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. А. Мариничев
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 21.11.20
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: AVL-дерево.

1 Описание

Программе подаются входные данные через стандартный поток ввода и, как следствие, весьма удобно считывать их циклом **while** (пока значение может быть прочитано, продолжать цикл).

AVL-дерево сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой вершины высота её двух поддеревьев различается не более чем на 1.

Данное дерево было решено реализовывать на основе двух структур **TAVLTree** и **TAVLNode**. В структуре дерева хранится указатель на корень дерева.

Структура узла содержит:

- Ключ;
- Значение;
- Высоту узла;
- Указатель на левого сына;
- Указатель на правого сына;

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $|h(L) - h(R)| = 2$, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева $|h(L) - h(R)| \leq 1$, иначе ничего не меняет. Есть 4 типа вращений для балансировки:

1. Малое левое вращение.
2. Малое правое вращение.
3. Большое левое вращение.
4. Большое правое вращение.

Вставка элемента. Пусть нам надо добавить ключ k. Будем спускаться по дереву, как при поиске ключа k. Если мы стоим в вершине x и нам надо идти в поддерево, которого нет, то делаем ключ k листом, а вершину его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то $balance[i]$ увеличивается на единицу, если из правого, то уменьшается на единицу. Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит высота поддерева

изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным 2 или -2, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем *O(h)* вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет *O(logn)* операций.

Удаление вершины. Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину x, переместим её на место удаляемой вершины и удалим вершину x. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину с из левого поддерева, то $balance[i]$ уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или -2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, *O(h)* операций. Таким образом, требуемое количество действий — *O(logn)*.

Также был реализован строковый тип **TString** для удобной работы с ключами.

2 Исходный код

Проект состоит из 4 файлов:
- **2-1.cpp**: главный файл в котором реализована функция main;
- **avltree.hpp**: реализация AVL-дерева;
- **dictionary.hpp**: реализация словаря;
- **string.hpp**: реализация строкового типа данных;

Таблица методов и функций

2-1.cpp	
Функция	Значение
int main()	Главная функция, в которой происходит чтение данных и создание словаря.
avltree.hpp	
Тип данных	Значение
class TAVLTree	Класс AVL-дерева.
struct TAVLNode	Структура узла AVL-дерева.
Функция	Значение
TAVLNode()	Конструктор по умолчанию.
TAVLNode(K key, V value)	Конструктор.
TAVLNode()	Деструктор.
unsigned long long Height(const TAVLNode *node)	Функция для вычисления высоты поддерева с корнем в данном узле.
const T& Max(const T& a, const T& b)	Вспомогательная функция для вычисления максимального эл-та.
void Reheight(TAVLNode *node)	Функция для корректировки данных о высоте данного узла.
int Balance(const TAVLNode *node)	Функция для вычисления баланса данного узла.
TAVLNode *RotateLeft(TAVLNode *x)	Левый попворот относительного данного узла.
TAVLNode *RotateRight(TAVLNode *y)	Правый поворот относительно данного узла.
TAVLNode *BigRotateLeft(TAVLNode *x)	Болшой левый поворот (правый поворот отн-но правого дочернего узла + левый поворот отн-но данного узла).

