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. Д. Волков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата: 02.05.2024
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №2

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. 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-черное дерево

Вариант ключа: `регистронезависимая последовательность букв английского алфавита длиной не более 256 символов.`

Вариант значения: `некоторый номер, от` 0 `до` $2^{64} - 1$

1 Описание

Требуется реализовать структуру красно-черного дерева, а также реализовать интерфейс для работы с ним. Для этого нам необходимо определить и реализовать некоторые методы для структуры узла дерева.

Красно-черное дерево представляет собой бинарное дерева поиска с красным или черным цветом в каждом узле. Для реализации красно-черного дерева необходимо знать несколько обязательных правил, связанных с ним:

1. Каждый узел является либо красным, либо черным.

2. Корень дерева является черным узлом.

3. Каждый лист дерева (NIL) является черным узлом.

4. Если узел красный, то оба его дочерних узла черные.

5. Для каждого узла все простые пути от него до листьев, являющихся потомкаси данного узла, содержат одно и то же количество черных узлов.

В соответствии с данными ограничениями ни один простой путь от корня в красно-черном дереве не отличается от другого по длине более чем в два раза, поэтому красно-черные деревья являются приближенно сбалансированными.[1].

2 Исходный код

Определим константу DEFAULT - начальная вместимость очереди (тип: const size_t).

Теперь опишем класс узла дерева TNode. Будем хранить несколько полей: key - в нем будем хранить ключ, по которому мы будем сравнивать узлы (тип std::string); value - номер, присвоенный каждому ключу в дереве (тип uint64_t); colour - цвет узла (тип TColour=int (enum)); left - указатель на левого сына узла (тип TNode*); right - указатель на правого сына узла (тип TNode*); parent - указатель на родителя узла (тип TNode*). У класса будет два конструктора: TNode() - конструктор по умолчанию; TNode(std::string key, uint64_t value) - конструктор по ключу и значению. Деструктор для узлов не нужен, так как освобождением памяти из под узлов будет заниматься деструктор дерева.

Теперь опишем класс самого красно-черного дерева. Будем хранить несколько полей: root - корень дерева (тип TNode*); nil - нулевой узел, нужен для унифицирования листьев дерева (тип TNode*). Теперь опишем методы класса: bool Insert(std::string key, uint64_t value) - производит вставку узла, при этом сам узел создается уже внутри метода; void FixAfterInsert(TNode *node) - произаодит ребалансировку дерева после вставки, если вставка нарушила какое-то свойство; bool Erase(std::string key) - удаляет элемент из дерева по ключу; void FixAfterErase(TNode *node) - производит ребалансировку дерева после удаления; TNode *FindNode(std::string key) - выполняет поиск узла в дереве по ключу; void Save(std::ofstream file) - сохраняет дерево в открытый файл; void Load(std::ifstream file) - загружает дерево из открытого файла; TNode *GetNil() - возвращает нулевой узел данного дерева; TNode *FindMinNode(TNode *root) - ищет минимальный элемент в поддереве; void Transplant(TNode *a, TNode *b) - занимается перемещением поддеревьев, она заменяет одно поддерево, являющееся дочерним по отношению к своему родиетлю, другим поддеревом; void RightRotate(TNode *node) - выполняет правый поворот поддерева; void LeftRotate(TNode *node) - выполняет левый поворот поддерева; void RecursiveDestroy(TNode *node) - рекурсивно удаляет все элементы поддерева. Также опишем конструктор и деструктор дерева - TRBTree() - конструктор по умолчанию, выделяет память под нулевой узел, и присваевает его корню дерева; TRBTree() - рекурсивно удаляет все элементы дерева и нулевой узел.

