DSA 2 assignment

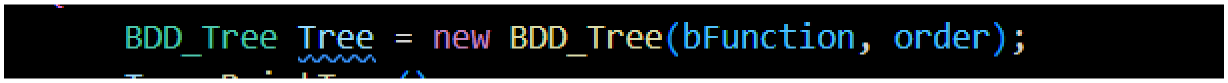
Binary Decision Diagram with reduction

Mykhailo Sichkaruk

Contents

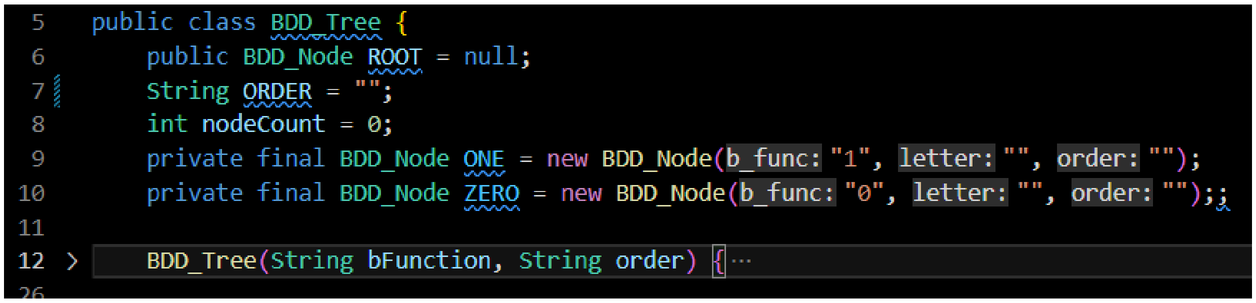
BDD_create(String bFunction, String order)

For creation of bdd tree I use java constructor of class BDD_tree.



```java
        BDD_Tree Tree = new BDD_Tree(bFunction, order);
```

It returns BDD_Tree class, that have such fields:



```java
public class BDD_Tree {
    public BDD_Node ROOT = null;
    String ORDER = "";
    int nodeCount = 0;
    private final BDD_Node ONE = new BDD_Node(b_func: "1", letter: "", order: "");
    private final BDD_Node ZERO = new BDD_Node(b_func: "0", letter: "", order: "");;

    BDD_Tree(String bFunction, String order) {···
```

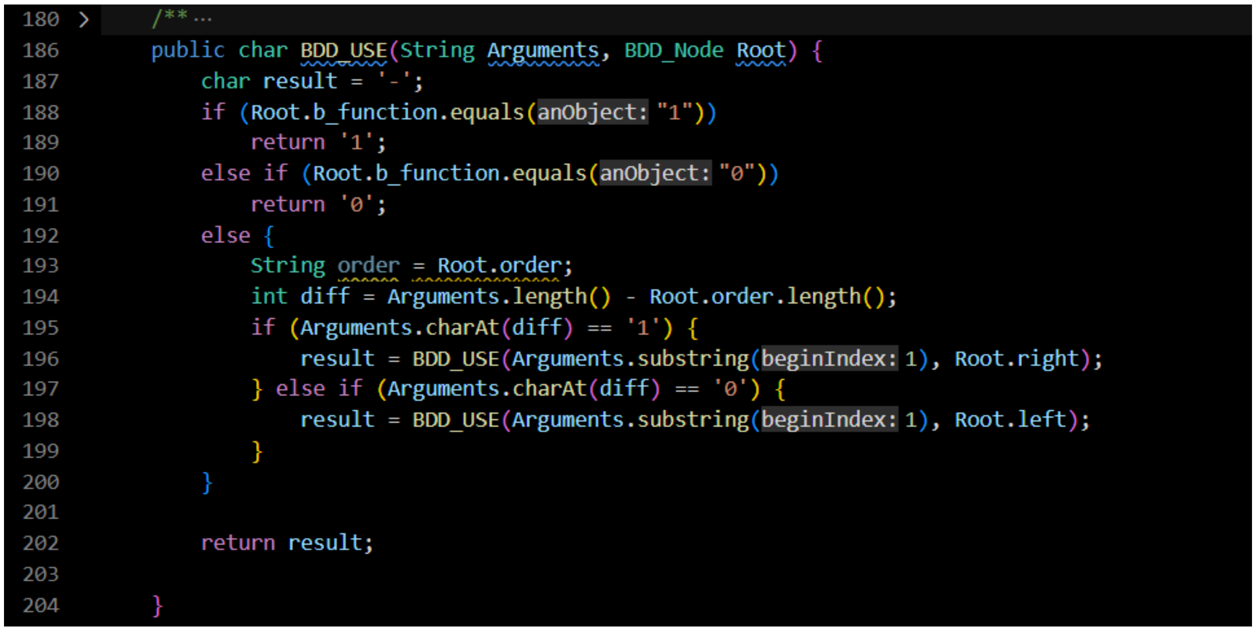
Inside creations od BDD also the reductions happens.

