|  |
| --- |
| 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: Данюков.К.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи (В списке №7, Вариант 7) 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 11](#_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 №7, Вариант 7)**

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: Найти максимальный элемент в массиве из n элементов.

## **2.2 Итерационный алгоритм**

Итерационный алгоритм - это алгоритм, который использует циклы для повторения определенной последовательности операций, прежде чем выполнить определённое условие. В отличие от рекурсивных алгоритмов, которые вызывают самих себя для решения подзадач, итерационные алгоритмы используют итерации (повторения) для достижения решения.

Для определения теоретической сложности алгоритма поиска максимального элемента в массиве из n элементов методом итераций, можно воспользоваться общепринятыми обозначениями сложности. В данном случае, метод итераций означает, что алгоритм будет проходить по всем элементам массива и сравнивать их для определения максимального элемента.

Такой алгоритм будет иметь временную сложность O(n), так как количество операций (сравнений) будет линейно зависеть от размера входного массива. Другими словами, время выполнения алгоритма будет расти пропорционально размеру входного массива - чем больше элементов в массиве, тем больше операций потребуется для нахождения максимального элемента.

Реализуем нахождение максимального элемента для некоторого массива из N элементов с помощью итерационного алгоритма(блок кода 1). Для этого будет использоваться библиотека iostream.

iostream: библиотека в C++, является заголовочным файлом с классами, функциями и переменными для предоставления ввода и вывода данных через стандартные потоки cout (для вывода на консоль) и cin (для ввода с консоли).

| #include <iostream>  int findMax(int arr[], int n) {  int maxElement = arr[0];  for (int i = 1; i < n; i++) {  if (arr[i] > maxElement) {  maxElement = arr[i];  }  }  return maxElement; }  int main() {  int arr[] = {3, 7, 2, 9, 5};  int n = sizeof(arr) / sizeof(arr[0]);   std::cout << "Max element (iterative): " << findMax(arr, n) << std::endl;   return 0; } |
| --- |

Блок схема 1 - реализация итерационного алгоритма для задачи 1

Проведем тестирование данного кода (рис. 1).

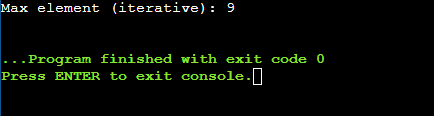


Рисунок 1 - Тестирование алгоритма

## **2.3 Рекуррентная зависимость**

Рекуррентная зависимость - это связь между текущим и предыдущими значениями функции или переменной в последовательности или алгоритме.

Рекуррентная зависимость для нахождения максимального элемента может быть описана следующим образом:

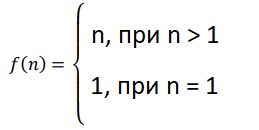


Рисунок 2 - Рекурентная зависимость

Таким образом, если изначально массив содержит больше чем один элемент, каждый рекурсивный вызов будет уменьшать счетчик n на 1 до тех пор, пока n не станет равно 1. В этот момент произойдет выход из рекурсии, и глубина рекурсии будет равна начальному размеру массива.

Если изначально массив содержит только один элемент (то есть n = 1), произойдет только один вызов рекурсивной функции и выход из нее. В этом случае глубина рекурсии будет равна 1.

**2.4 Рекурсивная функция**

Рекурсивная функция - это функция, которая вызывает саму себя внутри своего тела, что позволяет делить выполнение задачи на более простые части, каждая из которых решается с использованием той же функции.

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

В данном случае, функция findMaxRecursive() рекурсивно вызывает саму себя с уменьшающимся размером массива при каждом вызове. Базовый случай достигается, когда размер массива равен 1. Для массива размера n функция будет рекурсивно вызываться n раз. Таким образом, максимальная глубина рекурсии равна значению переменной n.

Давайте посмотрим, как изменится глубина рекурсии при различных исходных данных:

Если пользователь введет массив, размер которого будет равен 1 (например [5]), то функция выполнится 1 раз, так как сразу будет выполнен базовый случай. Глубина рекурсии: 1.

Если пользователь введет массив, размер которого будет равен 5 (например [100, 20, 30, 56, 78]), то функция выполнится 5 раз: сначала для числа 100 (findMaxRecursive([100, 20, 30, 56, 78], 5), потом для числа 20 (findMaxRecursive([20, 30, 56, 78], 4), затем для числа 30 (findMaxRecursive([30, 56, 78], 3), далее для числа 56 (findMaxRecursive([56, 78], 2), и наконец для числа 78 (findMaxRecursive([78], 1). Глубина рекурсии: 5

Если пользователь введет массив, размер которого будет равен 10 (например [1, 5, 8, 23, 32, 78, 98, 22, 3, 88]), то функция выполнится 10 раз. Глубина рекурсии: 10.

