

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

**ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ РЕАЛИЗАЦИИ
НЕКОТОРЫХ СТРУКТУР ДАННЫХ НА ЯЗЫКЕ ERLANG**

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
В. В. Быцюка

Научный руководитель:
старший преподаватель
В. Н. Брагилевский

Ростов-на-Дону
2016

Постановка задачи

1. Реализовать структуру данных «упорядоченное множество» на языке программирования Erlang.
2. Сравнить время выполнения основных операций реализованной структуры данных с реализациами из модулей `ordsets` и `sets`.

Содержание

Постановка задачи	2
Введение	5
1. Обзор используемых технологий и алгоритмов	6
1.1. Erlang	6
1.1.1. Переменные и атомы	6
1.1.2. Кортежи	6
1.1.3. Списки	7
1.1.4. Функции	7
1.2. Красно-черные деревья	7
2. Реализации	10
2.1. Структура дерева	10
2.2. Вставка и удаление	10
2.2.1. Вставка	10
2.2.2. Удаление	13
2.3. Логические функции	17
2.3.1. Принадлежность элемента множеству	18
2.3.2. Является ли одно упорядоченное множество под- множеством другого	18
2.3.3. Непересекаемость двух упорядоченных множеств	19
2.4. Перевод в список и обратно, свертка и фильтрация	20
2.4.1. Перевод упорядоченного множества в список	20
2.4.2. Перевод списка в упорядоченное множество	20
2.4.3. Свертка	21
2.4.4. Фильтрация	21
2.5. Объединение, пересечение, разность	22
2.5.1. Объединение	22
2.5.2. Пересечение	23
2.5.3. Разность	24

3. Сравнение с модулями sets и ordsets	25
3.1. Вставка и удаление	26
3.2. Логические функции	27
3.3. Перевод в список и обратно, свертка и фильтрация . . .	29
3.4. Объединение, пересечение, разность	32
Заключение	34
Список литературы	35

Введение

В данной работе было проведено сравнение времени выполнения стандартных реализаций множества и упорядоченного множества с собственной реализацией упорядоченного множества на языке программирования Erlang.

Сравнение проводится для всех функций рассматриваемых структур данных. Реализация функций подобрана исходя из скорости выполнения. Упорядоченное множество реализовано на основе красно-черного дерева.

Целью работы является сравнение времени выполнения функций различных структур данных и анализ зависимости этого времени от реализации структуры.

1. Обзор используемых технологий и алгоритмов

1.1. Erlang

Erlang – функциональный язык программирования, созданный для разработки распределенных динамических систем [1]. Основные его преимущества: быстрая и эффективная разработка, устойчивость системы к аппаратным сбоям и возможность обновления всей системы без остановки программ.

1.1.1. Переменные и атомы

Переменные в Erlang начинаются с заглавной буквы. Им можно присваивать значения только один раз. Переменная, которой значение уже присвоено, называется связанной. В противном случае она называется свободной. Попытка присвоить связанной переменной новое значение приведет к сообщению об ошибке.

```
X = 42.
```

Атомы используются для представления нечисловых констант. Значением атома является сам атом.

```
monday.
```

1.1.2. Кортежи

Кортеж – единая группа из фиксированного числа объектов. Группа является анонимной, как и каждое отдельное поле кортежа. Часто первым элементом кортежа используют атом, который описывает этот кортеж.

```
{1, september, 2012}.
```

```
{point, 6, 7}.
```

Возможно присваивать переменным значения отдельных элементов кортежа. Символ `_` называется анонимной переменной. Такой переменной не привязывается соответствующее значение.

```
{Name, _} = {joe, armstrong}.
```

1.1.3. Списки

Списки используются для хранения различных данных. Головой списка называется его первый элемент. Если удалить голову из списка, то останется хвост исходного списка.

```
[{joe, armstrong}, {1, september, 2012}, 42].
```

1.1.4. Функции

Рассмотрим описание функций в Erlang на примере нахождения площади прямоугольника и круга.

```
area({rectangle, Width, Height}) -> Width * Height;  
area({circle, Radius}) -> 3.14159 * Radius * Radius.
```

Функция `area` содержит 2 варианта сопоставления аргументов — клузы. Первый вариант необходим для нахождения площади прямоугольника, а второй — круга.

1.2. Красно-черные деревья

Красно-черное дерево — двоичное дерево поиска, узлы которого разделены на красные (red) и черные (black). Для таких деревьев должны выполняться красно-черные свойства (RB properties), гарантирующие, что глубины любых двух листьев отличаются не более чем в 2 раза [2].

Узлы красно-черного дерева обычно содержат следующие поля:

1. Значение

2. Цвет
3. Родитель
4. Левый ребенок
5. Правый ребенок

Важно отметить, что если ребенок или родитель отсутствует, то соответствующее поле содержит черный лист.

Рассмотрим упомянутые выше красно-черные свойства:

1. Каждый узел дерева — либо красный, либо черный.
2. Корень дерева — черный.
3. Каждый лист — черный.
4. Если узел красный, то оба его ребенка черные.
5. Все простые пути, идущие от корня к листьям, содержат одинаковое количество черных узлов.

Для удобства работы все листья заменяются одним черным листом. Это обычный узел дерева со значением `nil`, черным цветом и произвольными данными о потомках. Использование подобного узла позволяет рассматривать дочерний по отношению к нему черный лист как обычный узел с известным предком.

Черная высота узла X — количество черных узлов на любом простом пути от узла X (не считая сам узел) к листу. Обозначим черную высоту как $bh(X)$.

В соответствии со свойством 5 черная высота узла — точно определяемое значение, поскольку все нисходящие простые пути из узла содержат одно и то же количество черных узлов.

Черная высота дерева — черная высота его корня.

Лемма

Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2 \lg(n + 1)$.

Операции поиска, минимума, максимума, предков, потомков, вставки, удаления выполняется за время $O(\lg h)$, где h — высота красно-черного дерева.

Так как операции вставки и удаления изменяют красно-черное дерево, то в результате их работы могут нарушаться красно-черные свойства. Для восстановления красно-черных свойств необходимо изменить:

1. Цвета некоторых узлов дерева.
2. Родительско-дочерние связи некоторых узлов дерева.

Последнее выполняется с помощью поворотов. Это локальные операции в дереве поиска, сохраняющие красно-черные свойства. Существует 2 типа поворотов: левый и правый.



Рисунок 1 — Пример левого и правого поворотов.

Замечание

При выполнении левого поворота в узле X предполагается, что его правый ребенок Y не является черным узлом.

При выполнении правого поворота в узле Y предполагается, что его левый ребенок X не является черным узлом.

2. Реализации

2.1. Структура дерева

Упорядоченное множество реализовано с помощью красно-черного дерева. Упорядоченность и уникальность элементов обеспечивается тем, что красно-черное дерево является двоичным деревом поиска.

Дерево реализовано как кортеж, хранящий в себе свои поддеревья.

{Key, Color, Left, Right}

где Key — значение, Color — цвет узла, Left — левое поддерево, Right — правое поддерево.

Лист дерева представляется в виде

{nil, black, nil, nil}

так как у листа нет ни значения, ни потомков, а его цвет всегда черный.

