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

Содержание

Введение

Здесь нужно написать введение.

1. Знакомство

1.1. Erlang

Erlang - функциональный язык программирования, созданный для разработки распределенных динамических систем. Основные его приемущества: 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 объявляются следующим образом:

```
X = 42.
```

Все переменные начинаются с заглавной буквы. В Erlang переменным 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.

Атомы используются для представления нечисловых констант.

```
monday.
```

Все атомы начинаются с прописной буквы. Также атомы могут быть заключены в одиночные кавычки (’).

```
’January’.
```

В таком случае атом может начинаться с большой буквы. Значением атома является сам атом.

1.1.2. Сопоставление по образцу

В Erlang символ = означает операцию сопоставления по образцу.

```
2 + 4 = 3 + 3.
```

В процессе выполнения данного участка кода сначала вычислится 3 + 3, далее вычислится 2 + 4, а потом сопоставятся 2 результата.

```
Y = 6 * 7.
```

В процессе выполнения данного участка кода сначала вычислится 6 * 7, а потом так как переменная Y свободная, то ее значение станет равно значению правой стороны выражения, и равенство станет верным.

1.1.3. Кортежи

Кортеж - единая группа из фиксированного числа объектов. Группа является анонимной, как и каждое отдельное поле кортежа.

```
{1, september, 2012}.
{point, 6, 7}.
```

Часто первым элементом кортежа используют атом, котрый описывает этот кортеж.

Кортежи могут быть вложенными друг в друга.

```
{date,
    {day, 1},
    {month, september},
    {year, 2012}
}.
```

Возможно присваивать переменным значения отдельных элементов кортежа.

```
{Day, Month, Year} = {1, september, 2012}.
```

В переменную Day запишется значение 1, в Month - september, а в Year - 2012.

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

Символ _ называется анонимной переменной. Такой переменной не привыязывается соответствующее значение. Результатом выполнения данного участка кода это привязка переменной Name значения joe.

1.1.4. Списки

Списки используются для хранения различых данных.

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

Головой списка называется его первый элемент. Если удалить голову из списка, то останется хвост исходного списка.

```
[H|T] = [{joe, armstrong}, {1, september, 2012}, 42].
```

В результате к переменной H будет привязано значение

```
{joe, armstrong}
```

а переменной T значение

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

Следующим образом можно добавлять элементы в список:

```
[{82, 56}, morning|T].
```

Результатом будет список

```
[{82, 56}, morning, {1, september, 2012}, 42].
```

Конкатенация списков производится следующим образом:

```
[34, red] ++ [{point, 6, 7}].
```

Результатом будет список

```
[34, red, {point, 6, 7}].
```

1.1.5. 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 - клаузы. Первый вариант необходим для находения площади прямоугольника, а второй - круга. Результатом вызова

```
area({rectangle, 2, 3}).
```

будет число 6. Выберется первый вариант выполнения функции, так как первым элементом кортежа является rectangle.

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.

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 соответсвующее поле содержит черный лист.

Рассмотрим упомянутые выше красно-черные свойства (RB properties):

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 узлу 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 соответсвии со свойством 5 - черная высота узла - точно определяемое значение, поскольку все нисходящие простые пути из узла содержат олно 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, макисмума, 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. Родительски-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.

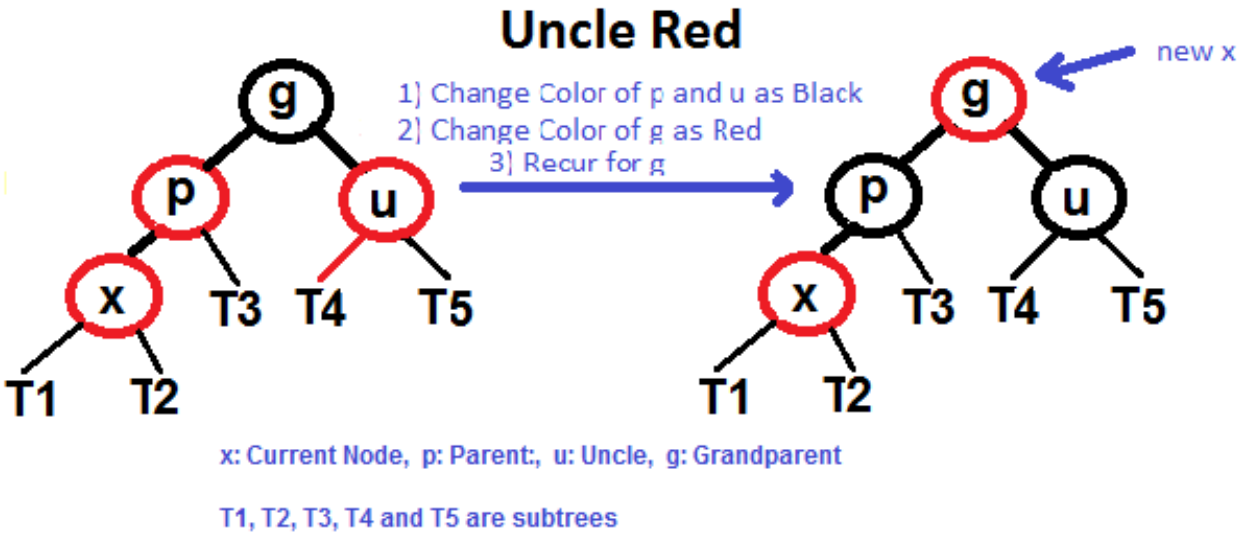

Рисунок 1 — Пример левого и правого поворотов.

