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: К.С. Саженов
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №2

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до $2^{64}-1$. Разным словам может быть поставлен в соответствие один и тот же номер.

Вариант структуры: Дерево PATRICIA.

1 Описание

Патриция - особая структура данных, позволяющая хранить в себе пары ключ-значение(или только ключи). Ключи в патриции являются какой бы то ни было последовательностью битов(чаще всего эти ключи - строки). Узлы дерева содержат в себе пару ссылок на узлы(исключением является корень дерева - у него только одна ссылка), ключ, значение(либо только ключ) и некоторый номер бита, по которому мы будем сравнивать элементы, делая поиск в дереве.

Каждая ссылка может указывать как на «младшие», так и на «старшие».

Как можно догадаться из описания структуры, все операции основаны сравнении битов, чьи номера представлены в узлах. Именно по этим причинам сложность операции поиска оценивается как $O(k)$, где k - длина обрабатываемого ключа(поскольку номера битов являются номерами на биты в ключе). Из всего вышеперечисленного следует, что операции в патриции не зависят от количества элементов в ней.

Также есть некоторая модификация, которая упрощает поиск в данном дереве, а именно - все номера битов расположены в порядке возрастания от корня, как в структуре данных «куча», что позволяет выявлять ссылки к предкам, просто сравнивая их номера битов номера.
Все структуры, классы и вспомогательные функции я описал в файле `patricia.h`.
Также стоит рассказать о сериализации/десериализации данного дерева. Перед сериализацией происходит распределение узлов по номерам(идентификационным номерам), причем такой номер не больше количества элементов в дереве(не учитывая корень). Данная особенность позволяет записывать узлы, при этом в качестве связей указывать лишь идентификационные номера детей. Десериализация происходит аналогичным образом - сначала создается массив из пустых узлов и затем заполняется в соответствии их номерам.

