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. С. Бычков
Преподаватель: Н. К. Макаров
Группа: М8О-201Б
Дата:
Оценка:
Подпись:

Москва, 2024

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. 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.

Структура данных: Красно-чёрное дерево.

1 Описание

Требуется реализовать аналог структуры данных std::map на основе красно-чёрного дерева.

Согласно [1], **красно-черное дерево** представляет собой бинарное дерево поиска с одним дополнительным битом **цвета** в каждом узле. Цвет узла может быть либо красным, либо черным. В соответствии с накладываемыми на узлы дерева ограничениями, ни один путь в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно **сбалансированными**.

Каждый узел дерева содержит поля $color, key, left, rightp$. Если не существует дочернего или родительского узла по отношению к данному, соответствующий указатель принимает значение NIL. Мы будем рассматривать эти значения NIL как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все "нормальные" узлы, содержащие поле ключа, становятся внутренними узлами дерева.

Красно-чёрное дерево поиска - это бинарное дерево поиска, удовлетворяющее **условиям**:

1. Каждый узел является красным или черным.
2. Корень дерева является черным.
3. Каждый лист дерева (NIL) является черным.
4. Если узел — красный, то оба его дочерних узла — черные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов

Благодаря этим условиям операции $insert, erase, find$ выполняются за время $O(\log n)$.

2 Исходный код

Первым делом необходимо реализовать саму вершину дерева. Для этого опишем структуру $Node$, которая содержит поля:

- std::pair<std::string, uint64_t> val - значение узла
- Node* left - левый сын
- Node* right - правый сын
- Node* parent - родитель

Я не храню отдельно бит цвета, он у меня хранится внутри указателя на родителя. Т.к. указатель выравниваются по 8 байт, то внутри указателя последние 3 бита всегда равны нулю. Именно там я храню один бит цвета.

После этого необходимо реализовать класс красно-чёрного дерева RB. У меня он содержит всего два поля:

- Node* root - корень дерева
- uint64_t sz - размер дерева

Теперь логика основных операций над деревом:

- Вставка
  1. Выполняется процедура $find$, которая находит место, где должно быть новое значение. Если данное значение уже существует, вставка завершается.
  2. Вставляется новая вершина, которая по умолчанию красная.
  3. Вызывается процедура $insert_fixup$, которая восстанавливает свойства красно-чёрного дерева. Пока текущая вершина и родитель красные, мы, в зависимости от цвета дяди выполняем повороты и перекраски. В конце всегде перекраиваем корень в черный.
- Удаление
  1. Выполняется процедура $find$, которая находит место, где должно быть значение, которое мы хотим удалить. Если данное значение не существует, удаление завершается. Иначе, в зависимости от числа детей мы удаляем либо саму вершину, либо максимум из левого поддрева, попутмо поменяв значения местами.

2. Вызывается процедура $erase_fixup$, которая восстанавливает свойства красно-чёрного дерева. Здесь уже мы смотрим на цвет брата и на цвета его детей, в зависимости от этих параметров, делаем повороты и перекраски.

