SLOVENSKÁ TECHNICKÁ UNIVERZITA
Fakulta informatiky a informačních technologií
Ilkovičova 2, 842 16 Bratislava 4

Zadanie 3 – Binárne rozhodovacie diagramy

Dávid Penťa ID: 110871
Datové štruktúry a algoritmy
FIIT STU 2020/2021
Cvičenie: Švrtok 9:00
10.5.2021

Dávid Penťa
ID: 110871

Zadanie 3 – Binárne rozhodovacie diagramy

Vytvorte program, ktorý bude vedieť vytvoriť, redukovať a použiť dátovú štruktúru BDD (Binárny Rozhodovací Diagram) so zameraním na využitie pre reprezentáciu Booleovských funkcií.
Konkrétne implementujte tieto funkcie:

- BDD *BDD_create(BF **bfunkcia*);
- int BDD_reduce(BDD **bdd*);
- char BDD_use(BDD **bdd*, char **vstupy*);

Samozrejme môžete implementovať aj ďalšie funkcie, ktoré Vám budú nejakým spôsobom pomáhať v implementácii vyššie spomenutých funkcii, nesmiete však použiť existujúce funkcie na prácu s binárnymi rozhodovacími diagramami.
Funkcia **BDD_create** má slúžiť na zostavenie úplného (t.j. nie redukovaného) binárneho rozhodovacieho diagramu, ktorý má reprezentovať/opisovať zadanú Booleovskú funkciu (vlastná štruktúra s názvom BF), na ktorú ukazuje ukazovateľ *bfunkcia*, ktorý je zadaný ako argument funkcie **BDD_create**. Štruktúru BF si definujete sami – podstatné je, aby nejakým (vami vymysleným/zvoleným spôsobom) bolo možné použiť štruktúru BF na opis Booleovskej funkcie. Napríklad, BF môže opisovať Booleovskú funkciu ako pravdivostnú tabuľku, vektor, alebo výraz. Návratovou hodnotou funkcie **BDD_create** je ukazovateľ na zostavený binárny rozhodovací diagram, ktorý je reprezentovaný vlastnou štruktúrou BDD. Štruktúra BDD musí obsahovať minimálne tieto zložky: počet premenných, veľkosť BDD (počet uzlov) a ukazovateľ na koreň (prvý uzol) BDD. Samozrejme potrebujete aj vlastnú štruktúru, ktorá bude reprezentovať jeden uzol BDD.
Funkcia **BDD_reduce** má slúžiť na redukciu existujúceho (zostaveného) binárneho rozhodovacieho diagramu. Aplikovaním tejto funkcie sa nesmie zmeniť Booleovská funkcia, ktorú BDD opisuje. Cieľom redukcie je iba zmenšiť BDD odstránením nepotrebných (redundantných) uzlov. Funkcia **BDD_reduce** dostane ako argument ukazovateľ na existujúci BDD (*bdd*), ktorý sa má redukovať. Redukcia BDD sa vykonáva priamo nad BDD, na ktorý ukazuje ukazovateľ *bdd*, a preto nie je potrebné vrátiť zredukovaný BDD návratovou hodnotou (na zredukovaný BDD bude totiž ukazovať pôvodný ukazovateľ *bdd*). Návratovou hodnotou funkcie **BDD_reduce** je číslo typu int (integer), ktoré vyjadruje počet odstránených uzlov. Ak je toto číslo záporné, vyjadruje nejakú chybu (napríklad ak BDD má ukazovateľ na koreň BDD rovný NULL). Samozrejme, funkcia **BDD_reduce** má aktualizovať aj informáciu o počte uzlov v BDD.
Funkcia **BDD_use** má slúžiť na použitie BDD pre zadanú (konkrétnu) kombináciu vstupných premenných Booleovskej funkcie a zistenie výsledku Booleovskej funkcie pre túto kombináciu vstupných premenných. V rámci tejto funkcie „prejdete“ BDD stromom smerom od koreňa po list takou cestou, ktorú určuje práve zadaná kombinácia vstupných premenných. Argumentami funkcie **BDD_use** sú ukazovateľ s názvom *bdd* ukazujúci na BDD (ktorý sa má použiť) a ukazovateľ s názvom *vstupy* ukazujúci na začiatok poľa charov (bajtov). Práve toto pole charov/bajtov reprezentuje nejakým (vami zvoleným) spôsobom konkrétnu kombináciu vstupných premenných Booleovskej funkcie. Napríklad, index poľa reprezentuje nejakú premennú a hodnota na tomto indexe reprezentuje hodnotu tejto premennej (t.j. pre premenné A, B, C a D, kedy A a C sú jednotky a B a D sú nuly, môže ísť napríklad o “1010“), môžete si však zvoliť iný spôsob. Návratovou hodnotou funkcie