2 Исходный код

Файл patricia.h:

```
1  //
2  // Created by sakost on 21.11.2020.
3  //
4
5  #ifndef LAB2_PATRICIA_H
6  #define LAB2_PATRICIA_H
7
8  #include <cinttypes>
9  #include <cstring>
10 #include <cassert>
11 #include <cmath>
12
13 #include <ostream>
14 #include <fstream>
15
16 const std::size_t KEY_LENGTH = 256;
17 using TKey = char;
18 using TValue = std::size_t;
19 const int INVALID_KEY = 1;
20
21
22 static inline bool CompareKeys(const TKey *lhs, const TKey *rhs) {
23   if (lhs == nullptr || rhs == nullptr)
24     return false;
25   return strcmp(lhs, rhs) == 0;
26 }
27
28 TKey *Copy(TKey *key) {
29   if (key == nullptr) {
30     return nullptr;
31   }
32   char *copy = new TKey[KEY_LENGTH + 1];
33   strcpy(copy, key);
34   memset(copy + strlen(copy), '\0', KEY_LENGTH+1-strlen(copy));
35   return copy;
36 }
37
38 static inline unsigned GetNBit(const TKey *key, int Bit) {
39   if (Bit < 0)
40     Bit = 0;
41   unsigned int i = (((unsigned) key[(unsigned) Bit / 8u]) >> ((7u - ((unsigned) Bit %
       8u)))) & 1u;
42   return i;
43 }
44
45 static inline std::size_t FindDifferBit(const TKey *key1, const TKey *key2) {
```

```
46    if (key1 == nullptr || key2 == nullptr) {
47      return INVALID_KEY;
48    }
49    size_t len1 = strlen(key1);
50    size_t len2 = strlen(key2);
51    auto min_size = std::min(len1, len2);
52    for (std::size_t i(0); i < min_size; i++) {
53      if (key1[i] != key2[i]) {
54        unsigned lmsb = sizeof(int) * 8 - __builtin_clz(((unsigned) key1[i] ^ (unsigned
              ) key2[i]) | 1u);
55        assert(lmsb < 8);
56        return (i << 3u) + 8 - lmsb;
57      }
58    }
59    if (len1 != len2) {
60      return (min_size << 3u) + INVALID_KEY;
61    }
62    return INVALID_KEY;
63  }
64
65  class TNode {
66  public:
67    using IdType = long long;
68    using BitType = int;
69    TNode() {
70      Key = nullptr;
71      Value = 0;
72      Left = Right = this;
73    }
74
75    TNode(TKey *key, TValue value) {
76      Key = Copy(key);
77      Value = value;
78      Left = Right = this;
79    }
80
81    ~TNode() {
82      delete[]Key;
83    }
84
85    TNode(TNode &other) {
86      if (strlen(this->Key) < strlen(other.Key)) {
87        delete[] this->Key;
88        this->Key = Copy(other.Key);
89      }
90      Left = Right = this;
91      this->Value = other.Value;
92    }
93
```

```
94    TNode &operator=(const TNode &other) {
95      if (this == &other) {
96        return *this;
97      }
98      this->Value = other.Value;
99      delete[] this->Key;
100     this->Key = Copy(other.Key);
101     return *this;
102   }
103
104   TNode *Left, *Right;
105   BitType Bit = -1;
106   TKey *Key = nullptr;
107   TValue Value = 0;
108   IdType id = -1;
109 };
110
111 bool IsBackref(TNode *from, TNode *to) {
112   return to->Bit <= from->Bit;
113 }
114
115 bool IsNormal(TNode *from, TNode *to) {
116   return !IsBackref(from, to);
117 }
118
119 static inline TNode *Choose(const TNode *cur, const TKey *key, bool reverse = false)
        {
120   if (reverse)
121     return GetNBit(key, cur->Bit) ? cur->Left : cur->Right;
122   return GetNBit(key, cur->Bit) ? cur->Right : cur->Left;
123 }
124
125
126 struct TPatricia {
127   TNode *Root = nullptr;
128   std::size_t Size = 0;
129
130   TPatricia() {
131     Root = new TNode();
132   }
133
134   ~TPatricia() {
135     Destruct(Root);
136   }
137
138   void Destruct(TNode *node) {
139     if (IsNormal(node, node->Left)) {
140       Destruct(node->Left);
141     }
```

```
142       if (IsNormal(node, node->Right)) {
143         Destruct(node->Right);
144       }
145       delete node;
146     }
147
148     TNode *AddItem(TKey *key, TValue value) {
149       TNode *cur = Get(key);
150       if (CompareKeys(key, cur->Key)) {
151         return nullptr;
152       }
153       int lBitPos = FindDifferBit(key, cur->Key);
154
155       TNode *parent = Root;
156       TNode *child = parent->Left;
157
158       while (IsNormal(parent, child) && lBitPos > child->Bit) {
159         parent = child;
160         child = Choose(child, key);
161       }
162
163       auto *node = new TNode(key, value);
164       Size++;
165
166       node->Bit = lBitPos;
167       node->Left = GetNBit(key, lBitPos) ? child : node;
168       node->Right = GetNBit(key, lBitPos) ? node : child;
169       if (GetNBit(key, parent->Bit)) {
170         parent->Right = node;
171       } else {
172         parent->Left = node;
173       }
174       return node;
175     }
176
177     TNode *Find(const TKey *key) const {
178       TNode *res = Get(key);
179       if (CompareKeys(res->Key, key)) {
180         return res;
181       }
182       return nullptr;
183     }
184
185     TNode *Get(const TKey *key) const {
186       TNode *parent, *cur;
187       parent = Root;
188       cur = Root->Left;
189       while (IsNormal(parent, cur)) {
190         parent = cur;
```

```
191         cur = Choose(cur, key);
192       }
193
194       return cur;
195     }
196
197     bool Erase(TKey *key) {
198       TNode *parent, *cur, *grand_parent = nullptr;
199
200       parent = Root;
201       cur = parent->Left;
202
203       while (parent->Bit < cur->Bit) {
204         grand_parent = parent;
