Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической
кибернетики и компьютерных наук

СТРУКТУРЫ ДАННЫХ ДЛЯ ПОИСКА ПО КЛЮЧУ. СРАВНИТЬ КАК МАСШТАБИРУЮТСЯ РАЗНЫЕ СТРУКТУРЫ – КРАСНО-ЧЕРНОЕ ДЕРЕВО, ДЕРАМИДА, AVL-ДЕРЕВО.

КУРСОВАЯ РАБОТА

студента 2 курса 251 группы
направления 09.03.04 – Программная инженерия
факультета КНиИТ
Ромащенко Егора Сергеевича

Научный руководитель
доцент, к. ф.-м. н. _____ Г. Г. Наркайтис

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной курсовой работы является изучение структур данных для поиска по ключу, реализация красно-черного дерева, дерамиды, AVL-дерева, сравнение масштабируемости данных трех структур.

1 Классификация структур данных для поиска по ключу

Поисковая структура данных — любая структура данных реализующая эффективный поиск конкретных элементов множества, например, конкретной записи в базе данных.

Простейшей, наиболее общей, но менее эффективной поисковой структурой является простая неупорядоченная последовательная всех элементов. Расположив элементы в такой список, неизбежно возникнет ряд операций, которые потребуют линейного времени, в худшем случае, а также в среднем случае. Используемые в реальной жизни поисковые структуры данных позволяют совершать операции более быстро, однако они ограничены запросами некоторого конкретного вида. Кроме того, поскольку стоимость построения таких структур пропорциональна количеству исходных элементов, их построение окупится, даже если поступает лишь несколько запросов.

Некоторые структуры данных, позволяющие поиск по ключу:

- отсортированный по ключам массив (бинарный поиск);
- сбалансированные двоичные деревья поиска;
- хеш-таблицы.

Статические поисковые структуры данных предназначены для ответа на запросы на фиксированной базе данных.

Динамические поисковые структуры также позволяют вставки, удаления или модификации элементов между последовательными запросами. В динамическом случае, необходимо также учитывать стоимость изменения структуры данных. Любую динамическую структуру данных можно сделать статической, если запретить вставку и удаление. Также если множество ключей известно, то можно его заранее упорядочить так, чтобы избежать худших случаев в поисках в структурах данных.

Наиболее важная классификация — по времени. Мы рассмотрим три структуры данных, которые в среднем отвечают на запросы вставки, удаления, поиска за $O(\log n)$, где n — количество элементов.

2 Используемые структуры

2.1 AVL-дерево

АВЛ-дерево (англ. AVL-Tree) — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона – Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962 году [1].

Обозначим за $h(T)$ высоту дерева T. Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $|h(R) - h(L)| = 2$, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева $|h(R) - h(L)| \leq 1$, иначе ничего не меняет. Для балансировки будем хранить для каждой вершины разницу между высотами её правого и левого поддеревьев `balance` $= h(R) - h(L)$:

```
typedef struct AVL_node {
    int data;
    // Разность между высотами правого и левого поддеревьев
    int balance;
    /* Используем массив из двух указателей(левый сын - 0),
       чтобы избежать симметричных случаев */
    struct AVL_node *link[2];
} AVL_node, *p_AVL_node;
```

Для балансировки вершины используются вращения:

```
// Функция малого поворота, dir - направление вращения (левый поворот - dir = 0)
p_AVL_node AVL_single_rot(p_AVL_node root, int dir) {
    p_AVL_node save = root->link[!dir];

    root->link[!dir] = save->link[dir];
    save->link[dir] = root;

    return save;
}

// Большое вращение
p_AVL_node AVL_double_rot(p_AVL_node root, int dir) {
    p_AVL_node save = root->link[!dir]->link[dir];

    root->link[!dir]->link[dir] = save->link[!dir];
    save->link[!dir] = root->link[!dir];
    root->link[!dir] = save;

    save = root->link[!dir];
    root->link[!dir] = save->link[dir];
    save->link[dir] = root;

    return save;
}
```

