Оглавление

[Двусвязный список с# консоль аисд 2](#_Toc167567206)

[Реализация cписка добавить в список вывести с++ консоль 3](#_Toc167567207)

[структура с++ двусвязный список. Консоль с++ Эгов 4](#_Toc167567208)

[Кольцевой спсиок с# консоль 5](#_Toc167567209)

[Бинарное дерево поиска c# консоль 6](#_Toc167567210)

[Граф список смежности 7](#_Toc167567211)

[Матрица смежности 8](#_Toc167567212)

[Быстрая сортировка 9](#_Toc167567213)

[Сортировка слияением 9](#_Toc167567214)

[Пузырьковая сортировка 10](#_Toc167567215)

[Сортировка подсчетом 10](#_Toc167567216)

[Сортировка вставками 11](#_Toc167567217)

[Сортировка выбором 11](#_Toc167567218)

[Сортировка по разрядам 12](#_Toc167567219)

[Откртыая адресация 12](#_Toc167567220)

[ХЭШтаблица метод цепочек 13](#_Toc167567221)

[Обход в глубину 13](#_Toc167567222)

[Обход в ширину 14](#_Toc167567223)

# Двусвязный список с# консоль аисд

using System;

public class DoublyLinkedList

{

// Узел двусвязного списка

public class Node

{

public int Data;

public Node Prev;

public Node Next;

public Node(int data)

{

Data = data;

Prev = null;

Next = null;

}

}

private Node head;

public DoublyLinkedList()

{

head = null;

}

// Добавление данных в конец списка

public void Add(int data)

{

Node newNode = new Node(data);

if (head == null)

{

head = newNode;

return;

}

Node last = head;

while (last.Next != null)

{

last = last.Next;

}

last.Next = newNode;

newNode.Prev = last;

}

// Удаление данных по значению

public void Remove(int data)

{

Node current = head;

// Поиск узла для удаления

while (current != null && current.Data != data)

{

current = current.Next;

}

if (current == null)

{

Console.WriteLine("Элемент не найден.");

return;

}

if (current.Prev != null)

{

current.Prev.Next = current.Next;

}

else

{

head = current.Next;

}

if (current.Next != null)

{

current.Next.Prev = current.Prev;

}

}

// Печать списка

public void PrintList()

{

Node current = head;

while (current != null)

{

Console.Write(current.Data + " ");

current = current.Next;

}

Console.WriteLine();

}

public static void Main(string[] args)

{

DoublyLinkedList dll = new DoublyLinkedList();

while (true)

{

Console.WriteLine("Выберите действие: ");

Console.WriteLine("1. Добавить элемент");

Console.WriteLine("2. Удалить элемент");

Console.WriteLine("3. Показать список");

Console.WriteLine("4. Выход");

Console.Write("Ваш выбор: ");

string choice = Console.ReadLine();

switch (choice)

{

case "1":

Console.Write("Введите значение для добавления: ");

int addValue = int.Parse(Console.ReadLine());

dll.Add(addValue);

break;

case "2":

Console.Write("Введите значение для удаления: ");

int removeValue = int.Parse(Console.ReadLine());

dll.Remove(removeValue);

break;

case "3":

Console.WriteLine("Текущий список:");

dll.PrintList();

break;

case "4":

return;

default:

Console.WriteLine("Неверный выбор, попробуйте снова.");

break;

}

}

}

}

# Реализация cписка добавить в список вывести с++ консоль

#include <iostream>

#include <vector>

#include <algorithm> // для std::sort

using namespace std;

class Reader {

public:

int TicketNumber;

string LastName;

string FirstName;

string MiddleName;

string Address;

Reader(int ticketNumber, string lastName, string firstName, string middleName, string address) :

TicketNumber(ticketNumber), LastName(lastName), FirstName(firstName), MiddleName(middleName), Address(address) {}

};

bool compareReaders(const Reader& r1, const Reader& r2) {

return r1.TicketNumber < r2.TicketNumber;

}

int main() {

vector<Reader> readers;

setlocale(LC\_ALL, "Russian");

while (true) {

cout << "Выберите действие:" << endl;

cout << "1. Добавить читателя" << endl;

cout << "2. Вывести список читателей" << endl;

cout << "3. Выйти из программы" << endl;

int choice;

cin >> choice;

switch (choice) {

case 1: {

int ticketNumber;

string lastName, firstName, middleName, address;

cout << "Введите номер читательского билета: ";

cin >> ticketNumber;

cout << "Введите фамилию: ";

cin >> lastName;

cout << "Введите имя: ";

cin >> firstName;

cout << "Введите отчество: ";

cin >> middleName;

cout << "Введите адрес проживания (только улицу): ";

cin >> address;

Reader newReader(ticketNumber, lastName, firstName, middleName, address);

readers.push\_back(newReader);

// Сортировка списка по номеру читательского билета

sort(readers.begin(), readers.end(), compareReaders);

cout << "Читатель успешно добавлен в список." << endl;

break;

}

case 2: {

if (readers.empty()) {

cout << "Список читателей пуст." << endl;

