Juraj Hušek

Úvod

Binárny rozhodovací strom (BDD) je v úplnom tvare binárny strom slúžiaci na rozhodovanie, pretože každý jeho vnútorný uzol predstavuje jedno čiastkové rozhodnutie. Prechodom zhotoveného stromu od začiatku do konca – od koreňa po koncový list, vieme zistiť konečnú odpoveď na daný problém za daných okolností. Najčastejšiou aplikáciou BDD je vyhodnocovanie pravdivostných hodnôt Boolovských funkcií. V našom zadaní sme implementovali BDD v úlohách na vyhodnocovanie pravdivostných hodnôt rôznych Boolovských funkcií s rôznymi počtami premenných. Naše BDD sme následne redukovali pravidlami S a I, aby boli čo najefektívnejšie.

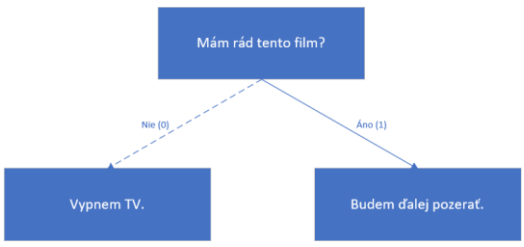
Technické parametre

Našu implementáciu BDD a funkcií na prácu s ním sme programovali v programovacom jazyku Java v IDE Eclipse 2022-12. Testovanie sme realizovali na zariadení s týmito parametrami:

Názov	Procesor	RAM	Pamäť	Grafická karta	Operačný systém
Dell G3 3590	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz	16,0 GB (15,8 GB usable) dual, DDR4	SSD PM991 NVMe Samsung 512 GB	NVIDIA GeForce GTX 1660 Ti with Max-Q	Windows 10 Home

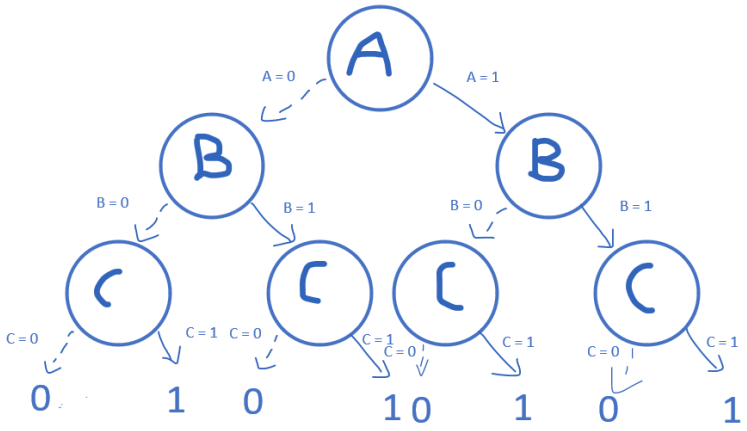
1 Teória Binárneho rozhodovacieho diagramu

BDD je často používané pri vyhodnocovaní komplexných problémov, napríklad dát, pokiaľ finálny výsledok závisí od čiastkových. Ukážka veľmi jednoduchého rozhodovacieho stromu pre vysvetlenie:



1.1 Aplikácia BDD na vyhodnotenie Boolovských funkcií v tvare DNF

Vďaka BDD vieme vyhodnotiť pravdivostné hodnoty akejkoľvek Boolovskej funkcie podobne ako pravdivostnou tabuľkou, Karnaughovou mapou či vektorom. Princíp spočíva v tom, že koreňom takéhoto BDD je samotná funkcia. Počet úrovní stromu je rovný počtu unikátnych premenných (nepočítame negácie), ktoré funkcia obsahuje. V každej úrovni stromu sa vyhodnocuje pravdivostná hodnota jednej premennej. Jednoduchá teoretická reprezentácia takéhoto BDD:



Pri aplikácii BDD na Boolovské funkcie sa riadime ich pravidlami. Tie najčastejšie používané, na ktoré sme pri riešení zadania narazili boli:

- Neutrálnosť konštanty (a+0=a; a*1=a)
- Vylúčenie tretieho (a+!a=1; a*!a=0)
- Komutatívnosť (a+b=b+a)
- Agresívnosť konštanty (a*0=0; a+1=1)
- Absorpcia (a+a=a; a*a=a)

Vzorce pre prácu s BDD použité v našom zadaní:

- Veľkosť BDD bez redukcií: $2^{(n+1)} - 1$
- Reduction rate: $100 - \left(\frac{počet\ uzlov\ po\ reduckii}{počet\ uzlov\ bez\ redukcie}\right) * 100$

1.2 Shannonova dekompozícia – rozklad DNF funkcie zhora nadol

BDD pre danú Boolovskú funkciu vieme vytvoriť prístupom zhora nadol – postupnou dekompozíciou jednotlivých premenných, kde sa v každej úrovni BDD vyhodnocuje jedna premenná funkcie. Táto dekompozícia sa volá Shannonova. Ukážka Shannonovej dekompozície na príklade:

Funkcia: AB+AC+BC

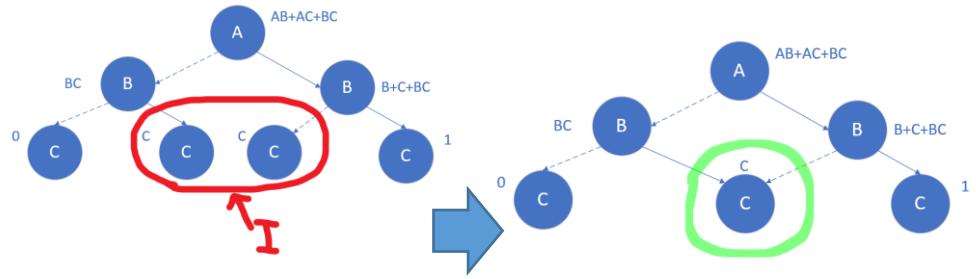
1. Koreň BDD reprezentuje samotná funkcia
2. Vyberieme si jednu premennú, napr. A = koreň je riadený premennou A (je praktické si pred rozkladom zaviesť poradie, podľa ktorého budeme určovať, ktorou premennou je aká úroveň riadená)

3. Vytvoríme potomkov a vložíme do nich zostatkové časti. Pokiaľ nám chýba premenná v nejakom výraze, vložíme tento výraz (časť výrazu) do obidvoch potomkov
4. Opakujeme kroky 2 a 3 pre každú premennú
5. V poslednej úrovni máme výraz len poslednú premennú alebo jej negovaný tvar – podľa neho určíme konečnú pravdivostnú hodnotu

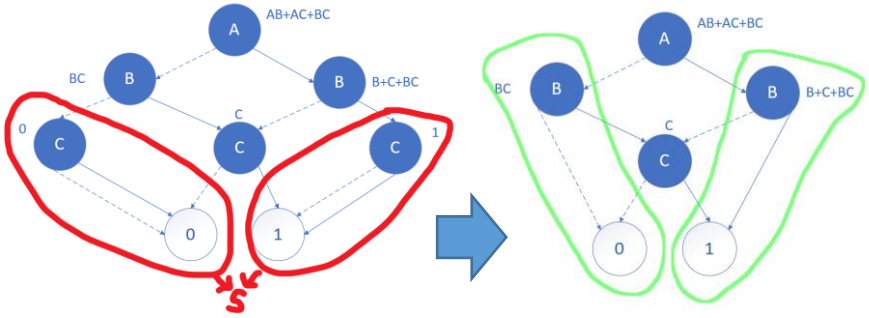
1.3 Veľkosť BDD a redukcie

Veľkosť BDD záleží od počtu uzlov. Vieme, že BDD má $\mathbf{2^{(n+1)} - 1}$ uzlov bez redukcie, kde n predstavuje počet unikátnych premenných funkcie, s ktorou BDD pracuje. Tým pádom má BDD s funkciou s tromi unikátnymi premennými napr. spomínaná AB+AC+BC presne $2^{(3+1)} - 1 = 15$ uzlov. BDD s 5 premennými ich má 63, s 10 presne 2047 a s 20 premennými až 2097151. Priveľký počet uzlov zapríčiní dlhší čas potrebný na vytvorenie BDD. Preto je potrebné BDD počas tvorby redukovať.

