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: Д. С. Ляшун
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

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, 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 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 структуры данных: B-дерево.

1 Описание

Требуется написать реализацию структуры данных Б-дерева. Согласно [1], Б-дерево является одним из видов сбалансированных деревьев, при котором обеспечивается эффективное хранение информации на магнитных дисках и других устройствах с прямым доступом. Говоря более строго, Б-деревом называется корневое дерево, устроенное следующим образом [1] :

1. В каждой вершине x содержатся поля, в которых хранится:
   (a) Количество ключей $n[x]$, хранящихся в ней;
   (b) Сами ключи $key_1[x] \leq key_2[x] \leq ... \leq key_{n[x]}[x]$ в неубывающем порядке;
   (c) Булевское значение $leaf[x]$, истинное, когда вершина x является листом.
2. Если x – внутренняя вершина, то она также содержит $n[x]+1$ указатель $c_1[x], c_2[x]$, ..., $c_{n[x]+1}[x]$ на её детей. У листьев детей нет, и эти поля для них не определены.
3. Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях:
   $$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq key_{n[x]}[x] \leq k_{n[x]+1},$$
   где k_i - любой из ключей, хранящихся в поддереве с корнем $c_i[x]$.
4. Все листья находятся на одной и той же глубине (равной высоте h дерева).
5. Число ключей, хранящихся в одной вершине, ограничено сверху и снизу; границы задаются для всего дерева числом $t \geq 2$, которое называется минимальной степенью Б-дерева. По этому значению определяется минимальное и максимальное число ключей во внутренних вершинах (кроме корня) – не менее $t-1$ и не более $2t-1$ соответственно, тогда количество детей в них может варьироваться от t до $2t$.

Стоит отметить, что минимальная степень Б-дерева обычно определяется в зависимости от размера оперативной памяти компьютера и его внешней памяти, поскольку они влияют на то, какое максимальное число элементов может иметь вершина, чтобы её можно было загрузить в оперативную память и обработать, а также сколько вершин может иметь дерево, ведь информация о них хранится во внешней памяти. Оба эти параметра, количество вершин и число элементов в одной вершине как раз и определяются минимальной степенью Б-дерева.

Основными операциями в Б-дереве являются соответственно поиск, вставка и удаление элементов, при этом дерево составляется таким образом, что обращения к жёсткому диску или другому внешнему носителю для получения информации о текущей вершине происходят минимальное число раз. При проведении операций вставки и

удаления элементов в дереве необходимо, чтобы условие №5 всегда выполнялось, тогда будет обеспечена асимптотика $t \log_t N$. Это достигается путём перестройки дерева по мере его обхода: перемещении ключей из одной вершины в другую, слияния двух вершин в одну, разбиение вершины на две и т.д.

Поиск в Б-дереве происходит как в двоичном дереве, но с учетом того, что в вершинах может храниться больше двух ключей. Т.е. в текущей рассматриваемой вершине x производится поиск среди всех хранимых её ключей $key_i[x]$ такого, который не превосходит заданный поиском key: $key \leq key_i[x]$, и является при этом наименьшим. Если они равны, то поиск на этом завершается, в противном случае происходит переход по указателю $c_i[x]$ в дочернюю вершину, в которой таким же способом продолжается поиск. При попадании в лист и не нахождении искомого ключа считается, что его нет в Б-дереве.

Вставка элемента в Б-дерево происходит только в его листья, при обходе по дереву для вставки необходимо всегда поддерживать инвариант: при переходе в вершину количество её хранимых элементов не должно равняться $2t - 1$. Иначе требуется изменить содержимое этой вершины по следующим правилам [1]:

1. Если вершина является корнем, создаётся новая пустая вершина, хранящая в числе указателей старый корень, который будет разрезан на две вершины с числом элементов $t-1$ в каждой без изменения их порядка следования, причем медианный элемент с индексом $t-1$ (при индексации с 0) добавляется в новую вершину, которая станет корнем Б-дерева.

2. Если это дочерняя вершина текущей рабочей вершины, то также происходит её разрезание на две и добавление медианного элемента в текущую вершину. По ключу медианного элемента будет определяться, в какую из полученных вершин нужно перейти для дальнейшей вставки. Если $key \leq key_{t-1}$, то производится переход в левую (разрезанную) вершину, иначе в правую (которая образовалась после разрезания).

В случае попадания в лист, который при соблюдении описанных выше правил обязательно будет незаполненным, происходит обычная вставка элемента с учётом порядка.

Удаление элемента с ключом key из Б-дерева осуществляется похожим образом, но с учётом другого инварианта: при переходе в вершину x количество её хранимых элементов не должно равняться $t-1$ (для корня исключение). Существуют следующие возможные случаи удаления [1]:

1. Если элемент с ключом key находится в вершине x, которая является листом, то происходит его обычное удаление.