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.

Рассмотрим алгоритм вставки в красно-черное дерево. Вставка выполняется в 2 этапа:

1. Вставка нового узла в красно-черное дерево, как в обычное бинарное дерево поиска, и окрашивает его в красный цвет.

2. Выполнение необходимых поворотов и перекрашиваний узлов красно-черного дерева.

При этом возникает следующая проблема - нарушаются красно-черные свойства. При выполнении необходимых поворотов и перекрашиваний узлов красно-черного дерева корень дерева может быть

окрашен в красный цвет, что будет противоречить свойству 2, а при вставки нового узла в красно-черное дерево, и окрашивании его в красный цвет может возникнуть ситуация, когда у красного узла будет красный ребенок.

При вставке возможны 4 случая нарушения четвертого красно-черного свойства:

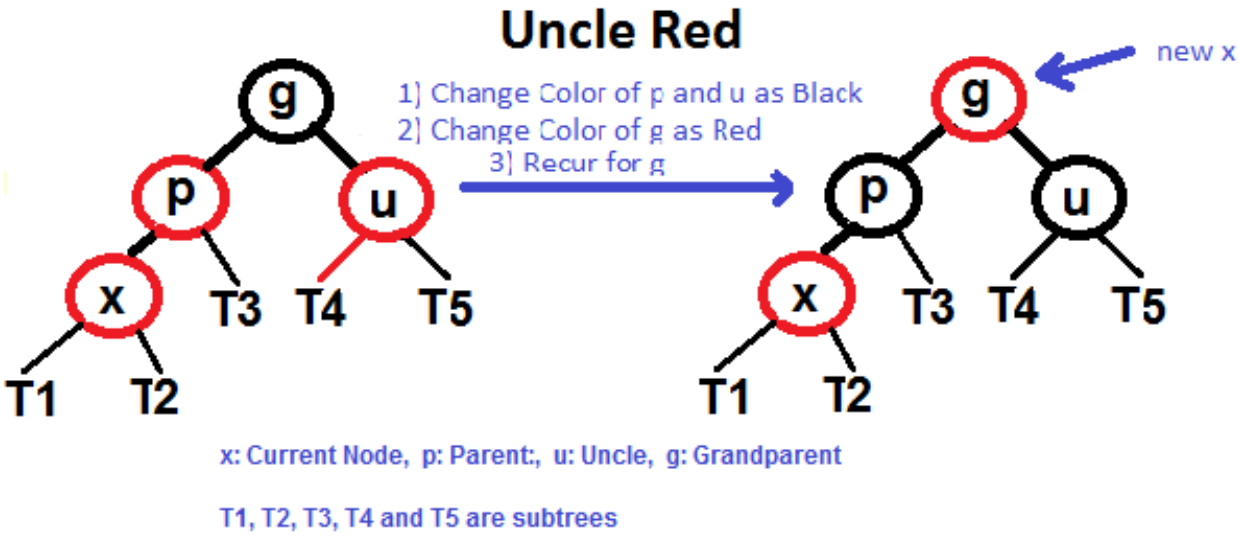

Рисунок 2 — Пример возможных нарушений красно-черных свойств после вставки.