1. **Redukcia typu I**
   - Pokiaľ uzly v jednej úrovni reprezentujú rovnakú funkciu, je možné ich redukovať. Stačí nám jeden takýto uzol, na ktorý budú ukazovať uzly ukazujúce na túto funkciu.
   - Zložitosť takejto redukcie je O((2^n)^2) kde n je počet premenných funkcie
   - Ukážka na funkcii AB+AC+BC



2. **Redukcia typu S**
   - Pokiaľ ľavý potomok uzla je rovnaký ako pravý potomok uzla = uzol ukazuje na rovnakého potomka, môžeme ho redukovať, pretože je takýto uzol zbytočný. Rodič tohto uzla môže rovno ukazovať na potomka zredukovaného uzla.
   - Ukážka na funkcii AB+AC+BC



Taktiež je istou formou redukcie I vytvorenie len jedného uzla hodnoty 1 a hodnoty 0 a nech koncové uzly ukazujú na tieto (nepotrebujeme vlastné uzly s 0 a 1 pre každý koncový uzol).

2 Naša implementácia BDD

Úspešne sme implementovali BDD pomocou programovacieho jazyka Java s použitím niekoľkých programátorských prístupov.

2.1 Základný koncept našej implementácie

Rozhodovací strom je tvorený rekurzívne, každý uzol má pridelenú Boolovskú funkciu (po dekompozícii ak nastala) a poradie, aby sme z neho vedeli vyčítať, na ktorej úrovni sa uzol v strome nachádza a akú premennú v danej úrovni rieši. Negácie reprezentujú premenné vo tvare malého písmena. Počítame s tým, že v DNF funkciách sa nenachádzajú rovnaké premenné alebo negovaná a nenegovaná premenná zároveň. Na začiatku vytvoríme prvý uzol obsahujúci Boolovskú funkciu. Prvý uzol nerobí nič iné, len zavolá opäť metódu na vytvorenie uzla, ktorej parametre sú funkcia a nezmenené poradie, a to dvakrát – vytvorí pravý a ľavý uzol. Vždy sa program najprv zamerá na tvorbu pravej časti stromu, funkciu si rozdelí na menšie výrazy, spraví dekompozíciu podľa pravidiel a opäť pospája výrazy do jednej funkcie, skontroluje redukciu I, skontroluje možnosť použitia redukcie S, poradie skráti o prvú premennú a znovu volá metódu na tvorbu uzla s parametrami funkcie po dekompozícii a upraveným poradím. Podobným postupom tvorí aj ľavú časť podstromu koreňa a celé BDD až kým nepríde na koniec poradia alebo nepoužije jednu z redukcií. Použitie redukcie I kontroluje pomocou hashtabuľky, ak sa daný výraz v úrovni už v hashtabuľke nachádza, netvorí nový uzol, ale rodičovi pridelí už existujúci. Použitie redukcie S kontroluje tak, že sleduje či jeho rodič neukazuje na dve rovnaké deti v podobe jedného uzla. Ak áno, vymaže rodiča a starého rodiča uzla prepojí so sebou. Pokiaľ sa program nachádza v poslednej úrovni alebo funkcia nejakého uzla je rovná 0 alebo 1, rodič tohto uzla sa automaticky spája s uzlami 0 alebo 1, ktoré vytvárame zvlášť na začiatku celej tvorby BDD.

2.2 Vysvetlenie najdôležitejších častí kódu a jeho správnosť

2.2.1 Triedy

Celý program stojí na 3 triedach: BDD, Uzol a Main.

```
package bdd;

import java.util.ArrayList;

public class BDD {
    public int pocetPremennych;
    public String Bfunkcia;
    public String Poradie;
    public int pocetUzlov;
    public int pocetUzlovPoRedukcii;
    public Uzol korenUzol;
    public float reductionRate;
    ArrayList<Uzol> uzly = new ArrayList<Uzol>();
}
```

Trieda BDD reprezentuje samotný diagram, uchováva o ňom informácie a aj list všetkých uzlov. Trieda BDD obsahuje aj zadané zložky – počet premenných v premennej pocetPremennych, veľkosť v premennej pocetUzlovPoRedukcii a ukazovateľ na koreň v premennej Uzol korenUzol.

```
package bdd;

public class Uzol {
    public Uzol LeftChild;
    public Uzol RightChild;
    public Uzol Parent;
    public String Udef;
    int Ustatus;
    Uzol(Uzol parent, String def, int status)
    {
        this.Parent = parent;
        this.Udef = def;
        this.Ustatus = status;
    }
}
```

Trieda Uzol reprezentuje jeden uzol stromu. Obsahuje ukazovatele na svoje deti a rodiča, Boolovskú funkciu, s ktorou daný uzol pracuje a status – ten rozhoduje, či je uzol ľavým („0“) alebo pravým dieťaťom („1“) rodiča.

```
package bdd;

import java.io.BufferedReader;

public class Main {
    static BDD tree = new BDD();
    @SuppressWarnings("unchecked")
    static Hashtable<String, String>[] hashtable = (Hashtable<String,String>[])new Hashtable<?,?>[10];
    static int pocetNodov = 0;
```

Trieda Main reprezentuje samotný program, obsahuje metódy a funkcie na vytvorenie stromu rekurziou, zadané metódy **BDD_create**, **BDD_create_with_best_order**, **BDD_use** a testovacie funkcie spolu s metódou main, z ktorej všetko riadime.

2.2.2 BDD_create(String bfunkcia, String poradie)

```
public static void BDD_create(String bfunkcia, String poradie)
{
    pocetNodov = 0;
    SetHashtables(poradie);

    tree.Bfunkcia = bfunkcia;
    tree.Poradie = poradie;
    tree.pocetPremennych = poradie.length();

    Uzol Empty = new Uzol(null, bfunkcia, 2);
    KonecneUzly(Empty);
    newUzol(bfunkcia,poradie,2,Empty);
    RunEndy();
    //System.out.println("Pocet Uzlov po redukcii: " + pocetNodov);
    tree.pocetUzlovPoRedukcii = pocetNodov;
    pocetNodovBezRedukcie = (int) (Math.pow(2,(poradie.length()+1)) - 1);
    //System.out.println("Pocet nodov bez redukcie: " + pocetNodovBezRedukcie);
    tree.pocetUzlov = pocetNodovBezRedukcie;
    System.out.println("\n\n");
    vypisRightt();
    tree.korenUzol = PoleUzlov.get(2);
    tree.uzly = new ArrayList<>(PoleUzlov);
    System.out.println("Pocet Uzlov po redukcii: " + pocetNodov);
    System.out.println("Pocet nodov bez redukcie: " + pocetNodovBezRedukcie);
    float reductionRate = 100 - (Float.valueOf(pocetNodov) / Float.valueOf(pocetNodovBezRedukcie))*100f;
    System.out.printf("Reduction Rate in percents: " + reductionRate);
    tree.reductionRate = reductionRate;
    PoleTestStromov.add(tree);
    tree = null;
    tree = new BDD();
    PoleUzlov.clear();
}
```

Táto metóda po zavolaní riadi proces tvorby nového BDD pre zadaná funkciu a poradie. Strom vytvára pomocou globálnej premennej BDD tree, po vytvorení pridá nový BDD do ArrayListu pre BDD, odkiaľ je ho možné kedykoľvek vrátiť. Ako prebieha proces tvorby BDD po zavolaní metódy BDD_create:

1. Zavolá sa metóda SetHashtables

```
public static void SetHashtables(String poradie)
{
        hashtable[0] = new Hashtable<String,String>();
}
```

   Tá vytvorí pole hashových tabuliek, v implementácii nakoniec používame podľa rady prednášajúceho len jednu hashovú tabuľku kvôli obmedzeniam pamäte pre listy.