Dávid Penťa
ID: 110871

BDD_use je char, ktorý reprezentuje výsledok Booleovskej funkcie – je to buď '1' alebo '0'. V prípade chyby je táto návratová hodnota záporná, podobne ako vo funkcii **BDD_reduce**.
Okrem implementácie samotných funkcií na prácu s BDD je potrebné vaše riešenie dôkladne otestovať. Vaše riešenie musí byť 100% korektné. V rámci testovanie je potrebné, aby ste náhodným spôsobom generovali Booleovské funkcie, podľa ktorých budete vytvárať BDD pomocou funkcie BDD_create. Vytvorené BDD následne zredukujete funkciou BDD_reduce a nakoniec overíte 100% funkčnosť zredukovaných BDD opakovaným (iteratívnym) volaním funkcie BDD_use tak, že použijete postupne všetky možné kombinácie vstupných premenných. Počet premenných v rámci testovania BDD by mal byť minimálne 13. Počet Booleovských funkcií / BDD diagramov by mal byť minimálne 2000. V rámci testovania tiež vyhodnocujte percentuálnu mieru zredukovania BDD (t.j. počet odstránených uzlov / pôvodný počet uzlov).
Príklad veľmi jednoduchého testu (len pre pochopenie problematiky):

```
#include <string.h>
int main(){
BDD* bdd;
bdd = BDD_create("AB+C"); // alebo vektorom BDD_create("01010111")
BDD_reduce(bdd);
if (BDD_use("000") == '1')
printf("error, for A=0, B=0, C=0 it should be 0.\n");
if (BDD_use("001") == '0')
printf("error, for A=0, B=0, C=1 it should be 1.\n");
if (BDD_use("010") == '1')
printf("error, for A=0, B=1, C=0 it should be 0.\n");
if (BDD_use("011") == '0')
printf("error, for A=0, B=1, C=1 it should be 1.\n");
if (BDD_use("100") == '1')
printf("error, for A=1, B=0, C=0 it should be 0.\n");
if (BDD_use("101") == '0')
printf("error, for A=1, B=0, C=1 it should be 1.\n");
if (BDD_use("110") == '0')
printf("error, for A=1, B=1, C=0 it should be 1.\n");
if (BDD_use("111") == '0')
printf("error, for A=1, B=1, C=1 it should be 1.\n");
return 0;
}
```

Okrem implementácie vášho riešenia a jeho testovania vypracujte aj dokumentáciu, v ktorej opíšete vaše riešenie, jednotlivé funkcie, vlastné štruktúry, spôsob testovania a výsledky testovania, ktoré by mali obsahovať (priemernú) percentuálnu mieru zredukovania BDD a (priemerný) čas vykonania vašich funkcií. Dokumentácia musí obsahovať hlavičku (kto, aké zadanie odovzdáva), stručný opis použitého algoritmu s názornými nákresmi/obrázkami a krátkymi ukážkami zdrojového kódu, vyberajte len kód, na ktorý chcete extra upozorniť. Pri opise sa snažte dbať osobitý dôraz na zdôvodnenie správnosti vášho riešenia – teda dôvody prečo je dobré/správne, spôsob a vyhodnotenie testovania riešenia. Nakoniec musí technická dokumentácia obsahovať odhad výpočtovej (časovej) a priestorovej (pamäťovej) zložitosti vášho algoritmu. Celkovo musí byť cvičiacemu jasné, že viete čo ste spravili, že viete odôvodniť, že to je správne riešenie, a viete aké je to efektívne.

