УДК 303.732.4

СРАВНЕНИЕ СПОСОБОВ РЕАЛИЗАЦИИ ДЕРЕВА КЛАССИФИКАЦИИ ДЛЯ ПРЕДМЕТНОЙ ОБЛАСТИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫМИ ЯЗЫКАМИ RUBY И LAZARUS

О.Б. ПОПОВА

Кубанский государственный технологический университет,
350002, Российская Федерация, Краснодар, ул. Московская, д. 2,
электронная почта: popova_ob@mail.ru

Сегодня актуально решать задачи классификации предметной области, для которых составляются современные структуры данных и подбираются эффективные методы реализации этих структур. В статье сравниваются способы реализации дерева классификации с помощью языков ruby и Lazarus. Рассмотрены способы описания самой структуры, блок схемы и алгоритмы вставки, алгоритмы обхода дерева. Установлено, что язык ruby позволяет получить простой и понятный код, в котором можно избежать лишних команд и переменных, но блок-схема и алгоритм метода insert составлен менее эффективно, чем в языке Lazarus. Поэтому можно считать язык Lazarus для реализации дерева классификации наиболее эффективным. Так как вместе с эффективным способом реализации основных алгоритмов, он обладает простым и понятным описанием, которое соответствует основным определениям из теории бинарных деревьев, рекурсивного перехода по ним и работы с динамическими структурами.

Ключевые слова: бинарное дерево, Ruby, предметная область, дерево классификации, Lazarus.

Сегодня актуальной задачей является описание предметной области, используя современные методы классификации. Для этого изучают процесс выбора «наиболее подходящего» метода [1-7], чтобы получить структуру данных, которую можно использовать в интеллектуальных информационных системах [8]. Поповой была получена новая структура данных – бинарное дерево системы вопросов и ответов [9], в основе которой используется бинарное дерево.

Для получения эффективно работающей интеллектуальной информационной системы [10] пришлось провести сравнение способов реализации деревьев классификации в языках программирования разного типа, так как от правильного выбора способа реализации будет зависеть эффективности работы программного продукта. Так как дерево классификации в современном понимании этой структуры – это бинарное дерево,

реализованное и используемое с помощью рекурсии, то необходимо при сравнении способов реализации обратить особое внимание на элементы, которые описывают эту структуру, формируют её и осуществляют перемещение по ней.

В теории структур данных используют следующие определения важных элементов, участвующих в реализации любой структуры данных:

- абстрактный тип данных, который включает описание структуры и операторы, которые с ней работают;
- способы реализации этого типа данных внутри языка программирования, которые определяются алгоритмами, используемыми внутри этих операторов.

Поэтому для выявления наиболее эффективной реализации среди этих двух языков объектно-ориентированного программирования – Ruby и Lazarus, необходимо сравнить самые эффективные способы реализации принятые в этих языках между собой. Для этого опишем структуру данных и основные методы, которые работают с ней:

- метод добавления элементов в дерево – insert;
- метод обхода дерева в симметричном порядке.

Сравнивать способы реализации будем по простоте представления кода и количеству строк, а так же блок-схемы, которые позволят определить затраты на следующие важные операции:

- сравнения;
- работу с памятью.

Сначала рассмотрим способ описания структуры бинарного дерева на языке ruby [11] для **class Tree** (рис. 1):

```
 # Описание полей ячейки
1 attr_accessor :left    # указатель на левого сына
2 attr_accessor :right   # указатель на правого сына
3 attr_accessor :data    # значения вносимого в дерево данного
 # Инициализация пустой ячейки указательного типа
4 def initialize(x=nil)
5  @left = nil
6  @right = nil
7  @data = x
8 end
```

Рисунок 1 – Описание типа данных бинарное дерево на языке ruby.

Как видно из рисунка 1, описать указательный тип для каждой ячейки дерева и тип самого дерева в этом языке можно очень просто, перечислив последовательно все поля одной ячейки дерева (строки 1 – 3). Одновременно с этим происходит инициализация пустой ячейки указательного типа, в которой заранее заполняются ячейки (строки 4 – 8), тем самым уменьшая в дальнейшем количество операций при создании и присоединении нового элемента в дереве.

