|  |
| --- |
| 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: Елисеев И.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 3](#_30j0zll)

[2.1 Формулировка задачи 3](#_1fob9te)

[2.2 Итерационный алгоритм 3](#_2et92p0)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 6](#_17dp8vu)

[2.5 Объединение программ 9](#_1ci93xb)

[3 ЗАДАНИЕ №2 12](#_2bn6wsx)

[3.1 Формулировка задачи 12](#_qsh70q)

[3.2 Рекурсивная функция 12](#_u290pxmzmczz)

[4 ВЫВОДЫ 14](#_ihv636)

[5 ЛИТЕРАТУРА 15](#_32hioqz)

# **1 ЦЕЛЬ**

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

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи**

Вариант 9, в списке 9.

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: Бинарный поиск элемента в массиве

## **2.2 Итерационный алгоритм**

Итерационный алгоритм бинарного поиска элемента в отсортированном массиве можно описать следующим образом. Устанавливаем начальные значения указателей: левый указатель left равен 0, а правый указатель right равен длине массива минус 1. Пока left не превышает right, выполняем следующие шаги. Находим средний индекс массива mid как среднее арифметическое left и right: mid = (left + right) / 2. Если значение элемента в позиции mid равно искомому элементу, возвращаем mid. Если значение элемента в позиции mid больше искомого элемента, сдвигаем правый указатель влево: right = mid - 1. Если значение элемента в позиции mid меньше искомого элемента, сдвигаем левый указатель вправо: left = mid + 1. Если элемент не найден (т.е. left превысил right), возвращаем -1.

Реализуем алгоритм в функции binarySearchIterative в блок кода 1.

| int binarySearchIterative(int arr[], int size, int target) {  int left = 0;  int right = size - 1;   while (left <= right) {  int mid = left + (right - left) / 2;   if (arr[mid] == target)  return mid;  else if (arr[mid] < target)  left = mid + 1;  else  right = mid - 1;  }   return -1; } |
| --- |

Блок код 1 - Реализация итерационного алгоритма для задачи 1

Проведем отладку данной функции на данных 51, 3, 1, 2, 5, 6, 22, 25, 4, 54, 92, 12, 69 с поиском элемента 93 и 22 (рис. 1).

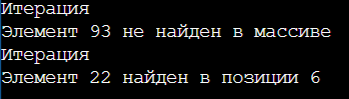


Рисунок 1 - Тестирование алгоритма

Теоретическая сложность итерационного алгоритма бинарного поиска элемента в отсортированном массиве составляет O(logn), где n - количество элементов в массиве.

На каждой итерации алгоритм делит диапазон поиска (интервал между left и right) пополам. Таким образом, на каждой итерации размер оставшегося диапазона уменьшается вдвое. Поскольку мы делим диапазон пополам до тех пор, пока не останется только один элемент или до тех пор, пока left не превысит right, количество итераций не превышает log2n.

Следовательно, теоретическая сложность алгоритма бинарного поиска в отсортированном массиве составляет O(log2n), где n - количество элементов в массиве. Это делает бинарный поиск очень эффективным алгоритмом для поиска элементов в больших отсортированных массивах.

## **2.3 Рекуррентная зависимость**

Рекуррентная зависимость для бинарного поиска элемента в отсортированном массиве:

Базовый случай: если массив имеет длину 0 или 1, то рекурсивный поиск не требуется, и возвращается индекс элемента (если он найден) или -1 (если элемент не найден).

В рекурсивном случае:

* Если средний элемент массива равен искомому элементу, возвращается его индекс.
* Если средний элемент больше искомого, рекурсивный поиск выполняется в левой половине массива.
* Если средний элемент меньше искомого, рекурсивный поиск выполняется в правой половине массива.