TAVLNode *BigRotateRight(TAVLNode *x)	Болшой правый поворот (левый поворот отн-но левого дочернего узла + правый поворот отн-но данного узла).
TAVLNode *Rebalance(TAVLNode *node)	Перебалансировка дерева.
TAVLNode *Insert(TAVLNode *node, K k, V v)	Вставка узла в АВЛ-дерево.
TAVLNode *RemoveMin(TAVLNode *node, TAVLNode *currentNode)	Поиск и удаление узла с минимальным ключом.
TAVLNode *Remove(TAVLNode *node, K k)	Удаление узла в АВЛ-дереве.
TAVLNode *Search(TAVLNode *node, K k)	Поиск узла в АВЛ-дереве.
TAVLTree()	Конструктор.
void Add(K k, V v)	Метод добавления в дерево.
void Delete(K k)	Метод удаления из дерева.
TAVLNode *Find(K k)	Метод нахождения узла в дереве.
void TreeDelete(TAVLNode *node)	Метод удаления дерева.
TAVLTree()	Деструктор.
void Save(std::ostream &os, const TAVLNode *node)	Метод сохранения дерева в файл.
TAVLNode *Load(std::istream &is, const TAVLNode *node)	Метод загрузки дерева из файла.
bool OpenFileSave(TString &fileName)	Метод открытия файла для сохранения.
bool OpenFileLoad(TString &fileName)	Метод открытия файла для загрузки.
dictionary.hpp	
Тип данных	Значение
struct TDictionary	Структура словарь.
Функция	Значение
void Lowercase(TString &str)	Метод, обеспечивающий регистронезависимость ключа.
void InputInsert()	Метод для вставки слова в словарь.
void InputRemove()	Метод для удаления слова из словаря.
void InputFind(const TString &k)	Метод для нахождения слова в словаре.
void InputSaveOrLoad()	Метод для сохранения словаря в файл или его загрузки.
string.hpp	
Тип данных	Значение

struct TString	Строковая структура.
Функция	Значение
TString()	Конструктор по умолчанию.
TString(const valueType *str)	Конструктор.
TString(const TString &str)	Конструктор.
TString(TString &&str)	Конструктор.
TString &operator=(const valueType *str)	Перегрузка оператора =.
TString &operator=(const TString &str)	Перегрузка оператора =.
TString &operator=(TString &&str)	Перегрузка оператора =.
void CstrMove(valueType *str)	Метод Move для данной строки.
void Swap(TString &str)	Обмен данными между строками.
void PushBack(const valueType &ch)	Метод для добавления символа в строку.
TString()	Деструктор.
iterator begin()	Начало итерирования по строке.
constIterator begin()	Начало итерирования по строке.
iterator end()	Завершение итерирования по строке.
constIterator end()	Завершение итерирования по строке.
const valueType *Cstr()	Метод для нахождения размера строки.
int Size()	Метод для нахождения размера строки.
TString &operator+=(const valueType &ch)	Перегрузка оператора +=.
const valueType &At(int index)	Получение символа в строке.
valueType &At(int index)	Получение символа в строке.
const valueType &operator[](int index)	Перегрузка оператора [].
valueType &operator[](int index)	Перегрузка оператора [].
friend std::ostream &operator«(std::ostream &os, const TString &str)	Перегрузка оператора вывода.
friend std::istream &operator»(std::istream &is, TString &str)	Перегрузка оператора ввода.
bool operator<(const TString &lhs, const TString &rhs)	Перегрузка оператора <.
bool operator>(const TString &lhs, const TString &rhs)	Перегрузка оператора >.
bool operator==(const TString &lhs, const TString &rhs)	Перегрузка оператора ==.

bool operator!=(const TString &lhs, const TString &rhs)	Перегрузка оператора !=.

3 Тест производительности

Тест производительности представляет из себя следующее: моя реализация AVL-дерева сравнивается с **std::map**, который реализован как красно-чёрное дерево. В то время как в обоих алгоритмах операции вставки/удаления равнозначны по сложности и выполняются за *O (log n)*, в случае перебалансировки красно-чёрного дерева поворот выполняется за *O (1)*, тогда как в AVL-дереве эта операция выполняется за *O (log n)*, что делает красно-чёрное дерево более эффективным в этом аспекте этапа ребалансировки. Тесты состоят из случайного числа строк. Время выводится наносекундах. Для замерки времени использовалась бибилиотека **chrono**.
Тесты создавались с помощью программы на языке Python:

```
import sys
import random
import string

def get_random_key():
    return random.choice( string.ascii_letters )

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print( "Usage: {0} <count of tests>".format( sys.argv[0] ) )
        sys.exit(1)

    count_of_tests = int( sys.argv[1] )

    actions = [ "+", "?" ]

    for enum in range( count_of_tests ):
        keys = dict()
        test_file_name = "tests/{:02d}".format( enum + 1 )
        with open( "{0}.t".format( test_file_name ), 'w' ) as output_file, \
             open( "{0}.a".format( test_file_name ), "w" ) as answer_file:

            for _ in range( random.randint(1, 10 ** 6) ):
                action = random.choice( actions )
                if action == "+":
                    key = get_random_key()
                    value = random.randint(1, 100)
                    output_file.write("+ {0} {1}\n".format( key, value ))
                    key = key.lower()
                    answer = "Exists"
                    if key not in keys:
                        answer = "OK"
                        keys[key] = value
                    answer_file.write( "{0}\n".format( answer ) )

                elif action == "?":
```