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}
```

т.к. у листа нет ни значения, ни потомков, а его цвет всегда черный

2.2. Вставка и удаление

2.2.1. Вставка

Рассмотри 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, т.к. 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 сбалансировать 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.

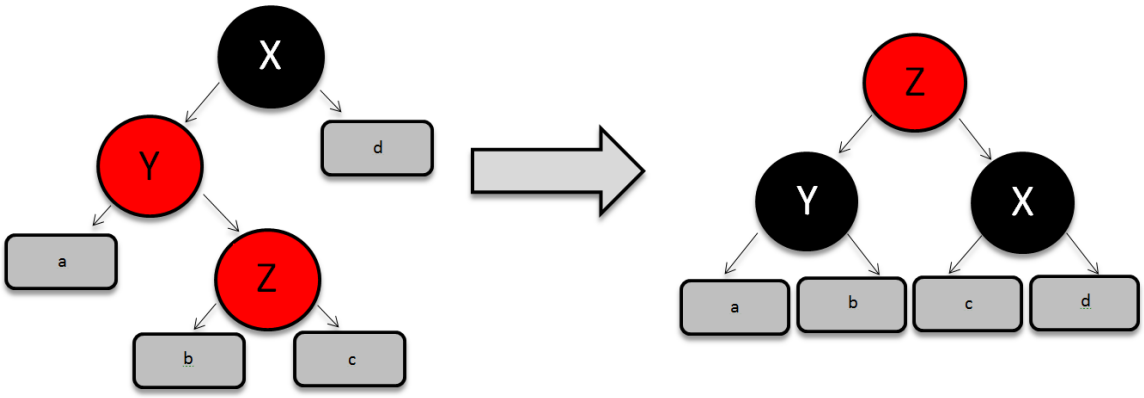

Рисунок 3 — Случаи нарушения красно-черных свойств при вставке

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.

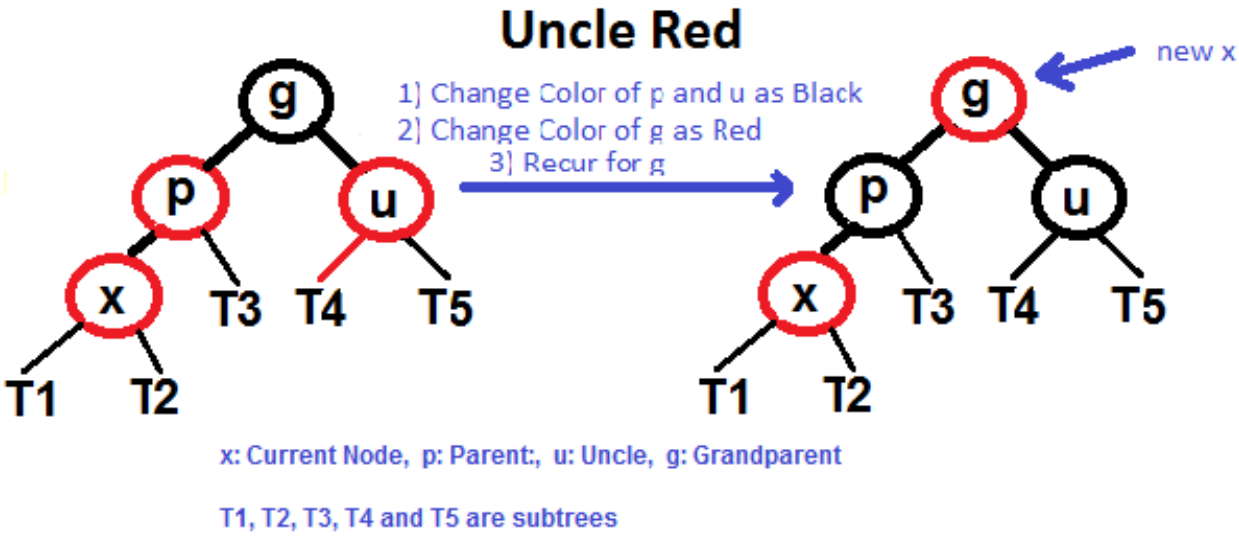

Рисунок 4 — Арифметика цветов

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.

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

1. 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.

2. Произвести балансировку всего дерева, от корня к листьям, исправляя цвета узлов.

3. Окрасить корень и листья в черный цвет, т.к. 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 агрументом является двойной черный лист 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 цвет удаляемого узла красный, 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 цвет удаляемого узла черный, 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, цвет удаленного узла, левый потомок без своего максимального значения, а 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 удаалении узла:

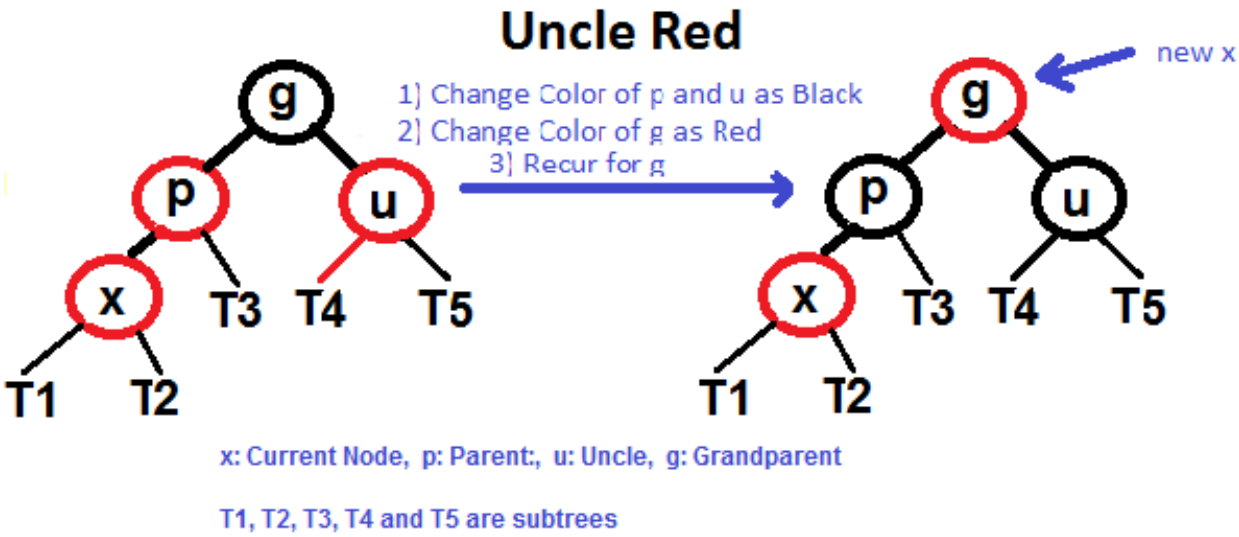

Рисунок 5 — Случаи нарушения красно-черных свойств при удалении

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 элемета в множество. К каждому элементу списка List, применяется функция 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.

2.5.1. Объединение

Операция объединения реализована рекурсивно.

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

Объединением упорядоченного множества OSetA с пустым, будет упорядоченное множество OSetA.