- Поиск

1. Выполняется процедура $find$, которая находит место, где должно быть значение, которое мы хотим найти. Если данное значение не существует, выбрасывается исключение. Иначе, возвращаем значение по ключу.

```
#include <iostream>
#include <string>
#include <fstream>
#include <cstdint>

namespace RB {

/*
    COMPARISON OPERATORS
    ------------------------------------------------------------------------------------------------------------

*/

bool operator<(const std::pair<std::string, uint64_t>& a, const std::pair<std::string,
     uint64_t>& b) {
    return a.first < b.first;
}

bool operator>(const std::pair<std::string, uint64_t>& a, const std::pair<std::string,
     uint64_t>& b) {
    return b < a;
}

bool operator==(const std::pair<std::string, uint64_t>& a, const std::pair<std::string
    , uint64_t>& b) {
    return a.first == b.first;
}

/*
    COMPARISON OPERATORS
    ------------------------------------------------------------------------------------------------------------

*/

class RB {
private:
    struct Node {
    public:
```

```
34          std::pair<std::string, uint64_t> val;
35          Node* left;
36          Node* right;
37          Node* parent;
38
39          Node();
40          Node(const std::pair<std::string, uint64_t>& val);
41          Node(const std::pair<std::string, uint64_t>& val, bool is_black);
42          Node(const std::pair<std::string, uint64_t> &val, Node* parent);
43          Node(const std::pair<std::string, uint64_t>& val, Node* parent, bool is_black);
44
45          ~Node() = default;
46      };
47      Node* root;
48      size_t sz;
49
50      /**
51       *  ,     .
52       *
53       * param val Значение, которое необходимо найти.* return `pair<Node*& place, Node*
             parent>` where `parent` - parent for place where `val` must be,
54       * `place` - `root` or `parent->left` if `perent->left->val == val` else `parent->
             right`
55       */
56      std::pair<Node*&, Node*> find(const std::pair<std::string, uint64_t>& val); // MB
            CONST
57
58      std::pair<Node*&, Node*> find_left_max(Node* root) const;
59
60      static void left_rotation(Node*& node);
61      static void right_rotation(Node*& node);
62
63      static bool is_black(Node* node);
64      static bool is_red(Node* node);
65      static void make_red(Node* node);
66      static void make_black(Node* node);
67      static Node* make_normal_ptr(Node* ptr);
68      static void set_parent(Node* child, Node* parent);
69
70      Node*& get_ref_to_node(Node* node);
71
72      void insert_fixup(Node* node);
73      void erase_fixup(Node* parent, bool left_bh_decreased);
74      void delete_tree(Node* node);
75
76      void serialize(std::fstream& file, Node* node);
77      Node* deserialize(std::fstream& file);
78
79      enum class SERIALIZE_TYPE : int8_t {
```

```
80          NULLPTR,
81          BLACK,
82          RED
83      };
84
85
86  public:
87
88      RB();
89      ~RB();
90
91      bool contains(const std::pair<std::string, uint64_t>& val); // MB CONST
92      bool insert(const std::pair<std::string, uint64_t>& val);
93      bool erase(const std::pair<std::string, uint64_t>& val);
94      uint64_t operator[](const std::string& str);
95      size_t size() const;
96      bool empty() const;
97
98      bool serialize(const std::string& filename);
99      bool deserialize(const std::string& filename);
100 };
101
102 /*
103     NODE
104     ------------------------------------------------------------------------------------------------------------
105 */
106
107 RB::Node::Node() : left(nullptr), right(nullptr), parent(nullptr) {}
108 RB::Node::Node(const std::pair<std::string, uint64_t>& val) : val(val), left(nullptr),
        right(nullptr), parent(nullptr) {}
109 RB::Node::Node(const std::pair<std::string, uint64_t>& val, Node* parent) : val(val),
        left(nullptr), right(nullptr), parent(parent) {}
110 RB::Node::Node(const std::pair<std::string, uint64_t>& val, Node* parent, bool
        is_black) : val(val), left(nullptr), right(nullptr), parent(parent) {
111     if (is_black) {
112         make_black(this);
113     }
114 }
115 RB::Node::Node(const std::pair<std::string, uint64_t>& val, bool is_black) : val(val),
        left(nullptr), right(nullptr), parent(nullptr) {
116     if (is_black) {
117         make_black(this);
118     }
119 }
120
121 /*
122     NODE
123     ------------------------------------------------------------------------------------------------------------
```

```
124 */
125
126
127
128 /*
129     TREE
130     ------------------------------------------------------------------------------------------------
131 */
132
133 RB::RB() : root(nullptr), sz(0) {}
134
135 size_t RB::size() const {
136     return sz;
137 }
138
139 bool RB::empty() const {
140     return sz == 0;
141 }
142
143 std::pair<RB::RB::Node*&, RB::RB::Node*> RB::find(const std::pair<std::string,
        uint64_t>& val) {
144
145     if (!root || root->val == val) {
146         return {root, nullptr};
147     }
148
149     Node* curr = root;
150     Node* prev = nullptr;
151     bool left_son = false;
152     while (curr) {
153         if (val < curr->val) {
154             prev = curr;
155             curr = curr->left;
156             left_son = true;
157         } else if (val > curr->val) {
158             prev = curr;
159             curr = curr->right;
