaní. Toto zadanie je rozdelené na tri funkcie, BDD\_create(), ktorá náš binárny rozhodovací diagram vytvorí, BDD\_reduce, ktorá z diagramu odstráni dané nepotrebné elementy(ďalej spomínané ako uzly), a BDD\_use, ktorá pomocou stavu vstupných premenných do našej booleovskej funkcie nájde jednoduchý výstup, 0 alebo 1.

**Obsah:**

1. **Pomocné funkcie/štruktúry, ktoré slúžia ako infraštruktúra pre môj program. (str. 3)**
2. **Hlavná časť programu - create, reduce, use. (str. 10)**
3. **Testovanie a zložitosť (str. 11)**
4. **Pomocné funkcie/štruktúry:**

**Štruktúry:**



**BF:** štruktúra, ktorá v sebe drží string, v ktorom je reprezentovaný vektor booleovskej funkcie. Takisto obsahuje počet premenných pre danú booleovskú funkciu.

**UZOL**: štruktúra, ktorá reprezentuje každý element/uzol nášho diagramu. Má dva typy, a to terminály a neterminály. Terminály sú listy nášho diagramu/stromu, ktoré majú hodnotu premennej terminal rovnú 1 alebo 0, určujúc ich hodnotu.

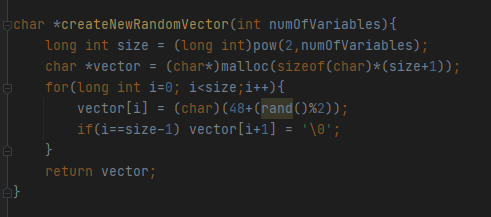
Neterminály sú všetky ostatné uzly, ktoré majú hodnotu terminal rovnú 2.

skipRight a skipLeft sú hodnoty reprezentujúce počet za sebou "preskočených" premenných pri čítaní v BDD\_use. Táto hodnota začína na 0, a upravuje sa v tretej fáze BDD\_reduce.

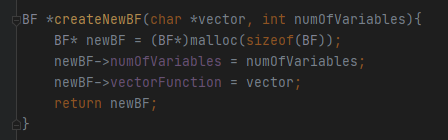
**BDD**: štruktúra reprezentujúca celkový BDD. Drží v sebe root diagramu, počet prvkov diagramu, a počet premenných diagramu.

**Pomocné funkcie:**

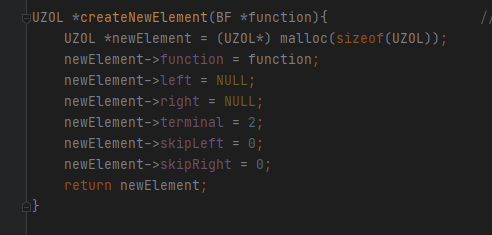
**createNewRandomVector()** vygeneruje náhodný vektor pre booleovskú funkciu s daným počtom premenných. Výsledok je string núl a jednotiek, s dĺžkou 2^(počet premenných).



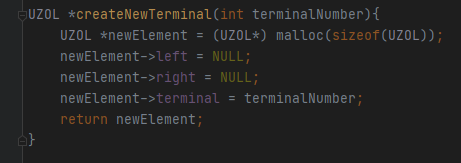
**createNewBF()** vytvorí a alokuje miesto pre novú booleovskú funkciu v tvare štruktúry BF. Inicializuje ju poskytnutými hodnotami.



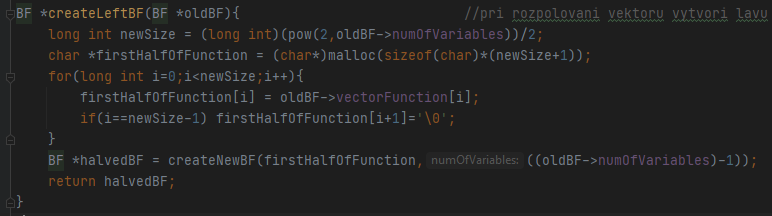
**createNewElement()** vytvorí a alokuje miesto pre nový UZOL, ktorý nie je terminál(**terminal**=2). Vloží do nej poskytnutú BF.



