` Лабораторная работа №2 по курсу «Программирование (объектно-ориентированное программирование)»

Техническое задание

1. Постановка задачи

Написать программу на языке C++, реализующую следующие алгоритмы:

1) Построение алфавитного указателя по заданному тексту и размеру страницы (размер страницы задается в словах).
2) Построение разреженной матрицы по заданной последовательности (предполагается, что в последовательности содержится 80-85% нулевых элементов).

2. Функциональные требования

2.1. Программа должна строить по тексту и размеру одной страницы строить специальный тип данных, алфавитный указатель (**AlphabetPointer**), представляющий собой множество объектов типа пар «ключ-значение», в котором ключ является словом, встречающимся в тексте, а значением последовательность целых чисел, обозначающий на каких страницах встречается указанное слово.

2.2. Программа должна преобразовывать последовательность (**Sequence**) в разреженный вектор (**SparseSeq**), являющийся набором пар типа «ключ-значение», в которых ключом является индекс элемента в исходной последовательности, а значением непосредственно сам элемент исходной последовательности. При этом SparseSeq не должен хранить в качестве значений нулевые элементы (подразумевая, что все элементы, у которых нет ключа, являются нулевыми).

2.3. Все основные типы данных покрыть unit-тестами. Программа должна иметь возможность по вводу пользователя проверять данные тесты.

2.4. Программа должна иметь консольный пользовательский интерфейс, имитирующий командную оболочку (в качестве ввода пользователю предлагается ввести команду для взаимодействия с программой).

3. Требования к типам данных

3.1. Базовый тип данных, **DictionaryTree<Key, Value>**, должен представлять из себя модификацию бинарного дерева поиска, в каждом узле которого должны храниться ключ и значение. Сравнение элементов данного дерева производить по

ключу.

Краткая спецификация DictionaryTree:

Название	Сигнатура	Описание
Атрибуты		
Ключ	`Key selfKey;`	Ключ узла дерева.
Значение	`Value value;`	Значение узла дерева.
Правое поддерево	`DictionaryTree *right;`	Указатель на правое поддерево.
Левое поддерево	`DictionaryTree *left;`	Указатель на левое поддерево.
Методы		
Конструктор инициализации узла	`DictionaryTree(Key k, Value v);`	Создает дерево с заданным ключом и значением.
IsEmpty	`bool isEmpty();`	Возвращает true, если дерево пустое, false в противном случае.
Add	`void add(Key k, Value val);`	Добавляет новую пару в дерево. Если такой ключ уже есть в дереве – исключение.
Get	`Value get(Key key);`	Возвращает значение элемента по заданному ключу. Исключение в случае отсутствия такого ключа.
Contains	`bool contains(Key key);`	Возвращает true, если такой ключ существует в дереве, false в противном случае.
Remove	`void remove(Key key);`	Удаляет данный узел из дерева.
Size	`int size();`	Возвращает количество элементов в дереве.
UpdateValue	`void updateValue(Key key, Value newValue);`	Если ключ key есть в дереве, то обновляет значение, отвечающее этому ключу. Если же такого key нет, то является аналогом Add.

Concatenate	`DictionaryTree<Key, Value> *unitedWith(DictionaryTree<Key, Value> *tree1);`	Возвращает указатель на дерево, являющееся объединением двух других деревьев.
Thread	`Sequence<std::pair<Key, Value>> *thread(std::string order);`	Прошивка дерева. Принимает параметр обхода (напр. “NLR”) возвращает последовательность пар «ключ-значение».

3.2. Тип данных «Словарь», **Dictionary<Key, Value>**, представляет собой тип данных, инкапсулирующий в себе структуру DictionaryTree<Key, Value>. Является более высокоуровневой абстракцией, не зависящей от способа реализации DictionaryTree<Key, Value>.

Краткая спецификация класса Dictionary<Key, Value>:

Название	Сигнатура	Описание
Атрибуты		
Бинарное дерево поиска	`DictionaryTree<Key, Value> *dictionaryTree;`	Представляет собой инкапсулированный объект типа `DictionaryTree`.
Методы		
Конструктор создания пустого словаря	`Dictionary();`	
Конструктор копирования	`Dictionary(const Dictionary<Key, Value> &dictionary);`	
IsEmpty	`bool isEmpty();`	Проверка на пустой словарь.
Get	`Value get(Key key);`	Возвращает значение элемента по заданному ключу.
Add	`void add(Key key, Value val);`	Добавляет новую пару в словарь.
Contains	`bool contains(Key key);`	Проверка на наличие .
Remove	`void remove(Key key);`	Удаление элемента из словаря по заданному ключу.

