Министерство образования и науки
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 6

Реализация словаря на основе красно-черного дерева. Реализация хэш-таблицы

по дисциплине «Теория графов»

Выполнил студент гр. 3530201/10002 _____ Кондраев Дмитрий Евгеньевич
Проверил _____ Востров Алексей Владимирович

Санкт-Петербург
2023

Содержание

Введение

Постановка задачи

1. Для выбранного текста на русском языке построить словарь на основе красно-черных деревьев. Реализовать функции добавления, удаления и поиска слова. Добавить функцию полной очистки словаря и загрузки/дополнения словаря из текстового файла.
2. Реализовать структуру данных — хэш-таблицу, для которой характерны операции: добавления, удаления и поиска. На ее основе реализовать приложение — словарь. В качестве хэш-функции, можно выбрать произвольную функцию, например, брать первую букву слова. Добавить функцию полной очистки словаря и загрузки/дополнения словаря из текстового файла.
3. Для выбранного текста на русском языке построить словарь на основе B+-деревьев. Реализовать функции добавления, удаления и поиска слова. Добавить функцию полной очистки словаря и дополнения словаря из текстового файла.
4. Минимальная реализация включает либо пункты 1 и 2, либо пункт 2 и 3. Улучшенная реализация — все три пункта.

Максимальная реализация также включает возможность оптимизации словаря (например, поиск потенциальных однокоренных слов в разных формах; самый простой критерий - $\frac{2}{3}$ длины слова). Выбор слов, которые следует оставить в словаре — за пользователем.

Использовать готовые решения из стандартной библиотеки не разрешается.

1 Математическое описание

1.1 Красно-черное дерево

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом *цвета* в каждом узле. Цвет узла может быть либо красным (`RED`), либо черным (`BLACK`). В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно *сбалансированными*.

Каждый узел дерева содержит атрибуты *color*, *key*, *left*, *right*. Если не существует дочернего узла по отношению к данному, соответствующий указатель принимает значение `NIL`. Значения `NIL` рассматриваются как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все «нормальные» узлы, содержащие поле ключа, становятся внутренними узлами дерева[1].

1.1.1 Инвариант красно-черного дерева

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим свойствам.

1. Каждый узел является либо ***красным***, либо ***черным***.
2. Корень дерева является черным узлом.
3. Каждый лист дерева (`NIL`) является черным узлом.
4. Если узел красный, то оба его дочерних узла черные.
5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов[1].

1.1.2 Поиск в красно-черном дереве

Поиск в красно-черном дереве происходит по общему правилу для бинарного дерева поиска.

Алгоритм стартует от корня дерева, на каждой итерации происходит проверка, соответствует ли ключ нашего узла ключу, который мы ищем.

1. Если *да*, то возвращается узел в качестве ответа.
2. Если *нет*, то происходит сравнение текущего значения ключа и искомого. В зависимости от того, больше или меньше, переходим соответственно в правое или левое поддерево. Алгоритм повторяется до тех пор, пока не дойдет до листа `NIL`.
3. Если ответ не найден, то возвращаем `NULL`.

Примеры выполнения данного алгоритма см. на рис. 1.

Сложность алгоритма: $\mathcal{O}(\log n)$, где n — количество узлов дерева[1].

1.1.3 Вставка элемента

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет `NIL` (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку (рис. 2).

Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

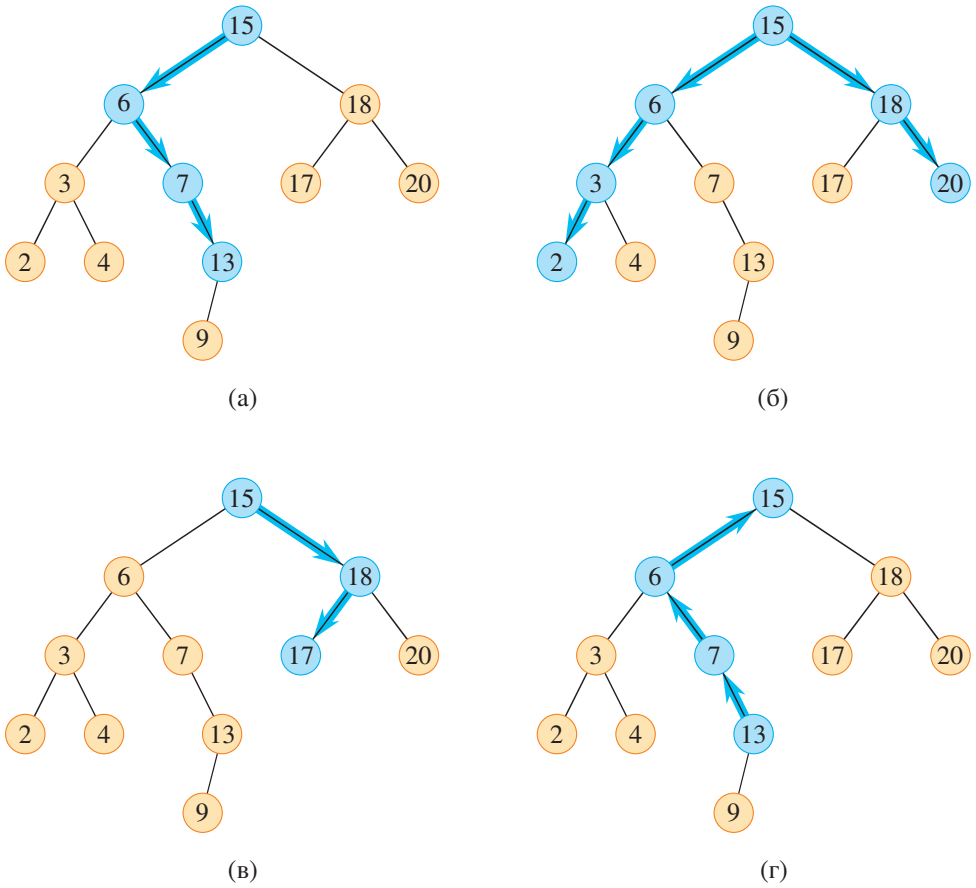


Рис. 1: Запросы к бинарному дереву поиска. Узлы и пути, по которым следует каждый запрос, окрашены в **синий цвет**. **(а)** Для поиска ключа 13 необходимо пройти путь 15 → 6 → 7 → 13 от корня. **(б)** Минимальным в дереве является ключ 2, который находится при следовании по указателям *left* от корня. **(б)** Максимальный ключ 20 достигается при следовании от корня по указателям *right*. **(в)** Последующим узлом после узла с ключом 15 является узел с ключом 17, поскольку это минимальный ключ в правом поддереве узла 15. **(г)** Узел с ключом 13 не имеет правого поддерева, так что следующим за ним является наименьший предок, левый наследник которого также является предком данного узла. В нашем случае это узел с ключом 15.