}

else {

cout << "Список читателей:" << endl;

for (const auto& reader : readers) {

cout << "Номер читательского билета: " << reader.TicketNumber << ", Фамилия: " << reader.LastName

<< ", Имя: " << reader.FirstName << ", Отчество: " << reader.MiddleName << ", Адрес проживания: " << reader.Address << endl;

}

}

break;

}

case 3:

return 0;

default:

cout << "Неверный выбор. Пожалуйста, выберите действие из списка." << endl;

}

}

return 0;

}

# структура с++ двусвязный список. Консоль с++ Эгов

#include <iostream> #include <string>

using namespace std;

// Структура для представления информации о читателе

struct Reader {

int ticketNumber;string lastName; string firstName; string middleName;string address;

// Конструктор для инициализации данных читателя

Reader(int ticketNumber, const string& lastName, const string& firstName, const string& middleName, const string& address) :

ticketNumber(ticketNumber), lastName(lastName), firstName(firstName), middleName(middleName), address(address) {}};

// Структура для элемента двусвязного списка

struct Node {

Reader data;Node\* prev; Node\* next;

Node(const Reader& data) : data(data), prev(nullptr), next(nullptr) {}};

// Класс для двусвязного списка

class DoublyLinkedList {

private:

Node\* head; Node\* tail;

public:

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

// Функция добавления читателя в упорядоченный список по номеру читательского билета

void insertOrdered(const Reader& newReader) {

Node\* newNode = new Node(newReader);

if (!head) { // Если список пуст, добавляем первый элемент

head = newNode; tail = newNode;

} else {

Node\* current = head;

// Находим место для вставки нового читателя

while (current && current->data.ticketNumber < newReader.ticketNumber) {

current = current->next;

}

// Вставляем нового читателя в упорядоченное место

if (!current) { // Вставляем в конец списка

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

} else if (current == head) { // Вставляем в начало списка

newNode->next = head;

head->prev = newNode;

head = newNode;

} else { // Вставляем между двумя узлами

newNode->next = current;

newNode->prev = current->prev;

current->prev->next = newNode;

current->prev = newNode;

} } }

// Функция для вывода списка на экран

void displayList() {

Node\* current = head;

cout << "Список читателей:" << endl;

while (current) {

cout << "Номер читательского билета: " << current->data.ticketNumber << ", Фамилия: " << current->data.lastName

<< ", Имя: " << current->data.firstName << ", Отчество: " << current->data.middleName

<< ", Адрес проживания: " << current->data.address << endl;

current = current->next; } } };

int main() {

DoublyLinkedList readersList;

while (true) {

int choice;

cout << "Выберите действие:" << endl;

cout << "1. Добавить читателя" << endl;

cout << "2. Вывести список читателей" << endl;

cout << "3. Выйти из программы" << endl;

cin >> choice;

switch (choice) {

case 1: {

int ticketNumber;

string lastName, firstName, middleName, address;

cout << "Введите номер читательского билета: "; cin >> ticketNumber;

cout << "Введите фамилию: ";cin >> lastName;

cout << "Введите имя: ";cin >> firstName;

cout << "Введите отчество: "; cin >> middleName;

cout << "Введите адрес проживания (только улицу): ";

cin >> address;

Reader newReader(ticketNumber, lastName, firstName, middleName, address);

readersList.insertOrdered(newReader);

cout << "Читатель успешно добавлен в список." << endl;

break; }

case 2:

readersList.displayList();

break;

case 3:

return 0;

default:

cout << "Неверный выбор. Пожалуйста, выберите действие из списка." << endl; }} return 0;}

# Кольцевой спсиок с# консоль

using System;

public class CircularLinkedList

{

// Узел кольцевого списка

public class Node

{

public int Data;

public Node Next;

public Node(int data)

{

Data = data;

Next = null;

}

}

private Node tail;

public CircularLinkedList()

{

tail = null;

}

// Добавление данных в конец списка

public void Add(int data)

{

Node newNode = new Node(data);

if (tail == null)

{

tail = newNode;

tail.Next = tail; // Первый узел ссылается на себя

}

else

{

newNode.Next = tail.Next; // Новый узел указывает на голову списка

tail.Next = newNode; // Старый хвост указывает на новый узел

tail = newNode; // Новый узел становится новым хвостом

}

}

// Удаление данных по значению

public void Remove(int data)

{

if (tail == null)

{

Console.WriteLine("Список пуст.");

return;

}

Node current = tail.Next;

Node previous = tail;

do

{

if (current.Data == data)

{

if (current == tail)

{

if (tail == tail.Next) // В списке один узел

{

tail = null;

}

else

{

tail = previous;

tail.Next = current.Next;

}

}

else

{

previous.Next = current.Next;

}

return;

}

previous = current;

current = current.Next;

} while (current != tail.Next);

Console.WriteLine("Элемент не найден.");

}

// Печать списка

public void PrintList()

{

if (tail == null)

{

Console.WriteLine("Список пуст.");

return;

}

Node current = tail.Next;

do

{

Console.Write(current.Data + " ");

current = current.Next;

} while (current != tail.Next);

Console.WriteLine();

}

public static void Main(string[] args)

{

CircularLinkedList cll = new CircularLinkedList();

while (true)

