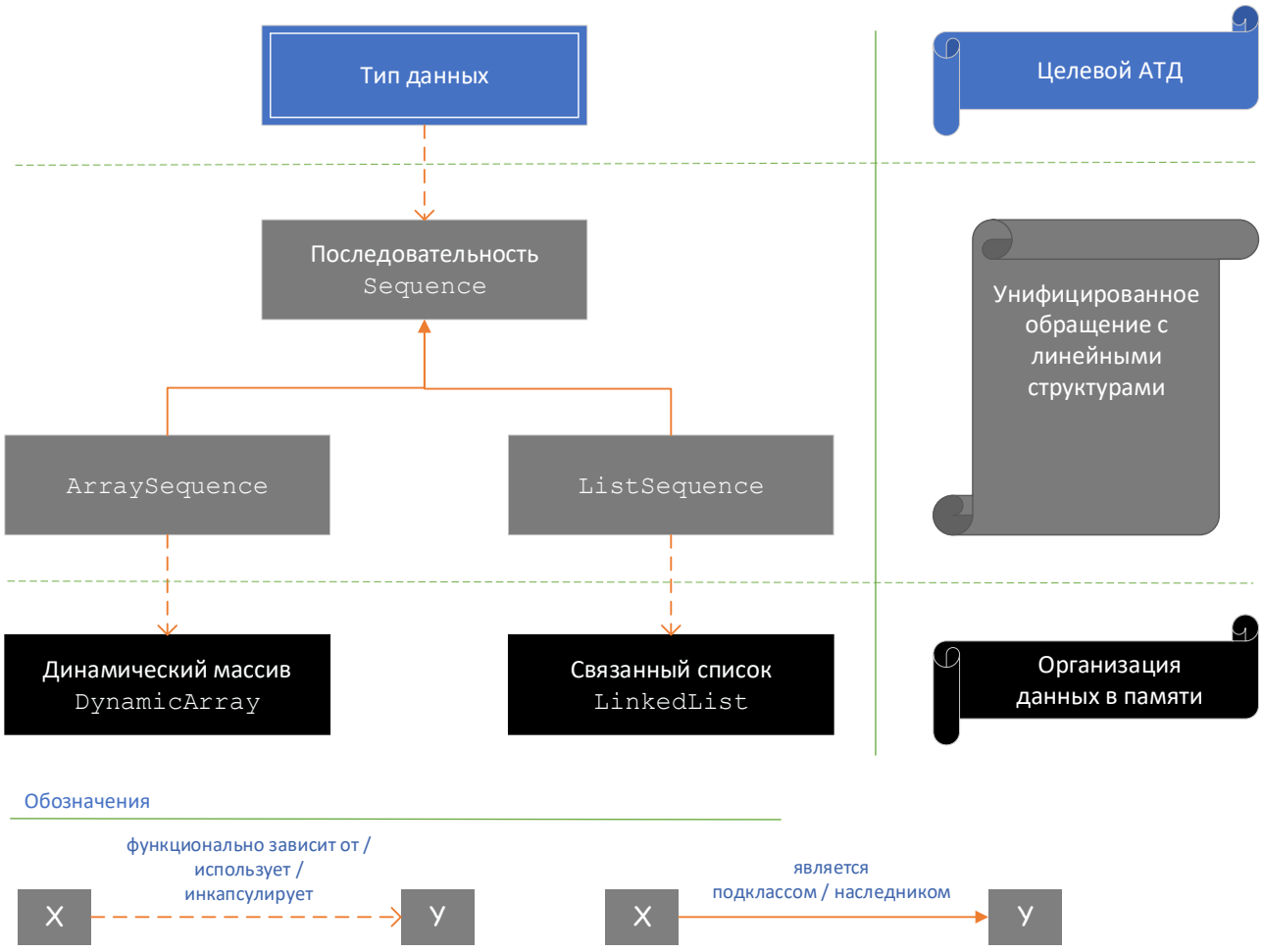
Лабораторная работа №2

по курсу информатики, 2 семестр

Варианты заданий

1. Постановка задачи

Написать на языке C++ систему для линейной организации с помощью нескольких полиморфных абстрактных типов данных – Динамический массив, Связанный список и Последовательность. При этом АТД Последовательность представляет собой абстракцию, предоставляющую унифицированный доступ к двум другим (см. рис. ниже). Реализуемая система должна предоставлять расширенный функционал для использования при реализации АТД и алгоритмов в последующих ЛР во 2-м и 3-м семестрах.