Dávid Penťa
ID: 110871

Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorene rozhodne cvičiaci, príp. aj o bodovej penalizácii). Odovzdáva sa jeden **zip** archív, ktorý obsahuje zdrojové súbory s implementáciou riešenia a testovaním + jeden súbor s dokumentáciou vo formáte **pdf**. **Vyžaduje sa tiež odovzdanie programu**, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu **main**).

Hodnotenie

Môžete získať celkovo 15 bodov, **nutné minimum je 6 bodov**.

Za implementáciu riešenia (3 funkcie) je možné získať celkovo 8 bodov, z toho 2 body sú za funkciu **BDD_create**, 4 body za funkciu **BDD_reduce** a 2 body za funkciu **BDD_use**. Pre úspešné odovzdanie implementácie musíte zrealizovať aspoň funkcie **BDD_create** a **BDD_use**. Za testovanie je možné získať 4 body (treba podrobne uviesť aj v dokumentácii) a za dokumentáciu 3 body (bez funkčnej implementácie 0 bodov). Body sú ovplyvnené aj prezentáciou cvičiacemu (napr. keď neviete reagovať na otázky vzniká podozrenie, že to **nie je vaša práca, a teda je hodnotená 0 bodov)**.

Dávid Penťa
ID: 110871

Použitá metóda

Binárny rozhodovací diagram sa vytvorí pomocou vektora. Keď príde ako vstup výraz, program najprv zistí jeho vektor a až potom vytvorí diagram. Z vektora postupným rozdeľovaním na polovice získa celý binárny rozhodovací diagram, ktorého koreň je pôvodný vektor a listy sú len 0 a 1.

Diagram sa redukuje od najnižšej hĺbky tak, že sa odstraňujú uzly s totožným obsahom. Všetky zostávajúce uzly sa musia správne poprepájať, odstránené uzly a všetky ich deti sa musia uvoľniť z pamäte.

Pri použití binárneho rozhodovacieho diagramu, program začína v koreni a pre každú 0 sa posunie do ľavého dieťaťa, pre každú 1 sa posunie do pravého dieťaťa. Po prejdení celého vstupu sa program musí nachádzať v liste. Ak sa v liste nenachádza, vstup mal nekorektnú dĺžku.

Po jeho využití program môže odstrániť aj zredukovaný rozhodujúci diagram tak, že umiestni ukazovatele na všetky jeho rozdielne uzly do spájaného zoznamu, ktorý následne vymaže. Tak sa zabezpečí, že sa program nebude snažiť vymazať už vymazaný uzol.

Implementácia funkcií

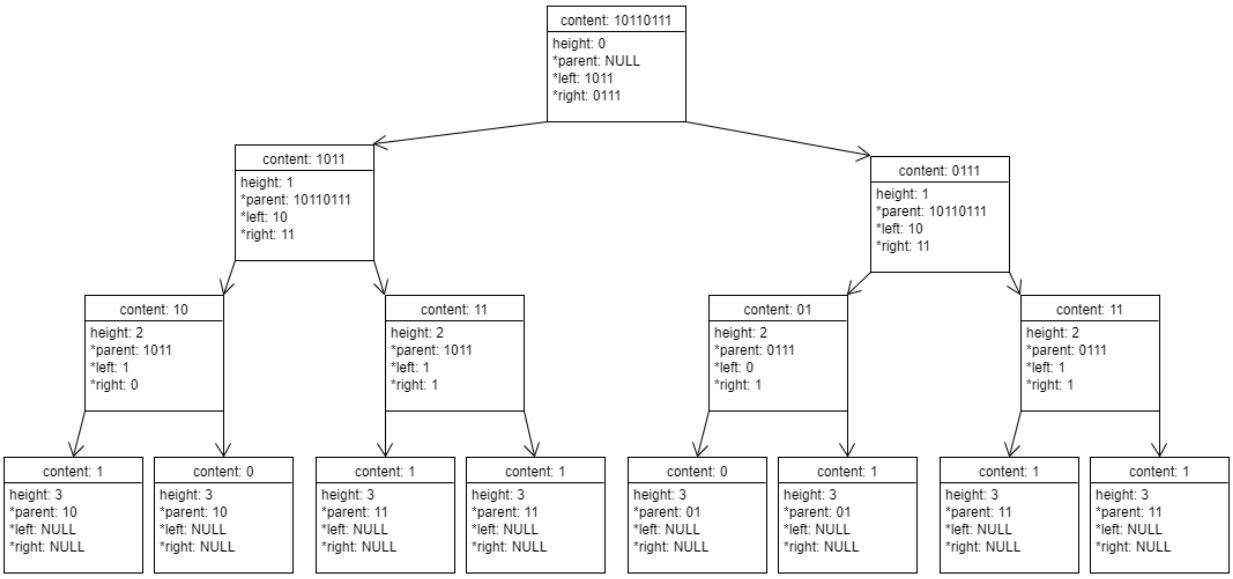
1. BDD *BDD_create(BF bfunkcia);