205         parent = cur;
206         cur = Choose(cur, key);
207       }
208
209       if (!CompareKeys(key, cur->Key))
210         return false;
211
212       if (parent != cur)
213         *cur = *parent;
214
215       TNode *child_backref_node, *backref_node;
216       TKey *cache;
217
218       if (parent != cur) {
219         cache = parent->Key;
220
221         backref_node = parent;
222         child_backref_node = GetNBit(cache, parent->Bit) ? parent->Right : parent->Left
              ;
223
224         while (IsNormal(backref_node, child_backref_node)) {
225           backref_node = child_backref_node;
226           child_backref_node = Choose(child_backref_node, cache);
227         }
228
229         if (!CompareKeys(cache, child_backref_node->Key)) {
230           return false;
231         }
232
233         if (GetNBit(cache, backref_node->Bit))
234           backref_node->Right = cur;
235         else
236           backref_node->Left = cur;
237       }
238
```

```
239       TNode *to_replace = Choose(parent, key, true);
240       if (GetNBit(key, grand_parent->Bit))
241         grand_parent->Right = to_replace;
242       else
243         grand_parent->Left = to_replace;
244
245       Size--;
246       delete parent;
247
248       return true;
249     }
250
251     void TreeDebugPrint(std::ostream &out, TNode *cur = nullptr, int tb = -1) const {
252       if (cur == nullptr) {
253         cur = Root;
254       }
255       if (IsNormal(cur, cur->Left)) {
256         TreeDebugPrint(out, cur->Left, tb + 1);
257       }
258       if (cur != Root) {
259         char *tbs = new char[tb + 1];
260         memset(tbs, '\t', tb);
261         tbs[tb] = '\0';
262         out << tbs << "element: " << cur->Key << " " << cur->Value;
263         out << " backref to ";
264         if(cur->Left != Root){
265           out << cur->Left->Key;
266         } else{
267           out << "Root";
268         }
269         out << " and to ";
270         if(cur->Right != Root){
271           out << cur->Right->Key;
272         } else{
273           out << "Root";
274         }
275         out << std::endl;
276         delete[]tbs;
277       }
278       if (IsNormal(cur, cur->Right)) {
279         TreeDebugPrint(out, cur->Right, tb + 1);
280       }
281     }
282
283     void Serialize(std::ofstream &file) {
284       file.write((const char *) &(Size), sizeof(decltype(Size)));
285
286
287       if(Size == 0){
```

```
288         return;
289       }
290       auto nodes = new TNode *[Size+1];
291       int index = 0;
292       GenerateIds(Root, nodes, index);
293
294       TNode *node;
295
296       for (std::size_t i = 0; i <= Size; ++i) {
297         node = nodes[i];
298
299         file.write((const char *) &(node->Value), sizeof(TValue));
300         file.write((const char *) &(node->Bit), sizeof(TNode::BitType));
301         std::size_t len = 0;
302         if(node->Key != nullptr){
303           len = strlen(node->Key);
304         }
305         file.write((const char *) &(len), sizeof(decltype(len)));
306         file.write(node->Key, sizeof(TKey) * len);
307         file.write((const char *) &(node->Left->id), sizeof(TNode::IdType));
308         file.write((const char *) &(node->Right->id), sizeof(TNode::IdType));
309       }
310
311       delete[]nodes;
312     }
313
314     static void GenerateIds(TNode *nodeLoader, TNode **nodes, int &index) {
315       nodeLoader->id = index;
316       nodes[index++] = nodeLoader;
317       if (IsNormal(nodeLoader, nodeLoader->Left)) {
318         GenerateIds(nodeLoader->Left, nodes, index);
319       }
320       if (IsNormal(nodeLoader, nodeLoader->Right)) {
321         GenerateIds(nodeLoader->Right, nodes, index);
322       }
323     }
324
325     TPatricia* Deserialize(std::ifstream &file) {
326       using size_type = decltype(Size);
327
328       file.read((char *) &Size, sizeof(size_type));
329       if (!Size) {
330         return this;
331       }
332
333       auto **nodes = new TNode *[Size + 1];
334       nodes[0] = Root;
335       for (std::size_t i = 1; i < Size+1; ++i) {
336         nodes[i] = new TNode;
```

```
337       }
338
339       TNode::BitType bit;
340       TKey *key = nullptr;
341       TValue value;
342       std::size_t len;
343       TNode::IdType left_id, right_id;
344
345       for (std::size_t i = 0; i < Size + 1; ++i) {
346         file.read((char *) &value, sizeof(TValue));
347         file.read((char *) &bit, sizeof(decltype(bit)));
348         file.read((char *) &len, sizeof(decltype(len)));
349         if(len != 0) {
350           key = new TKey[len + 1];
351           key[len] = '\0';
352           file.read(key, len);
353         }
354         file.read((char *) &left_id, sizeof(decltype(left_id)));
355         file.read((char *) &right_id, sizeof(decltype(right_id)));
356         nodes[i]->Bit = bit;
357         nodes[i]->Key = key;
358         key = nullptr;
359         nodes[i]->Value = value;
360         nodes[i]->id = i;
361         nodes[i]->Left = nodes[left_id];
362         nodes[i]->Right = nodes[right_id];
363       }
364
365       delete[]nodes;
366
367       return this;
368     }
369   };
370
371
372   #endif //LAB2_PATRICIA_H
```