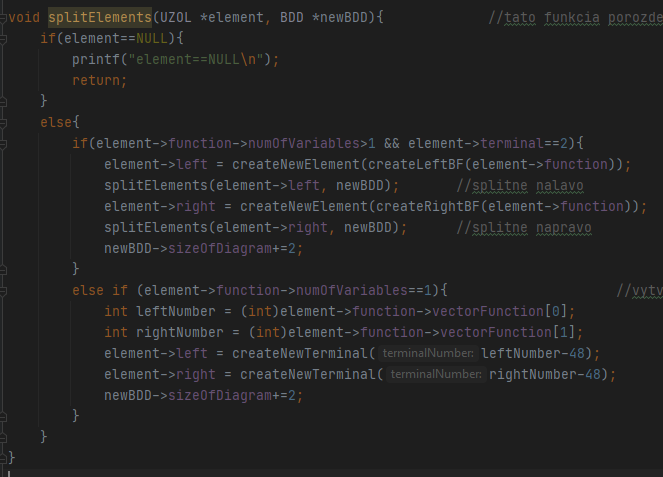
**createNewTerminal()** vytvorí a alokuje miesto pre nový UZOL, ktorý je terminál. Jeho hodnota premennej **terminal** je určená poskytnutým intom (1 alebo 0).



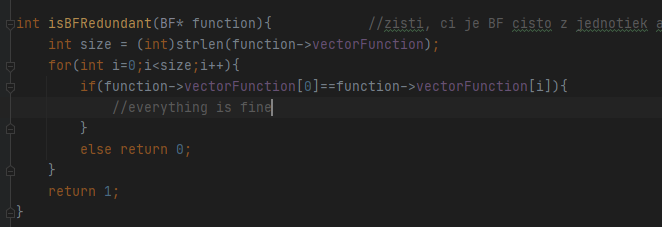
**createLeftBF()** a **createRightBF()** sú využívané na vytváranie ľavej/pravej polovice BF pri **BDD\_create()** vo funkcii **splitElements()**, ktorá bude opísaná nižšie.



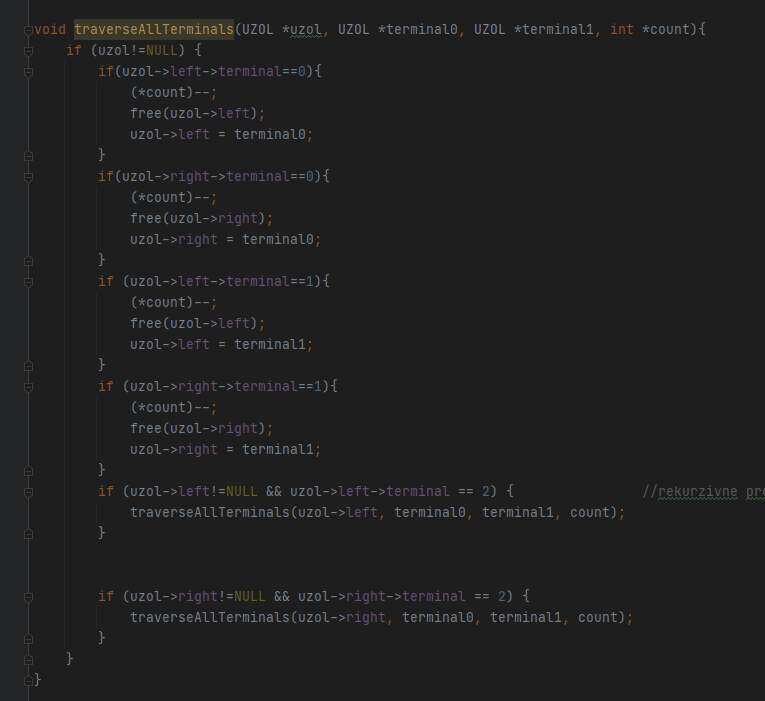
**splitElements()** je funkcia, ktorá rekurzívne rozpoľuje vektorové funkcie uložené v uzloch, až dokým sa to daľej nedá(vytvorí terminály). V podstate vytvorí celú uzlovú stavbu diagramu v **BDD\_create()**.



**isBFRedundant()** zistí, či je BF čisto z jednotiek a núl. Toto je potrebné kvôli výnimke pri tretej fáze **BDD\_reduce**, keďže pri preskakovaní premennej ukladám túto informáciu v parentovi preskočeného uzla, a root už parenta nemá.

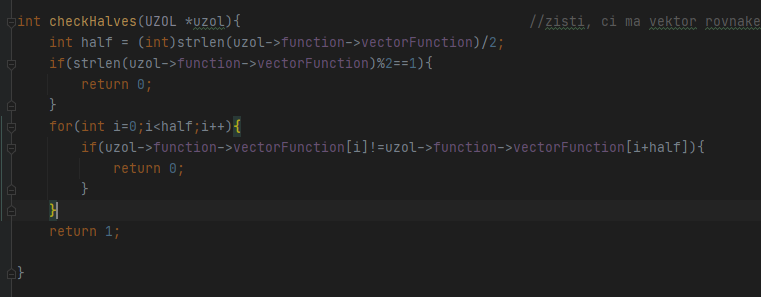


**traverseAllTerminals()** sa používa v prvej fáze **BDD\_reduce**, ktorá sa postará o prejdenie všetkých terminálov(rekurziou), ich odstránenie a nahradenie dvomi "globálnymi" pre jeden BDD.

