Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: А. К. Киреев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №2

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до $2^{64} - 1$..

Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

+ **word 34** — добавить слово «word» с номером 34 в словарь. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

- word — удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

word — найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! Load /path/to/file — загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутсвие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Вариант структуры данных: Красно-чёрное дерево.

1 Описание

Требуется написать реализацию структуры данных «красно-чёрное дерево» (RB tree). Как сказано в [1]: «Красно-чёрное дерево представляет собой бинарное дерево поиска с одним дополнительным байтом цвета в каждом узле. Цвет узла может быть либо красным, либо чёрным... Бинарное дерево поиска является красно-чёрным деревом, если оно удволетворяет следующим свойствам:
1. Каждый узел является либо красным, либо чёрным.
2. Корень дерева является чёрным узлом.
3. Каждый лист дерева (NULL) является чёрным узлом.
4. Если узел красный, то оба его дочерних узла чёрные.
5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество чёрных узлов.
В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-чёрном дереве не отличается от другого по длине более чем в два раза, так что красно-чёрные деревья являются приближенно сбалансированными.»
Красно-чёрное дерево поддерживает операции вставки, удаления и поиска в дереве, как и обычное бинарное дерево. Однако данные операции даже в худшем случае гарантируют время выполнения $O(log(N))$. В целом, сложность этих операций напрямую зависит от высоты дерева. Красно-чёрное дерево с N внутренними узлами имеет высоту, не превышающую $2log(N+1)$.
Доказательство. Докажем по индукции, что для любого x дерево с корнем в x содержит как минимум $2^{bh(x)}-1$ внутренних узлов, где $bh(x)$ — чёрная высота вершины x. Если чёрная высота x равна 0, то узел x - терминальный, значит по формуле поддерево узла x содержит не менее $2^0-1=0$ внутренних узлов. Теперь рассмотрим общий случай для вершины x, которая имеет высоту $bh(x)$, каждый дочерний узел имеет высоту либо $bh(x)$, либо $bh(x)-1$. Предположим, что для левого и правого поддеревьев x формула верна, тогда в случае, когда левое и правое поддеревья имеют чёрные высоты $bh(x)-1$, по предположению индукции мы имеем, что каждый потомок x имеет как минимум $2^{bh(x)-1}-1$ внутренних вершин. Таким образом, дерево с вершиной в x имеет не меньше $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$ вершин, что показывает, что формула верна.
Очевидно, что в худшем случае высота дерева $h(x)$ не больше чем в 2 раза превышает $bh(x)$, так как как минимум половина вершин на пути к листу от корня, не считая корень, должны быть чёрными. Отсюда из $N \geq 2^{bh(x)}-1$ и $2bh(x) \geq h(x) \Rightarrow N \geq 2^{\frac{h(x)}{2}}-1$, и значит $h \leq 2log(N+1)$, ЧТД.

2 Исходный код

Давайте сначала напишем свой класс $NPair :: TPair$ для хранения пар типа «ключ-значения», где типом ключа будет массив типа $char[257]$, а типом значения будет $unsigned\ long\ long$. В нём опишем конструктор копирования и копирующий оператор присваивания.

Листинг 1: pair.hpp

```
#pragma once

namespace NPair {
    using TUll = unsigned long long;
    const TUll MAX_LEN = 256;
    struct TPair {
        char First[MAX_LEN + 1];
        TUll Second;

        TPair() = default;
        TPair(char* first, TUll second);
        TPair(const TPair& p);
        TPair& operator=(const TPair& p);
        ~TPair() = default;
    };
}
```

В файле *main.cpp* будем обрабатывать запросы, вызывая нужные методы у класса красно-чёрного дерева. В цикле *while* до конца файла считываем очередной запрос, выполняем операцию, если она завершилась успехом, то выводим соответствующее сообщение, иначе либо просто выводим сообщение о неудаче, либо прерываем операцию, как, например, при выгрузке/сохранения дерева. Так как по условию мы работаем с регистронезависимыми строками, то напишем функцию $StrToLower$, которая преобразует всю строку к нижнему регистру.