160             left_son = false;
161         } else {
162             break;
163         }
164     }
165
166     if (left_son) {
167         return {prev->left, prev};
168     } else {
169         return {prev->right, prev};
```

```
170     }
171 }
172
173 bool RB::contains(const std::pair<std::string, uint64_t>& val) {
174     return find(val).first;
175 }
176
177 bool RB::insert(const std::pair<std::string, uint64_t>& val) {
178     std::pair<Node*&, Node*> place = find(val);
179     if (!place.first) {
180         place.first = new Node(val, place.second);
181         ++sz;
182         insert_fixup(place.first);
183         return true;
184     }
185     return false;
186 }
187
188 void RB::insert_fixup(Node* node) {
189
190     while (is_red(make_normal_ptr(node->parent))) {
191
192         Node* dad = make_normal_ptr(node->parent);
193         Node* granddad = make_normal_ptr(dad->parent);
194
195         if (granddad->left == dad) {
196             Node* uncle = granddad->right;
197             if (is_red(uncle)) {
198                 make_black(uncle);
199                 make_black(dad);
200                 make_red(granddad);
201                 node = granddad;
202             } else {
203                 if (dad->right == node) {
204                     std::swap(dad, node);
205                     left_rotation(granddad->left);
206                 }
207                 make_red(granddad);
208                 make_black(dad);
209                 if (make_normal_ptr(granddad->parent)) {
210                     right_rotation(make_normal_ptr(granddad->parent)->left == granddad ?
                                make_normal_ptr(granddad->parent)->left : make_normal_ptr(
                               granddad->parent)->right);
211                 } else {
212                     right_rotation(root);
213                 }
214             }
215
216         } else {
```

```
            Node* uncle = granddad->left;
            if (is_red(uncle)) {
                make_black(uncle);
                make_black(dad);
                make_red(granddad);
                node = granddad;
            } else {
                if (dad->left == node) {
                    std::swap(dad, node);
                    right_rotation(granddad->right);
                }
                make_red(granddad);
                make_black(dad);
                if (granddad->parent) {
                    left_rotation(make_normal_ptr(granddad->parent)->left == granddad ?
                        make_normal_ptr(granddad->parent)->left : make_normal_ptr(
                        granddad->parent)->right);
                } else {
                    left_rotation(root);
                }
            }
        }
    }
    make_black(root);
}

std::pair<RB::Node*&, RB::Node*> RB::find_left_max(Node* root) const {
    if (!root->left->right) {
        return {root->left, root};
    }
    Node* curr = root->left;
    Node* prev = nullptr;

    while (curr->right) {
        prev = curr;
        curr = curr->right;
    }

    return {prev->right, prev};
}

bool RB::erase(const std::pair<std::string, uint64_t>& val) {
    std::pair<Node*&, Node*> place = find(val);
    Node*& to_delete = place.first;
    Node* new_parent = place.second;

    if (!to_delete) {
        return false;
    }
```

```
264
265     if (!to_delete->left && !to_delete->right) {
266         bool need_fixup = false;
267         bool left_bh_decreased = false;
268         //    parent = NULL,    ,    .
269         if (is_black(to_delete) && new_parent) { //  4 .          .
270             need_fixup = true;
271             left_bh_decreased = new_parent->left == to_delete;
272         }
273
274         delete to_delete;
275         to_delete = nullptr;
276
277         if (need_fixup) {
278             erase_fixup(new_parent, left_bh_decreased);
279         }
280
281     } else if (!to_delete->left && to_delete->right) {
282         Node* deleted_node = to_delete;
283         to_delete = to_delete->right;
284         set_parent(to_delete, new_parent);
285         delete deleted_node;
286
287         make_black(to_delete); // ,  3.
288
289     } else if (to_delete->left && !to_delete->right) {
290         Node* deleted_node = to_delete;
291         to_delete = to_delete->left;
292         set_parent(to_delete, new_parent);
293         delete deleted_node;
294
295         make_black(to_delete); // ,  3.
296
297     } else {
298         bool need_fixup = false;
299         bool left_bh_decreased = false;
300
301         std::pair<Node*&, Node*> place = find_left_max(to_delete);
302         Node* to_delete_new = place.first;
303
304
305         //     .    1  4.
306         //        (  ,    )   .
307         if (is_black(to_delete_new) && !to_delete_new->left) {
308             need_fixup = true;
309             left_bh_decreased = place.second->left == to_delete_new;
310         }
311
312         std::swap(place.first->val, to_delete->val);
```

```
313
314         place.first = place.first->left;
315         if (place.first) {
316             set_parent(place.first, place.second);
317
318             make_black(place.first); // ,  3.
319         }
320
321         delete to_delete_new;
322
323         if (need_fixup) {
324             erase_fixup(place.second, left_bh_decreased);
325         }
326     }
327
328     --sz;
329     return true;
330 }
331
332 void RB::erase_fixup(Node* parent, bool left_bh_decreased) {
