*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 IA-64. Предполагаемое ускорение — порядка десятикратного.*

*Содержание работы: Написать код на языке Си с ассемблерными вставками, либо на чистом ассемблере в AT&T-синтаксисе для IA-64.*

*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: PDF-файл с иллюстрациями, диаграммами — поясняющими работу оптимизированного варианта, результаты тестов.*

*Требования к работе:*

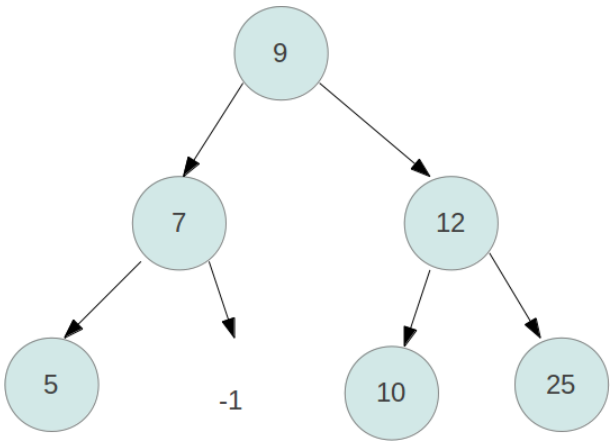
*Результаты работы должны быть опубликованы на ресурсе GitHub:*
*- исходный код - под лицензией GNU Affero GPLv3,*
*- изображения - под лицензией Creative Commons BY-SA 3.0,*
*- тексты документации - под лицензией Creative Commons BY-SA 3.0.*

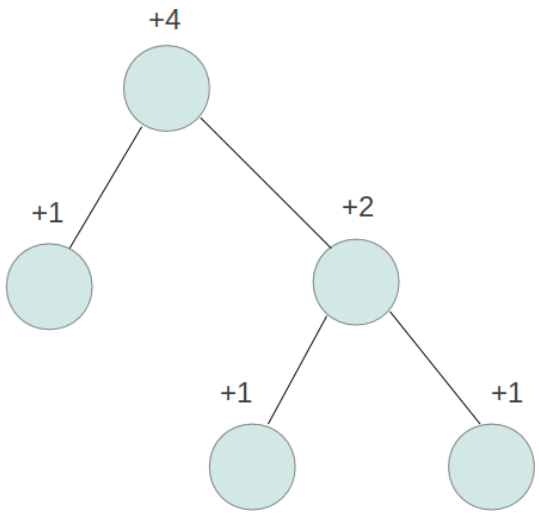
*Описание оригинального кода*

*1. Основной тип данных (структура данных):*

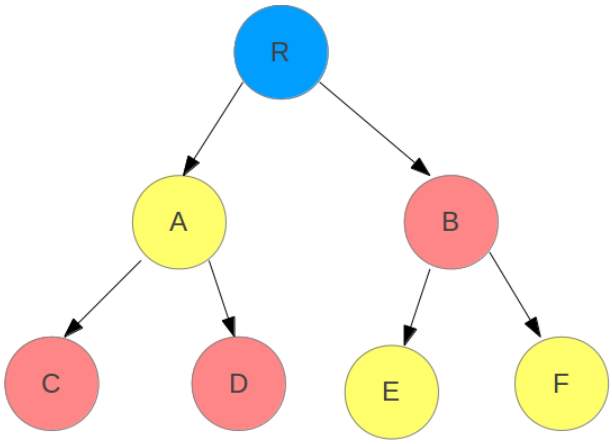
```
typedef struct TNode {
    TNumber value; // значение, привязанное к ноде
    TNodeIndex parent; // ссылка на уровень выше
    TNodeIndex left;  // ссылка на левое поддерево, = -1, если нет дочерних вершин
    TNodeIndex right; // ссылка на правое поддерево
    TNodeSize size; // size в понимании SBT
    int unused; // «удалённая»; это поле можно использовать и для других флагов
} TNode;
```