Если пользователь введет пустой массив, то функция выполнится 0 раз, так как данная функция не будет вызываться рекурсивно. Глубина рекурсии: 0.

И так далее. Общий принцип состоит в том, что глубина рекурсии равна размеру исходного массива.

В данном случае, рекурсивная функция будет проходиться по массиву и сравнивать каждый элемент с максимальным найденным элементом.

Пусть T(n) - функция, описывающая сложность алгоритма для массива размером n. Тогда, если операция сравнения элементов проводится за постоянное время O(1), рекуррентное соотношение для данного алгоритма будет:

T(n) = T(n-1) + O(1)

Таким образом, сложность алгоритма для нахождения максимального элемента в массиве из n элементов будет O(n), так как на каждом шаге мы уменьшаем размер массива на 1 и выполняем постоянное количество операций (одно сравнение). В итоге, общее количество операций будет линейно зависеть от размера массива n.

Реализуем нахождение максимального элемента для некоторого массива из N элементов с помощью рекурсивной функции(блок кода 2). Для этого будет использоваться библиотека iostream.

| #include <iostream>  int findMaxRecursive(int arr[], int n) {  if (n == 1) {  return arr[0];  }   int maxRest = findMaxRecursive(arr, n - 1);  return (arr[n - 1] > maxRest) ? arr[n - 1] : maxRest; }  int main() {  int arr[] = {3, 7, 2, 9, 5};  int n = sizeof(arr) / sizeof(arr[0]);   std::cout << "Max element (recursive): " << findMaxRecursive(arr, n) << std::endl;   return 0; } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем тестирование данного кода (рис 3).

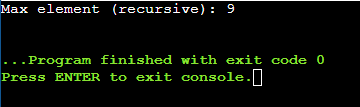


Рисунок 3 - Тестирование адаптированной программы

Изобразить схему рекурсивных вызовов для данной функции можно следующим образом:

findMaxRecursive(arr, 5)

↳ findMaxRecursive(arr, 4)

↳ findMaxRecursive(arr, 3)

↳ findMaxRecursive(arr, 2)

↳ findMaxRecursive(arr, 1)

В данном случае, мы изобразили схему рекуррентных вызовов для n = 5.

## **2.5 Объединение программ**

Объединим программы итерационного алгоритма и рекурсивной функции в блок коде 3 с выполнением обоих алгоритмов на заданном массиве. Продемонстрируем результаты работы программы на рисунке 4.

| #include <iostream>  int findMax(int arr[], int n) {  int maxElement = arr[0];  for (int i = 1; i < n; i++) {  if (arr[i] > maxElement) {  maxElement = arr[i];  }  }  return maxElement; }  int findMaxRecursive(int arr[], int n) {  if (n == 1) {  return arr[0];  }    int maxRest = findMaxRecursive(arr, n - 1);  return (arr[n - 1] > maxRest) ? arr[n - 1] : maxRest; }  int main() {  int arr[] = {3, 7, 2, 9, 5};  int n = sizeof(arr) / sizeof(arr[0]);   std::cout << "Max element (iterative): " << findMax(arr, n) << std::endl;  std::cout << "Max element (recursive): " << findMaxRecursive(arr, n) << std::endl;   return 0; } |
| --- |

Блок кода 3 - Объединение программ

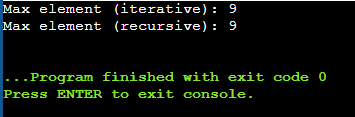


Рисунок 4 - Тестирование программы

## **2.6 Вывод по заданию №1**

Рекурсивный алгоритм использует вызов самого себя для разделения задачи на подзадачи. Он обычно более прост в понимании благодаря натуральности рекурсивного вызова. Однако он может занимать больше памяти из-за создания дополнительных вызовов стека и иметь более высокую временную сложность. При больших данных или глубокой рекурсии он может столкнуться с проблемой переполнения стека.

Итерационный подход использует циклы для выполнения задачи с точки зрения эффективности по времени и памяти, благодаря отсутствию создания дополнительных вызовов стека. Он может быть менее интуитивным для понимания по сравнению с рекурсивным методом, особенно в случае сложных задач. Однако итерационный подход является предпочтительным для обработки больших объемов данных или в условиях ограниченных ресурсов.

