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: М. А. Инютин
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19

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 словаря (ассоциативного массива) с помощью B-дерева. Помимо простого добавления, поиска и удаления требуется реализовать запись на диск и чтения с диска всего словаря.

Согласно [3], B-деревом называется структура данных, сбалансированное, сильно ветвистое дерево поиска.

Сбалансированность означает, что длина любых двух путей от корня до листьев различаются не более, чем на единицу.

Ветвистость дерева — свойство каждого узла ссылаться на большое число узлом-потомков.

B-дерево обладают следующими свойствами:

1. В каждом узле ключи упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2*t-1$ ключей. Любой другой узел содержит от $t-1$ до $2*t-1$. Здесь t - параметр дерева, не меньший 2.

2. Любой узел, содержащий n элементов вида $K_1, ..., K_n$, имеет $n+1$ потомков. При этом

   (a) Первый потомок содержит числа из интервала $(-\infty, K_1)$

   (b) Для $2 \leqslant i \leqslant n$, i потомок содержит числа из интервала (K_{i-1}, K_i)

   (c) $n+1$ потомок содержит числа из интервала $(K_n, +\infty)$

3. Глубина всех листьев одинакова.

Свойство 3 означает, что дерево идеально сбалансированно.

Поиск в B-дереве похож на поиск в обычном бинарном дереве поиска. Если ключ содержится в узле, то он найден. Иначе определяем интервал и идём к соответствующему потомку, пока не найдём элемент.

При вставке мы выполняем поиск элемента, который приведёт нас к одному из листов дерева. В этот лист требуется вставить элемент и затем, возвращаясь к корню, проверять свойство 1. Если в узле $2*t-1$ ключей, нужно разделить его на два узла размера $t-1$, а элемент посередине добавить в предка и проверить для него это же условие.

При удалении мы снова будем использовать поиск. Удаление элемента из листа также требует проверки условия 1. Если в узле стало меньше, чем $t-1$ ключей, то требуется взять элемент из левого или правого брата, сохраняя свойство 2. Если в братьях тоже по $t-1$ ключей, следует сделать объединение с элементом из предка в узел, содержащий $2*t-1$ ключей. Проверим теперь свойство 1 для предка. При удаление

элемента не из листа заменим удаляемый элемент на наименьший больше исходноого, который гарантировано будет в листе, и удалим такой элемент из листа.

За счёт свойства 3 все операции с деревом совершаются за $O(h)$, где h — высота дерева. Из свойства 1 следует, что если дерево содержащит n элементов, то его высота $h \approx \log_t n$ (выводится из формулы арифметической прогрессии), при чём $\log_t n = \log_2 n / \log_2 t$. Принимая $\log_2 t$ за константу, получим, что все операции с деревом имеют сложность $O(\log n)$.

Запись дерева в файл осуществляется при помощи индексации узлов дерева, отображении на массив и последовательной записи информации об узлах в компактном бинарном представлении. Такой способ представления прост, но в дереве может быть достаточно много пустых узлов. В дереве с высотой h узлы могут содержать от $t-1$ до $2*t-1$ элементов, согласно свойству 1. При отображении на массив будем считать дерево полным, то есть когда в каждом узле по $2*t-1$ элементов. Такое представление имеет достаточно много пустых узлов. Поэтому если узел пустой, то мы просто напишем в файл 0. Чтение дерева из файла осуществляется похожим образом. Обе операции имеют сложность $O(n)$.

2 Исходный код

Написание программы можно разбить на следующие шаги:

1. Реализация шаблонного динамического массива как контейнера для узлов дерева
2. Реализация вставки, поиска и удаления элементов для шаблонных узлов дерева
3. Реализация печати дерева для типа *int* и проверка корректности алгоритма на простых примерах
4. Реализация типа «ключ-значение» и алгоритма записи на диск и чтения с диска дерева
5. Тесты производительности, сравнение результатов с шаблонной реализацией *std::map*