{

Console.WriteLine("Выберите действие: ");

Console.WriteLine("1. Добавить элемент");

Console.WriteLine("2. Удалить элемент");

Console.WriteLine("3. Показать список");

Console.WriteLine("4. Выход");

Console.Write("Ваш выбор: ");

string choice = Console.ReadLine();

switch (choice)

{

case "1":

Console.Write("Введите значение для добавления: ");

int addValue = int.Parse(Console.ReadLine());

cll.Add(addValue);

break;

case "2":

Console.Write("Введите значение для удаления: ");

int removeValue = int.Parse(Console.ReadLine());

cll.Remove(removeValue);

break;

case "3":

Console.WriteLine("Текущий список:");

cll.PrintList();

break;

case "4":

return;

default:

Console.WriteLine("Неверный выбор, попробуйте снова.");

break;

}

}

}

}

Кольцевой спсисок - где последний узел ссылается на первый узел, образуя замкнутую цепь. Это означает, что в кольцевом списке, начиная с любого узла, можно обойти весь список и вернуться к исходному узлу.

1. **CircularLinkedList**: Класс, представляющий сам список, который содержит методы для добавления, удаления и печати узлов.

# Бинарное дерево поиска c# консоль

using System;

public class BinarySearchTree

{

public class Node

{

public int Data;

public Node Left;

public Node Right;

public Node(int data)

{

Data = data;

Left = null;

Right = null;

}

}

private Node root;

public BinarySearchTree()

{

root = null;

}

// Добавление элемента в дерево

public void Add(int data)

{

root = AddRecursive(root, data);

}

private Node AddRecursive(Node node, int data)

{

if (node == null)

{

return new Node(data);

}

if (data < node.Data)

{

node.Left = AddRecursive(node.Left, data);

}

else if (data > node.Data)

{

node.Right = AddRecursive(node.Right, data);

}

return node;

}

// Удаление элемента из дерева

public void Remove(int data)

{

root = RemoveRecursive(root, data);

}

private Node RemoveRecursive(Node node, int data)

{

if (node == null)

{

Console.WriteLine("Элемент не найден.");

return node;

}

if (data < node.Data)

{

node.Left = RemoveRecursive(node.Left, data);

}

else if (data > node.Data)

{

node.Right = RemoveRecursive(node.Right, data);

}

else

{

// Узел с одним или нулем потомков

if (node.Left == null)

{

return node.Right;

}

else if (node.Right == null)

{

return node.Left;

}

// Узел с двумя потомками

node.Data = MinValue(node.Right);

node.Right = RemoveRecursive(node.Right, node.Data);

}

return node;

}

private int MinValue(Node node)

{

int minv = node.Data;

while (node.Left != null)

{

minv = node.Left.Data;

node = node.Left;

}

return minv;

}

// Вывод элементов дерева

public void PrintInOrder()

{

PrintInOrderRecursive(root);

Console.WriteLine();

}

private void PrintInOrderRecursive(Node node)

{

if (node != null)

{

PrintInOrderRecursive(node.Left);

Console.Write(node.Data + " ");

PrintInOrderRecursive(node.Right);

}

}

public static void Main(string[] args)

{

BinarySearchTree bst = new BinarySearchTree();

while (true)

{

Console.WriteLine("Выберите действие: ");

Console.WriteLine("1. Добавить элемент");

Console.WriteLine("2. Удалить элемент");

Console.WriteLine("3. Показать дерево (InOrder)");

Console.WriteLine("4. Выход");

Console.Write("Ваш выбор: ");

string choice = Console.ReadLine();

switch (choice)

{

case "1":

Console.Write("Введите значение для добавления: ");

int addValue = int.Parse(Console.ReadLine());

bst.Add(addValue);

break;

case "2":

Console.Write("Введите значение для удаления: ");

int removeValue = int.Parse(Console.ReadLine());

bst.Remove(removeValue);

break;

case "3":

Console.WriteLine("Дерево (InOrder):");

bst.PrintInOrder();

break;

case "4":

return;

default:

Console.WriteLine("Неверный выбор, попробуйте снова.");

break;

}

}

}

}

это структура данных, которая поддерживает динамический набор упорядоченных элементов и позволяет выполнять операции поиска, вставки и удаления за логарифмическое время в среднем случае.

**(log2 N)**.

Если дерево не сбалансировано, в наихудшем случае сложность достигает O(N), так как в конечном счёте все элементы выстраиваются по одну сторону данного дерева.

