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: Н. П. Ежов
Преподаватель: Н. С. Капралов
Группа: М8О-204Б
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 264 - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

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 деревоЭ(RB tree). Как сказано в [?]: "Красно-чёрное дерево представляет собой бинарное дерево поиска с одни дополнительным байтов цвета в каждом узле. Цвет узла может быть либо красным, либо чёрным... Бинарное дерево поиска является красно-чёрным деревом, если оно удовлетворяет следующим своёствам:
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 соответсвии 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 2*log(N+1).
Доказательство. Докажем по индукции, что для любого x дерево с корнем x содержит как минимум $2^{bh(x)}$ - 1 внутренних узлов, где $bh(x)$ - чёрная высота вершины x. Если чёрная высота 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 $2^{bh(x)} - 1$ 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 $bh(x)$ не больше чем в 2 раза превышет $bh(x)$, так 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 $2bh(x) \geq h(x) \rightarrow N \geq 2^{h(x)/2} - 1$, и значит $h \leq 2log(N + 1)$, ЧТД.

2 Исходный код

Сначала напишем свой класс *NMyStd::TItem* для хранения пар типа "ключ-значения где типом ключа будет массив типа *char[257]*, а типом значения будет *unsigned long long*.

Листинг item.hpp

```
//
// item.hpp
//

#ifndef LAB2_ITEM_HPP
#define LAB2_ITEM_HPP
const long long MAX_LEN = 256;
namespace NMyStd{
    struct TItem{
        unsigned long long Value;
        char Key[MAX_LEN+1];
    };
}
#endif //LAB2_ITEM_HPP
```

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 соответсвующее сообщение, иначе показываем сообщения о неудаче. Так как по условию мы работает с регистронезависимыми строками, то при считывании ключа через цикл привёдем его к нижнему регистру.

Листинг main.cpp

```
//
// main.cpp
//
#include <iostream>
#include <cstring>
#include <cctype>
#include "item.hpp"
#include "tree.hpp"

int main(){
    char readStr[MAX_LEN+1];
    NMyStd::TRBTree* rbTree = new NMyStd::TRBTree;
    while(scanf("%s", readStr) > 0){
        if (strcmp(readStr,"+") == 0) {
            char key[MAX_LEN + 1];
            unsigned long long val;
```

```
17              scanf("%s%llu", key, &val);
18              for (int i = 0; i < strlen(key); ++i){
19                  key[i] = (char)tolower(key[i]);
20              }
21              NMyStd::TItem item;
22              item.Value = val;
23              std::memcpy(item.Key, key, sizeof(char)*(MAX_LEN+1));
24              if (rbTree->Insert(item)) {
25                  printf("OK\n");
26              } else {
27                  printf("Exist\n");
28              }
29          } else if (strcmp(readStr,"-") == 0) {
30              char key[MAX_LEN + 1];
31              scanf("%s", key);
32              for (int i = 0; i < strlen(key); ++i){
33                  key[i] = (char)tolower(key[i]);
34              }
35              if (rbTree->Remove(key)) {
36                  printf("OK\n");
37              } else {
38                  printf("NoSuchWord\n");
39              }
40          } else if (strcmp(readStr,"!") == 0) {
41              char path[MAX_LEN + 1];
42              scanf("%s %s", readStr, path);
43              bool isOK = true;
44              if (strcmp(readStr,"Save") == 0) {
45                  NMyStd::TRBTree::Save(path, *rbTree, isOK);
46                  if (isOK) {
47                      printf("OK\n");
48                  }
49              } else {
50                  NMyStd::TRBTree* tmpTreePtr = new NMyStd::TRBTree;
51                  NMyStd::TRBTree::Load(path, *tmpTreePtr, isOK);
52                  if (isOK) {
53                      printf("OK\n");
54                      delete rbTree;
55                      rbTree = tmpTreePtr;
56                  } else {
57                      delete tmpTreePtr;
58                  }
59              }
60          } else {
61              for (int i = 0; i < strlen(readStr); ++i){
62                  readStr[i] = (char)tolower(readStr[i]);
63              }
64              NMyStd::TItem ans;
65              if (rbTree->Search(readStr, ans)) {
```

```
66                  printf("OK: %llu\n", ans.Value);
67              } else {
68                  printf("NoSuchWord\n");
69              }
70          }
71      }
72      delete rbTree;
73      return 0;
74  }
```

