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: Т. Д. Голубев
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. 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*. Для удобства переопределим в классе структуру типа *TPair<std::string, 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 и битовые сдвиги. Функция *int BitDifference(const std::string& a, const std::string& b)* возвращает первый слева номер бита, различного для строк a и b. Её реализуем с помощью применения операции XOR для каждой буквы в строках.

Реализуем поиск по ключу в дереве. Напишем приватный вспомогательный метод *TPair<TPatriciaTrie::TNode*, int> FindPreviousNode(const std::string& 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 *void 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(std::ofstream& 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(std::ifstream& 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 BitDifference(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  template <class T, class U>
2  struct TPair {
3    T key;
4    U value;
5
6    TPair() = default;
7    TPair(const T& key, const U& value) : key(key), value(value) {}
8    TPair(T&& key, U&& value) : key(key), value(value) {}
9    TPair(const TPair& other) : key(other.key), value(other.value) {}
10   TPair(TPair&& other) noexcept : key(std::move(other.key)), value(std::move(other.
         value)) {}
11   TPair& operator=(const TPair& other) {
12     key = other.key;
13     value = other.value;
14     return *this;
15   }
16   TPair& operator=(TPair&& other) noexcept {
17     key = std::move(other.key);
18     value = std::move(other.value);
```

```
    return *this;
  }
};

class TPatriciaTrie {
private:
  using TData = TPair<std::string, uint64_t>;

  static const int KEY_LENGTH = 256;

  struct TNode {
    TData data;
    TNode* children[2];
    int bitNumber;
    int id;

    TNode() = default;
    TNode(const TData& data);
  };

  struct TSaveData {
    int id;
    char key[KEY_LENGTH];
    uint64_t value;
    int bitNumber;
    int leftId;
    int rightId;

    TSaveData() = default;
  };

  TNode* root;
  int size;

  TNode*& FindNode(const std::string& key, int bitNumber);
  TPair<TPatriciaTrie::TNode*, int> FindPreviousNode(const std::string& key, int
      bitNumber);
  void DestroyTrie(TNode* node);
  void TreeToArray(TNode** array, TNode* root, int& id) const;
  void ArrayToTree(TSaveData* array);
public:
  TPatriciaTrie();
  ~TPatriciaTrie();
  void Insert(const TData& data);
  const TData& Find(const std::string& key);
  void Erase(const std::string& key);
  void SaveToFile(std::ofstream& file) const;
  void LoadFromFile(std::ifstream& file);
  int Size() const;
```

```
67 || };
```

3 Консоль

```
cat_mood@nuclear-box:~/programming/mai-da-labs/lab02$ cat tests/e2e/01.t
+ UMjAwDPdsTIreIoUazQKanmGzPRfV 6469694522297445932
UMjAwDPdsTIreIoUazQKanmGzPRfV
+ eMjqiAsaEpQcSWszireOTdmxKodxZXaVgAOLtpjpPRSMrRHTbwbWdSqpyLvHMltcgXVhmZgEBs
2376015365768985450
+ CwByzdpKndACUO 3782028637663810452
+ bnNLoaZXzMwRndsZJySZQsDdVSwYgNjgmPlheYgTxJZDjgWQgmQmT 8041966132991934101
+ ahjH 2383040096089261832
+ gYpTihvBTqlcMhmyJHBqixheUJDLMmCIKJdBkaZzkbCTixusE 2087302814296979551
pMHaUSAjlOpryXlmoJITsjPAbhPYUJCm
+ svWDmUQhtlPHrRwJceWuQZekrwBaxNcCJJIWWG 8243555221965638719
+ UtKJMEZUCwOHAOKjtKofDpMUbmacFfkfvlXJT 1719112497809751096
-eMjqiAsaEpQcSWszireOTdmxKodxZXaVgAOLtpjpPRSMrRHTbwbWdSqpyLvHMltcgXVhmZgEBs
-asdsa
cat_mood@nuclear-box:~/programming/mai-da-labs/lab02$ build/lab02_exe <tests/e2e/01.t
OK
OK: 6469694522297445932
OK
OK
OK
OK
OK
NoSuchWord
OK
OK
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 случайный набор команд добавления и поиска.

```
[info] [2024-04-24 18:40:06] Running tests/e2e/01.t
std::map ms=23750
patricia ms=26714
[info] [2024-04-24 18:40:07] Running tests/e2e/02.t
std::map ms=23274
patricia ms=19875
[info] [2024-04-24 18:40:07] Running tests/e2e/03.t
std::map ms=7341
patricia ms=5843
```

Как видно, Patricia Trie в среднем работает за то же время, что и *std::map*.

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

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я реализовал структуру данных Patrcia Trie. В ходе работы столкнулся с проблемой представления строки в бинарном виде и сравнения двух строк в этом виде. В своей реализации я решил эту проблему. Также пришлось поработать с бинарным файлом и решить проблему сохранения и загрузки бинарного дерева в файл.

Список литературы

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