Постановка задачи по оптимизации библиотеки libSBT

Цель работы:

> *Ускорение работы библиотеки libSBT (выполняющей операции над size-balanced tree), в частности - за счёт использования 64-битных инструкций платформы Intel x64_64. Предполагаемое ускорение — порядка десятикратного. Для этого необходимо написать код на языке Си с ассемблерными вставками, либо на чистом ассемблере в AT&T-синтаксисе для Intel x64_64.*

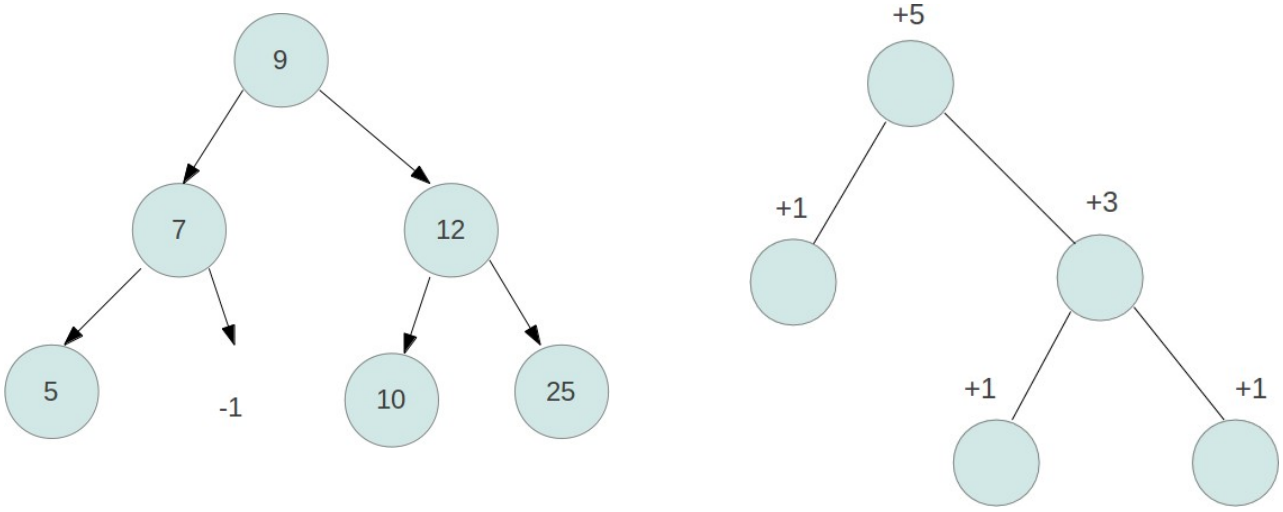
Требуемый результат работы:
1) Оптимизированный вариант библиотеки — программный код на Си с ассемблерными вставками (или код на ассемблере).
2) Набор тестов, иллюстрирующих ускорение вставки/удаления/поиска/перебора по сравнению с оригиналом (sbt.c/.h).
3) Подробная документация по применённой оптимизации: в чём отличия от оригинального кода на Си, как выполнена оптимизация. Документация должна быть в виде PDF - файла с иллюстрациями, диаграммами — поясняющими работу оптимизированного варианта, а также - содержать результаты тестов.

Описание оригинального кода

1. Основной тип данных (структура данных):

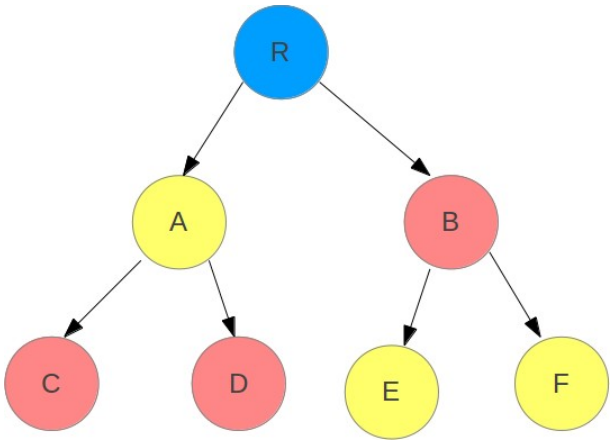
```
typedef struct TNode {
    TNumber value; // значение, привязанное к ноде
    TNodeIndex parent; // ссылка на уровень выше (поле не обязательно; используется только в parent-версии sbt.c)
    TNodeIndex left;  // ссылка на левое поддерево, = -1, если нет дочерних вершин
    TNodeIndex right; // ссылка на правое поддерево
    TNodeSize size; // size в понимании SBT
} TNode;
```

2. Пример дерева: значения вершин (слева) и размеры вершин (справа)



Если ссылка на левое или правое поддерево равна -1, то это значит, что поддерева нет (слева, или соответственно справа — вершин не подвешено).

