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: А. С. Федоров
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б
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 $2^{64} - 1$.

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

Вариант используемой структуры данных: B-дерево.

1 Описание

Требуется написать реализацию словарая на основе B-дерева. Должны быть реализованы операции добавления в словарь, удаления из словря, сохранение на диск и загрузка словаря с диска.

Как сказано в [1]: «B-деревья представляют собой сбалансированные деревья поиска, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом).».

Уззлы B-дерева могут иметь несколько детей, так что степень ветвления может быть очень большой (обычно она поределяется характеристиками используемых дисков). Так как эта струткута является сбалансированным делевом поиска, его высота растет как $O(\log n)$, в виду большой степени ветвления основние логарифма имеет большое значение, в сравнении с красно-черным деревом. Это делает B-дерево удобным для размещение на вторичной памяти, так как количество обращений к диску существенно снижено.

Свойства узла B-дерева:

1. Ключи узла хранятся в неубывающем порядке.
2. Количество ключей в узле не меньше $t-1$ и не больше $2*t-1$. Если узел является коренм, количество ключей может быть меньше $t-1$.
3. Узел содержит $n+1$, где n - количество ключей в узле. Потомки узла подчинаются следующему порядку:
   (a) Первый потомок имеет ключ в интеравле $(-\infty, K_1)$.
   (b) Начиная со втрого и до n-го потомка ключи располагаются в интеравле (K_{i-1}, K_i), где i пробегает всех потомков узла от второго и до предпоследнего.
   (c) Ключ последнего потомка находится в интерале (K_n, ∞).

Поиск в B-дереве схож с поиском в бинарном деревне поиска. Однако выбор осуществляется из множетсва путей, а не из двух. Если ключ содержиться в узле, то он найден и поиск завершается. Если нет, спускаемся в соответствущего потомка. Выбирать потомка можно любым алгоритмом, но для быстроты поиска, использую бинарный поиск, который работает за $O(\log n)$. Если принять сложность поиска потомка за константу, операция обладает сложностью $O(h)$, где h - высота дерева.

Новые элементы можно добавлять только в листья. При вставке, находим нужный лист и добавляем его, сохраняя порядок. Если после вставки в нем оказывается $2*t-1$ элементов, то разбиваем узел по середине, и перемещаем средний элемент в предка.

Тогда для среднего элемента левая и правая части станут детьми. Если же узел-предок после перемещения среднего узла стал иметь $2*t-1$ элементов, то разбиение выполняется и для него. Так будет продолжаться пока предок не окажется неперепольненым или же предок не окажется коренм. Если предок оказался коренм, то после разбиения средний элемент образует новый корень и высота дерева увеличивается на единицу. В худшем случае разбиение будет происходить до самого корня, что есть прохожение всей высоты высоты 2 раза (поиск места для вставки и разбиение до корня). Сложность снова $O(h)$.

Для удалеиня из листа выполнятется поиск нужного узла и проверяется количество элементов в нем. Если в листе больше $t-1$ элементов, то просто удаляем элемент. Иначе, если у левого брата есть больше $t-1$ элементов, то берем самый правый элемент у него, добавляем в текущий узел и удаляем нужный нам элемент. Если у левго брата недостаточно элементов, то пытаемся провернуть то же самое с правым братом. Если и у правого брата недостаточно элментов, то сливаем текущий узел с правм или левым братом, засовывая между ними элемент-разделитель из предка. Получаем узел размером $2*t-1$ и затем удаляем нужный элемент. Сложность аналогично $O(h)$.

Удаление не из листа выполняется поиском нужного для удаления узла и элемента в нем. Затем выполняется замена элемента для удаления на наибольшего элемента из левого поддерева или же на наименьшего из правого поддерева. Что наибольшей элемн, что наименьший элемент поддерева будет гарантированно находиться в листе. Поэтому элемент для замены удаляется как элемент из листа.

Высота B-дерева с минимальным количеством элементов считается как $1+\sum_{i=1}^{h} 2*(t-1)*t^{i-1}$. Раскрываем сумму и получаем $2*t^h-1$. Значит количество элементов дерева удовлетворяет следующему неравенству: $n \geq 2*t^h-1$. Соответственно $h \leq \log_t((n+1)/2)$ и $h \approx \log_t n$. Значит, что все операции над деревом выполняются со сложностью $O(\log n)$.