# Граф список смежности

Входные данные вершин 1 2 3

Ребе 1 2 2 3

using System;

using System.Collections.Generic;

public class GraphAdjacencyList

{

private Dictionary<int, List<int>> adjacencyList; // Словарь для хранения списков смежности

// Конструктор класса GraphAdjacencyList

public GraphAdjacencyList()

{

adjacencyList = new Dictionary<int, List<int>>(); // Инициализация словаря

}

// Метод для добавления вершины в граф

public void AddVertex(int vertex)

{

if (!adjacencyList.ContainsKey(vertex)) // Проверка, существует ли уже вершина с данным ключом

{

adjacencyList[vertex] = new List<int>(); // Добавление новой вершины с пустым списком смежности

}

}

// Метод для добавления ребра между двумя вершинами

public void AddEdge(int vertex1, int vertex2)

{

if (adjacencyList.ContainsKey(vertex1) && adjacencyList.ContainsKey(vertex2))

{

adjacencyList[vertex1].Add(vertex2); // Добавление vertex2 в список смежности вершины vertex1

adjacencyList[vertex2].Add(vertex1); // Добавление vertex1 в список смежности вершины vertex2 (для неориентированного графа)

}

}

// Метод для вывода графа на экран

public void PrintGraph()

{

foreach (var vertex in adjacencyList)

{

Console.Write(vertex.Key + ": "); // Вывод ключа (вершины)

foreach (var neighbor in vertex.Value)

{

Console.Write(neighbor + " "); // Вывод всех смежных вершин для данной вершины

}

Console.WriteLine(); // Переход на новую строку для следующей вершины

}

}

}

public static void Main(string[] args)

{

GraphAdjacencyList graph = new GraphAdjacencyList();

while (true)

{

Console.WriteLine("Выберите действие: ");

Console.WriteLine("1. Добавить вершину");

Console.WriteLine("2. Добавить ребро");

Console.WriteLine("3. Показать граф");

Console.WriteLine("4. Выход");

Console.Write("Ваш выбор: ");

string choice = Console.ReadLine();

switch (choice)

{

case "1":

Console.Write("Введите вершину: ");

int vertex = int.Parse(Console.ReadLine());

graph.AddVertex(vertex);

break;

case "2":

Console.Write("Введите первую вершину: ");

int vertex1 = int.Parse(Console.ReadLine());

Console.Write("Введите вторую вершину: ");

int vertex2 = int.Parse(Console.ReadLine());

graph.AddEdge(vertex1, vertex2);

break;

case "3":

Console.WriteLine("Граф:");

graph.PrintGraph();

break;

case "4":

return;

default:

Console.WriteLine("Неверный выбор, попробуйте снова.");

break;

}

}

}

}

Вершины и их смежные вершины хранятся в словаре.

Списки смежности используют коллекции для хранения смежных вершин для каждой вершины.  O(1).

Граф в виде списка смежности представляет собой структуру данных, используемую для представления графа. Он состоит из списка вершин и для каждой вершины хранит список смежных с ней вершин.

В списке смежности каждая вершина представляется уникальным идентификатором (например, числом или строкой), а список смежных вершин представляется как список других уникальных идентификаторов.

# Матрица смежности

Входные данные вершин 3

Ребра 0 1 1 2

using System;

public class GraphAdjacencyMatrix

{

private int[,] adjacencyMatrix;

private int vertexCount;

public GraphAdjacencyMatrix(int vertices)

{

vertexCount = vertices;

adjacencyMatrix = new int[vertices, vertices];

}

public void AddEdge(int vertex1, int vertex2)

{

if (vertex1 >= 0 && vertex1 < vertexCount && vertex2 >= 0 && vertex2 < vertexCount)

{

adjacencyMatrix[vertex1, vertex2] = 1;

adjacencyMatrix[vertex2, vertex1] = 1; // Для неориентированного графа

}

}

public void PrintGraph()

{

for (int i = 0; i < vertexCount; i++)

{

for (int j = 0; j < vertexCount; j++)

{

Console.Write(adjacencyMatrix[i, j] + " ");

}

Console.WriteLine();

}

}

public static void Main(string[] args)

{

Console.Write("Введите количество вершин: ");

int vertices = int.Parse(Console.ReadLine());

GraphAdjacencyMatrix graph = new GraphAdjacencyMatrix(vertices);

while (true)

{

Console.WriteLine("Выберите действие: ");

Console.WriteLine("1. Добавить ребро");

Console.WriteLine("2. Показать граф");

Console.WriteLine("3. Выход");

Console.Write("Ваш выбор: ");

string choice = Console.ReadLine();

switch (choice)

{

case "1":

Console.Write("Введите первую вершину: ");

int vertex1 = int.Parse(Console.ReadLine());

Console.Write("Введите вторую вершину: ");

int vertex2 = int.Parse(Console.ReadLine());

graph.AddEdge(vertex1, vertex2);

break;

case "2":

Console.WriteLine("Граф:");

graph.PrintGraph();

break;

case "3":

return;

default:

Console.WriteLine("Неверный выбор, попробуйте снова.");

break;

}

}

}

}

**GraphAdjacencyMatrix** — это класс, который представляет граф с использованием матрицы смежности. Матрица смежности — это двумерный массив, где элемент с индексами **[i, j]** указывает на наличие или отсутствие ребра между вершинами **i** и **j**.