3. Условие сбалансированности



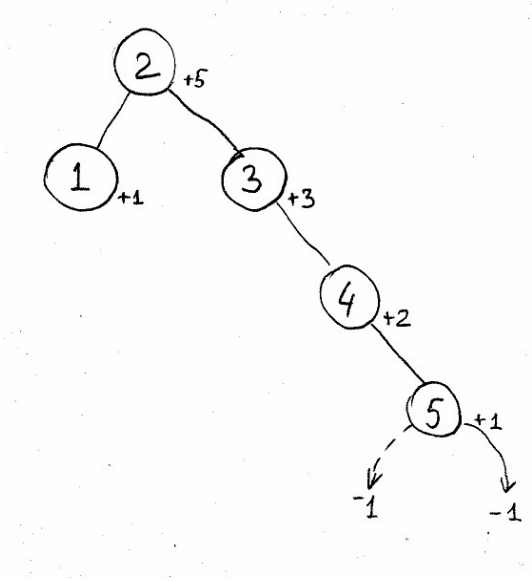


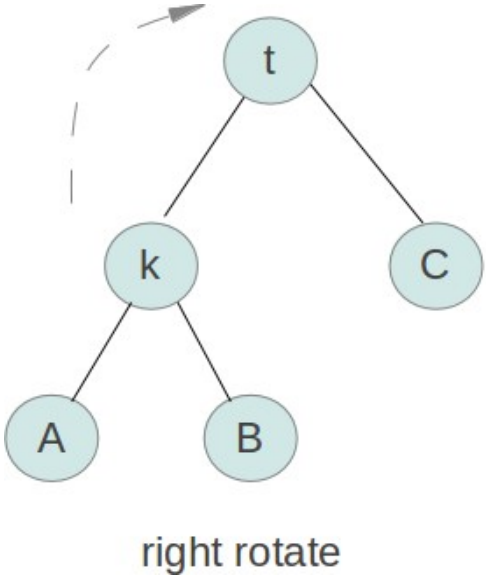
Рисунок 1: Размеры вершин

условие сбалансированности задаётся неравенствами:
E.size <= A.size C.size <= B.size
F.size <= A.size D.size <= B.size
(то есть, дерево как бы выровнено по весу горизонтальных (соседних) слоёв)

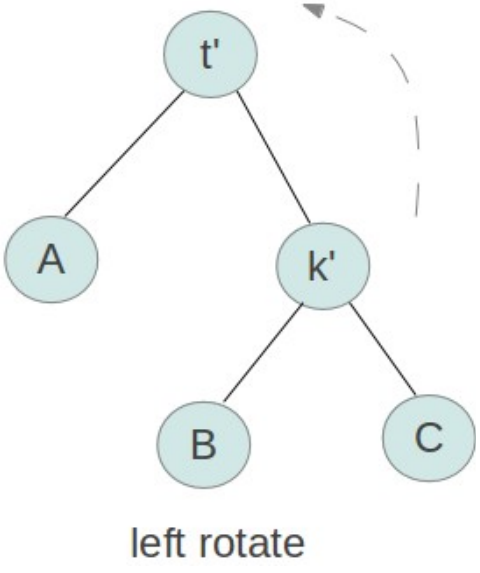
размеры (size) вершин определяются правилом:
R.size = A.size + B.size + 1
(то есть, размер вершины — это число вершин в соответствующем поддереве)

4. Вращения для балансировки
Для балансировки наших деревьев (SBT) используются «вращения»:

«правое вращение»



«левое вращение»



Основной момент здесь - это критерий «вращений» (rotates). Рассматривается четыре случая взаимного отношения «размеров» (size) вершин, близких к "корню" балансировки (t). В зависимости от соотношения «размеров», выполняется то или иное вращение.

Балансировка выполняется после добавления вершины (функция AddNode), а также — после удаления вершины из дерева (функция DeleteNode).