333     //  parent 100%
334     while (true) {
335         Node* brother = left_bh_decreased ? parent->right : parent->left;
336
337         if (is_red(brother)) { //  4,  2.  - .
338             make_red(parent);
339             make_black(brother);
340
341             Node*& parent_ref = get_ref_to_node(parent);
342             if (left_bh_decreased) {
343                 left_rotation(parent_ref);
344             } else {
345                 right_rotation(parent_ref);
346             }
347
348         } else { //  4,  3.  - .
349             if (is_black(brother->left) && is_black(brother->right)) { //  3.1
350                 bool parent_was_red = is_red(parent);
351
352                 make_red(brother);
353                 make_black(parent);
354
355                 if (parent_was_red) { //  3.1.1
356                     return;
357                 } else { //  3.1.2
358                     Node* new_parent = make_normal_ptr(parent->parent);
359
360                     if (!new_parent) { //   -> parent - . ,       .
361                         return;
```

```
362                 }
363                 left_bh_decreased = new_parent->left == parent;
364                 parent = new_parent;
365             }
366 
367         // brothers_red_son   s2
368         } else if (Node* brothers_red_son = left_bh_decreased ? brother->right :
                brother->left; is_red(brothers_red_son)) { //  3.3.
369 
370             //
371             bool parent_was_black = is_black(parent);
372 
373             if (parent_was_black) {
374                 make_black(brother);
375             } else {
376                 make_red(brother);
377             }
378             make_black(parent);
379             make_black(brothers_red_son);
380 
381             //
382             Node*& parent_ref = get_ref_to_node(parent);
383             if (left_bh_decreased) {
384                 left_rotation(parent_ref);
385             } else {
386                 right_rotation(parent_ref);
387             }
388 
389             return;
390 
391         } else { //  3.2
392             //  s1  .
393             //  ,        brother,
394             //   ,    bh,  parent.
395             Node* brother_red_son = left_bh_decreased ? brother->left : brother->
                    right;
396 
397             make_black(brother_red_son);
398             make_red(brother);
399 
400             Node*& brother_ref = get_ref_to_node(brother);
401             if (left_bh_decreased) {
402                 right_rotation(brother_ref);
403             } else {
404                 left_rotation(brother_ref);
405             }
406         }
407     }
408 
```

```
409     }
410 }
411 
412 RB::Node*& RB::get_ref_to_node(Node* node) {
413     Node* parent = make_normal_ptr(node->parent);
414 
415     if (!parent) {
416         return root;
417     }
418     return parent->left == node ? parent->left : parent->right;
419 }
420 
421 void RB::left_rotation(Node*& node) {
422     Node* a = node; //    .     .
423     Node* b = node->right; //    .     .
424     Node* parent = make_normal_ptr(node->parent);
425 
426     // Node* alpha = a->left; //    .     .
427     Node* betta = b->left; //    .     .
428     // Node* gamma = b->right; //    .     .
429 
430     node = b;
431     set_parent(b, parent);
432     b->left = a;
433     set_parent(a, b);
434     a->right = betta;
435 
436     if (betta) {
437         set_parent(betta, a);
438     }
439 }
440 
441 void RB::right_rotation(Node*& node) {
442     Node* a = node; //    .     .
443     Node* b = node->left; //    .     .
444     Node* parent = make_normal_ptr(node->parent);
445 
446     // Node* alpha = b->left; //    .     .
447     Node* betta = b->right; //    .     .
448     // Node* gamma = a->right; //    .     .
449 
450     node = b;
451     set_parent(b, parent);
452     b->right = a;
453     set_parent(a, b);
454     a->left = betta;
455 
456     if (betta) {
457         set_parent(betta, a);
```

```
458     }
459 }
460 
461 inline bool RB::is_black(Node* node) {
462     return !node || (reinterpret_cast<size_t>(node->parent) & 1ULL);
463 }
464 
465 inline bool RB::is_red(Node* node) {
466     return node && !(reinterpret_cast<size_t>(node->parent) & 1ULL);
467 }
468 
469 inline void RB::make_red(Node* node) {
470     node->parent = reinterpret_cast<Node*>(reinterpret_cast<size_t>(node->parent) & (
        UINT64_MAX - 1));
471 }
472 
473 inline void RB::make_black(Node* node) {
474     node->parent = reinterpret_cast<Node*>(reinterpret_cast<size_t>(node->parent) | 1
        ULL);
475 }
476 
477 inline RB::Node* RB::make_normal_ptr(Node* ptr) {
478     return reinterpret_cast<Node*>(reinterpret_cast<size_t>(ptr) & (UINT64_MAX - 1));
479 }
480 
481 inline void RB::set_parent(Node* child, Node* parent) {
482     if (is_black(child)) {
483         child->parent = parent;
484         make_black(child);
485     } else {
486         child->parent = parent;
487         make_red(child);
488     }
489 }
490 
491 RB::~RB() {
492     delete_tree(root);
493 }
494 
495 void RB::delete_tree(Node* node) {
496 
497     if (!node) {
498         return;
499     }
500 
501     if (node->left) {
502         delete_tree(node->left);
503     }
504 
```

```
505     if (node->right) {
506         delete_tree(node->right);
507     }
508 
509     Node* parent = make_normal_ptr(node->parent);