# **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 очереди на однонаправленном списке.

## **3.2 Рекурсивная функция**

Реализуем создание очереди на однонаправленном списке с помощью рекурсивной функции(блок кода 4). Создание списка будет разработан простой функцией. Для этого будут использоваться библиотеки iostream.

| #include <iostream>  using namespace std;  // Структура узла очереди struct Node {  int data; // Хранит данные  Node\* next; // Указатель на следующий узел };  // Функция для добавления элемента в конец очереди Node\* addQueue(Node\* head, int value) {  // Если очередь пустая  if (head == nullptr) {  // Создать новый узел  Node\* newNode = new Node;  newNode->data = value; // Присвоить данные новому узлу  newNode->next = nullptr; // Указать, что новый узел является последним  return newNode; // Вернуть новый узел в качестве головы очереди  } else {  // Иначе добавить новый узел в конец очереди  head->next = addQueue(head->next, value); // Рекурсивно вызвать функцию для добавления элемента в конец очереди  return head; // Вернуть голову очереди без изменений  } }  // Функция для вывода элементов очереди void printQueue(Node\* head) {  // Если очередь пустая  if (head == nullptr) {  cout << "Очередь пуста." << endl; // Вывести сообщение об ошибке  return; // Вернуться из функции  }   // Вывести данные текущего узла  cout << head->data << " ";  // Если есть следующий узел  if (head->next != nullptr) {  // Рекурсивно вызвать функцию для вывода следующего узла  printQueue(head->next);  } else {  // Если следующего узла нет, вывести перевод строки  cout << endl;  } }  int main() {  // Создать пустую очередь  Node\* queue = nullptr;   // Добавить элементы в очередь  queue = addQueue(queue, 1);  queue = addQueue(queue, 2);  queue = addQueue(queue, 3);   // Вывести очередь  cout << "Очередь: ";  printQueue(queue);   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

Функция addQueue() рекурсивно вызывает саму себя для добавления нового элемента в очередь. Таким образом, глубина рекурсии будет равна количеству элементов, которые были добавлены в очередь. В данном случае, в функции main мы добавили три элемента в очередь (1, 2 и 3) с использованием рекурсивной функции addQueue(). Поэтому глубина рекурсии для этого кода будет равна 3, так как было выполнено 3 рекурсивных вызова функции addQueue() для добавления трех элементов в очередь.

Теоретическая сложность данного фрагмента кода зависит от числа элементов в очереди, которые нужно пройти перед добавлением нового элемента (рекурсивных вызовов).

Если в очереди уже есть n элементов, то при добавлении нового элемента рекурсивная функция addQueue() будет вызвана n раз, чтобы дойти до конца очереди. Каждый вызов делает константное количество операций (создание нового узла, присваивание значений, переход к следующему узлу).

Таким образом, теоретическая сложность данного кода для добавления нового элемента в очередь составляет O(n), где n - количество элементов в очереди до добавления нового элемента.

Проведем тестирование программы и предоставим результаты на рисунке 5.

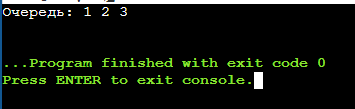


Рисунок 5 - Тестирование программы

## **3.3 Вывод по заданию №2**

Глубина рекурсии в данном случае зависит от конкретной реализации функции для работы со списком. Если для создания списка использовалась рекурсивная функция, то глубина рекурсии будет равна количеству элементов в списке.

Теоретическая сложность алгоритма определяется по количеству операций, необходимых для обработки и управления элементами очереди на однонаправленном списке. В данном случае сложность алгоритма может быть линейной, поскольку каждый элемент списка обрабатывается только один раз.

Программа, демонстрирующая работу функций для создания и управления очередью на однонаправленном списке, может быть использована для тестирования правильности работы алгоритма. Результаты тестов покажут, как система обрабатывает различные сценарии добавления и удаления элементов из очереди.

# **4 ВЫВОДЫ**

В процессе выполнения практической работы были выполнены следующие задачи:

- Получены знания по разработке рекурсивных процессов и реализации программ с их использованием;

- Получены практические навыки по разработке рекурсивных процессов и реализации программ с их использованием;

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

- Было проведено сравнение программ с использованием рекурсивной функции и итерационного алгоритма;

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

-Была рассчитана сложность и глубина рекурсивной функции;

- Проведено тестирование всех программ для проверки их на корректность;

Таким образом, основная цель работы, заключающаяся в усвоении знаний и навыков по работе с рекурсивными процессами, была успешно достигнута.

# **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).

12. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).