Разработка и реализация класса “Массив с динамически изменяемыми границами”

Содержание

1 Общие положения

В рамках данной работы массивы представляют собой одномерные изменяемые объекты, которые могут динамически сжиматься и расширяться. Каждый массив имеет нижнюю границу и последовательность элементов, пронумерованных, начиная с нижней границы. Все элементы массива относятся к одному и тому же типу [2].

Реализация массива с динамически изменяемыми границами будет базироваться на такой структуре данных, как дек или очередь с двумя концами (double-ended queue).

Дек — линейный список, в котором все включения и исключения (и обычно всякий доступ) делаются на обоих концах списка [1].

В соответствии с этим, выполнение работы состоит из нескольких этапов. Задание по каждому последующему этапу выдаётся после завершения выполнения текущего этапа.

Условием допуска к зачёту/экзамену является выполнение обязательных этапов с оценкой не ниже "**удовлетворительно**".

Разработка и реализация функций и методов для каждого этапа должна выполняться **строго** в соответствии с приведёнными в задании спецификациями этих функций и методов. Реализации этапов, выполненные с отступлениями от заданных спецификаций, рассматриваться не будут.

Перед передачей выполненной реализации на проверку данная реализация должна быть протестирована разработчиком.

Для успешной сдачи работы разработчик должен понимать и уметь объяснить все приведённые в работе объявления модулей, классов и шаблонов классов и их реализации.

2 Этап 1 (обязательный) "Дек целых значений (модуль, система программирования C)"

Разработка, реализация и тестирование модуля для работы с деком **целых** значений.

Инструмент: система программирования **C** [4].

2.1 Спецификация структуры данных модуля

```
/* Описание реализации элемента дека */
struct deque_element {
   int element;
   /* значение элемента */

   struct deque_element * next;
   /* указатель на следующий элемент */

   struct deque_element * prev;
   /* указатель на предыдущий элемент */
};

/* Описание реализации дека */
struct intdeque {
   struct deque_element * left;
   /* указатель на левый элемент дека */

   struct deque_element * right;
   /* указатель на правый элемент дека */

  int buffer;
   /* значение возвращаемого элемента для операций
      remove_left, remove_right */
};

typedef struct intdeque * intdeque;
```

2.2 Спецификация операций модуля

- Создание дека:

```
struct intdeque * create_deque();
```

Функция размещает в памяти новый дек. Функция возвращает указатель на созданный дек. Если дек не может быть создан, то функция возвращает значение NULL.

- Добавление элемента в дек с левого края:

```
int* add_left(struct intdeque * pdeque, int elem);
```

Функция добавляет новый элемент в дек, соответствующий структуре [pdeque], с левого края.
Значение нового элемента передаётся через параметр [elem].
В случае успешного выполнения функция помещает значение [elem] в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].
В противном случае функция возвращает значение NULL.

- Добавление элемента в дек с правого края:

```
int* add_right(struct intdeque * pdeque, int elem);
```

Функция добавляет новый элемент в дек, соответствующий структуре [pdeque], с правого края.
Значение нового элемента передаётся через параметр [elem].
В случае успешного выполнения функция помещает значение [elem] в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].
В противном случае функция возвращает значение NULL.

- Удаление элемента из дека с левого края:

```
int* remove_left(struct intdeque * pdeque);
```

Функция удаляет элемент из дека, соответствующего структуре [pdeque], с левого края.
В случае успешного выполнения функция помещает значение удаляемого элемента в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].
В противном случае функция возвращает значение NULL.

- Удаление элемента из дека с правого края:

```
int* remove_right(struct intdeque * pdeque);
```

Функция удаляет элемент из дека, соответствующего структуре [pdeque], с правого края.
В случае успешного выполнения функция помещает значение удаляемого элемента в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].
В противном случае функция возвращает значение NULL.