510     if (parent) {
511         if (parent->left == node) {
512             parent->left = nullptr;
513         } else {
514             parent->right = nullptr;
515         }
516     }
517     delete node;
518 
519 }
520 
521 uint64_t RB::operator[](const std::string& str) {
522     std::pair<Node*&, Node*> place = find({str, 0ULL});
523 
524     if (!place.first) {
525         throw std::invalid_argument("Key not found");
526     }
527 
528     return place.first->val.second;
529 }
530 
531 bool RB::serialize(const std::string& filename) {
532     std::fstream file;
533 
534     file.open(filename, std::ios::binary | std::ios::out | std::ios::trunc);
535     // if (file.fail()) {
536     // return false;
537     // }
538 
539     serialize(file, root);
540 
541     file.close();
542     return true;
543 }
544 
545 void RB::serialize(std::fstream& file, Node* node) {
546 
547     SERIALIZE_TYPE type = SERIALIZE_TYPE::NULLPTR;
548 
549     if (!node) {
550         file.write(reinterpret_cast<char*>(&type), sizeof(type));
551         return;
552     }
553 
```

```
554     type = is_black(node) ? SERIALIZE_TYPE::BLACK : SERIALIZE_TYPE::RED;
555     size_t string_size = node->val.first.size();
556 
557     file.write(reinterpret_cast<char*>(&type), sizeof(type));
558     file.write(reinterpret_cast<char*>(&string_size), sizeof(string_size));
559     file.write(node->val.first.c_str(), string_size);
560     file.write(reinterpret_cast<char*>(&node->val.second), sizeof(node->val.second));
561 
562     if (node->left) {
563         serialize(file, node->left);
564     } else {
565         type = SERIALIZE_TYPE::NULLPTR;
566         file.write(reinterpret_cast<char*>(&type), sizeof(type));
567     }
568 
569     if (node->right) {
570         serialize(file, node->right);
571     } else {
572         type = SERIALIZE_TYPE::NULLPTR;
573         file.write(reinterpret_cast<char*>(&type), sizeof(type));
574     }
575 }
576 
577 bool RB::deserialize(const std::string& filename) {
578     std::fstream file;
579     file.open(filename, std::ios::binary | std::ios::in);
580 
581 
582     // if (file.fail()) {
583     // return false;
584     // }
585 
586     delete_tree(root);
587     root = deserialize(file);
588 
589     file.close();
590     return true;
591 }
592 
593 RB::Node* RB::deserialize(std::fstream& file) {
594     SERIALIZE_TYPE type;
595 
596     file.read(reinterpret_cast<char*>(&type), sizeof(SERIALIZE_TYPE));
597 
598     if (type == SERIALIZE_TYPE::NULLPTR) {
599         return nullptr;
600     }
601 
602     ++sz;
```

```
603
604     size_t string_size;
605     std::string key;
606     uint64_t val;
607
608     file.read(reinterpret_cast<char*>(&string_size), sizeof(string_size));
609     key.resize(string_size);
610     file.read(key.data(), string_size);
611     file.read(reinterpret_cast<char*>(&val), sizeof(val));
612
613     Node* node = new Node({key, val}, type == SERIALIZE_TYPE::BLACK);
614     node->left = deserialize(file);
615     node->right = deserialize(file);
616
617     if (node->left) {
618         set_parent(node->left, node);
619     }
620
621     if (node->right) {
622         set_parent(node->right, node);
623     }
624
625     return node;
626 }
627
628 /*
629     TREE
630     ------------------------------------------------------------------------------------------------------------
631 */
632
633 };
634
635
636 using namespace std;
637
638 void lower(string& s) {
639     for (char& c : s) {
640         c = tolower(c);
641     }
642 }
643
644 int main() {
645     RB::RB tree;
646
647     string input1, input2, input3;
648
649     while (cin >> input1) {
650
```

```
651         if (input1.size() == 1 && input1[0] == '+') {
652             cin >> input2 >> input3;
653             uint64_t val = stoull(input3);
654 
655             lower(input2);
656 
657             if (tree.insert({input2, val})) {
658                 cout << "OK" << '\n';
659             } else {
660                 cout << "Exist" << '\n';
661             }
662 
663         } else if (input1.size() == 1 && input1[0] == '-') {
664             cin >> input2;
665             lower(input2);
666 
667             if (tree.erase({input2, 0ULL})) {
668                 cout << "OK" << '\n';
669 
670             } else {
671                 cout << "NoSuchWord" << '\n';
672             }
673 
674         } else if (input1.size() == 1 && input1[0] == '!') {
675             cin >> input2 >> input3;
676 
677             bool success = false;
678 
679             if (input2 == "Save") {
680                 if (tree.serialize(input3)) {
681                     success = true;
682                 }
683             } else {
684                 if (tree.deserialize(input3)) {
685                     success = true;
686                 }
687             }
688 
689             cout << "OK" << '\n';
690 
691             // if (success) {
692             // cout << "OK" << '\n';
693             // } else {
694             // cout << "ERROR: file fail" << '\n';
695             // }
696 
697         } else {
698 
699             lower(input1);
```

```
700             try {
701                 uint64_t res = tree[input1];
702                 cout << "OK: " << res << '\n';
703 
704             } catch(const std::exception& e) {
705                 cout << "NoSuchWord" << '\n';
706             }
707         }
708     }
709 }
```