Funkcia najprv zistí, či je *BF funkcia* vektor alebo výraz. Ak je výraz, tak sa zavolá funkcia *void init(char* expr),* ktorá získa premenné a umiestni do zoznamu tak, aby sa žiadna premenná neopakovala a zavolá funkciu *void set_vars(int pos,char* expr, int vars_len)*. Vo funkcii *void set_vars(int pos,char* expr, int vars_len)* sa pre každú možnú kombináciu hodnôt premenných zavolá funkcia *char* changeVarsToBool(char* expr, int vars_len),* ktorej výstup sa pošle do *char getFunctionValue(char* str).* Vo funkcii *char* changeVarsToBool(char* expr, int vars_len)* sa Booleovský výraz upraví tak, že sa za každú hodnotu premennej dosadí jej aktuálna hodnota podľa funkcie *void set_vars(int pos,char* expr, int vars_len).* Napríklad, výraz abc+d+a!d+b keď a=0, b=1, c=0 a d=0 sa prepíše na 010+0+01+1. Takto prepísaný výraz je poslaný do funkcie *char getFunctionValue(char* str),* ktorá získa jeho finálnu hodnotu. Napríklad, pre reťazec 010+0+01+1 sa vráti hodnota 1. Takto získaná hodnota sa pripíše do globálneho reťazca *s* kde sa po skončení funkcie *set_vars()* vytvorí vektor pre počiatoční výraz. Z vektora sa vytvorí koreň a zavolá sa funkcia *void divide(BDD* node),* ktorá vytvorí binárny strom umiestňovaním ľavej polovice vektora do ľavého dieťaťa a pravej polovice do pravého dieťaťa, dokým nie sú len 1 a 0. Listy majú ako deti NULL.

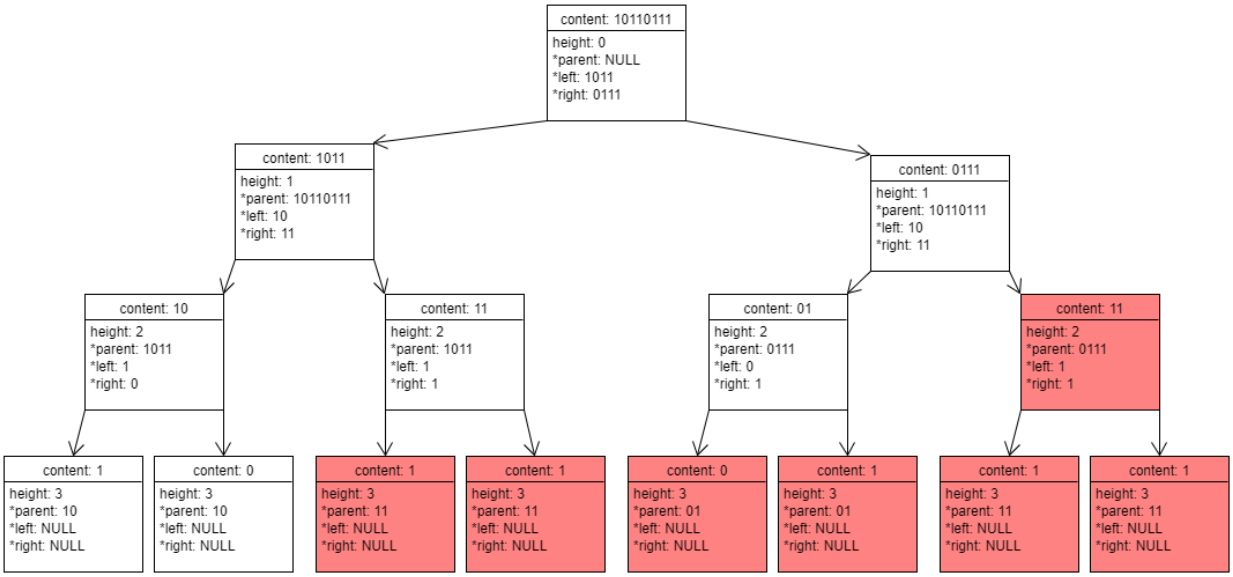
2. int BDD_reduce(BDD *bdd);

