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: Н. П. Ежов
Преподаватель: Н. С. Капралов
Группа: М8О-204Б
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 264 - 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.

Вариант структуры данных: `Красно-чёрное дерево.`

1 Описание

Требуется написать реализацию структуры данных "красно-чёрное деревоЭ(RB tree). Как сказано в [3]: "Красно-чёрное дерево представляет собой бинарное дерево поиска с одни дополнительным байтов цвета в каждом узле. Цвет узла может быть либо красным, либо чёрным... Бинарное дерево поиска является красно-чёрным деревом, если оно удовлетворяет следующим своёствам:
1. Каждый узел является либо красным, либо чёрным.
2. Корень дерева является чёрным узлом.
3. Каждый лист дерева (NULL) является чёрным узлом.
4. Если узел красный, то оба его дочерних узла чёрные.
5 Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество чёрных узлов.
В соответсвии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-чёрном дереве не отличается от другого по длине более чем в два раза, так что красно-чёрные деревья являются приближенно сбалансированными."
Красно-чёрное дерево поддерживает операции вставки, удаления и поиска в дереве, как и обычное бинарное дерево. Однако данные операции даже в худшем случае гарантируют время выполнения O(log(n)). В целом, сложность этих операций напрямую зависит от высоты дерева. Красно-чёрное дерево с N внутренними узлами имеет высоту, не превышающую 2*log(N+1).
Доказательство. Докажем по индукции, что для любого x дерево с корнем x содержит как минимум $2^{bh(x)}$ - 1 внутренних узлов, где $bh(x)$ - чёрная высота вершины x. Если чёрная высота равна 0, то узел x - терминальный, значит по формуле поддерево узла x содержит не менее $2^0 - 1 = 0$ внутренних узлов. Теперь рассмотрим общий случай для вершины x, которая имеет высоту $bh(x)$, каждый дочерний узел имеет высоту либо $bh(x)$, либо $bh(x) - 1$. Предположим, что для левого и правого поддеревьев x формула верна, тогда в случае, когда левое и правое поддеревья имеют чёрные высоты bh(x) - 1, по предположению индукции мы имеем, что каждый потомок x имеет как минимум $2^{bh(x)} - 1$ внутренних вершин. Таким образом, дерево с вершиной в x имеет не меньше $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ вершин, что показывает, что формула верна.
Очевидно, что в худшем случае высота дерева $bh(x)$ не больше чем в 2 раза превышет $bh(x)$, так как минимум половина вершин на пути к листу от корня, не считая корень, должны быть чёрными. Отсюда из $N \geq 2^{bh(x)} - 1$ и $2bh(x) \geq h(x) \rightarrow N \geq 2^{h(x)/2} - 1$, и значит $h \leq 2log(N + 1)$, ЧТД.

2 Исходный код

Сначала напишем свой класс *NMyStd::TItem* для хранения пар типа "ключ-значения где типом ключа будет массив типа *char[257]*, а типом значения будет *unsigned long long*.

Листинг item.hpp

```
//
// item.hpp
//

#ifndef LAB2_ITEM_HPP
#define LAB2_ITEM_HPP
const long long MAX_LEN = 256;
namespace NMyStd{
    struct TItem{
        unsigned long long Value;
        char Key[MAX_LEN+1];
    };
}
#endif //LAB2_ITEM_HPP
```

В файле *main.cpp* будем обрабатывать запросы, вызывая нужные методы у класса красно-чёрного дерева. В цикле while до конца файла считываем очередной запрос, выполняем операцию, если она завершилась успехом, то выводим соответсвующее сообщение, иначе показываем сообщения о неудаче. Так как по условию мы работает с регистронезависимыми строками, то при считывании ключа через цикл привёдем его к нижнему регистру.