Теперь рассмотрим представление типа данных «дерево» в языке Lazarus (рис. 2).

```
 {Описание сложного типа, который является указателем на ячейку}
1  TREE=^pTREE;
 {Описание строения ячейки, на которую указывает указатель типа TREE}
2  pTREE=record
3      left:TREE;   {указатель на левого сына}
4      right:TREE;  {указатель на правого сына}
5      data:integer; {значения вносимого в дерево данного}
6    end;
7  var A:TREE; {Инициализация ячейки, которая хранит значение текущего
                указателя на ячейку составного типа pTREE}
```

Рисунок 2 – Описание типа данных бинарное дерево на языке Lazarus.

Как видно из рисунка 2, тип описывает подробно структуру каждой ячейки дерева, а так же хорошо визуализирует её описание на физическом уровне, что характеризует такое представление с положительной стороны. Такое описание позволяет объяснить, каким образом соединяются элементы в дереве, учитывая соответствие типов, когда ячейка хранящая указатель на следующий элемент имеет тот же тип, что и этот указатель.

В отличие от предыдущего примера, инициализация ячейки в описании типа здесь не производится. Поэтому создание и присоединение новой ячейки происходит каждый раз с использованием стандартного набора команд, что увеличивает код программы. Это можно считать недостатком, хотя его можно устранить, введя процедуру создания нового пустого элемента – ***empty element***. Эта процедура должна так же заполнять ячейки ***left*** и ***right***, присваивая им **nil**. После чего присоединять её к узлу, на который указывает текущий указатель дерева.

Проанализировав оба примера можно сделать вывод, что очень хорошими идеями у данных реализаций является:

- понятная визуализация типа данных, отражающая структуру дерева подробно на концептуальном уровне;
- представление способа создания пустого элемента.

Перейдём теперь к рассмотрению метода добавления элементов в структуру данных дерево. Сначала рассмотрим коды программ, а потом сравним блок-схемы, описывающие их алгоритмы.

Рассмотрим метод ***insert*** на языке программирования ruby (рис. 3).

```
1 def insert(x)
# Если инициализированный узел (корень или лист) пустой, то запишем x
в поле данных data.
2   if @data==nil @data=x
# Если вводимое значение в дерево меньше значения текущего узла, то
выполнить следующие действия.
3      elsif x<=@data
# Если у текущего узла нет слева сына, то инициализировать новую
ячейку, записать в неё x и присоединить к текущему узлу слева.
4              if @left==nil    @left=Tree.new x
# У текущего узла есть слева сын, поэтому перемещаемся влево и ищем
дальше место вставки, запуская рекурсию.
5                          else @left.insert x   end
# Если вводимое значение в дерево больше значения текущего узла и у него
нет справа сына, то инициализировать новую ячейку, записать в неё x и
присоединить к текущему узлу справа.
6       else   if @right==nil   @right=Tree.new x
# Если у текущего узла есть справа сын, то перемещаемся вправо и ищем
дальше место вставки, запуская рекурсию.
7                          else @right.insert x   end   end
# Завершение insert
8 end
```

Рисунок 3 – Метод ***insert***, который добавляет новый элемент в дерево.

Необходимо заметить (см. рис. 3), в методе используется не явно текущий указатель на узел, к которому добавляется элемент, и указатель на добавляемую ячейку. Для доступа к полям текущей ячейки необходимо всего лишь записать ***@data*** (поле с данными), ***@left*** или ***@right*** (поля с указателями на левого и правого сына). Тогда результатом команды ***Tree.new x*** будет новая предзаполненная ячейка (смотри описание типа рис. 1) и значение указателя на неё. Следовательно, не нужно вводить дополнительных переменных, достаточно записать ***@left = Tree.new x*** (смотри строку 4 кода рис. 3) или ***@right = Tree.new x*** (смотри строку 6 кода рис. 3). Из-за такого представления легче осуществлять рекурсивный переход. Достаточно выполнить команду ***@left.insert x или @right.insert x***, чтобы заново стал выполняться алгоритм метода insert, в котором уже использовался указатель на левого сына (смотри строку 5 кода рис. 3) или на правого сына (смотри строку 7 кода рис. 3) от предыдущего текущего узла. Всё это положительным образом отражается на эффективности использования данного способа реализации.