Файл `main.cpp`:

```
1   #include "patricia.h"
2
3   #include <cstring>
4   #include <fstream>
5   #include <iostream>
6
7   void KeyToLower(char *key) {
8     for (std::size_t i = 0; i < strlen(key); ++i) {
9     key[i] = (char)tolower(key[i]);
10    }
11  }
```

```
12
13  const int BUFFER_SIZE = 1024 * 1024;
14
15  int main() {
16    std::ios_base::sync_with_stdio(false);
17    std::cin.tie(nullptr);
18    std::cout.tie(nullptr);
19
20    std::ofstream fout;
21    std::ifstream fin;
22
23    auto *patricia = new TPatricia();
24
25    char cmd[KEY_LENGTH + 1];
26
27  #ifdef ONPC
28    std::size_t index = 0;
29  #endif
30
31    while (std::cin >> cmd) {
32  #ifdef ONPC
33    index++;
34  #endif
35    if (!std::strcmp(cmd, "+")) {
36      std::cin >> cmd;
37      TValue val;
38      std::cin >> val;
39      KeyToLower(cmd);
40      if (patricia->AddItem(cmd, val) != nullptr)
41      std::cout << "OK" << std::endl;
42      else
43      std::cout << "Exist" << std::endl;
44
45    } else if (!std::strcmp(cmd, "-")) {
46      std::cin >> cmd;
47      KeyToLower(cmd);
48
49      if (patricia->Erase(cmd))
50      std::cout << "OK" << std::endl;
51      else
52      std::cout << "NoSuchWord" << std::endl;
53
54    } else if (!std::strcmp(cmd, "!")) {
55      std::cin >> cmd;
56      if (!strcmp(cmd, "Save")) {
57      std::cin >> cmd;
58      fout.open(cmd, std::ios::out | std::ios::binary | std::ios::trunc);
59
60      patricia->Serialize(fout);
```

```
61
62      fout.close();
63      std::cout << "OK" << std::endl;
64      } else if (!strcmp(cmd, "Load")) {
65      std::cin >> cmd;
66      fin.open(cmd, std::ios::in | std::ios::binary);
67
68      delete patricia;
69      patricia = new TPatricia();
70      patricia->Deserialize(fin);
71
72      fin.close();
73      std::cout << "OK" << std::endl;
74      } else {
75      std::cout << "ERROR: no such option" << std::endl;
76      }
77    } else {
78      KeyToLower(cmd);
79
80      TNode *res = patricia->Find(cmd);
81      if (res != nullptr)
82      std::cout << "OK: " << res->Value << std::endl;
83      else
84      std::cout << "NoSuchWord" << std::endl;
85    }
86  #ifdef ONPC
87    std::cerr << index << ":" << std::endl;
88    patricia->TreeDebugPrint(std::cerr);
89    std::cerr.flush();
90  #endif
91    }
92
93    delete patricia;
94    return 0;
95  }
```