Класс Dictionary должен инкапсулировать и все остальные методы класса DictionaryTree.

3.3. Тип данных «Алфавитный указатель», **AlphabetPointer**, должен представлять собой класс, в котором инкапсулирован процесс преобразования строкового типа данных (std::string) в словарь, в котором ключ представляет собой строку – слово, встречающееся в тексте, а значением является последовательность (Sequence) целых чисел, номеров страниц, на которых встречается слово.
Данный класс должен быть представлен единственным методом – конструктором, в котором реализуется соответсвующий алгоритм обработки.

3.4. Тип данных «Разреженный вектор», **SparseSeq<T>**, должен представлять собой класс, в котором в качестве его поля содержится словарь типа Dictionary<int, T>. Ключом является целое число, представляющее индекс ненулевого элемента в изначальной последовательности. Класс должен предоставлять 2 способа обработки исходной последовательности: стандартный способ с циклическим проходом коллекции элементов и реализацию в стиле map-reduce.
Краткая спецификация класса SparseSeq:

Название	Сигнатура	Описание
Атрибуты		
Словарь	`Dictionary<int, T> *storage;`	Словарь для хранения элементов разреженного вектора.
Length	`int length;`	Представляет собой размер изначальной последовательности.
Нулевое значение	`T null;`	Представляет собой нулевой элемент в данном векторе.
isNull	`bool (*isNull)(T);`	Указатель на функцию, возвращающую true в случае совпадения элемента с нулевым значением.
Методы		
Конструктор	`SparseSeq(Sequence<T> *seq, T null, bool (*isNull)(T), bool mapReduceOn = false);`	Основной конструктор, в котором происходит преобразование изначальной последовательности seq в разреженный вектор

		(mapRdeueceOn задает способ обработки).
Конструктор копирования	`SparseSeq(const SparseSeq<T> &seq);`	
Length	`int getLength();`	Возвращает размер исходной последовательности.
Get	`T get(int index);`	По заданному индексу возвращает значение элемента в векторе.
GetNull	`T getNull();`	Возвращает нулевое значение, установленное для этого вектора.

Основные отношения между классами должны соответствовать следующей диаграмме (рис.1).