```
union(OSetA, {Key, _, Left, Right}) ->
    union(union(add_element(Key, OSetA), 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([OSet1, OSet2 | []]) ->
    union(OSet1, OSet2);
```

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

```
union([OSet1, OSet2 | OSetsListTail]) ->
    OSetUnion = union(OSet1, OSet2),
    union([OSetUnion | OSetsListTail]).
```

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 упорядоченного множества OSetA вычитаем непустое упорядоченное множество OSetB, то удаляем из упорядоченного множества OSetB минимальный элемент, и если он принадлежит еще и множеству OSetA, то удаляем его из множества OSetA и вычитаем из нового упорядоченного множества OSetA новое упорядоченное множество OSetB, иначе вычитаем из неизмененного упорядоченного множества OSetA новое упорядоченное множество OSetB.

3. Сравнение с модулями sets и ordsets

В своей работе я произвел сравнение времени выполнения стандартных операций реализованной мной структуры, со стандарными структурами языка Erlang. Свою реализацию упорядоченного множества oset я сравниваю со стандарным упорядоченным множеством ordsets, и стандартным множеством sets.

В ходе сравнения данные считывались из текстового файла, заносились в структуру и для каждой структуры проводились замеры времени выполнения ее функций. Замер происходил дважды. В первый раз в текстовом файле находилось 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.

3.1. Вставка и удаление

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

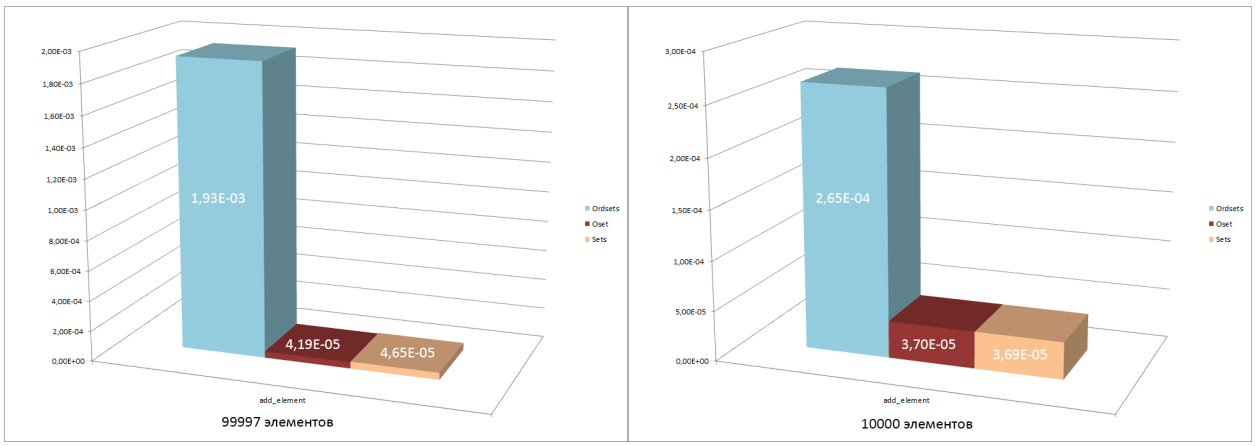