2.2. Вставка и удаление

2.2.1. Вставка

Рассмотрим алгоритм вставки. Для соблюдения красно-черных свойств необходимо:

1. Вставить новый узел в красно-черное дерево, как в обычное бинарное дерево поиска, и окрасить его в красный цвет.
2. Произвести балансировку всего дерева от корня к листьям.
3. Окрасить корень в черный цвет, так как в процессе балансировки он мог стать красным.

```
add_element(Key, Tree) ->  
    make_black(ins(Key, Tree)).
```

где

```
make_black({Key, _, Left, Right}) ->  
    {Key, black, Left, Right}.
```

окрашивает узел в черный цвет вне зависимости от того, какого цвета он был раньше.

```
ins(Key, Tree)
```

вставляет в дерево Tree значение Key и производит его балансировку.

```
ins(Key, {nil, black, nil, nil}) ->  
    {Key, red, {nil, black, nil, nil}, {nil, black, nil, nil}};
```

если функция вызвана для пустого дерева, то создать дерево с корнем, у которого значение Key красного цвета.

```
ins(Key, {Key, Color, Left, Right}) ->  
    {Key, Color, Left, Right};
```

если функция вызвана для дерева, в котором существует значение Key, то прекратить рекурсивные вызовы, а дерево оставить без изменений.

```
ins(Key, {Key1, Color, Left, Right}) when Key < Key1 ->  
    balance({Key1, Color, ins(Key, Left), Right});
```

если значение необходимо вставить в левое поддерево, то вызвать функцию для левого поддерева и сбалансиовать текущее дерево. Аналогично и для правого поддерева:

```
ins(Key, {Key1, Color, Left, Right}) when Key > Key1 ->  
    balance({Key1, Color, Left, ins(Key, Right)}).
```

Функция `balance`, выполняющая балансировку дерева, реализует 4 случая нарушения четвертого красно-черного свойства рассмотренные ранее.

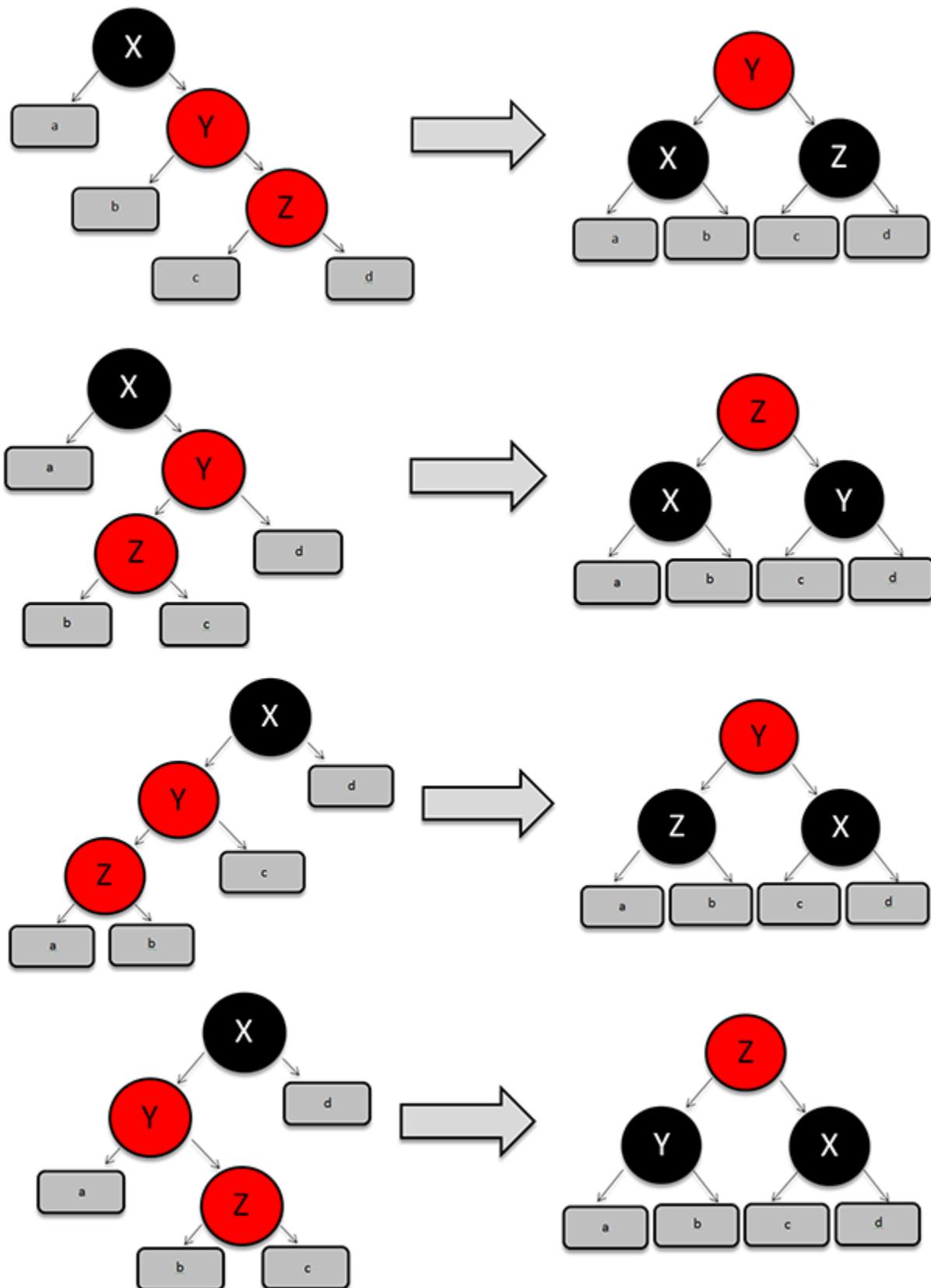


Рисунок 2 – Случаи нарушения красно-черных свойств при вставке

2.2.2. Удаление

Реализация использует арифметику цветов. К красному и черному цветам можно добавить или отнять черный цвет. Пусть при вычитании из красного цвета черного цвета получится негативный черный, вычитание из черного цвета черного даст красный цвет, добавление к красному цвету черного даст черный и добавление к черному цвету черного даст двойной черный цвет [3].

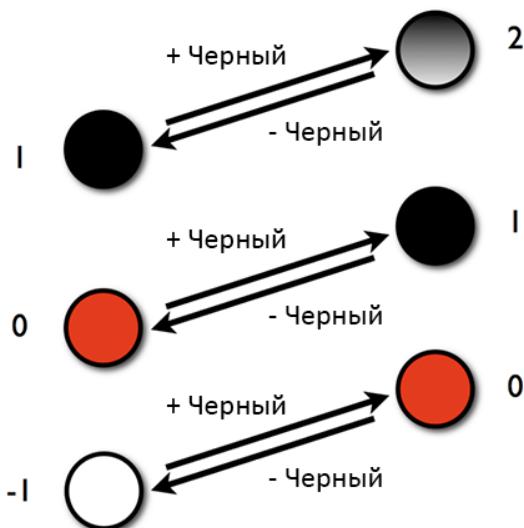


Рисунок 3 – Арифметика цветов

Реализуется арифметика цветов следующей функцией добавления цвета:

```
addBlack({Key, red, Left, Right}) ->  
{Key, black, Left, Right};
```

```
addBlack({Key, black, Left, Right}) ->  
{Key, doubleBlack, Left, Right}.
```

Функция вычитания цвета не используется.

Рассмотрим алгоритм удаления. Для соблюдения красно-черных свойств необходимо:

- Если у удаляемого узла 1 потомок и этот узел красный, то удаляем его, а на его место ставим единственного потомка. Если у удаляемого узла 1 потомок и этот узел черный, то удаляем его, а на его место ставим единственного потомка с увеличенным цветом. Если у удаляемого узла 2 потомка, то удаляем его, а на его место ставим узел из левого поддерева с максимальным значением, удаляя из левого поддерева этот узел.
- Произвести балансировку всего дерева от корня к листьям, исправляя цвета узлов.
- Окрасить корень и листья в черный цвет, так как в процессе удаления и балансировки они могли изменить цвет.

```
del_element(Key, Tree) ->
    nilFix(make_black(del(Key, Tree))).
```

где

```
nilFix({nil, doubleBlack, nil, nil}) ->
    {nil, black, nil, nil};

nilFix(Tree) ->
    Tree.
```

Если аргументом является двойной черный лист, то функция преобразует его в черный, а если аргумент — дерево, то возвращает его без изменений.

