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: И. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 05.12.2020
Оценка: хорошо
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 (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.

Используемая структура данных: B-дерево.

1 Описание

B-дерево - это сильноветвящееся дерево поиска. Согласно [1], оно обладает следующими свойствами:

- Каждый узел дерева содержит следующие атрибуты:
  - массив из n ключей, упорядоченных по возрастанию
  - массив из $n + 1$ указателей на дочерние узлы
- Ключи узла дерева разделяют поддиапазоны ключей, хранящихся в поддеревьях: в i-ом дочернем узле хранятся элементы, ключи которых больше $(i - 1)$-ого, но меньше i-ого ключа родительской вершины.
- Все листья расположены на одинаковой глубине.
- Каждый узел дерева должен содержать как минимум $t - 1$ ключ (исключение: корень должен содержать минимум 1 ключ). Количество ключей не должно превышать $2 * t - 1$, где $t \geq 2$ - степень дерева.

Алгоритм поиска по B-дереву напоминает поиск в бинарном дереве. Если искомая вершина отсутствует в текущей вершине, то необходимо найти поддиапазон, которому она принадлежит, и продолжить поиск в соответствующем дочернем узле.

Вставка новых элементов осуществляется только в листовые узлы B-дерева. Место для вставки определяется алгоритмом поиска. Если в найденной вершине меньше $2 * t - 1$ элементов, то необходимо просто вставить новый ключ в массив так, чтобы сохранилась его упорядоченность. В противном случае необходимо выполнить разбиение вершины: средний ключ перемещается в родительскую вершину, а первые и последние $t - 1$ ключей становятся его левым и правым ребенком соответственно.

При удалении элемента из B-дерева могут возникнуть следующие ситуации:

- Удаляемый элемент находится в листовом узле, размер которого больше $t - 1$. В этом случае необходимо просто удалить нужный ключ из листа.
- Удаляемый элемент находится во внутреннем узле. Если размер левого поддерева для удаляемого элемента больше $t - 1$, то необходимо заменить этот элемент на максимальный элемент левого поддерева (расположен в листе) и удалить соответствующий элемент из листа. Если размер правого поддерева больше $t - 1$, то необходимо проделать аналогичную операцию с минимальным элементом из правого поддерева. В противном случае необходимо выполнить слияние левого поддерева, удаляемого элемента и правого поддерева в одну вершину и продолжить удаление из новой вершины.

- Если текущая вершина не содержит удаляемого элемента, то необходимо найти дочерний узел, в котором должна располагаться вершина. Если размер найденного узла равен $t-1$, то необходимо переместить один ключ из его родителя в найденный узел, а крайний ключ из брата - в родителя. Если же размер обоих братьев этого узла тоже равен $t-1$, то необходимо выполнить слияние с одним из двух братьев и продолжить удаление.

Для сериализации дерева в файл записывались структуры следующего вида: лист/не лист + количество элементов в вершине + [размер ключа + ключ + значение] для каждого элемента + аналогичные структуры для всех потомков (если текущая вершина не является листом).

При десериализации данные из файла считывались согласно схеме выше и копировались в B-дерево.

2 Исходный код

Для хранения пары «ключ-значение» создадим структуру $TPair$, содержащую поле key, представленное массивом из 257 char'ов, и $value$, представленное типом unsigned long long.

Перегрузим операторы сравнения для структуры $TPair$. Пара «ключ-значение» меньше, если её ключ лексикографически меньше.

Создадим структуру $TNode$ для хранения узла дерева. Структура состоит из массива ключей $TPair$, массива дочерних узлов $TNode*$, целочисленной переменной $keys\_num$, показывающей количество ключей в вершине в текущий момент, и булевой переменной is_leaf, показывающей, является ли данная вершина листом.

```
struct TPair {
    char key[257];
    unsigned long long value;
};

class TNode {
public:
    bool is_leaf = true;
    int keys_num = 0;
    TPair keys[2 * DEGREE];
    TNode* children[2 * DEGREE + 1];
    TNode();
};
```

Для хранения самого дерева создадим класс $TBTree$, который включает в себя указатель на корень, публичные методы для поиска, вставки, удаления элементов, сериализации и десериализации, а также приватный метод для рекурсивного удаления дерева.

```
class TBTree {
public:
    TNode *root;
    TBTree();
    ~TBTree();
    void Search(char *str) const;
    void Insert(TPair &KV);
    void Remove(char *str);
    void Serialize(FILE* file);
    void Deserialize(FILE* file);

private:
    void Delete();
};
```

Каждый метод вызывает соответствующую рекурсивную функцию. Эти и вспомогательные функции будут описаны в таблице ниже.

TBTree.h	
void SearchNode(TNode *node, char* str, TNode *&res, int &pos)	Поиск ключа в дереве. В res помещается ссылка на вершину дерева с искомым ключом, в pos - позиция искомого элемента в вершине res. Если искомого элемента в дереве нет, то в res помещается $nullptr$
void SplitChild(TNode *parent, int pos)	Разбиение поддерева дерева $parent$ на позиции pos
void InsertNode(TNode *node, TPair &KV)	Вставка пары KV в вершину $node$
int SearchInNode(TNode *node, char* str)	Поиск ключа str в текущей вершине $node$. Реализована при помощи алгоритма бинарного поиска. Возвращает позицию элемента в дереве. Если искомого ключа нет, то возвращается отрицательное число n, означающее, что ключ нужно искать в $-n-1$ поддереве
void RemoveNode(TNode *node, char* str)	Удаление ключа str из дерева $node$
void RemoveFromNode(TNode *node, int pos)	Удаление ключа на позиции pos в листе $node$ при помощи сдвига
void MergeNodes(TNode *parent, int pos)	Слияние левого поддерева, $parent[pos]$ и правого поддерева. Результат помещается в $parent[pos]$
void Rebalance(TNode *node, int &pos)	Если при удалении ключа встретилась вершина с минимальным размером, то эта функция выполняет преобразования для увеличения размера (слияние или перемещение вершины из брата)
bool NodeToFile(TNode *node, FILE *file)	Сериализация дерева $node$ в файл. Возвращает 1 в случае успешного завершения
bool FileToTree(TNode *node, FILE *file)	Десериализация дерева $node$ из файла. Возвращает 1 в случае успешного завершения
void DeleteTree(TNode *node)	Рекурсивное удаление всех вершин дерева

3 Консоль

```
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O3 -lm main.cpp -o
solution
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./solution
+ a 1
OK
+ b 2
OK
+ c 3
OK
+ d 4
OK
a
OK: 1
b
OK: 2
c
OK: 3
d
OK: 4
! Save test
OK
-a
OK
-b
OK
-c
OK
-d
OK
a
NoSuchWord
b
NoSuchWord
c
NoSuchWord
d
NoSuchWord
+ e 5
```

```
OK
! Load test
OK
a
OK: 1
b
OK: 2
c
OK: 3
d
OK: 4
e
NoSuchWord
```

4 Тест производительности

Для анализа производительности моей программы напишем словарь на основе контейнера std::map стандартной библиотеки и сравним время работы.

Для сравнения производительности подготовим тесты следующего вида: в словарь добавляется n элементов, запрашивается поиск каждого элемента, удаляются все элементы и снова запрашивается их поиск. Протестируем программы при $n = 500, n = 10000, n = 100000$.

```
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O3 -lm main.cpp -o
btree
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ g++ -pedantic -Wall -std=c++11 -Werror
-Wno-sign-compare -O3 -lm -o stdmap stdmap.cpp
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./btree <test500.txt >res1.txt
Time: 0.0038201 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./stdmap <test500.txt >res2.txt
Time: 0.0030815 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ diff res1.txt res2.txt
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./btree <test10k.txt >res1.txt
Time: 0.0838783 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./stdmap <test10k.txt >res2.txt
Time: 0.0484995 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ diff res1.txt res2.txt
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./btree <test100k.txt >res1.txt
Time: 0.948157 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ ./stdmap <test100k.txt >res2.txt
Time: 0.591212 seconds
mosik@LAPTOP-69S778GL:~/da_lab2_v2$ diff res1.txt res2.txt
```

Заметим, что std::map, реализованный на основе красно-чёрного дерева, работает примерно в $1,5-2$ раза быстрее B-дерева. Но стоит учитывать, что сложность красно-чёрного дерева - $O(logn)$, а B-дерева - $O(log_t n * logt)$

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился работать с B-деревом - сильноветвящимся сбалансированным деревом поиска. При помощи многочисленных тестов своей программы я убедился, что B-дерево - это подходящая структура данных для хранения большого количества информации. Она позволяет искать, вставлять и удалять элементы за время $O(log_t n * logt)$. Тесты показали, что программа справляется с 400 тысячами операций всего лишь за секунду.

Однако, сравнив время работы своего B-дерева с красно-чёрным деревом из стандартной библиотеки (std::map), я выяснил, что дерево из стандартной библиотеки работает быстрее. Но стандартная реализация std::map не имеет встроенного функционала для сериализации словаря в компактном представлении.

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.))