Методы класса «RB» и структуры «Node».

main.cpp	
Node()	Конструктор по умолчанию
Node(const std::pair<std::string, uint64_t>& val)	Конструктор только со значением узла.
Node(const std::pair<std::string, uint64_t>& val, bool is_black)	Конструктор по значению и цвету.
Node(const std::pair<std::string, uint64_t> &val, Node* parent)	Конструктор по значению и указателю на родителя.
Node(const std::pair<std::string, uint64_t>& val, Node* parent, bool is_black)	Конструктор по значению, указателю на родителя и цвету.
Node()	Деструктор по умолчанию.
std::pair<Node*&, Node*> find(const std::pair<std::string, uint64_t>& val)()	Поиск элемента по значению
std::pair<Node*&, Node*> find_left_max(Node* root) const	Поиск максимума в левом поддереве
static void left_rotation(Node*& node)	Левый поворот
static void right_rotation(Node*& node)	Правый поворот
static bool is_black(Node* node)	Проверка, является ли вершина черной
static bool is_red(Node* node)	Проверка, является ли вершина красной
static void make_red(Node* node)	Покраска вершины в красный
static void make_black(Node* node)	Покраска вершины в черный
static Node* make_normal_ptr(Node* ptr)	Убрать бит цвета из указателя на родителя
static void set_parent(Node* child, Node* parent)	Изменить отца вершины
Node*& get_ref_to_node(Node* node)	Получить ссылку на отца
void insert_fixup(Node* node)	Починка после вставки