Táto funkcia zistí hĺbku stromu a zavolá funkciu *int level(BDD* root, int height).* Funkcia *int level(BDD* root, int height)* pre všetky výšky od 0 zavolá funkciu *int getLevel(BDD* root, BDD* b, int level),* ktorá získava všetky uzly v danej výške a pošle ich na porovnávanie do funkcie *int search(BDD* bddo, BDD* b).* Vo funkcii *int search(BDD* bddo, BDD* b)* sa zisťuje, či sa v rovnakej výške nachádza iný uzol s totožným obsahom. Ak taký nájde, tak sa ukazovateľ ktorý naň ukazuje zmení, aby ukazoval na uzol, ktorý bol poslaný do funkcie *int search(BDD* bddo, BDD* b).* Uzol, na ktorý už nič neukazuje, pošle do funkcie *int BDD_free(BDD* bddo),* ktorá ho a všetky jeho deti uvoľní z pamäte.

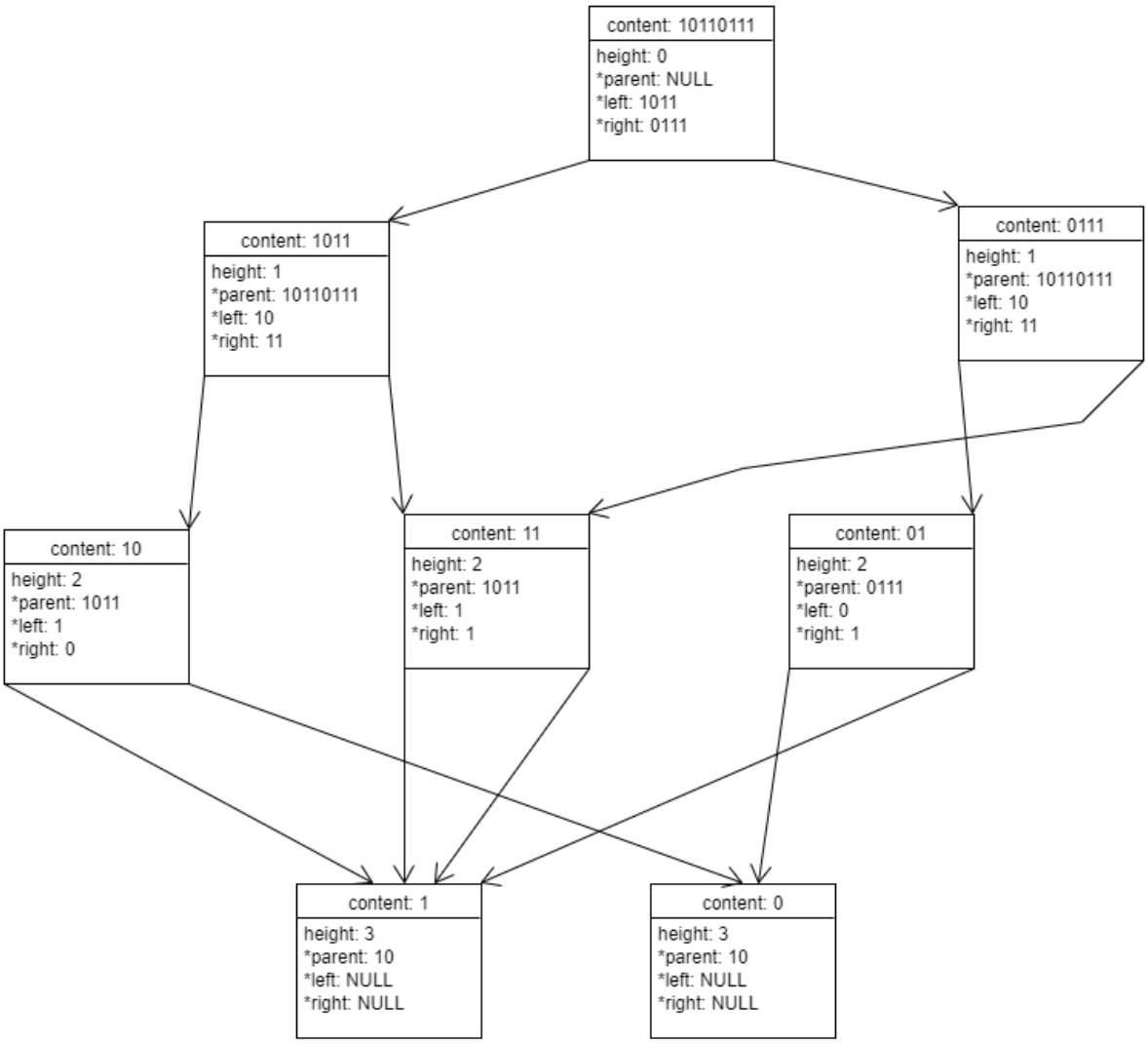
Dávid Penťa
ID: 110871



Obr. 1.: Reprezentácia binárneho rozhodovacieho diagramu pred redukciou.



Obr. 2.: Reprezentácia opakujúcich sa prvkov v binárnom rozhodovacom diagrame.



Obr. 3.: Reprezentácia binárneho rozhodovacieho diagramu po redukcii.

3. char BDD_use(BDD *bdd, char *vstupy);