Следует предусмотреть средства для оценки производительности полученной реализации.

Минимальные требования к программе. В составе лабораторной работы должны быть реализованы:

- АТД динамический массив,
- АТД линейный связанный список,
- АТД последовательность,

- оболочка с UI для тестирования полученной системы объектов,
- АТД последовательность должен быть реализован в двух вариантах: как изменяемая (mutable) и как неизменяемая (immutable) структура.

Для реализации необходимо использовать возможности ООП и шаблонов C++ (templates) – классов и функций. Во всех реализованных функциях необходимо обрабатывать случаи некорректных значений входных параметров – как правило, в таких случаях следует выбрасывать исключения.

Все реализованные классы и основные алгоритмы необходимо покрыть (модульными) тестами. Реализацию следует оснастить пользовательским интерфейсом (консольным) для проверки корректности реализации.

Изменяемые (mutable) и неизменяемые (immutable) структуры данных. Отличительная особенность неизменяемых структур по сравнению с изменяемыми состоит в том, что, реализуя схожий, а зачастую – идентичный, интерфейс и набор операций, – неизменяемые структуры являются «чистыми» (pure), а точнее, чистыми, или функционально чистыми, являются их методы. Термин «функциональная чистота», или просто «чистота», применяется, чтобы обозначить что некоторый код ведет себя как «чисто математическая» сущность: результат работы кода полностью определяется исходными данными для него, которые задаются явно. Это означает, что:

- работа методов не зависит от внешней среды (по крайней мере, отсутствуют ***неявные*** зависимости[1]);
- в результате работы не меняется окружение (среда), и, обратно, изменение окружения в процессе выполнения методов не влияет на их работу[2];
- результат выполнения методов может зависеть, и как правило зависит, от внутреннего состояния объекта, но не меняет это состояние[3].

Обычно свойство неизменяемости достигается тем, что операции, которые в ином случае меняли бы состояние объекта, вместо этого создают копию исходного объекта, вносят изменения в эту копию и в качестве результат возвращают именно эту копию, оставляя исходный объект без изменений.

В данной работе необходимо выполнить и сравнить работу как изменяемой, так и неизменяемой последовательности. Это необходимо сделать хотя бы для одного вариантов последовательности – ArraySequence или/и ListSequence.

Дополнительные требования к программе (на повышение оценки).

- Перегрузка операторов (например, оператора [] для обращения/задания значения элемента по индексу).
- Поддержка комплекса операций map-reduce:
  - from
  - map,
  - flatmap
  - reduce/fold,
  - find/where,
  - zip и unzip.

[1] Т.е. работает принцип абстрагирования зависимостей [от внешней среды], т.е. все зависимости от чего-либо, лежащего за пределами данного объекта, объявляются явно.
[2] Принцип изоляции, или замкнутости.
[3] В контексте ООП состоянием объекта уместно называть совокупность значений его полей (в т.ч. и унаследованных). Однако в контексте неизменяемых структур сам термин «состояние» не особенно применим.

- Поддержка итераторов: поддержка интерфейса IEnumerable<>, реализация IEnumerator<>.
- Реализация типа $Option\langle T\rangle$ и try-семантики. Try-семантика состоит в том, что «поисковые» методы типа find, first и некоторые другие снабжаются вариантами, которые отличаются следующим:
  - имя начинается с префикса Try-
  - возвращаемый тип не T, а $Option\langle T\rangle$
  - если элемент не удается найти, исключение не выбрасывается, а возвращается значение $Option\langle T\rangle.None$.
- Реализовать функцию Split, которая данную последовательность элементов разобьет на отдельные фрагменты. Границы между фрагментами – это элементы, удовлетворяющие условию, которое передается как параметр.
- Реализовать функцию Slice, которая работает примерно так же, как в JavaScript: на вход подается индекс i, количество элементов N и, возможно, пустая, последовательность элементов s. Если индекс отрицательный, отсчет ведется с конца последовательности, а если он по модулю больше длины последовательности, следует выбросить исключение. Функция удаляет N элементов начиная с позиции i. При этом вместо удаленных элементов вставляются элементы из последовательности s (очевидно, если она не пустая).

```
Sequence<int> * seq = new ArraySequence(new [] {1,2,3,4,5});
Seq->Slice(1,2, new ArraySequence(new [] {9,10}); // == {1,9,10,4,5}
```