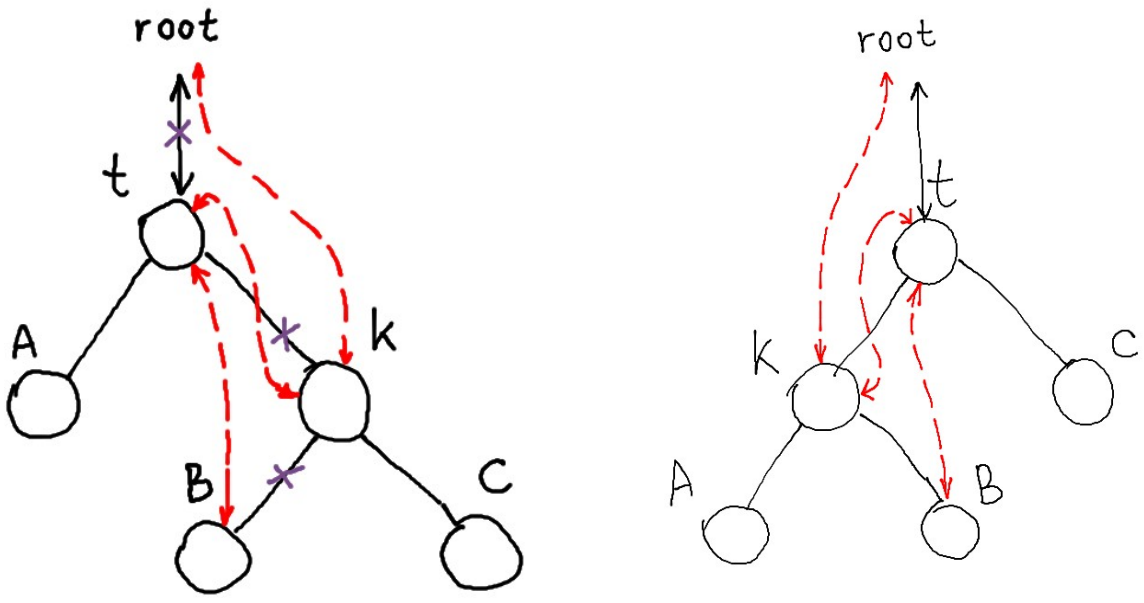
Алгоритм вращения (left или right rotate), более подробно — sbt.c

1. Если нет такой вершины t — игнорировать команду вращения.
2. Повернуть ребро дерева (см. рисунок выше), поменяв связи между вершинами (nodes).
3. Скорректировать «размер» size корня поддерева, t.

