Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: А. А. Каримов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

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.

Вариант используемой структуры: `Красно-чёрное дерево`

1 Описание

Требуется реализовать словарь с помощью красно-черного дерева. Красно-черное дерево представляет собой бинарное дерево поиска, в котором вершины покрашены в два вида цвета: красный и черный. Также требуется выполнение следующих условий:
1) Корень всегда черный
2) Красный узел не имеет красных детей
3) Количество черных вершин на пути от корня до любого листа одинаково

Выполнение этих требований гарантирует, что дерево будет сбалансированным, то есть длина пути от корня до любого листа отличается не более, чем на единицу.

2 Исходный код

Первоначально требуется реализовать структуру, с помощью которой будет работать ассоциативный массив. В варианте задания дано красно-черное дерево. Его код будет следующим:

```
template <class TPair>
class TRBTree {
    private:
        class TNode {
            private:
                const int SPACES_COUNT = 2;

            public:
                typedef TPair::second_type value_type;
                TPair item;
                colors color = BLACK;
                std::shared_ptr<TNode> child[2];

                TNode(colors _color, \
                    TPair _item, \
                    std::shared_ptr<TNode> _left, \
                    std::shared_ptr<TNode> _right) :
                        color(_color),
                        item(_item) {
                            child[LEFT] = _left;
                            child[RIGHT] = _right;
                        }

                ~TNode() = default;

                bool value_exists(TPair _item) {
                    return (item.first == _item.first) or
                        (child[LEFT] != nullptr and child[LEFT]->item.first == _item.
                            first) or
                        (child[RIGHT] != nullptr and child[RIGHT]->item.first == _item.
                            first);
                }

                void print(int space = 0) {
                    space += 2;

                    if(child[RIGHT])
                        child[RIGHT]->print(space);
                    for(int i = SPACES_COUNT; i < space; ++i) {
                        std::cout << ' ' ;
                    }
                    if(color == RED)
                        std::cout << red << item.first << reset << std::endl;
```

```
42                  else
43                      std::cout << item.first << std::endl;
44
45                  if(child[LEFT])
46                      child[LEFT]->print(space);
47              }
48
49              bool has_no_children() {
50                  return (child[LEFT] == nullptr) and (child[RIGHT] == nullptr);
51              }
52
53              bool has_one_child() {
54                  return (child[LEFT] != nullptr) ^ (child[RIGHT] != nullptr);
55              }
56
57              bool has_two_children() {
58                  return (child[LEFT] != nullptr) and (child[RIGHT] != nullptr);
59              }
60
61              void write_data_to_file(std::ostream& os) const {
62                  os.write(BNODE.c_str(), BNODE.size()); // 5 bytes
63
64                  // first value(Key)
65                  size_t sizeOfKey = item.first.size();
66                  os.write(reinterpret_cast<const char*>(&sizeOfKey), sizeof(size_t));
67                  os.write(item.first.c_str(), sizeOfKey);
68
69                  // second value(Value)
70                  uint64_t second_value = item.second;
71                  os.write(reinterpret_cast<const char*>((&second_value)), sizeof(
                        second_value));
72
73                  // node's color
74                  if(color == BLACK)
75                      os.put('1');
76                  else
77                      os.put('0');
78                  if(child[LEFT]) child[LEFT]->write_data_to_file(os);
79                  else os.write(ENODE.c_str(), ENODE.size()); // 5 bytes
80                  if(child[RIGHT]) child[RIGHT]->write_data_to_file(os);
81                  else os.write(ENODE.c_str(), ENODE.size()); // 5 bytes
82              }
83
84          };
85
86          public:
87
88              class TIterator {
89                  private:
```

```
90                  std::shared_ptr<TNode> current = nullptr;
91              public:
92                  TIterator() = default;
93                  TIterator(std::shared_ptr<TNode> _current) : current(_current) {}
94
95                  bool operator==(TIterator& other) {
96                      return current == other.current;
97                  }
98
99                  bool operator!=(TIterator& other) {
100                     return current != other.current;
101                 }
102
103                 TNode::value_type operator*() {
104                     return current->item.second;
105                 }
106         };
107
108         TIterator end() const {
109             return TIterator();
110         }
111
112         TIterator begin() const {
113             return TIterator(root);
114         }
115
116     private:
117
118     bool is_red(std::shared_ptr<TNode> TNode) const {
119         return (TNode != nullptr and TNode->color == RED);
120     }
121
122     bool is_black(std::shared_ptr<TNode> TNode) const {
123         return (TNode == nullptr or TNode->color == BLACK);
124     }
125
126     std::shared_ptr<TNode> rotate(std::shared_ptr<TNode> currentRoot, bool
        direction);
127
128     std::shared_ptr<TNode> doubleRotate(std::shared_ptr<TNode> currentRoot, bool
        direction);
129
130     // sets opposite color to root and its children
131     void change_colors(std::shared_ptr<TNode> TNode);
132
133     std::shared_ptr<TNode> check_add_correctness(std::shared_ptr<TNode> localRoot,
        bool direction);
134
```

```
135         std::shared_ptr<TNode> _insert(std::shared_ptr<TNode> currentTNode, const TPair
                & item, bool& is_found);
136
137         std::shared_ptr<TNode> get_minimum(std::shared_ptr<TNode> currentRoot);
138
139         std::shared_ptr<TNode> check_erase_correctness(std::shared_ptr<TNode>
                currentTNode, bool direction, bool& needBalance);
140     private:
141
142         std::shared_ptr<TNode> root = nullptr;
143
144     private:
145
146         std::shared_ptr<TNode> _erase(std::shared_ptr<TNode> currentTNode, TPair::
                first_type key, bool& needBalance, bool& erased);
147
148         TPair::second_type& get_lvalue(std::shared_ptr<TNode> currentNode, TPair::
                first_type key);
149
150         TIterator _search(const TPair::first_type& key, std::shared_ptr<TNode>
                currentTNode);
151
152     public:
153
154         TRBTree() = default;
155
156         bool insert(const TPair& item);
157
158         bool erase(const TPair::first_type& key);
159
160         TIterator search(const TPair::first_type& key) {
161             return _search(key, root);
162         }
163
164         void save_to_file(std::ofstream& os) {
165             if(root) root->write_data_to_file(os);
166         }
167
168         std::shared_ptr<TNode> load_from_file(std::shared_ptr<TNode> _root, std::
                ifstream& is);
169
170         TIterator operator[](TPair::first_type _key) {
171             return get_lvalue(root, _key);
172         }
173
174         void load_from_file(std::ifstream& is) {
175             root = load_from_file(root, is);
176         }
177
```

```
178         void clear() {
179             root = nullptr;
180         }
181 
182         friend void print<TPair>(TRBTree<TPair>* tree);
183 
184 };
```