Рисунок 6 — Замер времени выполнения операции вставки

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

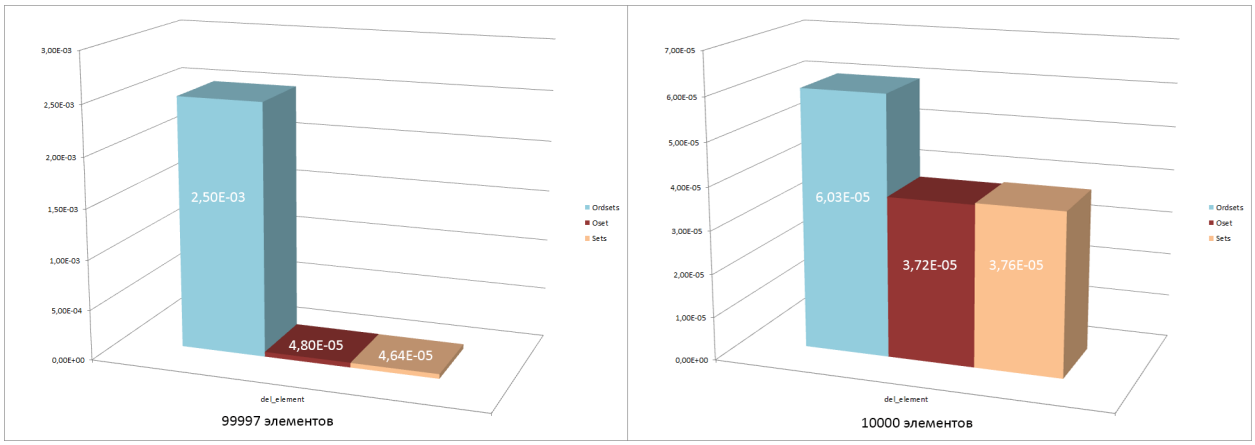

Рисунок 7 — Замер времени выполнения операции удаления

Как видно из рисунков 16 и 17, 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.

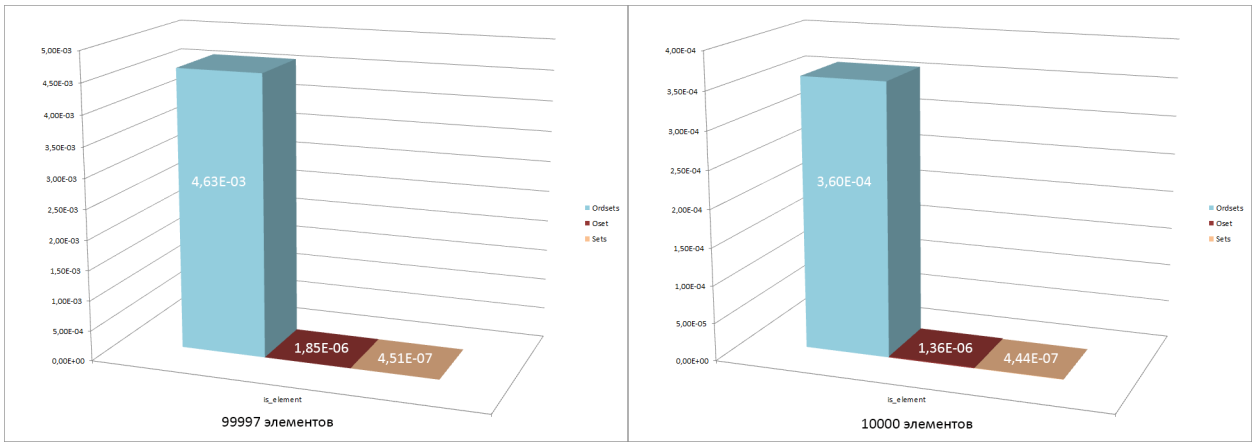

Рисунок 8 — Замер времени выполнения проверки на принадлежность элемента множеству

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.

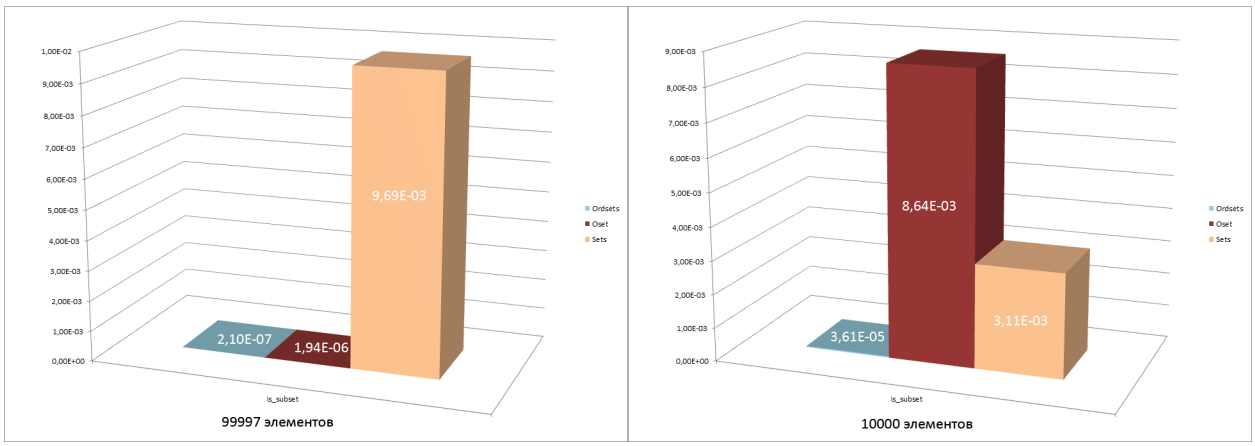

Рисунок 9 — Замер времени выполнения проверки на то, является ли одно множество подмножеством другого

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 струтуры данных с помощью списка Erlang.

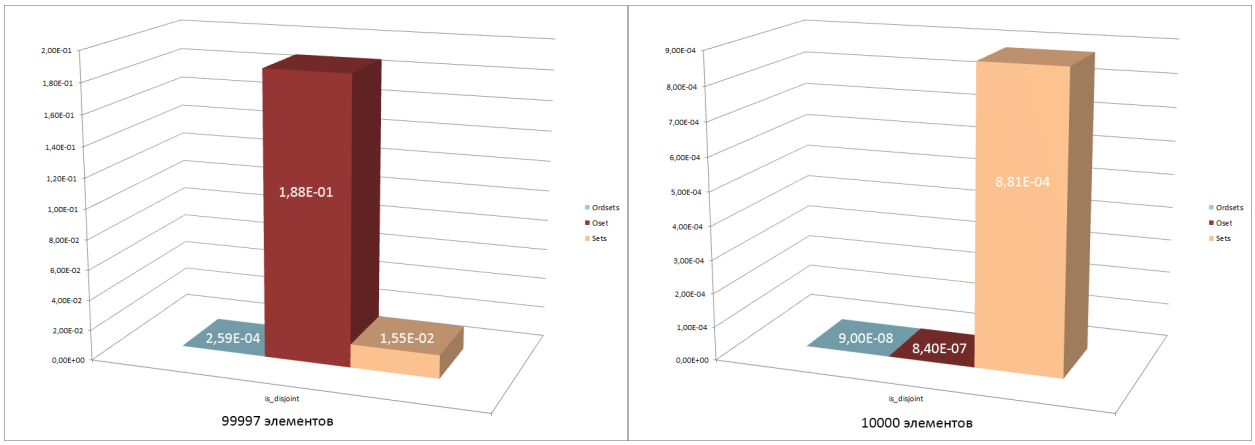