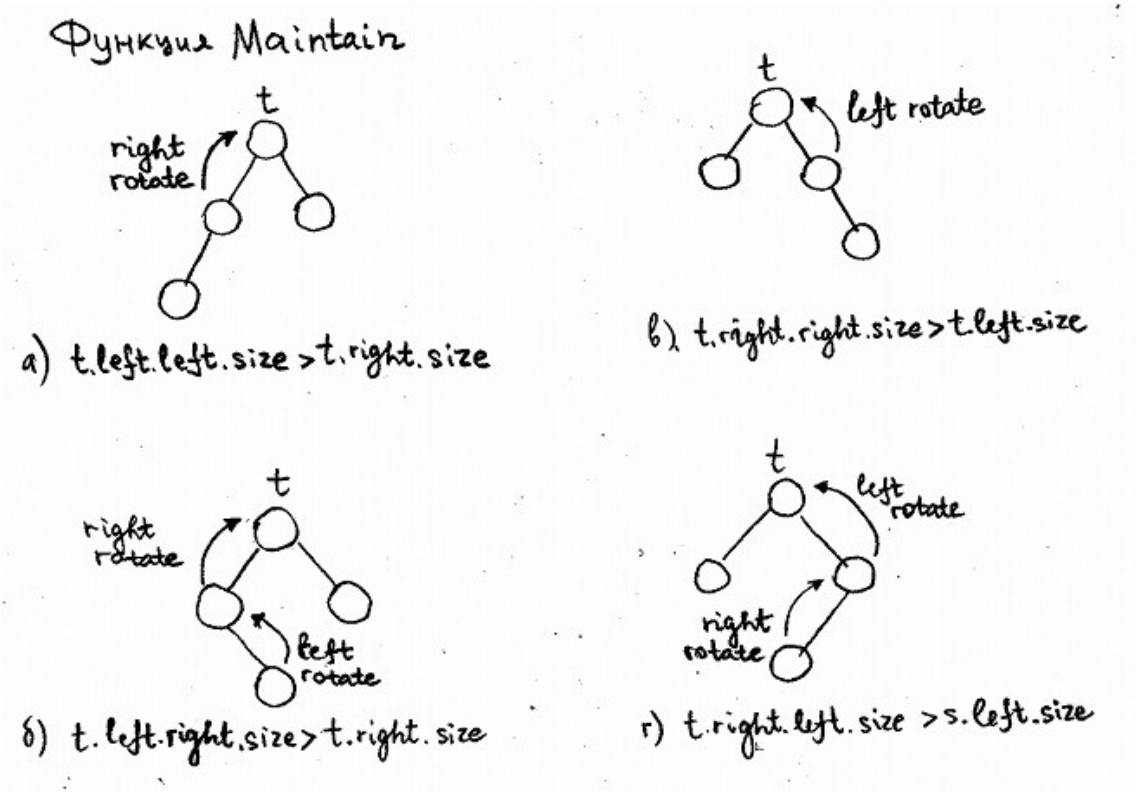
```
int SBT_LeftRotate(TNodeIndex t) {

        if (t < 0) return 0;
        TNodeIndex k = _nodes[t].right;
        if (k < 0) return 0;
        TNodeIndex p = _nodes[t].parent;

        // поворачиваем ребро дерева
        _nodes[t].right = _nodes[k].left;
        _nodes[k].left = t;

        // корректируем size
        _nodes[k].size = _nodes[t].size;
        TNodeIndex n_l = _nodes[t].left;
        TNodeIndex n_r = _nodes[t].right; // для ускорения — выборку из кэша
        TNodeSize s_l = ((n_l != -1) ? _nodes[n_l].size : 0);
        TNodeSize s_r = ((n_r != -1) ? _nodes[n_r].size : 0);
        _nodes[t].size = s_l + s_r + 1;

        // меняем трёх предков
        // 1. t.right.parent = t
        // 2. k.parent = t.parent
        // 3. t.parent = k
        if (_nodes[t].right != -1) _nodes[_nodes[t].right].parent = t; // из кэша
        _nodes[k].parent = p;
        _nodes[t].parent = k;

        // меняем корень, parent -> t, k
        if (p == -1) _tree_root = k; // это root
        else {
                if (_nodes[p].left == t) _nodes[p].left = k;
                else _nodes[p].right = k; // вторую проверку можно не делать
        }
        return 1;
}
```

(аналогично выполняется «right rotate»)

Как заменяются связи (в функции SBT_LeftRotate и SBT_RightRotate):