**O(E), где E — общее количество рёбер в графе.**

# Быстрая сортировка

using System;

public class QuickSort

{

// Метод для обмена элементов массива

private static void Swap(int[] array, int left, int right)

{

int temp = array[left];

array[left] = array[right];

array[right] = temp;

}

// Метод для разделения массива на две части

private static int Partition(int[] array, int low, int high)

{

int pivot = array[high]; // Опорный элемент

int i = low - 1; // Индекс меньшего элемента

for (int j = low; j < high; j++)

{

// Если текущий элемент меньше или равен опорному

if (array[j] <= pivot)

{

i++;

Swap(array, i, j);

}

}

Swap(array, i + 1, high);

return i + 1;

}

// Рекурсивный метод быстрой сортировки

private static void Sort(int[] array, int low, int high)

{

if (low < high)

{

int pi = Partition(array, low, high);

// Рекурсивно сортируем элементы до и после разделителя

Sort(array, low, pi - 1);

Sort(array, pi + 1, high);

}

}

// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array);

// Применяем быструю сортировку

Sort(array, 0, array.Length - 1);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

алгоритм, который позволяет отсортировать данные за один проход.

Он основан на классической пузырьковой сортировке и включает следующие шаги:

1. На очередном шаге выбирается опорный элемент — им может быть любой элемент массива.
2. Все остальные элементы массива сравниваются с опорным, и те, которые меньше него, ставятся слева от него, а которые больше или равны — справа.
3. Для двух получившихся блоков массива (меньше опорного и больше либо равны опорному) производится точно такая же операция — выделяется опорный элемент, и всё идёт точно так же, пока в блоке не останется один элемент.

O(nLogn) дучший случай худ-O(Logn)

# Сортировка слияением

using System;

public class MergeSort

{// Метод для слияния двух подмассивов array[left..middle] и array[middle+1..right]

private static void Merge(int[] array, int left, int middle, int right)

{ int n1 = middle - left + 1;

int n2 = right - middle;

// Создаем временные массивы

int[] leftArray = new int[n1];

int[] rightArray = new int[n2];

// Копируем данные во временные массивы

for (int i = 0; i < n1; ++i)

leftArray[i] = array[left + i];

for (int j = 0; j < n2; ++j)

rightArray[j] = array[middle + 1 + j];

// Индексы для обхода временных массивов

int leftIndex = 0, rightIndex = 0;

// Индекс начала объединенного подмассива

int mergeIndex = left;

// Объединяем временные массивы в основной

while (leftIndex < n1 && rightIndex < n2)

{if (leftArray[leftIndex] <= rightArray[rightIndex])

{array[mergeIndex] = leftArray[leftIndex];

leftIndex++;

} else

{ array[mergeIndex] = rightArray[rightIndex];

rightIndex++;

}

mergeIndex++;

} // Копируем оставшиеся элементы leftArray[], если они есть

while (leftIndex < n1)

{

array[mergeIndex] = leftArray[leftIndex];

leftIndex++;

mergeIndex++;

} // Копируем оставшиеся элементы rightArray[], если они есть

while (rightIndex < n2)

{

array[mergeIndex] = rightArray[rightIndex];

rightIndex++;

mergeIndex++;

}

} // Рекурсивный метод сортировки слиянием

private static void Sort(int[] array, int left, int right)

{

if (left < right)

{

int middle = left + (right - left) / 2;

// Рекурсивно сортируем две половины

Sort(array, left, middle);

Sort(array, middle + 1, right);

// Объединяем отсортированные половины

Merge(array, left, middle, right);

}

} // Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{ Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array); // Применяем сортировку слиянием

Sort(array, 0, array.Length - 1);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

**O(n log n)**.Этот алгоритм разделяет исходный массив на две половины, рекурсивно сортирует каждую из половин и затем объединяет их в отсортированный массив.

# Пузырьковая сортировка

using System;

public class BubbleSort

{

// Метод для пузырьковой сортировки

private static void Sort(int[] array)

{

int n = array.Length;

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

// Сравниваем соседние элементы и меняем их местами, если нужно

if (array[j] > array[j + 1])

{

int temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

}

// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array);

// Применяем пузырьковую сортировку

Sort(array);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

O(n^2), где n - количество элементов в массиве.

Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего.

# Сортировка подсчетом

using System;

public class CountingSort

{

public static void Sort(int[] array, int max)

{

int[] count = new int[max + 1];

int[] output = new int[array.Length];

// Подсчет количества вхождений каждого элемента

foreach (int num in array)

{

count[num]++;

} // Суммирование

for (int i = 1; i < count.Length; i++)

{

count[i] += count[i - 1];

}// Упорядочивание элементов в выходном массиве

for (int i = array.Length - 1; i >= 0; i--)

{

output[count[array[i]] - 1] = array[i];

count[array[i]]--;

}// Копирование отсортированных элементов обратно в исходный массив

Array.Copy(output, array, array.Length);

}// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];// Максимальное значение элемента

int max = int.MinValue;

for (int i = 0; i < inputArray.Length; i++)

{

int num = int.Parse(inputArray[i]);

array[i] = num;

if (num > max)

{

max = num;

}

}

Console.WriteLine("Исходный массив:");

PrintArray(array);// Применяем сортировку подсчётом

Sort(array, max);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

Сортировка подсчётом — это алгоритм, который позволяет отсортировать массив без сравнений элементов.

