

## Задание 4 для первого семестра 2 курса

### Общие замечания

Задания этого раздела предполагают реализацию множеств с быстрым доступом (быстрые деревья). Соответственно тесты должны показывать, что использование таких реализаций действительно эффективно, т.е. должны предусматривать работу с большими наборами данных. Например, в дерево добавляется несколько миллионов элементов, потом производится поиск каждого элемента, в процессе элементы добавляются или удаляются по ходу работы. Проверяется, что добавленные элементы действительно добавились, а удаленные действительно удалились.

Кроме этого, для деревьев нужно проверить, что они действительно имеют логарифмическую глубину. То есть, написать отдельную функцию, которая вычисляет глубину дерева, и протестировать как меняется глубина при добавлении большого количества элементов.

И еще надо сравнить эффективность своей реализации с библиотечной реализацией STL map (или set). Так как библиотечная реализация предполагает универсальность использования, то в некоторых задачах она может оказаться менее эффективной, чем специально написанная реализация. Поэтому интересно сравнить как и насколько различается время работы вашей реализации и библиотечной реализации в разных случаях, особенно если удастся обнаружить пример, где ваша реализация будет работать эффективнее.

### Отображения

Требуется построить параметризованный (template) класс, который реализует отображение где ключом является один класс, а значением — некоторый другой класс.

```
template <class key, class value>
```

Интерфейс отображения должен поддерживать следующие операции:

- добавить пару (ключ, значение);
- искать значение по указанному ключу;
- доступ к значению элемента дерева по указанному ключу;
- удалить ключ и соответствующее значение;
- получить количество хранящихся ключей;
- итератор по множеству пар (ключ, значение).

Доступ к значению элемента можно реализовать через итератор, т.е. функция поиска возвращает итератор, и он уже обеспечивает доступ к значению через операцию \* или ->.

В качестве примеров и тестов можно рассмотреть отображение строк на целые числа, отображения строк на строки (как в словаре), отображение пар чисел (координат точки) на некоторый указатель и т.п. Тесты должны предусматривать возможность загрузки множества ключей-значений из файла или их автоматической генерации.

Узел дерева может быть реализован внешним или внутренним классом, но обязательное требование — сокрытие его членов (значения, указатели) для не относящихся к дереву компонент программы.

Варианты заданий различаются способом реализации отображения и итераторов, которые могут быть либо внешними (EXternal), либо внутренними (INTernal).

1. Сбалансированное AVL дерево поиска (рекурсивные реализации, left-right).
2. Сбалансированное AVL дерево поиска (нерекурсивные реализации, left-right-parent).
3. Красно-черное дерево поиска (нерекурсивная реализация).
4. 2-3-дерево (рекурсивные реализации, нет указателя parent, однородные 3-вершины).
5. 2-3-дерево (нерекурсивные реализации, есть указатель parent, однородные 3-вершины).
6. 2-3-дерево (рекурсивные реализации, нет указателя parent, 3-вершина наследуется от 2 вершины).
7. 2-3-дерево (нерекурсивные реализации, есть указатель parent, 3-вершина наследуется от 2 вершины).
8. В-дерево (любая удобная реализация).

Итератор должен реализовывать двунаправленный проход по множеству элементов в смысле упорядоченности их ключей.

Выполнение задания включает следующие этапы.

а) Реализация быстрого дерева с параметризацией `template<class Key, class Value>`. Тестирование на простых типах данных на добавление, поиск, удаление.

- б) Тестирование итератора. Проверка глубины дерева в зависимости от количества элементов.
- с) Нагрузочный тест с большим количеством элементов.
- д) Сравнение с такими же тестами для `std::map`.
- е) Решение тестовой “практической” задачи.

Таковыми задачами могут быть, в частности,

**А** — частотный словарь: для большого текста определить сколько раз в нем встречается каждое слово. (отображение `key` - строка, `value` - число).

**В** — конфигурационный файл: в файле записаны строки типа “имя = значение” (целое, вещественное, строковое), надо в программе быстро получать соответствующее значение по данному имени. (отображение `key` - строка, `value` - строка и получение значения типа `GetInt`, `GetDouble` и т.д., подробности в рабочем порядке).

**С** — географическая карта: на плоскости есть множество точек с целочисленными координатами и каждой точке приписано некоторое “слово” — название места. Надо быстро получать координаты по слову либо слово по координатам. (прямое и обратное отображения `key` - строка, `value` - координаты)

**Д** — двумерное множество точек внутри заданного прямоугольника; поиск точек множества в заданной прямоугольной окрестности заданной точки. По координате `X` разбиваем множество на несколько полос одинаковой ширины. В каждой полосе храним подмножество точек в виде дерева, упорядоченного по `Y`. Поиск точки — это фактически объединение простейшей хеш-функции по `X` с подмножествами в виде дерева по `Y`.

**Е** — база данных: есть массив записей базы, нужно добавлять, удалять, искать записи по заданному ключу, которым может быть значение некоторого поля записи базы данных.

Суть и конкретная постановка второго этапа обсуждается в рабочем порядке после выполнения первого этапа.

Конкретное задание идентифицируется примерно такой записью:

4-int-A — вариант дерева 4, внутренний итератор, тестовая задача A

2-ext-E — вариант дерева 2, внешний итератор, тестовая задача E

и т.п.