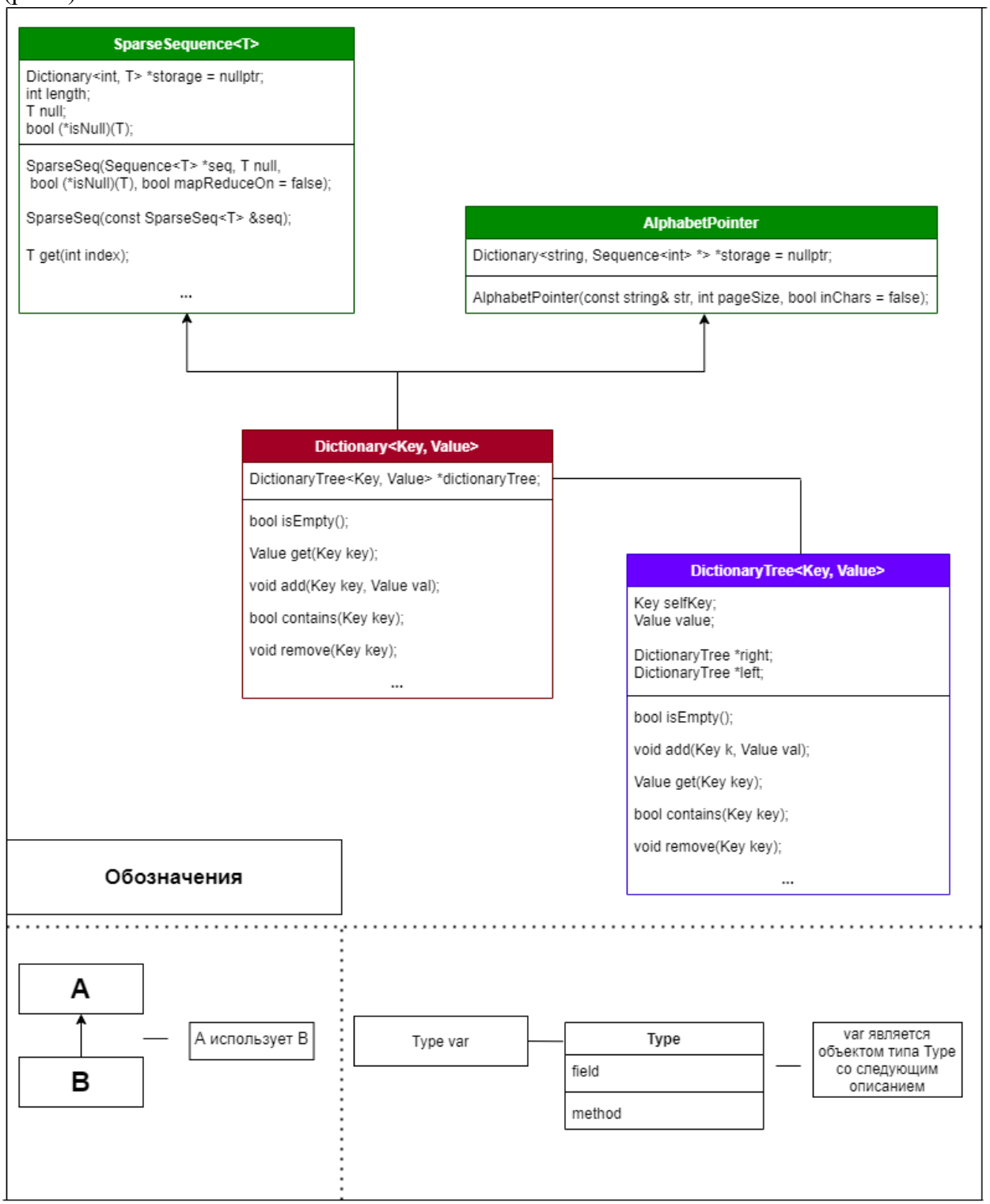

Рис. 1 (диаграмма классов)

4. Реализация

4.1. При создании класса DictionaryTree не использовался вспомогательный класс Node, представляющий собой узел дерева. Вместо под узлом дерева понималось само дерево. Вследствие такого подхода теряется необходимость создания различных дополнительных статических методов для реализации рекурсивных функций.

Метод Add был реализован рукрсивно по следующей схеме: если данный ключ меньше ключа узла, добавить в левое поддерево, больше – добавить в правое поддерево, равно – выбросить исключение (бинарное дерево не может содержать одинаковых элементов).

Метод Remove реализован по следующему принципу. Если удаляемый узел является листом (то есть не имеет никаких поддеревьев), то он прото удаляется, и указатель родителя узла на этот узел устанавливается в nullptr. Если же удалямый узел имеет только одно поддерево (второе – nullptr), то мы его удаляем, а указатель родителя устанавливаем на поддерево удаляемого элемента. Если же узел имеет 2 поддерева, то ключ и значение данного узла копируются у самого левого узла правого поддерева (так как там хранится минимальный элемент больший чем удаляемый). Далее этот самый левый узел правого поддерева должен быть удален (метод удаления узла с одним поддеревом уже был описан).

Метод Contains реализован стандартно для бинарного дерева. В данном случае реализован рекурсивный метод по следующей схеме: если ключ соответствует значению в текущем узле, то вернуть истину, если этого ключ меньше – вернуть this->Left->Contains(key), больше – вернуть this->Right->Contains(key).

Метод Get реализован также типичным способом для бинарного дерева. Происходит проход дерева с целью найти узел с указанным ключом. Если он был найдет, то возрващается значение узла, в противном случае выбрасывается исключение.

4.2. Класс Dictionary, как уже было сказано в предыдущих частях, в качестве поля класса имеет объект типа DictionaryTree<Key, Value>. Это значит, что весь класс Dictionary фактически является надстройкой над DictionaryTree. Соответственно, методы этого класса, проверяя данный словарь на пустоту, просто обращаются к методом DictionaryTree.

4.3. Одним из основных классов данного задания является AlphabetPointer, который по сути своей является абстракцией процесса построения алфавитного

указателя. Основным методом такого класса является конструктор, в котором непосредственно и происходят все вычисления.