BDD_use(String Arguments, BDD_Node Root)

This function recursively goes through our BDD tree to find the result of Bfunction if we assign variables with such a values.

1 argument is String of values for our variables. For example: “1101”, when order = “ABCD”

It returns char ‘1’ or ‘0’ as a result. If error occurs, it returns ‘–‘.



```
    /**...
    public char BDD_USE(String Arguments, BDD_Node Root) {
        char result = '-';
        if (Root.b_function.equals(anObject: "1"))
            return '1';
        else if (Root.b_function.equals(anObject: "0"))
            return '0';
        else {
            String order = Root.order;
            int diff = Arguments.length() - Root.order.length();
            if (Arguments.charAt(diff) == '1') {
                result = BDD_USE(Arguments.substring(beginIndex: 1), Root.right);
            } else if (Arguments.charAt(diff) == '0') {
                result = BDD_USE(Arguments.substring(beginIndex: 1), Root.left);
            }
        }

        return result;

    }
```

DNF class:

I have implemented DNF class, that helps substitute variables in DNF function.

Here functions with explanation:

```
/**
     * Returns String DNF from Array of conjunctions
     *
     * @param stringArray
     * @return
     *          Example :: "AB", "AC", "BC" ==> "AB+AC+BC"
     */
    private static String ConjunctionArrayToDNF(String[] stringArray)
```

```
/**
     * Returns Array of conjunctions without duplicates
     *
     * @param conjunction
     * @return
     *          Example :: "AB", "A", "AC", "BC", "A", "AB" ==> "AB", "A", "AC",
"BC"
     */
    private static String[] DeleteDuplicates(String[] conjunction)
```

```
/**
     * Gets conjuction and make it prettier -
     * Removes duplicates of variables, put variables in specified order
     *
     * @param b_function
     * @param order
     * @return
     *          Example :: BBCCAA ==> ABC (if order = "ABC")
     */
    private static String PrettyConjunction(String b_function, String order)
```

```
/**
     * Gets Array of conjunctions and make them prettier -
     * Removes duplicates of variables, put variables in specified order, put
bigger
     * conjunction closer to start of array
     *
     * @param conjunction
     * @param order
     * @return
     *        Example :: BAC+CCA+DDD+!A!B!A ==> ABC+AC+!A!B+D (if order =
"ABCD")
     */
    private static String[] Pretty(String[] conjunction, String order)
```

```
 /**
     * Return DNF with -
     * substituted variable
     * removed duplicates
     * in specified order
     * bigger conjunctions closer to start
     *
     * @param state
     * @param letter
     * @param Bfunction
     * @param order
     * @return
     *        Example :: "AB+!AB+AAAC+BCB+!B!A+!A" (A = 1) ==> "BC+B+C+!B"
     */
    public static String SubstituteVariable(boolean state, String letter, String
Bfunction, String order)
```

```
 /**
     * Returns uniq ID that represents
     *
     * @param b_func
     * @return
     */
    public static BigInteger HashCode(String b_func, String order)
```

```
    /**
     * Returns result of substitution of variables in DNF function- "0" or "1"
     *
     * @param State     : array of variables value - "1010"
     * @param Bfunction : DNF function - "A!B+CD+!AD"
     * @param Order     : Varibles thas appears in Bfunction - "ABCD"
     * @return        "1" / "0"
     */
    public static char SubstituteAllVariables (String State, String Bfunction,
String Order)
```

```
/**
    * Returns random DNF - without repeats, in Alphabet order
    *
    * @param Alphabet          - Letter that will be used in DNF
    * @param ConjunctionCount - Count of conjunctions (random by default)
    * @return String DNF
    * @Example "ABC+!AD+BD+!AC"
    */
   public static String GenerateDNF(String Alphabet, Integer ConjunctionCount)
```