Для начала работы с деревом требуется объявить переменную ***tree*** и присвоить ей указатель на корень дерева ***Tree.new***, то есть на первый созданный элемент:

tree = Tree.new,

поэтому запись ***tree.insert (x)*** передаст методу значение текущего указателя на корень дерева. Итого для работы с бинарным деревом требуется всего одна переменная tree, которая хранит указатель на корень дерева, что улучшает эффективность работы программы.

Теперь опишем код процедуры добавления элемента в дерево ***insert***, которая составлена на языке Lazarus (смотри рисунок 4).

```
1 procedure insert (x: integer; var A: Tree);
{Если узла (корня или листа) нет, то инициализируем узел и заполним его данными.}
2 begin if A=nil then begin new (A); A^.data:=x; A^.left:=nil; A^.right:=nil; end
{Если вводимое значение в дерево меньше значения текущего узла, то перемещаемся влево и ищем дальше место вставки, запуская рекурсию.}
3                 else if x<A^.data then insert (x, A^.left)
{Если вводимое значение в дерево больше значения текущего узла, то перемещаемся вправо и ищем дальше место вставки, запуская рекурсию.}
4                 else if x>A^.data then insert (x, A^.right);
{Завершение insert.}
5 end;
```

Рисунок 4 – Процедура ***insert***, которая добавляет новый элемент в дерево

Способ описания процедуры в этом языке предполагает подробную запись тех данные, которые передаются процедуре и отдаются ею программе (смотри строки 1, 3, 4 кода программы рис. 4). Синтаксис языка и его концепция представления указателей предполагает использование текущего указателя A, которые может указывать на рассматриваемый в данный момент программы узел. Такая реализация процедуры рекурсивной вставки в дерево insert позволяет получить очень короткий понятный код алгоритма добавления элемента в дерево, состоящий всего лишь из трёх строк – 2, 3 и 4 см. рисунок 4. Недостатком является организация последовательного создания пустой ячейки, заполнения её указательных полей, поля с данными и присоединения её к найденному месту в дереве, которое состоит из четырёх последовательно выполняемых действий (см. строку 2 рисунка 4):

- new (A); {Создание пустой ячейки}
- A^.data:=x; {Запись в поле data значения добавляемого в дерево узла}
- A^.left:=nil; {Запись в поле ***left*** служебного слова ***nil***}
- A^.right:=nil. { Запись в поле ***right*** служебного слова ***nil***}

Если одновременно A^.left:=nil и A^.right:=nil, то добавленный узел является новым листом дерева.

Тогда как в ruby эти четыре последовательные команды можно описать одной из команд:

- ***@left = Tree.new x***

- ***@right = Tree.new x***

Теперь сравним блок-схемы алгоритмов insert добавления элемента в дерево, составленных в ruby (см. рис. 5) и Lazarus (см. рис. 6).

Как видно из рисунка 5, в блок схеме метода ***insert*** языка ruby число условных операторов четыре, блоков обращения к памяти пять, среди которых 1, 2 и 4 блоки однотипные. Эти блоки можно было бы объединить в один, который добавлял бы элемент в найденное место в дереве. Но в языке ruby при добавлении элемента в дерево поддерживается концепция сначала создания корня, потом добавления элемента в левое поддерево узла или в правое поддерево узла, что снижает эффективность алгоритма добавления элемента в дерево, не смотря на сокращения кода программы после введения инициализации пустого элемента (см. строки 4-8 рис. 1).