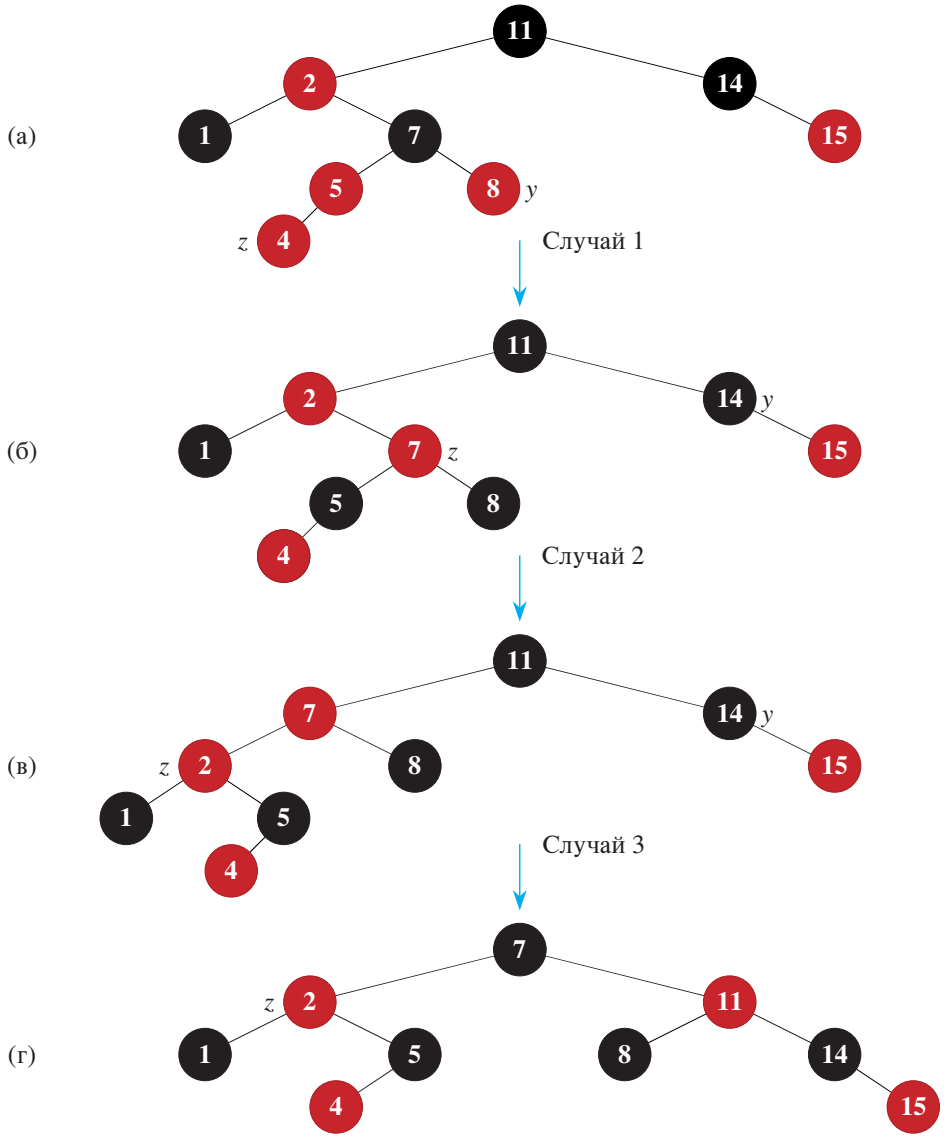


Рис. 2: Пример работы процедуры исправления после вставки. **(а)** Узел z после вставки. Поскольку и z и его родитель $z.p$ красные, наблюдается нарушение свойства 4. Поскольку «дядя» z, узел y, красный, в коде процедуры срабатывает случай 1. Мы перекрашиваем узлы и перемещаем указатель: вверх по дереву, получая результат, показанный в части (б). Вновь z и его родитель красные, но теперь «дядя» z, узел y, черный. Поскольку z является правым дочерним узлом по отношению к $z.p$ применим случай 2. Мы выполняем левый поворот, и полученное в результате дерево показано в части **(в)**. Теперь z является левым дочерним узлом своего родителя, так что применим случай 3. Перекрашивание и правый поворот дают показанное в части **(г)** корректное красно-черное дерево.

1. *«Дядя» этого узла тоже красный*. На рис. 3 показана ситуация, возникающая в случае 1, когда и $z.p$, и y красные узлы. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных узлов «отцы» черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2.

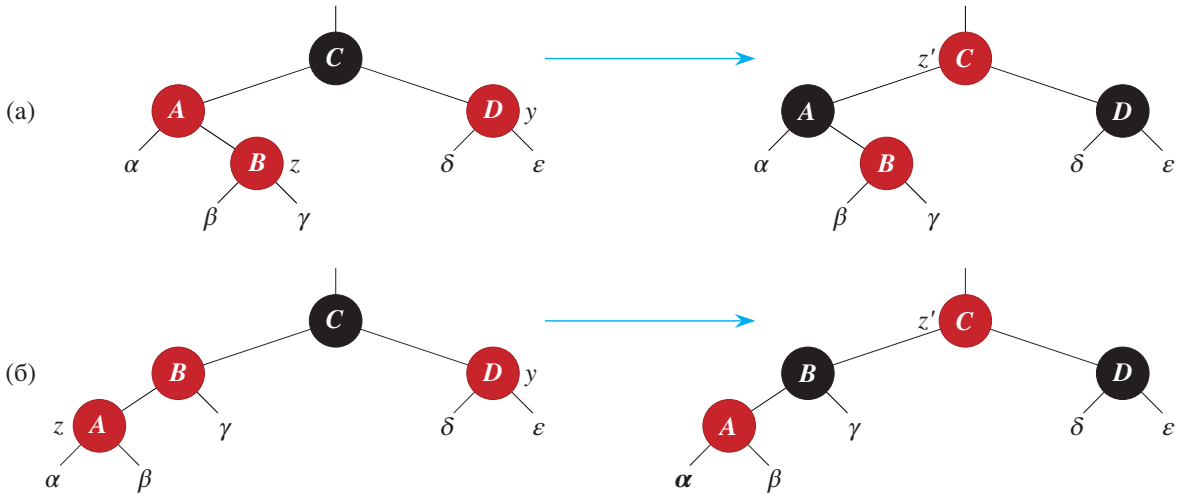


Рис. 3: Случай 1 процедуры исправления после вставки. Свойство 4 нарушено, поскольку и z, и его родительский узел $z.p$ красные. Мы предпринимаем одни и те же действия, когда **(а)** z является правым дочерним узлом и когда **(б)** z является левым дочерним узлом. Каждое из поддеревьев $\alpha, \beta, \gamma, \delta$ и ε имеет черный корень, и у каждого одна и та же черная высота. Код для случая изменяет цвета некоторых узлов, сохраняя свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Цикл продолжается с «дедом» $z.p.p$ узла в качестве нового узла z. Любое нарушение свойства 4 может теперь произойти только между новым (окрашенным в красный цвет) и его родителем, если он также красный.