Рассмотрим функцию del, которая удаляет узел с заданным значением из дерева, а затем вызывает функцию балансировки дерева delFix.

```
del(_, {nil, black, nil, nil}) ->
    {nil, black, nil, nil};
```

Если производится попытка удалить узел из пустого дерева, то вернуть пустое дерево.

```
del(Key, {Key, red, Left, {nil, black, nil, nil}}) ->
    Left;
```

```
del(Key, {Key, red, {nil, black, nil, nil}, Right}) ->
    Right;
```

Если узел окрашен в красный цвет и у него есть только один потомок, то вернуть потомка.

```
del(Key, {Key, black, Left, {nil, black, nil, nil}}) ->
    addBlack(Left);
```

```
del(Key, {Key, black, {nil, black, nil, nil}, Right}) ->
    addBlack(Right);
```

Если узел окрашен в черный цвет и у него есть только один потомок, то вернуть потомка с добавленным цветом для сохранения 5 красно-черного свойства.

```
del(Key, {Key, Color, Left, Right})      ->
    delFix({max(Left), Color, del(max(Left), Left), Right});
```

Если у удаляемого узла 2 потомка, то вернуть дерево, в котором вместо удаленного узла будет узел с максимальным значением из его левого потомка, цветом удаленного узла, левым потомком без своего максимального значения, а его правый потомок останется без изменений. Результат необходимо сбалансировать.

```
del(Key, {KeyTree, ColorTree, LeftTree, RightTree})
  when Key < KeyTree ->
    delFix({KeyTree, ColorTree, del(Key, LeftTree), RightTree});
```

```
del(Key, {KeyTree, ColorTree, LeftTree, RightTree})
  when Key > KeyTree ->
    delFix({KeyTree, ColorTree, LeftTree, del(Key, RightTree)}).
```

Рекурсивный поиск удаляемого узла в дереве и балансировка дерева после удаления.

Функция `delFix` реализует следующие варианты нарушения красно-черных свойств при удалении узла:

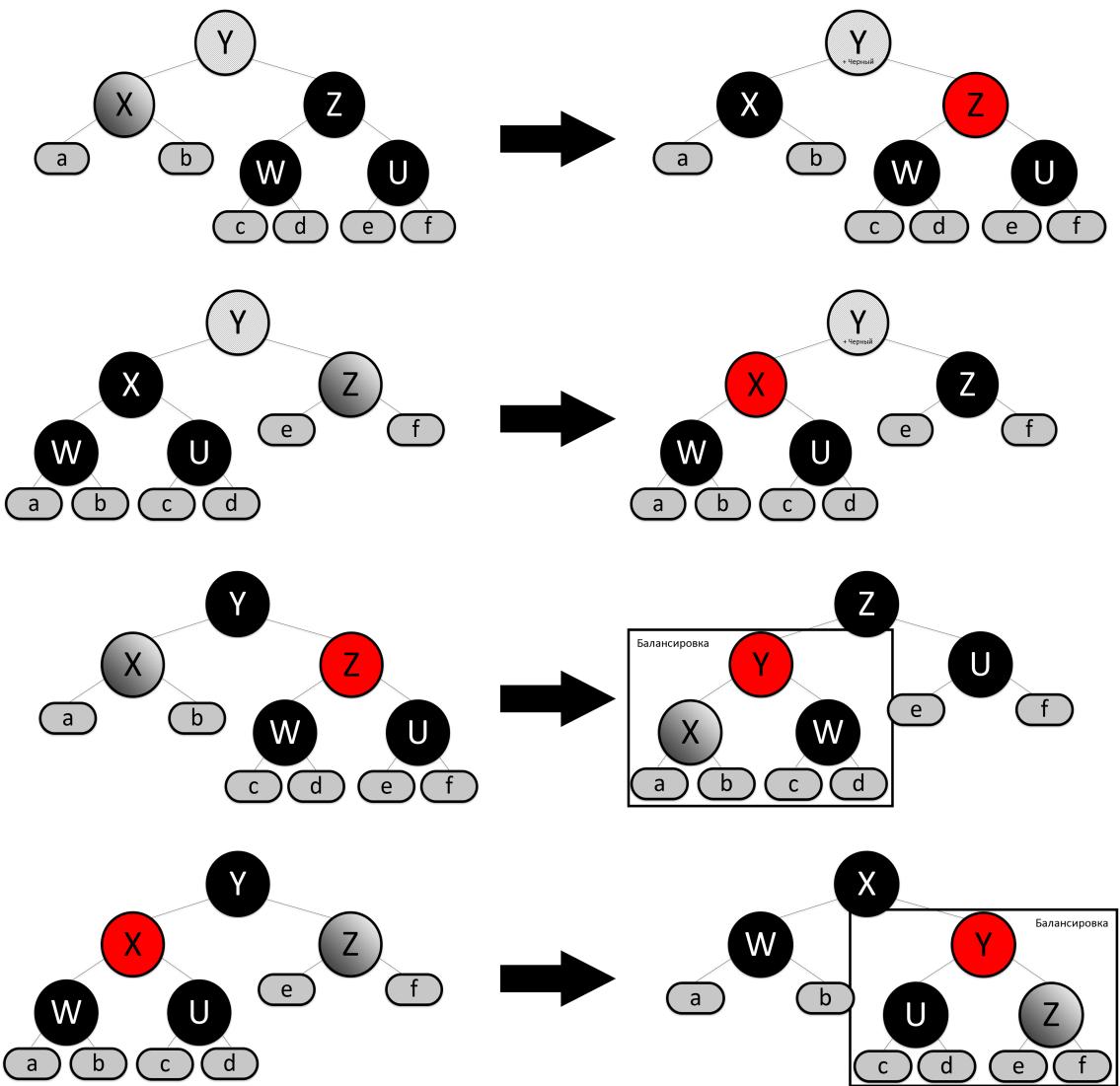


Рисунок 4 – Случаи 1 – 4 нарушения красно-чёрных свойств при удалении

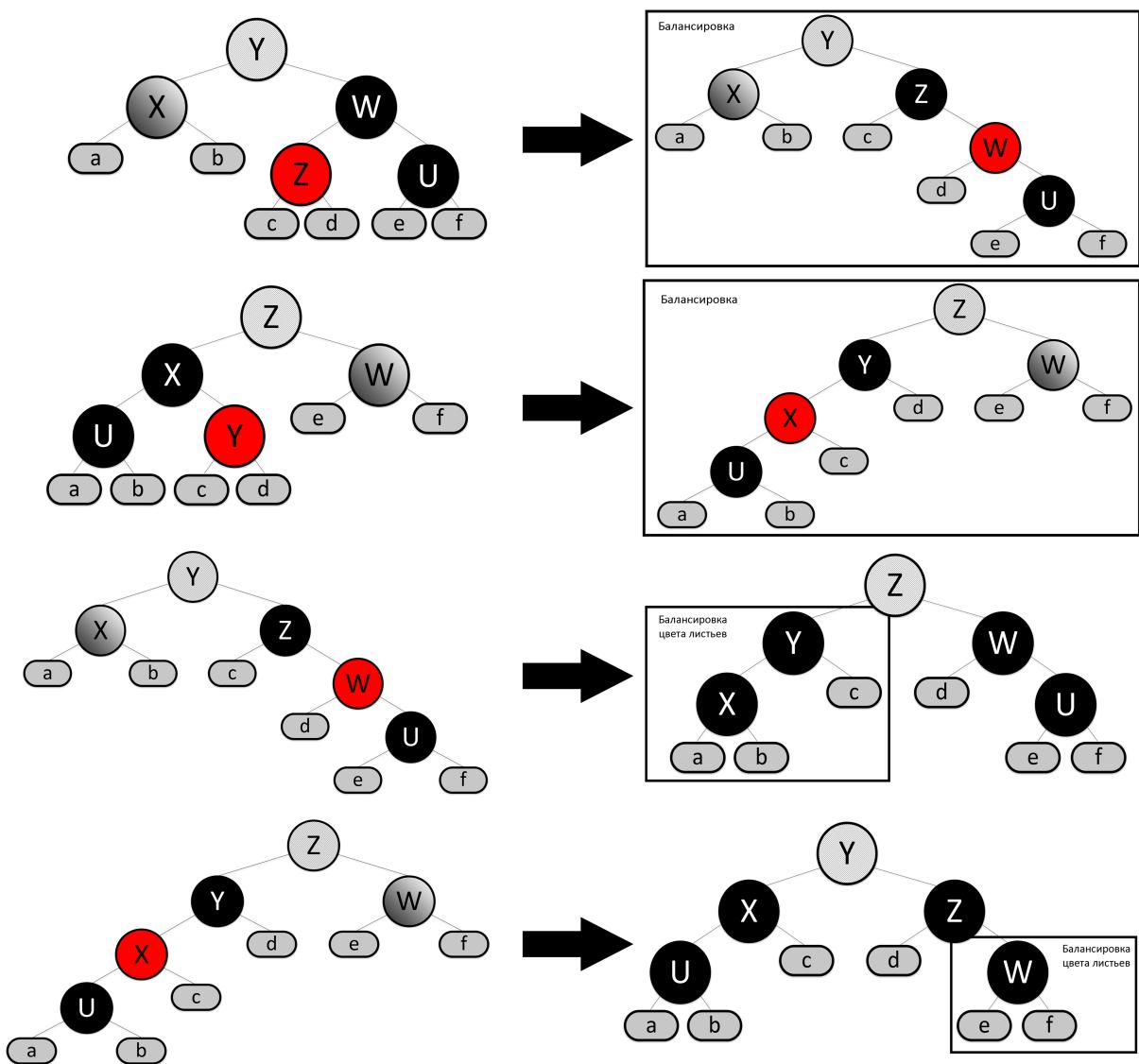


Рисунок 5 – Случаи 5 – 8 нарушения красно-черных свойств при удалении

2.3. Логические функции

Для работы с упорядоченным множеством реализованы следующие логические операции:

1. Проверка на принадлежность элемента упорядоченному множеству

2. Проверка на то, является ли одно упорядоченное множество подмножеством другого
 3. Проверка на непересекаемость двух упорядоченных множеств
- Рассмотрим каждую из них.

2.3.1. Принадлежность элемента множеству

Проверка на принадлежность элемента упорядоченному множеству реализуется с помощью функции `is_element(Elem, OSet)`, где `Elem` — значение элемента, а `OSet` — упорядоченное множество.

```
is_element(_, {nil, black, nil, nil}) ->
    false;
```

Пустому множеству не может принадлежать никакой элемент.

```
is_element(Elem, {Elem, _, _, _}) ->
    true;
```

Если найден узел с искомым значением, то элемент принадлежит множеству.

```
is_element(Elem, {CurrElem, _, Left, Right}) ->
    if
        Elem < CurrElem ->
            is_element(Elem, Left);
        Elem > CurrElem ->
            is_element(Elem, Right)
    end.
```

Поиск элемента в дереве, реализующем упорядоченное множество.

2.3.2. Является ли одно упорядоченное множество подмножеством другого

Проверка на то, является ли одно упорядоченное множество подмножеством другого, реализуется с помощью функции `is_subset(OSetA, OSetB)`, где `OSetA` — предполагаемое подмножество `OSetB`.

```
is_subset({nil, black, nil, nil}, _) ->
    true;
```

Пустое множество является подмножеством любого множества.

```
is_subset({Key, _, Left, Right}, OSetB) ->
    IsElem = is_element(Key, OSetB),
    if
        IsElem ->
            is_subset(Left, OSetB)
            and
            is_subset(Right, OSetB);
        true ->
            false
    end.
```

Проверяется принадлежность корня OSetA множеству OSetB, и если корень принадлежит OSetB, то проверяется принадлежность левого и правого поддерева множеству OSetB. Иначе OSetA не является подмножеством OSetB.

2.3.3. Непересекаемость двух упорядоченных множеств

Проверка на непересекаемость двух упорядоченных множеств реализуется с помощью функции is_disjoint(OSetA, OSetB), где OSetA и OSetB — упорядоченные множества.

```
is_disjoint(_, {nil, black, nil, nil}) ->
    true;
```

Никакое множество не пересекается с пустым.

```
is_disjoint(OSetA, {Key, _, Left, Right}) ->
    IsElem = is_element(Key, OSetA),
    if
        IsElem ->
            false;
        true ->
            is_disjoint(OSetA, Left)
            and
```

```
    is_disjoint(OSetA, Right)
end.
```

Проверяется принадлежность корня OSetB множеству OSetA, и если корень принадлежит OSetA, то множества не являются пересекающимися. Иначе проверяется непересекаемость левого и правого поддерева множеству OSetA.

2.4. Перевод в список и обратно, свертка и фильтрация

2.4.1. Перевод упорядоченного множества в список

```
to_list({nil, black, nil, nil}) ->
[];
```

Пустое множество переводится в пустой список.

```
to_list({Key, _, Left, Right}) ->
  to_list(Left) ++ [Key] ++ to_list(Right).
```

Для сохранения упорядоченности в список переводится левое поддерево, потом добавляется корневое значение, а затем в список переводится правое поддерево.

2.4.2. Перевод списка в упорядоченное множество

```
from_list(List) ->
  lists:foldl(fun(Elem, OSet) ->
    add_element(Elem, OSet)
  end,
  new(),
  List).
```

Перевод списка в упорядоченное множество реализуется с помощью стандартной функции свертки списка и функции добавления элемента во множество. К каждому элементу списка List применяется функ-

ция `fun(Elem, OSet)`, где `Elem` — элемент `List`, а в качестве `OSet` изначально берется пустое множество.

2.4.3. Свертка

Свертка `fold(Fun, Acc, OSet)` применяет к каждому элементу упорядоченного множества `OSet` функцию `Fun(Elem, Acc)` и возвращает итоговое значение `Acc`.

```
fold(_, Acc, {nil, black, nil, nil}) ->  
    Acc;
```

Если свертка применяется к пустому множеству, то просто возвращается `Acc`.

```
fold(Fun, Acc, {Key, _, Left, Right}) ->  
    Fun(Fun(Key, fold(Fun, Acc, Left)), fold(Fun, Acc, Right)).
```

Если же свертка применяется к не пустому множеству, то `Fun` применяется к левому, а затем и к правому поддереву.

2.4.4. Фильтрация

Фильтрация `filter(Pred, OSet)` применяет к каждому элементу `OSet` предикат `Pred` и возвращает упорядоченное множество элементов из `OSet` удовлетворяющих `Pred`.

```
filter(Pred, OSet) ->  
    OSetList = to_list(OSet),  
    FilteredList = lists:filter(Pred, OSetList),  
    from_list(FilteredList).
```

Упорядоченное множество переводится в список, список фильтруется с помощью стандартной функции, а затем результат переводится из списка обратно в упорядоченное множество.

2.5. Объединение, пересечение, разность

Реализации операций объединения, пересечения и разности отличаются друг от друга для достижения лучшей скорости выполнения.

2.5.1. Объединение

Операция объединения реализована рекурсивно.

```
union(0SetA, {nil, black, nil, nil}) ->  
    0SetA;
```

Объединением упорядоченного множества OSetA с пустым будет упорядоченное множество OSetA.

```
union(0SetA, {Key, _, Left, Right}) ->  
    union(union(add_element(Key, 0SetA), Left), Right).
```

Для объединения упорядоченных множеств OSetA и OSetB во множество OSetA добавляется корневой элемент OSetB, а после применяется операция объединения OSetA к левому и правому поддереву OSetB.

Для объединения более чем двух упорядоченных множеств используется функция объединения от списка упорядоченных множеств.

```
union([0Set1, 0Set2 | []]) ->  
    union(0Set1, 0Set2);
```

Если в списке всего два упорядоченных множества, то использовать операцию объединения от двух упорядоченных множеств.

```
union([0Set1, 0Set2 | 0SetsListTail]) ->  
    0SetUnion = union(0Set1, 0Set2),  
    union([0SetUnion | 0SetsListTail]).
```

В противном случае заменить в исходном списке два первых упорядоченных множества их объединением и найти объединение нового списка.

