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: Меджидли И. И. о
Преподаватель: Михайлова С. А
Группа: М8О-201Б-21

Дата: 07.09.2023
Оценка:
Подпись:

Москва, 2023

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.

Вариант структуры: PATRICIA.

1 Описание

Patricia:

Patricia — усовершенствованное суффиксное дерево (Trie). Корень дерева - единственный элемент в дереве, у которого 1 потомок (левый), у остальных элементов 2 потомка (левый и правый). У каждого узла (структура Node) содержится некоторый индекс — номер бита, который должен проверяться с целью выбора пути (потомка) из этого узла. У корня этот бит - 0, т.е. ничего не проверяем, а идем по ссылке в левый потомок. Далее при проверке у остальных элементов, если данный порядковый бит равен 0, то идём по ссылке налево, если - 1, то направо. Каждый раз, опускаясь по патриции, мы проверяем большие индексы.

Ссылки могут быть прямые (когда номер бита увеличивается) и обратные (когда уменьшается, либо не меняется, т.е. возвращается в себя). У каждого узла ровно одна обратная ссылка.

Каждый из узлов хранит в себе ключ (string), значение (unsigned long long), индекс (номер бита, по которому идёт сравнение для продвижения по дереву) `(size_t)`, указатели на левого и правого потомка (причем потомок может быть выше в дереве, чем текущий узел ввиду обратных ссылок).

В целом, Patricia является доработанным бинарным деревом. Сложность добавления, поиска и удаления элемента в словарь, основанный на Patricia — O(h), где h — высота дерева.

2 Исходный код

Основные этапы:

1) Реализация класса Patricia.

2) Чтение пользовательских запросов.