2. Если элемент с ключом key содержится во внутренней вершине x, то происходит следующее:

(a) Если ребенок y вершины x содержит не менее t элементов, то производится поиск элемента с ключом key', который непосредственно предшествует key и находится в листе с корнем y. После его поиска осуществляется замена элемента с ключом key в вершине x на key' и удаление key' в листе.

(b) Если ребенок z, следующий за элементом с ключом key, содержит не менее t элементов, поступаем аналогичным образом, но ищем уже ключ key' непосредственно последующий после key.

(c) Если z и y содержат по $t-1$ элементов каждый, тогда производится слияние вершины y, ключа key и вершины z в одну вершину и переход в неё для продолжения удаления ключа key.

3. Если x – внутренняя вершина, но элемента с ключом key в ней нет, нужно найти среди детей корень $c_i[x]$ поддерева, где должен лежать элемент с ключом key (если он вообще есть в дереве). Если корень этого поддерева содержит не менее t элементов, то происходит переход в него и продолжение удаления элемента с ключом key. Если же содержит $t-1$ элемент, то выполняется следующее:

   (a) Если один из соседей вершины $c_i[x]$ (например, правый) содержит не менее t элементов (под соседом понимается такой ребёнок вершины x, который отделён от $c_i[x]$ только одним ключом-разделителем), то в $c_i[x]$ добавляется элемент родителя x (его ключ-разделитель), в свою очередь в x передаётся левый ребёнок этого соседа. Тогда самый левый ребёнок соседа станет самым правым ребёнком вершины $c_i[x]$.

   (b) Если оба соседа вершины $c_i[x]$ содержат по $t-1$ элементу, то производится её объединение с одной из соседних вершин, при этом ключ разделитель станет медианой новой вершины.

Также стоит отметить, если в результате удаления корень стал пустым, то он удаляется и его единственный ребёнок становится новым корнем, а высота Б-дерева уменьшается на единицу, и тогда корень не пустой (если только всё дерево не пусто).

2 Исходный код

Для хранения информации об элементе в Б-дереве (его пары ключ-значение) была описана структура $TItem$:

```
1 struct TItem {
2     TItem();
3     ~TItem();
4     char* Key;
5     unsigned long long* Value;
6 };
```

При обработке ключей, являющихся строками, использовались следующие функции:

string.hpp	
Compare StrCompare(char* fir, char* sec)	Производит лексикографическое сравнение строк fir и sec. Возвращает перечислимый тип Compare, который может принимать значения EQUALS, MORE и LESS в зависимости от результата операции.
void Convert(char* str)	Производит перевод всех букв символьной строки str в нижний регистр.
void Assign(char* fir, char* sec)	Производит присваивание строки fir значения строки sec.

Для представления вершины Б-дерева была определена структура $TNode$:

```
1 struct TNode {
2     TNode();
3     ~TNode();
4     TNode* NodeArray[MAX_NODE_COUNT];
5     TItem* DataArray[MAX_DATA_COUNT];
6     bool IsLeaf;
7     int NodesCnt;
8     int DataCnt;
9 };
```

При осуществлении операций поиска, вставки, удаления элемента в Б-дереве, а также его чтении и записи в файл, использовались следующие функции, работающие непосредственно с $TNode$:

node.hpp	
void Print(TNode* node, int depth)	Процедура печати дерева с корнем node (использовалась при отладки программы).

unsigned long long* Search(TNode* node, char* const key)	Функция поиска ключа key в дереве с корнем node. Возвращает указатель на числовое значение элемента; если элемент по ключу найден не был, то возвращается нулевой указатель.
void SplitChild(TNode* node, const int& index)	Процедура разрезания в вершине дерева node дочерней вершины с индексом index на две с количеством элементов $t-1$ в каждой, медианный элемент добавляется в node.
void MergeChild(TNode* node, const int& big, const int& les)	Процедура слияния в вершине дерева node двух дочерних вершин с индексами big и les с произвольном числом элементов в одну вершину, элемент-разделитель вершин также добавляется в вершину как медианный.
Result Insert(TNode*& node, char* const key, const unsigned long long& value)	Функция вставки в дерево с корнем node элемента с ключом key и значением value, при этом производящая проверку и перестройку корня в случае заполненности. Возвращает перечислимый тип Result, который может принимать значение SUCCESS или FAIL в зависимости от результата операции.
Result InsertNonFull(TNode* node, char* const key, const unsigned long long& value)	Функция вставки в незаполненное дерево с корнем node элемента с ключом key и значением value.
Result Erase(TNode* node, char* const key)	Функция удаления элемента с ключом key из поддерева с корнем node с приемлемым числом элементов в нём.
Result EraseInRoot(TNode*& node, char* const key)	Функция удаления элемента с ключом key из дерева с корнем node. Перед удалением проверяется, может ли корень стать пустым, в случае этого производится его изменение.
Result ReadFromFile(TNode* const node, FILE* const file)	Функция чтения из файла с именем file представления вершины дерева и запись его в вершину node.
void WriteToFile(TNode* const node, FILE* const file)	Процедура записи в файл с именем file представления вершины дерева node.