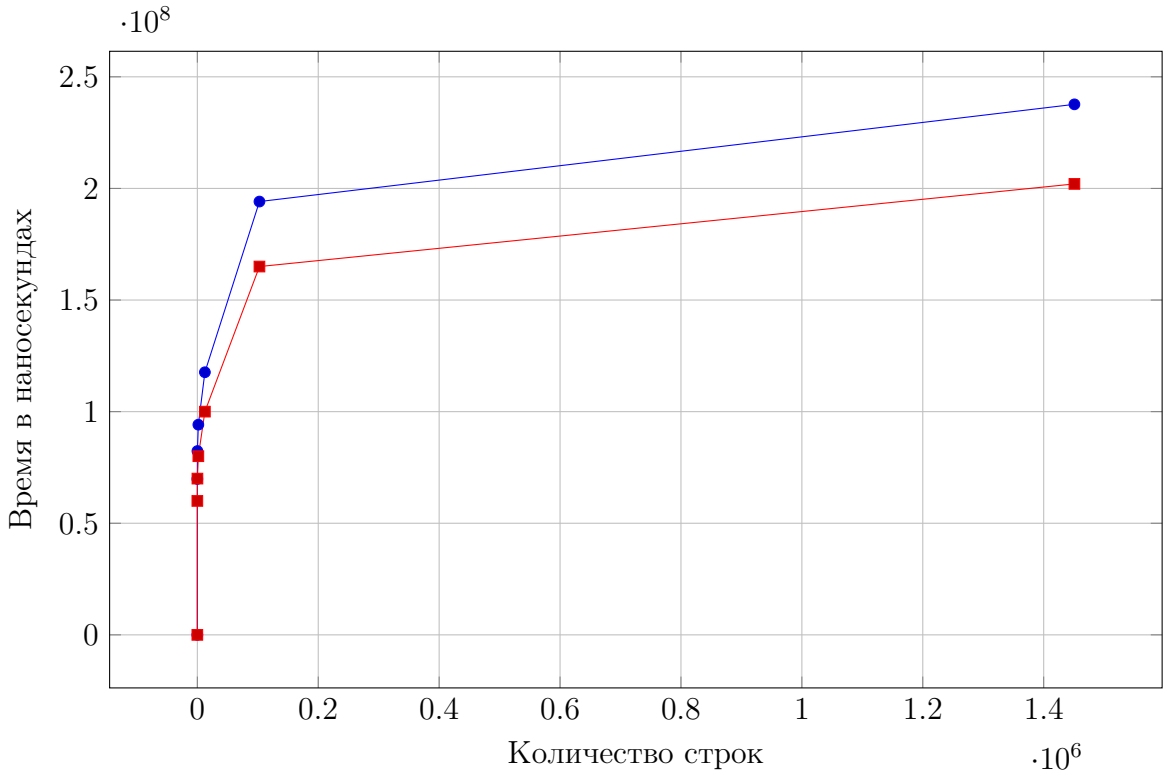
```
37                  search_exist_element = random.choice([True, False])
38                  key = random.choice([key for key in keys.keys() ]) if
                        search_exist_element and len(keys.keys()) > 0 else
                        get_random_key()
39                  output_file.write("{0}\n".format(key))
40                  key = key.lower()
41                  if key in keys:
42                      answer = "OK: {0}".format(keys[key])
43                  else:
44                      answer = "NoSuchWord"
45                  answer_file.write("{0}\n".format(answer))
```

Графики

Поиск элемента за *O (log n)*

Синий: **std::map** (красно-чёрное дерево).

