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. А. Харьков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

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.

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.

! 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, отсутствие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Структура данных: Красно-чёрное дерево.

1 Описание

Требуется написать реализацию красно-чёрного дерева.
Красно-чёрное дерево представляет собой бинарное приближенно сбалансированное дерево поиска, но с информацией о цвете: чёрном или красном. Для того, чтобы бинарное дерево являлось красно-чёрным деревом необходимо, чтобы:

1. Каждый узел был красным или чёрным.
2. Корень дерева - чёрным узлом.
3. Дети листьев - чёрными узлами.
4. Если узел красный, то его дети чёрные.
5. Для каждого узла все простые пути от него до его листьев содержали одно и то же количество чёрных узлов.

Согласно [1] высота красно-чёрного дерева с n внутренними узлами не превышает $2lg(n+1)$. Это значит, что сложность поиска, вставки и удаления равна $O(lg(n))$.

Поиск узла в красно-чёрном дереве происходит таким же образом, как и в обычном бинарном дереве поиска. Но после вставки или удаления узла в некоторых случаях необходимо стабилизировать высоту дерева. Случаи, когда необходимо стабилизировать высоту дерева при вставке и удалении, и способы стабилизации мы рассмотрим в описании кода программы.

2 Исходный код

Вставка узла в красно-чёрное дерево происходит следующим способом:

1. С помощью сравнений ключей узлов определяется, куда вставить новый узел. Новый узел окаршивается в красный цвет.
2. Если родитель нового узла чёрный, то не нарушается никакое свойство при вставке. Если же красный, то необходимо восстановить свойства дерева:
   (a) Если дядя красный, то перекрашиваем его и родителя рассматриваемого узла в чёрный цвет, а дедушку в красный. Теперь необходимо рассмотреть восстановление дерева относительно дедушки, так как его родитель мог быть красным.
   (b) Рассмотрим случай если дядя чёрный. Если дядя правый(левый) ребёнок, а рассматриваемый узел - левый(правый) ребёнок, то необходимо сделать левый(правый) поворот относительно родителя и рассматривать восстановление дерева относительно предыдущего родителя рассматриваемого узла. Затем, при любом случае, покрасить родителя рассматриваемого узла в чёрный, а дедушку в красный и совершить левый(правый) поворот относительно дедушки.

На этом вставка в дерево закончена.

Удаление из дерева происходит следующим способом:

1. Находим необходимый узел.
   (a) Если у узла двое детей, то ищем узел с ближайшим значением по ключу, заменяем ключ и значение на ключ и значение найденного узла. Теперь удалять мы будем найденный узел.
   (b) Если у узла один ребёнок, то делаем его ребёнком родителя.
   (c) Если у узла нет детей, то просто удаляем его.
2. Если удалённый узел был чёрным, то необходимо восстановить дерево, так как нарушилась «чёрная высота»
   (a) Если брат узла красный и правый(левый) ребёнок, то красим брата в чёрный, а родителя в красный и делаем левый(правый) поворот относительно родителя. «Чёрная высота» не восстанавливается, но теперь брат чёрный.
   (b) Если брат чёрный и оба ребёнка чёрные, то красим брата в красный, а родителя в чёрный, относительно родителя «чёрная высота» восстановлена, но теперь необходимо рассмотреть восстановление относительно родителя.

(c) Если брат правый(левый) ребёнок и чёрный, его правый(левый) ребёнок чёрный, то красим его левого(правого) ребёнка в чёрный, брата в красный и делаем правый(левый) поворот относительно брата.

(d) Если брат правый(левый) ребёнок и чёрный, его правый(левый) ребёнок красный, то красим брата в цвет родителя, родителя и правого(левого) ребёнка красим в чёрный и делаем левый(правый) поворот относительно родителя. Дерево восстановлено.

Таблица функций:

tree.cpp	
void VTree::RightRotate(VNode* x)	Функция, выполняющая правый поворот дерева.
void VTree::LeftRotate(VNode* x)	Функция, выполняющая левый поворот дерева.
void VTree::InsertFixTree(VNode* node)	Функция, балансирующая дерево после вставки узла.
VInsert VTree::Insert(char key[], TUll& value)	Функция, вставляющая новый узел в дерево.
void VTree::DeleteFixTree(VNode* node)	Функция, балансирующая дерево после удаления чёрного узла.
VNode* NearestValueNode(VNode* node)	Функция, возвращающая узел, значение которого ближе всего к передаваемому узлу.
void VTree::DeleteElem(VNode* node)	Функция, удаляющая узел из дерева.
VDelete VTree::Delete(char key[])	Функция, удаляющая узел из дерева.
VSearch VTree::Search(char key[])	Функция, выполняющая поиск узла с ключом key.
void DeleteNode(VNode* node)	Функция, удаляющая узел из памяти.
void VTree::DeleteTree()	Функция, удаляющая дерево из памяти.
sltree.cpp	
void PrintNode(std::ofstream& out, VNode* node)	Функция, выводящая узел в поток.
void PrintNodes(std::ofstream& out, VNode* node, int& count)	Функция, выводящая узлы в поток.
VSave VTree::Save(char path[])	Функция, сохраняющая дерево в файл по пути path.
bool IsLowerCase(char str[])	Функция, проверяющая, что строка в нижнем регистре.