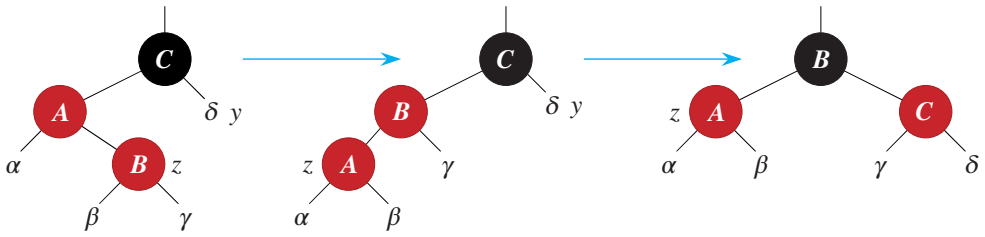
2. *«Дядя» чёрный.* В случаях 2 и 3 (рис. 4) цвет узла y, являющегося «дядей» узла z, черный. Эти два случая отличаются один от другого тем, что z является левым или правым дочерним узлом по отношению к родительскому узлу $z.p$. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.

Сложность алгоритма: также $\mathcal{O}(\log n)$.

1.1.4 Удаление элемента

При *удалении* узла могут возникнуть три случая в зависимости от количества её детей:

- Если у узла нет детей, то изменяем указатель на неё у родителя на `NIL`.
- Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этого узла.
- Если же имеются оба ребёнка, то находим узел со следующим значением ключа. У такого узла нет левого ребёнка (так как такой узел находится в правом поддереве исходного узла и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у узла есть левый ребенок). Удаляем уже этот узел описанным во втором пункте способом, скопировав её ключ в изначальный узел.



Случай 2 Случай 3

Рис. 4: Случаи 2 и 3 процедуры исправления после вставки. Как и в случае 1, свойство 4 нарушается либо случаем 2, либо случаем 3, поскольку z и его родительский узел $z.p$ красные. Каждое из поддеревьев α, β, γ и δ имеет черный корень (α, β и γ согласно свойству 4, δ — поскольку в противном случае мы получили бы случай 1), и каждое из них имеет одну и ту же черную высоту. Мы преобразуем случай 2 в случай 3 левым поворотом, который сохраняет свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Случай 3 приводит к определенному перекрашиванию и правому повороту, что также сохраняет свойство 5. Затем цикл завершается, поскольку свойство 4 удовлетворено: в одной строке больше нет двух красных узлов.

1.1.5 Балансировка дерева при удалении

Далее необходимо проверить балансировку дерева. Так как при удалении красного узла свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении черной. Рассмотрим ребенка удаленного узла.

1. *Брат w узла x — красный*. Если брат этого ребенка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в черный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

Если брат w текущего узла был черным, то получаем три случая (2, 3, 4).

2. *Оба ребенка узла w черные*. Красим брата в красный цвет и рассматриваем далее отца узла. Делаем его черным, это не повлияет на количество черных узлов на путях, проходящих через b, но добавит один к числу черных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного черного узла. Таким образом, после удаления узла черная глубина от отца этого узла до всех листьев в этом поддереве будет одинаковой.

3. *Если у брата w правый ребёнок чёрный, а левый красный*, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.

4. *Если у брата w правый ребёнок красный*, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня. Но у x теперь появился дополнительный чёрный предок: либо а стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

Сложность алгоритма удаления узла: $\mathcal{O}(\log n)$.

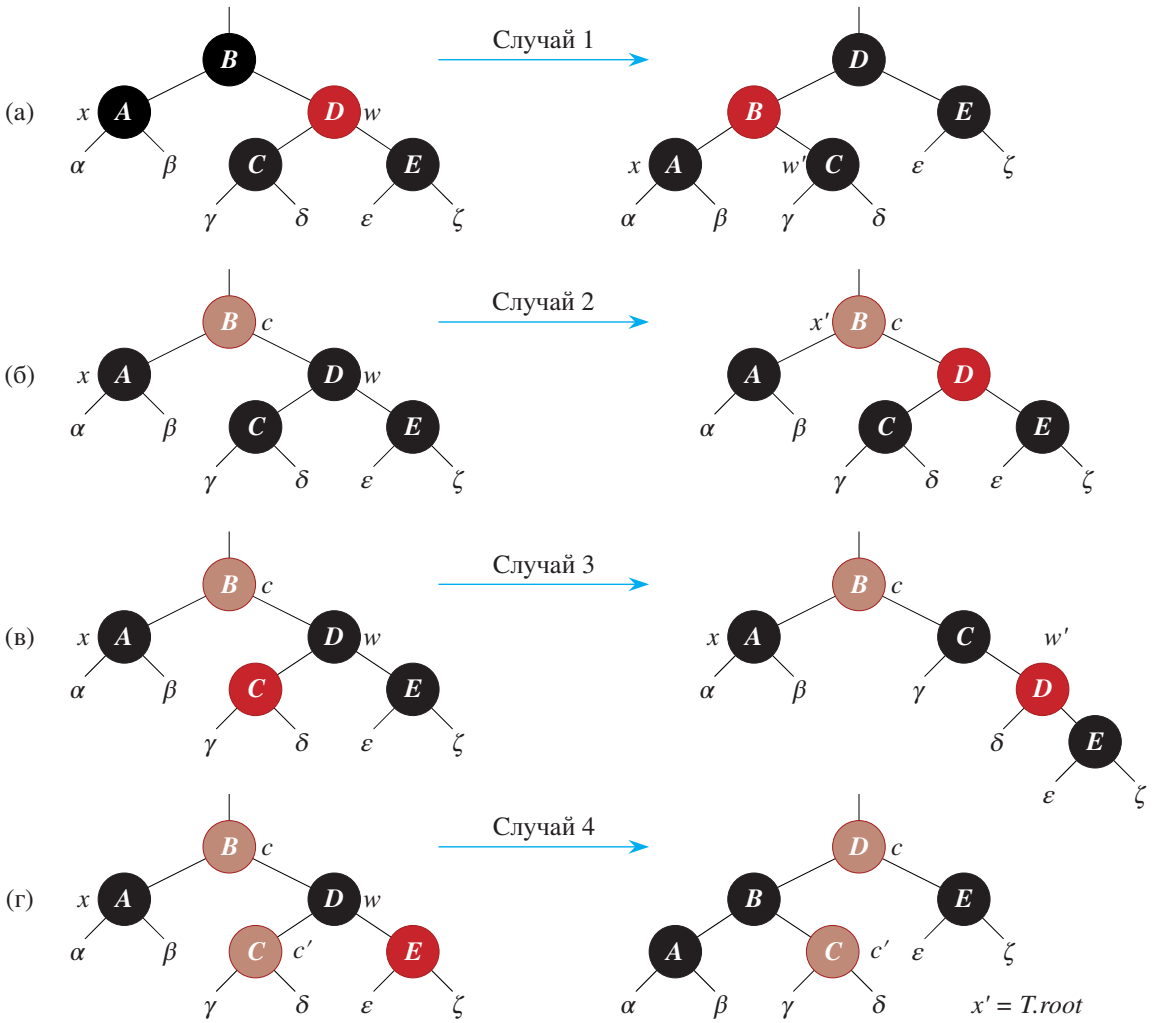


Рис. 5: Случаи при исправлении после удаления ключа. **Коричневым** обозначены узлы, чей цвет не важен.