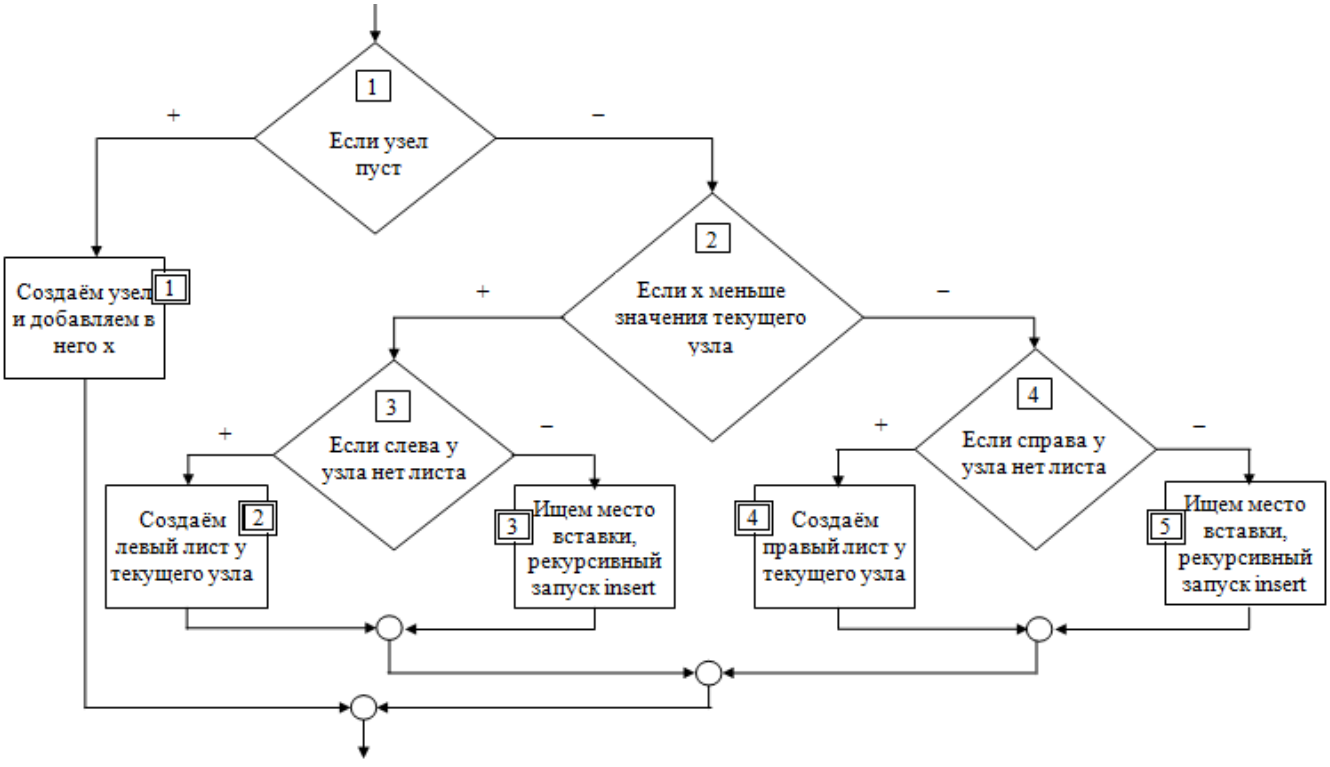

Рисунок 5 – Блок-схема алгоритма ***insert*** добавления элемента в дерево, составленная в ruby