3) Выполнение требуемых запросов.

```
#include <iostream>
#include <string>
#include <tuple>
#include <cctype>
#include <algorithm>
#include <vector>
#include <fstream>
#include <sstream>

using namespace std;

// Bits u need in one char
const int BIT_COUNT = 5;

struct forArray{
    int lenkey;
    string key;
    unsigned long long value;
};

typedef class Patricia{
private:
    struct Node;
    Node *root = nullptr;
    // Search but return 3 nodes
    tuple<Node*, Node*, Node*> SearchE(const string& findKey) const;
    // Search return only 1 node
    Node* Search(const string& findKey) const;
    // Insert node to tree
    void Insert(const string& key, const unsigned long long& value, const size_t& index
        );
    void ClearNode(Node *node);
    void Show(Node *node, vector <forArray>& savear, int& i);
    // void split(string str, vector<string>& res);
public:
    // Get value from map
    unsigned long long& At(const string& findKey) const;
    // Create node with key value and insert to tree
    void Add(const string& key, unsigned long long value);
    // Delete node from tree
    void Erase(const string& key);
```

```
41      void Save(string& path);
42      void Load(string& path);
43      void Clear();
44      ~Patricia(){}
45  } patr;
46
47  struct Patricia::Node{
48      string key;
49      unsigned long long value;
50      size_t index;
51      Node *right, *left;
52
53      Node(const string& key, const unsigned long long& value, const int& index)
54          : key(key), value(value), index(index), left(nullptr), right(nullptr) { }
55
56      ~Node(){}
57  };
58
59
60
61  Patricia::Node* Patricia::Search(const string& findKey) const{
62      Node *currentNode = root->left, *prevNode = root;
63
64      while(currentNode->index > prevNode->index){
65          // Index of char that we need to check
66          size_t charIndex = (currentNode->index - 1) / BIT_COUNT;
67
68          // findKey is less than need char
69          if(charIndex >= findKey.size()){
70              // Remember prevNode
71              prevNode = currentNode;
72              // Only '0'
73              currentNode = currentNode->left;
74              continue;
75          }
76
77          char currentChar = findKey[charIndex];
78          // How many times should we shift to the right
79          int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
80          // Get current bit
81          bool currentBit = (currentChar >> offset) & 1;
82
83          // Remember prevNode
84          prevNode = currentNode;
85          // If '1' go right, '0' go left
86          currentBit ? currentNode = currentNode->right
87                     : currentNode = currentNode->left;
88      }
89      return currentNode;
```

```
90  }
91
92  void Patricia::Add(const string& key, unsigned long long value){
93      if(!root){
94          root = new Node(key, value, 0);
95          root->left = root;
96          return;
97      }
98
99      Node *foundNode = Search(key);
100     if(foundNode->key == key)
101         throw runtime_error("Exist\n");
102
103     bool run = true;
104     size_t charIndex = 0;
105     while(run){
106         char foundedKey = (foundNode->key.size() > charIndex ? foundNode->key[charIndex
                ] : '\0');
107         char inputKey = (key.size() > charIndex ? key[charIndex] : '\0');
108         for(size_t i = 0; i < BIT_COUNT; ++i){
109             bool foundedKeyBit = foundedKey >> (BIT_COUNT - 1 - i) & 1;
110             bool inputKeyBit = inputKey >> (BIT_COUNT - 1 - i) & 1;
111             if(foundedKeyBit != inputKeyBit){
112                 Insert(key, value, charIndex * BIT_COUNT + i + 1);
113                 run = false;
114                 break;
115             }
116         }
117         ++charIndex;
118     }
119 }
120
121 void Patricia::Insert(const string& key, const unsigned long long& value, const size_t
        & index){
122     Node *currentNode = root->left, *prevNode = root;
123
124     while(currentNode->index > prevNode->index && currentNode->index <= index){
125         size_t charIndex = (currentNode->index - 1) / BIT_COUNT;
126         // FindKey is less than need char
127         if(charIndex >= key.size()){
128             prevNode = currentNode;
129             // Only '0'
130             currentNode = currentNode->left;
131             continue;
132         }
133         char currentChar = key[charIndex]; // If findkey.size less than currentNode->
                index
134         int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
135         bool currentBit = (currentChar >> offset) & 1;
```

```
136
137         // Remember prevNode
138         prevNode = currentNode;
139         // If '1' go right, '0' go left
140         currentBit ? currentNode = currentNode->right : currentNode = currentNode->left
                ;
141     }
142
143     char getCharFromKey = key[(index - 1) / BIT_COUNT];
144     bool getBit= getCharFromKey >> (BIT_COUNT - 1 - (index - 1) % BIT_COUNT) & 1;
145     Node *newNode = new Node(key, value, index);
146
147     if(prevNode->left == currentNode)