2.5.2. Пересечение

Операция пересечения реализована перебором одного из упорядоченных множеств.

```
intersection(OSetA, OSetB) ->
    intersection(OSetA, OSetB, {nil, black, nil, nil}).
```

При первом вызове операции пересечения для упорядоченных множеств OSetA и OSetB вызывается функция пересечения с пустым аккумулятором.

```
intersection({nil, black, nil, nil}, _, Acc) ->
    Acc;
```

При пересечении любого упорядоченного множества с пустым множеством необходимо вернуть аккумулятор.

```
intersection(OSetA, OSetB, Acc) ->
    OSetALeft = min(OSetA),
    OSetANew = del_element(OSetALeft, OSetA),
    IsElem = is_element(OSetALeft, OSetB),
    if
        IsElem =:= true ->
            AccNew = add_element(OSetALeft, Acc),
            intersection(OSetANew, OSetB, AccNew);
        IsElem =:= false ->
            intersection(OSetANew, OSetB, Acc)
    end.
```

В противном случае удаляем из упорядоченного множества OSetA минимальный элемент, и если он принадлежит еще и множеству OSetB, то добавляем его в аккумулятор, иначе находим пересечение нового упорядоченного множества OSetA и неизмененного OSetB.

Функция пересечения более чем двух упорядоченных множеств реализуется аналогично с объединением:

```
intersection([OSet1, OSet2 | []]) ->
    intersection(OSet1, OSet2);
```

```
intersection([OSet1, OSet2 | OSetsListTail]) ->
```

```
OSetIntersection = intersection(OSet1, OSet2),
intersection([OSetIntersection | OSetsListTail]).
```

2.5.3. Разность

Операция разности двух упорядоченных множеств как и операция пересечения реализована перебором одного из упорядоченных множеств.

```
subtract(OSetA, {nil, black, nil, nil}) ->
OSetA;
```

Вычитание из упорядоченного множества OSetA пустого множества даст упорядоченное множество OSetA.

```
subtract(OSetA, OSetB) ->
OSetBLeft = min(OSetB),
OSetBNew = del_element(OSetBLeft, OSetB),
IsElem = is_element(OSetBLeft, OSetA),
if
  IsElem =:= true ->
    OSetANew = del_element(OSetBLeft, OSetA),
    subtract(OSetANew, OSetBNew);
  IsElem =:= false ->
    subtract(OSetA, OSetBNew)
end.
```

Если же из OSetA вычитаем непустое OSetB, то удаляем из множества OSetB минимальный элемент. Если он принадлежит еще и OSetA, то удаляем его из старого OSetA и вычитаем из нового множества OSetA новое упорядоченное множество OSetB. Иначе вычитаем из неизмененного OSetA новое множество OSetB.

3. Сравнение с модулями sets и ordsets

В работе произведено сравнение времени выполнения стандартных операций собственной реализации структуры данных со стандартными структурами языка Erlang. Реализация упорядоченного множества oset [4] сравнивается со стандартным упорядоченным множеством ordsets [5] и стандартным множеством sets [6].

В ходе сравнения данныечитываются из текстового файла, заносятся в структуру и для каждой структуры проводятся замеры времени выполнения ее функций. Замер происходит дважды. В первый раз в текстовом файле находится 100 000 чисел от 0 до 2 147 483 647 и 99 997 уникальных значений, а во второй от 0 до 10 000 и 10 000 уникальных значений. Время указано в секундах.

Ниже приведены таблицы замеров времени выполнения функций модулей ordsets, oset и sets.

Таблица 1 – Таблица времени выполнения с 99997 элементами в структуре

	ordsets	oset	sets
add_element	1,93E-03	4,19E-05	4,65E-05
del_element	2,50E-03	4,80E-05	4,64E-05
is_element	4,63E-03	1,85E-06	4,51E-07
to_list	6,00E-08	2,44E-02	1,21E-02
from_list	7,39E-03	5,32E-01	5,91E-01
is_set	4,80E-03	1,08E-02	8,00E-08
union	2,10E-02	4,91E-01	1,53E+00
intesection	1,59E-02	7,63E-01	5,70E-02
subtract	1,86E-02	7,66E-01	6,13E-02
is_subset	2,10E-07	1,94E-06	9,69E-03
is_disjoint	2,59E-04	1,88E-01	1,55E-02
fold	1,36E-02	3,19E-02	1,83E-02
filter	1,75E-02	2,98E-01	2,46E-02

Таблица 2 — Таблица времени выполнения с 10000 элементами в структуре

	ordsets	oset	sets
add_element	2,65E-04	3,70E-05	3,69E-05
del_element	6,03E-05	3,72E-05	3,76E-05
is_element	3,60E-04	1,36E-06	4,44E-07
to_list	7,00E-08	2,08E-03	9,33E-04
from_list	6,13E-04	3,19E-02	1,79E-02
is_set	3,31E-04	1,18E-03	7,00E-08
union	6,01E-04	2,43E-02	1,07E-02
intesection	6,59E-04	8,38E-02	4,22E-03
subtract	3,70E-04	8,53E-02	4,64E-03
is_subset	3,61E-05	8,64E-03	3,11E-03
is_disjoint	9,00E-08	8,40E-07	8,81E-04
fold	7,58E-04	2,23E-03	9,14E-04
filter	1,01E-03	1,78E-02	1,70E-03

3.1. Вставка и удаление

В приведенных ниже временных показателях указано среднее время считывания 100 000 чисел из файла и занесения их в структуру.

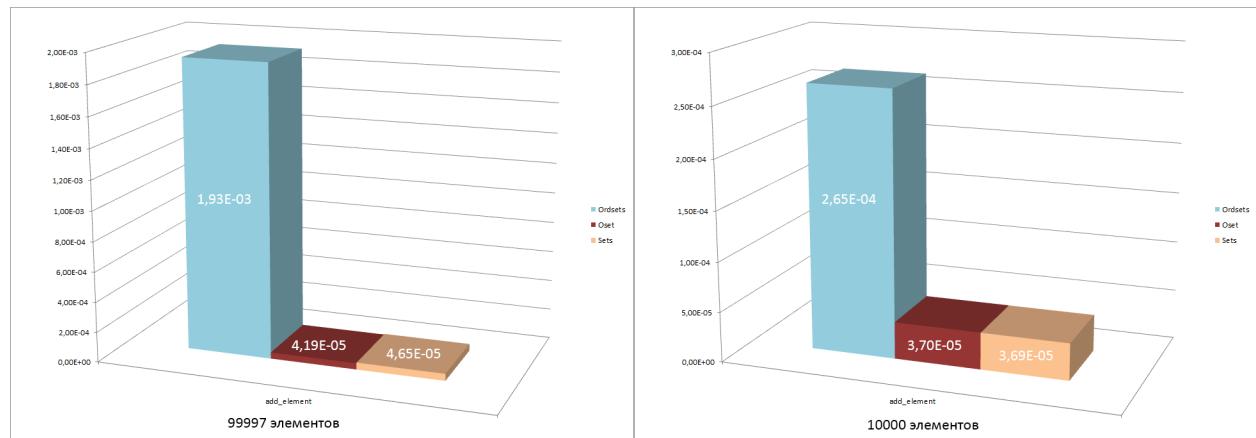


Рисунок 6 — Замер времени выполнения операции вставки

Аналогично и для операции удаления — время считывания чисел из файла и удаление их из структуры.