Для написания дерева сначала опишем структуру вершины красно-чёрного дерева *TRBNode*, в которой будем хранить указатель на левого сына, правого сына и родителя. Также будем хранть цвет вершины и поле с данными типа *TItem*. 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.
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.
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 жобавляется на место одного из листьев, окрашивается в красный цвет. Потом вызывается функция *InsertFixUp* для восстановления свойств дерева. При вставке красной вершины может испортиться только свойство 2 и 4. Нарушение свойства 2 будем обрабатывать сразу, вне функции *InsertFixUp*. В InsertFixUp в зависимости от цвета дяди делаем следующие действия. Если дядя красный, окрашиваем дядю и отца в чёрный, деда окрашиваем в красный и запускаем алгоритм вверх от деда, либо, если дядя чёрный, 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 $O(log(N))$ выполнений *InsertFixUp*. Задача, когда отец - правый сын деда решается зеркально.
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 элементои. 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 красные вершины встали подряд, то покрасим одну из них в чёрный, восстановив баланс и все свойства. Иначе, если брат красный, то сводим 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 сые 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 красный, чтобы просто перекрасить её. Поиск удаляемой вершины занимает $O(log(N))$. Починка дерева в худшем случае занимает $O(log(N))$, когда мы рекурсивно поднимаемся вверх в случае "оба сына брата чёрные". Задача для случая, когда рассматриваемая вершина - правый сын отца решается зеркально.
Сериализация дерева работает просто: обойдем дерево прямым обходом. Данные о вершине будем хранить следующим образом: 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 ... 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 именно 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.