- Удаление дека:

```
struct intdeque * delete_deque(struct intdeque * pdeque);
```

Функция удаляет дек, соответствующий структуре `[pdeque]`.
В качестве результата функция возвращает значение NULL.

2.3 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.h` — файл с описанием структуры данных и функций (операций) модуля;
- `intdeque.c` — файл с исходным текстом реализации функций (операций) модуля;
- `test01.c` — файл, содержащий исходный текст на языке программирования `C` с реализацией функции `main()`, обеспечивающей тестирование разработанного модуля.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводится следующая информация:

- имя вызываемой функции;
- имя дека, для которого используется вызов функции;
- передаваемые целые значения (для функций `add_left(), add_right`);
- результат выполнения функции.

Формат вывода информации на экран (примеры для каждой из функций):

- функция `create_deque`:

  `pdeque01 = create_deque() == OK`
  — если функция возвращает значение, не равное NULL;

  `pdeque01 = create_deque() == NOT OK`
  — если функция возвращает значение, равное NULL;

- функции `add_left()` и `add_right()`:

  `add_left(pdeque01, 10) == 10`
  — если функция выполнена успешно, то необходимо после знака `==` вывести значение, хранящееся в области памяти, соответствующей полю `buffer` структуры, реализующей дек;

  `add_left(pdeque01, 10) == NULL`
  — если функция не может быть выполнена (например, нет памяти для добавления элемента в дек);

- функции `remove_left()` и `remove_right()`:

  `remove_left(pdeque01) == 10`
  — если функция выполнена успешно, то необходимо после знака `==` вывести значение, хранящееся в области памяти, соответствующей полю `buffer` структуры, реализующей дек;

  `remove_left(pdeque01) == NULL`
  — если функция не может быть выполнена (например, в деке нет элементов);

- функция `delete_deque()`:

  `delete_deque(pdeque01)`.

В тестовой программе общее количество вызовов функций `remove_left()` и `remove_right()` должно превышать общее количество вызовов функций `add_left()` и `add_right()` для каждого из деков.

После вызова функции `delete_deque()` необходимо вызвать функции `remove_left()` и `remove_right()` для каждого из деков, в результате чего должно быть обеспечено предсказуемое поведение программы.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

3 Этап 2 (обязательный) "Дек целых значений (класс, система программирования C++)"

Разработка, реализация и тестирование класса для работы с деком **целых** значений.
Инструмент: система программирования **C++** [5] [3].

3.1 Спецификация классов

Разработка класса "Дек целых значений" выполняется на основер результатов этапа 1.
В соответствии с этим предлагается разработать и реализовать следующие два класса:

- **IntDequeElement** — класс, содержащий структуру данных и методы для работы с элементом дека;
- **IntDeque** — класс, содержащий структуру данных и методы для работы с деком.

3.1.1 Спецификация класса IntDequeElement

```
class IntDequeElement{
  int element;
  /* значение элемента */

  IntDequeElement * next;
  /* указатель на следующий элемент */

  IntDequeElement * prev;
  /* указатель на предыдущий элемент */

public:
  IntDequeElement();
  /* Конструктор по умолчанию для создания элемента списка.
     Обеспечивает задание следующих начальных значений элементам класса:
     element <- 0
     next <- NULL
     prev <- NULL
  */

  IntDequeElement(int _element);
  /* Конструктор для создания элемента списка, обеспечивающий
     задание начального значения [_element] элементу списка:
     element <- _element
     next <- NULL
     prev <- NULL
  */

  IntDequeElement(int _element, IntDequeElement * _prev, IntDequeElement * _next);
  /* Конструктор для создания элемента списка, обеспечивающий
```

```
     задание начального значения [_element] элементу списка,
     задание начального значения [_prev] указателю на предыдущий элемент списка
     задание начального значения [_next] указателю на следующий элемент списка */

  void SetElement(int _element);
  /* Присваивание значения [_element] элементу списка */

  int GetElement();
  /* Получение значения элемента списка */

  void SetNext(IntDequeElement * _next);
  /* Присваивание значения [_next] полю next */

  IntDequeElement * GetNext();
  /* Получение значения поля next */

  void SetPrev(IntDequeElement * _prev);
  /* Присваивание значения [_prev] полю prev */

  IntDequeElement * GetPrev();
  /* Получение значения поля prev */
};
```