И вспомогательные функции:

```
// Установка баланса перед большим поворотом
void adjust_balance(p_AVL_node root, int dir, int bal) {
    p_AVL_node n = root->link[dir];
    p_AVL_node nn = n->link[!dir];

    if (nn->balance == 0) {
        root->balance = n->balance = 0;
    }
    else if (nn->balance == bal) {
        root->balance = -bal;
        n->balance = 0;
    }
    else { // nn->balance == -bal
        root->balance = 0;
        n->balance = bal;
    }

    nn->balance = 0;
}

// Балансировка после вставки
p_AVL_node insert_balance(p_AVL_node root, int dir) {
    p_AVL_node n = root->link[dir];
    int bal = dir == 0 ? -1 : +1;

    if (n->balance == bal) {
        root->balance = n->balance = 0;
        root = AVL_single_rot(root, !dir);
    }
    else { // n->balance == -bal
        adjust_balance(root, dir, bal);
        root = AVL_double_rot(root, !dir);
    }

    return root;
}
```

Функция вставки новой вершины:

```
// Добавление вершины
p_AVL_node insert_node(p_AVL_node root, int data, int *done) {
    if (!root) {
        root = AVL_make_node(data);
    }
    else {
        int dir = root->data < data;

        root->link[dir] = insert_node(root->link[dir], data, done);

        if (!*done) {
            // Обновление баланса
            root->balance += dir == 0 ? -1 : +1;

            // Балансировка(если нужно) и выход из функции
            if (root->balance == 0) {
                *done = 1;
            }
            else if (abs(root->balance) > 1) {
```

```
                root = insert_balance(root, dir);
                *done = 1;
            }
        }
    }

    return root;
}
```

Так как в процессе добавления вершины мы рассматриваем не более, чем $O(h)$ вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log n)$ операций.

Полный код реализации АВЛ-дерева приведен в приложении А.

2.2 Красно-черное дерево

Красно-чёрное дерево (англ. red-black tree) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. red) и "чёрный" (англ. black).

Изобретателем красно-чёрного дерева считают немца Рудольфа Байера. Название «красно-чёрное дерево» структура данных получила в статье Л. Гимпаса и Р. Седжвика (1978) [2].

К красно-чёрным деревьям применяются следующие требования:

— узел либо красный, либо чёрный;
— корень — чёрный;
— оба потомка каждого красного узла — чёрные;
— всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

Чтобы поддерживать баланс для каждой вершины будем хранить её цвет:

```
typedef struct RB_node {
    int red; // Цвет (1 = вершина красная)
    int data;
    /* Используем массив из двух указателей(левый сын - 0),
       чтобы избежать симметричных случаев */
    struct RB_node *link[2];
} RB_node, *p_RB_node;
```

Для балансировки при вставке новой вершины используются вращения:

```
// Функция малого поворота, dir - направление вращения (левый поворот - dir = 0):
p_RB_node RB_single_rot(p_RB_node root, int dir) {
    p_RB_node save = root->link[!dir];

```

```
    root->link[!dir] = save->link[dir];
    save->link[dir] = root;

    root->red = 1;
    save->red = 0;

    return save;
}

// Большое вращение
p_RB_node RB_double_rot(p_RB_node root, int dir) {
    root->link[!dir] = RB_single_rot(root->link[!dir], !dir);

    return RB_single_rot(root, dir);
}
```

Функция вставки новой вершины:

```
// Добавление вершины
p_RB_node RB_insert_node(p_RB_node root, int data) {
    if (!root) {
        root = RB_make_node(data);
    }
    else if (data != root->data) {
        int dir = root->data < data;

        root->link[dir] = RB_insert_node(root->link[dir], data);

        // Балансировка
        if (is_red(root->link[dir])) {
            if (is_red(root->link[!dir])) {
                root->red = 1;
                root->link[0]->red = 0;
                root->link[1]->red = 0;
            } else {
                if (is_red(root->link[dir]->link[dir])) {
                    root = RB_single_rot(root, !dir);
                } else if (is_red(root->link[dir]->link[!dir])) {
                    root = RB_double_rot(root, !dir);
                }
            }
        }
    }

    return root;
}
```

Так как в процессе добавления вершины мы рассматриваем не более, чем $O(h)$ вершин дерева, и для каждой запускаем вращения не более одного раза, то суммарное количество операций — $O(\log n)$.