1.2 Хэш-таблица

1.2.1 Определение

В случае *прямой адресации* элемент с ключом k хранится в ячейке k. При хэшировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем *хэш-функцию* h для вычисления ячейки для данного ключа k. Функция h отображает совокупность ключей U на ячейки хэш-таблицы $T[0..m-1]$:

$$h : U \to 0, 1, .., m-1,$$

где размер хэш-таблицы $m \ll |U|$. Мы говорим, что элемент с ключом k *хэшируется* в ячейку $h(k)$; величина $h(k)$ называется *хэш-значением* ключа k[1].

Возникает проблема: два ключа могут быть хэшированы в одну и ту же ячейку. Такая ситуация называется *коллизией*.

1.2.2 Разрешение коллизий с помощью цепочек

При разрешении коллизий с помощью цепочек мы помещаем все элементы, хэшированные в одну и ту же ячейку, в связанный список, как показано на рис. 6. Ячейка j содержит указатель на заголовок списка всех элементов, хэш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение `NIL`.

Словарные операции в хэш-таблице с использованием цепочек для разрешения коллизий реализуются очень просто с помощью операций над связными списками[1].

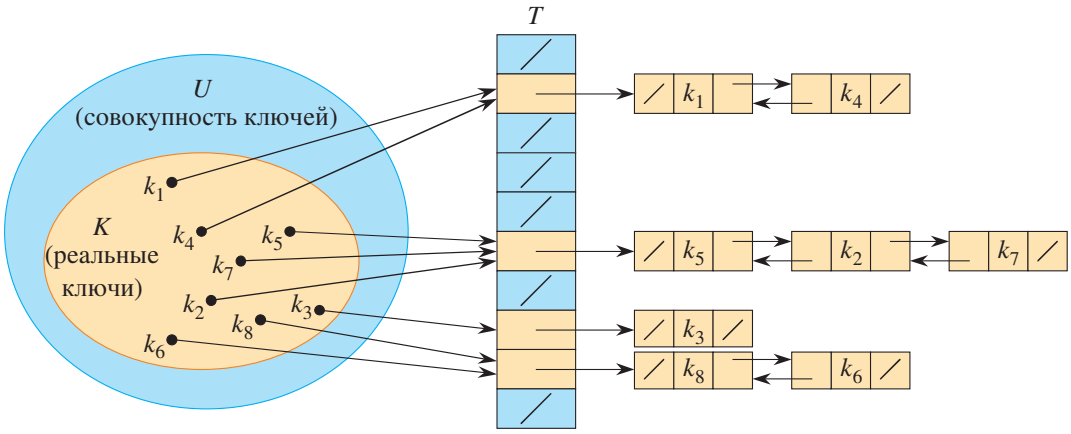


Рис. 6: Разрешение коллизий с помощью цепочек. Каждая ячейка хэш-таблицы $T[j]$ содержит (дву-) связный список всех ключей с хэш-значением j (возможно пустой).

1.2.3 Выбор хэш-функции

В качестве хэш-функции была выбрана стандартная хэш-функция языка Java для UTF-8 строк: String::hashCode[1]. Она вычисляется следующим образом:

$$H(s) \stackrel{\text{def}}{=} s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1},$$

[1] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html#hashCode()

где $s = s_0 s_1 \ldots s_{n-1}$ — строка, состоящая из 16-битных символов, $0 \leqslant s_i \leqslant 65535$. Значение вычисляется по правилам знаковой 32-битной арифметики.

$$H(s) \in [-2^{31}; 2^{31} - 1] \subset \mathbb{Z}.$$

Результат этой функции затем берется по модулю m, где m — размер хэш-таблицы:

$$j = h_m(s) \stackrel{\text{def}}{=} H(s) \mod m.$$

Например, если на вход подается строка `столбец`, то ее хэш-значение при $m = 53$ будет вычислено так:

$$1089 \cdot 31^6 + 1090 \cdot 31^5 + 1086 \cdot 31^4 + 1083 \cdot 31^3 + 1073 \cdot 31^2 + 1077 \cdot 31^1 + 1094 =$$

$$= 998\,733\,556\,292 = \text{E8 89 28 AA 44}_{16} \xrightarrow{\text{mod } 2^{32}} \text{89 28 AA 44}_{16} = -1\,993\,823\,676_{10} = 19 \pmod{53},$$

т. к. `с` = 1089, `т` = 1090,… `ц` = 1094. Для справки приведена таблица 1 кодов Unicode[2].

Таблица 1: Некоторые символы Unicode.

Код	Глиф	Описание	Десятичное значение
U+0431	б	Cyrillic Small Letter Be	1073
U+0435	е	Cyrillic Small Letter Ie	1077
U+043B	л	Cyrillic Small Letter El	1083
U+043E	о	Cyrillic Small Letter O	1086
U+0441	с	Cyrillic Small Letter Es	1089
U+0442	т	Cyrillic Small Letter Te	1090
U+0446	ц	Cyrillic Small Letter Tse	1094

2 Особенности реализации

Программа написана на языке Java 17.

2.1 Структуры данных

2.1.1 Динамическое множество

Интерфейс `Set` предоставляет набор методов, характерных для множества элементов. Этот интерфейс реализуют классы `HashSet` и `RedBlackSet`, которые делегируют соответствующие операции внутренним объектам (ассоциативным массивам) следующим образом:

- В качестве типа-значения используется `Object`.
- присутствие элемента обозначается значением `final Object PRESENT = new Object()`.
- отсутствие элемента обозначается значением `null`.

```
public interface Set<E> {
    // найти элемент
    boolean contains(@NotNull E element);
    // добавить элемент
    void add(@NotNull E element);
    // удалить элемента
    boolean remove(E element);
    // размер множества
    int size();
    // пустое ли множество
    boolean isEmpty();
    // опустошить множество
    void clear();
}
```

2.1.2 Двоичное дерево поиска

Интерфейс `Bst` представляет набор методов, которые должны быть двоичного дерева поиска со значениями в узлах.