3.1.2 Спецификация класса IntDeque

```
class IntDeque{
  IntDequeElement * left;
  /* указатель на крайний левый элемент списка */

  IntDequeElement * right;
  /* указатель на крайний правый элемент списка */

  int buffer;
  /* Значение добавленного/удалённого элемента списка */

public:

  IntDeque();
  /* Конструктор по умолчанию для создания дека:
     left <- NULL
     right <- NULL
     buffer <- 0
  */

  int* AddLeft(int element);
  /* Добавление элемента со значением [element] слева.
     После добавления элемента поле buffer получает значение [element].
     В случае успешного выполнения, в качестве результата
     возвращает значение указателя на поле buffer.
     Иначе, возвращает значение NULL.
  */

  int* AddRight(int element);
  /* Добавление элемента со значением [element] справа.
     После добавления элемента поле buffer получает значение [element].
     В случае успешного выполнения, в качестве результата
```

```
     возвращает значение указателя на поле buffer.
     Иначе, возвращает значение NULL.
  */

  int* RemoveLeft();
  /* Удаление слева.
     Поле buffer получает значение удалённого элемента.
     В качестве результата возвращает значение указателя на поле buffer.
     Если в списке нет элементов, то, в качестве результата,
     возвращает значение NULL.
  */

  int* RemoveRight();
  /* Удаление справа.
     Поле buffer получает значение удалённого элемента.
     В качестве результата возвращает значение указателя на поле buffer.
     Если в списке нет элементов, то, в качестве результата,
     возвращает значение NULL.
  */

  int GetElement();
  /* Получение значения последнего добавленного или удалённого элемента
     (получение значения поля buffer).
  */

  ~IntDeque();
  /* Деструктор.
     Обеспечивает удаление всего дека.
  */

};
```

3.2 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов;
- `test02.cpp` — файл, содержащий исходный текст на языке программирования `C++` с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводится следующая информация:

- имя дека и вызываемый метод;
- передаваемые целые значения (для методов `AddLeft(), AddRight()`);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждого из методов):

- методы `AddLeft()`, `AddRight()`, `GetElement()`:

  `pdeque01.AddLeft(10); pdeque01.GetElement() == 10`
  — необходимо после знака == вывести результат выполнения метода `GetElement()`;

- методы `RemoveLeft()`, `RemoveRight()`, `GetElement()`:

  `pdeque01.RemoveLeft; pdeque01.GetElement() == 10`
  — если метод `RemoveLeft()` (`RemoveRight()`) выполнен успешно, то необходимо после знака == вывести результат выполнения метода `GetElement()`;

  `pdeque01.RemoveLeft() == NULL`
  — если метод не может быть выполнен (например, в деке нет элементов).

В тестовой программе общее количество вызовов методов `RemoveLeft()` и `RemoveRight()` должно превышать общее количество вызовов методо `AddLeft()` и `AddRight()` для каждого из деков.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования `C++` **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования `C++`.

4 Этап 3 (обязательный) “Дек целых значений (класс, обработка исключительных ситуаций)”

Разработка, реализация и тестирование класса для работы с деком **целых** значений с учётом возникновения исключительных ситуаций.

Инструмент: система программирования **C++** [5] [3].

Спецификация класса `IntDequeElement` не изменяется.