Таким образом, рекуррентная зависимость может быть сформулирована следующим образом:

Рекуррентная зависимость:

* При каждом рекурсивном вызове функции бинарного поиска размер проблемы уменьшается вдвое, поскольку мы выбираем только одну половину массива для дальнейшего поиска.
* Базовый случай достигается при размере массива равном 0 или 1, когда больше не требуется делить массив.

Таким образом, рекуррентная зависимость состоит в том, что на каждом шаге размер проблемы уменьшается вдвое, пока не будет достигнут базовый случай.

## **2.4 Рекурсивная функция**

Реализуем рекурсивную функцию для данной задачи (блок кода 2).

| int binarySearchRecursive(int arr[], int left, int right, int target) {  if (left > right)  return -1;   int mid = left + (right - left) / 2;   if (arr[mid] == target)  return mid;  else if (arr[mid] < target)  return binarySearchRecursive(arr, mid + 1, right, target);  else  return binarySearchRecursive(arr, left, mid - 1, target); } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем отладку данной функции на тех же значениях, что и при отладке итерационного алгоритма (рис. 2).

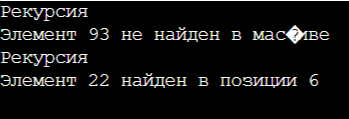


Рисунок 2 - Тестирование адаптированной программы

Глубина рекурсии определяется количеством раз, которое функция делится пополам, прежде чем достигнет базового случая. В бинарном поиске это количество равно количеству раз, которое можно разделить диапазон поиска пополам, чтобы найти искомый элемент.

Метод подстановки:

Пусть T(n) - время выполнения алгоритма для поиска элемента в массиве размера n.

Базовый случай: Если массив состоит из одного элемента (т.е., n = 1), то время выполнения будет константным, т.е., T(1) = O(1).

Рекуррентный случай: В каждой итерации алгоритм делит массив пополам и вызывает сам себя для одной из половин. Поэтому, при каждом шаге размер массива уменьшается вдвое. Таким образом, рекуррентное уравнение будет иметь вид:

T(n) = T(n/2) + O(1)

Дерево рекурсии:

Дерево рекурсии для бинарного поиска элемента в массиве будет иметь высоту log₂(n), так как на каждом уровне размер массива уменьшается вдвое.

Сложность:  
 Согласно методу подстановки, мы видим, что T(n) = O(log n).

Таким образом, сложность рекурсивного алгоритма бинарного поиска элемента в массиве составляет O(log n).

Приведем для массива [2, 4, 6, 8, 10, 12, 14, 16] схему рекурсивных вызовов рекурсии (рис. 3).

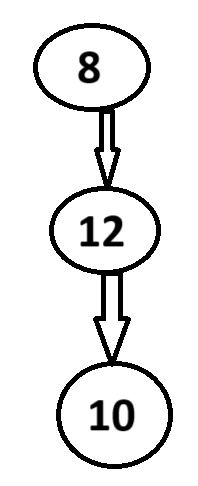


Рисунок 3 - Вызов рекурсии для массива значений

## **2.5 Объединение программ**

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

| #include <iostream> using namespace std;  //1 задание   int binarySearchIterative(int arr[], int size, int target) {  int left = 0;  int right = size - 1;   while (left <= right) {  int mid = left + (right - left) / 2;   if (arr[mid] == target)  return mid;  else if (arr[mid] < target)  left = mid + 1;  else  right = mid - 1;  }   return -1; }  int binarySearchRecursive(int arr[], int left, int right, int target) {  if (left > right)  return -1;   int mid = left + (right - left) / 2;   if (arr[mid] == target)  return mid;  else if (arr[mid] < target)  return binarySearchRecursive(arr, mid + 1, right, target);  else  return binarySearchRecursive(arr, left, mid - 1, target); }  void printSearchResult(int arr[], int size, int target) {  int resultIterative = binarySearchIterative(arr, size, target);  int resultRecursive = binarySearchRecursive(arr, 0, size - 1, target);  cout << "Итерация" << endl;  if (resultIterative != -1)  cout << "Элемент " << target << " найден в позиции " << resultIterative << endl;  else  cout << "Элемент " << target << " не найден в массиве" << endl;    cout << "Рекурсия" << endl;  if (resultRecursive != -1)  cout << "Элемент " << target << " найден в позиции " << resultRecursive << endl;  else  cout << "Элемент " << target << " не найден в массиве" << endl; }  int main() {  int arr[] = {51, 3, 1, 2, 5, 6, 22, 25, 4, 54, 92, 12, 69};   int size = sizeof(arr) / sizeof(arr[0]);   int target = 93;    printSearchResult(arr, size, target);    target = 22;   printSearchResult(arr, size, target);   return 0; } |
| --- |