5. Правила балансировки

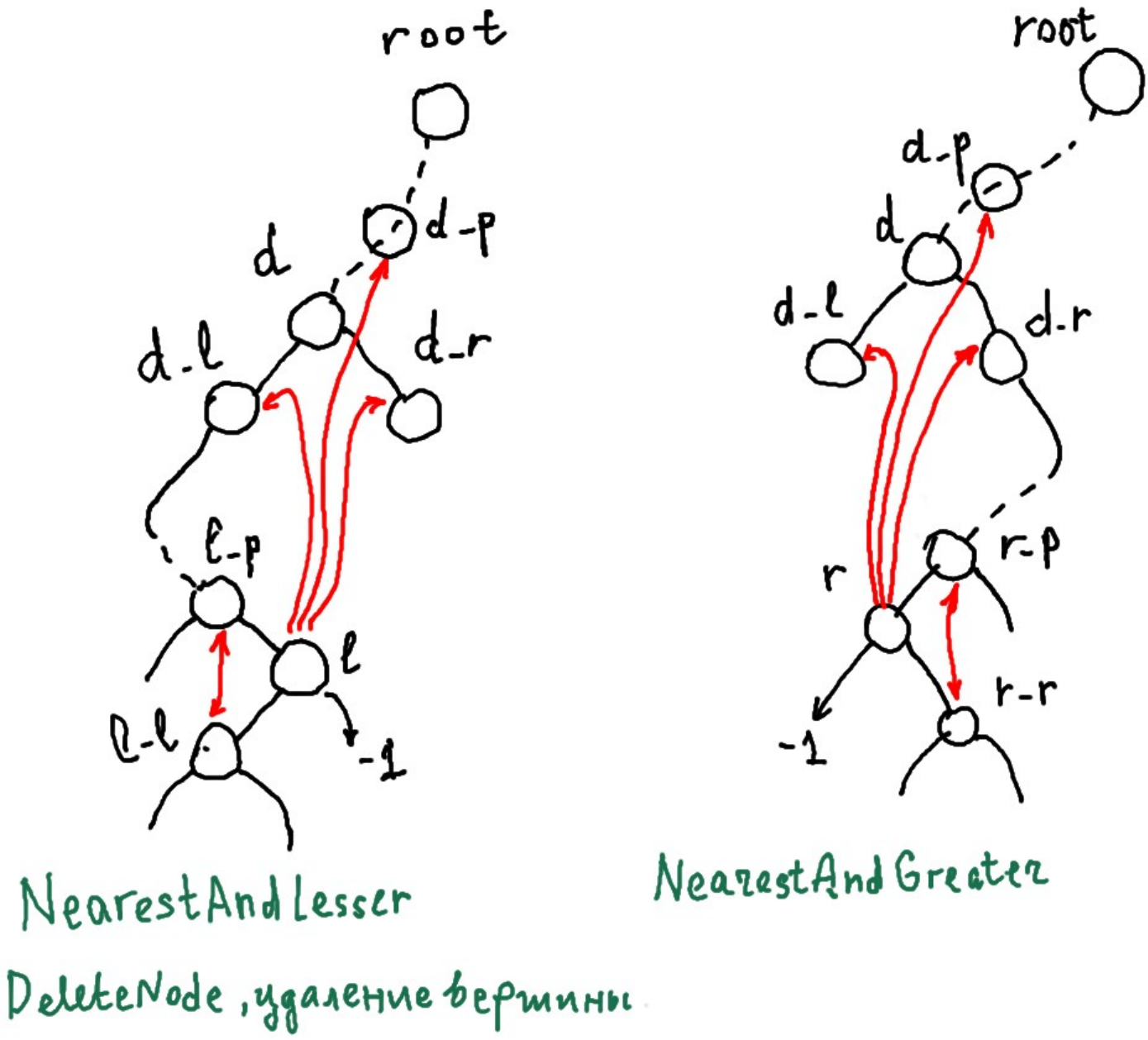


Как видно, правила и случаи балансировки — немногочисленны и просты.
Вот код функции балансировки, Maintain:

```
int SBT_Maintain(TNodeIndex t) {

        ...

        // поместили слева (?)
        if (SBT_Left_Left_size(t) > SBT_Right_size(t)) {
                SBT_RightRotate(t);
                CALC_T0 // восстановить значение t0
                SBT_Maintain(_nodes[t0].right);
                SBT_Maintain(t0);
        }
        else if (SBT_Left_Right_size(t) > SBT_Right_size(t)) {
                SBT_LeftRotate(_nodes[t].left);
                SBT_RightRotate(t);
                CALC_T0
                SBT_Maintain(_nodes[t0].left);
                SBT_Maintain(_nodes[t0].right);
                SBT_Maintain(t0);
        }
        // поместили справа (?)
        else if (SBT_Right_Right_size(t) > SBT_Left_size(t)) {
                SBT_LeftRotate(t);
                CALC_T0
                SBT_Maintain(_nodes[t0].left);
                SBT_Maintain(t0);
        }
        else if (SBT_Right_Left_size(t) > SBT_Left_size(t)) {
                SBT_RightRotate(_nodes[t].right);
                SBT_LeftRotate(t);
                CALC_T0
                SBT_Maintain(_nodes[t0].left);
                SBT_Maintain(_nodes[t0].right);
                SBT_Maintain(t0);
        }

        return 0;
}
```

Есть также более быстрый (упрощённый) код — функция SBT_Maintain_Simpler().

6. Удаление вершины



Удаление вершины — это:

1 шаг) собственно вырезание вершины из дерева (находим её через Find-функцию),

2 шаг) освобождение памяти из-под этой ячейки,

3 шаг) перевешивание на место удалённой вершины — другой, наиболее близкой по значению (NearestAndLesser, NearestAndGreater),

4 шаг) балансировка поддерева, которое изменили.

Вырезание вершины — это ключевая часть функции DeleteNode.

Балансировка выполняется после вырезания и перевешивания вершины. Выполняется от непустой вершины: L_P (или R_P) вверх, до корня.