Рисунок 10 — Замер времени выполнения проверки на непересекаемость двух упорядоченных множеств

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.

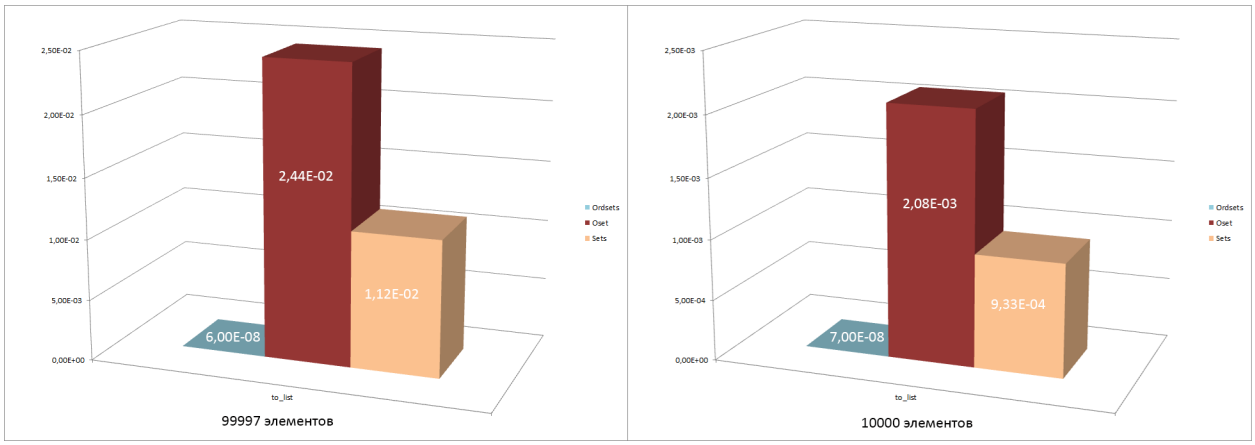

Рисунок 11 — Замер времени выполнения перевода множества в список

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 добаляет в список значение корня, и добавляет к нему в начало список из левого поддерева, а в конец - из правого.

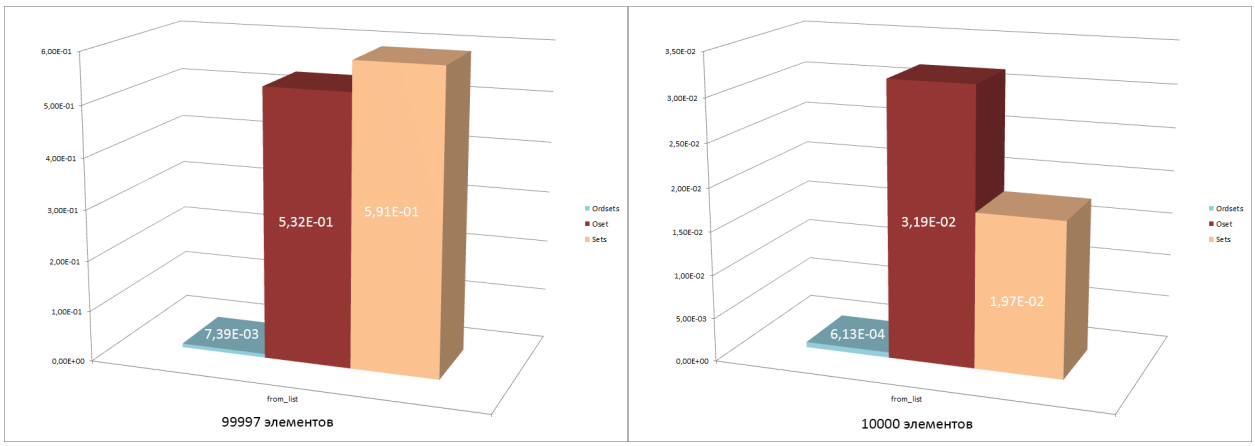

Рисунок 12 — Замер времени выполнения перевода списка в множество

На большом объеме множества реализации из модулей oset и sets работают приблизительно одинакого. Реализация модуля ordsets работает быстрее, так как она просто сортирует список. На малом объеме данных реализация модуля oset уступает реализации моделя sets.

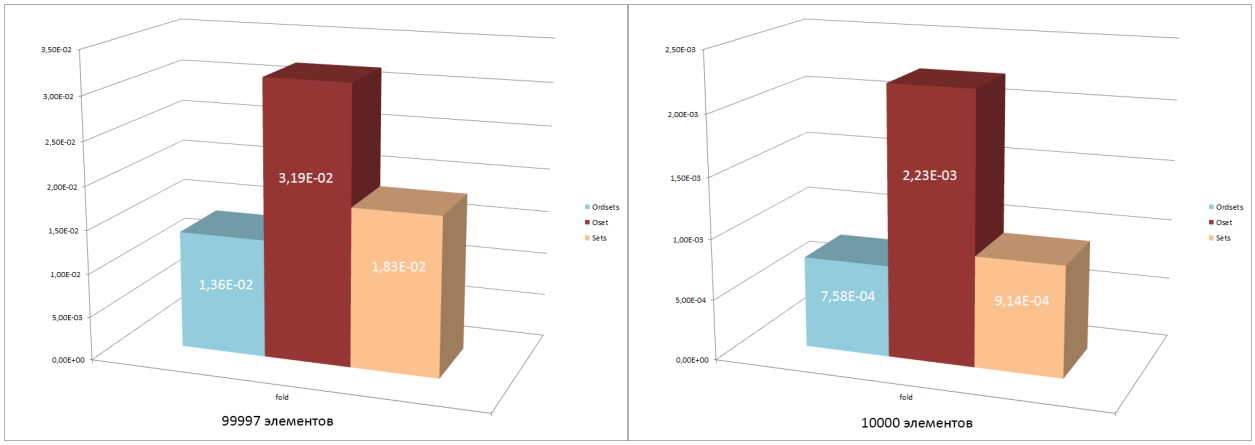