void ScanNodes(std::ifstream& in, VNode** node, VScanNode& state, int& count)	Функция, считывающая узлы из потока.
VLoad VTree::Load(char path[])	Функция, считывающая и сохраняющая дерево из файла по пути path.

Структуры и классы:

```
enum VColor{
    BLACK,
    RED,
};
enum class VInsert{
    OK,
    EXIST,
    ERR_OUT_OF_MEMORY,
};
enum class VDelete{
    OK,
    NO_SUCH_WORD,
};
struct VSearch {
    enum VState{
        OK,
        NO_SUCH_WORD,
    };
    VState state;
    unsigned long long Key;
};
enum class VLoad {
    OK,
    ERR_OPEN,
    ERR_WRONG_VARIABLE,
    ERR_OUT_OF_MEMORY,
};
enum class VScanNode{
    OK,
    ERR_WRONG_VARIABLE,
    ERR_OUT_OF_MEMORY,
};
enum class VSave {
    OK,
    ERR_OPEN,
};
class VNode {
public:
    VNode* Left;
    VNode* Right;
```

```
41      VNode* Par;
42      char Key[STR_SIZE];
43      unsigned long long Value;
44      VColor Color;
45      VNode(): Key("") {
46          Left = nullptr;
47          Right = nullptr;
48          Par = nullptr;
49          Color = RED;
50          Value = 0;
51      }
52  };
53  class VTree {
54  private:
55      // tree.cpp
56      void LeftRotate(VNode* node);
57      void RightRotate(VNode* node);
58      void InsertFixTree(VNode* node);
59      void DeleteFixTree(VNode* node);
60      void DeleteElem(VNode* node);
61      void DeleteTree();
62  public:
63      VNode* root;
64      VTree() {
65          root = nullptr;
66      };
67      ~VTree(){
68          this->DeleteTree();
69      }
70      // tree.cpp
71      VInsert Insert(char key[], unsigned long long& val);
72      VDelete Delete(char key[]);
73      VSearch Search(char key[]);
74      // sltree.cpp
75      VSave Save(char path[]);
76      VLoad Load(char path[]);
77  };
```

3 Консоль

```
p.kharkov$ make
g++ -std=c++14 -pedantic -Wall -O2 -c tree.cpp -o tree.o
g++ -std=c++14 -pedantic -Wall -O2 -c sltree.cpp -o solution/sltree.o
g++ -std=c++14 -pedantic -Wall -O2 main.cpp tree.o sltree.o -o laba2
p.kharkov$ cat tests/main
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
a
p.kharkov$ ./laba2 < tests/main
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: вставка, удаление и поиск строк с помощью контейнера «map» стандартной библиотеки сравнивается с красно-чёрным деревом. Тест состоит из 10^5 запросов:

```
p.kharkov$ make benchmark
g++ -std=c++14 -pedantic -Wall -Wextra -c tree.cpp -o tree.o
g++ -std=c++14 -pedantic -Wall -Wextra -c sltree.cpp -o solution/sltree.o
g++ -std=c++14 -pedantic -Wall -Wextra benchmark.cpp tree.o sltree.o -o benchmark
p.kharkov$ ./benchmark < tests/01.t
std::map ms= 13716
rb ms= 10528
```

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я лучше разобрался в бинарных деревьях поиска, в особенности красно-чёрные деревья. Их знать полезно, так как они иногда используются в стандартной библиотеки C++, к примеру, в контейнере map. Знание реализаций разных деревьев поможет, когда необходимо найти оптимальное решение для задачи. К примеру, если мы знаем, какие данные будут подаваться на вход, то даже поиск в обычном бинарном дереве может работать в среднем за $O(lg(n))$, а если нет, то AVL или RB, так как поиск в худшем случае работает за $O(lg(n))$.
По началу алгоритм стабилизации «чёрной высоты» в красно-чёрном дереве был не очень понятен, но оказалось, что он довольно простой. Также, к сожалению, так как я писал программу на Windows, а не на *Nix, я не смог реализовать проверку на отсутствие прав записи в файл, как требовалось по заданию.

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.))