Также, далее для сохранения дерева в файл, необходимо описать структуру очереди. Будем хранить несколько полей: buffer - хранит в себе элементы очереди (тип T*); size - размер очереди (тип size_t); capacity - вместимость очереди (тип size_t); head - индекс начала очереди (тип size_t). Опишем методы класса: void Expand() - расширяет буфер очереди при его заполнении; void PushBack(T value) - вставляет элементы в конец очереди; T PopFront() - удаляет элемент из начала очереди; size_t Size() - возвращает размер очереди; T operator[](std::size_t idx) - возвращает элемент по индексу; void Clear() - очищает буфер очереди. Также есть конструктор

и деструктор: TQueue() - конструктор по умолчанию, выделяет место под буффер; TQueue() - деструктор.

На каждой непустой строке файла располагается, в случае операции вставки в дерево команда для вставки и пара ключ-значение, которые необходимо вставить. В случае удаления из дерева, на строке будет располагаться команда и ключ, по которому необходимо удалить соответствующий узел. В случае поиска, на строке будет располагаться только значение, по которому необходимо найти соответствующий узел в дереве. В случае сохранения или загрузки дерева в файл, на строке будет располагаться маркер команды для работы с файлом, сама команда и путь файла.

В случае вставки, мы принимаем пользовательское значение, далее переводим ключ в нижний регистр и выполняем метод Insert. В методе Insert мы ищем место для нового узла и выполняем проверку на существование данного узла в дереве. Если такого узла в дереве нет, то мы выделяем место под новый узел и ставим необходимые значения для полей этого узла. Также мы присваеваем новое значение для правого или левого сына в родительском узле, а если данный элемент единственный в дереве, то он становится корнем дерева. **Примечание**: мы вставляем сразу крансый узел, так как для него в меньшем количестве случаев необходимо ребалансировать дерево. В случае вставки красного узла мы можем нарушить только два правила: чернота корня чернота детей красного узла. Теперь выполняем ребалансировку дерева. Всего у нас может быть 4 случая для ребалансировки. Сначала смотрим какого цвета у нас родитель, если черного цвета, то мы никак не могли нарушить правило черноты детей красного узла, поэтому не запускаем цикл ребалансировок, а просто на всякий случай делаем корень дерева черным и заканчиваем ребалансировку. Если же у родителя красный цвет, то запускаем цикл. Тут у нас может быть два случая, абсолютно зеркальных друг к другу. Первый случай - если родитель является левым сыном деда, второй - родитель является правым сыном деда. После данной проверки мы можем определить дядю. Если дядя красного цвета, то просто перекрашиваем родителя, деда и дядю и запускаем цикл уже от деда и продолжаем ребалансировку. Если же дядя черного цвета, то смотрим каким сыном является новый узел. Если дед, родитель и новый узел образуют треугольник, то выполняем поворот относительно родителя в сторону, противоположную стороне нового узла относительно родителя. Таким образом мы придем к противному случаю, когда дед, родитель и новый узел образуют прямую линию. В этом случае мы перекрашиваем родителя и деда и выполняем поворот относительно родителя в сторону, противоположную стороне нового узла относительно родителя. При выходе из цикла мы обязательно перекрашиваем корень в черный цвет, т.к. в процессе цикла цвет корня мог поменяться.

В случае удаления, мы принимаем пользовательский ключ, переводим его в нижний регистр и выполняем метод Erase. В методе Erase мы ищем в дереве узел с указанным ключом. Далее рассмотрим три случая. В случае когда у нас левый сын - это нулевой узел, мы выполняем перемещение поддеревьев удаляемого узла и его правого сына,

а затем физически удаляем наш узел. В случае когда у нас правый сын - нулевой узел, мы выполняем те же самые действия, отзеркалив все стороны. Остался третий случай, когда у нашего узла нет нулевых детей. Тогда мы ищем минимальный узел в правом поддереве, запоминаем цвет этого узла и запоминаем его правого сына. Если родитель минимального узла это удаляемый узел, то присваеваем значение родительского узла правому сыну минимального узла. Иначе перемещаем поддеревья минимального узла и его правого сына. Далее меняем необходимые ссылки. Потом мы преемещаем местами поддеревья удаляемого узла и минимального узла, переназначаем все ссылки и удаляем наш узел. Такими манипуляциями мы вытащили удаляемый узел из дерева. Теперь смотрим на исходный цвет минимального узла, если он черный, то выполняем ребалансировку. В ребалансировке мы смотрим на цвет брата и цвет его детей. С помощью поворотов и переназначений ссылок ребалансируем дерево.