Листинг main.cpp

```
1 //
2 // tree.hpp
3 //
4 #ifndef DISKRAN_TREE_HPP
```

```
5  #define DISKRAN_TREE_HPP
6  #include "item.hpp"
7  #include <iostream>
8  #include <fstream>
9  #include <cstring>
10
11 namespace NMyStd{
12     struct TRBNode {
13         int Color = 0;
14         TRBNode* Parent;
15         TRBNode* Left;
16         TRBNode* Right;
17         TItem Data;
18         TRBNode(): Color(0), Parent(NULL), Left(NULL), Right(NULL), Data() {}
19         TRBNode(const TItem& p): Color(0), Parent(NULL), Left(NULL), Right(NULL), Data(
               p) {}
20         ~TRBNode() = default;
21     };
22
23     class TRBTree{
24         TRBNode* Root;
25         bool Search(char* key, TItem& res, TRBNode* node);
26         bool Insert(const TItem& Data, TRBNode* node);
27         void Remove(TRBNode* node);
28         void RemoveFixUp(TRBNode* node, TRBNode* nodeParent);
29         void LeftRotate(TRBNode* node);
30         void RightRotate(TRBNode* node);
31         void InsertFixUp(TRBNode* node);
32         void DeleteTree(TRBNode* node);
33         static void RecursiveLoad(std::ifstream& fs, TRBNode*& node, bool &isOK);
34         static void RecursiveSave(std::ofstream& fs, TRBNode* node, bool &isOK);
35
36     public:
37         TRBTree(): Root(NULL) {};
38         TRBNode* GetRoot(){
39             return Root;
40         }
41         bool Search(char key[MAX_LEN + 1], TItem& res);
42         bool Insert(const TItem& Data);
43         bool Remove(const char key[MAX_LEN + 1]);
44         static void Load(const char path[MAX_LEN + 1], TRBTree& t, bool& isOK);
45         static void Save(const char path[MAX_LEN + 1], TRBTree& t, bool& isOK);
46         ~TRBTree();
47     };
48
49     bool TRBTree::Search(char key[MAX_LEN + 1], TItem& res) {
50         return Search(key, res, Root);
51     }
52     bool TRBTree::Search(char key[MAX_LEN + 1], TItem& res,TRBNode* node) {
```

```
53          if (node == NULL) {
54              return false;
55          } else if (strcmp(key, node->Data.Key) == 0) {
56              res = node->Data;
57              return true;
58          } else {
59             TRBNode* to = (strcmp(key, node->Data.Key) < 0) ? node->Left : node->Right;
60              return Search(key, res, to);
61          }
62      }
63
64      bool TRBTree::Insert(const TItem& data) {
65          if (Root == NULL) {
66              Root = new TRBNode(data);
67              Root->Color=0;
68              return true;
69          } else {
70              return Insert(data, Root);
71          }
72      }
73
74
75      bool TRBTree::Insert(const TItem& data, TRBNode* node) {
76          const char* key = data.Key;
77          if (strcmp(key, node->Data.Key) == 0) {
78              return false;
79          } else if (strcmp(key, node->Data.Key) < 0) {
80              if (node->Left == NULL) {
81                  node->Left = new TRBNode(data);
82                  node->Left->Parent = node;
83                  node->Left->Color = 1;
84                  if (node->Color == 1) {
85                      InsertFixUp(node->Left);
86                  }
87                  return true;
88              } else {
89                  return Insert(data, node->Left);
90              }
91          } else {
92              if (node->Right == NULL) {
93                  node->Right = new TRBNode(data);
94                  node->Right->Parent = node;
95                  node->Right->Color = 1;
96                  if (node->Color == 1) {
97                      InsertFixUp(node->Right);
98                  }
99                  return true;
100             } else {
101                 return Insert(data, node->Right);
```

```
102             }
103         }
104     }
105
106     void TRBTree::InsertFixUp(TRBNode* node) {
107         TRBNode* grandParent = node->Parent->Parent;
108         if (grandParent->Left == node->Parent) {
109             if (grandParent->Right != NULL && grandParent->Right->Color == 1) {
110                 grandParent->Left->Color = 0;
111                 grandParent->Right->Color = 0;
112                 grandParent->Color = 1;
113                 if (Root == grandParent) {
114                     grandParent->Color = 0;
115                     return;
116                 }
117                 if (grandParent->Parent != NULL && grandParent->Color == 1 &&
                        grandParent->Parent->Color == 1) {
118                     InsertFixUp(grandParent);
119                 }
120                 return;
121             } else if (grandParent->Right == NULL ||
122                        (grandParent->Right != NULL && grandParent->Right->Color == 0)) {
123                 if (node == node->Parent->Left) {
124                     grandParent->Color = 1;
125                     node->Parent->Color = 0;
126                     RightRotate(grandParent);
127                     return;
128                 } else {
129                     LeftRotate(node->Parent);
130                     node->Color = 0;
131                     node->Parent->Color = 1;
132                     RightRotate(node->Parent);
133                     return;
134                 }
135             }
136         } else {
137             if (grandParent->Left != NULL && grandParent->Left->Color == 1) {
138                 grandParent->Right->Color = 0;
139                 grandParent->Left->Color = 0;
140                 grandParent->Color = 1;
141                 if (Root == grandParent) {
142                     grandParent->Color = 0;
143                     return;
144                 }
145                 if (grandParent->Parent != NULL && grandParent->Color == 1 &&
                        grandParent->Parent->Color == 1) {
146                     InsertFixUp(grandParent);
147                 }
148                 return;
```

```
149             } else if (grandParent->Left == NULL ||
150                        (grandParent->Left != NULL && grandParent->Left->Color == 0 )) {
151                 if (node == node->Parent->Right) {
152                     grandParent->Color = 1;
153                     node->Parent->Color = 0;
154                     LeftRotate(grandParent);
155                     return;
156                 } else {
157                     RightRotate(node->Parent);
158                     node->Color = 0;
159                     node->Parent->Color = 1;
160                     LeftRotate(node->Parent);
161                     return;
162                 }
163             }
164         }
165     }
166 
167     bool TRBTree::Remove(const char key[MAX_LEN + 1]) {
168         TRBNode* node = Root;
169         while (node != NULL && strcmp(key, node->Data.Key) != 0) {
170             TRBNode* to = (strcmp(key, node->Data.Key) < 0) ? node->Left : node->Right;