Полный код реализации красно-чёрного дерева приведен в приложении Б.

2.3 Дерамида

Декартово дерево — это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree + heap) и дерамида (дерево + пирамида)).

Более строго, это структура данных, которая хранит пары (x, y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y. Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве имеем: $y < y_0$.

Дерамиды были предложены Сиделем (Siedel) и Арагон (Aragon) в 1989 г [3].

С точки зрения реализации, каждый элемент содержит в себе x (`key`), y (`prior`) и указатели на левого `l` и правого `r` сына:

```
typedef struct treap_item {
    int key, prior;
    struct treap_item *l, *r;
} item, *p_item;
```

Для реализации процедуры добавления нового элемента понадобится вспомогательная операция `split`:

```
/* Разделяет дерево t на два дерева l и r (которые являются возвращаемым значением) таким образом,
   что l содержит все элементы, меньшие по ключу key, а r содержит все элементы, большие key. */
void split(p_item t, int key, p_item *l, p_item *r) {
    if (!t) {
        *l = *r = NULL;
    }
    else if (key < t->key) {
        split(t->l, key, l, &t->l);
        *r = t;
    }
    else {
        split(t->r, key, &t->r, r);
        *l = t;
    }
}
```

Во время выполнения вызывается одна операция `split` для дерева хотя бы на один меньшей высоты и делается ещё $O(1)$ операций. Тогда итоговая трудоёмкость этой операции равна $O(h)$, где h — высота дерева.

```
Функция вставки новой вершины (за O (log N) в среднем):
// Добавление вершины
```

```
3  void insert(p_item *t, p_item it) {
4      if (!*t) {
5          *t = it;
6      }
7      else if (it->prior > (*t)->prior){
8          split(*t, it->key, &it->l, &it->r);
9          *t = it;
10     }
11     else {
12         insert(it->key < (*t)->key ? &(*t)->l : &(*t)->r, it);
13     }
14 }
```

В декартовом дереве из n вершин, приоритеты у которого являются случайными величинами с равномерным распределением, средняя глубина вершины $O(\log n)$, а значит добавление нового элемента будет в среднем работать за $O(\log n)$.

Полный код реализации дерамиды приведен в приложении B.

3 Сравнение структур

Сравним время работы функций вставки используемых структур. Для `n` от `1` до `MAX_N` (с шагом `STEP`) `MAX_TESTS` раз сгенерируем массив из `n` элементов и вставим его в структуру. Замерим время работы вставки всего массива и возьмем среднее t. В результат запишем $\frac{t}{n}$.

```
// АВЛ
for (int n = 1; n <= MAX_N; n += STEP) {
    printf("%d ", n);

    for (int test = 0; test < MAX_TESTS; test++) {
        srand(time(NULL));

        int *a = malloc(n * sizeof(int));

        for (int i = 0; i < n; ++i) {
            a[i] = rand();
        }

        p_AVL_tree tree = malloc(sizeof(AVL_tree));
        tree->root = NULL;

        clock_t time = clock();

        for (int i = 0; i < n; ++i) {
            AVL_insert(tree, a[i]);
        }

        printf("%f ", (double) (clock() - time) / CLOCKS_PER_SEC);

        AVL_clear(tree->root);
        free(tree);
        free(a);
    }

    printf("\n");
}
```

Полный код тестирования используемых структур приведен в приложении Г.

Полученные результаты тестирования представлены на рисунке 1.