В случае поиска мы просто выполняем обычный алгоритм поиска узла для двоичного дерева поиска.

В случае сохранения в файл, нам необходимо выполнить обход двоичного дерева в ширину и записывать встречающиеся ненулевые узлы в файл. Чтобы загрузить дерево из файла, просто считываем поочередно каждую строку в файле и вставляем элементы. По итогу получим то же самое дерево, что и до сохранения.

```
1  #include <iostream>
2  #include <string>
3  #include <cstdint>
4  #include <fstream>
5
6  const size_t DEFAULT = 1;
7
8  enum TColour
9  {
10     red,
11     black,
12 };
13
14 template <class T>
15 class TQueue
16 {
17 public:
18     T *buffer;
19     size_t size;
20     size_t capacity;
21     size_t head;
22     TQueue()
23     {
24         buffer = new T[DEFAULT];
25         size = 0;
26         capacity = 1;
```

```
27          head = 0;
28      }
29      void Expand()
30      {
31          T *new_buffer = new T[capacity * 2];
32          capacity *= 2;
33          for (std::size_t i = 0; i < size; ++i)
34          {
35              new_buffer[i] = buffer[i];
36          }
37          delete[] buffer;
38          buffer = new_buffer;
39      }
40      void PushBack(T& value)
41      {
42          if (size == capacity)
43          {
44              Expand();
45          }
46          buffer[size] = value;
47          ++size;
48      }
49      T PopFront()
50      {
51          T res = buffer[head];
52          size--;
53          head++;
54          return res;
55      }
56      size_t Size()
57      {
58          return size;
59      }
60      T& operator[](std::size_t idx)
61      {
62          return buffer[idx + head];
63      }
64      void Clear()
65      {
66          delete[] buffer;
67          size = 0;
68          head = 0;
69          capacity = 1;
70          buffer = new T[capacity];
71      }
72      ~TQueue()
73      {
74          if (size > 0)
75          {
```

```
            delete[] buffer;
        }
    }
};

class TNode
{
public:
    std::string key;
    uint64_t value;
    TColour colour = TColour::black;
    TNode *left = nullptr;
    TNode *right = nullptr;
    TNode *parent = nullptr;
    TNode() = default;
    TNode(std::string key, uint64_t value);
};

class TRBTree
{
public:
    TNode *root = nullptr;
    TNode *nil;
    TRBTree();
    bool Insert(std::string key, uint64_t value);
    bool Erase(std::string key);
    void Save(std::ofstream& file);
    void Load(std::ifstream& file);
    TNode *FindNode(std::string key);
    TNode *GetNil();
    ~TRBTree();
private:
    TNode *FindMinNode(TNode *root);
    void Transplant(TNode *a, TNode *b);
    void FixAfterInsert(TNode *node);
    void FixAfterErase(TNode *node);
    void RightRotate(TNode *node);
    void LeftRotate(TNode *node);
    void RecursiveDestroy(TNode *node);
};


TNode::TNode(std::string key, uint64_t value)
{
    this->key = key;
    this->value = value;
    this->colour = TColour::red;
}