Перечисленные возможности могут быть реализованы как на уровне только Sequence, так «протянуты» через все уровни абстракции.

2. Требования к структурам данных

2.1. Класс DynamicArray

template <class T> class DynamicArray	
Создание объекта	
DynamicArray(T* items, int count);	Копировать элементы из переданного массива
DynamicArray(int size);	Создать массив заданной длины
DynamicArray(DynamicArray<T> & dynamicArray const);	Копирующий конструктор
Декомпозиция	
T Get(int index); Может выбрасывать исключения: – IndexOutOfRange (если индекс отрицательный, больше/равен числу элементов или указывает на не заданный элемент)	Получить элемент по индексу.
int GetSize();	Получить размер массива

Операции	
void Set(int index, T value);	Задать элемент по индексу Может выбросить IndexOutOfRange
void Resize(int newSize);	Изменить размер массива. Если размер увеличивается, все элементы копируются в начало новой памяти. Если уменьшается – элементы, которые не помещаются, отбрасываются.

2.2. Класс LinkedList

template <class T> class LinkedList	
Создание объекта	
LinkedList (T* items, int count);	Копировать элементы из переданного массива
LinkedList ();	Создать пустой список
LinkedList (LinkedList <T> & list const);	Копирующий конструктор
Декомпозиция	
T GetFirst(); Может выбрасывать исключения: – IndexOutOfRange (если список пуст)	Получить первый элемент в списке
T GetLast(); Может выбрасывать исключения: – IndexOutOfRange (если список пуст)	Получить последний элемент в списке
T Get(int index); Может выбрасывать исключения: – IndexOutOfRange (если индекс отрицательный или больше/равен числу элементов)	Получить элемент по индексу.
LinkedList<T>* GetSubList(int startIndex, int endIndex); Может выбрасывать исключения: – IndexOutOfRange (если хотя бы один из индексов отрицательный или больше/равен числу элементов)	Получить список из всех элементов, начиная с startIndex и заканчивая endIndex.

int GetLength();	Получить длину списка
Операции	
void Append(T item);	Добавляет элемент в конец списка
void Prepend(T item);	Добавляет элемент в начало списка
void InsertAt(T item, int index); Может выбрасывать исключения: – IndexOutOfRange (если индекс отрицательный или больше/равен числу элементов)	Вставляет элемент в заданную позицию
LinkedList<T>* Concat(LinkedList<T> *list);	Сцепляет два списка

2.3. Класс Sequence

template <class T> class Sequence template <class T> class ArraySequence : Sequence<T> template <class T> class ListSequence : Sequence<T>	
Создание объекта	
ArraySequence (T* items, int count); ListSequence (T* items, int count);	Копировать элементы из переданного массива
ArraySequence (); ListSequence ();	Создать пустой список
ArraySequence (LinkedList <T> & list const); ListSequence (LinkedList <T> & list const);	Копирующий конструктор
Декомпозиция	
T GetFirst(); Может выбрасывать исключения: – IndexOutOfRange (если список пуст)	Получить первый элемент в списке
T GetLast(); Может выбрасывать исключения: – IndexOutOfRange (если список пуст)	Получить последний элемент в списке