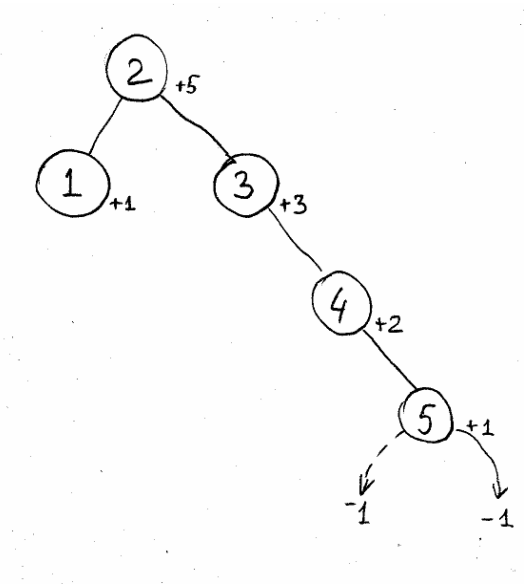
*2. Пример дерева: значения вершин (слева) и размеры вершин (справа)*



*3. Условие сбалансированности*



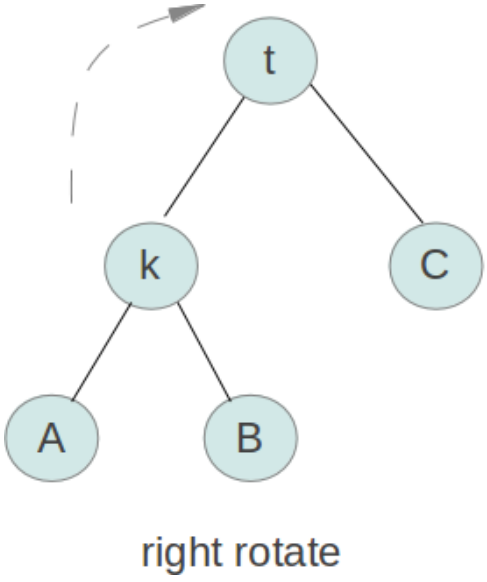




*условие сбалансированности задаётся неравенствами:*
*E.size <= A.size  C.size <= B.size*
*F.size <= A.size  D.size <= B.size*
*(то есть, дерево как бы выровнено по весу горизонтальных (соседних) слоёв)*

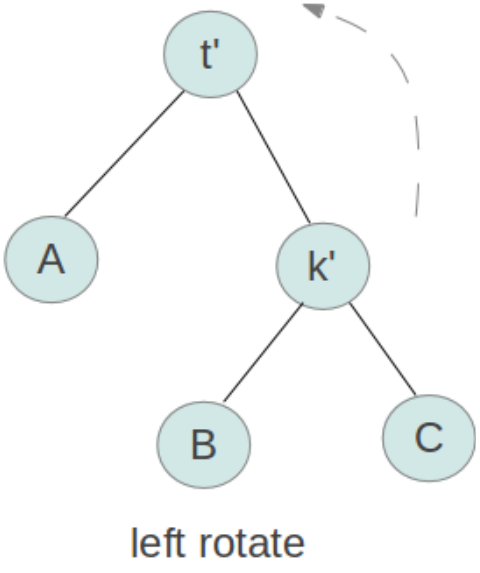
*размеры (size) вершин определяются правилом:*
*R.size = A.size + B.size + 1*
*(то есть, размер вершины — это число вершин в соответствующем поддереве)*