171             node = to;
172         }
173         if (node == NULL) {
174             return false;
175         }
176         Remove(node);
177         return true;
178     }
179 
180     void TRBTree::Remove(TRBNode* node) {
181         TRBNode* toDelete = node;
182         int toDeleteColor = toDelete->Color;
183         TRBNode* toReplace;
184         TRBNode* toReplaceParent;
185         if (node->Left == NULL) {
186             toReplace = node->Right;
187             if (toReplace != NULL) {
188                 toReplace->Parent = node->Parent;
189                 toReplaceParent = node->Parent;
190             } else {
191                 toReplaceParent = node->Parent;
192                 if (node == Root) {
193                     toReplaceParent = NULL;
194                     Root = NULL;
195                 }
196             }
197             if (node->Parent != NULL) {
```

```
198                 if (node->Parent->Left == node) {
199                     node->Parent->Left = toReplace;
200                 } else {
201                     node->Parent->Right = toReplace;
202                 }
203             } else {
204                 Root = toReplace;
205             }
206         } else if (node->Right == NULL) {
207             toReplace = node->Left;
208             toReplace->Parent = node->Parent;
209             toReplaceParent = node->Parent;
210             if (node->Parent != NULL) {
211                 if (node->Parent->Left == node) {
212                     node->Parent->Left = toReplace;
213                 } else {
214                     node->Parent->Right = toReplace;
215                 }
216             } else {
217                 Root = toReplace;
218             }
219         } else {
220             TRBNode* minInRight = node->Right;
221             while(minInRight->Left != NULL) {
222                 minInRight = minInRight->Left;
223             }
224             toDelete = minInRight;
225             toDeleteColor = toDelete->Color;
226             toReplace = toDelete->Right;
227             if (toDelete->Parent == node) {
228                 if (toReplace != NULL) {
229                     toReplace->Parent = toDelete;
230                 }
231                 toReplaceParent = toDelete;
232             } else {
233                 toDelete->Parent->Left = toReplace;
234                 if (toReplace != NULL) {
235                     toReplace->Parent = toDelete->Parent;
236                 }
237                 toReplaceParent = toDelete->Parent;
238                 toDelete->Right = node->Right;
239                 toDelete->Right->Parent = toDelete;
240             }
241             if (node->Parent != NULL) {
242                 if (node->Parent->Left == node) {
243                     node->Parent->Left = toDelete;
244                 } else {
245                     node->Parent->Right = toDelete;
246                 }
```

```
247             } else {
248                 Root = toDelete;
249             }
250             toDelete->Parent = node->Parent;
251             toDelete->Left = node->Left;
252             toDelete->Left->Parent = toDelete;
253             toDelete->Color = node->Color;
254         }
255         if (toDeleteColor == 0) {
256             RemoveFixUp(toReplace, toReplaceParent);
257         }
258         delete node;
259     }
260 
261     void TRBTree::RemoveFixUp(TRBNode* node, TRBNode* nodeParent) {
262         while ((node == NULL || node->Color == 0) && node != Root) {
263             TRBNode* brother;
264             if (node == nodeParent->Left) {
265                 brother = nodeParent->Right;
266                 if (brother->Color == 1) {
267                     brother->Color = 0;
268                     nodeParent->Color = 1;
269                     LeftRotate(nodeParent);
270                     brother = nodeParent->Right;
271                 }
272                 if (brother->Color == 0) {
273                     if ((brother->Left == NULL || brother->Left->Color == 0)
274                         && (brother->Right == NULL || brother->Right->Color == 0)) {
275                         brother->Color = 1;
276                         node = nodeParent;
277                         if (node != NULL) {
278                             nodeParent = node->Parent;
279                         }
280                     } else {
281                         if (brother->Right == NULL || brother->Right->Color == 0) {
282                             brother->Left->Color = 0;
283                             brother->Color = 1;
284                             RightRotate(brother);
285                             brother = nodeParent->Right;
286                         }
287                         brother->Color = nodeParent->Color;
288                         nodeParent->Color = 0;
289                         brother->Right->Color = 0;
290                         LeftRotate(nodeParent);
291                         break;
292                     }
293                 }
294             } else {
295                 brother = nodeParent->Left;
```

```
296                 if (brother->Color == 1) {
297                     brother->Color = 0;
298                     nodeParent->Color = 1;
299                     RightRotate(nodeParent);
300                     brother = nodeParent->Left;
301                 }
302                 if (brother->Color == 0) {
303                     if ((brother->Right == NULL || brother->Right->Color == 0)
304                         && (brother->Left == NULL || brother->Left->Color == 0)) {
305                         brother->Color = 1;
306                         node = nodeParent;
307                         if (node != NULL) {
308                             nodeParent = node->Parent;
309                         }
310                     } else {
311                         if (brother->Left == NULL || brother->Left->Color == 0) {
312                             brother->Right->Color = 0;
313                             brother->Color = 1;
314                             LeftRotate(brother);
315                             brother = nodeParent->Left;
316                         }
317                         brother->Color = nodeParent->Color;
318                         nodeParent->Color = 0;
319                         brother->Left->Color = 0;
320                         RightRotate(nodeParent);
321                         break;
322                     }
323                 }
324             }
325         }
326         if (node != NULL) {
327             node->Color = 0;
328         }
329     }
330
331     void TRBTree::LeftRotate(TRBNode* node) {
332         TRBNode* rightSon = node->Right;
333         if (rightSon == NULL) {
334             return;
335         }