4.1 Спецификация класса IntDeque

```
class IntDeque{
  IntDequeElement * left;
  /* указатель на крайний левый элемент списка */

  IntDequeElement * right;
  /* указатель на крайний правый элемент списка */

public:
  class NoMemory{};
  /* Класс, соответствующий исключительной ситуации
     недостаточного количества памяти для очередного
     элемента дека.
  */

  class DequeIsEmpty{};
  /* Класс, соответствующий исключительной ситуации
     отсутствия элементов в деке (пустой дек).
  */

  IntDeque();
```

```
  /* Конструктор по умолчанию для создания дека:
     left <- NULL
     right <- NULL
  */

  void AddLeft(int element) throw(NoMemory);
  /* Добавление элемента со значением [element] слева.
     Если нет памяти для добавления элемента,
     то порождается исключительная ситуация NoMemory.
  */

  void AddRight(int element) throw(NoMemory);
  /* Добавление элемента со значением [element] справа.
     Если нет памяти для добавления элемента,
     то порождается исключительная ситуация NoMemory.
  */

  int RemoveLeft() throw(DequeIsEmpty);
  /* Удаление слева.
     В качестве результата возвращает значение удалённого элемента.
     Если в очереди нет элементов,
     то порождается исключительная ситуация DequeIsEmpty.
  */

  int RemoveRight() throw(DequeIsEmpty);
  /* Удаление справа.
     В качестве результата возвращает значение удалённого элемента.
     Если в очереди нет элементов,
     то порождается исключительная ситуация DequeIsEmpty.
  */

  ~IntDeque();
  /* Деструктор.
     Обеспечивает удаление всего дека.
  */

};
```

4.2 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов;
- `test03.cpp` — файл, содержащий исходный текст на языке программирования `C++` с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводится следующая информация:

- имя дека и вызываемый метод;

- передаваемые целые значения (для методов `AddLeft(), AddRight()`);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждого из методов):

- методы `AddLeft()`, `AddRight()`:

  `pdeque01.AddLeft(10) == OK`
  — в случае успешного выполнения метода;

  `pdeque01.AddLeft(10) == NoMemory`
  — в случае возникновения исключительной ситуации `NoMemory`.

- методы `RemoveLeft()`, `RemoveRight()`:

  `pdeque01.RemoveLeft() == 10`
  — в случае успешного выполнения метода после знака `==` выводится значение удалённого из дека элемента;

  `pdeque01.RemoveLeft() == DequeIsEmpty`
  — в случае возникновения исключительной ситуации `DequeIsEmpty`.

В тестовой программе общее количество вызовов методов `RemoveLeft()` и `RemoveRight()` должно превышать общее количество вызовов методо `AddLeft()` и `AddRight()` для каждого из деков.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования `C++` **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования `C++`.

5 Этап 4 (обязательный) “Дек целых значений (класс, переопределение операций)”

Дополнить класс дека целых чисел, разработанный на этапе 3, следующими операциями:

- $int + Deque$ — добавить в дек $Deque$ целочисленный элемент int слева; если нет памяти под добавляемый элемент, то данная операция порождает исключительную ситуацию `NoMemory`;
- $Deque + int$ — добавить в дек $Deque$ целочисленный элемент int справа; если нет памяти под добавляемый элемент, то данная операция порождает исключительную ситуацию `NoMemory`;
- $--Deque$ — удалить из дека $Deque$ элемент слева; в качестве результата операция возвращает значение удалённого элемента; если в деке нет элементов, то данная операция порождает исключительную ситуацию `DequeIsEmpty`;
- $Deque--$ — удалить из дека $Deque$ элемент справа, в качестве результата операция возвращает значение удалённого элемента; если в деке нет элементов, то данная операция порождает исключительную ситуацию `DequeIsEmpty`.

Инструмент: система программирования **C++** [5] [3].

5.1 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов;
- `test04.cpp` — файл, содержащий исходный текст на языке программирования `C++` с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводится следующая информация:

- имя дека и вызываемый метод;
- передаваемые целые значения (для операции +);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждого из операций):

- операция +:

  `10 + pdeque01 == OK`
  — в случае успешного выполнения операции;

  `10 + pdeque01 == NoMemory`
  — в случае возникновения исключительной ситуации `NoMemory`.