2 Исходный код

Для хранения элементов в узле и последующим их удобным добавлением и удалением реализую класс для динамического массива TVector. Помимо обчного добавниения и удаления, добавлю еще вставку и удаение по индексу и вствку в упорядоченный массив.

```
#ifndef LAB1_TVector_H
#define LAB1_TVector_H
#include <iostream>
template <class T>
class TVector{
private:
    unsigned long long TVectorSize;
    unsigned long long TVectorCapacity;
    T* Data;
public:
    TVector();
    unsigned long long Size();
    void PushBack(const T elem);
    void PopBack();
    void Insert(const unsigned long long &pos,const T &elem);
    void OrdinaryInsert (const T &elem);
    void OrdinaryErase (const T &elem);
    void Erase(const unsigned long long &pos);
    T& operator[] (long long iterator);
    ~TVector();
};
```

TVector.h	
TVector()	Конструктр пустого вектора
unsigned long long Size()	Размер ветора
void PushBack(const T elem)	Добавление элемента в конец
void PopBack()	Удаление с конца
void Insert(const unsigned long long &pos,const T &elem)	Вставка элемента по индексу
void OrdinaryInsert (const T &elem)	Вставка в отсортированный вектор с сохранением порядка
void Erase(const unsigned long long &pos)	Удаление по индексу
T& operator[] (long long iterator)	Получить элемент по индексу
TVector()	Деструктор с удалением всего содержимого

Узел B-дерева реализован в виде струкуры с ветором данных и ветором потомков.

```
struct Node{
        TVector<T> Data;
        TVector<Node<T>*> Childs;
        Node<T>* Parent = nullptr;
        ~Node(){
            for (int i = 0; i < Childs.Size(); ++i){
                delete Childs[i];
            }
        };
    };
```

Для хранения пар ключ-значение напишу простую стрктуру Item

```
struct Item{
    char Key[KEY_SIZE];
    unsigned long long Value;
    Item(){
        for (int i = 0; i < KEY_SIZE; ++i){
            Key[i] = 0;
        }
        Value = 0;
    }
    Item& operator= (const Item it){
        for (unsigned long long i = 0; i < KEY_SIZE; ++i){
            Key[i] = it.Key[i];
        }
        Value = it.Value;
        return *this;
    }
};
```

Все реализации опреаций над деревом и реализация узла вынесены в отдельное пространство имен.

```
namespace BTree {
    const unsigned long long t = 8;
    const unsigned int KEY_SIZE = 256;
    template <class T>
    struct Node{
        ...
    };
    template <class T>
    T* SearchInTree (Node<T>* treeNode, T it);
    template <class T>
    void Split (Node<T>* &treeNode);
    template <class T>
    bool AddToTree (Node<T>* &Root, T it);
    template <class T>
    bool DeleteFromTree (Node<T>* &Root, T it);
    template <class T>
    void SaveTreeInFile(Node<T>* treeNode, FILE* saveFile);
```

```
18      template <class T>
19      Node<T>* LoadTreeFromFile(Node<T>* parent, FILE* loadFile);
20  }
```

BTree.h	
T* SearchInTree (Node<T>* treeNode, T it)	Поиск в B-дереве
unsigned long long Size()	Размер ветора
void Split (Node<T>* &treeNode)	Разбиение узла
bool AddToTree (Node<T>* &Root, T it)	Добавление нового элемента
bool DeleteFromTree (Node<T>* &Root, T it)	Удаление элемента
void SaveTreeInFile(Node<T>* treeNode, FILE* saveFile)	Сохраниение B-дерева в файл
Node<T>* LoadTreeFromFile(Node<T>* parent, FILE* loadFile)	Загрузка B-дерева из файла

Добавление реализовано с преждевременным разбиением, то есть в дереве никогда нет узлов размера $2 * t - 1$. Очевидно, что это не валияет на суть алгоритма.