336         node->Right = rightSon->Left;
337         if (rightSon->Left != NULL) {
338             rightSon->Left->Parent = node;
339         }
340         rightSon->Parent = node->Parent;
341         if (node->Parent == NULL) {
342             Root = rightSon;
343         } else if (node == node->Parent->Left) {
344             node->Parent->Left = rightSon;
```

```
345         } else {
346             node->Parent->Right = rightSon;
347         }
348         rightSon->Left = node;
349         node->Parent = rightSon;
350     }
351 
352     void TRBTree::RightRotate(TRBNode* node) {
353         TRBNode* leftSon = node->Left;
354         if (leftSon == NULL) {
355             return;
356         }
357         node->Left = leftSon->Right;
358         if (leftSon->Right != NULL) {
359             leftSon->Right->Parent = node;
360         }
361         leftSon->Parent = node->Parent;
362         if (node->Parent == NULL) {
363             Root = leftSon;
364         } else if (node == node->Parent->Right) {
365             node->Parent->Right = leftSon;
366         } else {
367             node->Parent->Left = leftSon;
368         }
369         leftSon->Right = node;
370         node->Parent = leftSon;
371     }
372 
373     void TRBTree::RecursiveLoad(std::ifstream& fs, TRBNode*& node, bool& isOK) {
374         if (!isOK) {
375             return;
376         }
377         TItem data;
378         short len = 0;
379         fs.read((char*)&len, sizeof(short));
380         if (fs.bad()) {
381             std::cerr << "ERROR: Unable to read from file\n";
382             isOK = false;
383             return;
384         }
385         if (len == -1) {
386             return;
387         } else if (len != -1 && (len <= 0 || len > MAX_LEN)) {
388             std::cerr << "ERROR: Wrong file format\n";
389             isOK = false;
390             return;
391         }
392         char color;
393         for (int i = 0; i < len; ++i) {
```

```
394             fs.read(&(data.Key[i]), sizeof(char));
395             if (isalpha(data.Key[i]) == 0) {
396                 std::cerr << "ERROR: Wrong file format\n";
397                 isOK = false;
398                 return;
399             }
400         }
401         if (fs.bad()) {
402             std::cerr << "ERROR: Unable to read from file\n";
403             isOK = false;
404             return;
405         }
406         data.Key[len] = '\0';
407         fs.read((char*)&data.Value, sizeof(unsigned long long));
408         if (fs.bad()) {
409             std::cerr << "Unable to read from file\n";
410             isOK = false;
411             return;
412         }
413         fs.read((char*)&color, sizeof(char));
414         if (fs.bad()) {
415             std::cerr << "ERROR: Unable to read from file\n";
416             isOK = false;
417             return;
418         }
419         node = new TRBNode(data);
420         if (color == 'r') {
421             node->Color = 1;
422         } else if (color == 'b') {
423             node->Color = 0;
424         } else {
425             std::cout << "ERROR: Wrong file format\n";
426             isOK = false;
427             return;
428         }
429         RecursiveLoad(fs, node->Left, isOK);
430         RecursiveLoad(fs, node->Right, isOK);
431         if (node->Left != NULL) {
432             node->Left->Parent = node;
433         }
434         if (node->Right != NULL) {
435             node->Right->Parent = node;
436         }
437     }
438 
439     void TRBTree::Load(const char* path, TRBTree& t, bool& isOK) {
440         std::ifstream fs;
441         fs.open(path, std::ios::binary);
442         if (fs.fail()) {
```

```
443             std::cerr << "ERROR: Unable to open file " << path << " in read mode\n";
444             isOK = false;
445             return;
446         }
447         RecursiveLoad(fs, t.Root, isOK);
448         fs.close();
449         if (fs.fail()) {
450             std::cerr << "ERROR: Unable to close file " << path << "\n";
451             isOK = false;
452             return;
453         }
454     }
455
456     void TRBTree::RecursiveSave(std::ofstream& fs, TRBNode* t, bool& isOK) {
457         if (!isOK) {
458             return;
459         }
460         short len = 0;
461         if (t == NULL) {
462             len = -1;
463             fs.write((char*)&len, sizeof(short));
464             if (fs.bad()) {
465                 std::cerr << "ERROR: Unable to write in file\n";
466                 isOK = false;
467                 return;
468             }
469             return;
470         }
471         char color = t->Color == 0 ? 'b' : 'r';
472         for (int i = 0; i < MAX_LEN && t->Data.Key[i] != '\0' && isalpha(t->Data.Key[i
                ]) != 0; ++i) {
473             len++;
474         }
475         fs.write((char*)&len, sizeof(short));
476         if (fs.bad()) {
477             std::cerr << "ERROR: Unable to write in file\n";
478             isOK = false;
479             return;
480         }
481         fs.write(t->Data.Key, sizeof(char) * len);
482         if (fs.bad()) {
483             std::cerr << "ERROR: Unable to write in file\n";
484             isOK = false;
485             return;
486         }
487         fs.write((char*)&(t->Data.Value), sizeof(unsigned long long));
488         if (fs.bad()) {
489             std::cerr << "ERROR: Unable to write in file\n";
490             isOK = false;
```

```
491             return;
492         }
493         fs.write((char*)&color, sizeof(char));
494         if (fs.bad()) {
495             std::cerr << "ERROR: Unable to write in file\n";
496             isOK = false;
497             return;
498         }
499         RecursiveSave(fs, t->Left, isOK);
500         RecursiveSave(fs, t->Right, isOK);
501     }
502 
503     void TRBTree::Save(const char* path, TRBTree& t, bool& isOK) {
504         std::ofstream fs;
505         fs.open(path, std::ios::binary);
506         if (fs.fail()) {
507             std::cerr << "ERROR: Unable to open file " << path << " in write mode\n";
508             isOK = false;
509             return;
510         }
511         RecursiveSave(fs, t.Root, isOK);
512         fs.close();
513         if (fs.fail()) {
514             std::cerr << "ERROR: Unable to close file " << path << "\n";
515             isOK = false;
516             return;
517         }
518     }
519 
520     void TRBTree::DeleteTree(TRBNode* node) {
521         if (node == NULL) {
522             return;
523         } else {
524             DeleteTree(node->Left);
525             DeleteTree(node->Right);
526             delete node;
527         }
528     }
529 
530     TRBTree::~TRBTree() {
531         DeleteTree(Root);
532         Root = NULL;
533     }
534 }
535 #endif //DISKRAN_TREE_HPP
```