2. Nastaví sa premenná tree, ktorú použijeme pre tvorbu BDD a zavoláme metódu na vytvorenie konečných uzlov 0 a 1 s parametrom prázdneho Uzla (potrebný len na začatie tvorby, nezasahuje do samotného BDD).

```
public static void KonecneUzly(Uzol tempParent)
{

    if(createdLast == false)
    {
        pocetNodov += 2;
        Uzol nula = new Uzol(tempParent, "0", 0);
        Uzol jedna = new Uzol(tempParent, "1", 1);
        PoleUzlov.add(nula);
        PoleUzlov.add(jedna);
    //  System.out.println("0");
    //  System.out.println("1");

    }

}
```

3. Zavolá sa metóda newUzol s parametrom Boolovskej funkcie, zadaného poradia premenných pre dekompozíciu, statusom rovným 2 (status 2 znamená špeciálny prípad, kedy tvoríme koreň BDD) a prázdnym Uzlom Empty (ktorý je potrebný len pre funkčnosť a modularitu funkcie, nijako nezasahuje do samotného BDD). Začneme rekurzívne tvoriť BDD.

```
Uzol Empty = new Uzol(null, bfunkcia, 2);
KonecneUzly(Empty);
newUzol(bfunkcia,poradie,2,Empty);
```

4. Samotný základ metódy newUzol spočíva v kontrole statusu z parametra. Pri tvorbe koreňa je dostala metóda status rovný 2, čo je špeciálny prípad.

```
    if(status == 2)
    {
        pocetNodov++;
    //  System.out.println(pomocnaF + " " + pomocnePoradie);
        Uzol Utemp = new Uzol(parent, pomocnaF,2);
        PoleUzlov.add(Utemp);
        newUzol(pomocnaF,pomocnePoradie,1,Utemp);
        newUzol(pomocnaF,pomocnePoradie,0,Utemp);
    }
    }
}
```

Zväčší sa počet uzlov, vytvorí sa nový Uzol Utemp nesúci zadanú funkciu a pridá sa do globálneho ArrayListu PoleUzlov, kde reprezentuje prvý uzol – koreň BDD. Zavolá sa newUzol s parametrami celej funkcie, zadaného poradia z parametra metódy, statusom 1 – pravé dieťa koreňa a rodičom, ktorým je koreň. Rovnako sa zavolá neUzol pre ľavé dieťa so statusom 0.

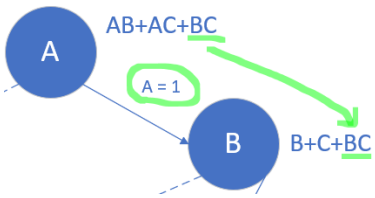
5. Zavolaná metóda newUzol so statusom 1 začne vyhodnocovanie pridania pravého dieťaťa rodičovi. Zo zadaného poradia vyberie do Stringu prvú premennú – podľa nej bude vyhodnocovať tento uzol. Skontroluje, či funkcia, ktorú dostala v parametri nie je náhodou 0 alebo 1, ak áno, rodičovi pridelí koncový uzol danej hodnoty ako pravé dieťa a nepokračuje. Ak pokračuje, rozdelí si Boolovksú funkciu na jednotlivé výrazy podľa znaku +.

```
String[] mensieVyrazy = pomocnaF.split("\\+");
```

Program pokračuje v časti pre status rovný 1. Začne prechádzať všetkými rozdelenými výrazmi. Ak sa vo výraze nenachádza hľadaná premenná a ani jej negácia a výraz nie je prázdny, prvok pridáme do Stringu novaFunkcia2, ktorý reprezentuje novú funkciu po dekompozícii.

```
if(prvok.indexOf(CheckChar) == -1 && prvok.indexOf(negaC)== -1 && prvok != " ")
{
    //System.out.println("!!!!! " + prvok);
    if(poc == 0){
        novaFunkcia2 = prvok;
        poc++;
    }
    else{

        novaFunkcia2 = novaFunkcia2 + "+" + prvok;
        poc++;
    }
}
```

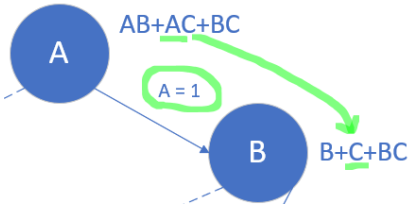
Príklad kedy nastane takáto situácia:



Ak sa vo výraze nachádza hľadaná premenná, prvok pridáme do Stringu finálnej funkcie.

```
if(prvok.indexOf(CheckChar) != -1 && prvok.indexOf(negaC)== -1 && prvok != " ") {
    if(poc == 0){
        novaFunkcia2 = prvok;
        poc++;
    }
    else{
        novaFunkcia2 =  novaFunkcia2 +"+" + prvok;
        poc++;
    }
}
```

Príklad kedy nastane táto situácia:



Ak sa vo výraze nenachádza hľadaná premenná ale jej negácia sa v ňom nachádza a funkcia obsahuje len jednu premennú, zmeníme finálny String na 0. Inak výraz do funkcie nepridáme.

```
if(prvok.indexOf(CheckChar) == -1 && prvok.indexOf(negaC)!= -1 && prvok != " " && prvok.length() == 1) {
    if(poc == 0){
        novaFunkcia2 = "0";
        poc++;
    }
    else{
        novaFunkcia2 =  novaFunkcia2;;
        poc++;
    }
}
```

Ak je výraz dĺžky 1 a obsahuje len hľadanú premennú a funkcia obsahuje len jednu premennú, finálna funkcia je rovná 1.

```
if(prvok.indexOf(CheckChar) != -1 && prvok.indexOf(negaC)== -1 && prvok != " " && prvok.length() == 1) {
    if(poc == 0){
        novaFunkcia2 = "1";
        poc++;
    }
    else{
        novaFunkcia2 =  novaFunkcia2;;
        poc++;
    }
}
```

Následne prebehne kontrola celého novovytvoreného Stringu. Pokiaľ je rovný hľadanej premennej, zmení sa na 1, pokiaľ negácii hľadanej premennej, zmení sa na 0. Pokiaľ funkcia obsahuje 1 a nie je dĺžky 1 (pravidlo a+1=1), zmení sa na 1. Po týchto kontrolách prebehne konečná úprava funkcie, vymaže sa z nej hľadaná premenná. Pokiaľ je po finálnej úprave String prázdny, zmení sa na 0. Poslednou kontrolou zistíme, či sa vo funkcii nachádza prípad pravidla a+!a alebo a!+a, pokiaľ áno, funkcia sa zmení na 1.

```
if(novaFunkcia2.indexOf(CheckChar+1) == 2){
    novaFunkcia2 = "1";
}
if(novaFunkcia2.indexOf(negaC+1) == 2){
    novaFunkcia2 = "0";
}
if(novaFunkcia1.indexOf("1") != -1)
{
    novaFunkcia1 = "1";
}
String odstranZPoradie = String.valueOf(CheckChar);
String novePoradie = pomocnePoradie.replace(odstranZPoradie, "");
String FinalF = novaFunkcia2.replace(negaC, "");
FinalF = novaFunkcia2.replace(OdstranC, "");

if(FinalF == "")
{
    FinalF = "0";
    System.out.println("Nasiel som prazdne, je tam " + FinalF);
}


if(FinalF.length() == 3)
{
    if(FinalF.charAt(0) == Character.toUpperCase(FinalF.charAt(2)) || Character.toUpperCase(FinalF.charAt(0)) == FinalF.charAt(2))
    FinalF = "1";
}
if(FinalF.length() == 2)
{
    if(FinalF.charAt(1) == '+')
    FinalF = String.valueOf(FinalF.charAt(0));
}
```

Proces pridania uzla pokračuje samotným pridaním do stromu. Najprv sa skontroluje, či sa funkcia po úprave nachádza v hashtabuľke – prebehne **Redukcia S**. Ak nie a zároveň nie je rovná ani 1 ani 0, prebehne klasické pridanie uzla do BDD. Zväčší sa počet uzlov, funkcia sa pridá do hashtabuľky a vytvorí sa Uzol Utemp obsahujúci rodičovský uzol z parametra, upravenú funkciu a status 1. Tento uzol sa pridá do ArrayListu všetkých uzlov a zavolá sa dvakrát metóda newUzol, prvá – pravé dieťa, s parametrami upravenej funkcie, upraveného poradia, statusom 1 a rodičovským uzlom Utemp a druhá – ľavé dieťa, s parametrami upravenej funkcie, upraveného poradia, statusom 0 a rodičom Utemp.

```
//FinalF = FinalF.substring(0,FinalF.length()-1);
if(hashtable[0].get(FinalF) == null) {
    if(FinalF != "1" && FinalF != "0")
    {
        pocetNodov++;
        hashtable[0].put(FinalF,FinalF);
//      System.out.println(FinalF + " " + novePoradie);
        Uzol Utemp = new Uzol(parent, FinalF,1);
        PoleUzlov.add(Utemp);
        int tempP = PoleUzlov.indexOf(parent);
        PoleUzlov.get(tempP).RightChild = Utemp;
        newUzol(FinalF,novePoradie,1,Utemp);
        newUzol(FinalF,novePoradie,0,Utemp);
    }
```