В файле **map.hpp** находится реализация словаря.

```
1  template<class T1, class T2>
2  class TPair {
3      public:
4 
5          typedef T1 first_type;
6          typedef T2 second_type;
7          T1 first;
8          T2 second;
9          TPair() = default;
10         TPair(const T1& fst, const T2& scd) : first(fst), second(scd) {}
11 
12         TPair(T1 _first) : first(_first) {
13             second = T2();
14         }
15 };
16 
17 template <typename key, typename data>
18 class TMap {
19     public:
20         typedef key key_type;
21         typedef data mapped_type;
22     private:
23 
24         TRBTree<TPair<key_type, mapped_type>> rbtree;
25 
26     public:
27 
28         class TMapIterator {
29             private:
30 
31                 typename TRBTree<TPair<key_type, mapped_type>>::TIterator current;
32 
33             public:
34 
35                 TMapIterator(typename TRBTree<TPair<key_type, mapped_type>>::TIterator
                        other) : current(other) {}
36 
37                 bool operator==(TMapIterator& other) {
38                     return current == other.current;
```

```
39                }
40
41                bool operator!=(TMapIterator& other) {
42                    return current != other.current;
43                }
44
45                const mapped_type operator*() {
46                    return *current;
47                }
48        };
49
50        TMapIterator begin() const {
51            return TMapIterator(rbtree.begin());
52        }
53
54        TMapIterator end() const {
55            return TMapIterator(rbtree.end());
56        }
57
58        bool insert(TPair<key_type, mapped_type> item) {
59            return rbtree.insert(item);
60        }
61
62        bool erase(const key_type& _key) {
63            return rbtree.erase(_key);
64        }
65
66        TMapIterator operator[](const key_type& _key) {
67            typename TRBTree<TPair<key_type, mapped_type>>::TIterator value = rbtree.
                search(_key);
68            if(value == rbtree.end()) {
69                rbtree.insert(TPair<key_type, mapped_type>(_key));
70            }
71            return *value;
72        }
73
74        TMapIterator find(const key_type& _key) {
75            return TMapIterator(rbtree.search(_key));
76        }
77
78        void save_to_file(std::ofstream& os) {
79            rbtree.save_to_file(os);
80        }
81
82        void load_from_file(std::ifstream& is) {
83            rbtree.clear();
84            rbtree.load_from_file(is);
85        }
86 };
```

3 Консоль

```
karseny99@karseny99:/study/DA/lab2/$ ./lab2 <1.in
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

4 Тест производительности

Производительность оценивается так: на один и тех же тестовых данных запускается мой словарь и std::map из библиотеки C++. Тесты на 10^4, 10^5 и 10^6 элементов.

```
karseny99@karseny99:/mnt/study/DA/lab1$ ./benchmark <tests/05.t
Count of lines is 10000
STL map time: 3582us
My map time: 6305us
karseny99@karseny99:/mnt/study/DA/lab1$ ./benchmark <tests/06.t
Count of lines is 100000
STL map time: 67123us
My map time: 89103
karseny99@karseny99:/mnt/study/DA/lab1$ ./benchmark <tests/07.t
Count of lines is 1000000
STL map time: 88889us
My map time: 126985us
```

Как видно,реализация STL выигрывает,но эта разница довольна маленькая,и я связываю это с тем,что в моей реализации проверка корректности происходит при каждом рекурсивном возврате,что делает общую сложность такой же,но с большей константой.

5 Выводы

В ходе выполнения лабораторной работы я смог разобраться в работе std::map из стандартной библиотеки c++ и в целом понял, как работают ассоциативные массивы языков. Для этого я разработал вспомогательную структуру данных красно-черное дерево, которая была не самой простой в понимании и реализации.

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*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево`