Содержание

Лабораторная работа 1 – Динамический массив

Для выполнения всех последующих работ нужно изучить понятие анализа алгоритма (сложность, затраченные время и память), нотация О (О-большое), а также структуру папок при выполнении лабораторной работы.

Анализ алгоритмов

Эффективность алгоритма определяется с помощью анализа:

- времени работы (обозначиться **T(n)**, где **n** – входные параметры). Временная сложность алгоритма показывает, как изменяется время выполнения алгоритма в зависимости от размера входных данных. Она обычно выражается с использованием нотации “О” (О-большое);
- объема дополнительной используемой памяти (обозначиться **M(n)**, где **n** – входные параметры). Пространственная сложность алгоритма показывает, сколько памяти требуется алгоритму в зависимости от размера входных данных. Она также выражается с использованием нотации "О";
- количество обращению к ПЗУ;
- др.

Нотация “О” (О-большое) описывает асимптотическое поведение алгоритма, то есть его поведение при стремлении размера входных данных к бесконечности. На рисунке 1 представлены сложности алгоритмов для нотации “О”.

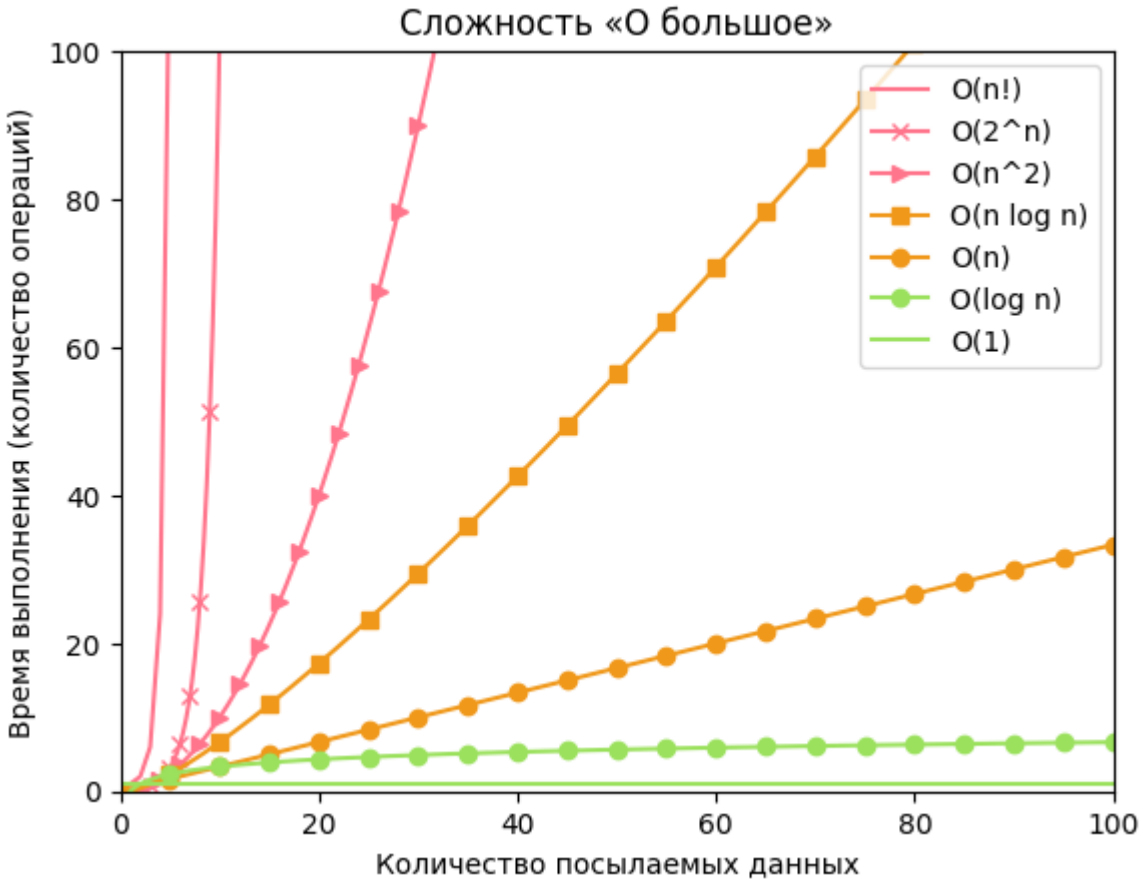

Рисунок 1 – Сложности алгоритмов для нотации “О”

Ниже перечислены основные нотации, расположенные от наилучшей к наихудшей:

- **O(1) – константная сложность**. Выполнение алгоритма не зависит от размера входных данных;
- **O(log n) – логарифмическая сложность**. Выполнение алгоритма растет логарифмически с увеличением размера входных данных;
- **O(n) – линейная сложность**. Выполнение алгоритма растет линейно с увеличением размера входных данных;
- **O(n log n) – линейно-логарифмическая сложность**. Выполнение алгоритма растет пропорционально произведению размера входных данных и логарифма от этого размера;
- **O(n^2) – квадратичная сложность**. Выполнение алгоритма растет пропорционально квадрату размера входных данных;
- **O(n^n) – экспоненциальная сложность**. Выполнение алгоритма растет экспоненциально с увеличением размера входных данных;
- **O(n!) – факториальная сложность**. Выполнение алгоритма растет факториально с увеличением размера входных данных.

Кроме “O” существуют другие нотации для оценки сложности алгоритмов:

- **Ω (омега – большое)** – нижняя граница. Показывает минимальное время выполнения алгоритма в худшем случае;
- **ω (омега – малое)** – нижняя граница, но не строгая. Показывает, что время выполнения алгоритма растет быстрее, чем указанная функция;
- **Θ (тета)** – точная граница. Показывает как верхнюю, так и нижнюю границу времени выполнения алгоритма;
- **o (малое o)** – верхняя граница, но не строгая. Показывает, что время выполнения алгоритма растет медленнее, чем указанная функция.

Эти нотации помогают более точно описать поведение алгоритмов и сравнить их эффективность в различных условиях.

Рассмотрим пример оценки расчета количества инструкций, из которой можно будет определить время выполнения и затраченной дополнительной памяти в нотации “O”. Ниже приведен листинг примера:

```
1. int getMax(int* array, int n)
2. {
3.   // проверка размерности
4.   …
5.   int maxValue = array[0];      // 3
6.   for(int i = 0; i < n; i++)    // 2+2*n
7.   {
8.     if (array[i] > maxValue)   // 2*n
9.     {
10.      maxValue = array[i];     // 2*n
11.    }
12.  }
13.  return maxValue;
14. }
```

В первой строке происходит три операции: создание переменной `maxValue`, получение по индексу значения из массива `array` и установка этого значения в переменную `maxValue`. Далее в шапке цикла `for` происходит создание переменной `i` и установка значения `0` данной переменной. После происходит проверка, что `i` меньше значения переменной `n`, а

также происходит инкремент в переменной `i`. Последние две операции происходит n раз. Поэтому в результате получается количество инструкций равное `2 + 2*n`.
После идет сравнение значения из массива с максимальным значением. Сначала идет получение элемента из массива `array`, а затем идет сравнения этого значения со значением переменной `maxValue`. Так, как это проходит в каждой итерации цикла, то количество выполненных инструкций будет равно `2*n`.
Если значение из массива `array` больше значения переменной `maxValue`, то идет присвоение этого значения в переменную `maxValue`. Тут выполняются две инструкции: получение элемента по индексу из массива `array` и присвоение значения в переменную `maxValue`.
В результате в наихудшем случае количество инструкций будет равно 4+6n, когда все значения будут по возрастанию в массиве, а в наилучшем 4+4n, когда все значения будут по убыванию в массиве. По затраченной дополнительной памяти можно обозначить создание двух переменных `maxValue` и `i`.
Для нотации “O” берется наихудший вариант и убираются константные значения. Например, затраченное время T(n) = O(4+6n) после того, как уберутся константные значения T(n) = O(n). По памяти M(n) = O(2), уберем константы, тогда получится M(n) = O(1).

Выбор варианта

Для выбора нужного варианта выполнения нужно знать свой номер в списке группы и использовать его как свой вариант. Например, всего вариантов 10, студентов в группе 21. Вариант 1 получать студенты под номером списка 1, 11 и 21 и так с каждым вариантом.

Задание 1 – Объяснение затрат по времени и памяти

Ниже приведен листинг кода и затраченное количества времени и памяти. Нужно объяснить, почему именно такое значение в нотации “O”.

Вариант 1:

```
1. int Fib1(int n)
2. {
3.   if (n < 2)
4.   {
5.     return n;
6.   }
7.   return Fib1(n-1) + Fib1(n-2);
8. }
```

T(n) = O(2^n)
M(n) = O(2^n)

Вариант 2:

```
1. int Fib2(int n)
2. {
3.   if (n == 0) return 1;
4.   int prev = 1, current = 1;
5.   for (int i = 0; i <= n; i++)
6.   {
7.     int t = current;
8.     current += prev;
9.     prev = temp;
```

```
10.   }
11.   return current;
12. }
```

T(n) = O(n)
M(n) = O(1)

Вариант 3:

```
 1. void BubbleSort(int arr[], int n)
 2. {
 3.     for (int i = 0; i < n-1; i++)
 4.     {
 5.         for (int j = 0; j < n-i-1; j++)
 6.         {
 7.             if (arr[j] > arr[j+1])
 8.             {
 9.                 swap(arr[j], arr[j+1]);
10.             }
11.         }
12.     }
13. }
```

T(n) = O(n^2)
M(n) = O(1)

Вариант 4:

```
 1. int BinarySearch(int arr[], int l, int r, int x)
 2. {
 3.     while (l <= r)
 4.     {
 5.         int mid = l + (r - l) / 2;
 6.
 7.         if (arr[mid] == x)
 8.             return mid;
 9.
10.         if (arr[mid] < x)
11.             l = mid + 1;
12.         else
13.             r = mid - 1;
14.     }
15.     return -1;
16. }
```

T(n) = O(log n)
M(n) = O(1)

Вариант 5:

```
 1. int LinearSearch(int arr[], int n, int x)
 2. {
 3.     for (int i = 0; i < n; i++)
 4.     {
 5.         if (arr[i] == x)
 6.             return i;
 7.     }
 8.
 9.     return -1;
10. }
```

T(n) = O(n)

M(n) = O(1)

Вариант 6:

```
 1. int Factorial(int n)
 2. {
 3.     if (n == 0)
 4.     {
 5.         return 1;
 6.     }
 7.     else
 8.     {
 9.         return n * Factorial(n-1);
10.     }
11. }
```

T(n) = O(n)
M(n) = O(n)

Вариант 7:

```
1. int Gcd(int a, int b)
2. {
3.     if (b == 0)
4.     {
5.   return a;
6.     }
7.     return Gcd(b, a % b);
8. }
```

T(n) = O(log(min(a, b)))
M(n) = O(log(min(a, b)))

Вариант 8:

```
1. int SumArrayRecursive(int arr[], int n)
2. {
3.     if (n == 0)
4.     {
5.         return 0;
6.     }
7.     return arr[n-1] + SumArrayRecursive(arr, n-1);
8. }
```

T(n) = O(n)
M(n) = O(n)

Вариант 9:

```
 1. double AverageArray(int arr[], int n)
 2. {
 3.     int sum = 0;
 4.     for (int i = 0; i < n; i++)
 5.     {
 6.         sum += arr[i];
 7.     }
 8.     return static_cast<double>(sum) / n;
 9. }
```

T(n) = O(n)
M(n) = O(1)

Вариант 10:

```
1. bool isPrime(int n) {
2.     if (n <= 1)
3.         return false;
4.     for (int i = 2; i * i <= n; i++) {
5.         if (n % i == 0)
6.             return false;
7.     }
8.     return true;
9. }
```

T(n) = O(sqrt(n))
M(n) = O(1)

Структура файлов в лабораторной работе

В C++ существует два основных вида файлов:

- файлы проектов C++ в формате .vcxproj. В нем хранятся зависимости проекта, используемые файлы исходных кодов и заголовочных файлов, настройки сборки приложение и др;
- заголовочные файлы. В них описывается сигнатура структуры, классов или отдельных функций;
- файлы с исходным кодов. В них хранится вся реализация, описанная в заголовочном файле.

Рассмотрим на примере первой лабораторной работы структуру файлов. На рисунке 2 изображен пример файловой структуры в первой лабораторной работе.

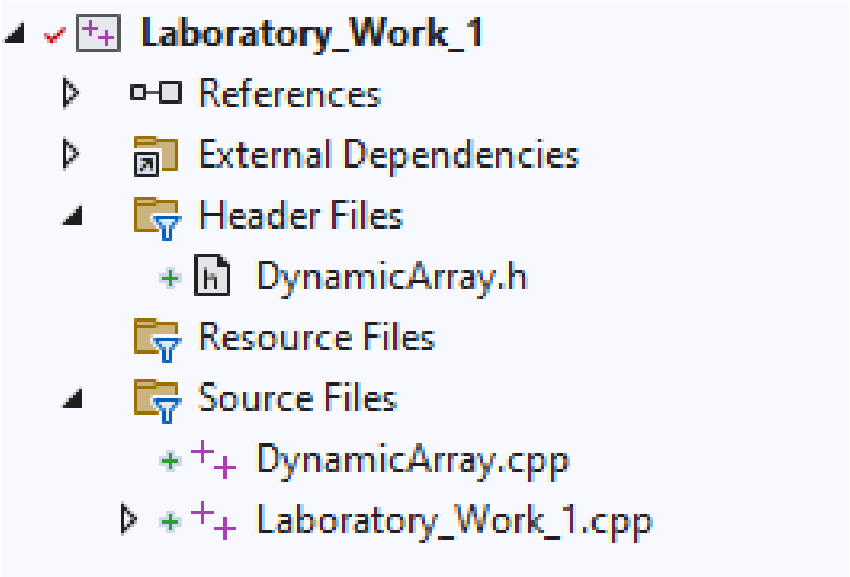

Рисунок 2 – Файловая структура первой лабораторной работы

- DynamicArray.h – заголовочный файл структуры динамического массива;
- DynamicArray.cpp – файл с исходным кодом структуры динамического массива;
- Laboratory_Work_1.cpp – файл с исходным кодом точки входа в программу, а также работа с вводом/выводом пользователю информации.

Важно отметить то, что внутри структуры динамического массива не должно быть работы с вводом/выводом информации пользователем. Данная структура должна заниматься только работой со самой структурой и хранить значения массива.
Рассмотрим структуру внутри папки с проектом первой лабораторной работы. На рисунке 3 представлена структура файлов в проводнике проекта с первой лабораторной работы.

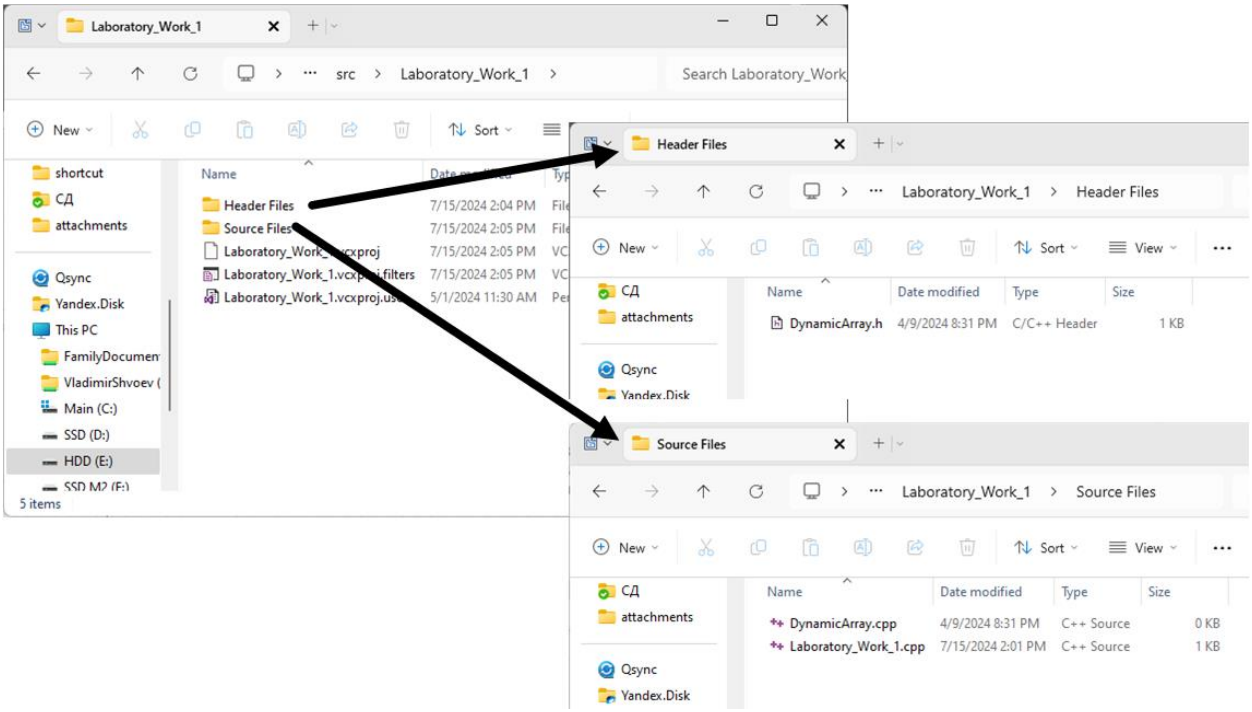

Рисунок 3 – Структура файлов в проводнике

Для того чтобы создавать файлы по нужным физическим путям приложения, нужно указать этот путь в поле (рисунок 4).

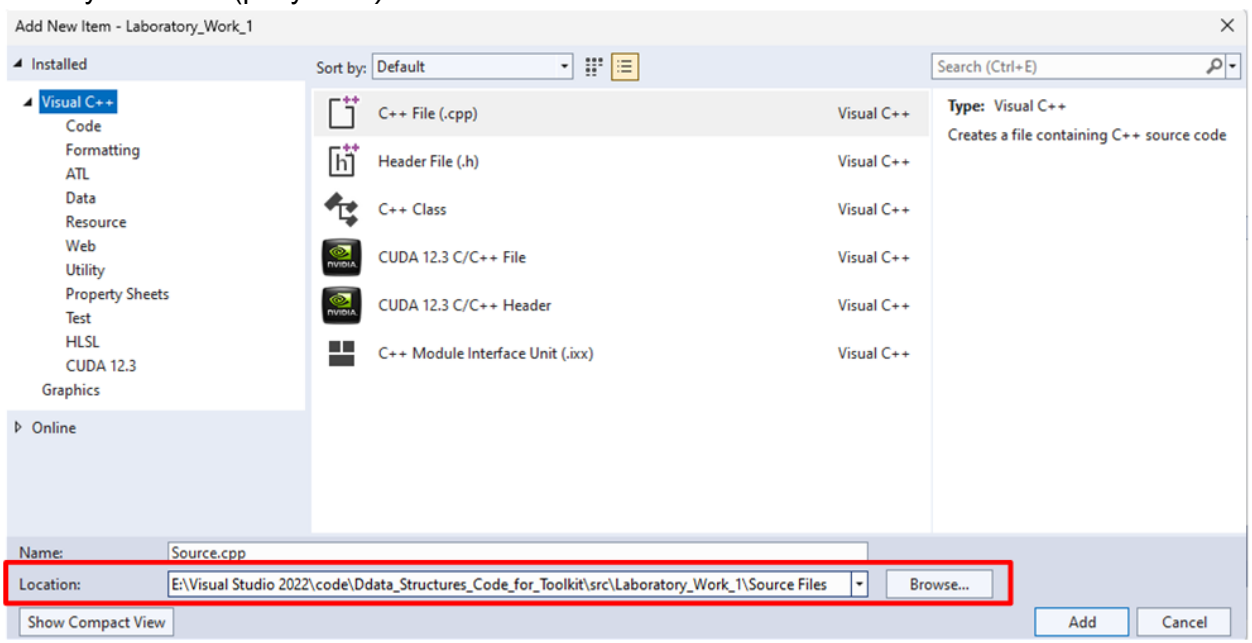

Рисунок 4 – Окно создания файла

Амортизационный анализ

Амортизационный анализ — это метод анализа вычислительной сложности алгоритмов, который используется для оценки среднего времени выполнения операций в последовательности. Этот метод особенно полезен, когда отдельные операции могут занимать разное время, но в среднем они выполняются быстрее[1].

Один из наиболее известных примеров использования амортизационного анализа — это динамический массив (также известный как resizeable array).

Рассмотрим пример динамического массива (про сам динамический массив можно прочитать ниже), который увеличивает свой размер в два раза, когда он заполняется. Например, если начальный размер массива равен 4, то после добавления пятого элемента массив увеличивается до 8.

Операция добавление в динамическом массиве:

1. изначально массив имеет размер 4 и пуст;
2. добавление первых 4 элементов занимает константное время (O(1)) для каждой операции;
3. добавление пятого элемента требует увеличения размера массива:
    3.1. создается новый массив размером 8;
    3.2. все элементы из старого массива копируются в новый массив;
    3.3. новый элемент добавляется в массив.

Амортизационный анализ:

- обычная операция добавление занимает (O(1)) времени;
- операция увеличения размера занимает (O(n)) времени, где (n) — количество элементов в массиве до увеличения.

Чтобы оценить амортизированное время выполнения операции добавление, рассмотрим последовательность из (n) операций:

- каждое добавление элемента занимает (O(1)) времени;
- увеличение размера происходит при добавлении элементов 5, 9, 17 и т.д. (каждый раз, когда массив заполняется).

Амортизированное время одной операции push можно оценить как:

$$T_{\text{амортизированное}} = (n\Theta(1) + \Theta(n))/(n+1) = \Theta(1)$$

Поскольку увеличение размера происходит редко и компенсируется множеством быстрых операций, амортизированное время одной операции добавление остается (O(1)).

Динамический массив

Динамический массив — это абстрактная структура данных, которая позволяет изменять свой размер во время выполнения программы. В отличие от статических массивов, размер которых фиксирован при создании, динамические массивы могут увеличиваться или уменьшаться по мере необходимости[2].

У динамического массива существуют три основных понятия:

1. **вместимость (capacity)** – максимальное количество элементов, которое может хранить массив без необходимости изменения размера. Когда массив достигает своей вместимости, он автоматически увеличивается;
2. **размер массива (size)** – текущее количество элементов в массиве;
3. **значение роста массива (growth factor)** – коэффициент, на который увеличивается вместимость массива при его переполнении. Обычно используется

коэффициент 2 (удвоение вместимости), но могут быть и другие значения, например, 1.5, в зависимости от используемого языка программирования.

На рисунке 5 представлен пример динамического массива, размер которого равен 6, а вместимость равна 9.

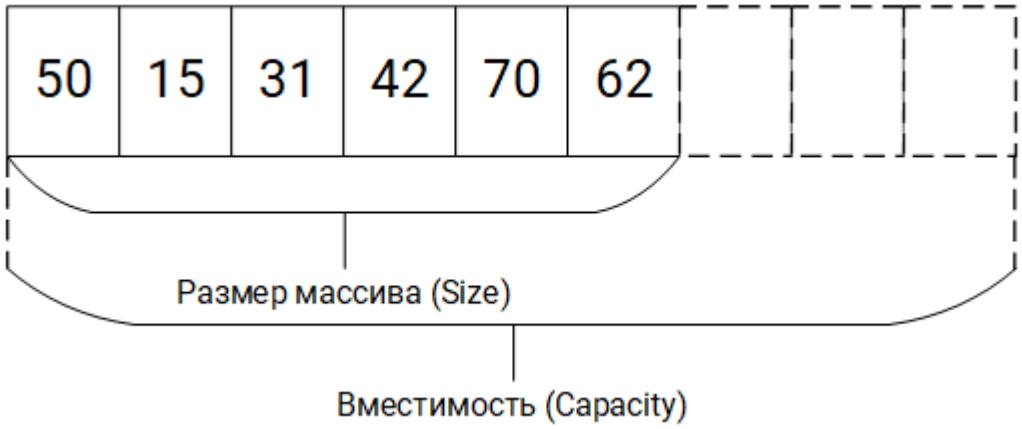

Рисунок 5 – Представление динамического массива

Существует несколько операций над динамическим массивом:

1. **добавление элемента в конец массива** (рисунок 6). Если есть свободное место, операция занимает O(1) времени. Если массив переполнен, требуется увеличение вместимости, что занимает O(n) времени, но в среднем это O(1) благодаря амортизации. По объему дополнительной памяти O(1), но если нужно будет изменить размер массива, то O(n);

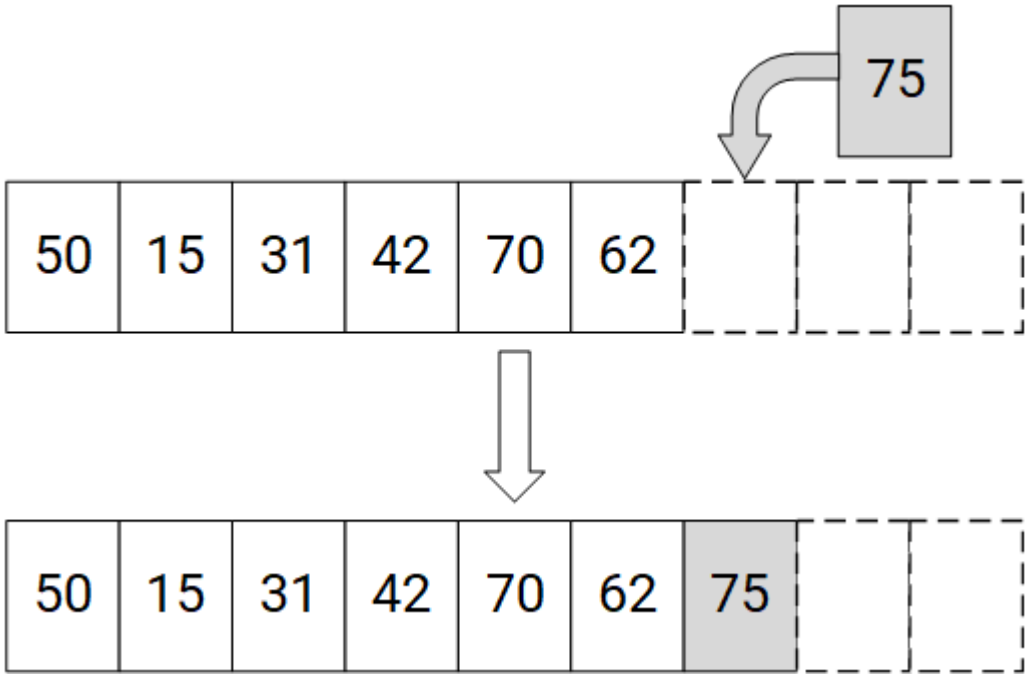

Рисунок 6 – Добавление в конец массива

2. **добавление элемента в произвольное место массива** (рисунок 7). Требуется сдвиг элементов, что занимает O(n) времени. По объему дополнительной памяти O(1), но если нужно будет изменить размер массива, то O(n);

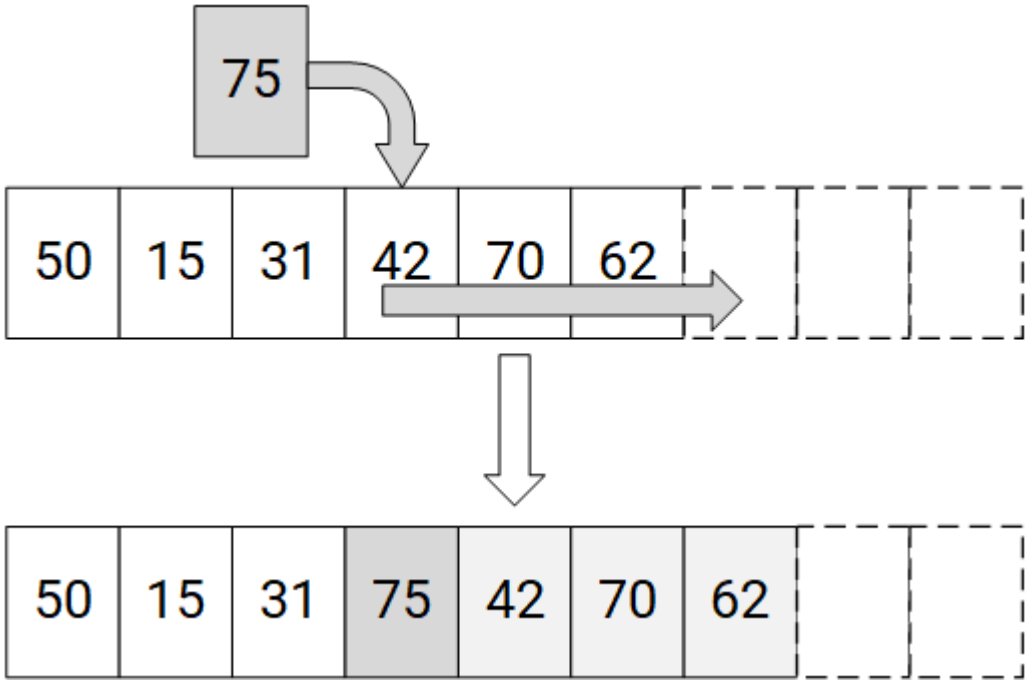

Рисунок 7 – Добавление в произвольное место

3. **удаление элемента по индексу** (рисунок 8). Требуется сдвиг элементов, что занимает O(n) времени. По объему дополнительной памяти O(1);

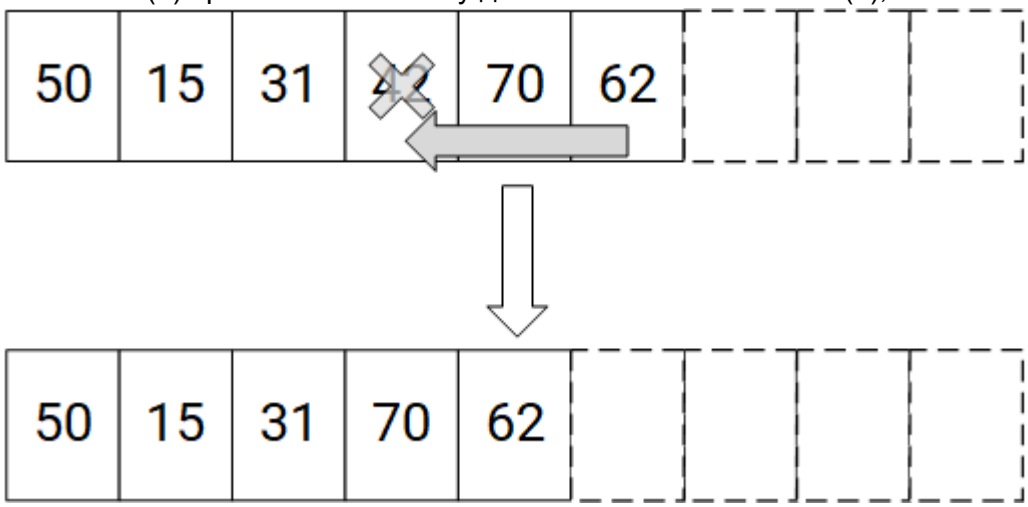

Рисунок 8 – Удаление элемента из массива

4. **удаление элемента по значению**. Требуется поиск элемента и его удаление, что занимает O(n) времени. По объему дополнительной памяти O(1). Существует два варианта удаления: удаление первого вхождения и удаление всех найденных;
5. **получение элемента по индексу**. Операция занимает O(1) времени, так как массив предоставляет доступ по индексу. По объему дополнительной памяти O(1);
6. **изменение размера массива**. Внутреннее поведение структуры, которая автоматически создает новый массив с большей вместимостью и переносит все элементы из старого в новый. Занимает O(n) времени из-за копирования элементов в новый массив. По объему дополнительной памяти O(n).

Для взаимодействия с пользователем программа должно быть реализовано консольное приложение с выбором действий над массивом. Пример такого меню представлен ниже:

```
Laboratory Work #1 – Dynamic Array
```

```
Current array:
12, 3, 8, 25

Select the action you want to do:
1. Remove an element by index from an array
2. Remove an element by value from an array
3. Insert an element at the beginning
4. Insert an element at the end
5. Insert after a certain element
6. Sort array
7. Linear search for an element in an array
8. Binary search for an element in an array

Your input:
```

При вводе некорректного значения должно отображаться сообщение об ошибке пользователю. Пример представлен ниже:

```
Your input: 12
Unknown command. Try entering the command again
Your input: -22
Unknown command. Try entering the command again
Your input: one
Unknown command. Try entering the command again
```

При выборе корректного номера действия над массивом действие выполняется или появляется сообщение об вводе каких либо значений или действий. Например, при удалении элемента по индексу должно появляться сообщение о требовании ввода индекса для удаления. Тут должна проходить проверка о возможности ввода числа, т.к. введенное число может не входить в возможный диапазон для массива. Например, если вводится число 6 при удалении элемента в массиве, который имеет только 3 элемента, то должна выводится ошибка о том, что такой индекс недоступен.

Подходы к разработке

При разработке приложения можно использовать как процедурный подход, так и подход с использованием объектно ориентированного программирования (ООП). В процедурном программировании нужно реализовать структуру, в которой будут только поля: доступный массив, его размер и вместимость. Все остальные функции реализовываются отдельно от структуры. Ниже представлен заголовочный файл такой реализации:

```
 1. //! \brief Структура динамического массива.
 2. struct DynamicArray
 3. {
 4.     //! \brief Размер массива.
 5.     int Size;
 6.
 7.     //! \brief Вместимость массива.
 8.     int Capacity;
 9.
10.     //! \brief Массив.
11.     int* Array;
```

```
};

//! \brief Добавляет элемент в массив.
//! \param array Структура динамического массива.
//! \param index Индекс элемента, куда нужно добавить элемент.
//! \param value Значение элемента.
void AddElement(DynamicArray* array, int index, int value);

//! \brief Удаляет элемент массива по передаваемому индексу.
//! \param array Структура динамического массива.
//! \param index Индекс элемента, который нужно удалить.
void RemoveByIndex(DynamicArray* array, int index);

//! \brief Удаляет значение элемента по его передаваемому значению.
//! \param array Структура динамического массива.
//! \param value Посылаемое значение, которое нужно удалить.
void RemoveByValue(DynamicArray* array, int value);

//! \brief Возвращает элемент по индексу.
//! \param array Структура динамического массива.
//! \param index Индекс, по которому нужно получить значение.
//! \return Возвращает значение, которое находится под индексом.
int GetElement(DynamicArray* array, int index);

//! \brief Сортирует массив.
//! \param array Структура динамического массива.
void SortArray(DynamicArray* array);

//! \brief Линейный поиск индекса элемента по передаваемому значению.
//! \param array Структура динамического массива.
//! \param value Значение, индекс которого нужно найти.
void LinerSearch(DynamicArray* array, int value);

//! \brief Бинарный поиск индекса элемента по передаваемому значению.
//! \param array Структура динамического массива.
//! \param value Значение, индекс которого нужно найти.
void BinarySearch(DynamicArray* array, int value);
```

При использовании ООП, нужно будет также добавить три поля, которые были описаны выше. Доступ к данным полям должен быть приватным и обеспечиваться через специальные методы, называемые сеттеры (от англ. set) и геттеры (от англ. get), которые должны быть реализованы в структуре динамического массива. Также в самой структуре должны быть реализованы методы взаимодействия с динамическим массивом (добавление и удаление элементов, сортировка, методы поиска, методы расширения). Ниже представлен заголовочный файл такой реализации (**в следующих лабораторных будет показываться пример интерфейса структуры только в таком виде**):

```
//! \brief Структура динамического массива.
struct DynamicArray
{
private:
  //! \brief Размер массива.
  int _size;

  //! \brief Вместимость массива.
  int _capacity;

  //! \brief Массив.
```

```
12.   int* _array;
13.
14. public:
15.   //! \brief Возвращает размер массива.
16.   //! \return Размер массива.
17.   int GetSize();
18.
19.   //! \brief Возвращает вместимость массива.
20.   //! \return Вместимость массива.
21.   int GetCapacity();
22.
23.   //! \brief Возвращает массив.
24.   //! \return Массив.
25.   int* GetArray();
26.
27.   //! \brief Добавляет элемент в массив.
28.   //! \param index Индекс элемента, куда нужно добавить элемент.
29.   //! \param value Значение элемента.
30.   void AddElement(int index, int value);
31.
32.   //! \brief Удаляет элемент массива по передаваемому индексу.
33.   //! \param index Индекс элемента, который нужно удалить.
34.   void RemoveByIndex(int index);
35.
36.   //! \brief Удаляет значение элемента по его передаваемому значению.
37.   //! \param value Посылаемое значение, которое нужно удалить.
38.   void RemoveByValue(int value);
39.
40.   //! \brief Возвращает элемент по индексу.
41.   //! \param index Индекс, по которому нужно получить значение.
42.   //! \return Возвращает значение, которое находится под индексом.
43.   int GetElement(int index);
44.
45.   //! \brief Сортирует массив.
46.   void SortArray();
47.
48.   //! \brief Линейный поиск индекса элемента по передаваемому значению.
49.   //! \param value Значение, индекс которого нужно найти.
50.   void LinerSearch(int value);
51.
52.   //! \brief Бинарный поиск индекса элемента по передаваемому значению.
53.   //! \param value Значение, индекс которого нужно найти.
54.   void BinarySearch(int value);
55. };
```

Бинарный поиск

Бинарный (или двоичный) поиск — это эффективный алгоритм для поиска элемента в отсортированном массиве. Он работает по принципу “разделяй и властвуй”, последовательно разделяя массив пополам и сравнивая искомый элемент с серединным элементом массива[3]. Ниже приведены шаги работы алгоритма:

1. Выбираем средний элемент массива.
2. Сравниваем искомое значение со значением в середине массива:
    - Если они равны, то элемент найден.
    - Если искомое значение меньше среднего, повторяем поиск в левой половине массива.

    - Если больше, повторяем поиск в правой половине массива.
3. Повторяем процесс, пока не найдем элемент или не останется элементов для поиска.

Этот алгоритм значительно быстрее линейного поиска, особенно для больших массивов, так как его сложность составляет O(log n).

Задание 2 – Реализация структуры динамический массив

Реализовать структуру данных «Динамический массив» и набор функций для работы с ней. Необходимо реализовать следующие функции:

- Функция создания и инициализации полей массива (length, capacity, array)
- Добавления элемента в массив
- Удаление элемента из массива (по индексу и значению). При удалении нужно каждый раз проверять условие capacity\length < growthFactor. Если данное условие удовлетворяется, то нужно уменьшить массив в growthFactor раз с сохранением значений старого массива
- Вставка элемента в начало
- Вставка элемента в конец
- Вставка после определенного элемента
- Сортировка массива (по вариантам см. ниже)
- Линейный поиск элемента в массиве
- Бинарный поиск элемента в массиве

При работе с массивом предполагается, что изначально выделяется буфер размера по умолчанию (4 или 8). Затем работа с элементами массива идет через реализованные функции. Внутренняя коллекция структуры данных должна быть стандартным типом данных "массив" языка C [4].

Программу необходимо оформить в виде меню. После запуска выводится список того, что можно сделать с массивом. Любой пункт можно выбирать множество раз.

Алгоритм сортировки реализовывается по вариантам по номеру в списке. В данной лабораторной 10 вариантов. Для студента, который находится под номером 11 достается вариант 1, 12 – 2 и т.д. по кругу. Теоретическое описание большей части приведенных алгоритмов сортировки можно найти на сайте вики конспектов ИТМО [5].

Варианты сортировки (с ссылками на примеры реализации):

1. **Пузырьковая сортировка (Bubble Sort)**. Сравнивает соседние элементы и меняет их местами, если они не в порядке. Повторяет процесс до тех пор, пока массив не будет отсортирован [6].
2. **Сортировка выбором (Selection Sort)**. Находит минимальный элемент в неотсортированной части массива и меняет его местами с первым элементом этой части. Повторяет процесс для оставшейся части массива [7].
3. **Сортировка вставками (Insertion Sort)**. Вставляет каждый элемент в его правильное место в уже отсортированной части массива. Повторяет процесс для всех элементов массива [8].
4. **Быстрая сортировка (Quick Sort)**. Выбирает опорный элемент (pivot) и разделяет массив на две части: элементы меньше опорного и элементы больше опорного. Рекурсивно сортирует обе части [9].

5. **Сортировка слиянием (Merge Sort)**. Разделяет массив на две половины, рекурсивно сортирует каждую половину и затем сливает их в один отсортированный массив [10].
6. **Сортировка Шелла (Shell Sort)**. Улучшенная версия сортировки вставками, которая сначала сортирует элементы, находящиеся на определенном расстоянии друг от друга, а затем уменьшает это расстояние [11].
7. **Сортировка расчёской (Comb Sort)**. Улучшенная версия пузырьковой сортировки, которая сначала сравнивает элементы на большом расстоянии друг от друга, постепенно уменьшая это расстояние [12].
8. **Сортировка перемешиванием (Shaker Sort)**. Двунаправленная версия пузырьковой сортировки, которая проходит по массиву сначала слева направо, затем справа налево [13].
9. **Гномья сортировка (Gnome Sort)**. Перемещает элементы по массиву, сравнивая их с предыдущими и меняя местами, если они не в порядке, пока не достигнет конца массива [14].

Вопросы

1. Чем динамический массив отличается от статического?
2. Каким образом происходит увеличение размера динамического массива?
3. Какие преимущества дает использование динамических массивов в программировании?
4. Что такое амортизационный анализ? Приведите пример применения такого анализа.
5. Какие операции позволяют изменять размер динамического массива?
6. Какие примеры использования динамических массивов вы знаете?
7. В каких случаях можно использовать бинарный поиск?
8. Расскажите об алгоритме сортировки выполненного варианта. Приведите пример работы. Какое значение по времени и памяти в нотации “O”?

Используемые источники

1. Lecture 7: Amortized Analysis. Carnegie Mellon University. [Электронный ресурс]: CMU School of Computer Science. 1. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0206.pdf (дата обращения: 04.10.2024).
2. Lambert, Kenneth Alfred (2009), "Physical size and logical size", Fundamentals of Python: From First Programs Through Data Structures, Cengage Learning, p. 510, ISBN 978-1423902188.
3. Задача на поиск элемента в массиве. [Электронный ресурс]: Tproger — издание о разработке и обо всём, что с ней связано. URL: https://tproger.ru/problems/searching-element-in-array (дата обращения: 04.10.2024).
4. Массивы (C++). [Электронный ресурс]: Официальная документация Microsoft для языка программирования C++. URL: https://learn.microsoft.com/ru-ru/cpp/cpp/arrays-cpp?view=msvc-170 (дата обращения: 04.10.2024).
5. Сортировки. [Электронный ресурс]: Конспекты Университета ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8 (дата обращения: 04.10.2024).

6. Bubble Sort Algorithm in C++. [Электронный ресурс]: Free online tutorials. URL: https://www.javatpoint.com/bubble-sort-algorithm-in-cpp (дата обращения: 04.10.2024).
7. C++ Program For Selection Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/cpp-program-for-selection-sort/ (дата обращения: 04.10.2024).
8. Insertion Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.javatpoint.com/insertion-sort-in-cpp (дата обращения: 04.10.2024).
9. Быстрая сортировка. [Электронный ресурс]: Конспекты Университета ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0 (дата обращения: 04.10.2024).
10. Merge Sort – Data Structure and Algorithms Tutorials. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/merge-sort/ (дата обращения: 04.10.2024).
11. Shell Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/shell-sort/ (дата обращения: 04.10.2024).
12. Comb Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/comb-sort/ (дата обращения: 04.10.2024).
13. Cocktail Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/cocktail-sort/ (дата обращения: 04.10.2024).
14. Gnome Sort. [Электронный ресурс]: Free online tutorials. URL: https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/ (дата обращения: 04.10.2024).