void erase_fixup(Node* parent, bool left_bh_decreased)	Починка после удаления
void delete_tree(Node* node)	Рекурсивное удаление вершин
void serialize(std::fstream& file, Node* node)	Рекурсивная сериализация
Node* deserialize(std::fstream& file)	Рекурсивная десериализация
RB()	Конструктор по умолчанию
~RB()	Деструктор по умолчанию
bool contains(const std::pair<std::string, uint64_t>& val)	Проверка наличия элемента
bool insert(const std::pair<std::string, uint64_t>& val)	Вставка элемента
bool erase(const std::pair<std::string, uint64_t>& val)	Удаление элемента
uint64_t operator[](const std::string& str)	Обращение к элементу по ключу
size_t size() const	Получение размера
bool empty() const	Проверка пустоты
bool serialize(const std::string& filename)	Сериализация
bool deserialize(const std::string& filename)	Десериализация

```
class RB {
private:
    struct Node {
    public:
        std::pair<std::string, uint64_t> val;
        Node* left;
        Node* right;
        Node* parent;

        Node();
        Node(const std::pair<std::string, uint64_t>& val);
        Node(const std::pair<std::string, uint64_t>& val, bool is_black);
        Node(const std::pair<std::string, uint64_t> &val, Node* parent);
        Node(const std::pair<std::string, uint64_t>& val, Node* parent, bool is_black);

        ~Node() = default;
    };
    Node* root;
    size_t sz;

    /**
     *  ,     .
```

```
23      *
24      * param val Значение, которое необходимо найти.* return `pair<Node*& place, Node*
            parent>` where `parent` - parent for place where `val` must be,
25      * `place` - `root` or `parent->left` if `perent->left->val == val` else `parent->
            right`
26      */
27      std::pair<Node*&, Node*> find(const std::pair<std::string, uint64_t>& val); // MB
            CONST
28
29      std::pair<Node*&, Node*> find_left_max(Node* root) const;
30
31      static void left_rotation(Node*& node);
32      static void right_rotation(Node*& node);
33
34      static bool is_black(Node* node);
35      static bool is_red(Node* node);
36      static void make_red(Node* node);
37      static void make_black(Node* node);
38      static Node* make_normal_ptr(Node* ptr);
39      static void set_parent(Node* child, Node* parent);
40
41      Node*& get_ref_to_node(Node* node);
42
43      void insert_fixup(Node* node);
44      void erase_fixup(Node* parent, bool left_bh_decreased);
45      void delete_tree(Node* node);
46
47      void serialize(std::fstream& file, Node* node);
48      Node* deserialize(std::fstream& file);
49
50      enum class SERIALIZE_TYPE : int8_t {
51          NULLPTR,
52          BLACK,
53          RED
54      };
55
56
57  public:
58
59      RB();
60      ~RB();
61
62      bool contains(const std::pair<std::string, uint64_t>& val); // MB CONST
63      bool insert(const std::pair<std::string, uint64_t>& val);
64      bool erase(const std::pair<std::string, uint64_t>& val);
65      uint64_t operator[](const std::string& str);
66      size_t size() const;
67      bool empty() const;
68
```

```
69 ||     bool serialize(const std::string& filename);
70 ||     bool deserialize(const std::string& filename);
71 || };
```