Блок кода 3 - Объединение программ

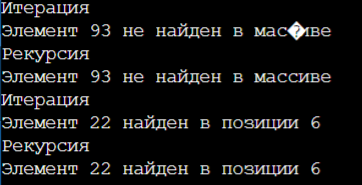


Рисунок 4 - Тестирование программы

# 

# **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 двунаправленного списка

## **3.2 Рекурсивная функция**

Алгоритм:

Базовый случай: Если мы достигли конца списка (т.е., указатель на следующий элемент равен nullptr), то возвращаем указатель на последний элемент.

Рекуррентный случай: Рекурсивно вызываем функцию для создания списка для следующего элемента. Затем, связываем текущий элемент с возвращаемым из рекурсии указателем на следующий элемент.

Реализуем задачу с помощью рекурсивной функции(блок кода 4).

| #include <iostream> using namespace std;  //Задание 2  struct Node {  int data;  Node\* prev;   Node\* next;    Node(int value) : data(value), prev(nullptr), next(nullptr) {} };  // Добавление элементов void addNode(Node\*& head, int value) {  if (head == nullptr) {  head = new Node(value);  } else {  Node\* current = head;  while (current->next != nullptr) {  current = current->next;  }  current->next = new Node(value);  current->next->prev = current;  } }  // Рекурсия обработки void processList(Node\* node) {  if (node == nullptr) {  return;  }  cout << node->data << " ";  processList(node->next); }  int main() {  Node\* head = nullptr;   addNode(head, 14);  addNode(head, 21);  addNode(head, 64);  addNode(head, 142);  addNode(head, 532);   cout << "Содержимое списка: ";  processList(head);  cout << endl;   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

Продемонстрируем результаты работы программы на рисунке 4.



Рисунок 4 - Тестирование программы

Глубина рекурсии:

Глубина рекурсии будет равна количеству элементов в списке, так как для каждого элемента вызывается рекурсивная функция.

Сложность алгоритма:

Пусть n - количество элементов в списке.

Время: Каждый элемент списка посещается ровно один раз, поэтому временная сложность алгоритма составляет O(n).

Память: Для каждого элемента создается узел списка, поэтому память, используемая алгоритмом, также равна O(n).

Таким образом, глубина рекурсии будет равна количеству элементов в списке, а временная и пространственная сложности алгоритма составляют O(n).

# **4 ВЫВОДЫ**

Работая над заданиями "Создание двунаправленного списка" и "Бинарный поиск элемента в массиве", были получены полезные знания и практические навыки в области разработки и реализации рекурсивных процессов.

В ходе выполнения задания "Создание двунаправленного списка" получены навыки использования рекурсии для построения структуры данных. Реализация данного алгоритма помогла лучше понять принципы работы рекурсивных вызовов, а также научилась решать задачи с использованием рекурсивных подходов. Это задание позволило углубить знания о структурах данных и методах их создания.

В задаче "Бинарный поиск элемента в массиве" был применен рекурсивный подход для эффективного поиска элемента в отсортированном массиве. Это позволило лучше понять принципы работы бинарного поиска и его реализацию с использованием рекурсии. Работа над этим заданием позволило улучшить навыки анализа временной и пространственной сложности алгоритмов.

# 

# **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).