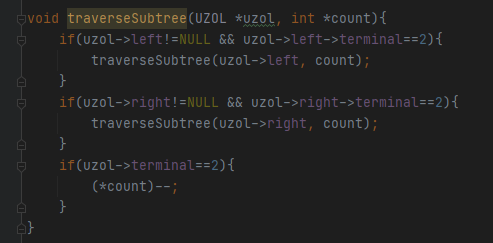


**mergeTerminals()** je funkcia, ktorá funguje ako sprostredkovávateľ **traverseAllTerminals()**. Vytvorí dva konkrétne konečné terminály a pošle ich do rekurzívnej funkcie, ktorá s nimi pracuje.

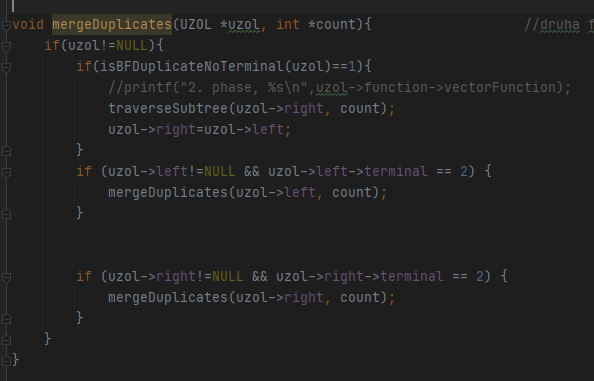
**checkHalves()** zistí, či vektor v uzle obsahuje dve rovnaké polovice. Toto je potrebné v tretej fáze **BDD\_reduce**.



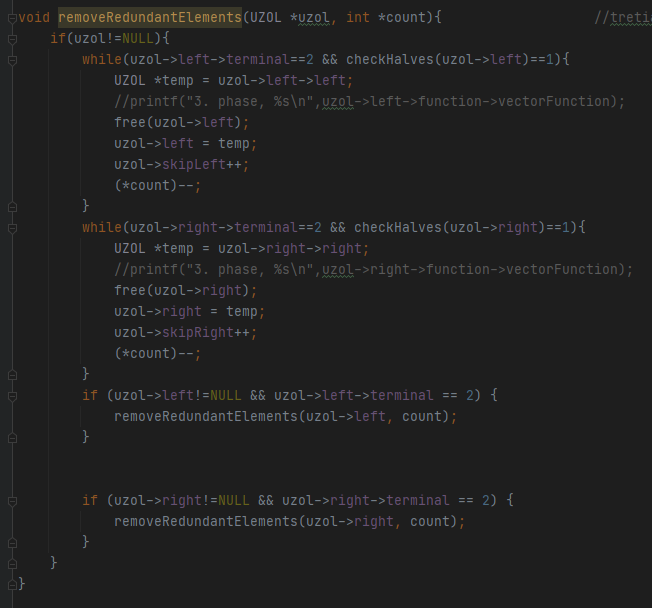
**traverseSubtree()** je funkcia, ktorá rekurzívne prejde odstránený podstrom v druhej fáze **BDD\_reduce**, pričom zráta zredukované uzly.



**mergeDuplicates()** sa vykonáva v druhej fáze redukcie, prechádza rekurzívne celý strom, nájde cez funkciu **isBFDuplicateNoTerminal()** uzol, ktorý sa skladá z rovnakých polovíc a pravého childa odstráni/redukuje.



**removeRedundantElements()** je funkcia, ktorá tvorí tretiu fázu redukcie. Rekurzívne prejde strom, pričom odstráni uzly, ktoré majú pointer na pravý aj ľavý child rovnaký. Pre parenta, ktorý ukazoval na odstránený child sa pointer upraví na nižší a upraví sa premenná skipLeft/skipRight.

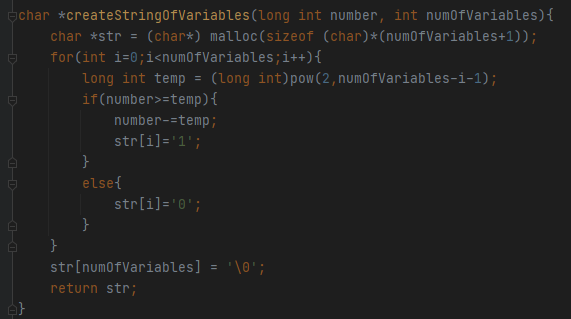


**createStringOfVariables()** je funkcia používaná pri testovaní. Pomocou parametrov dostane číslo určujúce vygenerovaný string, a počet premenných. Podľa týchto informácií vytvorí string, ktorý reprezentuje dané číslo v binárnej sústave.

