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. В. Семин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2022

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 enquoteERROR: и описывающую на английском языке возникшую ошибку.

AVL-дерево.

1 Описание

В интернет-ресурсе [4] можно найти следующее описание AVL-деревьев:

АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на 1. Эта структура данных разработана советскими учеными Адельсон-Вельским Георгием Максимовичем и Ландисом Евгением Михайловичем в 1962 году. Аббревиатура АВЛ соответствует первым буквам фамилий этих ученых. Первоначально АВЛ-деревья были придуманы для организации перебора в шахматных программах. Советская шахматная программа «Каисса» стала первым официальным чемпионом мира в 1974 году.

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранится показатель баланса – разность высот правого и левого поддеревьев. В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо. В АВЛ-дереве показатель баланса *balance* для каждого узла, включая корень, по модулю не превосходит 1.

Стоит добавить, что критерий баланса положительно сказывается на общей производительности. Но в процессе перестроения дерева, добавления и удалнеия узлов баланс может нарушиться, влседствие чего потребуется перебаансировка некоторой вершины. Очевидно, над деревом определены операции вставки, удаления, поиска.

Доказано, что высота АВЛ-дерева, имеющего N узлов, равна *log(2*N)*. Учитывая этот факт, а также факт того, что время выполнения операций удаления и добавления узла зависят напрямую лишь от операции поиска необходимой вершины, получаем, что временная сложность трех операций для худшего и среднего случая - *O(logN)*.

2 Исходный код

Программа разделена на три файла:

1. *main.cpp*
2. *avl.hpp*
3. *user_avl.hpp*

В файле *main.cpp* описан интерфейс программы, а вся основная логика же описана в двух других файлах.

```
1  #include <iostream>
2  #include <cstring>
3  #include "avl.hpp"
4  #include "user_avl.hpp"
5
6  using ull = unsigned long long;
7
8  int main() {
9    std::ios::sync_with_stdio(false);
10   std::string comand;
11   TUserAvl tr;
12   while (std::cin >> comand) {
13     if (comand[0] == '+') {
14       // std::cout << "main: if: in +\n";
15       tr.UserInsert();
16     }
17     else if (comand[0] == '-') {
18       tr.UserRemove();
19     }
20     else if (comand[0] == '!' && comand.size() == 1) {
21       tr.SaveLoad();
22     }
23     else {
24       tr.UserFind(std::move(comand));
25     }
26   }
27   return 0;
28 }
```

Файл *avl.hpp* содержит в себе класс *TAvl*, который состоит из определения структуры узла дерева *TAvlNode* и реализации необходимых методов для работы АВЛ-дерева. Среди таких методов четыре вращения дерева, балансировка, вставка (*Insert*), удаление(*RemoveNode*) и поиск(*SearchNode*) узла.

Стоит выделить балансировку - это такая операция относительно вершины, которая в случае разницы высот правого и левого поддеревьев на 2, изменяет связи между предками и наследниками в поддереве данной вершины таким образом, что разницы становится меньше, либо равной по модулю 1. Процесс балансировки происходит за счет вращений дерева относительно некоторой вершины.

Вращения делятся на четыре следующих вида:

1. Малое левое вращение AVL LR. Данное вращение используется тогда, когда высота b-поддерева — высота L = 2 и высота C <= высота R.
2. Большое левое вращение AVL BR. Данное вращение используется тогда, когда высота b-поддерева — высота L = 2 и высота c-поддерева > высота R.
3. Малое правое вращение AVL LL. Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота C <= высота L.
4. Большое правое вращение AVL BL. Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота c-поддерева > высота L.

```
1  using ull = unsigned long long;
2
3  template<typename typeK, typename typeV>
4  class TAvl {
5  protected:
6
7      struct TAvlNode {
8          typeK key;
9          typeV val;
10         ull height;
11         TAvlNode* left;
12         TAvlNode* right;
13
14         TAvlNode(): key(), val(), height(1), left(nullptr), right(nullptr) {};
15         TAvlNode(typeK k, typeV v): key(k), val(v), height(1), left(nullptr), right(
               nullptr) {};
16
17     };
18     TAvlNode* root;
19      ull GetHeight(const TAvlNode* nd) {
20         return nd == nullptr ? 0 : nd->height;
21     }
22
23     ull GetBalance(const TAvlNode* nd) {
24         return GetHeight(nd->left) - GetHeight(nd->right);
25     }
26
```

```
27      void CalcHeight(TAvlNode* nd) {
28          nd->height = std::max(GetHeight(nd->left), GetHeight(nd->right)) + 1;
29      }
30
31      TAvlNode* Rotate_LL(TAvlNode* nd) {
32          TAvlNode* nd_new = nd->right;
33          nd->right = nd_new->left;
34          nd_new->left = nd;
35          CalcHeight(nd);
36          CalcHeight(nd_new);
37          return nd_new;
38      }
39
40      TAvlNode* Rotate_RR(TAvlNode* nd) {
41          TAvlNode* nd_new = nd->left;
42          nd->left = nd_new->right;
43          nd_new->right = nd;
44          CalcHeight(nd);
45          CalcHeight(nd_new);
46          return nd_new;
47      }
48
49      TAvlNode* Rotate_RL(TAvlNode* nd) {
50          nd->right = Rotate_RR(nd->right);
51          return Rotate_LL(nd);
52      }
53
54      TAvlNode* Rotate_LR(TAvlNode* nd) {
55          nd->left = Rotate_LL(nd->left);
56          return Rotate_RR(nd);
57      }
58
59      TAvlNode* DoBalance(TAvlNode* nd) {
60          if (nd == nullptr) {
61              return nullptr;
62          }
63          CalcHeight(nd);
64          int nd_balance = GetBalance(nd);
65          if (nd_balance == -2) {
66              if (GetBalance(nd->right) == 1) {
67                  return Rotate_RL(nd);
68              }
69              return Rotate_LL(nd);
70          } else if (nd_balance == 2) {
71              if (GetBalance(nd->left) == -1) {
72                  return Rotate_LR(nd);
73              }
74              return Rotate_RR(nd);
75          }
```

```
76          return nd;
77      }
78
79      TAvlNode* Insert(TAvlNode* nd, typeK k, typeV v) {
80          if (nd == nullptr) {
81              try {
82                  nd = new TAvlNode(std::move(k), v);
83              } catch (std::bad_alloc &err) {
84                  std::cout << "ERROR: " << err.what() << "\n";
85                  return nullptr;
86              }
87              std::cout << "OK\n";
88              return nd;
89          }
90          if (k < nd->key) {
91              nd->left = Insert(nd->left, k, v);
92          } else if (k > nd->key) {
93              nd->right = Insert(nd->right, k, v);
94          } else {
95              std::cout << "Exist\n";
96          }
97  // std::cout << "end insert\n";
98          return DoBalance(nd);
99      }
100
101   TAvlNode* RemoveMinRight(TAvlNode* nd, TAvlNode* tmp_nd) {
102     if (tmp_nd->left != nullptr) {
103       tmp_nd->left = RemoveMinRight(nd, tmp_nd->left);
104     }
105     else {
106       TAvlNode* right_ch = tmp_nd->right;
107       nd->key = std::move(tmp_nd->key);
108       nd->val = tmp_nd->val;
109       delete tmp_nd;
110       tmp_nd = right_ch;
111     }
112     return DoBalance(tmp_nd);
113   }
114
115     TAvlNode* RemoveNode(TAvlNode* nd, typeK k) {
116         if (nd == nullptr) {
117             std::cout << "NoSuchWord\n";
118             return nullptr;
119         }
120         if (k < nd->key) {
121             nd->left = RemoveNode(nd->left, k);
122         } else if (k > nd->key) {
123             nd->right = RemoveNode(nd->right, k);
124         } else {
```

```
125             TAvlNode* nd_left = nd->left;
126             TAvlNode* nd_right = nd->right;
127             if (nd_left == nullptr && nd_right == nullptr) {
128                 delete nd;
129                 std::cout << "OK\n";
130                 return nullptr;
131             }
132             if (nd_left == nullptr) {
133                 delete nd;
134                 std::cout << "OK\n";
135                 return nd_right;
136             }
137             if (nd_right == nullptr) {
138                 delete nd;
139                 std::cout << "OK\n";
140                 return nd_left;
141             }
142             nd->right = RemoveMinRight(nd, nd_right);
143             std::cout << "OK\n";
144         }
145         return DoBalance(nd);
146     }
147
148     TAvlNode* SearchNode(TAvlNode* nd, typeK k) {
149     if (nd == nullptr) {
150       return nullptr;
151     }
152     if (k < nd->key) {
153             return SearchNode(nd->left, k);
154         } else if (k > nd->key) {
155             return SearchNode(nd->right, k);
156         } else {
157             return nd;
158         }
159     }
160
161     void PrintTree(TAvlNode* nd) {
162         static int count = 0;
163         if (nd != nullptr) {
164             count++;
165             PrintTree(nd->right);
166             count--;
167             for (int i = 0; i < count; i++) {
168                 std::cout << "\t";
169             }
170             std::cout << nd->key << "\n";
171             count++;
172             PrintTree(nd->left);
173             count--;
```

```
174         }
175     }
176
177
178 public:
179     TAvl(): root(nullptr) {};
180
181     void Add(typeK k, typeV v) {
182         root = Insert(root, std::move(k), v);
183     }
184
185     void DeleteNode(typeK k) {
186         root = RemoveNode(root, std::move(k));
187     }
188
189     TAvlNode* Find(typeK k) {
190         return SearchNode(root, std::move(k));
191     }
192
193     void Print() {
194         PrintTree(root);
195     }
196
197     void DeleteTree(TAvlNode* nd) {
198         if (nd != nullptr) {
199       DeleteTree(nd->left);
200       DeleteTree(nd->right);
201       delete nd;
202     }
203     }
204
205     ~TAvl() {
206         DeleteTree(root);
207     }
208 };
```

