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: И. В. Кочкожаров
Группа: М8О-208Б-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. 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.

Различия вариантов заключаются только в используемых структурах данных.

Вариант структуры данных: PATRICIA.

1 Описание

Требуется написать реализацию PATRICIA Trie.

Как сказано в [1]: «Particia – сжатый бинарный trie, в котором ветка и узлы элементов объединены в единый узел». Каждый узел помимо сохраняемых данных хранит номер бита, который будет проверяться у ключа во время поиска, и два указателя: на левого и правого ребёнка.

2 Исходный код

Так как словарь должен хранить пару «ключ-значение», создадим структуру *TPair*. Для узлов дерева в классе *TPatriciaTrie* определим приватную структуру *TNode*, которая будет содержать поля *data, children, bitNumber, id*. Так же определим структуру *TTCaseInsensitiveString*, которая будет содержать в себе регистронезависимую строку фиксированного размера. Для удобства переопределим в классе структуру типа *TPair<TTCaseInsensitiveString, uint64_t>* как *TData*.

Определим битовые операции со строками. Функция *bool GetBitByIndex(const std::string& str, int index)* возвращает значение бита с номером *index* в строке *str*, используя двоичную маску. Функция *int GetBitDifference(const std::string& a, const std::string& b)* возвращает первый слева номер бита, различного для строк a и b.

Реализуем поиск по ключу в дереве. Напишем приватный вспомогательный метод *TPair<TPatriciaTrie::TNode*, int> FindPreviousNode(const TTCaseInsensitiveString& key, int bitNumber)*. Он возвращает пару, содержащую указатель на предыдущий узел (родитель искомого) и значение n-ного бита, которое необходимо для перехода к искомому узлу. Функция осуществляет обход дерева. В каждом узле вычисляется значения бита с номером *bitNumber* в ключе. Если оно равно 0, то происходит переход по левому указателю, иначе – по правому. Обход останавливается в момент, когда происходит переход по обратному указателю, то есть значение *bitNumber* следующего меньше или равно значения *bitNumber* предыдущего.

Приватный метод *TNode*& FindNode(const std::string& key, int bitNumber)* и публичный *const TData& Find(const std::string& key)* используют метод *FindPreviousNode*.

Реализуем вставку в дерево – метод *bool Insert(const TData& data)*. Алгоритм вставки *theKey* (взят из [1]):

1. Найти *theKey*. Пусть *reachedKey* – ключ узла, на котором поиск закончился.
2. Определить первый бит слева *lBitPos*, различный для *theKey* и *reachedKey*.
3. Создать новый узел с ключом *theKey*, в поле *bitNumber* записать *lBitPos*. Вставить этот узел между другими узлами, которые были пройдены во время поиска, так, чтобы последовательность из *bitNumber* была возрастающей. Эта вставка сломает указатель между двумя узлами p и q. Указатель из p теперь содержит новый узел.
4. Если *lBitPos* узла с ключом *theKey* равен 1, то указатель на правого ребёнка становится обратным указателем на этот узел. Иначе – указатель на левого ребёнка становится обратным. Оставшийся указатель будет содержать q.

Реализуем удаление из дерева – метод *void Erase(const std::string& key)*. Пусть узел p – узел, который мы хотим удалить. Возможны два случая:

1. У p есть указатель на самого себя. Если p – корень, то узел удаляется, а дерево становится пустым. Иначе – указатель на p его родителя устанавливаем к несобственному (который не указывает на самого себя) указателю.

2. У p нет указателя на самого себя. Ищем узел q, у которого есть обратный указатель на p. Данные, которые хранятся в q перемещаются в p, и мы удаляем q. Чтобы удалить q, нужно найти узел r, у которого есть обратный указатель на q. Обратный указатель на q изменяем, чтобы он указывал на p. Прямой указатель от родителя q изменяем, чтобы он указывал на ребёнка q.

Метод *void SaveToFile(FILE* file) const* сохраняет дерево в файл. Для этого сначала все узлы дерева записываются в массив с помощью метода *void TreeToArray(TNode** array, TNode* root, int& id) const*. Все узлы, при этом, пронумерованы (поле *id*). Далее в файл записывается количество узлов, затем все узлы последовательно.

