|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования |
| **«МИРЭА – Российский технологический университет»** |
| **РТУ МИРЭА** |
|  |

| **Отчет по выполнению практического задания № 7** | |
| --- | --- |
| **Тема:** | |
| **«Рекурсивные алгоритмы и их реализация»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Величко В.Д. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи (В списке №5, Вариант 5) 4](#_1fob9te)

[2.2 Итерационный алгоритм 4](#_3znysh7)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 7](#_2et92p0)

[2.5 Объединение программ 10](#_tyjcwt)

[2.6 Вывод по заданию №1 12](#_3dy6vkm)

[3 ЗАДАНИЕ №2 13](#_1t3h5sf)

[3.1 Формулировка задачи 13](#_4d34og8)

[3.2 Рекурсивная функция 13](#_2s8eyo1)

[3.3 Вывод по заданию №2 16](#_17dp8vu)

[4 ВЫВОДЫ 17](#_3rdcrjn)

[5 ЛИТЕРАТУРА 18](#_26in1rg)

# **1 ЦЕЛЬ**

Получить знания и практические навыки по разработке и реализации рекурсивных процессов.

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи (В списке №5, Вариант 5)**

Разработать и протестировать рекурсивные функции в соответствии с задачами варианта

Требования к выполнению первой задачи варианта:

• приведите итерационный алгоритм решения задачи

• реализуйте алгоритм в виде функции и отладьте его

• определите теоретическую сложность алгоритма

• опишите рекуррентную зависимость в решении задачи

• реализуйте и отладьте рекурсивную функцию решения задачи

• определите глубину рекурсии, изменяя исходные данные

• определите сложность рекурсивного алгоритма, используя метод подстановки и дерево рекурсии

• приведите для одного из значений схему рекурсивных вызовов

• разработайте программу, демонстрирующую выполнение обеих функций и покажите результаты тестирования.

Задание: Дан массив из n элементов вещественного типа. Вычислить среднее значение всех элементов массива.

## **2.2 Итерационный алгоритм**

Итерационный алгоритм - это алгоритм, который использует циклы для повторения последовательности операций определенное количество раз или до выполнения определенного условия. В отличие от рекурсивных алгоритмов, которые вызывают самих себя для решения подзадач, итерационные алгоритмы используют итерации (повторения) для достижения решения.

Создать переменную sum и установить ее значение равным нулю. Она будет использоваться для суммирования всех элементов массива. Используя цикл for, пройти по всем элементам массива arr. На каждой итерации добавить значение текущего элемента массива к переменной sum. После завершения цикла разделить сумму всех элементов на количество элементов в массиве n. Вернуть полученное значение в качестве среднего значения.

Реализуем вычисление значения цифрового корня для некоторого целого числа N с помощью итерационного алгоритма(блок кода 1). Для этого будут использоваться библиотеки iostream и string.

| #include <iostream> using namespace std; // Итерация double averageIterative(double arr[], int n) {  double sum = 0;  for (int i = 0; i < n; ++i) {  sum += arr[i];  }  return sum / n; } int main() {  int size;  cout << "Введите размер массива: ";  cin >> size;   double \*arr = new double[size];  cout << "Введите элементы массива:\n";  for (int i = 0; i < size; ++i) {  cout << "Элемент " << i + 1 << ": ";  cin >> arr[i];  }    cout << "Среднее значение (итерация): " << averageIterative(arr, size) << endl;  delete[] arr;  return 0; } |
| --- |

Блок кода 1 - реализация итерационного алгоритма для задачи 1

Проведем тестирование данного кода на разных значениях (рис. 1-3).

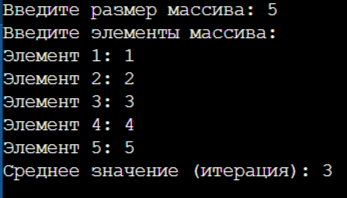


Рисунок 1 - Тестирование алгоритма

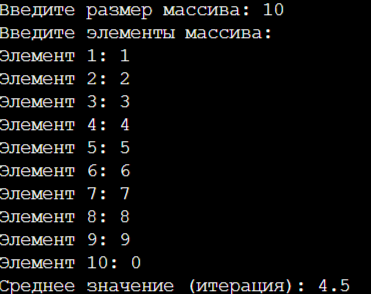


Рисунок 2 - Тестирование алгоритма

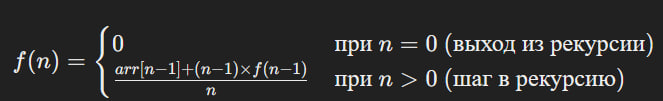
Теоретическая сложность алгоритма определяется количеством операций, которые выполняются в зависимости от размера входных данных, то есть от размера массива n. В данном случае, у нас есть один цикл for, который выполняет n итераций, где n - количество элементов в массиве. Внутри этого цикла выполняется константное количество операций: обращение к элементу массива и выполнение операции сложения. Таким образом, общее количество операций в алгоритме равно n (итераций цикла) + n (обращений к элементам массива) = 2n. Следовательно, теоретическая сложность алгоритма - O(n).

## **2.3 Рекуррентная зависимость**

Рекуррентная зависимость - это связь между последовательными значениями функции (или переменной) в последовательности или алгоритме, которая выражается в виде рекуррентного соотношения или уравнения. Это означает, что текущее значение функции зависит от предыдущих значений.

Рекуррентная зависимость для данной реализации:

Здесь arr[n-1] представляет текущий элемент массива, а f(n-1) представляет среднее значение оставшегося массива. Каждый раз мы добавляем текущий элемент к сумме предыдущих элементов и делим на текущее количество элементов массива n, чтобы получить среднее значение.



## **2.4 Рекурсивная функция**

Рекурсивная функция - это функция, которая вызывает саму себя внутри своего тела. В рекурсивной функции процесс выполнения определенной задачи разбивается на более простые подзадачи, и для решения каждой из подзадач используется та же функция.

Глубина рекурсии - это количество вызовов рекурсивной функции, которые происходят в процессе ее выполнения до достижения базового случая, который завершает рекурсию.

Реализуем вычисление значения цифрового корня для некоторого целого числа N с помощью рекурсивной функции(блок кода 2). Для этого будут использоваться библиотеки iostream и string.

| #include <iostream> using namespace std;  // Рекурсия (считаем сумму элементов массива) double sumRecursive(double arr[], int n) {  if (n == 0) return 0; // если сумма массива равна 0  return arr[n - 1] + sumRecursive(arr, n - 1); }  // Рекурсия (считаем ср знач) double averageRecursive(double arr[], int n) {  return sumRecursive(arr, n) / n; }  int main() {  int size;  cout << "Введите размер массива: ";  cin >> size;   double \*arr = new double[size];  cout << "Введите элементы массива:\n";  for (int i = 0; i < size; ++i) {  cout << "Элемент " << i + 1 << ": ";  cin >> arr[i];  }    cout << "Среднее значение (рекурсия): " << averageRecursive(arr, size) << endl;   delete[] arr;  return 0; } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем тестирование данного кода на разных значениях (рис. 3-4).

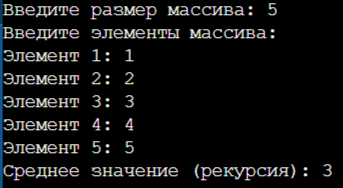


Рисунок 3 - Тестирование рекурсивной программы

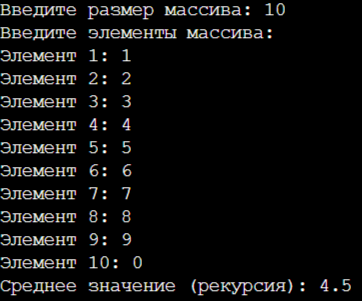


Рисунок 4 - Тестирование рекурсивной программы

Метод подстановки:

Устанавливаем рекурсивную зависимость для времени выполнения функции sumRecursive(arr, n):

T(n)=T(n−1)+O(1)

Разрешаем рекурсивное соотношение:

T(n)=T(n−1)+O(1)=(T(n−2)+O(1))+O(1)=T(n−2)+2×O(1)=T(n−3)+3×O(1)=…=T(1)+(n−1)×O(1)

Поскольку базовый случай выполняется за константное время, T(1)=O(1), то итоговая сложность рекурсивной функции sumRecursive равна O(n).

Теперь, когда мы знаем сложность sumRecursive, мы можем использовать это для определения сложности averageRecursive.

Функция averageRecursive вызывает sumRecursive(arr, n) один раз и выполняет деление на n, что занимает константное время. Таким образом, сложность averageRecursive также составляет O(n).