Pokiaľ sa funkcia rovná 0 alebo 1, rodičovi sa ako pravé dieťa priradí uzol s danou hodnotou funkcie. Zároveň tu prebieha **Redukcia I**, pokiaľ rodič obidvomi deťmi ukazuje na 0, vymaže sa rodičovský uzol z poľa uzlov a jeho funkcia z hashtabuľky a rodičovi vymazaného rodiča sa pridelí ľavé dieťa uzol 0. Zmenší sa počet uzlov. Pokiaľ obe rodičove deti ukazujú na 1, vymaže sa rodič a jeho funkcia rovnakým spôsobom, rodičovi vymazaného rodiča sa pridelí pravé dieťa uzol 1.

```
else if(FinalF == "0")
{
    for(Uzol t : PoleUzlov)
    {
        if(t.Udef == "0")
        {
            parent.RightChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
        //      System.out.println("priasd");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
            }
        }
    }
}
```

```
else
{
    for(Uzol t : PoleUzlov)
    {
        if(t.Udef == "1")
        {
            parent.RightChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
        //      System.out.println("prinad");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
            }
        }
    }
}
```

V prípade, že uzol už existuje a neprebieha proces pridania detí, skontrolujeme pomocou hashtabuľky či rodič nepridaného prvku ukazuje obomi deťmi na rovnaký prvok. Ak áno, prebehne **Redukcia I**, vymaže sa rodič a nájdený uzol je priradený rodičovi vymazaného rodiča buď vpravo alebo vľavo, podľa toho kde sa vymazaný rodičovský uzol nachádzal.

```
else {
    for(Uzol t : PoleUzlov)
    {
        if(hashtable[0].get(FinalF) == hashtable[0].get(t.Udef))
        {
        //  System.out.println("Nasiel som " + FinalF);
            parent.RightChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
                //  System.out.println("prinad");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";
                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
            }
        }
    }
}
```

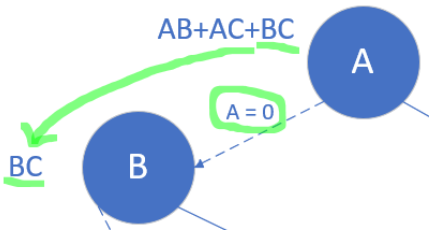
6. Zavolaná metóda newUzol so statusom 0 začne vyhodnocovanie pridania ľavého dieťaťa rodičovi. Zo zadaného poradia vyberie do Stringu prvú premennú – podľa nej bude vyhodnocovať tento uzol. Skontroluje, či funkcia, ktorú dostala v parametri nie je náhodou 0 alebo 1, ak áno, rodičovi pridelí koncový uzol danej hodnoty ako ľavé dieťa a nepokračuje. Ak pokračuje, rozdelí si Boolovksú funkciu na jednotlivé výrazy podľa znaku +.

```
String[] mensieVyrazy = pomocnaF.split("\\+");
```

Program pokračuje v časti pre status rovný 0. Začne prechádzať všetkými rozdelenými výrazmi. Ak sa vo výraze nenachádza hľadaná premenná a ani jej negácia a výraz nie je prázdny, prvok pridáme do Stringu novaFunkcia1, ktorý reprezentuje novú funkciu po dekompozícii.

```
for(String prvok : mensieVyrazy)
{
    if(prvok.indexOf(CheckChar) == -1 && prvok != " " && prvok.indexOf(negC) == -1)
    {
        if(poc1 == 0)
            novaFunkcia1 = prvok;
        else
            novaFunkcia1 = prvok + "+" + novaFunkcia1;
        poc1++;
    }
```

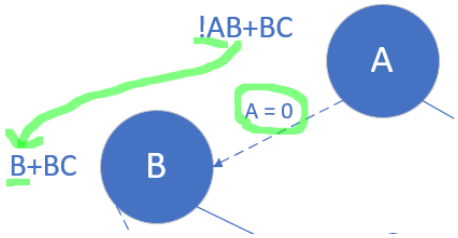
Obrázok reprezentujúci túto situáciu:



Pokiaľ sa vo výraze nachádza negácia hľadanej premennej, pridáme výraz do novej funkcie.

```
if(prvok.indexOf(negC) != -1 && prvok.length() != 1)
{
    if(poc1 == 0)
    novaFunkcia1 = prvok;
    else novaFunkcia1 = prvok + "+" + novaFunkcia1;
    poc1++;

}
```

Obrázok reprezentujúci túto situáciu:



Následne prebehne kontrola celého novovytvoreného Stringu. Pokiaľ je rovný negácii hľadanej premennej, zmení sa na 0, podobne ak je prázdny. Pokiaľ funkcia obsahuje 1 a nie je dĺžky 1 (pravidlo a+1=1), zmení sa na 1. Po týchto kontrolách prebehne konečná úprava funkcie, vymaže sa z nej negácia hľadanej premennej. Pokiaľ je po finálnej úprave String prázdny, zmení sa na 0. Poslednou kontrolou zistíme, či sa vo funkcii nachádza prípad pravidla a+!a alebo a!+a, pokiaľ áno, funkcia sa zmení na 1.

```
if(novaFunkcia1.indexOf(CheckChar+1) == 2){
    novaFunkcia1 = "1";
}
if(novaFunkcia1.indexOf(negC+1) == 2){
    novaFunkcia1 = "0";
}
if(novaFunkcia1 == ""){
    novaFunkcia1 = "0";
}
if(novaFunkcia1.indexOf("1") != -1)
{
    novaFunkcia1 = "1";
}
String FinalF = novaFunkcia1.replace(negC, "");
String odstranZPoradie = String.valueOf(CheckChar);
String novePoradie = pomocnePoradie.replace(odstranZPoradie, "");

if(FinalF == "")
{
    FinalF = "0";
//  System.out.println("Nasiel som prazdne, je tam " + FinalF);
}
if(FinalF.length() == 3)
{
    if(FinalF.charAt(0) == Character.toUpperCase(FinalF.charAt(2)) || Character.toUpperCase(FinalF.charAt(0)) == FinalF.charAt(2))
    FinalF = "1";
}
if(FinalF.length() == 2)
{
    if(FinalF.charAt(1) == '+')
    FinalF = String.valueOf(FinalF.charAt(0));
}
```

Proces pridania uzla pokračuje samotným pridaním do stromu. Najprv sa skontroluje, či sa funkcia po úprave nachádza v hashtabuľke – prebehne **Redukcia S**. Ak nie a zároveň nie je rovná ani 1 ani 0, prebehne klasické pridanie uzla do BDD. Zväčší sa počet uzlov, funkcia sa pridá do hashtabuľky a vytvorí sa Uzol Utemp obsahujúci rodičovský uzol z parametra, upravenú funkciu a status 0. Tento uzol sa pridá do ArrayListu všetkých uzlov a zavolá sa dvakrát metóda newUzol, prvá – pravé dieťa, s parametrami upravenej funkcie, upraveného poradia, statusom 1 a rodičovským uzlom Utemp a druhá – ľavé dieťa, s parametrami upravenej funkcie, upraveného poradia, statusom 0 a rodičom Utemp.

```
if(hashtable[0].get(FinalF) == null) {
    if(FinalF != "0" && FinalF != "1")
    {
        pocetNodov++;
        hashtable[0].put(FinalF,FinalF);
    //  System.out.println(FinalF + " " + novePoradie);
        Uzol Utemp = new Uzol(parent, FinalF,0);
        PoleUzlov.add(Utemp);
        int tempP = PoleUzlov.indexOf(parent);
        PoleUzlov.get(tempP).LeftChild = Utemp;
        newUzol(FinalF,novePoradie,1,Utemp);
        newUzol(FinalF,novePoradie,0,Utemp);
    }
```