3 Консоль

```
sakost@sakost-pc ~/university/2 course/diskran/lab2 <master*>$ cat in.txt
e
-D
! Save ./db0
+ d 81173595730823721
+ E 4059181324264207759
+ b 621877380789371611
a
A
-A
-D
b
! Save ./db1
! Load ./db1
! Load ./db0
-d
sakost@sakost-pc ~/university/2 course/diskran/lab2 <master*>$ cmake-build-debug/lab2
<in.txt
NoSuchWord
NoSuchWord
OK
OK
OK
OK
NoSuchWord
NoSuchWord
NoSuchWord
OK
OK: 621877380789371611
OK
OK
OK
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: генерируется тест посредством своего собственного чекера/генератора опеределенное количество тестов(без тестов записи-загрузки) и затем данный тест подается на вход двум программам с замерой времени внутри них. В таблице указаны значения в микросекундах.

Длина ключа от 1 до 3	
168616	PATRICIA
239682	Встроенная структура map
Длина ключа от 2 до 250	
806266	PATRICIA
1390827	Встроенная структура map

Как видно, `PATRICIA` значительно превосходит встроенную структуру `map`, основанную на красно-черном дереве, из-за того, что `PATRICIA` не производит полного сравнения с ключом каждый раз, приходя в узел, а лишь при конечном сравнении. В красно-черном дереве все операции(поиска/удаления/вставки) оцениваются в $O(log(n) * k)$, где n - количество узлов в дереве, а k - длина ключа, в то время как в `PATRICIA` сложность этих же операций оценивается в $O(k)$, где k - длина ключа. Причем даже если сравнивать `PATRICIA` и `Trie`, то получается, что `PATRICIA` работает даже быстрее своего предшественника, поскольку в `Trie` производится сравнение всего ключа, в то время как в `PATRICIA` производится сравнение только отдельных битов(и только в конце сравнивается вся строка). Итого: при абсолютно рандомном тесте `PATRICIA` уже работает быстрее, чем встроенный `map`.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я узнал, как работают сбалансированные деревья поиска, а именно PATRICIA.

PATRICIA деревья - очень удобны, практичны и активно используются в местах, где требуется быстрая работа с ассоциативными массивами, ключи которых можно представить в виде строк/последовательность бит. PATRICIA, в частности, применяется в сфере IP маршрутизации, где возможность содержать большие диапазоны значений, за некоторыми исключениями, особенно подходит иерархической организации IP адресов. PATRICIA также может использоваться для инвертированных индексов текстовых документов в информационном поиске.

С первого взгляда может показаться, что PATRICIA работает медленнее других сбалансированных деревьях, таких как КЧ-дерево или АВЛ-дерево, и всё же стоит понимать, что при каждом проходе в узлах PATRICIA сравнивает лишь один бит, в то время как другие структуры данных сравнивают всю строку целеком. Следовательно, PATRICIA деревья работают быстрее других сбалансированных, которые я прошел на куре «Дискретный анализ», с ключами в виде строк.

И всё же преимущество также является недостатком деревьев PATRICIA, поскольку требование того, что ключ можно представить в представлении строки/последовательности бит не всегда выполнимо. Данный факт играет в пользу других сбалансированных структур данных, поскольку они являются более «универсальными», для которых нужно лишь отношение порядка, а не как в случае с деревьями PATRICIA - представление в виде строки/последовательности битов.

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] *Radix tree - Wikipedia.*
URL: `https://en.wikipedia.org/wiki/Radix_tree` (дата обращения: 27.11.2020).

[3] *Patricia Tries - Dr Dobb's.*
URL: `https://www.drdobbs.com/architecture-and-design/patricia-tries/208800854` (дата обращения: 27.11.2020).

[4] Handbook of data structures and applications */ edited by Dinesh P. Mehta and Sartaj Sahni. p. cm. - (Chapman & Hall/CRC computer & information science)* — ISBN 1-58488-435-5 (alk. paper)