BDD_tree class

```
/**
     * Creates new level of BDD_Tree, using Table, to reduse repaeats
     *
     * @param lvl
     * @param current
     * @param Table
     * @param Root
     */
    private void createLvl(int lvl, int current, KeyValue[] Table, BDD_Node Root)
```

```
 /**
     * Returns new or existing BDD_Node in Table
     *
     * @param Table
     * @param Bfunction
     * @param Order
     * @param Letter
     * @return
     */
    private BDD_Node insertTable(KeyValue[] Table, String Bfunction, String
Order, String Letter)
```

```
 public void PrintTree()
```

```
 /**
     *
     * @param Arguments
     * @param Root
     * @return
     */
    public char BDD_USE(String Arguments, BDD_Node Root)
```

BDD_Node class

```java
/**
 * This class represents BDD Tree Node with:
 * Boolean function
 * Letter what will be substituted
 * Order of variables - "ABCD"
 * Left and Right pointers to new Nodes with smaller function
 */
public class BDD_Node{
    public String b_function;
    public String letter;
    public BDD_Node right;
    public BDD_Node left;
    public String order;

    BDD_Node(String b_func, String letter, String order){
        this.letter = letter;
        this.b_function = b_func;
        this.right = null;
        this.left = null;
        this.order = order;
    }
}
```

KeyValue class

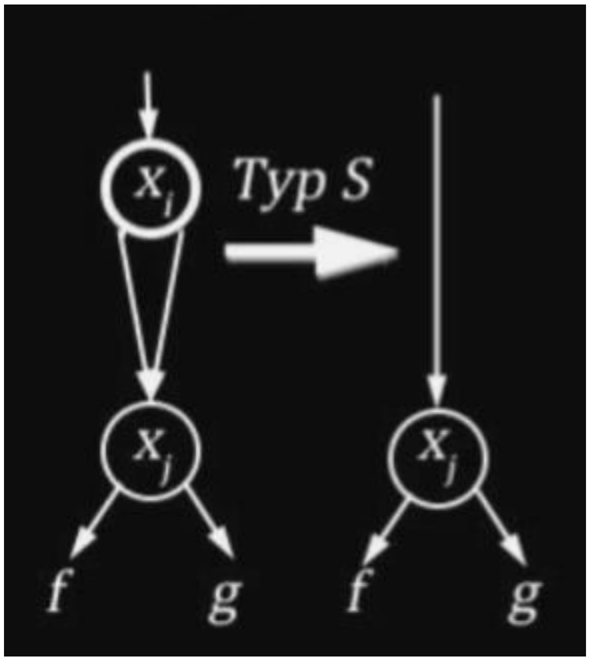
```
/**
 * This class represents basic object KEY:VALUE idea
 * KEY : hash of Boolean Function
 * VALUE : Pointer to BDD_Node that represents Boolean function
 */
public class KeyValue{
    private BigInteger hash;
    public BDD_Node Node;
```

Reduction:

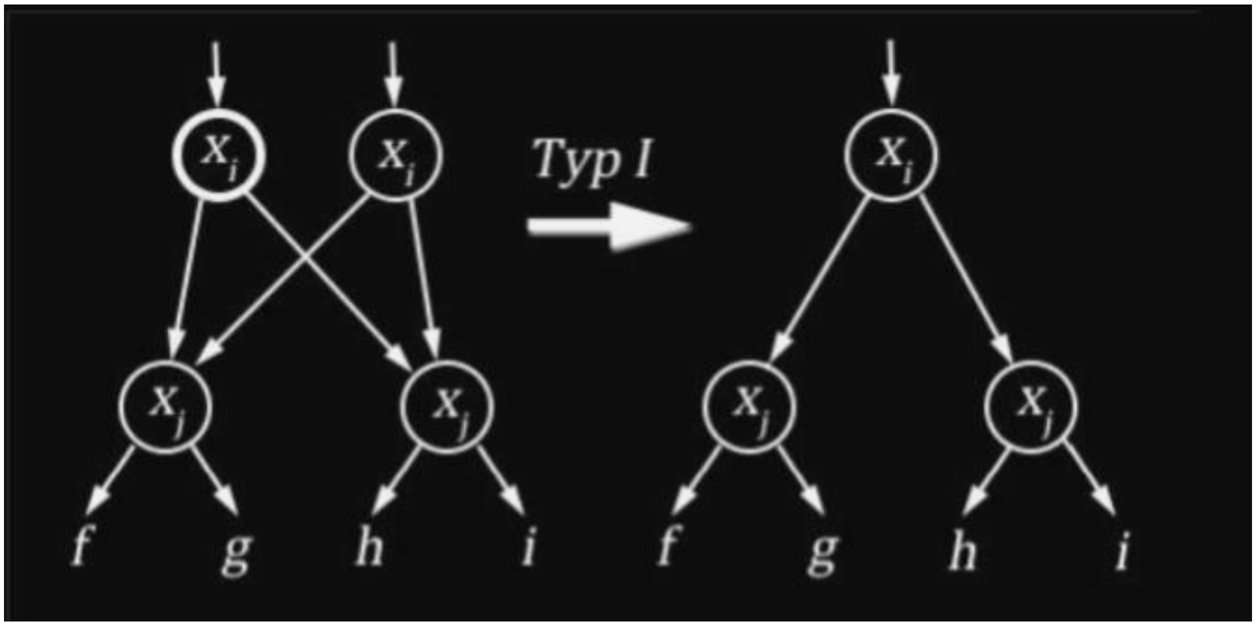
How my reduction works?

First of all I define 2 types of reduction

<u>Vertical :</u> when we reduce nodes that repeats one below other



<u>Horizontal :</u> when we reduce Nodes that repeats in one level of BDD_Tree



For horizontal reduction I have implemented hashObjects.

Hash Object is a KEY:VALUE data structure where:

KEY : unique code of bdd function. For example - 432

VALUE: Node that represents that function. For example – “AB+BC+!A!D”

So, when we create level, we push new Nodes in array of Nodes, which called Table[]

When we create single Node, we try to push it in Table[],

IF there is no same HashCode, than we should create new Node

Else if there is the same Node, we just return pointer to existed one, so we didn’t created duplicate. (link)

For vertical reduction I have implemented algorithm that add new Node to the level Table[]

Only if this new Node will contain letter of the level. And don’t add new Node to the level Table[] when there is no letter of level.

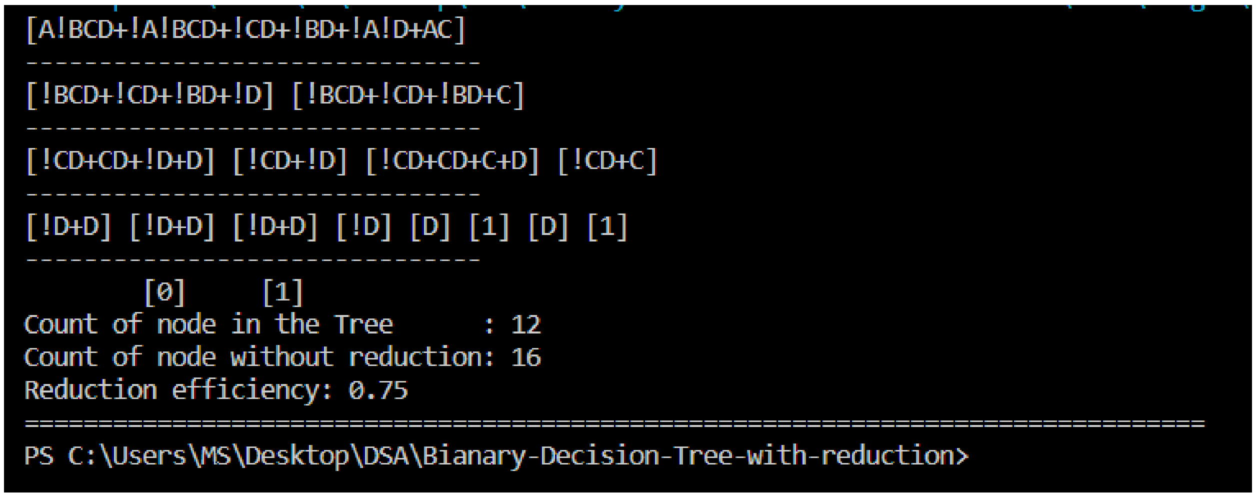
For example – we create level with letter “C”. Algorithm tries to create left node for function

“BD+!DFG+!FG”, if we substitute letter ”B” with 0, then we will have new “!DFG+!FG” function

But is didn’t contain letter “C”, so we shouldn’t create tis node, we will try to add this node on other level.

Testing

Printing BDD Tree:



```
[A!BCD+!A!BCD+!CD+!BD+!A!D+AC]
-------------------------------
[!BCD+!CD+!BD+!D] [!BCD+!CD+!BD+C]
-------------------------------
[!CD+CD+!D+D] [!CD+!D] [!CD+CD+C+D] [!CD+C]
-------------------------------
[!D+D] [!D+D] [!D+D] [!D] [D] [1] [D] [1]
-------------------------------
        [0]     [1]
Count of node in the Tree       : 12
Count of node without reduction: 16
Reduction efficiency: 0.75
=================================================================================
PS C:\Users\MS\Desktop\DSA\Bianary-Decision-Tree-with-reduction>
```

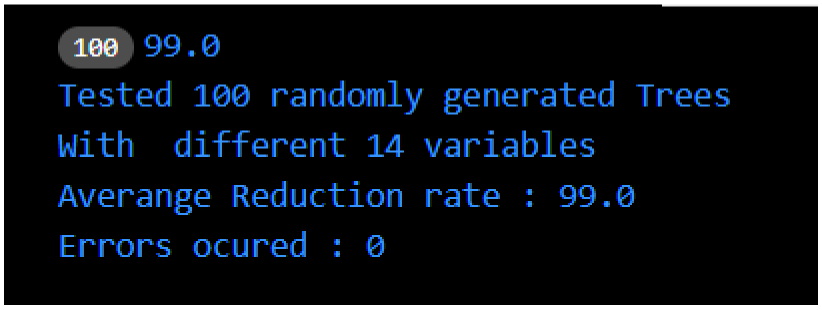
Errors:

There is class TestApp that implements error testing of BDD Tree

It creates 1000 of random trees and tests every tree with all possible values input in BDD_use.

After than it compares result of BDD_use with alternative result to prove that BDD_Tree and BDD_Use work properly.