Для работы с массивами данных в вершинах дерева были использованы следующие функции:

node.hpp	
template<class T> void Insert(T* array[], int& count, T* elem, const int index)	Процедура вставки в массив array размера count элемента elem в позицию index.
template<class T> void Delete(T* array[], int& count, const int index)	Процедура удаления из массива array размера count элемента в позиции index.
int BinSearch(TItem* array[], const int& count, char* key)	Функция бинарного поиска в массиве array типа TItem и размера count элемента с ключом key.

Для более удобной работы пользователя с Б-деревом был описан внешний интерфейс в виде класса $TBtree$:

```
1  class TBtree {
2      public :
3          TBtree();
4          ~TBtree();
5          unsigned long long* Search(char* const);
6          Result Insert(char* const, const unsigned long long&);
7          Result Erase(char* const);
8          void Print();
9          void WriteToFile(FILE* const file);
10         Result ReadFromFile(FILE* const file);
11     private :
12         TNode* Node;
13 };
```

Методы класса $TBtree$ имеют следующее назначение:

btree.hpp	
unsigned long long* Search(char* const key)	Функция поиска элемента с ключом key в Б-дереве. Возвращает указатель на числовое значение элемента; если элемент по ключу найден не был, то возвращается нулевой указатель.
Result Insert(char* const key, const unsigned long long& value)	Функция вставки элемента с ключом key и значением value в Б-дерево. Возвращает перечислимый тип Result, который может принимать значение SUCCESS или FAIL в зависимости от результата операции.

Result Erase(char* const key)	Функция удаления элемента с ключом key из Б-дерева.
void Print()	Процедура печати Б-дерева (использовалась при отладке программы).
void WriteToFile(FILE* const file)	Процедура записи Б-дерева в файл с именем file.
Result ReadFromFile(FILE* const file)	Процедура чтения Б-дерева из файла с именем file.

3 Консоль

```
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ make
g++ -std=c++11 -O2 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result
-pedantic -o solution main.cpp
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ cat test.txt
+ ab 36
+ vg 1
+ ak 12
ab
-Ab
+ kj 12
-vv
-ak
ak
+ vg 22
-vg
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ ./solution <test.txt
OK
OK
OK
OK: 36
OK
OK
NoSuchWord
OK
NoSuchWord
Exist
OK
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$
```

4 Тест производительности

Тест производительности представляет из себя сравнение суммарного времени работы операций вставки, удаления и поиска элементов между std::map из стандартной библиотеки шаблонов, реализованного посредством красно-черного дерева, и собственного Б-дерева. Общее количество операций - тысяча, десять тысяч и один миллион, длина ключей элементов составляет 5 символов, а их значения не превосходят 10^8.

```
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ g++ benchmark.cpp -o benchmark
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ ./benchmark <test1000.txt
std::map work time = 2543 ms,btree work time = 1609 ms.
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ ./benchmark <test10000.txt
std::map work time = 38095 ms,btree work time = 27266 ms.
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$ ./benchmark <test1000000.txt
std::map work time = 4257768 ms,btree work time = 2412531 ms.
dmitry@dmitry-VirtualBox:~/DA_labs/lab2$
```

Как видно, Б-дерево выиграло у std::map по времени работы, несмотря на большую асимптотику операций – $O(8 \log_8 N)$ при $t = 8$ по сравнению с $O(\log_2 N)$. Это связано с тем, что данная реализация Б-дерева более эффективная и базировалась на использовании указателей на ключи и значения элементов, тем самым ускоряя процесс перестраивания дерева, поскольку не требуется работать с долгим присваиванием элементов, а только изменять указатели на них.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать одно из сбалансированных деревьев поиска – Б-дерево, которое позволяет проводить эффективную работу с объектами, упорядоченными по ключу. При написании реализации Б-дерева я узнал новое о работе указателей в C++, в частности в каких случаях необходимо передавать указатель по ссылке в функции, а также чем отличается константный указатель от указателя на константу.

Основную трудность для меня составила работа с объектами в динамической памяти и указателями на них, например, возникновением висячих указателей, не указывающих не на что, или же потеря объектов, когда на них больше ничто не указывает. Также могу сказать, что отладка данной программы была довольно долгой из-за большого размера исходного кода и его сложности.

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] *Linux Man Pages*
URL: `http://ru.manpages.org/` (дата обращения: 18.11.2020).