```

```
125 TRBTree::TRBTree()
126 {
127     nil = new TNode();
128     root = nil;
129 }
130
131 TRBTree::~TRBTree()
132 {
133     RecursiveDestroy(root);
134     delete nil;
135 }
136
137 void TRBTree::RecursiveDestroy(TNode *node)
138 {
139     if (node != nil)
140     {
141         RecursiveDestroy(node->left);
142         RecursiveDestroy(node->right);
143         delete node;
144     }
145 }
146
147 TNode *TRBTree::GetNil()
148 {
149     return nil;
150 }
151
152 TNode *TRBTree::FindMinNode(TNode *root)
153 {
154     TNode *currentNode = root;
155     while (currentNode->left != nil)
156     {
157         currentNode = currentNode->left;
158     }
159     return currentNode;
160 }
161
162 TNode *TRBTree::FindNode(std::string key)
163 {
164     TNode *currentNode = this->root;
165     while (currentNode->key != key)
166     {
167         if (currentNode == nil)
168         {
169             break;
170         }
171         currentNode = (key < currentNode->key) ? currentNode->left : currentNode->right
              ;
172     }
```

```
173     return currentNode;
174 }
175 
176 void TRBTree::LeftRotate(TNode *node)
177 {
178     TNode *tmp = node->right; // set tmp
179     node->right = tmp->left; // turn tmp's left subtree into node's right subtree
180     if (tmp->left != nil)
181     {
182         tmp->left->parent = node;
183     }
184     tmp->parent = node->parent; // link node's parent to tmp
185     if (node->parent == nil)
186     {
187         this->root = tmp;
188     }
189     else if (node == node->parent->left)
190     {
191         node->parent->left = tmp;
192     }
193     else
194     {
195         node->parent->right = tmp;
196     }
197     tmp->left = node; // put node on tmp's left
198     node->parent = tmp;
199 }
200 
201 void TRBTree::RightRotate(TNode *node)
202 {
203     TNode *tmp = node->left; // set tmp
204     node->left = tmp->right; // turn tmp's right subtree into node's left subtree
205     if (tmp->right != nil)
206     {
207         tmp->right->parent = node;
208     }
209     tmp->parent = node->parent; // link node's parent to tmp
210     if (node->parent == nil)
211     {
212         this->root = tmp;
213     }
214     else if (node == node->parent->left)
215     {
216         node->parent->left = tmp;
217     }
218     else
219     {
220         node->parent->right = tmp;
221     }
```

```
222     tmp->right = node; // put node on tmp's right
223     node->parent = tmp;
224 }
225 
226 void TRBTree::FixAfterInsert(TNode *node)
227 {
228     TNode *uncle;
229     while (node->parent->colour == TColour::red)
230     {
231         if (node->parent == node->parent->parent->left)
232         {
233             uncle = node->parent->parent->right;
234             if (uncle->colour == TColour::red)
235             {
236                 node->parent->colour = TColour::black; // Case 1
237                 uncle->colour = TColour::black; // Case 1
238                 node->parent->parent->colour = TColour::red; // Case 1
239                 node = node->parent->parent; // Case 1
240             }
241             else
242             {
243                 if (node == node->parent->right)
244                 {
245                     node = node->parent; // Case 2
246                     this->LeftRotate(node); // Case 2
247                 }
248                 node->parent->colour = TColour::black; // Case 3
249                 node->parent->parent->colour = TColour::red; // Case 3
250                 this->RightRotate(node->parent->parent); // Case 3
251             }
252         }
253         else
254         {
255             uncle = node->parent->parent->left;
256             if (uncle->colour == TColour::red)
257             {
258                 node->parent->colour = TColour::black; // Case 1
259                 uncle->colour = TColour::black; // Case 1
260                 node->parent->parent->colour = TColour::red; // Case 1
261                 node = node->parent->parent; // Case 1
262             }
263             else
264             {
265                 if (node == node->parent->left)
266                 {
267                     node = node->parent; // Case 2
268                     this->RightRotate(node); // Case 2
269                 }
270                 node->parent->colour = TColour::black; // Case 3
```

```
271                 node->parent->parent->colour = TColour::red; // Case 3
272                 this->LeftRotate(node->parent->parent); // Case 3
273             }
274         }
275     }
276     this->root->colour = TColour::black; // Case 0
277 }
278 
279 bool TRBTree::Insert(std::string key, uint64_t value)
280 {
281     TNode *currentNode = this->root;
282     TNode *parentNode = nil;
283     while (currentNode != nil)
284     {
285         if (currentNode->key == key)
286         {
287             return false;
288         }
289         parentNode = currentNode;
290         currentNode = (key < currentNode->key) ? currentNode->left : currentNode->right
                ;
291     }
292     TNode *newNode = new TNode(key, value);
293     newNode->parent = parentNode;
294     newNode->left = nil;
295     newNode->right = nil;
296     if (parentNode == nil)
297     {
298         this->root = newNode;
299     }
300     else if (key < parentNode->key)
301     {
302         parentNode->left = newNode;
303     }
304     else
305     {
306         parentNode->right = newNode;
307     }
308     this->FixAfterInsert(newNode);
309     return true;
310 }
311 
312 void TRBTree::Transplant(TNode *a, TNode *b)
313 {
314     if (a->parent == nil)
315     {
316         this->root = b;
317     }
318     else if (a == a->parent->left)
```

```
319     {
320         a->parent->left = b;
321     }
322     else
323     {
324         a->parent->right = b;
325     }
326     b->parent = a->parent;
327 }
328 
329 void TRBTree::FixAfterErase(TNode *node)
330 {
331     while ((node != this->root) && (node->colour == TColour::black))
332     {
333         if (node == node->parent->left)
334         {
335             TNode *sibling = node->parent->right;
336             if (sibling->colour == TColour::red) // Case 1
337             {
338                 sibling->colour = TColour::black; // Case 1
339                 node->parent->colour = TColour::red; // Case 1
340                 this->LeftRotate(node->parent); // Case 1
341                 sibling = node->parent->right; // Case 1
342             }
343             if ((sibling->left->colour == TColour::black) && (sibling->right->colour ==
                     TColour::black)) // Case 2
344             {
345                 sibling->colour = TColour::red; // Case 2
346                 node = node->parent; // Case 2
347             }
348             else
349             {
350                 if (sibling->right->colour == TColour::black) // Case 3
351                 {
352                     sibling->left->colour = TColour::black; // Case 3
353                     sibling->colour = TColour::red; // Case 3
354                     this->RightRotate(sibling); // Case 3
355                     sibling = node->parent->right; // Case 3
356                 }
357                 sibling->colour = node->parent->colour; // Case 4
358                 node->parent->colour = TColour::black; // Case 4