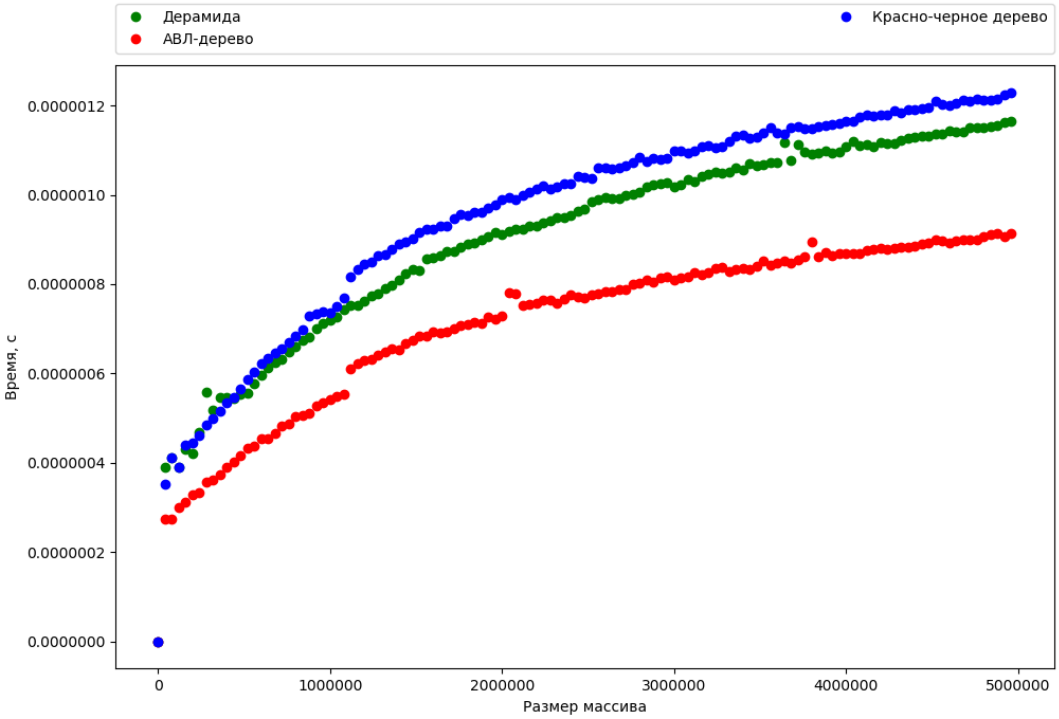

Рисунок 1 – Зависимость времени вставки от количества элементов

ЗАКЛЮЧЕНИЕ

В ходе данной курсовой работы были изучены структуры данных для поиска по ключу, реализованы: красно-черное дерево, дерамида, AVL-дерево, было проведено сравнение времени работы функции вставки данных структур.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 АВЛ-дерево - Викиконспекты [Электронный ресурс]. — URL: https://neerc.ifmo.ru/wiki/index.php?title=АВЛ-дерево (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

2 Красно-черное дерево - Викиконспекты [Электронный ресурс]. — URL: https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

3 Декартово дерево (treap, дерамида) - E-maxx.ru [Электронный ресурс]. — URL: http://e-maxx.ru/algo/treap (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

4 AVL Trees - Eternally Confuzzled [Электронный ресурс]. — URL: http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_avl.aspx (Дата обращения 02.05.2018). Загл. с экр. Яз. англ.

5 Red Black Trees - Eternally Confuzzled [Электронный ресурс]. — URL: http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx (Дата обращения 02.05.2018). Загл. с экр. Яз. англ.

6 Красно-черные деревья - AlgoList [Электронный ресурс]. — URL: http://algolist.manual.ru/ds/rbtree.php (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

7 Красно-чёрные деревья (Red black trees) - rfLinux [Электронный ресурс]. — URL: http://rflinux.blogspot.ru/2011/10/red-black-trees.html (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

8 АВЛ-деревья / Хабр - Habrahabr [Электронный ресурс]. — URL: https://habr.com/post/150732/ (Дата обращения 02.05.2018). Загл. с экр. Яз. рус.

9 *Cormen T. H.*, Introduction to Algorithms (3rd ed.) / T. H. Cormen. — MIT Press, 2009.

10 *Knuth D.*, The Art of Computer Programming / D. Knuth. — Addison-Wesley, 1968.