Данная функция является линейной, поэтому мы можем изобразить её схему так (рисунок 5)

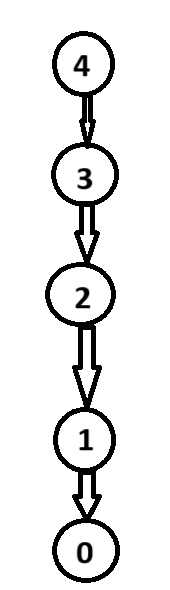


Рисунок 5 - изображение схемы рекурсивных вызовов

Как видно из дерева, глубина рекурсии равна n, и каждый узел выполняет константное количество работы, поэтому общее время выполнения равно O(n), что согласуется с результатами метода подстановки.

Теперь, чтобы определить сложность функции averageRecursive, мы видим, что она вызывает sumRecursive(arr, n) один раз и выполняет деление на n, что также занимает константное время. Следовательно, сложность averageRecursive также составляет O(n).

## **2.5 Объединение программ**

Объединим программы итерационного алгоритма и рекурсивной функции в блок коде 3 с выполнением обоих алгоритмов на заданном числе. Продемонстрируем результаты работы программы на рисунках 8-10.

| #include <iostream> using namespace std; // Итерация double averageIterative(double arr[], int n) {  double sum = 0;  for (int i = 0; i < n; ++i) {  sum += arr[i];  }  return sum / n; }  // Рекурсия (считаем сумму элементов массива) double sumRecursive(double arr[], int n) {  if (n == 0) return 0; // если сумма массива равна 0  return arr[n - 1] + sumRecursive(arr, n - 1); }  // Рекурсия (считаем ср знач) double averageRecursive(double arr[], int n) {  return sumRecursive(arr, n) / n; }  int main() {  int size;  cout << "Введите размер массива: ";  cin >> size;   double \*arr = new double[size];  cout << "Введите элементы массива:\n";  for (int i = 0; i < size; ++i) {  cout << "Элемент " << i + 1 << ": ";  cin >> arr[i];  }    cout << "Среднее значение (итерация): " << averageIterative(arr, size) << endl;  cout << "Среднее значение (рекурсия): " << averageRecursive(arr, size) << endl;   delete[] arr;  return 0; } |
| --- |

Блок кода 3 - Объединение программ

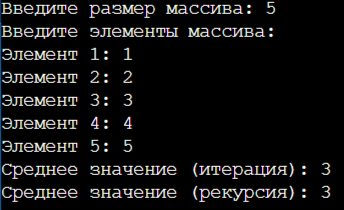


Рисунок 6 - Тестирование программы

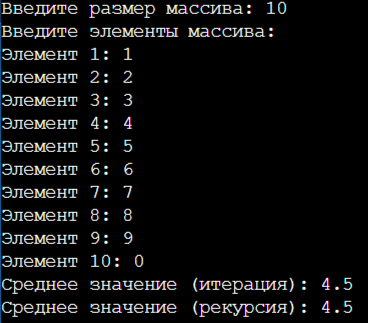


Рисунок 7 - Тестирование программы

## **2.6 Вывод по заданию №1**

Мы увидели, как можно реализовать рекурсивную функцию для вычисления среднего значения всех элементов массива. Рекурсия позволяет нам выразить алгоритм более наглядно и компактно. Анализируя рекурсивную функцию, мы использовали метод подстановки и построили дерево рекурсии, чтобы понять её временную сложность. Это помогло нам понять, как работает рекурсия и как оценить её эффективность. Решение задачи на вычисление среднего значения всех элементов массива помогло нам лучше понять, как алгоритмы могут быть применены к конкретным задачам обработки данных.

В целом, выполнение этого задания позволило нам получить практические навыки в разработке и реализации рекурсивных процессов, а также понять, как рекурсия может быть полезна при обработке массивов данных.

# **3 ЗАДАНИЕ №2**

## **3.1 Формулировка задачи**

Требования к выполнению второй задачи варианта:

• рекурсивную функцию для обработки списковой структуры согласно варианту. Информационная часть узла – простого типа – целого;

• для создания списка может быть разработана простая или рекурсивная функция по желанию (в тех вариантах, где не требуется рекурсивное создание списка);

• определите глубину рекурсии

• определите теоретическую сложность алгоритма

• разработайте программу, демонстрирующую работу функций и покажите результаты тестов.

Задание: Создание связанного стека из n элементов.

## **3.2 Рекурсивная функция**

Реализуем нахождение в двунаправленном списке количество четных элементов с помощью рекурсивной функции(блок кода 4). Создание списка будет разработан простой функцией. Для этого будут использоваться библиотеки iostream.