В файле $user\_avl.hpp$ описан алгоритм сохранения и загрузки из бинарного файла объектов моего класса $TUserAvl$, который являтся наследником ранее опианного класса $TAvl$. Именно с этим классом взаимодействует пользователь.

```
1 #pragma once
2 #include <fstream>
3 #include <string>
4 #include <new>
5 #include <cctype>
6
7 using ull = unsigned long long;
8
9 class TUserAvl: public TAvl<std::string, ull> {
```

```
10
11      void StrToLow(std::string& str) {
12          for (int i = 0; i < str.size(); i++) {
13              str[i] = std::tolower(str[i]);
14          }
15      }
16
17      void Save(std::ostream& os, const TAvlNode* nd) {
18      if (nd == nullptr) {
19        return;
20      }
21      int keySize = nd->key.size();
22      os.write((char *)&keySize, sizeof(keySize));
23      os.write(nd->key.c_str(), keySize);
24      os.write((char *)&nd->val, sizeof(nd->val));
25
26          bool left_exist = nd->left != nullptr;
27      bool right_exist= nd->right != nullptr;
28
29      os.write((char *)&left_exist, sizeof(left_exist));
30      os.write((char *)&right_exist, sizeof(right_exist));
31
32      if (left_exist) {
33        Save(os, nd->left);
34      }
35      if (right_exist) {
36        Save(os, nd->right);
37      }
38    }
39
40    TAvlNode* Load(std::istream &is) {
41      TAvlNode *root = nullptr;
42      int keySize;
43      is.read((char *)&keySize, sizeof(keySize));
44
45      if (is.gcount() == 0) {
46        return root;
47      }
48
49      char *key = new char[keySize + 1];
50      key[keySize] = '\0';
51      is.read(key, keySize);
52
53      ull value;
54      is.read((char *)&value, sizeof(value));
55
56      bool hasLeft = false;
57      bool hasRight = false;
58      is.read((char *)&hasLeft, sizeof(hasLeft));
```

```
59      is.read((char *)&hasRight, sizeof(hasRight));
60
61      root = new TAvlNode();
62      root->key = std::move(key);
63      root->val = value;
64
65      if (hasLeft) {
66        root->left = Load(is);
67      } else {
68        root->left = nullptr;
69      }
70      if (hasRight) {
71        root->right = Load(is);
72      } else {
73        root->right = nullptr;
74      }
75      return root;
76    }
77
78    void OpenFileLoad(std::string &fileName) {
79      std::ifstream is{fileName, std::ios::binary | std::ios::in};
80  //  if (is) {
81        DeleteTree(root);
82        root = Load(is);
83    // } else {
84    //  return;
85    // }
86    // is.close();
87    }
88
89    void OpenFileSave(std::string& fileName) {
90      std::ofstream os{fileName, std::ios::binary | std::ios::out};
91    // if (os) {
92        Save(os, root);
93    // } else {
94    // return;
95    // }
96    // os.close();
97    }
98
99  public:
100   void UserInsert() {
101     std::string key;
102     ull value = 0;
103     // std::cout << "UserInsert: Before cin\n";
104     std::cin >> key >> value;
105
106      // std::cout << "Before str_toLOW\n";
107     StrToLow(key);
```

```
108     // std::cout << "Before add\n";
109     Add(std::move(key), value);
110   }
111
112   void UserRemove() {
113     std::string key;
114     std::cin >> key;
115     StrToLow(key);
116     DeleteNode(std::move(key));
117   }
118
119   void UserFind(const std::string &k) {
120     std::string key{k};
121     StrToLow(key);
122     TAvlNode* res_find = Find(std::move(key));
123     if (res_find != nullptr) {
124       std::cout << "OK: " << res_find->val << "\n";
125     }
126     else {
127       std::cout << "NoSuchWord\n";
128     }
129
130   }
131
132   void SaveLoad() {
133     std::string comand;
134     std::string fileName;
135     std::cin >> comand >> fileName;
136     if (comand == "Save") {
137       OpenFileSave(fileName);
138
139     } else if (comand == "Load") {
140       OpenFileLoad(fileName);
141     }
142     std::cout << "OK\n";
143   }
144 };
```