Основные методы:

- `Value get(Key key)` — возвращает значение по заданному ключу или `null`, если такого ключа нет
- `void put(Key key, Value value)` — сохраняет заданное значение по указанному ключу
- `Value remove(Key key)` — удаляет заданный ключ (и связанное с ним значение, возвращая его)

Служебные методы:

- `int size()` — возвращает количество узлов в дереве.
- `void clear()` — очищает содержимое дерева.

```
public interface Bst<Key extends Comparable<Key>, Value> {
    @Nullable Value get(@NotNull Key key);

    default boolean containsKey(@NotNull Key key) {
        return get(key) != null;
    }

    void put(@NotNull Key key, @NotNull Value value);
```

```
    @Nullable Value remove(@NotNull Key key);

    int size();

    default boolean isEmpty() {
        return size() == 0;
    }

    void clear();
}
```

2.2 Красно-черное дерево

2.2.1 Поиск элемента

Функция `private @Nullable Value RedBlackBst::get(@Nullable Node node, @NotNull Key key)`

Вход функции Корень поддерева `node`, ключ `key`, по которому производится поиск в дереве.

Выход функции Значение по данному ключу; `null`, если ключа в дереве нет.

Описание работы функции Поиск реализован рекурсивно.

База рекурсии: пустое поддерево. Возвращаем `null`.

Дальше сравниваем ключ искомого элемента, с ключом корня поддерева.

По результату сравнения перемещаемся вправо или влево по дереву.

Если ключ совпал, то возвращем значение текущего узла.

Исходный код

```
private Value get(@Nullable Node node, @NotNull Key key) {
    if (node == null) {
        return null;
    }
    int compare = key.compareTo(node.key);
    if (compare < 0) {
        return get(node.left, key);
    }
    if (compare > 0) {
        return get(node.right, key);
    }
    return node.value;
}
```

2.2.2 Добавление элемента в дерево

Функция `private Node RedBlackBst::put(@Nullable Node node, @NotNull Key key, @NotNull Value value)`

Вход функции Корень поддерева `node`, ключ `key` и значение `value`, вставляемые в дерево.

Выход функции Изменяемый узел дерева.

Описание работы функции При добавлении элемента первым делом проверяем не пустое ли текущее поддерево. Если дерево не содержит элементов, то перекрашиваем добавленный элемент в черный и делаем его корнем дерева.

Находим потенциального родителя для добавленного элемента, далее по ключу добавляем его слева или справа (если ключ меньше, чем ключ родителя, тогда слева, если больше — справа).

Если найден элемент с совпадающим ключем, происходит перезапись значения.

С помощью дополнительных функций поворота и перекраски узлов сохраняем свойства красно-черных деревьев.

Исходный код

```
private Node put(@Nullable Node node, @NotNull Key key, Value value) {
    if (node == null) {
        size++;
        return new Node(key, value);
    }
    final int compare = key.compareTo(node.key);
    if (compare < 0) {
        node.left = put(node.left, key, value);
    } else if (compare > 0) {
        node.right = put(node.right, key, value);
    } else {
        node.value = value;
    }
    return fixUp(node);
}
```

2.2.3 Удаление элемента

Функция `Value RedBlackBst::remove(@Nullable Node node, @NotNull Key key)`

Вход функции Корень поддерева `node`, ключ `key` для удаления.

Выход функции Удаляемый узел дерева, если по ключу узел не найден, то `null`.

Описание работы функции Функция удаления является рекурсивной. Сначала по значению находим узел, который нужно удалить. При удалении узла необходимо учитывать его цвет, а точнее, если узел окрашен в черный цвет, то вызывается метод, который следит, чтобы черная высота дерева оставалась неизменной для каждой ветви.

Исходный код

```
private Node remove(@Nullable Node node, @NotNull Key key) {
    if (node == null) {
        return null;
    }
    if (key.compareTo(node.key) < 0) {
        if (node.left == null) {
            return fixUp(node);
        }
```

```
        if (!isRed(node.left) && !isRed(node.left.left)) {
            node = node.moveRedLeft();
        }
        node.left = remove(node.left, key);
        return fixUp(node);
    }
    if (isRed(node.left)) {
        node = node.rotateRight();
        node.right = remove(node.right, key);
        return fixUp(node);
    }
    if (key.compareTo(node.key) == 0 && node.right == null) {
        deleted = node.value;
        size--;
        return null;
    }
    if (node.right != null && !isRed(node.right) && !isRed(node.right.left)) {
        node = node.moveRedRight();
    }
    if (key.compareTo(node.key) == 0) {
        deleted = node.value;
        size--;
        Node min = node.right.min();
        node.value = min.value;
        node.key = min.key;
        node.right = removeMin(node.right);
    } else {
        node.right = remove(node.right, key);
    }
    return fixUp(node);
}
```

2.2.4 Левый и правый повороты

Функции `@NotNull private Node RedBlackBst::Node::rotateLeft()`

`@NotNull private Node RedBlackBst::Node::rotateRight()`

Вход функции Текущий узел (`this`).

Выход функции Узел, оказывающийся после поворота на месте текущего.

Описание работы функции *Повороты* — локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска.

При выполнении левого поворота в узле `this` предполагается, что его правый дочерний узел `x` не является листом `null`; `this` может быть любым узлом дерева, правый дочерний узел которого — не `null`. Левый поворот выполняется “вокруг” связи между `this` и `x`, делая у новым корнем поддерева, левым дочерним узлом которого становится `this`, а бывший левый потомок узла `x` — правым потомком `this`.

Правый поворот выполняется аналогично.

Исходный код

```
@NotNull
private Node rotateLeft() {
```

```
    Node x = right;
    right = x.left;
    x.left = this;
    x.color = color;
    color = RED;
    return x;
}
```

```
@NotNull
private Node rotateRight() {
    Node x = left;
    left = x.right;
    x.right = this;
    x.color = color;
    color = RED;
    return x;
}
```

2.2.5 Балансировка красно-черного дерева

Функция `private Node RedBlackBst::fixUp(@NotNull Node node)`

Вход функции Корень исправляемого поддерева node.

Выход функции Узел, находящийся на месте узла node после балансировки.

Описание работы функции Функция производит левый или правый поворот, или меняет цвет, в зависимости от того, какие цвета у текущего узла и его потомков. Вызов данной функции в рекурсивных функциях добавления и удаления восстанавливает инвариант красно-черного дерева.

Исходный код

```
private Node fixUp(@NotNull Node node) {
    if (isRed(node.right) && !isRed(node.left)) {
        node = node.rotateLeft();
    }
    if (isRed(node.left) && isRed(node.left.left)) {
        node = node.rotateRight();
    }
    if (isRed(node.left) && isRed(node.right)) {
        node.flipColors();
    }
    return node;
}
```

2.2.6 Полная очистка дерева

Функция `@Override public void RedBlackBst::clear()`

Вход функции Красно-черное дерево.

Выход функции Дерево не содержит узлов.

Описание работы функции Из-за того, что JVM производит сборку мусора автоматически, достаточно обнулить указатель на корневую вершину дерева.