В узле B-дерева хранится несколько ключей и указателей на потомков, нужно уметь разделять и объединять узлы. Линейный список хорошо справляется с этой задачей, но в нём невозможен поиск элементов за $O(\log n)$, что в данной задаче существенно скажется на время работы программы. Будем использовать динамический массив *TVector*.

```
template<class T>
class TVector {
private:
  T* Data;
  unsigned long long DataCurSize;
  unsigned long long DataMaxSize;
public:
  TVector();
  TVector(const unsigned long long n);
  TVector(const unsigned long long n, const T & elem);
  TVector(const TVector & vec);
  ~TVector();

  bool Empty();
  void Erase(const unsigned long long & pos);
  void Insert(const unsigned long long & pos, const T & elem);
  void PopBack();
  void PushBack(const T & elem);
  void Resize(const unsigned long long & n);
  unsigned long long Size();

  T & operator [] (const unsigned long long & index);
  TVector & operator = (const TVector & vec);
};
```

Глобальные переменные $EXPAND\_FACTOR$ и $SHRINK\_FACTOR$ отвечают за то, во сколько раз увеличиться размер вектора при добавлении нового элемента сверх запаса и во сколько раз должен уменьшиться вектор, чтобы мы уменьшили размер выделенной памяти. Это нужно для того, чтобы минимизировать время, нужное на перевыделение памяти.

vector.hpp	
TVector()	Конструктор по умолчанию
TVector(const unsigned long long n)	Конструктор вектора заданного размера
TVector(const unsigned long long n, const T & elem)	Конструктор вектора заданного размера из заданных элементов
~TVector()	Деструктор вектора
bool Empty()	Проверка вектора на пустоту
void Erase(const unsigned long long & pos)	Удаление элемента на указанной позиции
void Insert(const unsigned long long & pos, const T & elem)	Вставка элемента на указанную позицию
void PopBack()	Удалить последний элемент
void PushBack(const T & elem)	Вставить элемент в конец вектора
Resize(const unsigned long long & n)	Функция изменения размера вектора
T & operator [] (const unsigned long long & index)	Перегрузка оператора []
TVector & operator = (const TVector & vec)	Перегрузка оператора присваивания

Шаблонный узел дерева хранит вектор ключей и вектор потомков.

```
1 template<class T>
2 struct TBtreeNode {
3   TVector<T> Data;
4   TVector<TBtreeNode*> Childs;
5   TBtreeNode() : Data(1), Childs(2) {}
6   ~TBtreeNode();
7 };
```

Функции с узлами дерева:

vector.hpp	
void FindNode(TBtreeNode<T>* curNode, TBtreeNode<T>* & res, unsigned long long & pos, const T & elem)	Поиска элемента в дереве. Возвращает указатель на узел в *res* и позицию в *pos*
void SaveNode(const unsigned int & id, TBtreeNode<T>* node, TVector< TVector<T> > & vec)	Отображение дерева на массив
TBtreeNode<T>* LoadNode(const unsigned long long & id, TBtreeNode<T>* & node, TVector< TVector<T> > & vec)	Чтение дерева из массива
~TBtreeNode()	Деструктор узла дерева
unsigned long long TreeHeight(TBtreeNode<T>* node)	Функция нахождения высоты дерева
void EraseNode(TBtreeNode<T>* & node, const T & elem)	Удаление элемента из дерева
void InsertNode(TBtreeNode<T>* & node, TBtreeNode<T>* & toInsertNode, const T & elem)	Вставка элемента в дерево

Реализуем словарь как обёртку над структурой узла дерева.

```
1  template<class T>
2  struct TBtree {
3    TBtree();
4    ~TBtree();
5    void Erase(const T & elem);
6    void Find(TBtreeNode<T>* & res, unsigned long long & pos,const T & elem);
7    void Insert(const T & elem);
8    void WriteInFile(FILE* file);
9    void LoadFromFile(FILE* file);
10 };
```

btree.hpp	
TBtree()	Конструктор B-дерева по умолчанию
~TBtree()	Деструктор B-дерева
void Erase(const T & elem)	Удаление элемента из словаря
void Find(TBtreeNode<T>* & res, unsigned long long & pos, const T & elem)	Поиск элемента в словаре
void Insert(const T & elem)	Вставка элемента в словарь
void WriteInFile(FILE* file)	Функция записи словаря в файл
void LoadFromFile(FILE* file)	Функция чтения словаря из файла

Словарь должен хранить пару ключ-значение, поэтому создадим структуру *TItem* , в которой будем хранить статический массив ключа, размер ключа и значение.

```
1 struct TItem {
2   char Key[MAX_KEY_SIZE + 1];
3   unsigned short KeySize;
4   unsigned long long Value;
5   TItem();
6   TItem(const TItem & it);
7   bool Empty();
8   TItem & operator = (const TItem & it);
9 };
```

item.hpp	
void Clear(char arr[MAX_KEY_SIZE + 1])	Функция очистки массива *char*
TItem()	Конструктор элемента по умолчанию
~TItem()	Деструктор элемента
bool Empty()	Проверка элемента, пустой ли он
TItem & operator = (const TItem & it)	Перегрузка оператора присваивания
bool operator < (const TItem & lhs, const TItem & rhs)	Перегрузка оператора сравнения
bool operator == (const TItem & lhs, const TItem & rhs)	Перегрузка оператора равенства

Наконец, реализуем запросы, описанные в задаче для моего алгоритма и *std::map* в *main.cpp*.

3 Консоль

```
engineerxl@engineerxl:~/DA/lab2$ make
g++ -std=c++17 -pedantic -Wall -Werror -Wextra -c main.cpp
g++ -std=c++17 -pedantic -Wall -Werror -Wextra main.o -o solution
engineerxl@engineerxl:~/DA/lab2$ cat testing/test1.txt
+ Moscow 11920000
! Save Population.b
-Moscow
Moscow
+ NewYork 8348000
+ StPetersburg 4991000
Warsaw
! Load Population.b
+ StPetersburg 4991000
+ NewYork 8348000
Moscow
StPetersburg
Washington
NewYork
London
engineerxl@engineerxl:~/DA/lab2$ ./solution <testing/test1.txt
OK
OK
OK
NoSuchWord
OK
OK
NoSuchWord
OK
OK
OK
OK: 11920000
OK: 4991000
NoSuchWord
OK: 8348000
NoSuchWord
```

4 Тест производительности

Реализованная структура данных сравнивается с *std::map*, в котором ключи представлены *std::string*, а значения *unsigned long long*.

Для вставки в *std::map* используется метод *insert(std::string)*, для удаления *erase(std::string)*, а для поиска *find(std::string)*. Запись и чтение из файла не реализовано в *std::map*, поэтому будем сравнивать только операции вставки, удаления и поиска.

Тесты состоят из 10^3, 10^4 и 10^5 строк. Все ключи в тестах имеют длинну 8 символов.

```
engineerxl@engineerxl:~/DA/lab2$ g++ bench.cpp -O3
engineerxl@engineerxl:~/DA/lab2$ ./a.out <test1000.txt
B-tree: 1.192 ms
std::map: 0.773 ms
engineerxl@engineerxl:~/DA/lab2$ ./a.out <test10000.txt
B-tree: 12.123 ms
std::map: 8.710 ms
engineerxl@engineerxl:~/DA/lab2$ ./a.out <test100000.txt
B-tree: 135.824 ms
std::map: 114.825 ms
```

std::map оказался примерно в полтора раза быстрее, чем мой алгоритм. Стандартный контейнер реализован с помощью красно-чёрного дерева, которое делает меньше копирований элементов при вставке и удалении. В моём алгоритме в узлах дерева хранится динамический массив, который при разбиении и объединение копирует все строки в новые массивы.

5 Выводы

В ходе выполнения лабораторной работы я вспомнил прицип построения бинарного дерева поиска, изучил и реализовал структуру данных B-дерево.

Основной трудностью была работа с указателями, справиться с этим мне помогла утилита Valgrind.

Вызвало затруднение запись дерева в файл, потому что B-дерево нельзя так же просто индексировать, как бинарное дерево поиска (id — индекс вершины, $2 * id$ — индекс левого потомка, $2 * id + 1$ — индекс правого потомка). Я решил индексировать B-дерево, считая его полным. Такая программа работала очень долго, когда она записывала в файл всю информацию о пустом узле (файл с деревом весил до 500 Мб!). После оптимизации мне удалось сжать файл с деревом до пары десятков мегабайт.

Список литературы

[1] *B-tree — Habr.*
URL: `https://habr.com/ru/post/114154/` (дата обращения: 01.11.2020).

[2] *Б, Б+ и Б++ деревья — Algolist.*
URL: `http://algolist.ru/ds/s_btr.php` (дата обращения: 02.11.2020).

[3] *B-дерево — Википедия.*
URL: `http://ru.wikipedia.org/wiki/B_дерево` (дата обращения: 02.11.2020).