148         prevNode->left = newNode;
149     else
150         prevNode->right = newNode;
151
152     getBit ? (newNode->right = newNode, newNode->left = currentNode) : (newNode->left =
           newNode, newNode->right = currentNode);
153 }
154
155 unsigned long long& Patricia::At(const string& findKey) const{
156     if(!root)
157         throw runtime_error("NoSuchWord\n");
158
159     Node* get = Search(findKey);
160
161     if(get->key == findKey)
162         return get->value;
163
164     throw runtime_error("NoSuchWord\n");
165 }
166
167 tuple<Patricia::Node*, Patricia::Node*, Patricia::Node*> Patricia::SearchE(const
        string& findKey) const{
168     Node *currentNode = root->left, *prevNode = root, *prevPrevNode = root;
169     while(currentNode->index > prevNode->index){
170         size_t charIndex = (currentNode->index - 1) / BIT_COUNT;
171
172     // FindKey is less than need char
173     if(charIndex == findKey.size()){
174         prevPrevNode = prevNode;
175         prevNode = currentNode;
176         // Only '0'
177         currentNode = currentNode->left;
178         continue;
179     }
180
181     char currentChar = findKey[charIndex];// If findkey.size less than currentNode-&>;
```

```
        index
    int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
    bool currentBit = (currentChar >> offset) & 1;

    // Remember prevNode &amp;amp; prevPrevNove
    prevPrevNode = prevNode;
    prevNode = currentNode;
    // If '1' go right, '0' go left
    currentBit ? currentNode = currentNode->right : currentNode = currentNode->left;
}

return make_tuple(currentNode, prevNode, prevPrevNode);

}

void Patricia::Erase(const string& key){
    if(!root) throw runtime_error("NoSuchWord\n");

    // Get delete node, owner delete node and parent owner
    tuple<Node*, Node*, Node*> delOwnerParentTuple = SearchE(key);
    // A
    Node *deleteNode = get<0>(delOwnerParentTuple);
    //
    Node *ownerDeleteNode = get<1>(delOwnerParentTuple);
    //  parent
    Node *parentOwnerDeleteNode = get<2>(delOwnerParentTuple);

    if(deleteNode->key != key){
        throw runtime_error("NoSuchWord\n");
    }
    // If delete node is root and it's one
    if(deleteNode == root && root->left == root){
        delete root;
        root = nullptr;
        return;
    }
 // If delete node is leaf
    if(ownerDeleteNode == deleteNode){
        if(parentOwnerDeleteNode->right == deleteNode)
            if(deleteNode->right == deleteNode)
                parentOwnerDeleteNode->right = deleteNode->left;
            else
                parentOwnerDeleteNode->right = deleteNode->right;
        else
            if(deleteNode->right == deleteNode)
                parentOwnerDeleteNode->left = deleteNode->left;
            else
                parentOwnerDeleteNode->left = deleteNode->right;
        delete deleteNode;
```

```
230         deleteNode = nullptr;
231         return;
232     }
233 
234     // Get owner owner delete node
235     tuple<Node*, Node*, Node*> ownerOwnerTuple = SearchE(ownerDeleteNode->key);
236     Node *ownerOwnerDelNode = get<1>(ownerOwnerTuple);
237 
238     // Change item from owner delete node to delete node
239     deleteNode->key = ownerDeleteNode->key;
240     deleteNode->value = ownerDeleteNode->value;
241      // If owner delete node is leaf
242     if(ownerOwnerDelNode == ownerDeleteNode){
243         if(parentOwnerDeleteNode->right == ownerDeleteNode)
244             parentOwnerDeleteNode->right = deleteNode;
245         else
246             parentOwnerDeleteNode->left = deleteNode;
247     }
248     else{
249         // Tie parent owner delete node with child
250         if(parentOwnerDeleteNode->right == ownerDeleteNode)
251             if(ownerDeleteNode->right == deleteNode)
252                 parentOwnerDeleteNode->right = ownerDeleteNode->left;
253             else
254                 parentOwnerDeleteNode->right = ownerDeleteNode->right;
255         else
256             if(ownerDeleteNode->right == deleteNode)
257                 parentOwnerDeleteNode->left = ownerDeleteNode->left;
258             else
259                 parentOwnerDeleteNode->left = ownerDeleteNode->right;
260 
261         if(ownerOwnerDelNode->right == ownerDeleteNode)
262             ownerOwnerDelNode->right = deleteNode;
263         else
264             ownerOwnerDelNode->left = deleteNode;
265     }
266     delete ownerDeleteNode;
267     ownerDeleteNode = nullptr;
268 }
269 
270 void Patricia::Clear(){
271     if(!root)
272         return;
273     if(root != root->left)
274         ClearNode(root->left);
275     delete root;
276     root = nullptr;
277 }
278 
```

```
279 void Patricia::ClearNode(Node *node){
280     if((node->left->index > node->index) && (node->left != node) && (node->left != root
        ))
281         ClearNode(node->left);
282     if((node->right->index > node->index) && (node->right != node) && (node->right !=
        root))
283         ClearNode(node->right);
284 
285     delete node;
286     node = nullptr;
287 }
288 
289 void Patricia::Load(string& path)