Исходный код

```
@Override
public void clear() {
    root = null;
    size = 0;
}
```

2.3 Хэш-таблица

Реализация хэш-таблицы с разрешением коллизий методом цепочек.

Основные методы:

- `Value get(Key key)` — возвращает значение по заданному ключу или `null`, если такого ключа нет
- `void put(Key key, Value value)` — сохраняет заданное значение по указанному ключу
- `Value remove(Key key)` — удаляет заданный ключ (и связанное с ним значение, возвращая его).

Служебные методы:

- `int size()` — вовзращает количество ключей.
- `boolean isEmpty()` — проверка на пустое множество.

2.3.1 Односвязный список

Элемент таблицы — объект типа `Node` — реазизует односвязный список, хранящий в своих элементах ключ `key`, хэш `hash` и значение `value`.

Исходный код класса приведен ниже.

```
private class Node {
    final Key key;
    final int hash;
    Value value;
    Node next;

    Node(int hash, Key key, Value value, Node next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    Node(int hash, Key key, Value value) {
        this(hash, key, value, null);
    }

    Value get(int hash, Key key) {
        Node node = findOrGetLast(hash, key);
        if (!node.isLast() || node.matches(hash, key)) {
            return node.value;
        }
        return null;
```

```
    }

    void put(int hash, Key key, Value value) {
        Node node = this;
        while (!node.isLast() && !node.matches(hash, key)) {
            node = node.next;
        }
        if (!node.isLast() || node.matches(hash, key)) {
            node.value = value;
            return;
        }
        node.next = new Node(hash, key, value);
        size++;
    }

    Value remove(int hash, Key key) {
        Node node = this;
        while (!node.next.isLast() && !node.next.matches(hash, key)) {
            node = node.next;
        }
        if (!node.next.isLast() || node.next.matches(hash, key)) {
            size--;
        }
        Value previous = node.next.value;
        node.next = node.next.next;
        return previous;
    }

    Node findOrGetLast(int hash, Key key) {
        Node node = this;
        while (!node.isLast() && !node.matches(hash, key)) {
            node = node.next;
        }
        return node;
    }

    boolean isLast() {
        return next == null;
    }

    boolean matches(int hash, Key key) {
        return this.hash == hash && this.key.equals(key);
    }
}
```

2.3.2 Поиск элемента

Функция `@Override public @Nullable Value HashTableImpl::get(@NotNull Key key)`

Вход функции Ключ, по которому производится поиск

Выход функции Значение, если в хэш-таблице есть этот ключ, `null` в ином случае.

Описание работы функции Для ключа вычисляется его хэш-значение и индекс по нему. Затем производится поиск в связном списке `bucket`.

Исходный код

```
@Override
public @Nullable Value get(@NotNull Key key) {
    if (table == null) {
        return null;
    }
    final int hash = key.hashCode();
    final Node bucket = table[index(hash)];
    return bucket == null ? null : bucket.get(hash, key);
}
```

2.3.3 Добавление элемента

Функция `@Override public void HashTableImpl::put(@NotNull Key key, @NotNull Value value)`

Вход функции Ключ и значение для добавления в таблицу.

Выход функции Ключ и значение добавлены в таблицу. Если этот ключ уже присутствует в таблице, то значение заменяется.

Описание работы функции Если размер таблицы превысил допустимый размер (⌊ `LOAD_FACTOR * capacity()`), то происходит Увеличение размера таблицы.

Для ключа вычисляется его хэш-значение и индекс по нему.

Затем производится добавление в связный список `table[index]` или создание списка с одним элементом, если он пуст.

Исходный код

```
@Override
public void put(@NotNull Key key, @NotNull Value value) {
    if (table == null || size >= LOAD_FACTOR * capacity()) {
        table = resize();
    }
    final int hash = key.hashCode();
    final int index = index(hash);
    if (table[index] == null) {
        table[index] = new Node(hash, key, value);
        size++;
        return;
    }
    table[index].put(hash, key, value);
}
```

2.3.4 Удаление элемента

Функция `@Override public @Nullable Value HashTableImpl::remove(@NotNull Key key)`

Вход функции Ключ, по которому нужно удалить элемент.

Выход функции Значение по данному ключу до удаления, если его не было, то `null`.

Описание работы функции Для ключа вычисляется его хэш-значение и индекс по нему.

Если найденный список из одного элемента и он совпадаает как по хэшу, так и по значению, то список становится пустым (`null`).

Если нет, то производится удаление в связном списке.

Исходный код

```
@Override
public @Nullable Value remove(@NotNull Key key) {
    if (table == null) {
        return null;
    }
    final int hash = key.hashCode();
    final int index = index(hash);
    if (table[index] == null) {
        return null;
    }
    if (table[index].isLast() && table[index].matches(hash, key)) {
        final Value previous = table[index].value;
        table[index] = null;
        size--;
        return previous;
    }
    return table[index].remove(hash, key);
}
```

2.3.5 Увеличение размера таблицы

Функция `private Node[] HashTableImpl::resize()`

Вход функции Хэш-таблица.

Выход функции Новый массив увеличенного размера, содержащий все предыдущие списки на правильных местах.

Описание работы функции Размер таблицы увеличивается по следующему правилу:

Изначальный размер таблицы `DEFAULT_INITIAL_CAPACITY = 53`. С каждым следующим вызовом `resize` размер таблицы устанавливается равным следующему элементу массива `PRIME_CAPACITIES`. Если достигли последнего, то больше размер не увеличивается.

```
    private static final int[] PRIME_CAPACITIES = {
            53, 97, 193, 389, 769, 1543, 3079,
            6151, 12289, 24593, 49157, 98317, 196613, 393241,
            786433, 1572869, 3145739, 6291469, 12582917, 25165843, 50331653,
            100663319, 201326611, 402653189, 805306457, 1610612741
    };

    private static final int DEFAULT_INITIAL_CAPACITY = PRIME_CAPACITIES[0];
```

В новую таблицу заносятся списки `bucket` из массива предыдущего размера, и индекс списков перевычисляется с учетом нового размера.

Исходный код

```
private Node[] resize() {
    if (capacityIndex >= PRIME_CAPACITIES.length) {
        return table;
    }
    if (table != null) {
        capacityIndex++;
    }
    final int newCapacity = PRIME_CAPACITIES[capacityIndex];
    // необходимо, так как Node === HashTableImpl<Key, Value>.Node,
    // а массивы с generic типом компонента вне закона (JLS 15.10.1)
    @SuppressWarnings("unchecked")
    final Node[] newTable = (Node[]) new HashTableImpl<?, ?>.Node[newCapacity];
    if (table == null) {
        return newTable;
    }
    for (Node bucket : table) {
        for (Node node = bucket, next; node != null; node = next) {
            next = node.next;
            final int index = Math.floorMod(node.hash, newCapacity);
            node.next = newTable[index];
            newTable[index] = node;
        }
    }
    return newTable;
}
```

2.4 Общие функции

Благодаря интерфейсу `Set`, две различные реализации динамического множества могут использоваться одним и тем же (т. н. полиморфным) кодом.