  `pdeque01 + 20 == OK`
  — в случае успешного выполнения операции;

  `pdeque01 + 20 == NoMemory`
  — в случае возникновения исключительной ситуации `NoMemory`.

- операция −−:

  `--pdeque01 == 10`
  — в случае успешного выполнения операции;

  `--pdeque01 == DequeIsEmpty`
  — в случае возникновения исключительной ситуации `DequeIsEmpty`.

  `pdeque01-- == 20`
  — в случае успешного выполнения операции;

  `pdeque01-- == DequeIsEmpty`
  — в случае возникновения исключительной ситуации `DequeIsEmpty`.

В тестовой программе общее количество операций −− должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования `C++` **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования `C++`.

6 Этап 5 (обязательный) “Список целых значений (класс, наследование)”

Разработать и реализовать класс `IntList`, обеспечивающий работу с двунаправленным списком целых значений. Данный класс должен быть классом-наследником класса `IntDeque`, который был разработан и реализован в рамках предыдущего этапа.

Поведение объектов класса `IntList` определяется набором следующих методов и операций:

`void GoToLeft()` — переход к крайнему левому элементу списка;

`void GoToRight()` — переход к крайнему правому элементу списка;

`void GoToNext()` — переход к следующему элементу списка; если текущий элемент списка неопределён (указатель на текущий элемент имеет значение `NULL`), то данный метод должен порождать исключительную ситуацию `NoCurrentElement`; если текущий элемент является крайним правым элементом (нет следующего элемента), то данный метод должен порождать исключительную ситуацию `NoNextElement` и, при этом, текущий элемент не изменяется;

`void GoToPrev()` — переход к предыдущему элементу списка; если текущий элемент списка неопределён (указатель на текущий элемент имеет значение `NULL`), то данный метод должен порождать исключительную ситуацию `NoCurrentElement`; если текущий элемент является крайним левым элементом (нет предыдущего элемента), то данный метод должен порождать исключительную ситуацию `NoPrevElement` и, при этом, текущий элемент не изменяется;

`int Fetch()` — получение значения текущего элемента списка; в качестве результата метод возвращает значение текущего элемента списка; если текущий элемент списка неопределён (указатель на текущий элемент имеет значение `NULL`), то данный метод должен порождать исключительную ситуацию `NoCurrentElement`;

`void Store(int elem)` — изменение значения текущего элемента списка на значение `elem`; если текущий элемент списка неопределён (указатель на текущий элемент имеет значение `NULL`, то данный метод должен порождать исключительную ситуацию `NoCurrentElement`;

`int + List` — добавление в список `List` целочисленного элемента `int` слева; при успешном добавлении элемента, добавленный элемент становится текущим элементом списка, иначе текущий элемент не изменяется; если нет памяти под добавляемый элемент, то данная операция порождает исключительную ситуацию `NoMemory`;

`List + int` — добавление в список `List` целочисленного элемента `int` справа; при успешном добавлении элемента, добавленный элемент становится текущим элементом списка, иначе текущий элемент не изменяется; если нет памяти под добавляемый элемент, то данная операция порождает исключительную ситуацию `NoMemory`;

`--List` — удаление крайнего левого элемента списка; при успешном удалении элемента, элемент, следующий за удалённым, становится текущим, иначе, указатель на текущий элемент принимает значение `NULL`; в качестве результата операция возвращает значение удаляемого элемента; если в списке нет элементов, то данный метод должен порождать исключительную ситуацию `ListIsEmpty`;

`List--` — удаление крайнего правого элемента списка; при успешном удалении элемента, элемент, предыдущий по отношению к удалённому, становится текущим, иначе, указатель на текущий элемент принимает значение `NULL`; в качестве результата операция возвращает значение удаляемого элемента; если в списке нет элементов, то данный метод должен порождать исключительную ситуацию `ListIsEmpty`.