ПРИЛОЖЕНИЕ А
Реализация AVL-дерева

Код заголовочного файла AVL.h.

```
typedef struct AVL_node {
    int data;
    // Разность между высотами правого и левого поддеревьев
    int balance;
    /* Используем массив из двух указателей(левый сын - 0),
       чтобы избежать симметричных случаев */
    struct AVL_node *link[2];
} AVL_node, *p_AVL_node;

typedef struct AVL_tree {
    p_AVL_node root; // Корень дерева
} AVL_tree, *p_AVL_tree;

void AVL_insert(p_AVL_tree tree, int data);

void AVL_remove(p_AVL_tree tree, int data);

void AVL_clear(p_AVL_node root);
```

Код файла AVL.c.

```
#include<stdio.h>
#include<stdlib.h>

#include "AVL.h"

p_AVL_node AVL_make_node(int data) {
    p_AVL_node rn = malloc(sizeof(AVL_node));

    if (rn) {
        rn->data = data;
        rn->balance = 0;
        rn->link[0] = rn->link[1] = NULL;
    }

    return rn;
}

// Функция малого поворота, dir - направление вращения (левый поворот - dir = 0)
p_AVL_node AVL_single_rot(p_AVL_node root, int dir) {
    p_AVL_node save = root->link[!dir];

    root->link[!dir] = save->link[dir];
    save->link[dir] = root;

    return save;
}

// Большое вращение
p_AVL_node AVL_double_rot(p_AVL_node root, int dir) {
    p_AVL_node save = root->link[!dir]->link[dir];

    root->link[!dir]->link[dir] = save->link[!dir];
    save->link[!dir] = root->link[!dir];
```

```
34      root->link[!dir] = save;
35
36      save = root->link[!dir];
37      root->link[!dir] = save->link[dir];
38      save->link[dir] = root;
39
40      return save;
41  }
42
43  // Установка баланса перед большим поворотом
44  void adjust_balance(p_AVL_node root, int dir, int bal) {
45      p_AVL_node n = root->link[dir];
46      p_AVL_node nn = n->link[!dir];
47
48      if (nn->balance == 0) {
49          root->balance = n->balance = 0;
50      }
51      else if (nn->balance == bal) {
52          root->balance = -bal;
53          n->balance = 0;
54      }
55      else { // nn->balance == -bal
56          root->balance = 0;
57          n->balance = bal;
58      }
59
60      nn->balance = 0;
61  }
62
63  // Балансировка после вставки
64  p_AVL_node insert_balance(p_AVL_node root, int dir) {
65      p_AVL_node n = root->link[dir];
66      int bal = dir == 0 ? -1 : +1;
67
68      if (n->balance == bal) {
69          root->balance = n->balance = 0;
70          root = AVL_single_rot(root, !dir);
71      }
72      else { // n->balance == -bal
73          adjust_balance(root, dir, bal);
74          root = AVL_double_rot(root, !dir);
75      }
76
77      return root;
78  }
79
80  // Добавление вершины
81  p_AVL_node insert_node(p_AVL_node root, int data, int *done) {
82      if (!root) {
83          root = AVL_make_node(data);
84      }
85      else {
86          int dir = root->data < data;
87
88          root->link[dir] = insert_node(root->link[dir], data, done);
89
90          if (!*done) {
91              // Обновление баланса
92              root->balance += dir == 0 ? -1 : +1;
93
94              // Балансировка(если нужно) и выход из функции
```

```
95              if (root->balance == 0) {
96                  *done = 1;
97              }
98              else if (abs(root->balance) > 1) {
99                  root = insert_balance(root, dir);
100                 *done = 1;
101             }
102         }
103     }
104
105     return root;
106 }
107
108 void AVL_insert(p_AVL_tree tree, int data) {
109     int done = 0;
110
111     tree->root = insert_node(tree->root, data, &done);
112 }
113
114 // Балансировка после удаления
115 p_AVL_node AVL_remove_balance(p_AVL_node root, int dir, int *done) {
116     p_AVL_node n = root->link[!dir];
117     int bal = dir == 0 ? -1 : +1;
118
119     if (n->balance == -bal) {
120         root->balance = n->balance = 0;
121         root = AVL_single_rot(root, dir);
122     }
123     else if (n->balance == bal) {
124         adjust_balance(root, !dir, -bal);
125         root = AVL_double_rot(root, dir);
126     }
127     else { // n->balance == 0
128         root->balance = -bal;
129         n->balance = bal;
130         root = AVL_single_rot(root, dir);
131         *done = 1;
132     }
133
134     return root;
135 }
136
137 // Удаление вершины
138 p_AVL_node AVL_remove_node(p_AVL_node root, int data, int *done) {
139     if (root != NULL) {
140         int dir;
141
142         // Удаление найденной вершины
143         if (root->data == data) {
144             // Замена текущей вершины на подходящего потомка
145             if (root->link[0] == NULL || root->link[1] == NULL) {
146                 p_AVL_node save;
147
148                 dir = root->link[0] == NULL;
149                 save = root->link[dir];
150                 free(root);
151
152                 return save;
153             }
154             else {
155                 // Находим подходящего потомка
```

```
156                 p_AVL_node heir = root->link[0];
157 
158                 while (heir->link[1] != NULL) {
159                     heir = heir->link[1];
160                 }
161 
162                 // Копируем значение ключа
163                 root->data = heir->data;
164                 data = heir->data;
165             }
166         }
167 
168         dir = root->data < data;
169         root->link[dir] = AVL_remove_node(root->link[dir], data, done);
170 
171         if (!*done) {
172             // Обновляем баланс
173             root->balance += dir != 0 ? -1 : +1;
174 
175             // Балансировка или выход из функции
176             if (abs(root->balance) == 1) {
177                 *done = 1;
178             }
179             else if (abs(root->balance) > 1) {
180                 root = AVL_remove_balance(root, dir, done);
181             }
182         }
183     }
184 
185     return root;
186 }
187 
188 void AVL_remove(p_AVL_tree tree, int data) {
189     int done = 0;
190 
191     tree->root = AVL_remove_node(tree->root, data, &done);
192 }
193 
194 void AVL_clear(p_AVL_node root) {
195     if(root) {
196         AVL_clear(root->link[0]);
197         AVL_clear(root->link[1]);
198         free(root);
199         root = NULL;
200     }
201 }
202 
```