Принцип работы:

1. Создаётся вспомогательный массив, который на старте заполняется нулями.
2. Проходит по всему исходному массиву и проверяется очередное значение в ячейке.
3. Берётся содержимое этой ячейки и увеличивается на единицу значение вспомогательного массива под этим номером.
4. После цикла во вспомогательном массиве хранятся данные о том, сколько раз встречается каждый элемент.
5. Проходит по вспомогательному массиву. Если в очередной ячейке есть значение больше нуля, то в исходный массив столько же раз отправляется номер этой ячейки.

Поскольку алгоритм использует массивы длины k + 1 и n, общее использование пространства алгоритмом также равно O(n + k).

# Сортировка вставками

using System;

public class InsertionSort

{

public static void Sort(int[] array)

{

for (int i = 1; i < array.Length; i++)

{

int key = array[i];

int j = i - 1;

// Перемещаем элементы массива, которые больше ключа, на одну позицию вперед

while (j >= 0 && array[j] > key)

{

array[j + 1] = array[j];

j--;

}

// Вставляем ключ на правильное место в отсортированной части массива

array[j + 1] = key;

}

}

// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array);

// Применяем сортировку вставками

Sort(array);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

алгоритм сортировки, который сортирует массив путем пошагового включения каждого элемента на его правильное место в уже отсортированной части массива. Алгоритм начинает с сравнения второго элемента с первым, и, если они находятся в неправильном порядке, меняет их местами. Затем он сравнивает третий элемент с первыми двумя и перемещает его на соответствующее место, и так далее, до тех пор, пока все элементы не будут включены в отсортированную часть массива.

В худшем случае, когда массив отсортирован в обратном порядке, сложность составляет O(n^2). В лучшем случае, когда массив уже отсортирован, сложность составляет O(n).

# Сортировка выбором

using System;

public class SelectionSort

{

public static void Sort(int[] array)

{

int n = array.Length;

// Проходим по всем элементам массива

for (int i = 0; i < n - 1; i++)

{

// Ищем минимальный элемент в оставшейся части массива

int minIndex = i;

for (int j = i + 1; j < n; j++)

{

if (array[j] < array[minIndex])

{

minIndex = j;

}

}

// Обмениваем минимальный элемент с текущим элементом

int temp = array[minIndex];

array[minIndex] = array[i];

array[i] = temp;

}

}

// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array);

// Применяем сортировку выбором

Sort(array);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

Сортировка выбором (Selection Sort) - это простой алгоритм сортировки, который на каждой итерации находит минимальный (или максимальный) элемент из оставшихся элементов массива и перемещает его в начало (или конец) массива

зависит от числа элементов в массиве. Временная сложность алгоритма составляет O(n^2), где n - количество элементов в массиве.

# Сортировка по разрядам

using System;

using System.Collections.Generic;

public class RadixSort

{

public static void Sort(int[] array)

{

if (array == null || array.Length == 0)

return;

// Находим максимальный элемент массива, чтобы узнать количество разрядов

int max = array[0];

for (int i = 1; i < array.Length; i++)

{

if (array[i] > max)

{

max = array[i];

}

}

// Применяем сортировку подсчётом для каждого разряда

for (int exp = 1; max / exp > 0; exp \*= 10)

{

CountSort(array, exp);

}

}

private static void CountSort(int[] array, int exp)

{

int n = array.Length;

int[] output = new int[n];

int[] count = new int[10];

// Заполняем массив count счетчиками количества различных цифр по указанному разряду

for (int i = 0; i < n; i++)

{

count[(array[i] / exp) % 10]++;

}

// Преобразуем массив count так, чтобы каждый элемент хранил сумму предыдущих элементов

for (int i = 1; i < 10; i++)

{

count[i] += count[i - 1];

}

// Помещаем элементы в правильные позиции в массиве output

for (int i = n - 1; i >= 0; i--)

{

output[count[(array[i] / exp) % 10] - 1] = array[i];

count[(array[i] / exp) % 10]--;

}

// Копируем отсортированные элементы обратно в исходный массив

for (int i = 0; i < n; i++)

{

array[i] = output[i];

}

}

// Метод для вывода массива на экран

private static void PrintArray(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

public static void Main(string[] args)

{

Console.Write("Введите элементы массива (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

int[] array = new int[inputArray.Length];

for (int i = 0; i < inputArray.Length; i++)

{

array[i] = int.Parse(inputArray[i]);

}

Console.WriteLine("Исходный массив:");

PrintArray(array);

// Применяем сортировку Radix Sort

Sort(array);

Console.WriteLine("Отсортированный массив:");

PrintArray(array);

}

}

алгоритм сортировки, который работает с целыми числами и использует стратегию сортировки по разрядам. Он сортирует элементы путем их разбиения на разряды и последующей сортировки этих разрядов. O(n \* k) зависит от числа элементов в массиве (n) и максимальной длины чисел в массиве (k).