В процессе выполнения данного этапа допускается модификация классов `IntDeque` и `IntDequeElement`, но при условии, что внесённые в указанные классы изменения не повлекут изменений тестовой программы, реализованной на предыдущем этапе.

Рассмотреть возможность часть методов модифицированных классов `IntDeque` и `IntDequeElement` реализовать как защищённые методы (вид доступа `protected`).

Инструмент: система программирования **C++** [5] [3].

6.1 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов `IntDequeElement` и `IntDeque`;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов `IntDequeElement` и `IntDeque`;
- `intlist.hpp` — файл с описанием класса `IntList`;
- `intlist.cpp` — файл с исходным текстом реализации методов класса `IntList`;
- `test05.cpp` — файл, содержащий исходный текст на языке программирования `C++` с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу с двумя списками с именами `plist01` и `plist02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны списка и не менее трёх элементов с правой стороны списка для каждого из списков, а также, получение и изменение значений некоторых из элементов, не являющихся крайними элементами списка. При каждом действии на экран должна выводится следующая информация:

- имя списка и вызываемый метод;
- передаваемые целые значения (для операции + и метода `Store`);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждого из методов или операций):

- операция +:

  `10 + plist01 == OK`
  — в случае успешного выполнения операции;

  `10 + plist01 == NoMemory`
  — в случае возникновения исключительной ситуации `NoMemory`.

  `plist01 + 20 == OK`
  — в случае успешного выполнения операции;

  `plist01 + 20 == NoMemory`
  — в случае возникновения исключительной ситуации `NoMemory`.

- операция −−:

  `--plist01 == 10`
  — в случае успешного выполнения операции;

  `--plist01 == ListIsEmpty`
  — в случае возникновения исключительной ситуации `ListIsEmpty`.

  `plist01-- == 20`
  — в случае успешного выполнения операции;

`plist01-- == ListIsEmpty`
— в случае возникновения исключительной ситуации `ListIsEmpty`.

- методы `GoToLeft()` и `GoToRight()`:

  `plist01.GoToLeft() == OK`

- `GoToNext()` и `GoToPrev()`:

  `plist01.GoToNext() == OK`
  — в случае успешного выполнения метода;

  `plist01.GoToNext() == NoCurrentElement`
  — в случае возникновения исключительной ситуации `NoCurrentElement`.

  `plist01.GoToNext() == NoNextElement`
  – в случае возникновения исключительной ситуации `NoNextElement`;

  `plist01.GoToPrev() == NoPrevElement`
  – в случае возникновения исключительной ситуации `NoPrevElement`;

- `Fetch()`:

  `plist01.Fetch() == 10`
  — в случае успешного выполнения метода;

  `plist01.Fetch() == NoCurrentElement` — в случае возникновения исключительной ситуации `NoCurrentElement`.

- `Store(elem)`:

  `plist01.Store(20) == OK`
  — в случае успешного выполнения метода;

  `plist01.Store(20) == NoCurrentElement`
  — в случае возникновения исключительной ситуации `NoCurrentElement`.

В тестовой программе общее количество операций −− должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования `C++` **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования `C++`.

7 Этап 6 (обязательный) “Массив целых чисел с динамически изменяемыми границами (класс, наследование)”

Разработать и реализовать класс `IntArray`, обеспечивающий работу с массивом целых чисел с изменяемыми границами [2]. При выполнении работы использовать реализованные на предыдущих этапах классы `IntDequeElement`, `IntDeque`, `IntList`.

В рамках данной работы массивы представляют собой одномерные изменяемые объекты, которые могут динамически сжиматься и расширяться. Каждый массив имеет нижнюю границу и последовательность элементов, пронумерованных, начиная с нижней границы. Все элементы массива относятся к одному и тому же типу.