Листинг 2: main.cpp

```
using TUll = unsigned long long;
const TUll MAX_LEN = 256;

void StrToLower(char* str) {
    for (int i = 0; i < MAX_LEN && str[i] != '\0'; ++i) {
        str[i] = std::tolower(str[i]);
    }
}

void RequestHandler() {
```

```
    char req[MAX_LEN + 1];
    NRBTree::TRBTree* tPtr = new NRBTree::TRBTree;
    while (std::cin >> req) {
        if (strcmp(req,"+") == 0) {
            char key[MAX_LEN + 1];
            TUll val;
            std::cin >> key >> val;
            StrToLower(key);
            if (tPtr->Insert({key, val})) {
                std::cout << "OK\n";
            } else {
                std::cout << "Exist\n";
            }
        } else if (strcmp(req,"-") == 0) {
            char key[MAX_LEN + 1];
            std::cin >> key;
            StrToLower(key);
            if (tPtr->Remove(key)) {
                std::cout << "OK\n";
            } else {
                std::cout << "NoSuchWord\n";
            }
        } else if (strcmp(req,"!") == 0) {
            char path[MAX_LEN + 1];
            std::cin >> req;
            std::cin.get();
            std::cin.getline(path, MAX_LEN, '\n');
            bool isOK = true;
            if (strcmp(req,"Save") == 0) {
                NRBTree::TRBTree::Save(path, *tPtr, isOK);
                if (isOK) {
                    std::cout << "OK\n";
                }
            } else {
                NRBTree::TRBTree* tmpTreePtr = new NRBTree::TRBTree;
                NRBTree::TRBTree::Load(path, *tmpTreePtr, isOK);
                if (isOK) {
                    std::cout << "OK\n";
                    delete tPtr;
                    tPtr = tmpTreePtr;
                } else {
                    delete tmpTreePtr;
                }
            }
        } else {
            StrToLower(req);
            NPair::TPair ans;
            if (tPtr->Search(req, ans)) {
                std::cout << "OK: " << ans.Second << "\n";
```

```
60              } else {
61                  std::cout << "NoSuchWord\n";
62              }
63          }
64      }
65      delete tPtr;
66  }
67
68  int main() {
69      std::ios_base::sync_with_stdio(false);
70      std::cin.tie(NULL);
71      RequestHandler();
72      return 0;
73  }
```

Для написания дерева давайте сначала опишем структуру вершины красно-чёрного дерева $TRBTreeNode$, в которой будем хранить указатель на левого сына, правого сына, родителя, цвет вершины и поле с данными типа $NPair :: TPair$. Теперь создадим класс красно-чёрного дерева с одним полем — корнем дерева, и шестью пользовательскими методами: вставка, удаление, поиск, сохранение в файл, выгрузка из файла и геттер корня дерева. Для этих шести операций напишем вспомогательные приватные методы поиска, удаления, вставки, загрузки, выгрузки, а также напишем вспомогательные методы поворотов, перекраски после вставки и удаления.