# Откртыая адресация

При коллизии элемент вставляется в ближайшую свободную ячейку, вычисленную с помощью второй хэш-функции. В этом методе отсутствуют указатели на дополнительные списки, так что хранение данных более компактное. Однако при загруженных таблицах возможно увеличение времени поиска, так как увеличивается количество проб. Сложность операций в среднем составляет O(1), но может увеличиться при наличии коллизий.

using System;

public class OpenAddressingHashTable

{ private int[] table;

private int size;

private int count;

public OpenAddressingHashTable(int size)

{

this.size = size;

table = new int[size];

count = 0; }

private int HashFunction(int key)

{ return key % size;}

private int SecondHashFunction(int key)

{ // Вторая хэш-функция должна быть взаимно простой с размером таблицы

return 7 - (key % 7);}

public void Insert(int key)

{if (count == size)

{ Console.WriteLine("Хэш-таблица полна, вставка невозможна");

return;

}

int index = HashFunction(key);

int step = SecondHashFunction(key);

while (table[index] != 0)

{

index = (index + step) % size;

}

table[index] = key;

count++;

}

public bool Search(int key)

{

int index = HashFunction(key);

int step = SecondHashFunction(key);

while (table[index] != 0)

{ if (table[index] == key)

return true;

index = (index + step) % size;

}

return false; }

public void Print()

{ for (int i = 0; i < size; i++)

{ if (table[i] != 0)

{

Console.WriteLine($"Index {i}: {table[i]}");

}

}

}

public static void Main(string[] args)

{ Console.Write("Введите элементы для добавления (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

OpenAddressingHashTable hashTable = new OpenAddressingHashTable(inputArray.Length);

foreach (string item in inputArray)

{ int value;

if (int.TryParse(item, out value))

{ hashTable.Insert(value);}

else

{Console.WriteLine($"Некорректное значение: {item}");

}

}

Console.WriteLine("Хэш-таблица:");

hashTable.Print();

}}

# ХЭШтаблица метод цепочек

using System;

using System.Collections.Generic;

public class ChainingHashTable

{ private LinkedList<int>[] table; // массив связанных списков, который будет использоваться для хранения элементов хэш-таблицы

private int size; // размер хэш-таблицы

// Конструктор класса, который инициализирует хэш-таблицу указанным размером и создает пустые связанные списки в каждой ячейке массива

public ChainingHashTable(int size)

{ this.size = size;

table = new LinkedList<int>[size];

for (int i = 0; i < size; i++)

{ table[i] = new LinkedList<int>();

}

} // Хэш-функция, которая возвращает индекс для вставки элемента в хэш-таблицу

private int HashFunction(int key)

{ return key % size;

}

// Метод для вставки элемента в хэш-таблицу

public void Insert(int key)

{int index = HashFunction(key);

table[index].AddLast(key);

}

// Метод для поиска элемента в хэш-таблице

public bool Search(int key)

{ int index = HashFunction(key);

return table[index].Contains(key);

}

// Метод для вывода содержимого хэш-таблицы

public void Print()

{

for (int i = 0; i < size; i++)

{

Console.Write($"Index {i}: ");

foreach (int key in table[i])

{

Console.Write(key + " ");

}

Console.WriteLine();

}

} // Главный метод программы, который запускает демонстрацию хэш-таблицы с ручным вводом данных

public static void Main(string[] args)

{ Console.Write("Введите размер хэш-таблицы: ");

int size = int.Parse(Console.ReadLine());

ChainingHashTable hashTable = new ChainingHashTable(size);

Console.WriteLine("Введите элементы для добавления (через пробел): ");

string input = Console.ReadLine();

string[] inputArray = input.Split(' ');

foreach (string item in inputArray)

{ int value;

if (int.TryParse(item, out value))

{ hashTable.Insert(value);

}

else

{Console.WriteLine($"Некорректное значение: {sitem}");

}

}

Console.WriteLine("Хэш-таблица:");

hashTable.Print();

Console.Write("Введите элемент для поиска: ");

int searchKey = int.Parse(Console.ReadLine());

Console.WriteLine($"Результат поиска элемента {searchKey}: {hashTable.Search(searchKey)}");

}

}

Временная сложность операций (вставка, поиск, удаление) в хэш-таблице методом цепочек в среднем случае составляет O(1 + α), где α - среднее число элементов в каждом списке (фактор загрузки

Хэш-таблица с методом цепочек - это структура данных, использующая массив и списки связанных элементов для разрешения коллизий. Каждый элемент массива представляет собой ячейку хэш-таблицы, которая содержит список элементов (цепочку), у которых хэш-значения совпадают.

Процесс работы хэш-таблицы методом цепочек:

Хэширование: Ключи элементов хэшируются с использованием хэш-функции, чтобы определить индекс ячейки массива, в которую элемент будет вставлен.

Вставка: Элементы вставляются в список, соответствующий вычисленному индексу. Если ключи элементов имеют одинаковые хэш-значения (коллизия), они добавляются в одну и ту же ячейку в виде элементов списка.

Поиск: При поиске элемента хэшируется его ключ и находится соответствующая ячейка. Затем элементы в списке этой ячейки проверяются на наличие искомого ключа.

# Обход в глубину

методов обхода графа или дерева, при котором мы спускаемся вглубь структуры, посещая все вершины на пути до достижения конечной вершины, а затем возвращаемся к предыдущей вершине и продолжаем поиск.