Napríklad, pri čísle 6 a počte premenných 3 -> 110.

Pri čísle 6 a počte premenných 4 -> 0110.

Toto je potrebné pri testovaní všetkých možných kombinácií vstupných premenných v testovacej funkcii.



**testEverything()** je testovacia funkcia, ktorá dostane v parametroch počet iterácií, a počet premenných. Podľa počtu iterácií N vytvorí N náhodných BF, použije na ne BDD\_create() a BDD\_reduce(), a nakoniec pre každú možnú kombináciu vtupných premenných prejde BDD\_use() a porovná ho s pravým výsledkom. Funkcia vypočíta priemernú mieru redukcie diagramu, a čas, ktorý náš test trval.



1. **Hlavná časť programu:**

Hlavná časť programu sa skladá z funkcií **BDD\_create**, **BDD\_reduce** a **BDD\_use**.

**BDD\_create**:

Vytvorí pre booleovskú funkciu diagram. Vytvorí a alokuje miesto pre štruktúru BDD, to isté spraví pre root, ktorý do nej zapíše. Zavolá funkciu **splitElements()**, ktorá v skratke z rootu tvorí uzly(ľavý a pravý child) s rozpolenou funkciou. Toto sa rekurzívne opakuje, až dokým sa to ďalej nedá, teda funkcia vytvorila terminály/listy.

Funkcia vracia pointer na BDD.

**BDD\_reduce**:

Táto funkcia je rozdelená do troch častí.

1. V prvej časti sa volá funkcia **mergeTerminals()**, ktorá vytvorí dva "globálne" terminály pre daný BDD, všetky ostatné terminály odstráni a napojí tam tieto dva konkrétne terminály.
2. V druhej časti sa zavolá funkcia **mergeDuplicates()**, ktorá prejde celý strom, a pre uzly, ktorých childi majú rovnakú BF, jedného z nich odstráni(celý podstrom), a nastaví ho na toho, ktorý zostal.
3. V tretej časti sa zavolá funkcia **removeRedundantElements()**, ktorá prejde celý strom, a uzly, ktoré majú BF z rovnakých polovíc, odstráni. Toto celé robí z pohľadu parenta, a ak odstraňuje uzol, inkrementuje premennú skip(Left/Right, podľa toho, ktorý child bol odstránený). Táto premenná je potrebná pre orientáciu v diagrame, vo funkcii **BDD\_use().** Umožní nám preskočiť "premennú" v stringu so vstupnými premennými.

Funkcia vracia počet odstránených uzlov, pri chybe vráti -1.

**BDD\_use:**

Dostane v podobe parametra string so stavom vstupných premenných. Tento string je zapísaný v podobe núl a jednotiek, prvý char je A, druhý char je B, atď...

Funkcia v loope prechádza stromom od rootu v závislosti od vstupných premenných, pričom 0 znamená ľavý child, a 1 znamená pravý child. Ak je skipLeft/skipRight >0, pri pohybe v daný smer sa preskakujú premenné v stringu, keďže sú irelevantné.

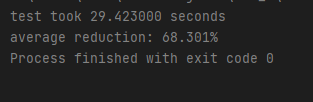
Funkcia vracia výsledný nájdený char('1' alebo '0'). Pri chybe vráti '9'.

1. **Testovanie a zložitosť:**

**Testovacia funkcia je podrobne opísaná vyššie.**

Požadovaný test je 2000 iterácií, 13 premenných.

Pre tento test mám:



Priemerná redukcia: približne 68%.

Priemerná časová zložitosť testu: 28 až 30,5 sekúnd.

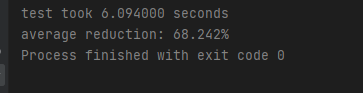
Ak počet iterácií zmenším na 200(desatina), čas je takisto desatina.

Ak počet premenných zvýšim o 1, čas sa zdvojnásobí(pre presnosť je to trošku viac ako dvojnásobok, ale pri odhadoch budem pre demonštračné účely pracovať s číslom 2).

Pre ďalší test (200 iterácií, 14 premenných):

Ak je časová zložitosť správne odhadnutá, predpokladáme čas približne 6 sekúnd(30/10\*2)

Výsledok:



Ostatné podobné testy sú podobne odhadnuteľné.

Pamäťová zložitosť môjho programu je:

sizeof(BDD) + (sizeof(UZOL) + sizeof(BF)) \* (2^(n+1)-1) + (rôzna\*\* dĺžka stringu vektora v BF) \* sizeof(char)

\*\* = tento string má rôznu veľkosť podľa počtu premenných, ktoré vyjadruje. Root string má dĺžku 2^n, pričom každá úroveň nižšie má polovicu svojho parenta. Terminály majú dĺžku 1.