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: Р. С. Лисин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-20
Дата:
Оценка:
Подпись:

Москва, 2022

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

Вариант структуры данных: `B-дерево.`

1 Описание

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

1. Каждый узел дерева содержит следующие атрибуты:
   (a) массив из n ключей, упорядоченных по возрастанию
   (b) массив из $n+1$ указателей на дочерние узлы
2. 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.
3. Все листья расположены на одинаковой глубине.
4. 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 $t >= 2$ - 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:

1. Удаляемый элемент находится в листовом узле, размер которого больше $t-1$. В этом случае необходимо просто удалить нужный ключ из листа.
2. 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.

3. 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 пар «ключ-значение» будем использовать структуру $TPair$, так как это удобно. Ключ будем хранить в статическом массиве на 257 элементов типа char. А значения будем хранить как uint_64. 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. Переменная $DEGREE$ играет роль t в B-дереве.

```
const int DEGREE = 5;

struct TPair {
  char key[257];
  uint64_t value;
};

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

  TNode() {
    for (int i = 0; i < 2 * DEGREE + 1; ++i) {
      children[i] = nullptr;
    }
  }
};
```

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. Эти методы будут использовать другие функции, чтобы код удобнее читался. Все функции реализованы в соответсвии с описанием.

```

void SearchNode(TNode* node, char* str, TNode*& res, int& pos); // if key not in tree,
     res will be nullptr
void SplitChild(TNode* parent, int pos); // splitting parent at pos
void InsertNode(TNode* node, TPair& KV);
int SearchInNode(TNode* node, char* str); // searching str. pos is index of str or
    index of a child with str
void RemoveNode(TNode* node, char* str);
```

```
7  void RemoveFromNode(TNode* node, int pos); // remove key[pos] in current node by
       making shifts
8  void MergeNodes(TNode* parent, int pos); // merging left child, parent[pos] and right
       child to left child
9  void Rebalance(TNode* node, int& pos); // rebalancing tree if a node has critical size
10 void NodeToFile(TNode* node, ostream& file);
11 void FileToTree(TNode* node, istream& file);
12 void DeleteTree(TNode* node);
13
14
15 class TBTree {
16   public:
17   TNode* root;
18
19   TBTree() {
20     root = new TNode;
21   }
22
23   ~TBTree() {
24     Delete();
25   }
26
27   void Search(char* str) const {
28     TNode* res;
29     int pos;
30     SearchNode(root, str, res, pos);
31     if (res == nullptr) {
32       cout << "NoSuchWord\n";
33     }
34     else {
35       cout << "OK: " << res->keys[pos].value << "\n";
36     }
37   }
38
39   void Insert(TPair& KV) {
40     TNode* search_res;
41     int search_pos;
42     SearchNode(root, KV.key, search_res, search_pos);
43     if (search_res != nullptr) {
44       cout << "Exist\n";
45       return;
46     }
47     // if a root is full, we'll make a split
48     if (root->keys_num == 2 * DEGREE - 1) {
49       TNode* new_root = new TNode;
50       new_root->is_leaf = false;
51       new_root->children[0] = root;
52       root = new_root;
53       SplitChild(root, 0);
```

```
54      }
55      InsertNode(root, KV);
56      cout << "OK\n";
57    }
58
59    void Remove(char* str) {
60      TNode* res;
61      int pos;
62      SearchNode(root, str, res, pos);
63      if (res == nullptr) {
64        cout << "NoSuchWord\n";
65      }
66      else {
67        RemoveNode(root, str);
68        cout << "OK\n";
69        // if our root is empty, its child becomes a new one
70        if (root->keys_num == 0 && !root->is_leaf) {
71          TNode* tmp = root->children[0];
72          delete root;
73          root = tmp;
74        }
75      }
76    }
77
78    void Serialize(ostream& out) {
79      NodeToFile(root, out);
80      cout << "OK\n";
81    }
82
83    void Deserialize(istream& in) {
84      FileToTree(root, in);
85      cout << "OK\n";
86    }
87
88    private:
89    void Delete() {
90      DeleteTree(root);
91    }
92  };
93  }
```

3 Консоль

```
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O3 -lm main.cpp -o
solution
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./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
```

```
6
OK
! Load test
OK
a
OK: 1
b
OK: 2
c
OK: 3
d
OK: 4
e
NoSuchWord
```

4 Тест производительности

Для анализа производительности будем использовать std::map стандартной библиотеки и сравним время работы с моей программой. 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.

```
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O3 -lm main.cpp -o
btree
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ g++ -pedantic -Wall -std=c++11 -Werror
-Wno-sign-compare -O3 -lm -o stdmap stdmap.cpp
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./btree <test500.txt >res1.txt
Time: 0.0038201 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./stdmap <test500.txt >res2.txt
Time: 0.0030815 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ diff res1.txt res2.txt
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./btree <test10k.txt >res1.txt
Time: 0.0838783 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./stdmap <test10k.txt >res2.txt
Time: 0.0484995 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ diff res1.txt res2.txt
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./btree <test100k.txt >res1.txt
Time: 0.948157 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ ./stdmap <test100k.txt >res2.txt
Time: 0.591212 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab2$ 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 реализовывать структуру данных B-дерево, вспомнил алгоритмы на бинарном дереве поиска и освоил работу с файлами на C++. Это поможет мне в ситуации, когда нужно будет использовать бинарное дерево поиска и хранить данные на внешнем носителе или на диске. Но если будет важна скорость работы, я буду использовать std::map в аналогичной ситуации.

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] *Информация о std::shared_ptr*
URL: `https://en.cppreference.com/w/cpp/memory/shared_ptr` (дата обращения: 24.04.2022).