Листинг 3: rb_tree.hpp

```
1   namespace NRBTree {
2       using TUll = unsigned long long;
3       const TUll MAX_LEN = 256;
4
5       enum class TColor {
6           Red,
7           Black
8       };
9
10      struct TRBTreeNode {
11          TColor Color = TColor::Black;
12          TRBTreeNode* Parent;
13          TRBTreeNode* Left;
14          TRBTreeNode* Right;
15          NPair::TPair Data;
16
17          TRBTreeNode(): Color(TColor::Black), Parent(NULL), Left(NULL), Right(NULL),
                Data() {}
18          TRBTreeNode(const NPair::TPair& p): Color(TColor::Black), Parent(NULL), Left(
                NULL), Right(NULL), Data(p) {}
19          ~TRBTreeNode() = default;
```

```
20      };
21
22      class TRBTree {
23          private:
24              TRBTreeNode* Root;
25              bool Search(char* key, NPair::TPair& res, TRBTreeNode* node);
26              bool Insert(const NPair::TPair& data, TRBTreeNode* node);
27              void Remove(TRBTreeNode* node);
28              void RemoveFixUp(TRBTreeNode* node, TRBTreeNode* nodeParent);
29              void LeftRotate(TRBTreeNode* node);
30              void RightRotate(TRBTreeNode* node);
31              void Recolor(TRBTreeNode* node);
32              void DeleteTree(TRBTreeNode* node);
33              static void RecursiveLoad(std::ifstream& fs, NRBTree::TRBTreeNode*& node,
                    bool& isOK);
34              static void RecursiveSave(std::ofstream& fs, NRBTree::TRBTreeNode* node,
                    bool& isOK);
35
36          public:
37              TRBTree(): Root(NULL) {};
38              TRBTreeNode* GetRoot() const;
39              bool Search(char key[MAX_LEN + 1], NPair::TPair& res);
40              bool Insert(const NPair::TPair& data);
41              bool Remove(const char key[MAX_LEN + 1]);
42              static void Load(const char path[MAX_LEN + 1], NRBTree::TRBTree& t, bool&
                    isOK);
43              static void Save(const char path[MAX_LEN + 1], NRBTree::TRBTree& t, bool&
                    isOK);
44              ~TRBTree();
45      };
46  }
```

Поиск в красно-чёрном дереве ничем не отличается от поиска в бинарном дереве. Мы также идём налево, если ключ, который мы ищем, меньше того, что в вершине, если ключ больше, то идём направо. Если ключи равны, то мы нашли наш элемент. Если мы дошли до NULL-листа, то такого элемента с искомым ключом нет.

Листинг 4: rb_tree.cpp(Search)

```
1       bool TRBTree::Search(char key[MAX_LEN + 1], NPair::TPair& res) {
2           return Search(key, res, Root);
3       }
4       bool TRBTree::Search(char key[MAX_LEN + 1], NPair::TPair& res, TRBTreeNode* node) {
5           if (node == NULL) {
6               return false;
7           } else if (strcmp(key, node->Data.First) == 0) {
8               res = node->Data;
9               return true;
```

```
10          } else {
11              TRBTreeNode* to = (strcmp(key, node->Data.First) < 0) ? node->Left : node->
                    Right;
12              return Search(key, res, to);
13          }
14      }
```

Вставка в красно-чёрное дерево отличается от вставки в обычное бинарное дерево поиска. При вставке нужно учитывать, что некоторые свойства дерева могут нарушиться. Новый узел в красно-чёрное дерево добавляется на место одного из листьев, окрашивается в красный цвет. Потом вызывается функция *Recolor* для восстановления свойств дерева. При вставке красной вершины может испортиться только свойство 2 и 4. Нарушение свойства 2 будем обрабатывать сразу, вне функции *Recolor*. В *Recolor* в зависимости от цвета дяди делаем либо, если он красный, окрашиваем его и отца в чёрный, деда окрашиваем в красный и запускаем алгоритм вверх от деда, либо при помощи поворотов подвешиваем поддерево с корнем-дедом за отца вершины, делая деда дочерней вершиной отца добавленной вершины. Поиск места для вставки занимает $O(log(N))$. Так как в худшем случае мы можем перекрашивать дерево рекурсивно вплоть до корня, то, зная ограничения на высоту красно-чёрного дерева, можно сказать, что будет не более чем $O(log(N))$ выполнений *Recolor*
. Задача, когда отец — правый сын деда решается зеркально.