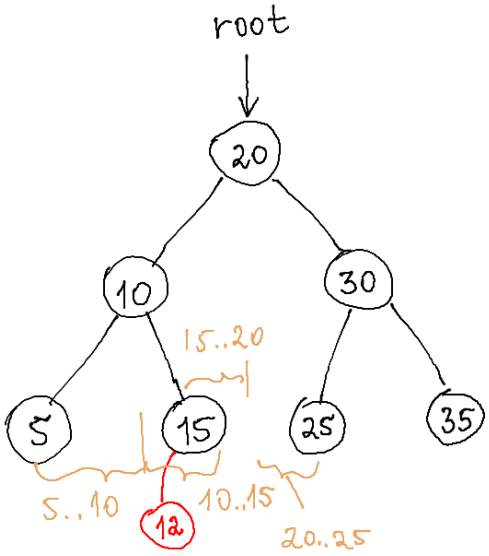
7. Добавление вершины, балансировка

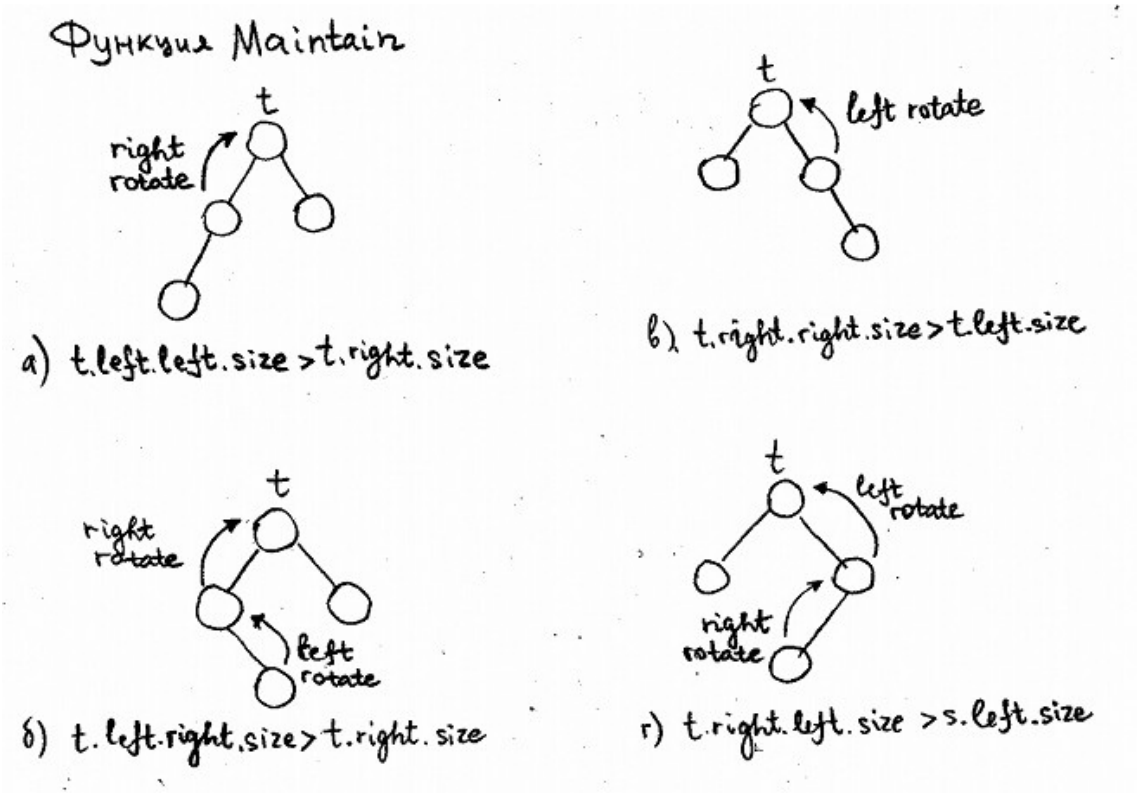
Добавление вершины происходит в три шага:

1 шаг) Выделение памяти под ячейку (node);

2 шаг) Добавление вершины (node) в дерево: все узлы дерева (ноды) делят числовую ось (область значений вершин) на интервалы; ищем интервал, которому принадлежит новый лист, соответственно, ищем лист - к которому можно подвесить новый, и добавляем; по-умолчанию — добавляем с приоритетом «вправо», поэтому «левых» вращений при балансировке больше, чем правых;

3 шаг) Упорядочивание — балансировка вершин (вызов Maintain для соответствующего поддерева и «соседних» поддеревьев); например (см. рисунок) — если добавили вершину «12» - то баланс не нарушился, функция Maintain это сама «поймёт», и вершины переставлять не будет.





8. Возможные пути оптимизации

а) Заменить Maintain на Maintain_Simpler в функции DeleteNode().
б) Интеграция балансировки (Maintain) в функции AddNode и DeleteNode.
в) Для ускорения прохода дерева можно преобразовать структуру дерева в прошитое (threaded).