290 {
291     ifstream fin;
292     string str;
293     fin.open(path);
294 
295     while(getline(fin, str)){
296         stringstream ss(str);
297         int a;
298         unsigned long long val;
299         string key;
300         ss >> a >> key >> val;
301         Add(key, val);
302     }
303 }
304 
305 
306 void Patricia::Save(string& path)
307 {
308         ofstream fout;
309         fout.open(path);
310     if(root){
311         Node *currentNode = root;
312         Node *prevNode = nullptr;
313         vector <forArray> savear;
314 
315         forArray info;
316         info.lenkey = currentNode->key.length();
317         info.key = currentNode->key;
318         info.value = currentNode->value;
319         savear.push_back(forArray());
320         savear[0] = info;
321         prevNode = currentNode;
322         currentNode = currentNode->left;
323         if(prevNode != currentNode){
324         int i = 1;
325         int& a = i;
```

```
326         Show(currentNode, savear, a);
327         }
328         for(auto& k:savear){
329         fout << k.lenkey << " " << k.key << " " << k.value << "\n";
330         }
331     }
332 }
333 
334 void Patricia::Show(Node *node, vector <forArray>& savear, int& i)
335 {
336     forArray info;
337     info.lenkey = node->key.length();
338     info.key = node->key;
339     info.value = node->value;
340     savear.push_back(forArray());
341     savear[i] = info;
342     // cout <<node->value << "\n"; //
343     if (node->left->index > node->index){
344         Show(node->left, savear, ++i); //
345     } //  ,
346 
347     if (node->right->index > node->index){
348         Show(node->right, savear, ++i); //
349     } //  ,
350     return;
351 }
352 
353 
354 int main(void){
355     patr imedy;
356     string command;
357     while(cin >> command && command != "\x4"){
358         if(command == "+"){
359             string key; cin >> key;
360             transform(key.begin(), key.end(), key.begin(),
361             [](unsigned char c){ return tolower(c); });
362             // key = tolower(key);
363             unsigned long long value; cin >> value;
364             try{
365                 imedy.Add(key, value);
366                 cout << "OK\n";
367             }
368             catch(const runtime_error& e){
369                 cout << e.what();
370             }
371         }
372         else if(command == "-"){
373             string key; cin >> key;
374             transform(key.begin(), key.end(), key.begin(),
```

```
375             [](unsigned char c){ return tolower(c); });
376             try{
377                 imedy.Erase(key);
378                 cout << "OK\n";
379             }
380             catch(const runtime_error& e){
381                 cout << e.what();
382             }
383         }
384         else if(command == "clear" || command == "new"){
385             imedy.Clear();
386         }
387         else if(command == "!"){
388             string key, path;
389             cin >> key;
390             cin >> path;
391             if(key == "Save"){
392                 try{
393                     imedy.Save(path);
394                     cout << "OK\n";
395                 }
396                 catch(const runtime_error& e){
397                     cout << e.what();
398                 }
399             }
400 
401             if(key == "Load"){
402                 try{
403                 patr forload;
404           imedy.Clear();
405           forload.Load(path);
406                 // imedy.Load(path);
407                 imedy = forload;
408                 cout << "OK\n";
409                 }
410                 catch(const runtime_error& e){
411                     cout << e.what();
412                 }
413             }
414         }
415         else{
416             try{
417                 string key = command;
418             transform(key.begin(), key.end(), key.begin(),
419             [](unsigned char c){ return tolower(c); });
420                 unsigned long long imkey = imedy.At(key);
421                 cout << "OK: "<< imkey << '\n';
422             }
423             catch(const runtime_error& e){
```

```
424                 cout << e.what();
425             }
426         }
427     }
428     imedy.Clear();
429     return 0;
430 }
```

3 Тесты

Тестировать программу буду ручным способом.

```
imedzhidli@imedzhidli:~/Desktop/DA/LABA2$ ./lab2
+ a 1
OK
A
OK: 1
+ A 2
Exist
+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
18446744073709551615
OK
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
OK: 18446744073709551615
A
OK: 1
-A
OK
a
NoSuchWord
+ a 77
OK
+ aav 3
OK
a
OK: 77
aav
OK: 3
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
OK: 18446744073709551615
imedzhidli@imedzhidli:~/Desktop/DA/LABA2$
```

Как можно заметить, все верно.

4 Выводы

При выполнении второй лабораторной работы по курсу «Дискретный анализ» я познакомился с такой интересной структурой данных, как Patricia, реализовал ее и методы работы с ней. Надеюсь мне понадобятся знания, полученные при выполнении этой лабораторной работы.