Листинг main.cpp

```
//
// main.cpp
//
#include <iostream>
#include <cstring>
#include <cctype>
#include "item.hpp"
#include "tree.hpp"

int main(){
    char readStr[MAX_LEN+1];
    NMyStd::TRBTree* rbTree = new NMyStd::TRBTree;
    while(scanf("%s", readStr) > 0){
        if (strcmp(readStr,"+") == 0) {
            char key[MAX_LEN + 1];
            unsigned long long val;
```

```
17              scanf("%s%llu", key, &val);
18              for (int i = 0; i < strlen(key); ++i){
19                  key[i] = (char)tolower(key[i]);
20              }
21              NMyStd::TItem item;
22              item.Value = val;
23              std::memcpy(item.Key, key, sizeof(char)*(MAX_LEN+1));
24              if (rbTree->Insert(item)) {
25                  printf("OK\n");
26              } else {
27                  printf("Exist\n");
28              }
29          } else if (strcmp(readStr,"-") == 0) {
30              char key[MAX_LEN + 1];
31              scanf("%s", key);
32              for (int i = 0; i < strlen(key); ++i){
33                  key[i] = (char)tolower(key[i]);
34              }
35              if (rbTree->Remove(key)) {
36                  printf("OK\n");
37              } else {
38                  printf("NoSuchWord\n");
39              }
40          } else if (strcmp(readStr,"!") == 0) {
41              char path[MAX_LEN + 1];
42              scanf("%s %s", readStr, path);
43              bool isOK = true;
44              if (strcmp(readStr,"Save") == 0) {
45                  NMyStd::TRBTree::Save(path, *rbTree, isOK);
46                  if (isOK) {
47                      printf("OK\n");
48                  }
49              } else {
50                  NMyStd::TRBTree* tmpTreePtr = new NMyStd::TRBTree;
51                  NMyStd::TRBTree::Load(path, *tmpTreePtr, isOK);
52                  if (isOK) {
53                      printf("OK\n");
54                      delete rbTree;
55                      rbTree = tmpTreePtr;
56                  } else {
57                      delete tmpTreePtr;
58                  }
59              }
60          } else {
61              for (int i = 0; i < strlen(readStr); ++i){
62                  readStr[i] = (char)tolower(readStr[i]);
63              }
64              NMyStd::TItem ans;
65              if (rbTree->Search(readStr, ans)) {
```

```
66                 printf("OK: %llu\n", ans.Value);
67             } else {
68                 printf("NoSuchWord\n");
69             }
70         }
71     }
72     delete rbTree;
73     return 0;
74 }
```

Для написания дерева сначала опишем структуру вершины красно-чёрного дерева *TRBNode*, в которой будем хранить указатель на левого сына, правого сына и родителя. Также будем хранть цвет вершины и поле с данными типа *TItem*. Теперь создадим класс красно-чёрного дерева с одним полем - корнем дерева, и шестью пользовательскими методами: вставка, удаление, поиск, сохранение в файл, выгрузка из файла и геттер корня дерева. Для этих шести операций напишем вспомогательные приватные методы поиска, удаления, вставки, загрузки, выгрузки, а также напишем вспомогательные методы поворотов, перекраски после вставки и удаления.
Поиск в красно-чёрном дереве ничем не отличается от поиска в бинарном дереве. Мы также идём налево, если ключ, который мы ищем, меньше того, что в вершине, если ключ больше, то идём направо. Если ключи равны, то мы нашли наш элемент. Если мы дошли до NULL-листа, то такого элемента с искомым ключом нет.
Вставка в красно-чёрное дерево отличается от вставки в обычное бинарное дерево поиска. При вставке нужно учитывать, что некоторые свойства дерева могут нарушиться. Новый узел в красно-чёрное дерево жобавляется на место одного из листьев, окрашивается в красный цвет. Потом вызывается функция *InsertFixUp* для восстановления свойств дерева. При вставке красной вершины может испортиться только свойство 2 и 4. Нарушение свойства 2 будем обрабатывать сразу, вне функции *InsertFixUp*. В InsertFixUp в зависимости от цвета дяди делаем следующие действия. Если дядя красный, окрашиваем дядю и отца в чёрный, деда окрашиваем в красный и запускаем алгоритм вверх от деда, либо, если дядя чёрный, при помощи поворотов подвешиваем поддерево с корнем-дедом за отца вершины, делая деда дочерней вершиной отца добавленной вершины. Поиск места для вставки занимает $O(log(N))$. Так как в худшем случае мы можем перекрашивать дерево рекурсивно вплоть до корня, то, зная ограничения на высоту красно-чёрного дерева, можно сказать, что будет не более $O(log(N))$ выполнений *InsertFixUp*. Задача, когда отец - правый сын деда решается зеркально.
При удалении узла с двумя не листовыми потомками в обычном двоичном дереве поиска мы ищем либо наибольший элемент в его левом поддереве, либо наименьший элемент в его правом поддереве и перемещаем его значение в удаляемый узел. Затем мы удаляем узел, из которого копировали значение. Копирование значения из одного узла в другой не нарушает свойств красно=чёрного дерева, так как структура дерева и цвета узлов не изменяются. Стоит заметить, что новый удаляемый узел не

может иметь сразу два дочерних нелистовых узла, так как в противном случае он не будет являться наибольшим/наименьшим элементои.