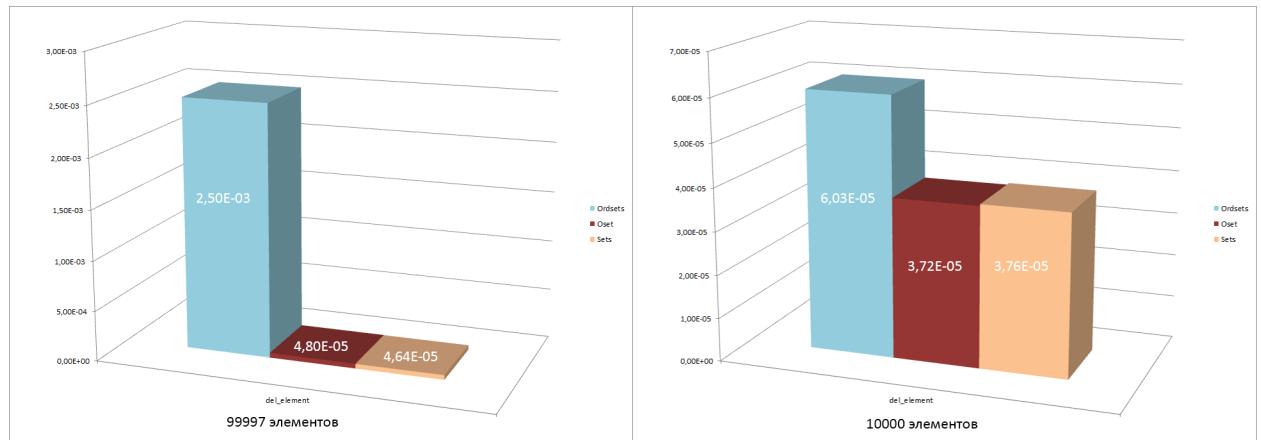


Рисунок 7 — Замер времени выполнения операции удаления

Как видно из рисунков 6 и 7, время выполнения операций вставки и удаления из модуля oset сопоставимо с временем выполнения операций в модуле sets и существенно меньше времени выполнения этих операций в модуле ordsets. Это связано с тем, что упорядоченное множество в модуле ordsets реализовано списком Erlang, а операции вставки и удаления линейны.

3.2. Логические функции

Логические функции, такие как `is_element`, `is_subset` и `is_disjoint`, запускаются по 100 раз каждая, и в результат идет среднее время выполнения.

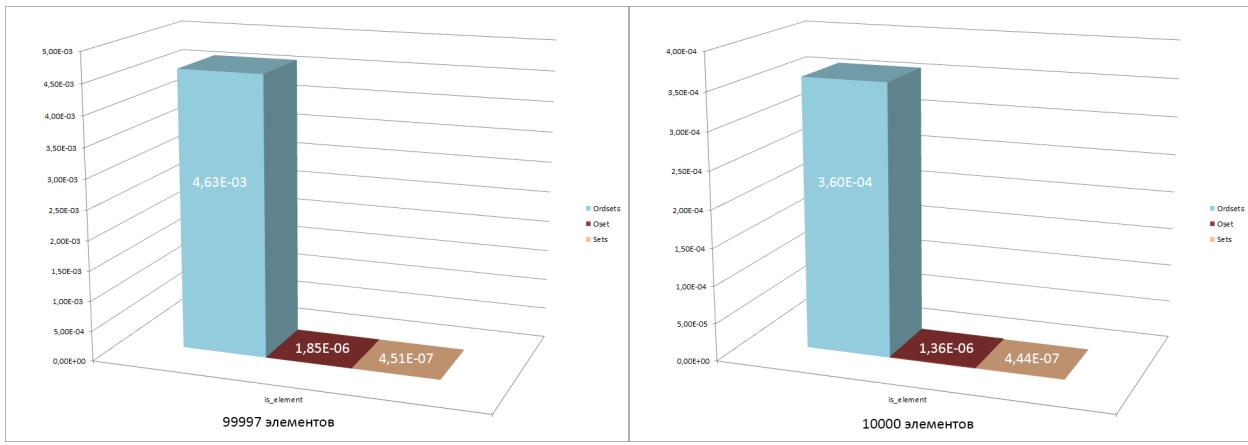


Рисунок 8 – Замер времени выполнения проверки на принадлежность элемента множеству

Благодаря тому, что модуль oset реализует упорядоченное множество на основе красно-черного дерева, операция проверки на принадлежность элемента множеству работает примерно за то же время, что и у модуля sets, и на несколько порядков быстрее, чем у модуля ordsets, так как в нем эта операция реализована перебором всех элементов списка.

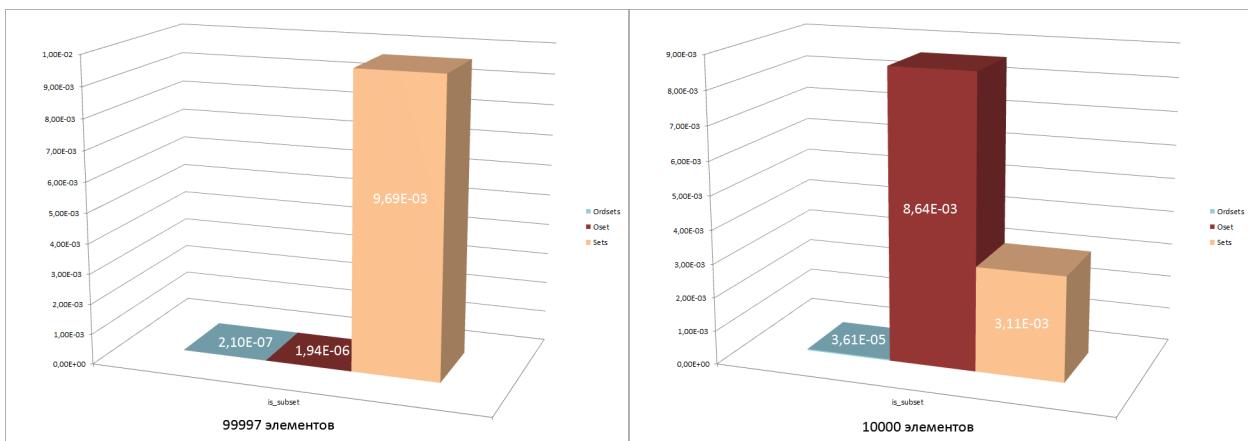


Рисунок 9 – Замер времени выполнения проверки на то, является ли одно множество подмножеством другого

При большом объеме данных множества реализация проверки в модуле sets заметно уступает по времени выполнения модулям ordsets и sets. Это связано с тем, что в модуле sets проверка реализована последовательным перебором элементов одного множества и проверкой на его принадлежность другому. При малом объеме данных реализация в модуле ordsets работает быстрее из-за реализации структуры данных с помощью списка Erlang.

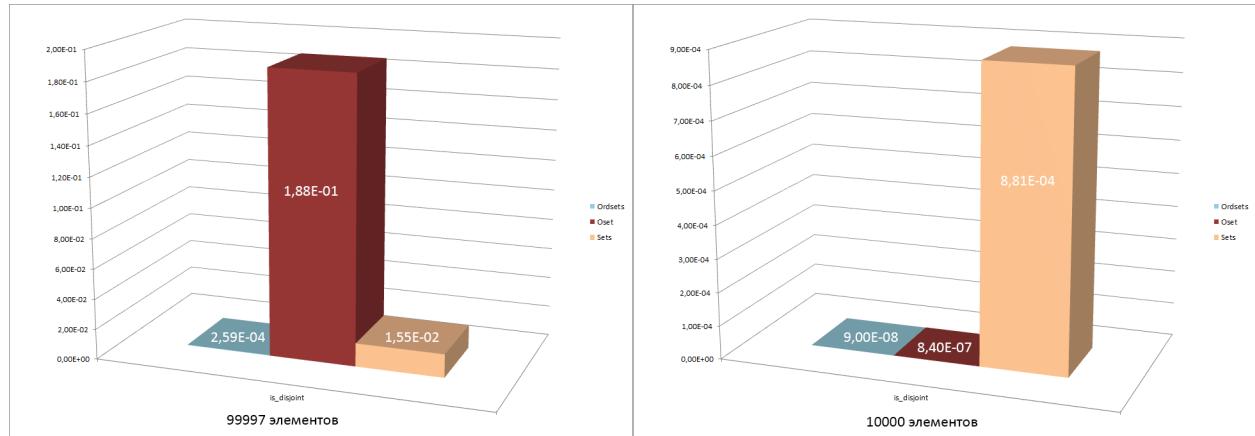


Рисунок 10 – Замер времени выполнения проверки на непересекаемость двух упорядоченных множеств

Реализация данной проверки в модуле sets при малом количестве данных работает медленнее, как и в случае с проверкой на то, является ли одно множество подмножеством другого, из-за последовательного перебора элементов одного множества. При большом объеме данных рекурсивная реализация oset уступает реализациям из модулей ordsets и sets.

3.3. Перевод в список и обратно, свертка и фильтрация

При замере времени, как и с логическими функциями, каждая функция запускается по 100 раз, а в результат идет среднее время выполнения.