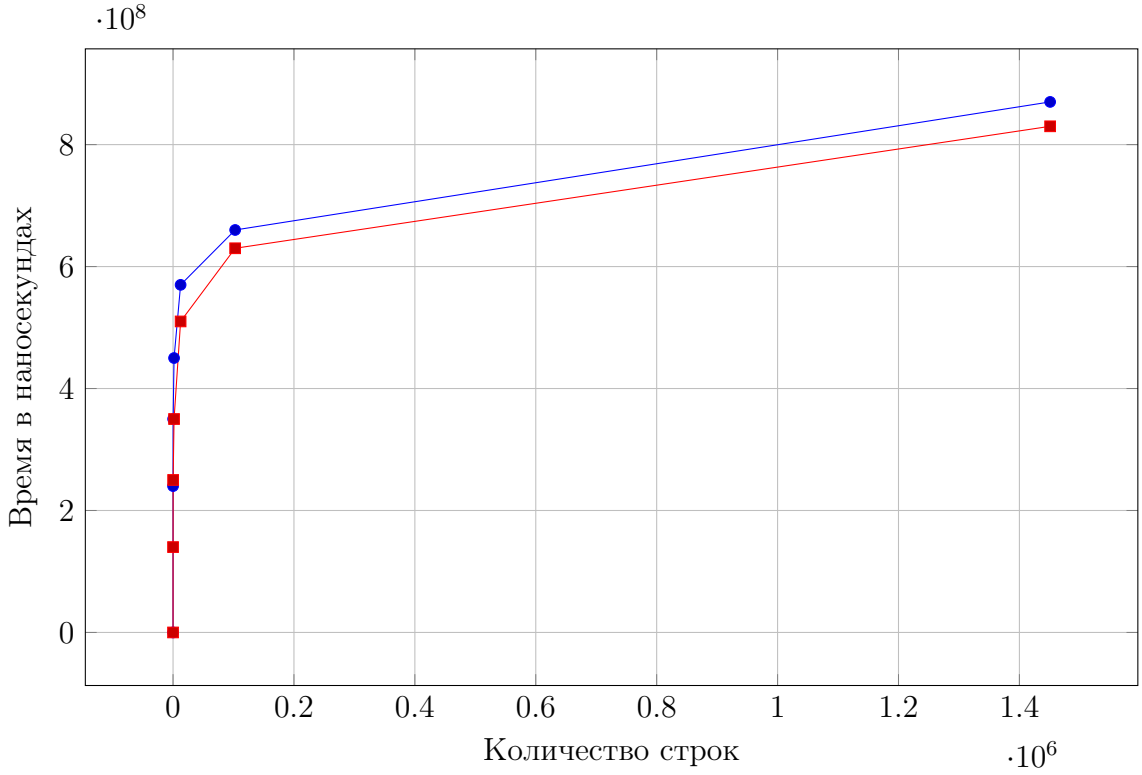
Красный: AVL-дерево.



Вставка элемента за *O (log n)*

Синий: **std::map** (красно-чёрное дерево).