Pokiaľ sa funkcia rovná 0 alebo 1, rodičovi sa ako ľavé dieťa priradí uzol s danou hodnotou funkcie. Zároveň tu prebieha **Redukcia I**, pokiaľ rodič obidvomi deťmi ukazuje na 0, vymaže sa rodičovský uzol z poľa uzlov a jeho funkcia z hashtabuľky a rodičovi vymazaného rodiča sa pridelí ľavé dieťa uzol 0. Zmenší sa počet uzlov. Pokiaľ obe rodičove deti ukazujú na 1, vymaže sa rodič a jeho funkcia rovnakým spôsobom, rodičovi vymazaného rodiča sa pridelí pravé dieťa uzol 1.

```
else if(FinalF == "0")
{
    for(Uzol t : PoleUzlov)
    {
        if(t.Udef == "0")
        {
            parent.LeftChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
        //      System.out.println("pripad");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";
                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";
                    pocetNodov--;
                }
            }
        }
    }
}
```

```
else if(FinalF == "1")
{
    for(Uzol t : PoleUzlov)
    {
        if(t.Udef == "1")
        {
            parent.LeftChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
        //      System.out.println("pripad");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";
                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
            }
        }
    }
}
```

V prípade, že uzol už existuje a neprebieha proces pridania detí, skontrolujeme pomocou hashtabuľky či rodič nepridaného prvku ukazuje obomi deťmi na rovnaký prvok. Ak áno, prebehne **Redukcia I**, vymaže sa rodič a nájdený uzol je priradený rodičovi vymazaného rodiča buď vpravo alebo vľavo, podľa toho kde sa vymazaný rodičovský uzol nachádzal.

```
else {
    for(Uzol t : PoleUzlov)
    {
        if(hashtable[0].get(FinalF) == hashtable[0].get(t.Udef))
        {
        //  System.out.println("Nasiel som " + FinalF);
            parent.LeftChild = t;
            if(parent.LeftChild != null && parent.RightChild != null && parent.LeftChild.Udef == parent.RightChild.Udef)
            {
                //System.out.println("prinad");
                if(parent.Parent.LeftChild == parent)
                {
                    parent.Parent.LeftChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).LeftChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";

                    pocetNodov--;
                }
                else
                {
                    parent.Parent.RightChild = t;
                    PoleUzlov.get(PoleUzlov.indexOf(parent.Parent)).RightChild = t;
                    hashtable[0].remove(parent.Udef);
                    PoleUzlov.get(PoleUzlov.indexOf(parent)).Udef = "Zmazany";
                    pocetNodov--;
                }
            }
        }
    }
}
```

7. Po skončení tvorby BDD sa zavolá ešte metóda RunEndy

```
newUzol(bfunkcia,poradie,2,Empty);
RunEndy();
```

Tá má za úlohu prebehnúť koncové uzly a skontrolovať, či ukazujú správne na 0 alebo 1. Ide o prípad, ak má konečný uzol funkciu dĺžky jeden obsahujúci negáciu.

```
public static void RunEndy()
{

    for(Uzol t : PoleUzlov)
    {

        if(t.Udef.length() == 1)
            {
            if(t.Udef != "0" && t.Udef != "1" && t.LeftChild.Udef == "0" && t.RightChild.Udef == "1" )

            {
                char[] prepis = t.Udef.toCharArray();
                if(Character.isUpperCase(prepis[0]) != true && t.Udef != "0" && t.Udef != "1")
                {
                    for(Uzol t2 : PoleUzlov)
                    {
                        if(t2.Udef == "1")
                        {
                            t.LeftChild = t2;
                        }
                    }
                    for(Uzol t3 : PoleUzlov)
                    {
                        if(t3.Udef == "0")
                        {
                            t.RightChild = t3;
                        }
                    }
                }

            }
        }
    }
}
```

Nastavia sa atribúty triedy BDD, vypočíta sa počet uzlov bez redukcií (vzorec $\mathbf{2^{(n+1)} - 1}$), zavolá sa metóda, ktorá prejde poľom uzlov a spraví výpis BDD. Ukážka výpisu BDD na funkcii AB+AC+BC na poradí ABC:

```
0 lc: nema rc: nema
1 lc: nema rc: nema
AB+AC+BC lc: BC rc: B+C+BC
B+C+BC lc: C rc: 1
C lc: 0 rc: 1
BC lc: 0 rc: C
Pocet Uzlov po redukcii: 6
Pocet nodov bez redukcie: 15
Reduction Rate in percents: 60.0
```

Stromu tree typu BDD sa nastaví koreň ako uzol z ArrayListu na pozícii 2, a ArrayListu stromu tree sa nastaví list uzlov BDD.

```
tree.pocetUzlovPoRedukcii = pocetNodov;
pocetNodovBezRedukcie = (int) (Math.pow(2,(poradie.length()+1)) - 1);
//System.out.println("Pocet nodov bez redukcie: " + pocetNodovBezRedukcie);
tree.pocetUzlov = pocetNodovBezRedukcie;
System.out.println("\n\n");
vypisRightt();
tree.korenUzol = PoleUzlov.get(2);
tree.uzly = new ArrayList<>(PoleUzlov);
```

8. Vypočíta sa Reduction rate vzorcom $\mathbf{100} - \left(\frac{\textit{počet uzlov po redukcii}}{\textit{počet uzlov bez redukcie}}\right) * \mathbf{100}$ ktorý je v percentách. Do ArrayListu BDD stromov sa pridá vytvorený BDD tree, odkiaľ je ho možné kedykoľvek zavolať a vypísať. Globálna premenná tree sa vynuluje a vytvorí sa prázdny BDD strom pripravený na pripravený na ďalšie volanie metódy BDD_create. Podobne sa vyprázdni aj ArrayList PoleUzlov.

```
    float reductionRate = 100 - (Float.valueOf(pocetNodov) / Float.valueOf(pocetNodovBezRedukcie))*100f;
    System.out.printf("Reduction Rate in percents: " + reductionRate);
    tree.reductionRate = reductionRate;
    PoleTestStromov.add(tree);
    tree = null;
    tree = new BDD();
    PoleUzlov.clear();
}
```

2.2.3 BDD_create_with_best_order(String bfunkcia, String poradie)

```
public static void BDD_create_with_best_order(String bfunkcia, String origin_poradie)
{
    getFact(origin_poradie,"");
    int pocetPremennych = origin_poradie.length();

    for(int i = 0; i < PolePoradi.size(); i++)
    {
        BDD_create(bfunkcia,PolePoradi.get(i));

    }
    bestTree = PoleTestStromov.get(0);
    for(int i = 0; i < PoleTestStromov.size(); i++)
    {System.out.println(PoleTestStromov.get(i).pocetUzlovPoRedukcii);
        if(ZapisujemeOrder == true)
        {
            OrderVypis.add(Integer.toString(PoleTestStromov.get(i).pocetUzlovPoRedukcii));
        }
        if(bestTree.pocetUzlovPoRedukcii > PoleTestStromov.get(i).pocetUzlovPoRedukcii)
        {

            bestTree = PoleTestStromov.get(i);

        }
    }

    System.out.println("\nNajlepsi order pre BDD s funkciou " + bestTree.Bfunkcia + " z poctu diagramov: " + PoleTestStromov.size() + " je " + bestTree.Poradie + ", p
}
```

Metóda má za úlohu nájsť najlepšie možné poradie z rôznych poradí pre vytvorenie BDD danej Boolovskej funkcie. Volaním metódy getFact s parametrom default poradia, z parametra BDD_create_with_best_order, vytvoríme ArrayList rôznych poradí, ktoré potom použijeme pre tvorbu BDD. Pre funkcie s počtom unikátnych

premenných **menším ako 5 tvoríme všetky možné kombinácie poradia**, pre **väčšie funkcie tvoríme náhodných n*n poradí**, kde n je počet premenných.

```
public static void getFact(String p, String an) {
    if(p.length() < 5)
    {
     if (p.length() == 0) {
            PolePoradi.add(an);
            return;
        }

        for (int i = 0; i < p.length(); i++) {
            char ch = p.charAt(i);

            String rest = p.substring(0, i) +
                        p.substring(i + 1);
            getFact(rest, an+ch);
        }
    }
    else
    {
        for(int i = 0; i < (p.length()*p.length()); i++)
        {
            List<String> characters = Arrays.asList(p.split(""));
            Collections.shuffle(characters);
            String after = "";
            for(String c : characters)
            {
                after += c;
            }
            PolePoradi.add(after);
        }
    }
}
```