*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).*

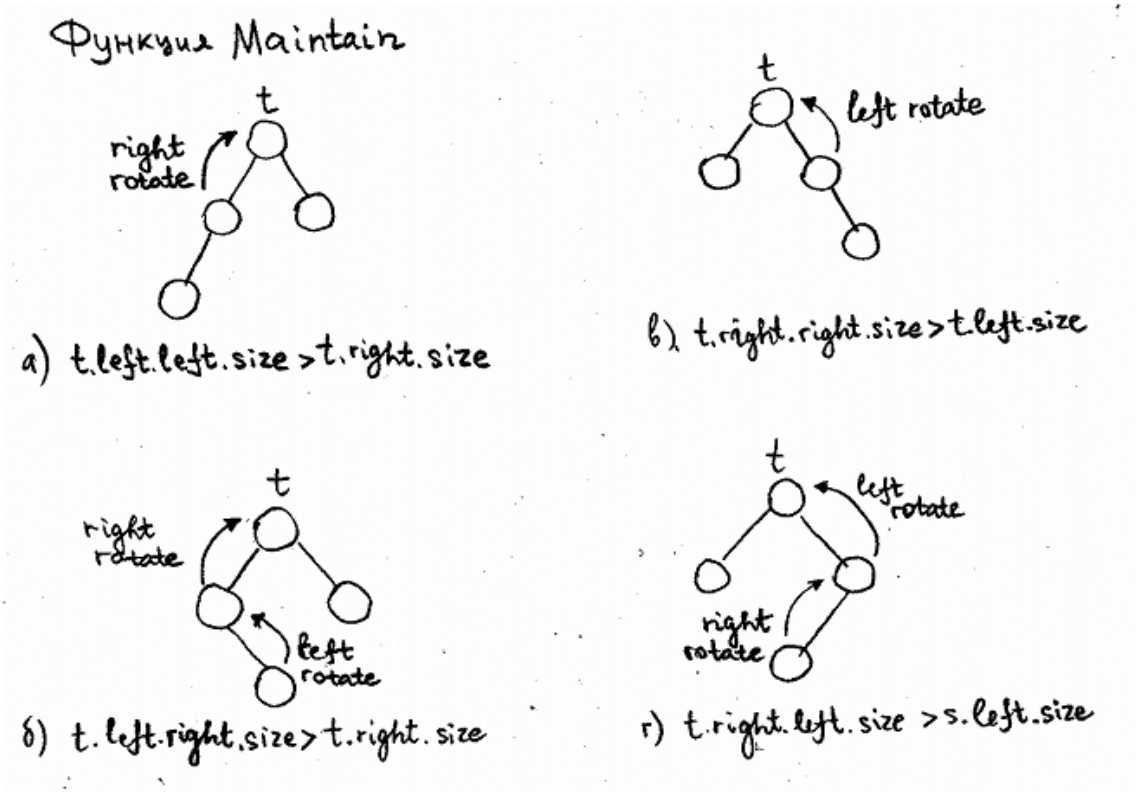
*Алгоритм вращения (left или right rotate), более подробно — sbt.c*

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.