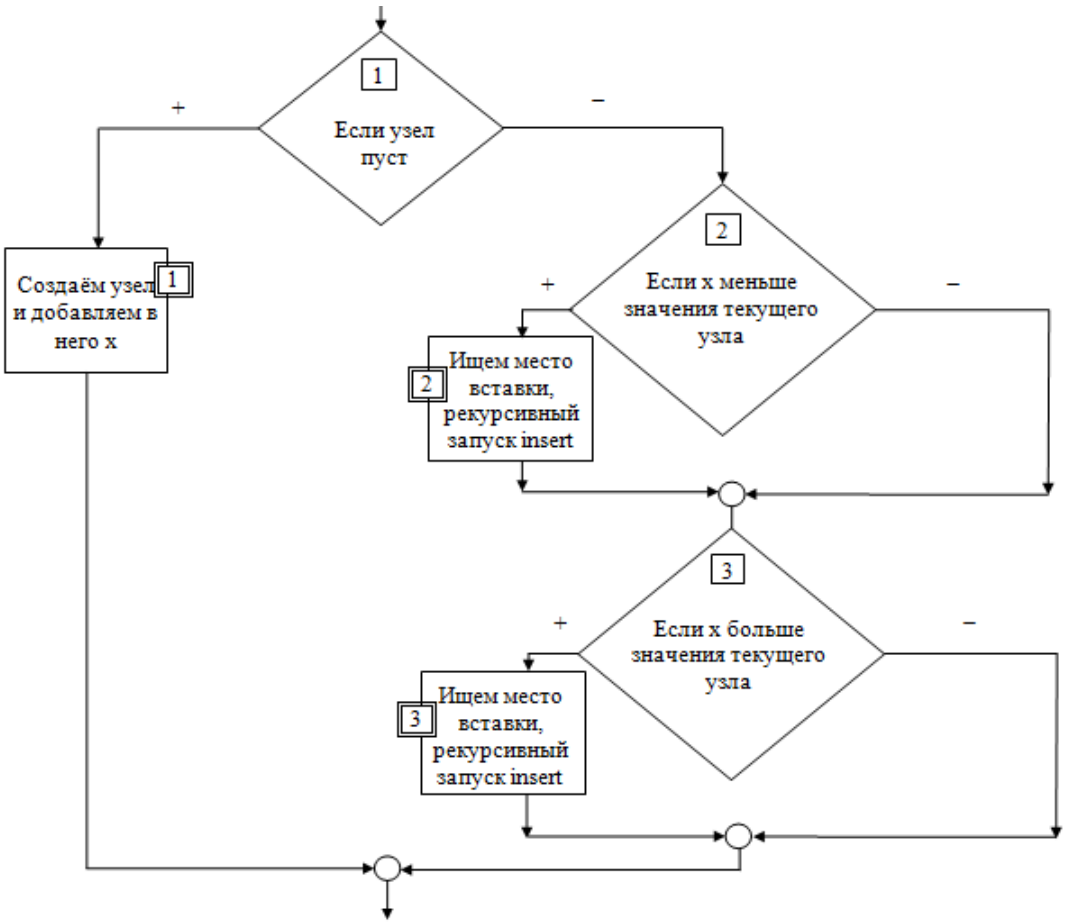

Рисунок 6 – Блок-схема алгоритма ***insert*** добавления элемента в дерево, составленная в Lazarus

Блок-схема рисунка 5 представляет более эффективный и понятный алгоритм добавления элемента в дерево. Он максимально точно отражает концепцию рекурсивного добавления элементов в дерево, когда сначала ищется в дереве нужное место вставки для элемента, а потом производится сама вставка. Как видим, число условных блоков три. Число блоков обращения к памяти то же три. Поэтому, при каждом проходе по дереву от корня, до искомого места вставки в дерево будет выполняться число условных операторов на один меньше, чем в предыдущем алгоритме. Следовательно, данный алгоритм можно считать более эффективным.

Приведём теперь сравнения двух алгоритмов симметричного обхода деревьев, реализованных на языке ruby и Lazarus (см. рис. 7 и 8).

```
1 def inoder()
2  @left.inoder {|y| yield y} if @left != nil  # Если узел имеет левого сына,
                    то происходит рекурсивный переход в левое поддерево.
3   yield @data  # Вывод значения текущего узла из поддерева.
4  @right.inoder {|y| yield y} if right != nil  # Если узел имеет правого сына,
                    то происходит рекурсивный переход в правое поддерево.
5 end  # Завершение работы метода симметричного обхода
```

Рисунок 7 – Метод симметричного обхода дерева ***inoder*** на языке ruby

```
1 procedure inoder(p:TREE);
2  begin
3   if p<>nil then inoder(p^.left); {Если узел имеет левого сына,
                    то происходит рекурсивный переход в левое поддерево.}
4   writeln(p^.data); {Вывод значения текущего узла из поддерева.}
5   if p<>nil then inoder(p^.right); {Если узел имеет правого сына,
                    то происходит рекурсивный переход в правое поддерево.}
6  end; {Завершение работы метода симметричного обхода}
```

Рисунок 8 – Процедура симметричного обхода дерева ***inoder*** на языке Lazarus

Коды этих алгоритмов симметричного обхода (см. рис. 7 и 8) имеют одинаковую эффективность, так как позволяют в рамках концепций этих языков организовать лаконичный и понятный код обхода дерева, где максимально эффективно организована рекурсия и вывод нужного текущего узла в список обхода.