ПРИЛОЖЕНИЕ Б

Реализация красно-черного дерева

Код заголовочного файла RB.h.

```
typedef struct RB_node {
    int red; // Цвет (1 = вершина красная)
    int data;
    /* Используем массив из двух указателей(левый сын - 0),
       чтобы избежать симметричных случаев */
    struct RB_node *link[2];
} RB_node, *p_RB_node;

typedef struct RB_tree {
   p_RB_node root; // Корень дерева
} RB_tree, *p_RB_tree;

void RB_insert(p_RB_tree tree, int data);

void RB_remove(p_RB_tree tree, int data);

void RB_clear(p_RB_node root);
```

Код файла RB.c.

```
#include <stdlib.h>

#include "RB.h"

// Проверка цвета вершины
int is_red(p_RB_node root) {
    return root && root->red == 1;
}

// Функция малого поворота, dir - направление вращения (левый поворот - dir = 0):
p_RB_node RB_single_rot(p_RB_node root, int dir) {
    p_RB_node save = root->link[!dir];

    root->link[!dir] = save->link[dir];
    save->link[dir] = root;

    root->red = 1;
    save->red = 0;

    return save;
}

// Большое вращение
p_RB_node RB_double_rot(p_RB_node root, int dir) {
    root->link[!dir] = RB_single_rot(root->link[!dir], !dir);

    return RB_single_rot(root, dir);
}

// Создание вершины с заданным ключом
p_RB_node RB_make_node(int data) {
    p_RB_node rn = malloc(sizeof(RB_node));

    if (rn) {
```

```
35          rn->data = data;
36          rn->red = 1; // 1 - красный
37          rn->link[0] = rn->link[1] = NULL;
38      }
39
40      return rn;
41  }
42
43  // Добавление вершины
44  p_RB_node RB_insert_node(p_RB_node root, int data) {
45      if (!root) {
46          root = RB_make_node(data);
47      }
48      else if (data != root->data) {
49          int dir = root->data < data;
50
51          root->link[dir] = RB_insert_node(root->link[dir], data);
52
53          // Балансировка
54          if (is_red(root->link[dir])) {
55              if (is_red(root->link[!dir])) {
56                  root->red = 1;
57                  root->link[0]->red = 0;
58                  root->link[1]->red = 0;
59              } else {
60                  if (is_red(root->link[dir]->link[dir])) {
61                      root = RB_single_rot(root, !dir);
62                  } else if (is_red(root->link[dir]->link[!dir])) {
63                      root = RB_double_rot(root, !dir);
64                  }
65              }
66          }
67      }
68
69      return root;
70  }
71
72  void RB_insert(p_RB_tree tree, int data) {
73      tree->root = RB_insert_node(tree->root, data);
74      tree->root->red = 0;
75  }
76
77  // Балансировка после удаления
78  p_RB_node RB_remove_balance(p_RB_node root, int dir, int *done) {
79      p_RB_node p = root;
80      p_RB_node s = root->link[!dir];
81
82      if (is_red(s)) {
83          root = RB_single_rot(root, dir);
84          s = p->link[!dir];
85      }
86
87      if (s) {
88          if (!is_red(s->link[0]) && !is_red(s->link[1])) {
89              if (is_red(p)) {
90                  *done = 1;
91              }
92
93              p->red = 0;
94              s->red = 1;
95          }
```

```
96          else {
97              int save = p->red;
98              int new_root = (root == p);
99
