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: П. Ф. Гришин
Преподаватель: С. А. Михайлова
Группа: М8О-201Б-21

Дата:
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.

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.

Структура данных: Красно-черное дерево.

1 Описание

Требуется написать реализацию красно-черного дерева. Данная структура данных является одной из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счет введения дополнительного атрибута узла дерева — «цвет». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный». Красно-чёрное дерево обладает следующими свойствами:

1. Каждый узел окрашен либо в красный, либо в черный цвет.
2. Корень и конечные узлы (листья) дерева — чёрные.
3. У красного узла родительский узел — чёрный.
4. Пути от узла к его листьям должны содержать одинаковое количество черных узлов.
5. Чёрный узел может иметь чёрного родителя.

Из указанных свойств можно вывести, что высота дерева ограничена логарифмической функцией от числа узлов. Есть такие алгоритмы выполнения операции добавления узлов (INSERT) и удаления узла (DELETE), которые поддерживают свойства красно-чёрного дерева, а значит и свойство сбалансированности высоты. Когда узел добавляется или удаляется из красно-черного дерева, выполняются различные операции перекраски и поворота, чтобы сохранить свойства красно-черного дерева. Операция добавления узла включает в себя несколько случаев, которые могут возникнуть в зависимости от текущей структуры дерева и цвета узлов. Аналогично, операция удаления узла также требует учета различных случаев.

2 Исходный код

Непосредственно дерево представляется классом RBTree, в котором имеется поле root, представленное классом Node. Этот класс реализовывает модель вершины дерева и имеет поля red, parent, left и right. Такое разделение на классы произведено с двумя целями:

1. В определённый момент времени дерево может быть пустым — значение root равно NIL. Любое прямое обращение к нему приводило бы к ошибке.

2. Методы класса Node имеют дополнительные параметры, необходимые для их описания.

Использование дополнтиельной структуры RBTree позволяет решить обе этих проблемы. Перейдём к описанию поиска, вставки и удаления элементов.

• Поиск. Поиск в RB-дереве производится точно так же, как и в BST. Запускаем поиск от корня и на каждом этапе смотрим, совпадает ли ключ вершины с ключом, по которому мы ищем элемент. Если да, то элемент найден — поиск закончен. Иначе сравниваем ключ вершины с ключом поиска. Если ключ вершины больше, то проверяем, есть ли у вершины левое поддерево: да — запускаем поиск в нём, нет — элемента в дереве нет.

Аналогично для случая, когда ключ вершины меньше ключа поиска, но работаем уже с правым поддеревом.

• Вставка. На первом этапе производим поиск переданного ключа — такой элемент в дереве уже может существовать. В таком случае вставка заканчивается, ничего не изменив. Если же в процессе поиска мы не нашли элемента с таким ключом, то создаём новую вершину и связываем её с деревом (в процессе поиска мы уже нашли её место — она станет поддеревом вершины, на которой мы завершили поиск, не найдя у неё левого или правого поддерева). Конечным этапом будет ребалансировка: Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

1. "Дядя"этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца"и "дядю"в чёрный цвет, а "деда"— в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы"черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2.

2. "Дядя"чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем пово-

рот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.

• Удаление. Первый этап удаления такой же, как и в BST: находим удаляемый элемент, и если он есть, то выбираем один из 3 вариантов в зависимости от количества поддеревьев у удаляемого элемента. Если их нет вообще — просто удаляем вершину и у родителя значение соответствующего поддерева делаем NIL. Если есть один — связываем поддерево удаляемого элемента с соответствующим поддеревом родительской вершины, после чего удаляем вершину. Если есть два — находим минимальный элемент в правом поддереве, удаляем его, а значение записываем в нашу такущую вершину. Следующий этап — ребалансировка. При удалении красной вершины свойства RB-дерева не нарушаются. В случае, если мы удаляем черную вершину, то потребуется балансирока. Рассмотрим ребенка удаленной вершины:

1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева.

2. Если брат текущей вершины был чёрным, то получаем три случая:

   (a) Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины и делаем его черным.

   (b) Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем поворот.

   (c) Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение.