Листинг 5: rb_tree.cpp(Recolor)

```
1   void TRBTree::Recolor(TRBTreeNode* node) {
2       TRBTreeNode* grandParent = node->Parent->Parent;
3       if (grandParent->Left == node->Parent) {
4           if (grandParent->Right != NULL && grandParent->Right->Color == TColor::Red)
                {
5               grandParent->Left->Color = TColor::Black;
6               grandParent->Right->Color = TColor::Black;
7               grandParent->Color = TColor::Red;
8               if (Root == grandParent) {
9                   grandParent->Color = TColor::Black;
10                  return;
11              }
12              if (grandParent->Parent != NULL && grandParent->Color == TColor::Red &&
                    grandParent->Parent->Color == TColor::Red) {
13                  Recolor(grandParent);
14              }
15              return;
16          } else if (grandParent->Right == NULL ||
17          (grandParent->Right != NULL && grandParent->Right->Color == TColor::Black))
                {
18              if (node == node->Parent->Left) {
19                  grandParent->Color = TColor::Red;
20                  node->Parent->Color = TColor::Black;
```

```
21                  RightRotate(grandParent);
22                  return;
23              } else {
24                  LeftRotate(node->Parent);
25                  node->Color = TColor::Black;
26                  node->Parent->Color = TColor::Red;
27                  RightRotate(node->Parent);
28                  return;
29              }
30          }
31      } else {
32          if (grandParent->Left != NULL && grandParent->Left->Color == TColor::Red) {
33              grandParent->Right->Color = TColor::Black;
34              grandParent->Left->Color = TColor::Black;
35              grandParent->Color = TColor::Red;
36              if (Root == grandParent) {
37                  grandParent->Color = TColor::Black;
38                  return;
39              }
40              if (grandParent->Parent != NULL && grandParent->Color == TColor::Red &&
                    grandParent->Parent->Color == TColor::Red) {
41                  Recolor(grandParent);
42              }
43              return;
44          } else if (grandParent->Left == NULL ||
45          (grandParent->Left != NULL && grandParent->Left->Color == TColor::Black ))
                {
46              if (node == node->Parent->Right) {
47                  grandParent->Color = TColor::Red;
48                  node->Parent->Color = TColor::Black;
49                  LeftRotate(grandParent);
50                  return;
51              } else {
52                  RightRotate(node->Parent);
53                  node->Color = TColor::Black;
54                  node->Parent->Color = TColor::Red;
55                  LeftRotate(node->Parent);
56                  return;
57              }
58          }
59      }
60  }
```

При удалении узла с двумя не листовыми потомками в обычном двоичном дереве поиска мы ищем либо наибольший элемент в его левом поддереве, либо наименьший элемент в его правом поддереве и перемещаем его значение в удаляемый узел. Затем мы удаляем узел, из которого копировали значение. Копирование значения из одного узла в другой не нарушает свойств красно-чёрного дерева, так как структура дерева

и цвета узлов не изменяются. Стоит заметить, что новый удаляемый узел не может иметь сразу два дочерних нелистовых узла, так как в противном случае он не будет являться наибольшим/наименьшим элементом. Таким образом, получается, что случай удаления узла, имеющего два нелистовых потомка, сводится к случаю удаления узла, содержащего как максимум один дочерний нелистовой узел. Удаление чёрного узла может нарушить свойства 2, 4 и 5. Удаление красного узла не требует починки дерева. При удалении будем рассматривать брата вершины, которая встала на место удаленной вершины. Если после удаления две карсные вершины встали подряд, то давайте покрасим одну из них в чёрный, востановив баланс и все свойства. Иначе, если брат красный, то сводим задачу к той, когда брат чёрный, меня цвета отца и брата и делая левый поворот. Сведя задачу к той, когда брат чёрный, мы рассматриваем детей брата. Если они оба чёрные, то мы красим брата в красный, теряя красный уже во всём поддереве с корнем в отце, запуская починку дерева уже от отца. Когда правый сын брата чёрный мы сводим задачу к той, когда правый сын красный, делая один поворот, перекрашивая брата и его левого сына. Наконец в этом случае мы делаем левый поворот, подвешивая поддерево за брата, меняем его цвет на цвет отца, цвет отца меняем на чёрный, как и цвет правого сына брата, таким образом восстанавливая баланс чёрных вершин, так как слева добавилась одна чёрная вершина. Мы проводим починку дерева, пока не дойдем до корня, либо, пока вершина, от которой мы запускаем починку, не станет красной, чтобы просто перекрасить её. Поиск удаляемой вершины занимает $O(log(N))$. Починка дерева в худшем случае занимает $(O(log(N)))$, когда мы рекурсивно подниаемся вверх в случае «оба сына брата чёрные». Задача для случая, когда рассматриваемая вершина — правый сын отца решается зеркально.