Následne voláme BDD_create toľkokrát, koľko poradí sme vytvorili. Parametrami BDD_create je daná Boolovská funkcia a vytvorené poradie.

```
for(int i = 0; i < PolePoradi.size(); i++)
{
    BDD_create(bfunkcia,PolePoradi.get(i));

}
```

Po vytvorení BDD stromu pre každé poradie prejdeme všetkými BDD a hľadáme to s najmenším počtom uzlov po redukciách. Po nájdení uložíme toto BDD do **BestTree**, ktorý je globálnou premennou a dá sa kedykoľvek znovu zavolať. Taktiež realizujeme výpis.

```
bestTree = PoleTestStromov.get(0);
for(int i = 0; i < PoleTestStromov.size(); i++)
{System.out.println(PoleTestStromov.get(i).pocetUzlovPoRedukcii);
    if(ZapisujemeOrder == true)
    {
        OrderVypis.add(Integer.toString(PoleTestStromov.get(i).pocetUzlovPoRedukcii));
    }
    if(bestTree.pocetUzlovPoRedukcii > PoleTestStromov.get(i).pocetUzlovPoRedukcii)
    {

        bestTree = PoleTestStromov.get(i);

    }
}

System.out.println("\nNajlepsi order pre BDD s funkciou " + bestTree.Bfunkcia + " z poctu diagramov: " + PoleTestStromov.size() + " je " + bestTree.Poradie + ", po
```

Ukážka výpisu po zrealizovaní BDD_create_with_best_order:

```
Najlepsi order pre BDD s funkciou ABCDEFGHIJKLMN+AbcDEFgHIJklMn+ABCDEFGhijklm+ABCdefghijkl z poctu diagramov: 196 je ACHNBFDMGLJKEI, pocet pred redukciou: 32767 pocet uzlov po redukcii: 3 re

1 - Default BDD create na bfunkcii ABCDEFGHIJKLMN+AbcDEFgHIJklMn+ABCDEFGhijklm+ABCdefghijkl s poradím ABCDEFGHIJKLMN
2 - Default BDD best order na bfunkcii ABCDEFGHIJKLMN+AbcDEFgHIJklMn+ABCDEFGhijklm+ABCdefghijkl s poradím ABCDEFGHIJKLMN
3 - Default BDD use na bfunkcii AB+AC+BC s poradím ABC
4 - Skus vlastnu funkciu
5 - 100* najde najlepsi order pre funkcie s 13 premennymi
6 - 5* 100* zbehne bdd_create pre funkcie s 15, 13, 8, 4 a 3 premennymi, vypise casovu zlozitost do textfilov
```

2.2.4 BDD_use(String bfunkcia, String poradie, String premenne)

```
public static void BDD_use(String bfunkcia, String poradie, String premenne)
{
    BDD_create(bfunkcia,poradie);
    BDD strom = PoleTestStromov.get(0);

    System.out.println("\n" + strom.uzly.get(2).Udef);
    Uzol aktualny = strom.uzly.get(2);
    boolean mam = false;
    for(int i = 0; i < premenne.length(); i++)
    {
        System.out.println(premenne.charAt(i) + " " + aktualny.Udef);
        if(premenne.charAt(i) == '1')
        {
            aktualny = strom.uzly.get(strom.uzly.indexOf(aktualny.RightChild));
            if(aktualny.Udef == "1" || aktualny.Udef == "0")
            {
                System.out.println("Vysledok je: " + aktualny.Udef);
                mam = true;
                break;
            }
        }
        if(premenne.charAt(i) == '0')
        {

            aktualny = strom.uzly.get(strom.uzly.indexOf(aktualny.LeftChild));
            if(aktualny.Udef == "1" || aktualny.Udef == "0")
            {
                System.out.println("Vysledok je: " + aktualny.Udef);
                mam = true;
                break;
            }
        }
    }
    if(mam == false)
    {
        System.out.println("Vysledok je: -1");
    }System.out.println("\n\n\n");
}
```

Metóda má za úlohu prejsť BDD stromom a podľa zadaných hodnôt premenných určiť výslednú pravdivostnú hodnotu. Parametrom tejto funkcie je Boolovská funkcia, poradie premenných a String reprezentujúci hodnoty premenných v poradí z poradia premenných. Pre príklad zavolaním BDD_use(„AB+AC+BC“, „ABC“, „001“) zisťujeme výslednú pravdivostnú hodnotu BDD stromu pre AB+AC+BC s poradím premenných pre dekompozíciu ABC a hodnotami A=0, B=0 a C=1. Najprv sa vytvorí samotné BDD pomocou BDD_create. Následne sa do premennej Uzol aktuálny načíta koreň BDD danej funkcie.

```
BDD_create(bfunkcia,poradie);
BDD strom = PoleTestStromov.get(0);

System.out.println("\n" + strom.uzly.get(2).Udef);
Uzol aktualny = strom.uzly.get(2);
boolean mam = false;
```

Pre prejdenie stromom po jednotlivých úrovniach použijeme cyklus. Ten sa podľa hodnoty zo Stringu premenné rozhoduje, či sa v úrovni, ktorá sa rozhoduje podľa danej premennej posunie do ľavého alebo pravého dieťaťa (či je hodnota danej premennej v Stringu premenne 0 alebo 1). Takto prejde až do koncového uzlu 0 alebo 1, tento reprezentuje finálnu pravdivostnú hodnotu pre zadané hodnoty premenných. Finálnu hodnotu je možné volať z funkcie uzla aktuálny, spravíme aj výpis výsledku. Pokiaľ sa nenašiel výsledok, niekde nastala chyba a výsledkom je -1.

```
for(int i = 0; i < premenne.length(); i++)
{
    System.out.println(premenne.charAt(i) + " " + aktualny.Udef);
    if(premenne.charAt(i) == '1')
    {
        aktualny = strom.uzly.get(strom.uzly.indexOf(aktualny.RightChild));
        if(aktualny.Udef == "1" || aktualny.Udef == "0")
        {
            System.out.println("Vysledok je: " + aktualny.Udef);
            mam = true;
            break;
        }
    }
    if(premenne.charAt(i) == '0')
    {

        aktualny = strom.uzly.get(strom.uzly.indexOf(aktualny.LeftChild));
        if(aktualny.Udef == "1" || aktualny.Udef == "0")
        {
            System.out.println("Vysledok je: " + aktualny.Udef);
            mam = true;
            break;
        }
    }
}
if(mam == false)
{
    System.out.println("Vysledok je: -1");
}System.out.println("\n\n\n");
```

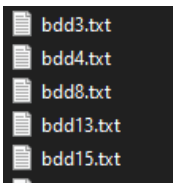
3 Testovanie našej implementácie BDD

Našu implementáciu BDD sme aj dôkladne otestovali. Pre tento účel sme vytvorili niekoľko metód, ktoré rôznymi spôsobmi testujú funkčnosť a efektivitu našej implementácie pomocou metód BDD_create, BDD_create_with_best_order a BDD_use.

3.1 Random DNF funkcie na testovanie

Pre lepšiu efektivitu a jednoduchšie testovanie sme vytvorili 500 náhodných DNF funkcií. Pre počty premenných 3, 4, 8, 14 a 15 sme vytvorili textové súbory, každý obsahuje 100 DNF funkcii s daným počtom unikátnych premenných. Funkcie z týchto súborov následne pomocou metódy ReadFiles načítame do jednotlivých ArrayListov, ktoré používame v niektorých testovacích funkciách.

```
//Test scenarios
static ArrayList<String>bdd_15_p = new ArrayList<>();
static ArrayList<String>bdd_13_p = new ArrayList<>();
static ArrayList<String>bdd_8_p = new ArrayList<>();
static ArrayList<String>bdd_4_p = new ArrayList<>();
static ArrayList<String>bdd_3_p = new ArrayList<>();

public static void ReadFiles() throws IOException
{

    BufferedReader bf0 = new BufferedReader(new FileReader("bdd15.txt"));
    String line0 = bf0.readLine();
    while(line0 != null)
    {
        bdd_15_p.add(line0);
        line0 = bf0.readLine();
    }
    bf0.close();


    BufferedReader bf1 = new BufferedReader(new FileReader("bdd13.txt"));
    String line = bf1.readLine();
    while(line != null)
    {
        bdd_13_p.add(line);
        line = bf1.readLine();
    }
    bf1.close();

    BufferedReader bf2 = new BufferedReader(new FileReader("bdd8.txt"));
    String line2 = bf2.readLine();
    while(line2 != null)
    {
        bdd_8_p.add(line2);
        line2 = bf2.readLine();
    }
    bf2.close();

    BufferedReader bf3 = new BufferedReader(new FileReader("bdd4.txt"));
    String line3 = bf3.readLine();
    while(line3 != null)
    {
        bdd_4_p.add(line3);
        line3 = bf3.readLine();
    }
    bf3.close();

    BufferedReader bf4 = new BufferedReader(new FileReader("bdd3.txt"));
    String line4 = bf4.readLine();
    while(line4 != null)
    {
        bdd_3_p.add(line4);
        line4 = bf4.readLine();
```



3.2 Limitácie nášho testovania

Našu implementáciu sme testovali na DNF funkciách, ktorých počet unikátnych premenných nebol väčší ako 15, implementácia by však mala zvládnuť aj väčší počet premenných.