2.4.1 Добавление в словарь текста из файла

Функция `private void DictionaryApp::addFromFile() throws FileNotFoundException`

Вход функции Имя файла (пользовательский ввод), содержимое файла

Выход функции Слова из файла добавлены в словарь `set`.

Описание работы функции Текст файла разбивается по пробельным и пунктуационным знакам на слова. Эти слова добавляются в словарь.

Исходный код

```
private void addFromFile() throws FileNotFoundException {
    out.println("Введите путь до файла без пробелов:");
    Scanner input = new Scanner(new
     ↪  File(askWord())).useDelimiter("(?U)[\\p{Punct}\\s&&[^-']]+");
    input.forEachRemaining(set::add); // прочитать
}
```

3 Результаты работы программы

3.1 Красно-черное дерево

На рис. 7-12 представлены типичные действия над красно-черным деревом.

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> import
Введите путь до файла без пробелов:
> ~/Desktop/onegin.txt
В словаре 10522 эл.
> find
Введите слово, чтобы найти его в словаре:
> Онегин
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> Ленского
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> дуэль
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> дуэлянт
Слова нет в словаре.
```

Рис. 7: Построение красно-черного дерева на основе выбранного файла (текст романа А.С. Пушкина «Евгений Онегин»)

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> красно-черное
В словаре 1 эл.
> find
Введите слово, чтобы найти его в словаре:
> красно-черное
Слово есть в словаре.
```

Рис. 8: Добавление слова

На рис. 13 представлено красно-черное дерево, построенное программой для данного множества слов.

3.2 Хэш-таблица

На рис. 14-19 представлены типичные действия над хэш-таблицей.

На рис. 20 представлена хэш-таблица, построенная программой для данного множества слов.

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> красно-черное
В словаре 1 эл.
> remove
Введите слово, чтобы удалить из словаря:
> красно-черное
Слово удалено из словаря.
> find
Введите слово, чтобы найти его в словаре:
> красно-черное
Слова нет в словаре.
```

Рис. 9: Успешное удаление слова

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> красно-черное
В словаре 1 эл.
> remove
Введите слово, чтобы удалить из словаря:
> черное
Слова в словаре не было.
```

Рис. 10: Удаление отсутствующего слова

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> красно-черное красное черное
В словаре 3 эл.
> find
Введите слово, чтобы найти его в словаре:
> красное
Слово есть в словаре.
```

Рис. 11: Успешный поиск слова

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> красно-черное красное черное
В словаре 3 эл.
> find
Введите слово, чтобы найти его в словаре:
> зеленое
Слова нет в словаре.
```

Рис. 12: Поиск отсутствущего слова

```
Выберите, с какой структурой данных вы хотите работать:
> rb
> add
Введите слово, чтобы добавить в словарь:
> великаны не так просты как кажется великаны как луковицы лук многослоен я тоже слой за слоем
 ↪  мы многослойные
В словаре 16 эл.
> show
Структура:
```

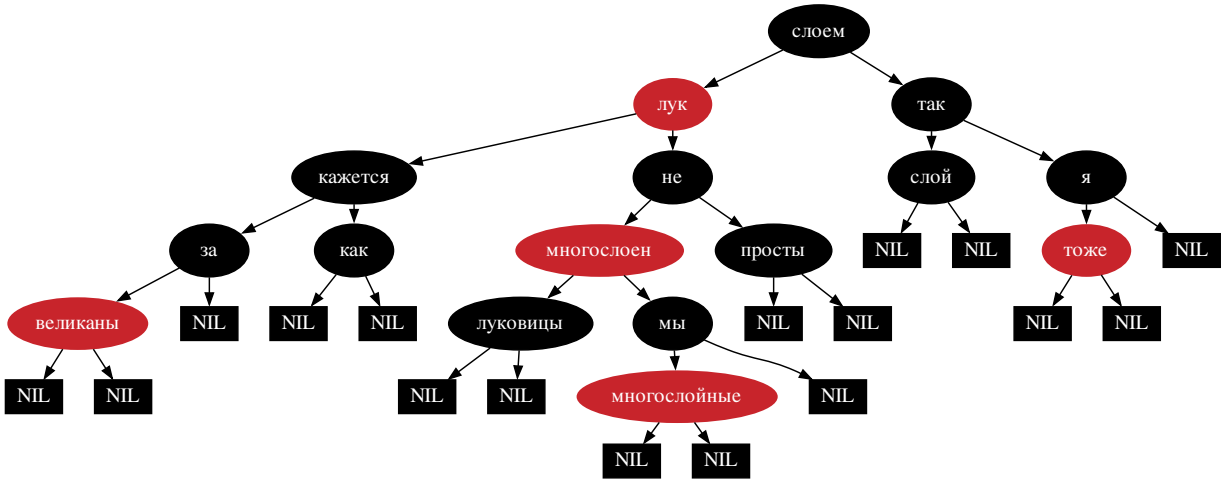


Рис. 13: Красно-черное дерево, построенное для множества слов. Порядок для этих слов такой: `великаны` < `за` < `кажется` < `как` < `лук` < `луковицы` < `многослоен` < `многослойные` < `мы` < `не` < `просты` < `слоем` < `слой` < `так` < `тоже` < `я`.

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> import
Введите путь до файла без пробелов:
> ~/Desktop/onegin.txt
В словаре 10522 эл.
> find
Введите слово, чтобы найти его в словаре:
> Онегин
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> Ленского
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> дуэль
Слово есть в словаре.
> find
Введите слово, чтобы найти его в словаре:
> дуэлянт
Слова нет в словаре.
```

Рис. 14: Построение хэш-таблицы на основе выбранного файла (текст романа А.С. Пушкина «Евгений Онегин»)

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> хэш-таблица
В словаре 1 эл.
> find
Введите слово, чтобы найти его в словаре:
> хэш-таблица
Слово есть в словаре.
```

Рис. 15: Добавление слова

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> хэш-таблица
В словаре 1 эл.
> remove
Введите слово, чтобы удалить из словаря:
> хэш-таблица
Слово удалено из словаря.
> find
Введите слово, чтобы найти его в словаре:
> хэш-таблица
Слова нет в словаре.
```

Рис. 16: Успешное удаление слова

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> хэш-таблица
В словаре 1 эл.
> remove
Введите слово, чтобы удалить из словаря:
> таблица
Слова в словаре не было.
```

Рис. 17: Удаление отсутствующего слова

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> хэш-таблица хэш таблица
В словаре 3 эл.
> find
Введите слово, чтобы найти его в словаре:
> хэш
Слово есть в словаре.
```

Рис. 18: Успешный поиск слова

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> хэш-таблица хэш таблица
В словаре 3 эл.
> find
Введите слово, чтобы найти его в словаре:
> хэш-значение
Слова нет в словаре.
```