Funkcia zistí výstup pre kombináciu premenných tým, že začína v koreni a pre každú 0 sa posunie do ľavého dieťaťa, a pre každú 1 sa posunie do pravého dieťaťa. Po prejdení celého vstupu sa program musí nachádzať v liste. Ak sa v liste nenachádza, vstup mal nekorektnú dĺžku. Hodnota v liste je výstup pre kombináciu premenných. Pri chybe program vráti – a do konzoly vypíše, kde sa stala chyba. Táto funkcia nepoužíva žiadne iné funkcie.

3. int BDD_free2(BDD* bddo)

Táto funkcia slúži na vymazanie zredukovaného diagramu a bolo nutné ju naprogramovať, aby sa v pamäti zbytočne nezdržiavali štruktúry, na ktoré už neukazujú žiadne ukazovatele. V tejto funkcii program vytvorí začiatok spájaného zoznamu a zavolá funkciu *int BDD_free2_1(BDD* bddo, struct bdd_free *root).* Vo

funkcii *int BDD_free2_1(BDD* bddo, struct bdd_free *root)* program prechádza každým uzlom binárneho rozhodovacieho diagramu, a ak sa v spájanom zozname ešte nenachádza, tak ho tam pridá. Následne funkcia *int BDD_free2(BDD* bddo)* vymaže všetky uzly na ktoré prvky v spájanom zozname ukazovali a následne aj celý spájaný zoznam.