Листинг 6: rb_tree.cpp(RemoveFixUp)

```
1   void TRBTree::RemoveFixUp(TRBTreeNode* node, TRBTreeNode* nodeParent) {
2       while ((node == NULL || node->Color == TColor::Black) && node != Root) {
3           TRBTreeNode* brother;
4           if (node == nodeParent->Left) {
5               brother = nodeParent->Right;
6               if (brother->Color == TColor::Red) {
7                   brother->Color = TColor::Black;
8                   nodeParent->Color = TColor::Red;
9                   LeftRotate(nodeParent);
10                  brother = nodeParent->Right;
11              }
12              if (brother->Color == TColor::Black) {
13                  if ((brother->Left == NULL || brother->Left->Color == TColor::Black)
14                  && (brother->Right == NULL || brother->Right->Color == TColor::Black
                        )) {
15                      brother->Color = TColor::Red;
```

```
16                      node = nodeParent;
17                      if (node != NULL) {
18                          nodeParent = node->Parent;
19                      }
20                  } else {
21                      if (brother->Right == NULL || brother->Right->Color == TColor::
                            Black) {
22                          brother->Left->Color = TColor::Black;
23                          brother->Color = TColor::Red;
24                          RightRotate(brother);
25                          brother = nodeParent->Right;
26                      }
27                      brother->Color = nodeParent->Color;
28                      nodeParent->Color = TColor::Black;
29                      brother->Right->Color = TColor::Black;
30                      LeftRotate(nodeParent);
31                      break;
32                  }
33              }
34          } else {
35              brother = nodeParent->Left;
36              if (brother->Color == TColor::Red) {
37                  brother->Color = TColor::Black;
38                  nodeParent->Color = TColor::Red;
39                  RightRotate(nodeParent);
40                  brother = nodeParent->Left;
41              }
42              if (brother->Color == TColor::Black) {
43                  if ((brother->Right == NULL || brother->Right->Color == TColor::
                        Black)
44                  && (brother->Left == NULL || brother->Left->Color == TColor::Black))
                         {
45                      brother->Color = TColor::Red;
46                      node = nodeParent;
47                      if (node != NULL) {
48                          nodeParent = node->Parent;
49                      }
50                  } else {
51                      if (brother->Left == NULL || brother->Left->Color == TColor::
                            Black) {
52                          brother->Right->Color = TColor::Black;
53                          brother->Color = TColor::Red;
54                          LeftRotate(brother);
55                          brother = nodeParent->Left;
56                      }
57                      brother->Color = nodeParent->Color;
58                      nodeParent->Color = TColor::Black;
59                      brother->Left->Color = TColor::Black;
60                      RightRotate(nodeParent);
```

```
61                      break;
62                  }
63              }
64          }
65      }
66      if (node != NULL) {
67          node->Color = TColor::Black;
68      }
69  }
```

Сериализация дерева работает довольно просто: давайте обойдём дерево прямым обходом. Данные о вершине будем хранить следующим способом: запишем длину строки, значение, цвет вершины. В случае с NULL-вершинами будем записывать только длину строки, которая будет равна -1, что будет говорить о том, что вершины нет. Десериализация работает похожим образом: проходимся по файлу, строим дерево в прямом порядке, то есть вершину, левого сына, левого сына левого сына, потом правого сына отца самого левого сына и т.д. Перед считыванием всех данных о вершине мы сначала считываем длину строки, если она равна -1, то мы выходим из функции, так как это NULL-вершина, иначе записываем другие данные об узле и запускаемся от левого сына вершины, потом от правого сына вершины, после обновляя у них поле родителя. Важно передавать имеено ссылку на вершину, чтобы изменять сам указатель на сыновей, а не то, что по нему лежит. Сложность по времени равна $O(N)$ для сериализации и для десериализации, так как по каждой вершине мы проходимся только один раз.