100             if (is_red(s->link[!dir])) {
101                 p = RB_single_rot(p, dir);
102             }
103             else {
104                 p = RB_double_rot(p, dir);
105             }
106
107             p->red = save;
108             p->link[0]->red = 0;
109             p->link[1]->red = 1;
110
111             if (new_root) {
112                 root = p;
113             }
114             else {
115                 root->link[dir] = p;
116             }
117
118             *done = 1;
119         }
120     }
121
122     return root;
123 }
124
125 // Удаление вершины
126 p_RB_node RB_remove_node(p_RB_node root, int data, int *done) {
127     if (!root) {
128         *done = 1;
129     }
130     else {
131         int dir;
132
133         if (root->data == data) {
134             if (!root->link[0] || !root->link[1]) {
135                 p_RB_node save = root->link[!root->link[0]];
136
137                 if (is_red(root)) {
138                     *done = 1;
139                 }
140                 else if (is_red(save)) {
141                     save->red = 0;
142                     *done = 1;
143                 }
144
145                 free(root);
146
147                 return save;
148             }
149             else {
150                 p_RB_node heir = root->link[0];
151
152                 while(heir->link[1]) {
153                     heir = heir->link[1];
154                 }
155
156                 root->data = heir->data;
```

```
157                 data = heir->data;
158             }
159         }
160 
161         dir = root->data < data;
162         root->link[dir] = RB_remove_node(root->link[dir], data, done);
163 
164         if (!*done) {
165             root = RB_remove_balance(root, dir, done);
166         }
167     }
168 
169     return root;
170 }
171 
172 void RB_remove(p_RB_tree tree, int data) {
173     int done = 0;
174 
175     tree->root = RB_remove_node(tree->root, data, &done);
176 
177     if (tree->root) {
178         tree->root->red = 0;
179     }
180 }
181 
182 void RB_clear(p_RB_node root) {
183     if (root) {
184         RB_clear(root->link[0]);
185         RB_clear(root->link[1]);
186         free(root);
187         root = NULL;
188     }
189 }
190 
```

ПРИЛОЖЕНИЕ В

Реализация дерамиды

Код заголовочного файла `treap.h`.

```
typedef struct treap_item {
    int key, prior;
    struct treap_item *l, *r;
} item, *p_item;

void split(p_item t, int key, p_item *l, p_item *r);

void insert(p_item *t, p_item it);

void merge(p_item *t, p_item l, p_item r);

void erase(p_item *t, int key);

void clear(p_item t);

```

Код файла `treap.c`.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

#include "treap.h"

/* Разделяет дерево t на два дерева l и r (которые являются возвращаемым значением) таким образом,
   что l содержит все элементы, меньшие по ключу x, а r содержит все элементы, большие x. */
void split(p_item t, int key, p_item *l, p_item *r) {
    if (!t) {
        *l = *r = NULL;
    }
    else if (key < t->key) {
        split(t->l, key, l, &t->l);
        *r = t;
    }
    else {
        split(t->r, key, &t->r, r);
        *l = t;
    }
}