Рис. 19: Поиск отсутствущего слова

```
Выберите, с какой структурой данных вы хотите работать:
> hash
> add
Введите слово, чтобы добавить в словарь:
> великаны не так просты как кажется великаны как луковицы лук многослоен я тоже слой за слоем
 ↪  мы многослойные
В словаре 17 эл.
> show
Структура:
[ 5] → "великаны" → ∅
[ 6] → "тоже" → ∅
[11] → "луковицы" → ∅
[12] → "кажется" → "многослоен" → ∅
[18] → "как" → "за" → ∅
[20] → "слой" → ∅
[21] → "так" → ∅
[31] → "лук" → ∅
[39] → "просты" → ∅
[41] → "мы" → ∅
[43] → "я" → "слоем" → ∅
[50] → "не" → ∅
[52] → "многослойные" → ∅
```

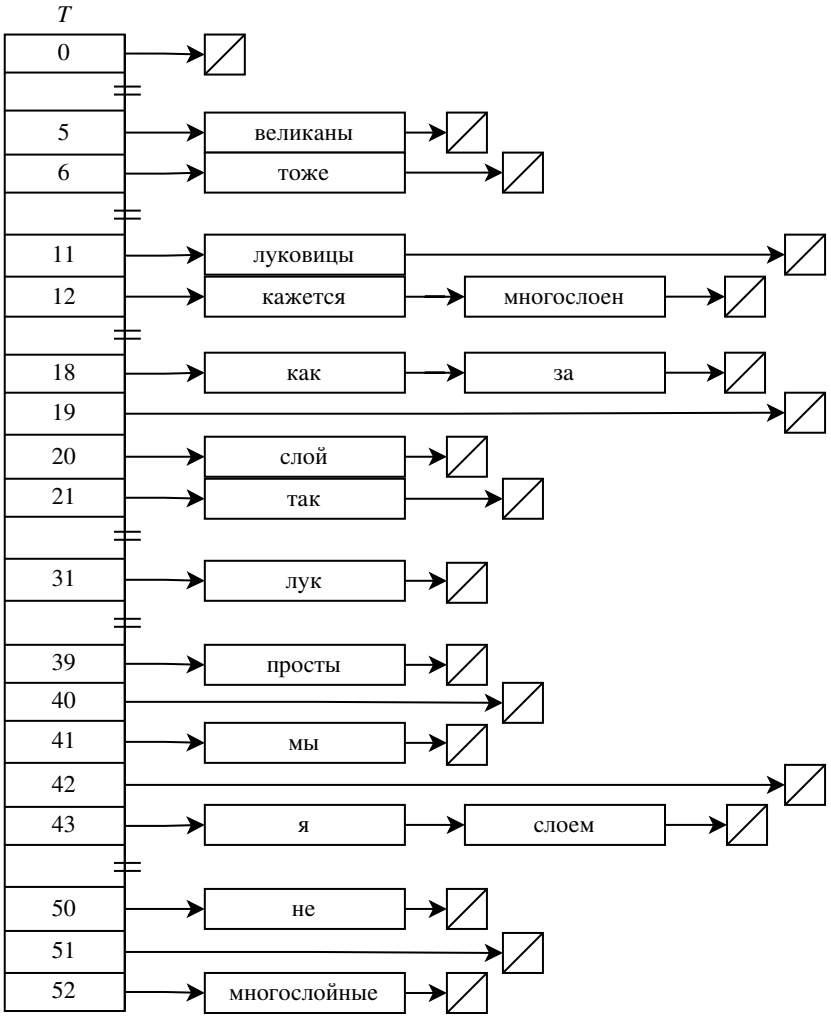


Рис. 20: Хэш-таблица для данного множества слов. Элементы с пустыми списками опущены.

Заключение

Результат выполнения данной лабораторной работы — приложение «Словарь». В качестве реализации было использовано два варианта: *красно-черное дерево* и *хэш-таблица*.

Приложение позволяет выполнять такие операции с элементами словаря, как:

- пополнение словаря посредством чтения текста из файла;
- добавление нового слова в словарь;
- удаление выбранного слова из словаря;
- поиск выбранного слова по словарю;
- полная очистка словаря.

Использованные структуры позволяют эффективно (по времени работы) реализовывать данные операции.

Выбор структуры данных для эффективной реализации динамического множества должен учитывать следующие аспекты:

- Эффективность красно-черного дерева достигается благодаря балансировке при восстановлении инварианта при вставке и удалении.
- Эффективность хэш-таблицы — благодаря свойствам используемой хэш-функции: малое время работы, желательно $\mathcal{O}(1)$ и равномерность (малое количество коллизий).
- Операции над красно-черным деревом имеют сложность $\mathcal{O}(\log n)$ в худшем случае.
- В среднем случае сложность операций над хэш-таблицей $\mathcal{O}(1)$, но в худшем — $\mathcal{O}(n)$.
- Красно-черное дерево требует наличие отношения порядка для ключей, а хэш-таблица — хэш-функции.

Достоинства программы:

- классы `HashTableImpl` и `RedBlackBst` можно использовать не только для динамических множеств, как это сделано в данной работе, но и для ассоциативных массивов.
- используемое отношение порядка при построении красно-черного дерева определяется типом ключа, как используемая хэш-функция в реализации хэш-таблицы, это позволяет использовать в качестве ключа не только строку, но и любой другой подходящий тип.
- загрузка данных осуществляется по адресу, предоставляемому пользователем.

Недостатки программы:

- реализация красно-черного дерева не предусматривает обратных ссылок на родителя, как и реализация хэш-таблицы не предусматривает обратной ссылки на предыдущий элемент. эта особенность замедляет операцию удаления узла.
- приведенные реализации структур данных бесполезны в практическом смысле (но не в учебном), так как в стандартной библиотеке уже определены такие классы как `TreeSet`[2] и `HashSet`[3], реализующие те же структуры данных.

Масштабирование программы:

- программу можно расширить, добавив, например реализацию динамического множества на базе B-деревьев.
- Критерий совпадения слов можно определить так, чтобы разные формы одного слова считались одинаковыми, например, `фломастеров` = `фломастерами` ≠ `мастерами`. Для

[2] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeSet.html
[3] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashSet.html

слов русского языка точное решение этой задачи сложное.

- Обе реализации никак не предусматривают одновременные операции записи и операции чтения. Программа может быть дополнена необходимыми для этого критическими секциями.

Список использованных источников

1. Кормен, Т.Х. Алгоритмы: построение и анализ. – М.: ООО «И. Д. Вильямс», 2013. – С. 1328.
2. Официальная таблица символов Консорциума Юникода [Электронный ресурс]. – URL: http://www.unicode.org/charts/PDF/U0400.pdf (дата обращения: 10.06.2023).