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: Е. А. Айрапетова
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2021

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.

Как сказано в [1]: «красно-чёрное дерево является сбалансированным бинарным деревом поиска, удовлетворяющим условиям:».

- Каждый узел является красным или чёрным;
- Корень дерева является чёрным;
- Если узел красный, то оба его дочерних узла — чёрные;
- Каждый лист является чёрным;
- Соблюдается правило чёрной высоты (для любого узла все простые пути от него до листьев имеют одно и то же количество чёрных вершин).

Поиск происходит также, как и в обычном бинарном дереве.

После вставки и удаления в некоторых случаях необходимо сбалансировать дерево так, чтобы продолжало соблюдаться правило чёрной высоты. В случае вставки балансировка нужна в двух сучаях:

- Если дядя красный, нужно покрасить дядю и отца в чёрный цвет, а дедушку в красный. Баланс не изменился, однако нужно востановить баланс относительно дедушки в случае, если его родитель был красным;
- Если дядя чёрный, нужно проверить, что вставляемый узел — левый ребёнок, если дядя — правый или вставляемый узел — правый ребёнок, если дядя левый. В противном случае нужно сделать левый или правый поворот. Также, если отец является красным, его нужно покрасить в черный, а дедушку в красный и совершить поворот.

При удалении узла из дерева, нужно, в первую очередь, проверить, сколько детей было у удаляемого узла. Если у него не было детей, он просто удаляется из дерева. Если у него был один ребёнок, соединяем ребёнка с родителем. Если же у этого узла два ребёнка, нужно заменить ключ удаляемого узла на ключ следующего или предыдущего, а затем выполнить удаление относительно следующего или предыдущего узла.

Балансировка при удалении нужна в случае, если удаляемый элемент был чёрным.

- Если брат и оба ребёнка удаляемого узла — чёрные, красим брата в красный, а отца в чёрный. Если отец изначально был чёрным, производим балансировку относительно него;

- Если брат удаляемой вершины красный, совершаем поворот между отцом и братом. Брата красим в чёрный, а отца в красный. Чёрная высота не восстановилась, но, так как брат чёрный, далее следует предыдущий алгоритм для такого случая;

- Если брат удаляемого узла — чёрный и является правым (левым) ребёнком, левый (правый) сын брата — красный, а правый (левый) — чёрный, то красим брата в красный, а красного сына в чёрный;

- Если брат является правым (левым) ребенком, и правый (левый) сын брата красный, красим правого сына брата в черный и совершаем левый поворот относительно отца.

Согласно [1] высота красно-чёрного дерева с n узлами меньше или равна $2lg(n+1)$. Следовательно, сложность поиска, вставки и удаления равна $O(lg(n))$.

2 Исходный код

Структура узлов:

```
#include <iostream>

const char BLACK = 1;
const char RED = 2;

class TRBNode {
public:
    char* Key;
    unsigned long long Value;
    char Colour = BLACK;

    TRBNode* Left = nullptr;
    TRBNode* Right = nullptr;
    TRBNode* Parent = nullptr;

    TRBNode();
    TRBNode(char* k, unsigned long long v);
    ~TRBNode();

    TRBNode* Sibling();
    bool Isleft();
    bool Islist();
};
```

node.h	
TRBNode()	Конструктор по умолчанию
TRBNode(char* k, unsigned long long v)	Конструктор от двух аргументов: ключ и значение
TRBNode()	Деструктор
TRBNode* Sibling()	Функция, возвращающая указатель на брата
bool Isleft()	Функция, проверяющая, является ли данный узел левым ребёнком
bool Islist()	Функция, проверяющая, что у узла один или ноль детей
int strequal (const char* lhs, const char* rhs)	Функция, проверяющая, что два узла эквивалентны
void Swap(TRBNode* lhs, TRBNode* rhs)	Обмен местами двух узлов
TRBNode* LeftRotate(TRBNode* centre)	Левый поворот относительно centre

TRBNode* RightRotate(TRBNode* centre)	Правый поворот относительно centre
TRBNode* RMreballance(TRBNode* start_element, TRBNode* root)	Балансировка дерева при удалении элемента
TRBNode* ISreballance(TRBNode* start_element,TRBNode* root)	Балансировка дерева при вставке элемента
int BlackHeigth(TRBNode* start)	Функция, возвращающая значение чёрной высоты

Структура дерева:

```
#include "node.h"

class TRBTree {
private:
    TRBNode* root = nullptr;
public:
    TRBTree ();
    ~TRBTree ();
    TRBNode* Root();
    int Insert(char* key, unsigned long long& value);
    int Remove(char* inpkey);
    void Print();
    TRBNode* Search(char* inpkey);
    void Destroy();
    int SaveToDisk(const std::string& path);
    int LoadFromDisk(const std::string& path);
};
```

tree.h	
TRBTree()	Конструктор
~TRBTree()	Деструктор
TRBNode* Root()	Функция, возвращающая корень дерева
int Insert(char* key, unsigned long long& value)	Вставка элемента
int Remove(char* inpkey)	Удаление элемента
void Print()	Печать дерева
TRBNode* Search(char* inpkey)	Бинарный поиск
void Destroy()	Удаление дерева
int SaveToDisk(const std::string& path)	Сохранение словаря в бинарном компактном представлении в файл
int LoadFromDisk(const std::string& path)	Загрузка словаря из файла

3 Консоль

```
jane@Evgenia:/mnt/c/Files/ДА/ЛР2/solution$ make
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c main.cpp -o
main.o
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable main.o -o solution
jane@Evgenia:/mnt/c/Files/ДА/ЛР2/solution$ cat file.txt
+ word0 40
+ word1 1
+ word2 42
+ word3 50
+ word4 56
-word0 40
+ word1 1
jane@Evgenia:/mnt/c/Files/ДА/ЛР2/solution$ ./solution <file.txt
OK
OK
OK
OK
OK
OK
NoSuchWord
Exist
```

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:

```
jane@Evgenia:/mnt/c/Files/ДА/ЛР2/solution$ make benchmark
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable -c benchmark.cpp
-o benchmark.o
g++  benchmark.o -o benchmark
jane@Evgenia:/mnt/c/Files/ДА/ЛР2/solution$ ./benchmark <file.txt
std::map ms = 13716
rb ms = 10528
```

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я на практике ознакомилась с написанием бинарных деревьев, а конекретно, красно-чёрных. Красно-чёрные деревья довольно часто используются в программировании для оптимального решения задач.

Балансировка RB-дерева вызвала некоторые трудности, так как довольно сложно ничего не упустить, например, не забыть вызвать функцию балансировки относительно отца, если узел перекрашивается в красный и т.п. Но, в конце концов, я разобралась и это не заняло много времени.

Также возникли трудности с тестированием программы на запись дерева в файл, а именно — в проверке на отсутствие прав записи, так как я писала программу на Windows.

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.))

[2] *Красно-чёрные деревья — Хабр.*
URL: `https://habr.com/ru/post/330644/ Красно-чёрные_деревья` (дата обращения: 12.06.2021).