На этом ребалансировка не заканчивается. Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Но как показывает практика при удалении совершается не более 3 вращений.

Node	
Функция	Описание
Node(string key, unsigned long long value);	Создает вершину с заданным ключом и его значением
Node(string key, unsigned long long value, bool red);	Создает вершину с заданным ключом и его значением, а также помечает вершину красной
void LeftRotate(RBTree* t);	Левый поворот дерева
void RightRotate(RBTree* t);	Правый поворот дерева
void InsertRebalance(RBTree* t);	Ребаланс дерева после его изменения (была совершена вставка)
void DeleteRebalance(RBTree* t);	Ребаланс дерева после его изменения (было совершенно удаление вершины)
static bool IsRed(const Node* node);	Проверяет является ли вершина красной

RBTree	
Функция	Описание
void insert(const string &key, unsigned long long value);	Вставка в дерево вершины с заданным ключом и значением
Node *Find(const string &key);	Находит ноду по ключу
void FindNode(const string &key) ;	Проверяет есть ли слово в словаре и выводит его
void print();	Печать дерева (функция для отладки)
void writeTree(string& path);	Запись в бинарный файл
static RBTree* readTree(string& source);	Чтение из файла
void clean(Node* t);	Удаляет словарь

```
1  class RBTree {
2  public:
3      Node *root;
4
5      void insert(const string &key, unsigned long long value);
6      Node *Find(const string &key);
7      void FindNode(const string& key) ;
8      void DeleteNode(const string &key);
9      void print();
10     void writeTree(string& path);
11     static RBTree* readTree(string& source);
12     void clean(Node* t);
13
14 private:
15     static string getKey(int& i, string& file);
16     static unsigned long long getValue(int& i, string& file);
17     static bool getIsRed(int& i, string& file);
18     void writeTree(ofstream& file, Node *node);
```

```
19      void insert(Node *newNode, Node *currentNode);
20      void print(Node *node, int height);
21      void DeleteNode(Node *node);
22      static Node *FindNext(Node *node);
23
24  };
25
26
27  class Node {
28  public:
29      string key;
30      unsigned long long value;
31      bool red;
32      Node* left;
33      Node* right;
34      Node* parent;
35
36      Node(string key, unsigned long long value);
37      Node(string key, unsigned long long value, bool red);
38
39      void LeftRotate(RBTree* t);
40      void RightRotate(RBTree* t);
41      void InsertRebalance(RBTree* t);
42      void DeleteRebalance(RBTree* t);
43
44      static bool IsRed(const Node* node);
45  };
```

3 Консоль

```
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ make
Consolidate compiler generated dependencies of target Lab1
[ 50%] Building CXX object CMakeFiles/Lab1.dir/main.cpp.o
[100%] Linking CXX executable Lab1
[100%] Built target Lab1
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ cat test1.txt
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ ./a.out
<test1.txt
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

4 Тесты производительности

Тест производительности представляет из себя следующее: сравнение эффективности реализованного RB-дерева и BST.

```
# 100000 lines
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ wc -l
test.txt
100000 test.txt
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ ./Lab2
<test.txt
RB tree time: 31828us
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA1/Lab1$ wc -l
test2.txt
1000000 test2.txt
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/DA/Lab2$ ./BST
<test.txt
BST tree time: 36357us
```

В приведённом выше тесте команды генерировались случайным образом, но уже на них видно, что RB-дерево работает быстрее. Более наглядно разницу между RB и BST можно увидеть на таких тестах, в которых стуктура элементов в дереве целенаправлено должна быть распределена неравномерно (худший случацй — BST вырождается в линейный список). На таких данных RB-дерево рабоатет за 4.153 секунды, а BST — дольше 15 секунд.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я познакомился с такой структурой данных, как RB-дерево, и научился его реализовывать. Я узнал о том, чем может быть неудобно обычное бинарное дерево поиска и зачем нужны сбалансированные деревья, как именно производится балансирвока в RB-дереве и что такое левый и правый поворот. Помимо RB-дерева, мною были изучены и другие сбалансированные структуры: AVL-дерево, B-дерево, PATRICIA.

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.))