3.3 Default testovanie BDD_use + testovanie ostatných metód na vlastnej Boolovskej funkcii

Po zavolaní testovacej metódy Ukaz_BDD_use za zavolá BDD_create s funkciou AB+AC+BC, poradím ABC a všetkými možnými hodnotami premenných. Jedná sa o jednoduchý test, ktorý však potvrdzuje správnosť nášho riešenia.

```
public static void Ukaz_BDD_USE()
{
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 001");
    BDD_use("AB+AC+BC","ABC","001");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 010");
    BDD_use("AB+AC+BC","ABC","010");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 100");
    BDD_use("AB+AC+BC","ABC","100");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 011");
    BDD_use("AB+AC+BC","ABC","011");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 111");
    BDD_use("AB+AC+BC","ABC","111");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 101");
    BDD_use("AB+AC+BC","ABC","101");
    System.out.println("bfunkcia AB+AC+BC, poradie ABC, premenne 110");
    BDD_use("AB+AC+BC","ABC","110");
}
```

Očakávaný výsledok a vyriešenie programom:

Tabuľka č. 1 – Testovanie BDD_use pre AB+AC+BC

Hodnoty premenných A, B, C	Očakávaný výsledok	Výsledok BDD_use
001	0	0
010	0	0
100	0	0
011	1	1
111	1	1
101	1	1
110	1	1

Korektnosť BDD_use = 100%

BDD_use je možné testovať aj zavolaním testovacej metódy SkusVlastnuF, kde používateľ môže otestovať BDD_create, BDD_create_with_best_order a BDD_use pre vlastnú DNF funkciu, s vlastným poradím a vlastnými hodnotami premenných.

```
@SuppressWarnings("resource")
public static void SkusVlastnuF()
{
    PoleTestStromov.clear();
    String bfunkcia;
    String poradie;
    Scanner scanIn = new Scanner(System.in);
    String scanLine = scanIn.nextLine();
    bfunkcia = scanLine;
    poradie = scanIn.nextLine();
    BDD_create(bfunkcia,poradie);
    String premenne = scanIn.nextLine();
    BDD_use(bfunkcia,poradie,premenne);
    PoleTestStromov.clear();
    BDD_create_with_best_order(bfunkcia,poradie);
}
```

Tabuľka č. 2 – Testovanie BDD_use pre ABCDEFGH+A!B!CDEF!GH+ABCDEFG!H+ABC!D!E!F!G!H

Hodnoty premenných A, B, C, D, E, F, G, H	Očakávaný výsledok	Výsledok BDD_use
10101010	0	0
11111111	1	1
00000000	0	0
00000001	0	0
11110000	0	0

Korektnosť BDD_use = 100%

3.4 Testovanie efektívnosti BDD_create_with_best_order na zredukovanosť BDD

Zmenou poradia premenných pri tvorbe BDD je možné dosiahnuť lepšie využitie Redukcie S a I a tým dosiahnuť lepšie zredukovaný strom. Testovali sme efektívnosť zmeny poradia premenných na zredukovanosť BDD metódou Test_100_13_BDD_Order a SkuskVlastnuF.