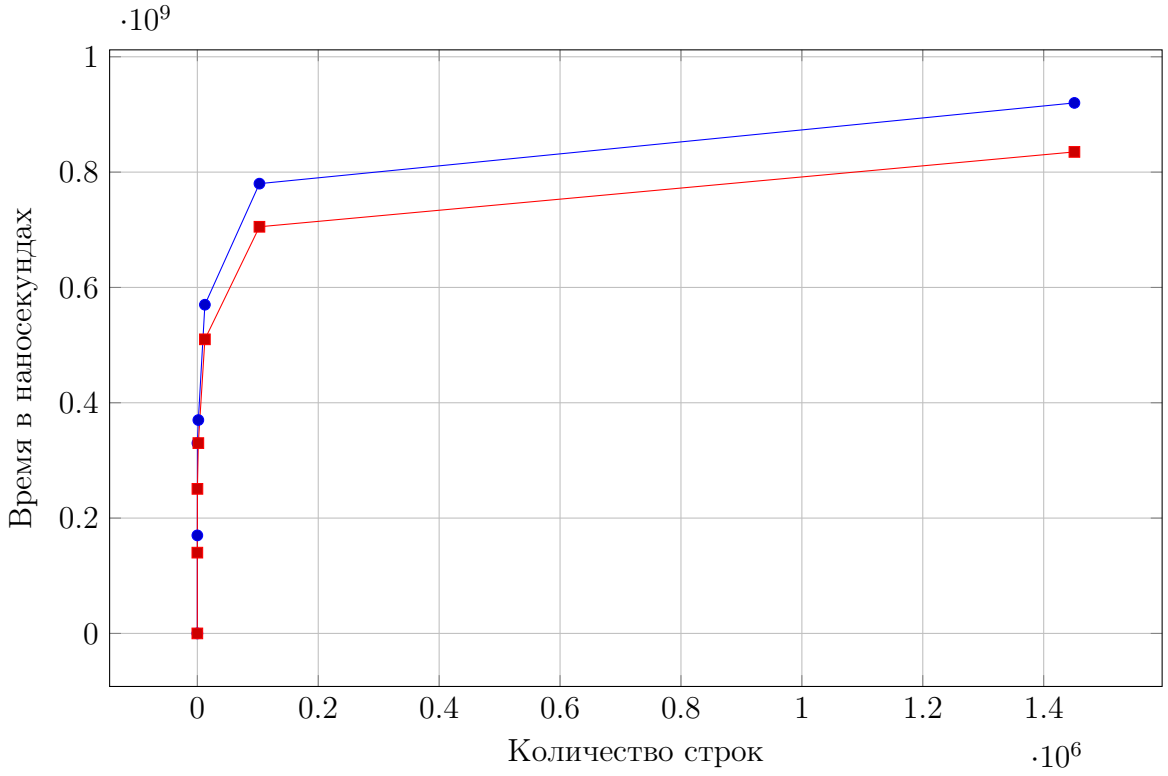
Красный: AVL-дерево.



Удаление элемента за *O (log n)*

Синий: **std::map** (красно-чёрное дерево).

Красный: AVL-дерево.



4 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился применять теоретические знания о таких структурах данных, как сбалансированные деревья, а именно о AVL-дереве, на практике. Смог написать свою реализацию строкового типа, которая может понадобиться при выполнении дальнейших лабораторных работ. Закрепил опыт написания генератора тестов и benchmark-а, для замеров времени работы определенной реализации алгоритма. А также улучшил свои навыки в отлаживании программы при работе с чекером.

Поговорим о целесообразности применения такой структуры данных, как AVL-дерево. В худшем случае, когда обычное бинарное дерево поиска вырождено в линейный список, хранение данных в упорядоченном бинарном дереве никакого выигрыша в сложности операций по сравнению с массивом или линейным списком не дает. Для сбалансированных деревьев, которым и является AVL-дерево, для всех операций получается логарифмическая сложность, что гораздо лучше. Но в тоже время существует такая структура данных, как хэш-таблица, где те же операции работают за $O\ (1)$, в худшем случае за $O\ (n)$. На мой взгляд, AVL-дерево и хеш-таблица являются доволно конкурентоспособными структурами данных и выбор между ними будет зависеть от требований той или иной задачи.

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] Дональд Э. Кнут. *Искусство программирования. Том 3. Сортировка и поиск, 2-е издание.* — Правообладатель «Диалектика-Вильямс», 2018. Перевод с английского: И. В. Красиков, В. Т. Тертышный. — 834 с. (ISBN 978-5-8459-0082-1, 0-201-89685-0)

[3] *AVL Tree | Set 1 (Insertion)*
URL: `https://www.geeksforgeeks.org/avl-tree-set-1-insertion/?ref=lbp` (дата обращения: 09.11.2020).

[4] *AVL Tree | Set 2 (Deletion)*
URL: `https://www.geeksforgeeks.org/avl-tree-set-2-deletion/?ref=lbp` (дата обращения: 12.11.2020).

[5] *Chrono in C++*
URL: `https://www.geeksforgeeks.org/chrono-in-c/` (дата обращения: 02.11.2020).