Рисунок 13 — Замер времени выполнения свертки

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.

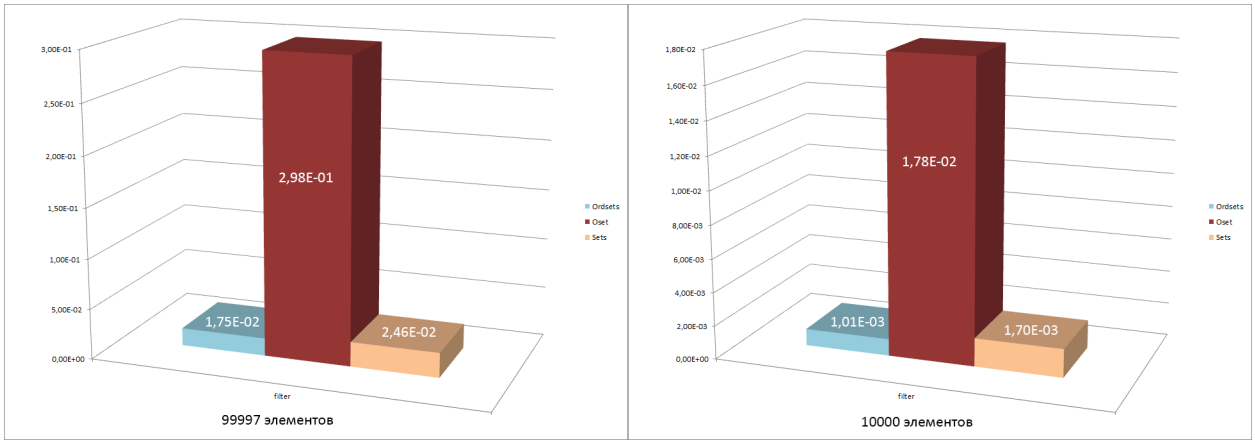

Рисунок 14 — Замер времени выполнения фильтрации

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.

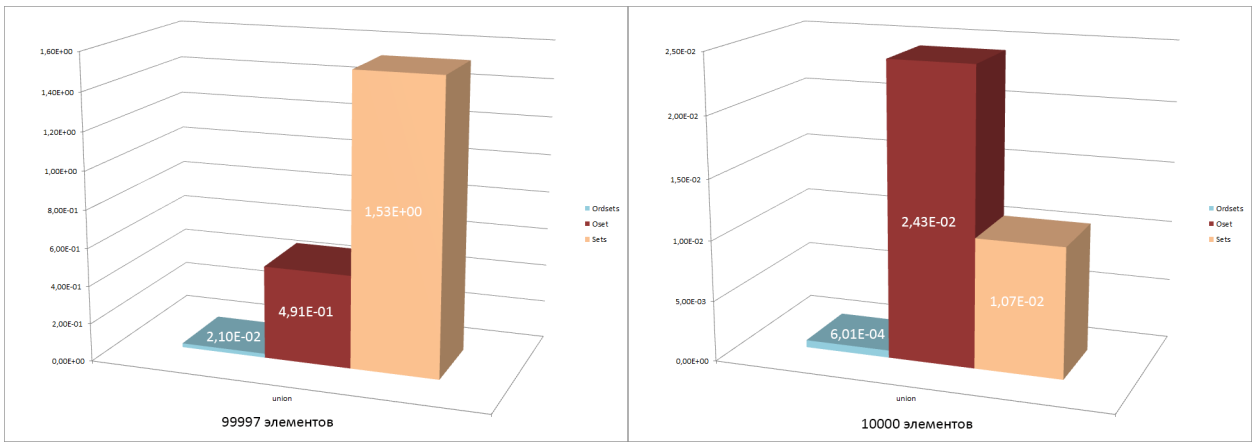

Рисунок 15 — Замер времени выполнения объединеия двух множеств

На большом объеме данных реализация модуля sets уступает двум другим, потому что реализована с поможщью 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.

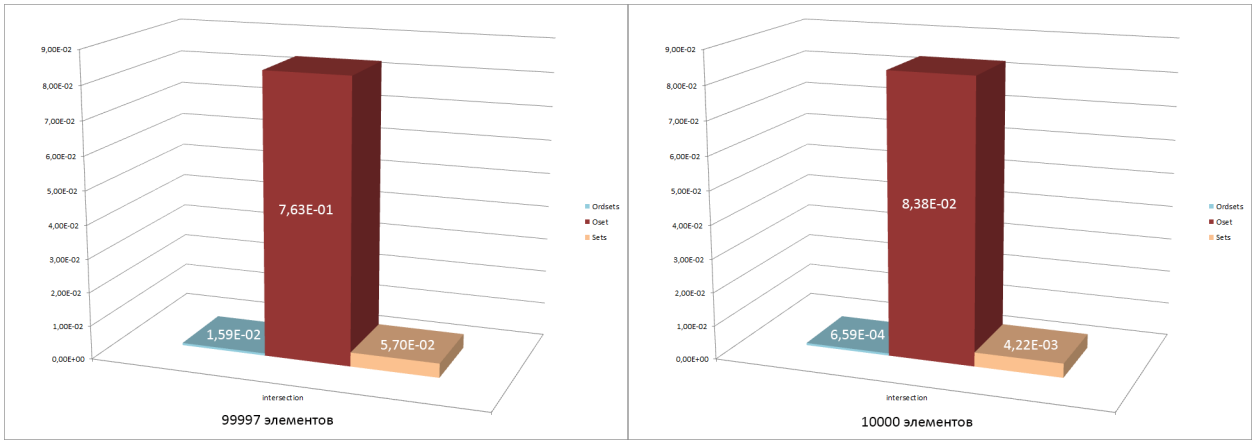

Рисунок 16 — Замер времени выполнения пересечения двух множеств

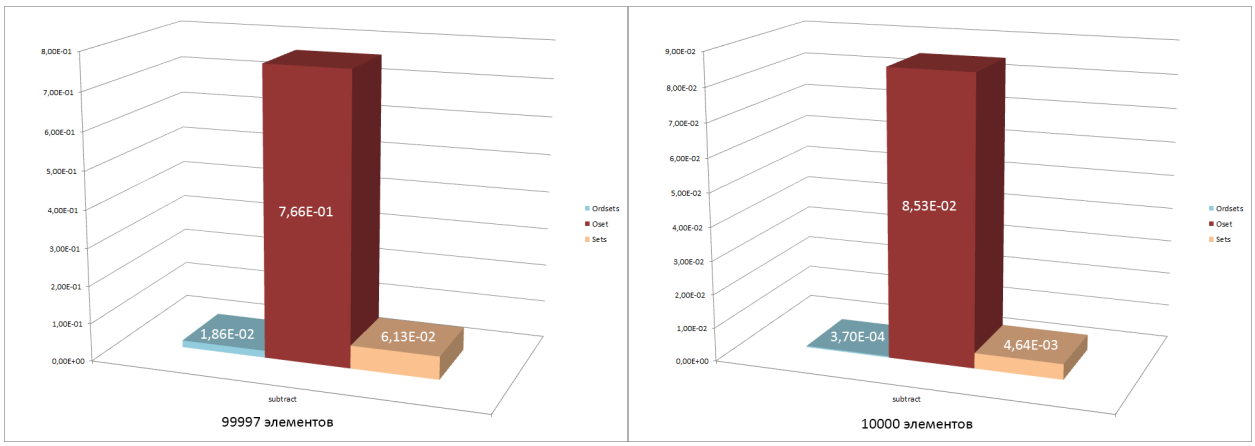

Рисунок 17 — Замер времени выполнения разности двух множеств

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