3 Консоль

test:

```
+ a 1
+ A 2
+ aaaaaaaaaa 18446744073709551615
aaaaaaaaaa
A
-A
a
```

console output:

```
a@WIN-THNQL51M105:~/Desktop/DA/lab2# ./a.out <test
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

4 Тест производительности

Тест представляет собой сравнение моего AVL-дерева с std::map, реализованной на красно-черном дереве.

Задачей было построить деревья из n различных элементов, сделать поиск n различных значений в этих деревьях, удалить n различных элементов(элементы сразу же добавлялись обратно, но отслеживалось только время удаления). В результате работы *benchmark.cpp* видны следующие результаты:

```
a@WIN-THNQL51M105:~/Desktop/DA/lab2# ./benchmark
--numder_of_nodes = 1000 --
Insert:
std::map ms=75
avl ms=28

Find:
std::map ms=1
avl ms=0

Delete:
std::map ms=5
avl ms=3

--numder_of_nodes = 10000 --
Insert:
std::map ms=518
avl ms=3060

Find:
std::map ms=202
avl ms=45

Delete:
std::map ms=165
avl ms=201

--numder_of_nodes = 100000 --
Insert:
std::map ms=5940
avl ms=90917
```

```
Find:
std::map ms=2535
avl ms=822

Delete:
std::map ms=6610
avl ms=17866

--numder_of_nodes = 1000000 --
Insert:
std::map ms=96845
avl ms=996143

Find:
std::map ms=43218
avl ms=11257

Delete:
std::map ms=118482
avl ms=821968
```

Из данного теста видно, что АВЛ-дерево работаетт сильно эффективнее для поиска элементов, однако вставка и удаление быстрее в красно-черном дереве, что связано с необходимостью выполнять балансировку для каждой операции вставки или удаления.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я изучил различные виды деревьев поиска и реализован один из них - АВЛ-дерево.

Знание устройства и реализации этих структур данных позволит писать более эффективные программы, которые будут связаны с обработкой данных больших размеров.

АВЛ-дерево показывает отличные результаты при решении задач поиска элемента, однако и остальные опреации (добавления и удаления) происходят довольно быстро, поскольку напрямую зависят от операции поиска, временная сложность которой – O(logN).

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] *АВЛ-дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/АВЛ-дерево` (дата обращения: 14.04.2022).

[3] *АВЛ-дерево — Kvodo.*
URL: `https://kvodo.ru/avl-tree.html` (дата обращения: 14.04.2022).

[4] О. В. Сенюкова. *Сбалансированные деревья поиска. Учебно-методическое пособие.* — М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова, 2014. — 69 с.