359                 sibling->right->colour = TColour::black; // Case 4
360                 this->LeftRotate(node->parent); // Case 4
361                 node = this->root; // Case 4
362             }
363         }
364         else
365         {
366             TNode *sibling = node->parent->left;
```

```
367             if (sibling->colour == TColour::red) // Case 1
368             {
369                 sibling->colour = TColour::black; // Case 1
370                 node->parent->colour = TColour::red; // Case 1
371                 this->RightRotate(node->parent); // Case 1
372                 sibling = node->parent->left; // Case 1
373             }
374             if ((sibling->left->colour == TColour::black) && (sibling->right->colour ==
                     TColour::black)) // Case 2
375             {
376                 sibling->colour = TColour::red; // Case 2
377                 node = node->parent; // Case 2
378             }
379             else
380             {
381                 if (sibling->left->colour == TColour::black) // Case 3
382                 {
383                     sibling->right->colour = TColour::black; // Case 3
384                     sibling->colour = TColour::red; // Case 3
385                     this->LeftRotate(sibling); // Case 3
386                     sibling = node->parent->left; // Case 3
387                 }
388                 sibling->colour = node->parent->colour; // Case 4
389                 node->parent->colour = TColour::black; // Case 4
390                 sibling->left->colour = TColour::black; // Case 4
391                 this->RightRotate(node->parent); // Case 4
392                 node = this->root; // Case 4
393             }
394         }
395     }
396     node->colour = TColour::black;
397 }
398
399 bool TRBTree::Erase(std::string key)
400 {
401     TNode *deleteNode = this->FindNode(key);
402     TNode *tmp;
403     TColour originalColour = deleteNode->colour;
404     if (deleteNode == nil)
405     {
406         return false;
407     }
408     if (deleteNode->left == nil)
409     {
410         tmp = deleteNode->right; // Case 1
411         this->Transplant(deleteNode, deleteNode->right); // Case 1
412         delete deleteNode; // Case 1
413     }
414     else if (deleteNode->right == nil)
```

```
415     {
416         tmp = deleteNode->left; // Case 2
417         this->Transplant(deleteNode, deleteNode->left); // Case 2
418         delete deleteNode; // Case 2
419     }
420     else
421     {
422         TNode *rightMinimum = this->FindMinNode(deleteNode->right); // Case 3
423         originalColour = rightMinimum->colour; // Case 3
424         tmp = rightMinimum->right; // Case 3
425         if (rightMinimum->parent == deleteNode) // Case 3
426         {
427             tmp->parent = rightMinimum; // Case 3
428         }
429         else
430         {
431             this->Transplant(rightMinimum, rightMinimum->right); // Case 3
432             rightMinimum->right = deleteNode->right; // Case 3
433             rightMinimum->right->parent = rightMinimum; // Case 3
434         }
435         this->Transplant(deleteNode, rightMinimum); // Case 3
436         rightMinimum->left = deleteNode->left; // Case 3
437         rightMinimum->left->parent = rightMinimum; // Case 3
438         rightMinimum->colour = deleteNode->colour; // Case 3
439         delete deleteNode; // Case 3
440     }
441     if (originalColour == TColour::black)
442     {
443         this->FixAfterErase(tmp);
444     }
445     return true;
446 }
447 
448 void TRBTree::Save(std::ofstream& file)
449 {
450     if (root != nil)
451     {
452         TQueue<TNode *> level;
453         TQueue<TNode *> next_level;
454         level.PushBack(root);
455         while (level.Size() != 0)
456         {
457             next_level.Clear();
458             for (size_t i = 0; i < level.Size(); ++i)
459             {
460                 file << level[i]->key << " " << level[i]->value << "\n";
461                 if (level[i]->left != nil)
462                 {
463                     next_level.PushBack(level[i]->left);
```

```
464                 }
465                 if (level[i]->right != nil)
466                 {
467                     next_level.PushBack(level[i]->right);
468                 }
469             }
470             level.Clear();
471             for (size_t i = 0; i < next_level.Size(); ++i)
472             {
473                 level.PushBack(next_level[i]);
474             }
475         }
476     }
477 }
478 
479 void TRBTree::Load(std::ifstream& file)
480 {
481     this->RecursiveDestroy(this->root);
482     this->root = nil;
483     std::string fkey;
484     uint64_t fvalue;
485     while (file >> fkey >> fvalue)
486     {
487         this->Insert(fkey, fvalue);
488     }
489 }
490 
491 std::string mtolower(std::string str)
492 {
493     for (size_t i = 0; i < str.size(); ++i)
494     {
495         str[i] = tolower(str[i]);
496     }
497     return str;
498 }
499 
500 int main()
501 {
502     std::string input;
503     TRBTree tree;
504     while (std::cin >> input)
505     {
506         if (input == "+")
507         {
508             uint64_t value;
509             std::string key;
510             std::cin >> key >> value;
511             if (tree.Insert(mtolower(key), value))
512             {
```

```
513                 std::cout << "OK\n";
514             }
515             else
516             {
517                 std::cout << "Exist\n";
518             }
519         }
520         else if (input == "-")
521         {
522             std::string key;
523             std::cin >> key;
524             if (tree.Erase(mtolower(key)))
525             {
526                 std::cout << "OK\n";
527             }
528             else
529             {
530                 std::cout << "NoSuchWord\n";
531             }
532         }
533         else if (input == "!")
534         {
535             std::string action;
536             std::string path;
537             std::cin >> action >> path;
538             if (action == "Save")
539             {
540                 std::ofstream file;
541                 try
542                 {
543                     file.open(path);
544                 }
545                 catch (std::exception& ex)
546                 {
547                     std::cout << "ERROR: " << ex.what() << "\n";
548                 }
549                 tree.Save(file);
550             }
551             else if (action == "Load")
552             {
553                 std::ifstream file;
554                 try
555                 {
556                     file.open(path);
557                 }
558                 catch (std::exception& ex)
559                 {
560                     std::cout << "ERROR: " << ex.what() << "\n";
561                 }
```

```
562                 tree.Load(file);
563             }
564             std::cout << "OK\n";
565         }
566         else
567         {
568             TNode *FoundNode = tree.FindNode(mtolower(input));
569             if (FoundNode == tree.GetNil())
570             {
571                 std::cout << "NoSuchWord\n";
572             }
573             else
574             {
575                 std::cout << "OK: " << FoundNode->value << "\n";
576             }
577         }
578     }
579 }
```

main.cpp	
std::string mtolower(std::string str)	Функция перевода строки в нижний регистр
int main()	Входная точка программы

```
1  enum TColour
2  {
3      red,
4      black,
5  };
6  template <class T>
7  class TQueue
8  {
9  public:
10     T *buffer;
11     size_t size;
12     size_t capacity;
13     size_t head;
14     TQueue();
15     void Expand();
16     void PushBack(T& value);
17     T PopFront();
18     T& operator[](std::size_t idx);
19     void Clear();
20     ~TQueue();
21 };
22 class TNode
23 {
24 public:
25     std::string key;
```

```
26      uint64_t value;
27      TColour colour = TColour::black;
28      TNode *left = nullptr;
29      TNode *right = nullptr;
30      TNode *parent = nullptr;
31      TNode() = default;
32      TNode(std::string key, uint64_t value);
33  };
34  class TRBTree
35  {
36  public:
37      TNode *root = nullptr;
38      TNode *nil;
39      TRBTree();
40      bool Insert(std::string key, uint64_t value);
41      bool Erase(std::string key);
42      void Save(std::ofstream& file);
43      void Load(std::ifstream& file);
44      TNode *FindNode(std::string key);
45      TNode *GetNil();
46      ~TRBTree();
47  private:
48      TNode *FindMinNode(TNode *root);
49      TNode *FindMaxNode(TNode *root);
50      void Transplant(TNode *a, TNode *b);
51      void FixAfterInsert(TNode *node);
52      void FixAfterErase(TNode *node);
53      void RightRotate(TNode *node);
54      void LeftRotate(TNode *node);
55      void RecursiveDestroy(TNode *node);
56  };
```

3 Консоль

```
lexasy@MSI:$ g++ main.cpp
lexasy@MSI:$ cat test
+ a 1
+ A 2
+ aaa 18446744073709551615
aaa
A
-A
a
lexasy@MSI:$ ./a.out <test
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: вставка или поиск 40000 рандомно сгенерированных пар ключ-значение с помощью самостоятельно реализованной структуры красно-черного дерева и стандартным шаблоном структуры std::map, которая изнутри представляет собой красно-черное дерево.

```
lexasy@MSI:$ g++ main.cpp
lexasy@MSI:$ ./a.out <tests/01.t >tmp
lexasy@MSI:~/Desktop/Prog/DA_labs/lab2$ cat tmp | grep "time"
time: 108000ms
lexasy@MSI:$ g++ benchmark.cpp
lexasy@MSI:$ ./a.out <tests/01.t >tmp
lexasy@MSI:$ cat tmp | grep "time"
time: 103074ms
```

Как видно, самодельная структура дерева не сильно уступает стандартному шаблону структуры. Так как обе структуры представляют собой красно-черное дерево, то они и доджны работать примерно одинаковое время. Отставание моей структуры может быть связано с моим недостатком опыта программирования на C++, и поэтому я могу не знать каких-то вещей, которые могли бы ускорить мою структуру.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать красно-черное дерево, а также его балансировать после удаления или вставки. Также я узнал, что сбалансированное дерево это не только AVL-дерево. Есть множество сбалансированных деревьев. Сбалансированные деревья также бывают приближенно сбалансированными или же идеально сбалансированны. Деревья полезная структура данных, особенно если учесть, что вся файлова система в Unix подобных системах основана на дереве. Искусство работы с деревом улучшает навык программирования и навык слежки за памятью, который пригождается чуть ли ни в каждой программе на C++.

Список литературы

[1] Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн "Алгоритмы построение и анализ"3-е издание (2013)(дата обращения: 01.05.2024)