3 Консоль

```
MacBook-Air-K:support AK$ cat ../tests/01.t
+ Zsq 10
zXG
zsq
zsq
-tSI
+ BJg 56
-bjg
+ hIL 70
+ TBA 11
KpC
MacBook-Air-K:support AK$ ../solution/solution <../tests/01.t
OK
NoSuchWord
OK: 10
OK: 10
NoSuchWord
OK
OK
OK
OK
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: красно-чёрное дерево сравнивается с map стандартной библиотеки C++, время на считывание данных не учитывается. Тест производительности состоит из трёх тестов, каждый из которых состоит из 10, 1000 и 100000 запросов.

```
[info] [2020-11-05 20:17:40] Stage #4 Benchmarking...
g++ -O3 -std=c++17 -Wextra -Wall -Werror -pedantic -Wno-sign-compare -Wno-unused-resul
-c ../solution/rb_tree.cpp
g++ -O3 -std=c++17 -Wextra -Wall -Werror -pedantic -Wno-sign-compare -Wno-unused-resul
-c ../solution/pair.cpp
g++ -O3 -std=c++17 -Wextra -Wall -Werror -pedantic -Wno-sign-compare -Wno-unused-resul
rb_tree.o pair.o benchmark.cpp -o benchmark
[info] [2020-11-05 20:17:44] Running ../tests/01.t
Count of requests is       10
==============START============
INSERT std::map time: 12ms
INSERT rb tree time: 15ms
==============================
DELETE std::map time: 1ms
DELETE rb tree time: 6ms
==============================
SEARCH std::map time: 0ms
SEARCH rb tree time: 0ms
==============END==============
[info] [2020-11-05 20:17:44] Running ../tests/02.t
Count of requests is     1000
==============START============
INSERT std::map time: 486ms
INSERT rb tree time: 911ms
==============================
DELETE std::map time: 119ms
DELETE rb tree time: 161ms
==============================
SEARCH std::map time: 86ms
SEARCH rb tree time: 98ms
==============END==============
[info] [2020-11-05 20:17:44] Running ../tests/03.t
Count of requests is   100000
==============START============
```

```
INSERT std::map time: 55534ms
INSERT rb tree time: 49214ms
==============================
DELETE std::map time: 22166ms
DELETE rb tree time: 25241ms
==============================
SEARCH std::map time: 20415ms
SEARCH rb tree time: 17663ms
==============END==============
```

Как видно из тестов, std::map на небольших тестах работает быстрее моего красно-чёрного дерева. Но с ростом числа запросов к структуре, общее время работы вставки, удаления и поиска в красно-чёрном дереве почти не отличается от общего времени работы этих методов в std::map. Такое время работы во-первых показывает, что моё красно-чёрное дерево написано правильно, ведь в реализации std::map также используется красно-чёрное дерево, а значит, что время работы std::map и моего красно-чёрного дерева не должно сильно отличаться на больших данных, а во-вторых такое время работы демонстрирует сложность $O(log(N))$ для процедур вставки, удаления и поиска.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился писать и использовать такую структуру данных, как красно-чёрное дерево. Эта структура данных, предназначенная для хранения данных, для быстрого изменения и доступа к коллекции, очень полезна для практических задач, в которых важна скорость ответа на запросы. Стоит отметить, что лучше всего применять красно-чёрное дерево для данных, чьи ключи можно легко и быстро сравнивать, в противном случае (например для строк) скорость работы наших операций над деревом существенно падает.
Помимо самого алгоритма, я также познакомился с Valgrind, с помощью которого я смог отследить и отладить несколько ошибок, связанных с утечкой памяти. Valgrind — очень полезный инструмент для отслеживания использования памяти.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2013. Перевод с английского: И. В. Красикова — 1296 с. (ISBN 978-5-8459-1794-2 (рус.))

[2] *C++ Reference.*
URL: `https://en.cppreference.com/w/` (дата обращения: 03.11.2020).

[3] *Википедия — Красно-чёрное дерево.*
URL: `https://ru.wikipedia.org/wiki/Красно-чёрное_дерево` (дата обращения: 03.11.2020).