3 Консоль

```
g++ main.cpp -std=c++20 -Werror -Wall -Wpedantic -Wextra
cat input
+ A 1
+ a 22
+ b 2
+ c 3
! Save data.bin
A
B
C
-a
-b
-c
a
b
c
! Load data.bin
a
b
c
./a.out <input
OK
Exist
OK
OK
OK
OK: 1
OK: 2
OK: 3
OK
OK
OK
NoSuchWord
NoSuchWord
NoSuchWord
OK
OK: 1
OK: 2
OK: 3
```

4 Тест производительности

Тест производительности представляет из себя следующее: случайная вставка/удаление/поиск в словаре. Входные данные сгенерированы случайным образом. Всего 1000000 строк во входном файле. В качестве ключа выступает случайная строка длины от 1 до 5 символов, состоящая только из латинских строчных букв. В качестве значения - число от 0 до 1000. Файл сгенерирован следующей программой:

```
from random import randint, choices, choice
from string import ascii_lowercase, ascii_uppercase, digits

def string_gen(strlen : int = 10, alphabet : str = ascii_lowercase):
    return ''.join(choices(alphabet, k=strlen))

with open("benchmark_input.txt", "w") as file:

    for _ in range(1_000_000):
        operation_type = choice(['?', '+', '-'])

        strlen = randint(1, 5)
        string = string_gen(strlen)

        if operation_type == '?':
            file.write(f'? {string}')

        elif operation_type == '+':
            value = randint(0, 1000)
            file.write(f'+ {string} {value}')

        elif operation_type == '-':
            file.write(f'- {string}')
        file.write('\n')
```

```
make benchmark
./build/RB-tree_benchmark
Time RB: 1652 ms
Time std::map: 1860 ms
diff report/map_output.txt report/RB_output.txt
```

Как видно, моя версия словаря работает быстрее, чем $std :: sort$. Изначально у моё дерево работало в 1.5 раза медленне, чем $std :: map$, а именно, приблизительно за $2900ms$. Это происходило из-за того, что я выбрасывал исключение, если элемент не найден. Заменив выброс исключения на пару из $bool, uint64\_t$ мне получилось в два раза ускорить программу!!!

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я смог более детально узнать, как работает $std :: map$, т.к. в STL эта структура данных реализована именно на красно-чёрном дереве. Изначально я думал, что всё это организовано очень сложно, но выполнив эту работу, я понял, что здесь всего одна сложность - реализовать правильное восстановление свойств дерева, что делается довольно просто, когда под рукой есть Кормен или другой источник информации. После того, как реализовано восстановление свойств остаётся лишь сделать удобный для пользователя интерфейс.

Изначально, как я писал выше, у меня всё работало в полтора раза медленнее. Я думал, так и должно быть, не может же один студент написать такую структуру данных, которая будет работать быстрее «STL'ной» версии, над которой трудится огромный пласт специалистов. Однако, когда я убрал выброс исключений я был крайне удивлён и рад тому, что у меня получилось написать более эффективную версию этой структуры!!!

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-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.))

[2] *Красно-черное дерево - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево` (дата обращения: 28.04.2024).