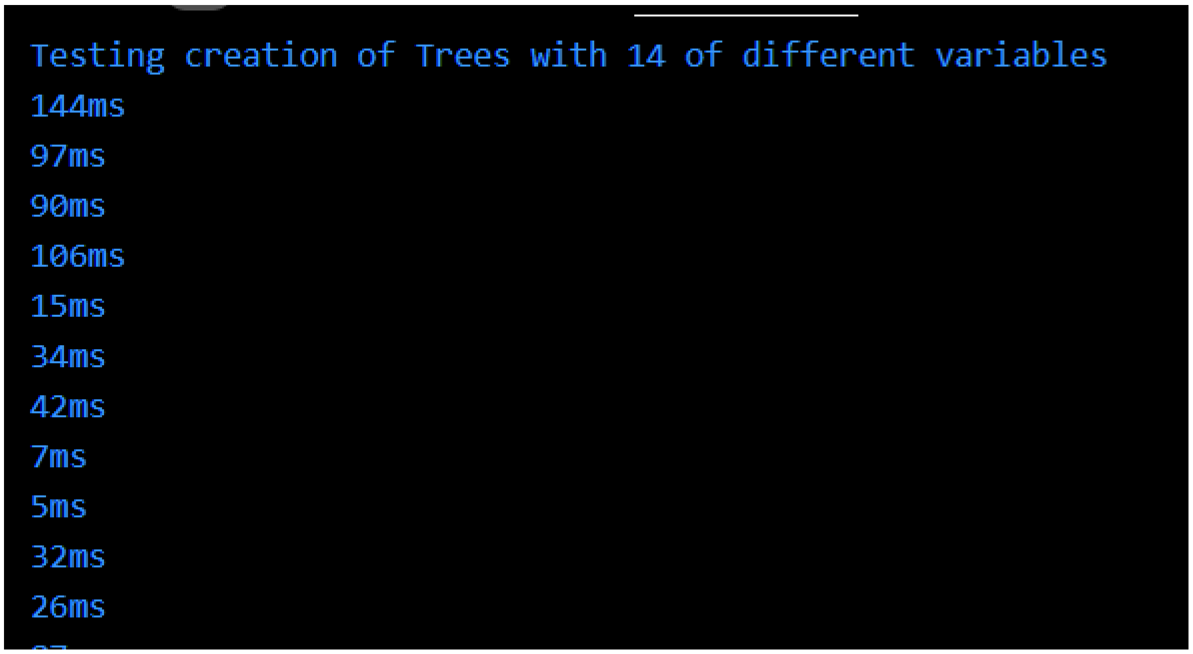
After testing of 1000 trees, there was no mistakes. That means that BDD USE and BDD_create work properly



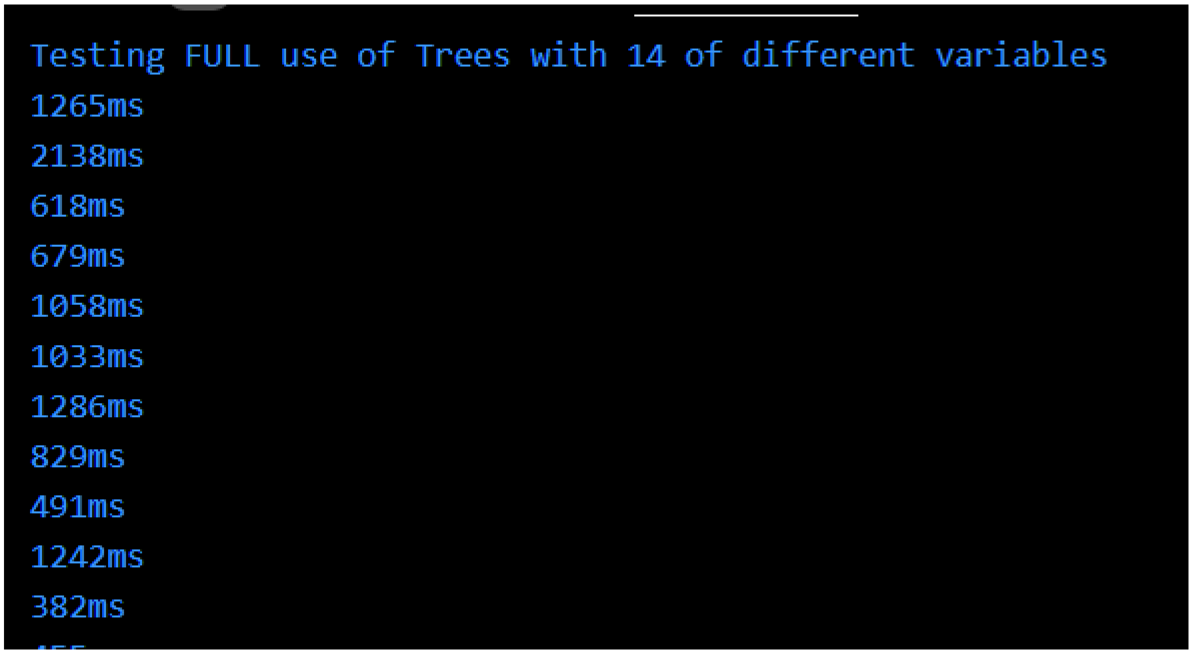
```
100 99.0
Tested 100 randomly generated Trees
With  different 14 variables
Averange Reduction rate : 99.0
Errors ocured : 0
```