Глубина рекурсии определяется максимальным количеством вложенных вызовов функции, которые выполняются одновременно. В данном случае, функция push вызывается в цикле for, добавляя элементы в стек. Каждый вызов функции push создает новый узел и добавляет его в стек. Поскольку цикл for выполняется n раз, где n - количество элементов, глубина рекурсии равна n. Таким образом, глубина рекурсии в этом случае равна 5.

Теперь определим теоретическую сложность алгоритма. В данном алгоритме используется два основных метода: push и pop, которые выполняют константное количество операций. Операции в цикле for также выполняются за константное время, поскольку они не зависят от размера входных данных. Таким образом, общая сложность алгоритма равна O(n), где n - количество элементов, добавляемых в стек.

| #include <iostream> using namespace std;  struct Node {  int data;   Node\* next; };  // Добавление элемента в стек void push(Node\*& top, int value) {  Node\* newNode = new Node;   newNode->data = value;  newNode->next = top; // Новый узел указывает на текущий верхний узел  top = newNode; // Верхний узел теперь указывает на новый узел }  // Функция для удаления элемента из стека void pop(Node\*& top) {  if (top == nullptr) {  cout << "Стек пуст" << endl;  return;  }  Node\* temp = top; // Временный указатель на текущий верхний узел  top = top->next; // Верхний узел теперь указывает на следующий узел  delete temp; // Удаление старого верхнего узла }  // Функция для вывода содержимого стека void display(Node\* top) {  if (top == nullptr) {  cout << "Стек пуст" << endl;  return;  }  cout << "Содержимое стека:" << endl;  while (top != nullptr) {  cout << top->data << " ";  top = top->next;  }  cout << endl; }  int main() {  Node\* top = nullptr; // Указатель на верхний элемент стека   // Создание связанного стека  int n = 5;  cout << "Добавление элементов в стек:" << endl;  for (int i = 1; i <= n; ++i) {  push(top, i);  display(top); // Вывод текущего состояния стека после добавления каждого элемента  }   // Удаление элементов из стека  cout << "Удаление элементов из стека:" << endl;  for (int i = 0; i < n; ++i) {  pop(top);  display(top); // Вывод текущего состояния стека после удаления каждого элемента  }   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

Проведем тестирование программы и предоставим результаты на рисунке 11.

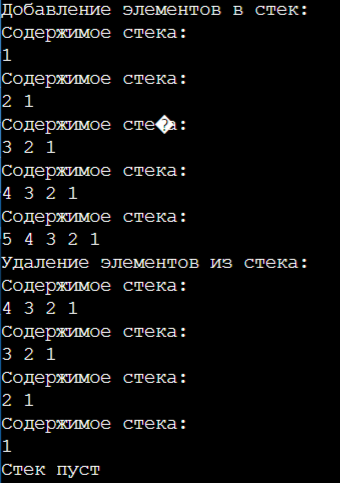


Рисунок 11 - Тестирование программы

## **3.3 Вывод по заданию №2**

Мы изучили, как связанный стек может быть реализован с использованием рекурсии. Рекурсивный подход позволяет нам эффективно управлять структурами данных, такими как стеки, и выполнять различные операции над ними. Мы научились разрабатывать рекурсивные функции для работы со структурами данных. Это позволяет нам писать более компактный и читаемый код, который легче поддерживать и модифицировать. Выполняя задание, мы получили понимание того, как рекурсивные процессы работают и какие могут быть их характеристики, такие как глубина рекурсии и сложность. Полученные знания и навыки могут быть применены в реальных проектах для реализации и управления связанными структурами данных, что делает это задание полезным для развития навыков программирования.

# **4 ВЫВОДЫ**

В ходе практической работы были выполнены следующие задачи:

- Получены знания по разработке и реализации рекурсивных процессов;

- Получены практические навыки по разработке и реализации рекурсивных процессов;

- Были реализованы программы с использованием рекурсивной функции и итерационного алгоритма;

- Были реализовано сравнение программ с использованием рекурсивной функции и итерационного алгоритма;

- Была реализована программа с использованием рекурсивной функции для двунаправленного списка;

-Была рассчитана сложность и глубина рекурсивной функции;

- Проведено тестирование всех программ;

Таким образом, главную цель практической работы, а именно получение знаний и практических навыков по разработке и реализации рекурсивных процессов, можно считать выполненной.

# **5 ЛИТЕРАТУРА**

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).