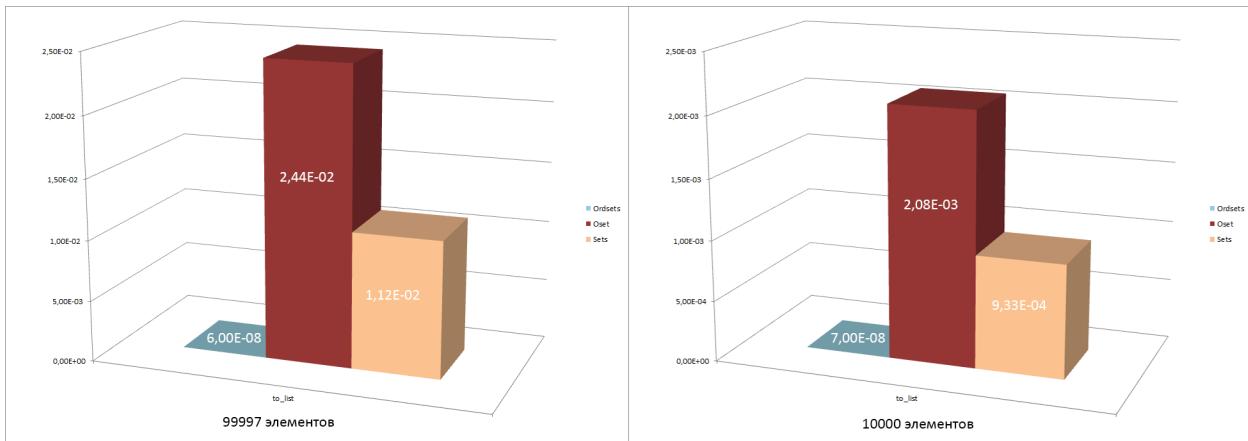


Рисунок 11 – Замер времени выполнения перевода множества в список

Самой быстрой операцией перевода множества в список является реализация из модуля ordsets, так как само множество реализовано с помощью списка. Реализация модуля sets использует операцию свертки и поэлементно заполняет список. Реализация модуля oset рекурсивно добавляет в список значение корня и добавляет к нему в начало список из левого поддерева, а в конец – из правого.

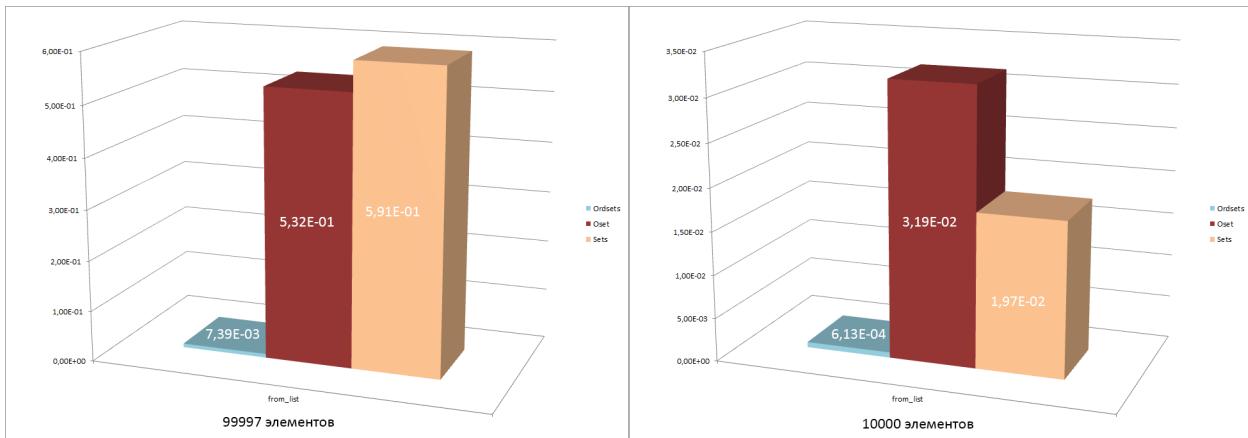


Рисунок 12 – Замер времени выполнения перевода списка в множество

На большом объеме множества реализации из модулей oset и sets работают приблизительно одинаково. Реализация модуля ordsets

работает быстрее, так как она просто сортирует список. На малом объеме данных реализация модуля oset уступает реализации модуля sets.

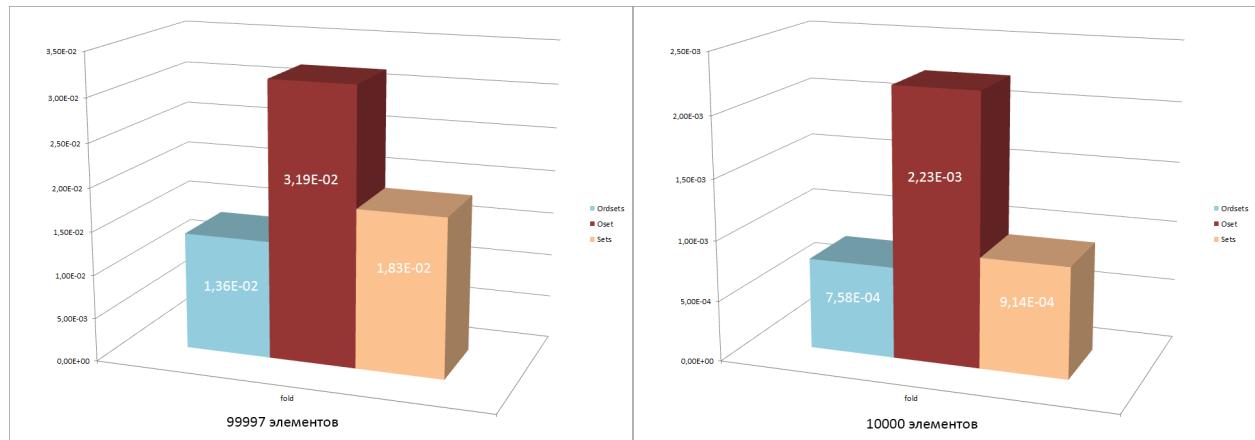


Рисунок 13 – Замер времени выполнения свертки

Свертка проверялась на функции сложения. Результатом ее выполнения являлась сумма элементов множества. Erlang умеет работать с большими числами, поэтому ошибок при вычислении свертки не происходило. Показатели времени выполнения операции свертки не сильно отличаются друг от друга, несмотря на то, что свертка в модуле ordsets реализована стандартной операцией свертки над списком.

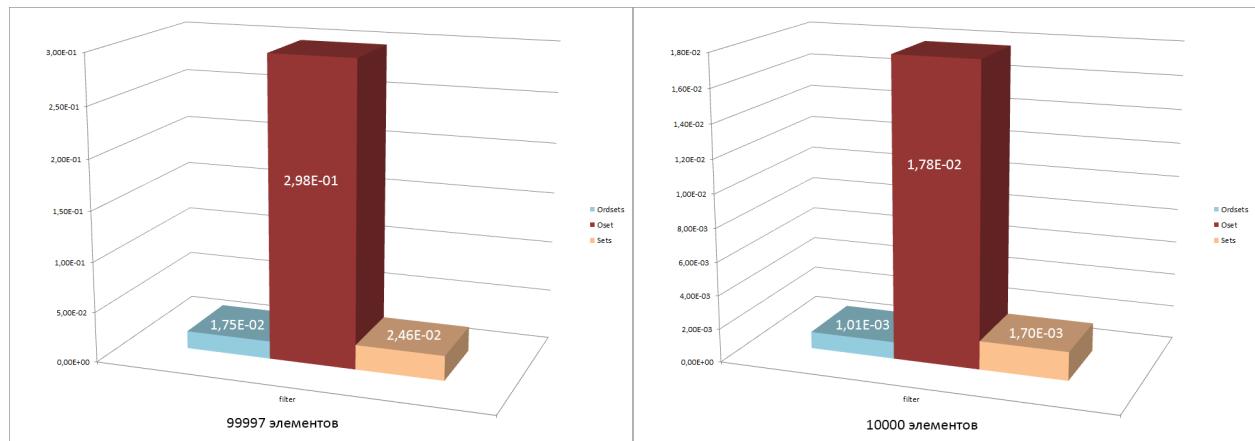


Рисунок 14 – Замер времени выполнения фильтрации

Фильтрация проверялась на предикате проверяющем четность числа. Результатом ее выполнения являлось множество четных элементов. Операция фильтрации множества в модуле oset по времени выполнения уступает реализациям модулей sets и ordsets. Отставание по времени связано с тем, что реализация oset использует перевод множества в список и стандартную реализацию операции фильтрации для списка.