(Program prints % of mistakes, and mistake if that happens)

Time complexity:

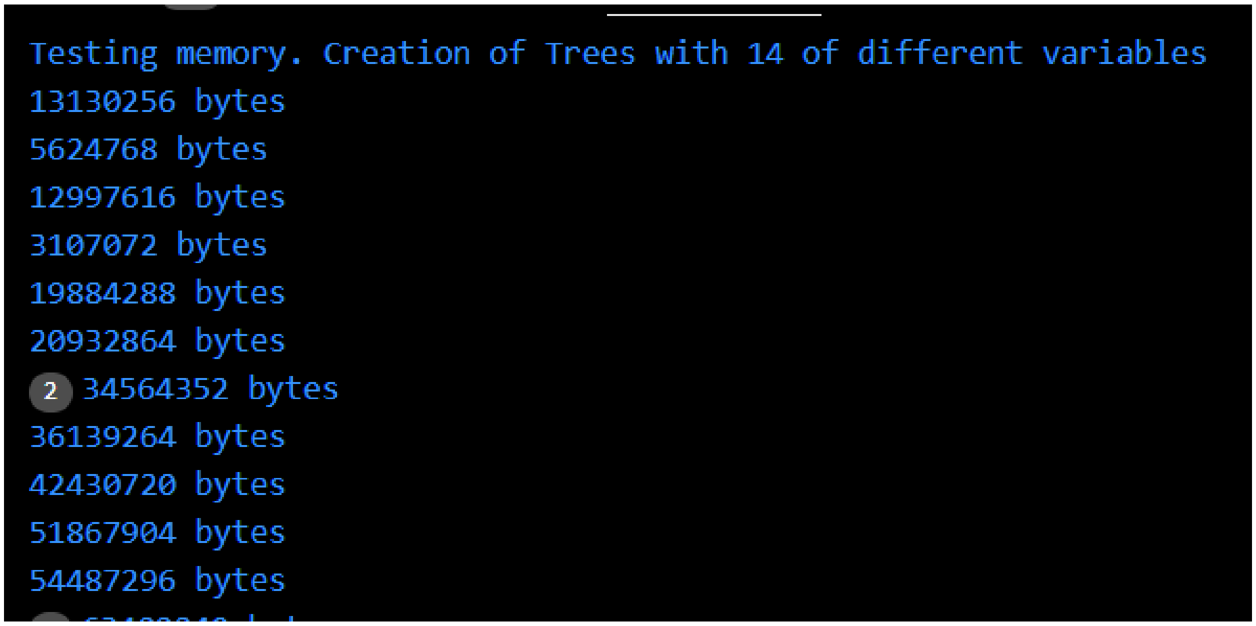


```
Testing creation of Trees with 14 of different variables
144ms
97ms
90ms
106ms
15ms
34ms
42ms
7ms
5ms
32ms
26ms
```



```
Testing FULL use of Trees with 14 of different variables
1265ms
2138ms
618ms
679ms
1058ms
1033ms
1286ms
829ms
491ms
1242ms
382ms
```

Memory tests:



```
Testing memory. Creation of Trees with 14 of different variables
13130256 bytes
5624768 bytes
12997616 bytes
3107072 bytes
19884288 bytes
20932864 bytes
2 34564352 bytes
36139264 bytes
42430720 bytes
51867904 bytes
54487296 bytes
```