T Get(int index); Может выбрасывать исключения: – IndexOutOfRange (если индекс отрицательный или больше/равен числу элементов)	Получить элемент по индексу.
Sequence<T>* GetSubsequence(int startIndex, int endIndex); ArraySequence<T>* GetSubsequence(int startIndex, int endIndex); ListSequence<T>* GetSubsequence(int startIndex, int endIndex); Может выбрасывать исключения: – IndexOutOfRange (если хотя бы один из индексов отрицательный или больше/равен числу элементов)	Получить список из всех элементов, начиная с startIndex и заканчивая endIndex.
int GetLength();	Получить длину списка
Операции	
Sequence<T>* Append(T item); ArraySequence<T>*Append(T item); ListSequence<T>* Append(T item);	Добавляет элемент в конец списка
Sequence<T>*Prepend(T item); ArraySequence<T>*Prepend(T item); ListSequence<T>* Prepend(T item);	Добавляет элемент в начало списка
Sequence<T>*InsertAt(T item, int index); ArraySequence<T>*InsertAt(T item, int index); ListSequence<T>* InsertAt(T item, int index); Может выбрасывать исключения: – IndexOutOfRange (если индекс отрицательный или больше/равен числу элементов)	Вставляет элемент в заданную позицию
Sequence <T>* Concat(Sequence <T> *list);	Сцепляет два списка

Кроме того, здесь же оптимально реализовывать операции из серии map-reduce: map, reduce, zip, unzip, where (и др.).

Примерная схема реализации – на примере класса ArraySequence:

```
      // Упрощенная реализация, без «умного» управления буфером.
template <class T>
class ArraySequence : Sequence<T>
{
   private:
      DynamicArray<T>* items;
```

```
        //…
    public:
        //…
        int GetLength()
        {
              return this->items->GetSize();
        }
        //…
        void Append(T item)
        {
              this->items->Resize(this->items->GetSize()+1);
              this->items->Set(this->items->GetSize()-1, item);
        }
};
```

Т.е. для реализации используется инкапсуляция DynamicArray и делегирование ему большей части работы.

Реализация mutable- и immutable-вариантов. По сути, с точки зрения кода, различие межу mutable/immutable вариантами большинства структур состоит лишь в том, что в изменяемом варианте необходимые изменения вносятся в текущий объект, а в случае неизменяемой структуры – предварительно создается копия, с которой производятся те же действия, которые до этого производились над исходным объектом. Один из наиболее очевидных и прямолинейных вариантов реализации обеих вариаций состоит в вынесении основной части кода в базовый класс. Например:

```
Sequence<T>* ArraySequence::Append(T item)
{
      return Instance()->AppendInternal(item);
}
virtual Sequence<T>* ArraySequence::Instance() = 0;
virtual Sequence<T>* MutableArraySequence::Instance() override
{
      return this;
}
virtual Sequence<T>* ImmutableArraySequence::Instance() override
{
      return this->Clone();
}
```

Комплекс map-reduce. Если $l = [a_1,..,a_n]$ – некоторый список элементов типа T, а $f: T \to T$, то:

$$\text{map}(f, l) \mapsto [f(a_1),..,f(a_n)]$$

Если, при тех же соглашениях, $h: T \to \text{Bool}$ – некоторая функция, возвращающая булево значение, то результатом $\text{where}(h, l)$ будет новый список l', такой что: $a'_i \in l' \Leftrightarrow h(a'_i) = \text{true}$. Т.е. where фильтрует значения из списка l с помощью функции-фильтра h.

На практике при решении задач, использующих map, иногда требуется знать не только значение элемента, но и информация о списке в целом, например, положение данного элемента относительно других. Например, при подсчете количества инверсий в списке полезно знать индекс каждого элемента, чтобы не вычислять его отдельно. В этом случае функция, отвечающая за поэлементное преобразование, принимает дополнительный аргумент – индекс элемента в списке: $f: T_1 \times Int \to T_2$.