На этапе split входная строка разбивается на список слов строки. В качестве разделителя выбирается пробел. Сигнатура метода split:

```
static Sequence<string> *split(const string &str);
```

Следующим важным этапом является расчет количества страниц. Для этих целей создана отдельная функция, принимающая список строк (разделенный на предыдущем этапе текст) и возвращающая количество страниц в соответствии с переданным размером страницы. Сигнатура метода pageCount:

```
static int pageCount(Sequence<string> *seq, int pageSize);
```

Следующий этап – DivideIntoPages. На данном этапе список из слов преобразуется в список, в котором элементы являются страницами. Тип возвращаемого значения Sequence<Sequence<string>*> (в качестве Sequence<string> здесь подразумевается тип «страница»). В данном методе как раз и требуется количество страниц, посчитанное не предыдущем шаге. Также на этом шаге учитывается, что количество слов на разных страницах может быть разное, а именно на первой половина размера страницы, а на каждой десятой – $\frac{3}{4}$ страницы. Сигнатура этого метода:

```
static Sequence<Sequence<string> *> *divideIntoPages(Sequence<string> *seq, int pageSize);
```

Заключающим этапом в построении указателя является преобразование Sequence<Sequence<string>*> в Dictionary<string, Sequecne<int>*>. В методе makeDictionary мы идем по списку страниц, а в каждой странице идем по каждому слову, добавляя его в наш словарь. Если же такое слово уже есть в словаре, то мы обновляем значение, отвечающее этому ключу. Таким образом, мы получаем словарь типа Dictionary<string, Sequence<int>*>, который является полем класса AlphabetPointer. Сигнатура метода:

```
static Dictionary<string, Sequence<int> *> *makeDictionary(Sequence<Sequence<string> *> *dividedSeq);
```

Тогда основной конструктор класса AlphabetPointer будет иметь следующий вид:

```
AlphabetPointer::AlphabetPointer(const string& str, int pageSize, bool inChars) {
    storage = makeDictionary(divideIntoPages(split(str), pageSize, inChars));
```

```
}
```

4.4. Реализация следующей задачи, абстрактного типа данных «разреженный вектор» (SparseSeq<T>), представляет собой класс, содержащий словарь типа Dictionary<int, T>, нулевой элемента типа T (элемент, который именно в данной задаче задан как нулевой), указатель на функцию isNull(T), определяющую, является ли элемент типа T нулевым, и длину изначального вектора.

В качестве методов в данном классе представлены основной конструктор, метод GetLength, метод Get, позволяющий получать доступ к элементам разреженной матрицы, и GetNull, возвращающий установленный нулевой элемент.

Основным в данном классе является непосредственно построение такого вектора, осуществляемое в конструкторе класса. В данном случае в конструктор добавлен параметр bool mapReduceOn. При значении false данного параметра сценарий обработки представляет обычное прохождение по массиву, проверку элемента на равенство нулю и добавление в словарь ненулевых элементов (их индексов на место ключа, самих же элементов в качестве значений). При значении true запускается процесс обработки в стиле map-reduce, на котором стоит остановиться чуть подробнее.

Стиль map-reduce является способом программирования, при котором основой обработки данных являются функции **map**, отображающая одну коллекцию элементов в другую, а также функция свертки списка **reduce**. Также подразумевается, что обработка данных будет протекать поочередно, то есть результат предыдущей операции является входным параметром для следующей. В данном случае схема обработки в таком стиле такова. На первом этапе, входной вектор (Sequence<T>) с помощью функции map должен преобразовываться в Sequence<Dictionary<int, T>*>, где каждый элемент списка является словарем, состоящим только из одного элемента (индекса базового вектора и значением, отвечающим этому вектору). Так как map должна быть чистой функцией, то в данном случае ее использовать некорректно, так как она может на одинаковых входных данных преобразовывать их в разные словари (если имеется 2 одинаковых элемента в базовом векторе). В этом случае лучше ввести некую новую функцию **indexing**, действующую так же как и map, но имеющую доступ к индексу элемента. На втором этапе происходит свертка полученного списка в один словарь, являющийся результатом. Функция свертки в данном случае представляет собой

конкатенацию двух словарей. При этом если у одного из словарей только один элемент и его значение равно нулю, то такой словарь отбрасывается. Таким образом, в конструкторе инициализация словаря будет иметь следующий вид:

```
storage = reduce<Dictionary<int, T>*>(indexing(seq), reducingFunction(isNull));
```

Реализация функций GetLength и GetNull составляет мало трудностей, так как эти функции просто возвращают значения соответствующих полей.

Функция Get пытается получить значение элемента из словаря и в случае исключения возвращает значение, являющееся нулевым элементом.