3 Консоль

```
(base) nikita@nikita-desktop:~/Diskran/lab1$ make
g++ -O3 -std=c++14 -o solution main.cpp vector.cpp vector.h
(base) nikita@nikita-desktop:~/Diskran/lab1$ cat test_input.txt
a5db3d43b37a95cd00618a7ac3112f7e LQFZzcjKUIgaNHdxMhDRzSojQdKKdJRxqntO
d40c0348390bdb0e0fee9348cc8d2e67 vVNAzruNkZUPUUqHcmfWgkwdOGTSJeaAINZ
364590e5c89b6b1c54231793d261a3b2 wMmpTNRfxIrkciktSkSdLYabpVgehC
734bf391f564bd5aff79a1fefeb358b7 m
e31adffb1b4418881731733600387d82 vJCqcbdkeEVJLWwRbPTRWk
caf836a9f342e48deb43e8215b23b177 FbGyVbPqdyvlQoJoqmvdiaCboIkNOILfldOAPtjxbzoVUxFKSOvp
5a3378d5ca2706bffab672e07894212b QKutCZfccVqEZORVvVxPICHQYAOemh
0472d854d83d11c287ec7d9eff9b8146 CvqgXrzHlEWFRIORDYhvJH
c41d2af1b376b9fc1f95fce356012712 SYhTAeNdhtZlTAWOreQeMNPaGgFSYNRMpoldNrUDsEAIR
8a18a352c07e3af6411007385bffd0ed ibdyiGQrGFQbLZngBwpsNoMiUrIGvkQwHwYGwVSWZ
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./solution <test_input.txt
0472d854d83d11c287ec7d9eff9b8146 CvqgXrzHlEWFRIORDYhvJH
364590e5c89b6b1c54231793d261a3b2 wMmpTNRfxIrkciktSkSdLYabpVgehC
5a3378d5ca2706bffab672e07894212b QKutCZfccVqEZORVvVxPICHQYAOemh
734bf391f564bd5aff79a1fefeb358b7 m
8a18a352c07e3af6411007385bffd0ed ibdyiGQrGFQbLZngBwpsNoMiUrIGvkQwHwYGwVSWZ
a5db3d43b37a95cd00618a7ac3112f7e LQFZzcjKUIgaNHdxMhDRzSojQdKKdJRxqntO
c41d2af1b376b9fc1f95fce356012712 SYhTAeNdhtZlTAWOreQeMNPaGgFSYNRMpoldNrUDsEAIR
caf836a9f342e48deb43e8215b23b177 FbGyVbPqdyvlQoJoqmvdiaCboIkNOILfldOAPtjxbzoVUxFKSOvp
d40c0348390bdb0e0fee9348cc8d2e67 vVNAzruNkZUPUUqHcmfWgkwdOGTSJeaAINZ
e31adffb1b4418881731733600387d82 vJCqcbdkeEVJLWwRbPTRWk
```