3 Консоль

```
(base) nikita@nikita-desktop:~/Diskran/lab1$ make
g++ -O3 -std=c++14 -o solution main.cpp vector.cpp vector.h
(base) nikita@nikita-desktop:~/Diskran/lab1$ cat test_input.txt
a5db3d43b37a95cd00618a7ac3112f7e LQFZzcjKUIgaNHdxMhDRzSojQdKKdJRxqntO
d40c0348390bdb0e0fee9348cc8d2e67 vVNAzruNkZUPUUqHcmfWgkwdOGTSJeaAINZ
364590e5c89b6b1c54231793d261a3b2 wMmpTNRfxIrkciktSkSdLYabpVgehC
734bf391f564bd5aff79a1fefeb358b7 m
e31adffb1b4418881731733600387d82 vJCqcbdkeEVJLWwRbPTRWk
caf836a9f342e48deb43e8215b23b177 FbGyVbPqdyvlQoJoqmvdiaCboIkNOILfldOAPtjxbzoVUxFKSOvp
5a3378d5ca2706bffab672e07894212b QKutCZfccVqEZORVvVxPICHQYAOemh
0472d854d83d11c287ec7d9eff9b8146 CvqgXrzHlEWFRIORDYhvJH
c41d2af1b376b9fc1f95fce356012712 SYhTAeNdhtZlTAWOreQeMNPaGgFSYNRMpoldNrUDsEAIR
8a18a352c07e3af6411007385bffd0ed ibdyiGQrGFQbLZngBwpsNoMiUrIGvkQwHwYGwVSWZ
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./solution <test_input.txt
0472d854d83d11c287ec7d9eff9b8146 CvqgXrzHlEWFRIORDYhvJH
364590e5c89b6b1c54231793d261a3b2 wMmpTNRfxIrkciktSkSdLYabpVgehC
5a3378d5ca2706bffab672e07894212b QKutCZfccVqEZORVvVxPICHQYAOemh
734bf391f564bd5aff79a1fefeb358b7 m
8a18a352c07e3af6411007385bffd0ed ibdyiGQrGFQbLZngBwpsNoMiUrIGvkQwHwYGwVSWZ
a5db3d43b37a95cd00618a7ac3112f7e LQFZzcjKUIgaNHdxMhDRzSojQdKKdJRxqntO
c41d2af1b376b9fc1f95fce356012712 SYhTAeNdhtZlTAWOreQeMNPaGgFSYNRMpoldNrUDsEAIR
caf836a9f342e48deb43e8215b23b177 FbGyVbPqdyvlQoJoqmvdiaCboIkNOILfldOAPtjxbzoVUxFKSOvp
d40c0348390bdb0e0fee9348cc8d2e67 vVNAzruNkZUPUUqHcmfWgkwdOGTSJeaAINZ
e31adffb1b4418881731733600387d82 vJCqcbdkeEVJLWwRbPTRWk
```

4 Тест производительности

Время на ввод и посторение структур данных не учитывается, замеряется только время, затраченное на сортировку. Количество пар «ключ-значение» для каждого файла равно десять в степени номер теста минус один. Например, $02.t$ содержит десять пар, а $07.t$ миллион.

```
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/04.t >04.a
custom bitwise sort 6 ms
stable sort from std 0 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/05.t >05.a
custom bitwise sort 9 ms
stable sort from std 6 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/06.t >06.a
custom bitwise sort 37 ms
stable sort from std 168 ms
(base) nikita@nikita-desktop:~/Diskran/lab1$ ./benchmark <tests/07.t >07.a
custom bitwise sort 478 ms
stable sort from std 2371 ms
```

Глядя на результаты, видно, что $std :: stable\_sort$ выигрывает больше всего на самых маленьких тестах, а $BitWisesort$ на самых больших. Сложность $std :: stable\_sort$ $O(n * log\ n)$, а сложность $Radixsort$ $O(m * n)$, где m - количество разрядов в числе. Так как логарифм возрастающая функция, переломный момент наступает, когда $log\ n$ становится больше, чем постоянная m.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать шаблонный класс с динамическим выделением памяти на примере реализации класса $TVector$. Реализовал поразрядную сортировку и стабильную сортировку подсчетом. Узнал, что оператор копирования для массива $char$-ов работает очень медленно, соответсвенно выгодно в векторе хранить указатели на масисв, но не массив целиком. Хоть на больших тестах(10^5 и больше) соритровка по разрядам быстрее чем $std :: stable\_sort$, но пространсвенная сложность $O(n+m)$ (где n - рамер входного массива, а m - размер разряда) дает о себе знать. Также из-за сложности $O(n*m)$ поразрядная сортировка очень медленно работает для чисел, где много разрядов, из-за чего она может быть медленее $std :: stable\_sort$, поэтому конкретно в этом варианте задачи целесообразнее разделить 32-ричное число на 8 разрядов по 4 числа

Список литературы

[1] *Википедия - Красно-чёрное дерево.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево` (дата обращения: 03.11.2020).

[2] *C++ Reference.*
URL: `https://en.cppreference.com/w/cpp/container/map` (дата обращения: 03.11.2020).

[3] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 3-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.)) стр. 220 - 229 (дата обращения: 03.11.2020).