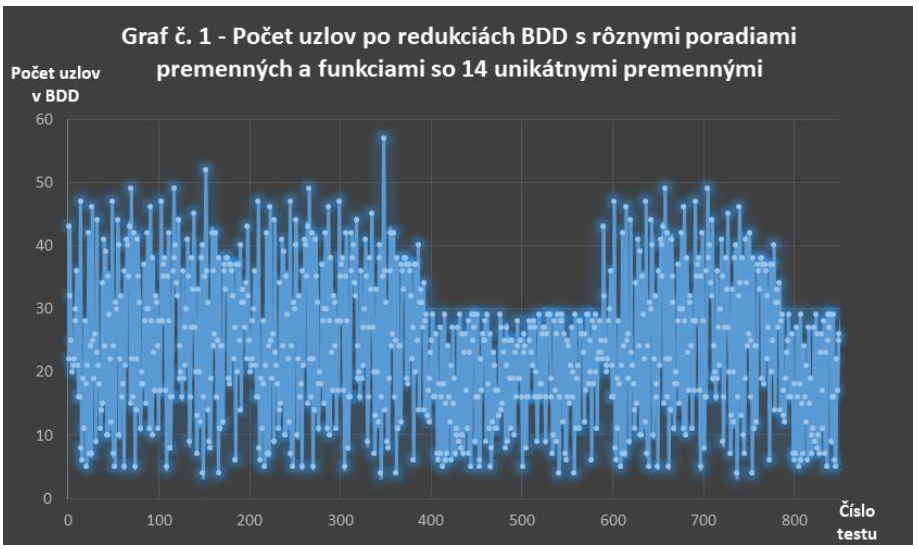
```
public static void Test_100_13_BDD_Order()
{
    //Killer f1
    ZapisujemeOrder = true;
    ArrayList<String>Vypis = new ArrayList<String>();
    String origin_poradie = "ABCDEFGHIJKLMN";
    for(String f : bdd_13_p)
    {
        BDD_create_with_best_order(f,origin_poradie);
    }

    Path outputOrder = Paths.get("outputOrder.txt");
    try {
        Files.write(outputOrder, OrderVypis);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Jej využitím sme otestovali efektivitu zmeny poradia na 100 DNF funkciách s 14 unikátnymi premennými – nezredukovaný BDD by mal 32767 uzlov. Po vytvorení každého zredukovaného BDD sme zapísali počet jeho uzlov do textového súboru. Pre každú funkciu sme pomocou BDD_create_with_best_order vytvorili 196 BDD s rôznymi poradiami premenných. Celkovo sme zapísali 19600 počtov uzlov jednotlivých zredukovaných BDD. Testovaním sme potvrdili, že zmena poradia premenných má veľký vplyv na možnú zredukovanosť BDD.

Tabuľka č. 3 - Závislosť zredukovanosti BDD od zmeny poradia premenných

Počet testovaných BDD pre danú Boolovskú funkciu	Počet unikátnych premenných Boolovskej funkcie	Počet uzlov BDD bez redukcií	Najhorší počet uzlov BDD po redukcii a zmene poradia	Najhorší Reduction rate (%)	Najlepší počet uzlov BDD po redukcii a zmene poradia	Najlepší Reduction rate (%)
196	14	32767	49	99,8504593	3	99,9908444
196	14	32767	29	99,9114963	5	99,9847407
196	14	32767	40	99,8779260	7	99,9786370
196	14	32767	27	99,9176000	5	99,9847407
196	14	32767	36	99,8901334	3	99,9908444
196	14	32767	29	99,9114963	5	99,9847407
196	14	32767	28	99,9145482	3	99,9908444
196	14	32767	41	99,8748741	4	99,9877926
196	14	32767	16	99,9511704	3	99,9908444
196	14	32767	48	99,8535112	6	99,9816889

Daná tabuľka bola vytvorená pre funkcie so 14 unikátnymi premennými skúšaním rôzneho poradia premenných, reprezentuje 10 použití BDD_create_with_best_order pre lepšiu prehľadnosť.



Graf č. 1 reprezentuje využitie BDD_create_with_best_order na zaznamenanie počtu uzlov BDD náhodných 5 funkcií so 14 unikátnymi premennými s použitím rôznych poradí premenných.

Tabuľka č. 4 – Ukážka zredukovanosti BDD s funkciou ABC+!BC+A!C+A!B!C s použitím rôznych poradí premenných

Poradie premenných	Počet uzlov BDD bez redukcií	Počet uzlov BDD po redukcii	Reduction rate (%)
ABC	15	7	53,333333333
ACB	15	5	66,666666667
BAC	15	6	60,000000000
BCA	15	4	73,333333333
CAB	15	6	60,000000000
CBA	15	6	60,000000000

Tabuľka č. 5 – Ukážka zredukovanosti BDD s funkciou ABCDEFGHIJKLMN+A!B!CDEF!GHIJ!K!LM!N+ABCDEFGH!I!J!K!L!M+ ABC!D!E!F!G!H!I!J!K!L s použitím rôznych poradí premenných

Poradie premenných	Počet uzlov BDD bez redukcií	Počet uzlov BDD po redukcii	Reduction rate (%)
ABCDEFGHIJKLMN	32767	41	99,874874111
HCDIFNBKGJAEML	32767	34	99,896237068
EJANCBDIMKLHFG	32767	11	99,966429640
CDIMNJKAHELBFG	32767	23	99,929807428
INLCFKDJEGBMAH	32767	44	99,865718558
MBEJHGILKFCNDA	32767	14	99,957274087
DHJMKLINCAFBGEL	32767	31	99,905392621
DNGJHBELCFAIKM	32767	23	99,929807428
CLEAGMNBJKFDHI	32767	31	99,905392621
JFBDIMGKALNEHC	32767	19	99,942014832
IEMJBFCKGNHALD	32767	13	99,960325938
KCMILBDHEGFNAJ	32767	28	99,914548173
DHFGKELIACJBMN	32767	47	99,856563005
ACIBEMFGJNDHKL	32767	3	99,990844447

Tabuľky č. 4 a č. 5 reprezentujú rôznu efektivitu redukcie BDD pre dané premenné s použitím rôznych poradí premenných.

3.5 Testovanie časovej a pamäťovej zložitosti BDD_create pre funkcie s rôznym počtom unikátnych premenných

Dôkladne sme otestovali časovú a pamäťovú náročnosť vytvorenia rôznych BDD stromov s rôznym počtom unikátnych premenných a abecedným poradím premenných. Test sme zrealizovali metódou BDD_CREATE_HUGE_TEST, ktorá volala BDD_create pre rôzne funkcie s rôznym počtom premenných z našich testovacích súborov. Merala časovú náročnosť realizácie BDD_create a výslednú hodnotu zapísala do súboru.

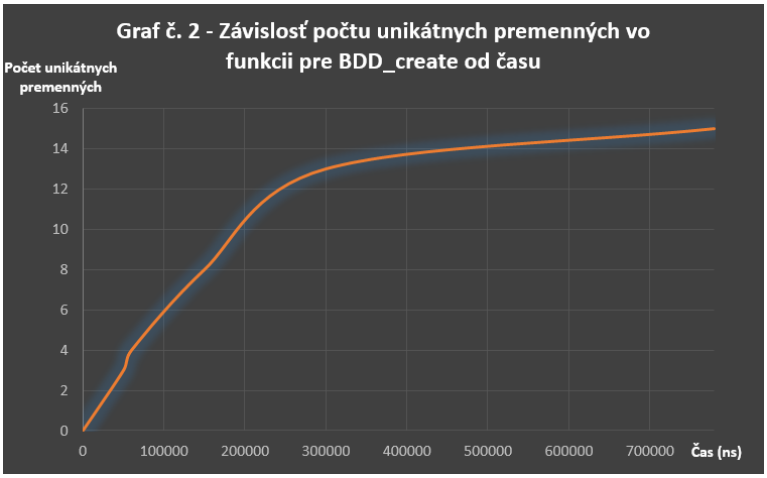
```java
public static void BDD_CREATE_HUGE_TEST()
{
    //KILLER F2
    ArrayList<String>casyPre15 = new ArrayList<String>();
    ArrayList<String>casyPre13 = new ArrayList<String>();
    ArrayList<String>casyPre8 = new ArrayList<String>();
    ArrayList<String>casyPre4 = new ArrayList<String>();
    ArrayList<String>casyPre3 = new ArrayList<String>();

    String poradie15 = "ABCDEFGHIJKLMNXY";
    String poradie13 = "ABCDEFGHIJKLMN";
    String poradie8 = "ABCDEFGH";
    String poradie4 = "ABCD";
    String poradie3 = "ABC";

    for(String f15: bdd_15_p)
    {
        long startTime = System.currentTimeMillis();
        BDD_create(f15,poradie15);
        long endTime = System.currentTimeMillis();
        long elapsed = (endTime-startTime)*1000000;
        casyPre15.add(String.valueOf(elapsed));
    }
    for(String f13: bdd_13_p)
    {
        long startTime = System.currentTimeMillis();
        BDD_create(f13,poradie13);
        long endTime = System.currentTimeMillis();
        long elapsed = (endTime-startTime)*1000000;
        casyPre13.add(String.valueOf(elapsed));
    }
```

```java
for(String f4: bdd_4_p)
{
    long startTime = System.currentTimeMillis();
    BDD_create(f4,poradie4);
    long endTime = System.currentTimeMillis();
    long elapsed = (endTime-startTime)*1000000;
    casyPre4.add(String.valueOf(elapsed));
}
for(String f3: bdd_3_p)
{
    long startTime = System.currentTimeMillis();
    BDD_create(f3,poradie3);
    long endTime = System.currentTimeMillis();
    long elapsed = (endTime-startTime)*1000000;
    casyPre3.add(String.valueOf(elapsed));
}
Path output0 = Paths.get("output15.txt");
Path output = Paths.get("output13.txt");
Path output2 = Paths.get("output8.txt");
Path output3 = Paths.get("output4.txt");
Path output4 = Paths.get("output3.txt");
try {
    Files.write(output0, casyPre15);
    Files.write(output, casyPre13);
    Files.write(output2, casyPre8);
    Files.write(output3, casyPre4);
    Files.write(output4, casyPre3);
} catch (Exception e) {
    e.printStackTrace();
}
```

Časovú náročnosť sme testovali pre Boolovské funkcie s počtom premenných 3, 4, 8, 14 a 15. Týchto funkcii bolo pre každý počet premenných 100, celkovo sme tak zrealizovali 500 testov s prednastavenými abecednými poradiami premenných. Testovaním sme potvrdili, že tvorba BDD s väčším počtom unikátnych premenných trvá dlhšie, pokiaľ sa nedá efektívne zredukovať. Testovaním sme potvrdili, že tvorba BDD s väčším počtom unikátnych premenných využíva viac pamäte, pokiaľ sa nedá efektívne zredukovať.

Tabuľka č. 6 – Priemerná časová a priestorová zložitosť BDD_create pre funkcie s rôznymi počtami unikátnych premenných

Počet unikátnych premenných	Priemerná časová zložitosť BDD_create (ns)	Priemerná pamäťová zložitosť BDD_create (MB)
3	50000	421
4	60000	465
8	150000	522
14	300000	580
15	780000	620



V tabuľke a aj v grafe môžeme vidieť, že priemerná časová a pamäťová zložitosť stúpa s počtom unikátnych premenných vo funkcii.

Záver

Úspešne sme implementovali BDD pracujúce s Boolovskými funkciami. Naše riešenie bolo testované na funkciách s najväčším počtom unikátnych premenných 15. Implementáciu sme dôkladne otestovali na 500 rôznych Boolovských funkciách s rôznym počtom unikátnych premenných. Testovaním sme potvrdili, že zmenou poradia premenných pre dekompozíciu vieme výrazne zefektívniť tvorbu BDD a zväčšiť jej Reduction rate. Taktiež sme potvrdili zvyšujúcu sa časovú a pamäťovú náročnosť pri rôznom počte premenných, pokiaľ sa nedá BDD efektívne zredukovať. Otestovali sme aj správnosť našej implementácie dosádzaním hodnôt premenných a následnou kontrolou výsledku. Všetky naše implementácie, testy a analýzy sme zdokumentovali, našu implementáciu je možné otestovať pomocou Main.java. Trieda obsahuje všetky testové operácie, ktoré sme použili.