Сохраниение на файл осуществляется с помощью трех видов скобок, которые берут на себя роль терминирующих символов. Круглые - начало и конец элемента, квадратные - начало и конец узла и фигурные - начало и конец поддерева. Соответсвенно, чтение анализирутет эти скобки и, исходя из их расположения, воссоздает дерево. Проход по файлу происходит только один раз.

3 Консоль

```
protaxy@protaxY:~/DA/da_lab2$ make
g++ -std=c++17 -o solution main.cpp
protaxy@protaxY:~/DA/da_lab2$ cat test.txt
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
a
! Save Tree.bt
+ bb 123
+ cc 456
Bb
-cc
cc
-aa
aa
! Load Tree.bt
aa
protaxy@protaxY:~/DA/da_lab2$ ./solution <test.txt
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
OK
OK
OK
OK: 123
OK
NoSuchWord
OK
NoSuchWord
OK
OK: 18446744073709551615
```

4 Тест производительности

Для сравнения использую стандартный контейнер std::map, который реализован через красно-черное дерево. Вставка, удаление и поиск в B-дереве будут сравниваться сответствующими методами контейнера: std::insert, std::erase и std::find. Тип ключа и значения аналогичен: char[256] и unsigned long long. Операции сохраниения и згрузки на файл в сравнение не беру, так как такие методы не реализованы для конейнера std::map. Скорость проверю на тестах в 10^2, 10^3, 10^4, 10^5, 10^6 запросов.

```
protaxy@protaxY:~/DA/da_lab2/benchmark$ g++ benchmark.cpp
protaxy@protaxY:~/DA/da_lab2/benchmark$ ./a.out <test100.txt
std::map: 0.497 ms
protaxy@protaxY:~/DA/da_lab2/benchmark$ ./a.out <test1000.txt
std::map: 4.885 ms
protaxy@protaxY:~/DA/da_lab2/benchmark$ ./a.out <test10000.txt
std::map: 17.495 ms
protaxy@protaxY:~/DA/da_lab2/benchmark$ ./a.out <test100000.txt
std::map: 164.620 ms
protaxy@protaxY:~/DA/da_lab2/benchmark$ ./a.out <test1000000.txt
std::map: 1141.508 ms

protaxy@protaxY:~/DA/da_lab2$ make
g++ -std=c++17 -o solution main.cpp
protaxy@protaxY:~/DA/da_lab2$ ./solution <test100.txt
B-tree: 1.896 ms
protaxy@protaxY:~/DA/da_lab2$ ./solution <test1000.txt
B-tree: 18.654 ms
protaxy@protaxY:~/DA/da_lab2$ ./solution <test10000.txt
B-tree: 65.258 ms
protaxy@protaxY:~/DA/da_lab2$ ./solution <test100000.txt
B-tree: 488.267 ms
protaxy@protaxY:~/DA/da_lab2$ ./solution <test1000000.txt
B-tree: 3596.755 ms
```

Из тестов видно, что время растет примерно одинакаво, но с разными коэффициентами. B-дерево требует больше времени, так как операции разбиения и слияния узлов требуют поэлементного копирования, также бинарный поиск занимает время, чего не происходит в std::map. В тестировании использовалось B-дерево степени 8 (то есть t = 8).

5 Выводы

В ходе выполненения лабораторной работы я изучил структуру данных B-дерево. Узнал сложностную оценку операций над ним и реализовал его в виде прораммы на языке C++. Для понимания тонкостей реализации B-дерева использовал онлайн визуализа- тор. Это значительно упростило написане кода, так как можно поэтапно рассомтреть последовательность выполняемых оперций. Сохранение на файл изначально думал сделать, указывая размеры узлов, но впоследствии понял, что реализовывать че- рез терминирующие элементы граздо проще. Однако, потребуется лишняя память на хранения разделяющих символов, но я не считаю это критичным. В ходе отладки изучил инструменты Valgrind и Coverage. А их интеграция в используемой мной среде разработки сделало отладку еще проще.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2013. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — Глава 18. (ISBN 5-8459-0857-4 (рус.))

[2] *B-дерево*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=B-дерево` (дата обращения: 12.11.2020).

[3] *B-tree*
URL: `https://habr.com/ru/post/114154/` (дата обращения: 12.11.2020).