Для массива должны быть определены следующие операции:

- Конструкторы, обеспечивающие создание объектов класса `IntArray`. Конструкторы должны обеспечить следующие возможности создания массивов:

  – `IntArray a` — создание пустого массива с именем `a`; при добавлении первого элемента массива, индекс этого элемента должен принять значение `1`;

- `IntArray a(lb)` — создание пустого массива с именем `a`; при добавлении первого элемента массива, индекс этого элемента должен принять значение, заданное целочисленным параметром `lb`, в качестве которого также допускается использование целочисленной константы;
- `IntArray a(b)` — конструктор копирования, обеспечивающий создание массива с именем `a`, являющегося копией уже существующего массива `b` (массив `b` также является объектом класса `IntArray`), то есть, массив `a` после создания должен иметь те же самые индексы первого и последнего элементов, тот же набор, порядок и значения элементов, что и массив `b`; если нет памяти для создания массива, то порождается исключительная ситуация `NoMemory`;
- `IntArray a(lb, cnt, val)` — создание массива `a`, состоящего из элементов, количество которых соответствует значению целочисленного параметра `cnt`, причём каждый элемент имеет значение заданное целочисленным параметром `val`, а индекс первого элемента массива задаётся целочисленным параметром `lb`; любой из перечисленных параметров может быть задан целочисленной константой; если нет памяти для создания массива, то порождается исключительная ситуация `NoMemory`;

- Операции, обеспечивающие работу с массивом:
  - `a.Low()` — получение значения индекса первого по счёту элемента массива `a` (получение значения нижней границы массива); если в массиве нет элементов, то порождается исключительная ситуация ArrayIsEmpty;
  - `a.High()` — получение значения индекса последнего по счёту элемента массива `a` (получение значения верхней границы массива); если в массиве нет элементов, то порождается исключительная ситуация ArrayIsEmpty;
  - `a.Size()` — получение текущего количества элементов массива `a`;
  - `a[i]` — получение значения элемента массива `a` с индексом `i`; если в массиве нет элемента с заданным индексом, то порождается исключительная ситуация `WrongIndex`; если в массиве нет элементов, то порождается исключительная ситуация `ArrayIsEmpty`;
  - `a[i] = intval` — присваивание элементу массива `a` с индексом `i` целочисленного значения, заданного выражением `intval`; целочисленная переменная и целочисленная константа являются частными случаями целочисленного выражения; если в массиве нет элемента с заданным индексом, то порождается исключительная ситуация `WrongIndex`; если в массиве нет элементов, то порождается исключительная ситуация `ArrayIsEmpty`;
  - `a + intval` — добавление к массиву `a` элемента с целочисленным значением `intval` со стороны верхней границы; в результате выполнения этой операции количество элементов и значение верхней границы увеличиваются на единицу и, соответственно, добавленный элемент становится последним по счёту элементом массива; если нет памяти для добавления элемента, то порождается исключительная ситуация `NoMemory`;
  - `intval + a` — добавление к массиву `a` элемента с целочисленным значением `intval` со стороны нижней границы; в результате выполнения этой операции количество элементов массива увеличивается на единицу, а значение нижней границы уменьшается на единицу и, соответственно, добавленный элемент становится первым по счёту элементом массива; если нет памяти для добавления элемента, то порождается исключительная ситуация `NoMemory`;
  - `a--` — удаление элемента массива `a` со стороны верхней границы (удаление последнего по счёту элемента массива); в качестве результата данная операция возвращает значение удаляемого элемента массива; при удалении количество элементов в массиве и значение верхней границы уменьшаются на единицу; если в массиве нет элементов, то порождается исключительная ситуация `ArrayIsEmpty`;
  - `--a` — удаление элемента массива `a` со стороны нижней границы (удаление первого по счёту элемента); в качестве результата данная операция возвращает значение

удаляемого элемента массива; при удалении количество элементов в массиве уменьшается на единицу, а значение нижней границы увеличивается на единицу; если в массиве нет элементов, то порождается исключительная ситуация `ArrayIsEmpty`;

- `a = b` — присваивание массиву `a` значения массива `b`; в результате выполнения данной операции массив `a` становится копией массива `b`; если перед выполнением операции присваивания в массиве `a` есть элементы, то они удаляются.