Implementácia štruktúr

```
typedef struct {
    char name;
    char val;
} var;
```

Táto štruktúra slúži na uloženie všetkých rozdielnych premenných v Booleovskom výraze. Využíva sa pri zmene Booleovského výrazu na vektor.

```
typedef struct bdd{
    char content[20000];
    int height;
    struct bdd* parent;
    struct bdd* left;
    struct bdd* right;
} BDD;
```

Táto štruktúra slúži na uloženie uzlov binárneho rozhodovacieho diagramu. Každý uzol má v sebe uloženú informáciu o časti vektora, hĺbke v strome, ktorý uzol na ňu v diagrame ukazuje a jej dve deti, ľavé dieťa, ktoré v sebe uchováva ľavú polovicu časti vektora, a pravé, ktoré v sebe uchováva pravú polovicu časti vektora.

```
struct bdd_free{
    struct bdd* this;
    struct bdd_free* next;
};
```

Táto štruktúra slúži na uloženie všetkých rôznych uzlov zredukovaného binárneho rozhodovacieho diagramu do spájaného zoznamu, aby mohol byť korektne vymazaný.

Dávid Penťa
ID: 110871

```
typedef struct bf{
    char content[20000];
} BF;
```

Táto štruktúra slúži na opis Booleovskej funkcie. Môže v nej byť uložený buď vektor alebo výraz.

Dokumentácia testovaní

Testoval som 500 náhodných Booleovských výrazov s 13 premennými, 500 náhodných Booleovských výrazov s 14 premennými, 500 náhodných vektorov s 13 premennými a 500 náhodných vektorov s 14 premennými. Príklad, náhodne generovaných Booleovských výrazov generovaných a overovaných programom: !d!h!b!i+daf+d!gl!mj+nid!c!d!je!m!k,

ce!e!jn!gn+e+!llj!a+c+!fb!h!lcbm!cd!h+!i!hm+k,

!i!hf+!g!nd+!k!a+!h!ma!c+!ib+ale!ijhl!l.

Počítal som čas a počet uzlov diagramu pred a po zredukovaní. Každý diagram som po zredukovaní otestoval pre všetky možné kombinácie hodnôt premenných porovnaním s pôvodným vektorom.

Výpis testovania

```
8191500 8175690 99.806995%
Time: 443.367000 seconds
16383500 16365666 99.891147%
Time: 1622.284000 seconds
8191500 7544315 92.099310%
Time: 774.695000 seconds
16383500 15225000 92.928861%
Time: 3384.839000 seconds

Total: 49150000 47310671 96.257723%
Total time: 6225.186000 seconds
```

Rozbor zložitosti

Časová zložitosť

1. *BDD *BDD_create(BF *bfunkcia)* má najhoršiu časovú zložitosť, keď na vstupe príde výraz a musí vytvoriť aj vektor $\Omega(2*2^{n+1})$, pretože 2^{n+1} operácií je potrebných na vytvorenie binárneho stromu delením predchádzajúcich uzlov a rovnaký počet operácií je potrebných na vytvorenie vektora. Preto $O(2^n)$.

Dávid Penťa
ID: 110871

2. *int BDD_reduce(BDD *bdd)* má v najhoršom prípade časovú zložitosť Ω (2^{2n+2}), pretože program prechádza každú výšku v strome a porovnáva v nej každý uzol s každým. Preto O(2^n).

3. *char BDD_use(BDD *bdd, char *vstupy)* má časovú zložitosť O(n), pretože program vykoná len *n* operácii (posun do pravého alebo ľavého dieťaťa).

Pamäťová zložitosť je 2^{n+1} - 1, kde n je počet premenných zo vstupu, vynásobená veľkosťou štruktúry BDD. Toľko pamäte je potrebnej na uloženie úplného binárneho rozhodovacieho diagramu. Ak chceme vymazať zredukovaný diagram, pamäťová zložitosť bude väčšia o spájaný zoznam, ktorý sa musí vytvoriť.

Záver

Pri testovaní sa overila 100% korektnosť môjho riešenia, pretože sa ani raz počas testovania nenašla chyba. Pre prvé dva scenáre, keď sa overovali náhodné Booleovské výrazy, bola vysoká miera zredukovania až nad 99%. To bolo spôsobené tým, že sa v ich vektoroch často opakovalo viacero 0 alebo 1 za sebou a bolo jednoduché ich redukovať. Ďalšie dva scenáre, ktoré overovali náhodné vektory mali oveľa menej dlhých častí, ktoré by sa často opakovali a preto ich zredukovanie dosahovalo len okolo 92%.