3.4. Объединение, пересечение, разность

Операции объединения, пересечения и разности запускаются по 100 раз, и в результат идет среднее время выполнения.

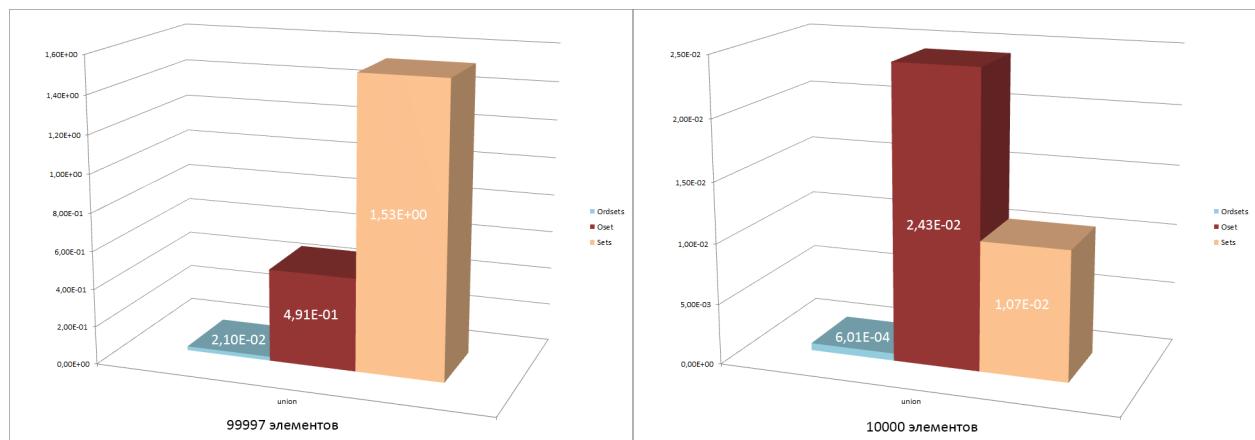


Рисунок 15 — Замер времени выполнения объединения двух множеств

На большом объеме данных реализация модуля sets уступает двум другим, потому что реализована с помощью свертки. На малом объеме данных в модуле oset самая медленная реализация операции объединения двух множеств, так как реализована рекурсивно. Реализация модуля ordsets в обоих случаях работает быстрее. Это происходит из-за реализации самого множества с помощью списка и быстроты операций с ним.

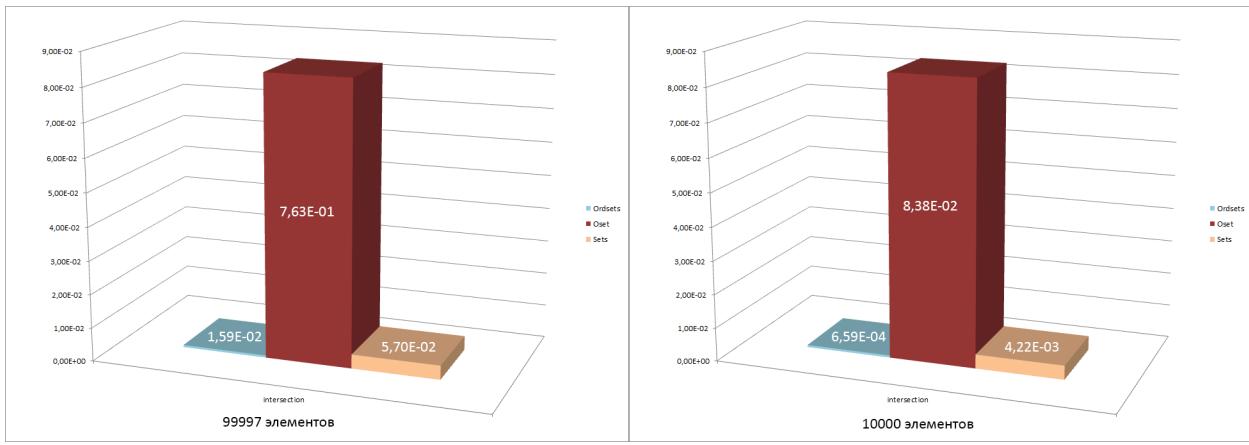


Рисунок 16 – Замер времени выполнения пересечения двух множеств

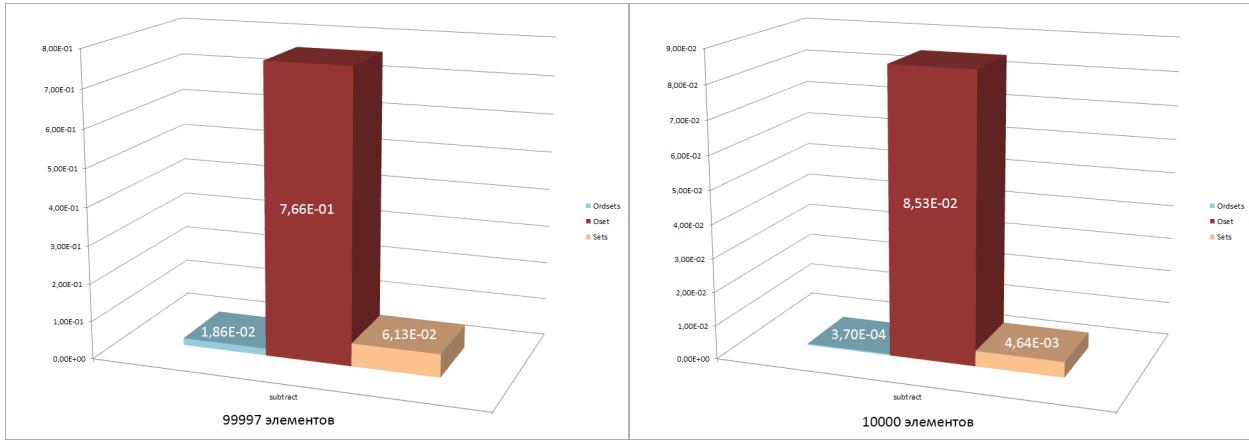


Рисунок 17 – Замер времени выполнения разности двух множеств

Время выполнения операций пересечения двух упорядоченных множеств и их разности в модуле oset уступает времени выполнения этой же операции в модулях sets и ordsets. Длительность выполнения связана с последовательным перебором всех элементов одного из множеств.

Заключение

В рамках работы изучены принципы функционального программирования, а также реализовано упорядоченное множество на основе красно-черного дерева. Было произведено сравнение времени выполнения основных функций модулей `ordsets`, `oset`, `sets` и проанализирована зависимость времени выполнения от реализации структуры данных.

Список литературы

1. Erlang. — URL: <https://ru.wikipedia.org/wiki/Erlang>.
2. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и анализ. — 6-е изд. — М. : МЦНМО, 2001. — С. 955.
3. The missing method: Deleting from Okasaki's red-black trees. — URL: <http://matt.might.net/articles/red-black-delete/>.
4. Repository VladBytsyuk/oerset. — URL: <https://github.com/VladBytsyuk/Oset-Erlang/blob/oerset.erl>.
5. Repository Erlang-OTP/ordsets. — URL: <https://github.com/blackberry/Erlang-OTP/blob/master/lib/stdlib/src/ordsets.erl>.
6. Repository Erlang-OTP/sets. — URL: <https://github.com/blackberry/Erlang-OTP/blob/master/lib/stdlib/src/sets.erl>.