- Деструктор, обеспечивающий удаление элементов массива.

В процессе выполнения данного этапа допускается модификация классов `IntDeque`, `IntDequeElement` и `IntList`, но при условии, что внесённые в указанные классы изменения не повлекут изменений тестовой программы, реализованной на предыдущем этапе.

Рассмотреть возможность часть методов модифицированных классов `IntDeque`, `IntDequeElement` и `IntList` реализовать как защищённые методы (вид доступа `protected`).

Инструмент: система программирования **C++** [5] [3].

7.1 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов `IntDequeElement` и `IntDeque`;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов `IntDequeElement` и `IntDeque`;
- `intlist.hpp` — файл с описанием класса `IntList`;
- `intlist.cpp` — файл с исходным текстом реализации методов класса `IntList`;
- `intarray.hpp` — файл с описанием класса `IntArray`;
- `intarray.cpp` — файл с исходным текстом реализации методов класса `IntArray`;
- `test06.cpp` — файл, содержащий исходный текст на языке программирования `C++` с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу с двумя массивами с именами `parray01` и `parray02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов со стороны нижней границы массива и не менее трёх элементов со стороны верхней границы массива для каждого из массивов, а также, получение и изменение значений некоторых из элементов, не являющихся крайними элементами массива. При каждом действии на экран должна выводится следующая информация:

- выполняемое действие;
- результат выполнения действия;
- состояние массива после выполнения действия.

7.1.1 Пример вывода информации на экран

```
IntArray a
OK
++
!!
++
-

a.Low() == ArrayIsEmpty
```

```
ArrayIsEmpty
++
!!
++
-

a + 10
OK
+------+
!    10!
+------+
      1

20 + a
OK
+------+------+
!    20!    10!
+------+------+
     -1      1

a[1] == 10
OK
+------+------+
!    20!    10!
+------+------+
     -1      1

a[2] == WrongIndex
WrongIndex
+------+------+
!    20!    10!
+------+------+
     -1      1
```

В тестовой программе общее количество операций −− должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

8 Требования к оформлению писем электронной почты

Подготовленные задания отправляются на адрес электронной почты:

```
andrei.stankevich@gmail.com
```

Тема письма должна быть задана в соответствии со следующим форматом:

```
<Фамилия И.О.> <Группа> C++ ЭТАП <Номер этапа>
```

Пример:

```
Иванов И.И. 06-221 C++ ЭТАП 2
```

Любое из писем должно относиться ровно к одному этапу работы. То есть, в одном письме не должно содержаться результатов по нескольким этапам работы.

Первое письмо, относящееся к каждому последующему этапу, НЕ должно отправляться, как ответ (replay) на письмо преподавателя с ответом по предыдущему этапу, а должно создаваться, как новое письмо.

Все последующие письма по определённому этапу должны отправляться, как ответы на соответствующие письма с сохранением истории переписки. Во вложении, в этом случае, должны быть только обновлённые файлы. Это значит, что предыдущие версии файлов в очередной ответ не вкладываются.

Внимание! Письма, оформленные с отступлением от приведённых правил, рассматриваться не будут.

Список литературы

[1] Дональд Кнут, Искусство программирования для ЭВМ, том 1 “Основные алгоритмы”, М., “Мир”, 1976.

[2] Барбара Лисков, Джон Гатэг, Использование абстракций и спецификаций при разработке программ, М., “Мир”, 1989.

[3] Т.А. Павловская, C/C++. Программирование на языке высокого уровня, СПб., “Питер”, 2013.

[4] Брайан Керниган, Денис Ритчи, Язык программирования C.

[5] Бьерн Страуструп, Язык программирования C++.

[6] Никлаус Вирт, Алгоритмы и структуры данных.