Отличаются лишь концепции этих языков. Для рекурсивного обращения к поддеревьям общего дерева на языке ruby, которое состоим из корня и двух узлов, используется передача управления конкретному блоку программы оператором yield, после выполнения которого управление передаётся к основному коду программы:

- @left.inoder {|y| **yield y**} if @left != nil # Строка 2 кода рисунка 7
- @right.inoder {|y| **yield y**} if right != nil # Строка 4 кода рисунка 7
- **yield @data** # Строка 3 кода рисунка 7.

В языке Lazarus используется стек предыдущих значений переданных процедуре inoder(), который позволяет перемещаться по дереву, так как в нём хранятся посещённые процедурой обхода inoder() корни поддеревьев. Достаточно передать рекурсивной функции ***p^.left*** или ***p^.right***, чтобы потом в

теле процедуры проверить на неравенство переменную – ***p<>nil***, а в стеке останется указатель на текущую ячейку.

Таким образом, можно сделать вывод, что реализация бинарного дерева на языке Lazarus более эффективна, чем на языке ruby, хотя в нём есть много интересных идей:

- простое описание указательного типа с инициализацией пустой ячейки, которая позволяет проще соединять новую ячейку с деревом, сократив программный код;
- отсутствие лишних переменных за счёт особого представления метода insert;
- синтаксис языка позволяет убрать лишние переменные, использовать простые коды рекурсивного перехода, вызова методов и обращения к дереву.

По отношению к структуре данных «бинарное дерево» язык ruby полностью выполнил ту цель, которую заложил в этот язык его создатель Юкихиро Мацумото – «создание простых и в то же время понятных программ, где важна не скорость работы программы, а малое время разработки, понятность и простота синтаксиса» [12].

Работа выполнена при финансовой поддержке Российского гуманитарного научного фонда № 16-03-00382 от 18.02.2016 в рамках темы “Мониторинг исследовательской деятельности образовательных учреждений в условиях информационного общества”.

ЛИТЕРАТУРА

1. Попова О.Б. Системный подход к исследованию процесса оптимизации. Деп. в ВИНИТИ №83-В2010 от 17.02.2010.

2. Попова О.Б. Структура технической системы процесса выбора метода оптимизации. Деп. в ВИНИТИ №243-В2012 от 25.05.2012.

3. Попова О.Б. Системный анализ и управление процессом оптимизации. Деп. в ВИНИТИ №256-В2010 от 07.05.2010.

4. Попова О.Б. Участие процесса оптимизации в развитии сложных технических систем. Деп. в ВИНИТИ №257-В2010 от 07.05.2010.

5. Попова О.Б., Попов Б.К. Связи в исследуемой системе процесса оптимизации. Деп. в ВИНИТИ №112-В2012 от 22.03.2012.

6. Попова О.Б., Попов Б.К. Анализ процесса оптимизации. Определение понятий. Деп. в ВИНИТИ №111-В2012 от 22.03.2012.

7. Попова О.Б., Попов Б.К., Ключко В.И. Анализ связей в реальной и технической системах процесса оптимизации // Международный журнал экспериментального образования. – 2013. – №10-2. – С. 405-408.

8. Системный анализ процесса выбора метода оптимизации информационной системы: монография / О.Б. Попова, Б.К. Попов, В.И. Ключко; ФБГОУ ВПО «Кубан. гос. технол. ун-т». – Краснодар: Издательский Дом – Юг, 2012 – 135 с.

9. Popova O., Popov B., Karandey V., Evseeva M. Intelligence amplification via language of choice description as a mathematical object (binary tree of question-answer system) // Procedia – Social and Behavioral Sciences.-2015.-V. 214.-C. 897–905.