Функция reduce работает несколько иначе: «сворачивает» список в одно значение по заданному правилу $f: T \times T \to T$:

$$\text{reduce}(f, l, c) \mapsto f\left(a_n, \left(f\left(a_{n-1}, \left(\dots f\left(a_2, \left(f(a_1 c)\right)\right)\right)\right)\right)\right)$$

где c – константа, «стартовое» значение. Например, $l = [1,2,3]$, $f(x_1, x_2) = 2x_1 + 3x_2$, тогда:

$$\begin{aligned} &reduce(f, [1,2,3], 4) = f\left(3, f\big(2, f(1,4)\big)\right) = \\ &= 2 \cdot 3 + 3\big(2 \cdot 2 + 3(2 \cdot 1 + 3 \cdot 4)\big) = \\ &= 2 \cdot 3 + 3(2 \cdot 2 + 3 \cdot 14) = 2 \cdot 3 + 3 \cdot 42 = 132 \end{aligned}$$

Пара функций zip и unzip позволяет «сцеплять вдоль» несколько списков и разъединять их.

$$zip: List\langle T_1 \rangle \times List\langle T_2 \rangle \to List\big\langle Tuple\langle T_1, T_2 \rangle\big\rangle$$

$$zip3: List\langle T_1 \rangle \times List\langle T_2 \rangle \times List\langle T_3 \rangle \to List\big\langle Tuple\langle T_1, T_2, T_3 \rangle\big\rangle$$

$$zipN: List\big\langle List\langle T \rangle\big\rangle \to List\big\langle List\langle T \rangle\big\rangle$$

Примеры:

$$zip([1,2,3], ["a", "b", "c", "d",]) = [(1, "a"), (2, "b"), (3, "c")]$$

$$zipN\big(\big[[1,2,3], [1,2,3], [1,2,3]\big] = \big[[1,1], [2,2], [3,3]\big]\big)$$

Функция unzip является обратной к zip: «распаковывает» список кортежей в кортеж списков.

Указатели на функции. Ниже – минимальный пример, как создать «список функций»:

```
const int array_length = 3;
int(**f)(int) = malloc(array_length * sizeof(int(*)(int)));
f[0] = &inc1;
f[1] = &inc2;
f[2] = &inc3;
for (int index = 0; index < length; index++)
      printf("%i ", f[index](0));
// Вывод: 1 2 3
```

(Задание должно включать пункты общей «стоимостью» не менее 9 баллов.)

Таблица 2. Содержание вариантов заданий			
Код	**Задача**	**Пояснения**	**Баллы**
Б-1.	Связанный список		2
Б-2.	Динамический массив		2
Б-3.	Последовательность		2
Б-3.1.	ListSequence		
Б 3.1.1.	MutableListSequence		1
Б 3.1.2.	ImmutableListSequence		1
Б-3.2.	ArraySequence		
Б 3.2.1.	MutableArraySequence		1
Б 3.2.2.	ImmutableArraySequence		1
Б-3.3.	AdaptiveSequence		3
Б-3.4.	SegmentedList		5
М-1.	ICollection<T>	Основные методы – T Get(size_t) –size_t GetCount() –копирующий конструктор	2
М-2.	Map-Reduce	–	
М-2.1.	Базовые операции	From, Map, Reduce,	1
М-2.2.	Дополнительные	Zip/unzip, Split, Where, Concat, Subsequence	2
М-3.	Энумератор (IEnumerable + IEnumerator)	–	3
М-4.	Использовать IGroup/IRing	–	4
М-5.	Перегрузка операторов[8)]		2
М-6.		–	
М-7.		–	
М-8.		–	
П-1.			
П-2.			
П-3.			
П-4.			
П-5.			
П-6.			
П-7.			

3. Критерии оценки

1.	Качество программного кода	– стиль (в т.ч.: имена, отступы и проч.) – структурированность (напр. декомпозиция сложных функций на более простые) – качество основных и второстепенных алгоритмов	0-5 баллов

		(напр. обработка граничных случаев и некорректных исходных данных и т.п.)	
2.	Качество тестов	– степень покрытия – читаемость – качество проверки (граничные и некорректные значения, и др.)	0-5 баллов
		Итого	0-10 баллов
3.	Дополнительные возможности АТД	Учитывается, реализованы ли пункты из раздела М (количество и качество реализованных возможностей)	0-5 баллов
4.	Наработки по UI	Наличие UI и задел на дальнейшее развитие	0-5 баллов

Для получения зачета за выполнения лабораторной работы необходимо соблюдение всех перечисленных условий:

- оценка за п. 1 должна быть не менее 3 баллов
- оценка за п. 2 должна быть не менее 3 баллов