4 Тест производительности

Время на ввод и посторение структур данных не учитывается, замеряется только время, затраченное на сортировку. Количество пар «ключ-значение» для каждого файла равно десять в степени номер теста минус один. Например, $02.t$ содержит десять пар, а $07.t$ миллион.

```
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/04.t >04.a
custom bitwise sort 6 ms
stable sort from std 0 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/05.t >05.a
custom bitwise sort 9 ms
stable sort from std 6 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/06.t >06.a
custom bitwise sort 37 ms
stable sort from std 168 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/07.t >07.a
custom bitwise sort 478 ms
stable sort from std 2371 ms
```

Глядя на результаты, видно, что $std :: stable\_sort$ выигрывает больше всего на самых маленьких тестах, а $BitWisesort$ на самых больших. Сложность $std :: stable\_sort$ $O(n * log\ n)$, а сложность $Radixsort$ $O(m * n)$, где m - количество разрядов в числе. Так как логарифм возрастающая функция, переломный момент наступает, когда $log\ n$ становится больше, чем постоянная m.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать шаблонный класс с динамическим выделением памяти на примере реализации класса $TVector$. Реализовал поразрядную сортировку и стабильную сортировку подсчетом. Узнал, что оператор копирования для массива $char$-ов работает очень медленно, соответсвенно выгодно в векторе хранить указатели на масисв, но не массив целиком. Хоть на больших тестах(10^5 и больше) соритровка по разрядам быстрее чем $std :: stable\_sort$, но пространсвенная сложность $O(n+m)$ (где n - рамер входного массива, а m - размер разряда) дает о себе знать. Также из-за сложности $O(n*m)$ поразрядная сортировка очень медленно работает для чисел, где много разрядов, из-за чего она может быть медленее $std :: stable\_sort$, поэтому конкретно в этом варианте задачи целесообразнее разделить 32-ричное число на 8 разрядов по 4 числа

Список литературы

[1] *Википедия - Красно-чёрное дерево.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево` (дата обращения: 03.11.2020).

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

[3] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.)) стр. 220 - 229 (дата обращения: 03.11.2020).