10. Попова О.Б., Попов Б.К. Интеллектуальная информационная система выбора «Оптимэль». Патент на изобретение RUS № 2564641 от 27.05.2014.

11. Книга: Программирование на языке Ruby. 9.3.1. Реализация двоичного дерева, 2016. http://wm-help.net/lib/b/book/827961078/267

12. Ruby. Материал из Википедии – свободной энциклопедии, 2016. https://ru.wikipedia.org/wiki/Ruby

REFERENCES

1. Popova O.B. Sistemnyy podkhod k issledovaniyu protsessa optimizatsii. Dep. v VINITI №83-V2010 ot 17.02.2010.

2. Popova O.B. Struktura tekhnicheskoy sistemy protsessa vybora metoda optimizatsii. Dep. v VINITI №243-V2012 ot 25.05.2012.

3. Popova O.B. Sistemnyy analiz i upravlenie protsessom optimizatsii. Dep. v VINITI №256-V2010 ot 07.05.2010.

4. Popova O.B. Uchastie protsessa optimizatsii v razvitii slozhnykh tekhnicheskikh sistem. Dep. v VINITI №257-V2010 ot 07.05.2010.

5. Popova O.B., Popov B.K. Svyazi v issleduemoy sisteme protsessa optimizatsii. Dep. v VINITI №112-V2012 ot 22.03.2012.

6. Popova O.B., Popov B.K. Analiz protsessa optimizatsii. Opredelenie ponyatiy. Dep. v VINITI №111-V2012 ot 22.03.2012.

7. Popova O.B., Popov B.K., Klyuchko V.I. Analiz svyazey v realnoy i tekhnicheskoy sistemakh protsessa optimizatsii // Mezhdunarodnyy zhurnal eksperimentalnogo obrazovaniya. – 2013. ˜ №10-2. – S. 405-408.

8. Sistemnyy analiz protsessa vybora metoda optimizatsii informatsionnoy sistemy: monografiya / O.B. Popova, B.K. Popov, V.I. Klyuchko; FBGOU VPO «Kuban. gos. tekhnol. un-t». – Krasnodar: Izdatelskiy Dom – Yug, 2012 – 135 s.

9. Popova O., Popov B., Karandey V., Evseeva M. Intelligence amplification via language of choice description as a mathematical object (binary tree of question-answer system) // Procedia – Social and Behavioral Sciences.-2015.-V. 214.-S. 897–905.

10. Popova O.B., Popov B.K. Intellektualnaya informatsionnaya sistema vybora «Optimel». Patent na izobretenie RUS № 2564641 ot 27.05.2014.

11. Kniga: Programmirovanie na yazyke Ruby. 9.3.1. Realizatsiya dvoichnogo dereva, 2016. http://wm-help.net/lib/b/book/827961078/267

12. Ruby. Material iz Vikipedii – svobodnoy entsiklopedii, 2016. https://ru.wikipedia.org/wiki/Ruby

THE COMPARISON OF THE WAYS OF IMPLEMENTATION THE CLASSIFICATION TREE OF THE SUBJECT AREA BY THE OBJECT-ORIENTED LANGUAGES RUBY AND LAZARUS

O.B. POPOVA

Kuban State Technological University,
2, Moskovskaya st., Krasnodar, Russian Federation, 350002,
e-mail: popova_ob@mail.ru

Today it is urgent to solve the problem of classification of the subject areas for which compiled modern data structure and selected effective methods of implementation of these structures. The article compares the ways of implementing the classification tree using ruby and Lazarus languages. The methods of describing the structure itself, block diagrams and insert algorithms, traversal algorithms. It is established that the language ruby provides a clean and simple code in which you can avoid unnecessary commands and variables, but a block diagram of the algorithm and insert method is made less effective than in the language of Lazarus. Therefore we can assume Lazarus language to implement the most effective classification tree. Since, together with an effective way to implement the basic algorithms, it has a simple and clear description that is consistent with the basic definitions of the theory of binary tree recursive transition for him and work with dynamic structures.

Key words: binary tree, Ruby, subject area, classification tree, Lazarus.