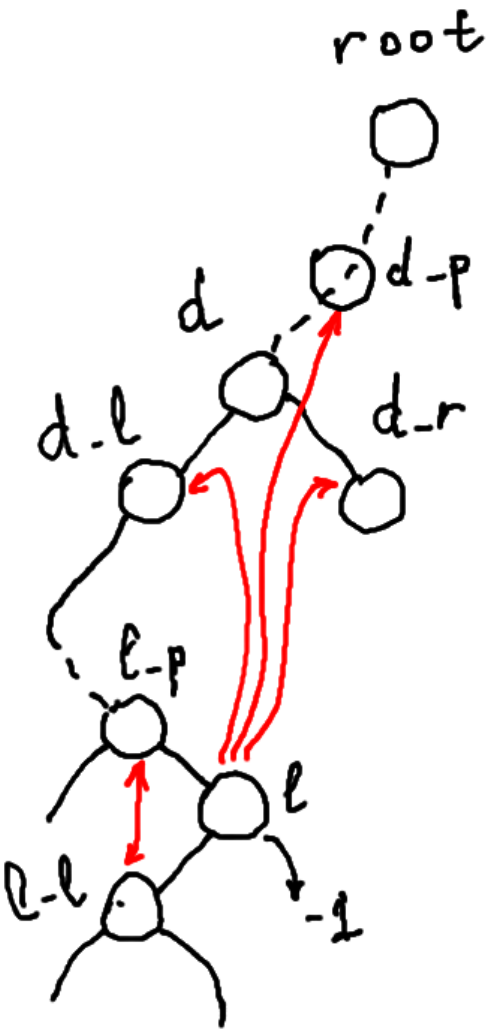
```
int SBT_LeftRotate(TNodeIndex t) {

        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»)*

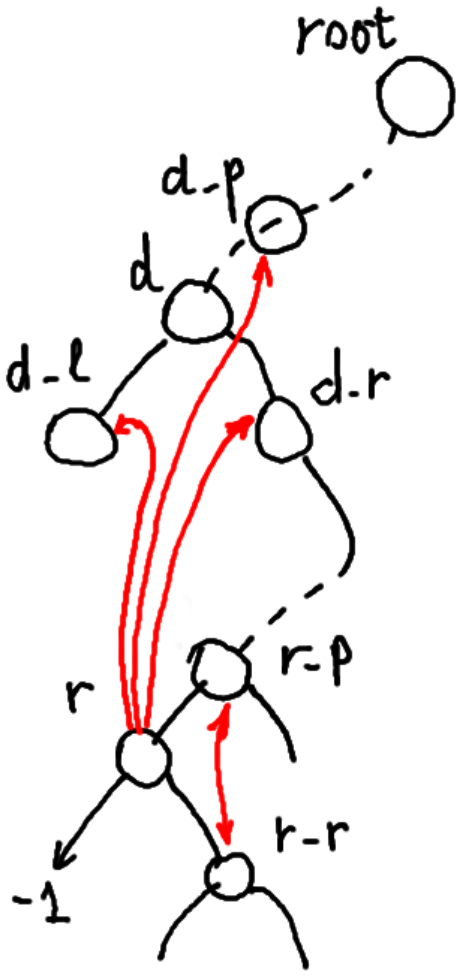
*5. Правила балансировки*



*6. Удаление вершины*




NearestAndLesser




NearestAndGreater

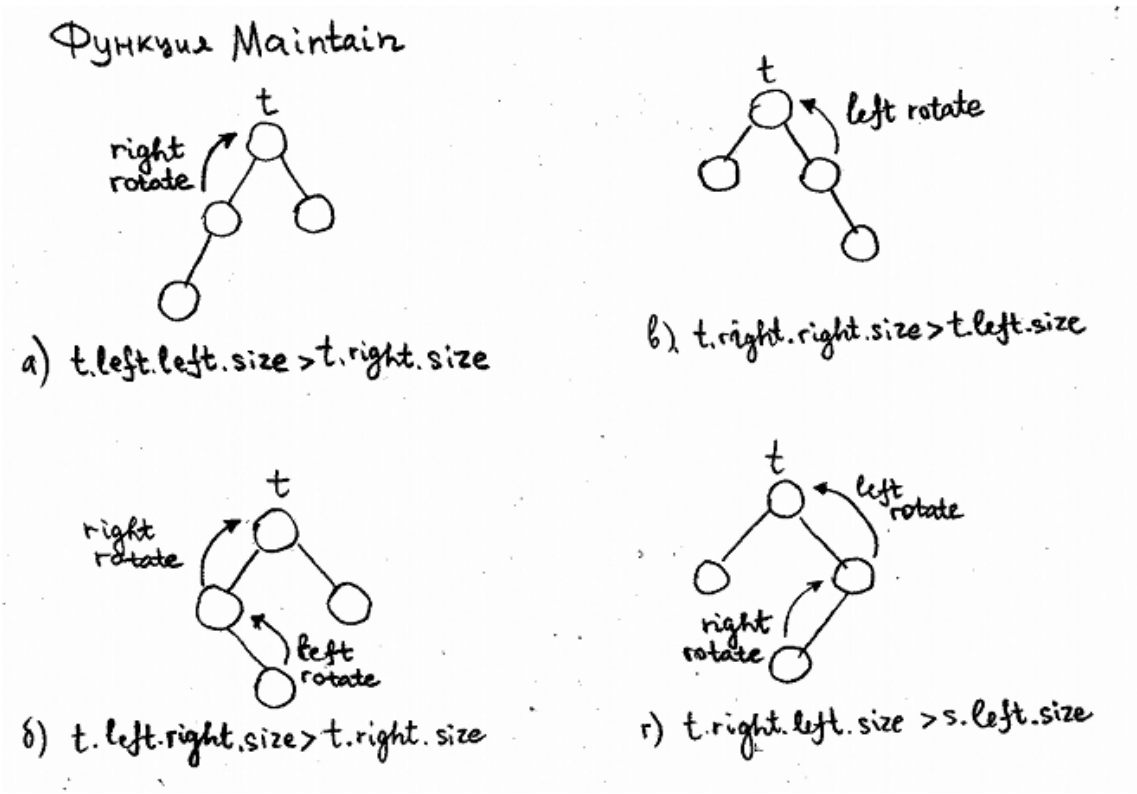
DeleteNode, удаление вершины

*Удаление вершины — это:*
*1) собственно вырезание вершины из дерева,*
*2) освобождение памяти из-под этой ячейки,*
*3) перевешивание на место удалённой вершины — другой, наиболее близкой по значению (NearestAndLesser, NearestAndGreater),*
*4) балансировка поддерева, которое изменили.*

*Вырезание вершины — это ключевая часть функции DeleteNode.*

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

*7. Добавление вершины, балансировка*




*Добавление вершины происходит в три шага:*
*1) Выделение памяти под ячейку (node),*
*2) Добавление вершины (node) в дерево,*
*3) Упорядочивание — балансировка вершин (вызов Maintain для соответствующего поддерева и «соседних» поддеревьев).*