Метод *void LoadFromFile(FILE* file)* загружает дерево из файла. Для этого все узлы из файла считываются, а потом с помощью метода *void ArrayToTree(TSaveData* array)* строится дерево. Указатели восстанавливаются с помощью *id*.

binary_string.h	
int GetBitSize(const std::string& str)	Функция получения количества бит в строке.
int GetBitDifference(const std::string& a, const std::string& b)	Функция получения индекса различного бита.
bool GetBitByIndex(const std::string& str, int index)	Функция получения значения бита по индексу.

```
1
2  struct TFile {
3      enum class FileType { Save, Load };
4      FILE* file;
5      TCaseInsensitiveString name;
6      TFile(TCaseInsensitiveString s, FileType mode) {
7          name = s;
8          if (mode == FileType::Save) {
9              file = fopen(s.CStr(), "wb");
10         } else {
11             file = fopen(s.CStr(), "rb");
12         }
13
14     }
15     bool check() {
16         return file != nullptr;
17     }
18     FILE* GetFile() { return file; }
19     TFile() = delete;
```

```
20      TFile(const TFile& other) = delete;
21      TFile(TFile&& other) = delete;
22      TFile& operator=(const TFile& other) = delete;
23      TFile& operator=(TFile&& other) = delete;
24      ~TFile() { fclose(file); }
25  };
26
27  template <class T, class U>
28  struct TPair {
29      T key;
30      U value;
31
32      TPair() = default;
33      TPair(const T& key, const U& value) : key(key), value(value) {}
34      TPair(T&& key, U&& value) : key(key), value(value) {}
35      TPair(const TPair& other) : key(other.key), value(other.value) {}
36      TPair(TPair&& other) noexcept
37          : key(std::move(other.key)), value(std::move(other.value)) {}
38      TPair& operator=(const TPair& other) {
39          key = other.key;
40          value = other.value;
41          return *this;
42      }
43      TPair& operator=(TPair&& other) noexcept {
44          key = std::move(other.key);
45          value = std::move(other.value);
46          return *this;
47      }
48  };
49
50  class TPatriciaTrie {
51     private:
52      using TData = TPair<TCaseInsensitiveString, uint64_t>;
53
54      struct TNode {
55          TData data;
56          TNode* children[2];
57          int bitNumber;
58          int id;
59
60          TNode() = default;
61          TNode(const TData& data);
62      };
63
64      struct TSaveData {
65          int id;
66          char key[257];
67          uint64_t value;
68          int bitNumber;
```

```
69          int leftId;
70          int rightId;
71      };
72
73      TNode* root;
74      int size;
75
76      TNode*& FindNode(const TCaseInsensitiveString& key, int bitNumber);
77      TPair<TPatriciaTrie::TNode*, int> FindPreviousNode(
78          const TCaseInsensitiveString& key, int bitNumber);
79      void DestroyTrie(TNode* node);
80      void TreeToArray(TNode** array, TNode* root, int& id) const;
81      void ArrayToTree(TSaveData* array);
82
83    public:
84      TPatriciaTrie();
85      ~TPatriciaTrie();
86      bool Insert(const TData& data);
87      const TData *Find(const TCaseInsensitiveString& key);
88      bool Erase(const TCaseInsensitiveString& key);
89      bool SaveToFile(FILE* file) const;
90      bool LoadFromFile(FILE* file);
91      int Size() const;
92  };
```

3 Консоль

```
ivan@asus-vivobook ~/c/d/b/lab2 (master)>cat testcase
+ word 444
+ word 3
+ wordy 666
! Save patricia.dat
+ wo 555
! Load patricia.dat
word
-word
wo
ivan@asus-vivobook ~/c/d/b/lab2 (master)>./lab2 <testcase
OK
Exist
OK
OK
OK
OK
OK: 444
OK
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: Patricia Trie сравнивается с *std::map* на 3 тестах с разным количеством входных данных от 10^3 до 10^5, входные данные из себя представляют набор команд добавления случайных ключей.

```
ivan@asus-vivobook ~/c/d/build (master)>./lab2/lab2_benchmark 10000
std::map ms=11012
patricia ms=7952
ivan@asus-vivobook ~/c/d/build (master)>./lab2/lab2_benchmark 100000
std::map ms=145460
patricia ms=127428
ivan@asus-vivobook ~/c/d/build (master)>./lab2/lab2_benchmark 1000000
std::map ms=2088952
patricia ms=1680058
```

Как видно, Patricia Trie в среднем работает за то же время, что и *std::map*, а при большом количестве входных данных обгоняет *std::map* в производительности.

Это связано с тем, что *std::map* работает на красно-чёрном дереве. Сложность поиска и вставки для него – $O(\log n)$, где n – количество элементов в дереве. Так как ключом является строка, то сложность поиска и вставки становится равной $O(k \log n)$, где k – длина ключа.

Сложность вставки и поиска в Patricia Trie равна $O(h)$, где h – высота дерева. Получение i-того бита в строке обходится в $O(1)$. В конце поиска происходит полное сравнение ключей, поэтому сложность поиска и вставки равна $O(\max(h, k))$, где k – длина строки (ключа).

5 Выводы

В результате выполнения второй лабораторнлй работу по курсу «Дискретный анализ», была реализована структура данных Patrcia Trie. В ходе решения этой задачи самой трудной проблемой было написание алгоритмов осуществления различных операций над trie, а так же способа сохранения trie в бинарном файле.

Список литературы

[1] Mehta, Dinesh P, Sahni, Sartaj Handbook of data structures and applications. – Chapman & Hall/CRC, 2004. – 1321 с.