// Добавление вершины
void insert(p_item *t, p_item it) {
    if (!*t) {
        *t = it;
    }
    else if (it->prior > (*t)->prior){
        split(*t, it->key, &it->l, &it->r);
        *t = it;
    }
    else {
        insert(it->key < (*t)->key ? &(*t)->l : &(*t)->r, it);
    }
}

```

```
37 // Объединяет два поддерева l и r, и возвращает это новое дерево
38 void merge(p_item *t, p_item l, p_item r) {
39     if (!l || !r) {
40         *t = l ? l : r;
41     }
42     else if (l->prior > r->prior) {
43         merge(&l->r, l->r, r);
44         *t = l;
45     }
46     else {
47         merge(&r->l, l, r->l);
48         *t = r;
49     }
50 }
51
52 void erase(p_item *t, int key) {
53     if ((*t)->key == key) {
54         p_item save_t = *t;
55         merge(t, (*t)->l, (*t)->r);
56         free(save_t);
57         save_t = NULL;
58     }
59     else {
60         erase (key < (*t)->key ? &(*t)->l : &(*t)->r, key);
61     }
62 }
63
64
65 void inorder(p_item t) {
66     if (t) {
67         inorder(t->l);
68         printf("%d ", t->key);
69         inorder(t->r);
70     }
71 }
72
73 void clear(p_item t) {
74     if (t) {
75         clear(t->l);
76         clear(t->r);
77         free(t);
78         t = NULL;
79     }
80 }
```

ПРИЛОЖЕНИЕ Г
Тестирование структур

Код файла `main.c`.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

#include "treap.h"
#include "AVL.h"
#include "RB.h"

const int MAX_N = 5000000;
const int STEP = 40000;
const int MAX_TESTS = 10;

int main() {
    freopen("out.txt", "w", stdout);

    // Дерамида
    for (int n = 1; n <= MAX_N; n += STEP) {
        printf("%d ", n);

        for (int test = 0; test < MAX_TESTS; test++) {
            srand(time(NULL));

            int *a = malloc(n * sizeof(int));

            for (int i = 0; i < n; ++i) {
                a[i] = rand();
            }

            p_item t = NULL;

            clock_t time = clock();

            for (int i = 0; i < n; ++i) {
                p_item it = malloc(sizeof(item));
                it->l = it->r = NULL;
                it->prior = rand();
                it->key = a[i];

                insert(&t, it);
            }

            printf("%f ", (double) (clock() - time) / CLOCKS_PER_SEC);

            clear(t);
            free(a);
        }

        printf("\n");
    }

    // АВЛ
    for (int n = 1; n <= MAX_N; n += STEP) {
        printf("%d ", n);
```

```
56
57          for (int test = 0; test < MAX_TESTS; test++) {
58              srand(time(NULL));
59
60              int *a = malloc(n * sizeof(int));
61
62              for (int i = 0; i < n; ++i) {
63                  a[i] = rand();
64              }
65
66              p_AVL_tree tree = malloc(sizeof(AVL_tree));
67              tree->root = NULL;
68
69              clock_t time = clock();
70
71              for (int i = 0; i < n; ++i) {
72                  AVL_insert(tree, a[i]);
73              }
74
75              printf("%f ", (double) (clock() - time) / CLOCKS_PER_SEC);
76
77              AVL_clear(tree->root);
78              free(tree);
79              free(a);
80          }
81
82          printf("\n");
83      }
84
85      // Красно-черное
86      for (int n = 1; n <= MAX_N; n += STEP) {
87          printf("%d ", n);
88
89          for (int test = 0; test < MAX_TESTS; test++) {
90              srand(time(NULL));
91
92              int *a = malloc(n * sizeof(int));
93
94              for (int i = 0; i < n; ++i) {
95                  a[i] = rand();
96              }
97
98              p_RB_tree tree = malloc(sizeof(RB_tree));
99              tree->root = NULL;
100
101             clock_t time = clock();
102
103             for (int i = 0; i < n; ++i) {
104                 RB_insert(tree, a[i]);
105             }
106
107             printf("%f ", (double) (clock() - time) / CLOCKS_PER_SEC);
108
109             RB_clear(tree->root);
110             free(tree);
111             free(a);
112         }
113
114         printf("\n");
115     }
116
```

```
117     return 0;
118 }
119
```