Введите количество вершин в графе: 4

Введите количество ребер в графе: 5

Введите вершины ребер:

Ребро 1: 0 1

Ребро 2: 0 2

Ребро 3: 1 2

Ребро 4: 2 3

Ребро 5: 3 3

using System;

using System.Collections.Generic;

public class Graph

{private int V; // количество вершин

private List<int>[] adj; // массив списков смежности

// Конструктор класса Graph

public Graph(int vertices)

{V = vertices;

adj = new List<int>[V];

for (int i = 0; i < V; i++)

{ adj[i] = new List<int>();

}

}

// Метод добавления ребра в граф

public void AddEdge(int v, int w)

{

adj[v].Add(w); // Добавление вершины w к списку вершины v

}

// Рекурсивная функция обхода графа в глубину

private void DFSUtil(int v, bool[] visited)

{

visited[v] = true; // Помечаем текущую вершину как посещенную

Console.Write(v + " "); // Выводим вершину

// Рекурсивно обходим все вершины, смежные с текущей вершиной

foreach (int i in adj[v])

{

if (!visited[i])

DFSUtil(i, visited);

}

} // Метод для обхода графа в глубину с начальной вершиной v

public void DFS(int v)

{

bool[] visited = new bool[V]; // Массив для отслеживания посещенных вершин

DFSUtil(v, visited); }

// Метод для проверки наличия пути между вершинами v и w

public bool HasPath(int v, int w)

{bool[] visited = new bool[V]; // Массив для отслеживания посещенных вершин

DFSUtil(v, visited); // Запускаем обход графа из вершины v

return visited[w]; // Если вершина w была посещена, значит, есть путь от v до w}

public static void Main(string[] args)

{Console.Write("Введите количество вершин в графе: ");

int V = int.Parse(Console.ReadLine());

Graph graph = new Graph(V);

Console.Write("Введите количество ребер в графе: ");

int E = int.Parse(Console.ReadLine());

Console.WriteLine("Введите вершины ребер:");

for (int i = 0; i < E; i++)

{ Console.Write("Ребро " + (i + 1) + ": ");

string[] edge = Console.ReadLine().Split(' ');

int v1 = int.Parse(edge[0]);

int w1 = int.Parse(edge[1]);

graph.AddEdge(v1, w1);

}

Console.WriteLine("Обход графа в глубину (DFS) с начальной вершиной 0:");

graph.DFS(0);

Console.WriteLine("\n\nПроверка наличия пути между вершинами:");

Console.Write("Введите первую вершину: ");

int v = int.Parse(Console.ReadLine());

Console.Write("Введите вторую вершину: ");

int w = int.Parse(Console.ReadLine());

bool hasPath = graph.HasPath(v, w);

Console.WriteLine(hasPath ? "Путь существует" : "Путь не существует"); }}

# Обход в ширину

using System;

using System.Collections.Generic;

public class Graph

{

private int V; // количество вершин

private List<int>[] adj; // массив списков смежности

// Конструктор класса Graph

public Graph(int vertices)

{

V = vertices;

adj = new List<int>[V];

for (int i = 0; i < V; i++)

{

adj[i] = new List<int>();

}

}

// Метод добавления ребра в граф

public void AddEdge(int v, int w)

{

adj[v].Add(w); // Добавление вершины w к списку вершины v

}

// Рекурсивная функция обхода графа в глубину

private void DFSUtil(int v, bool[] visited)

{

visited[v] = true; // Помечаем текущую вершину как посещенную

Console.Write(v + " "); // Выводим вершину

// Рекурсивно обходим все вершины, смежные с текущей вершиной

foreach (int i in adj[v])

{

if (!visited[i])

DFSUtil(i, visited);

}

}

// Метод для обхода графа в глубину с начальной вершиной v

public void DFS(int v)

{

bool[] visited = new bool[V]; // Массив для отслеживания посещенных вершин

DFSUtil(v, visited);

}

// Метод для проверки наличия пути между вершинами v и w

public bool HasPath(int v, int w)

{

bool[] visited = new bool[V]; // Массив для отслеживания посещенных вершин

DFSUtil(v, visited); // Запускаем обход графа из вершины v

return visited[w]; // Если вершина w была посещена, значит, есть путь от v до w

}

public static void Main(string[] args)

{

Console.Write("Введите количество вершин в графе: ");

int V = int.Parse(Console.ReadLine());

Graph graph = new Graph(V);

Console.Write("Введите количество ребер в графе: ");

int E = int.Parse(Console.ReadLine());

Console.WriteLine("Введите вершины ребер:");

for (int i = 0; i < E; i++)

{

Console.Write("Ребро " + (i + 1) + ": ");

string[] edge = Console.ReadLine().Split(' ');

int v1 = int.Parse(edge[0]);

int w1 = int.Parse(edge[1]);

graph.AddEdge(v1, w1);

}

Console.WriteLine("Обход графа в глубину (DFS) с начальной вершиной 0:");

graph.DFS(0);

Console.WriteLine("\n\nПроверка наличия пути между вершинами:");

Console.Write("Введите первую вершину: ");

int v = int.Parse(Console.ReadLine());

Console.Write("Введите вторую вершину: ");

int w = int.Parse(Console.ReadLine());

bool hasPath = graph.HasPath(v, w);

Console.WriteLine(hasPath ? "Путь существует" : "Путь не существует");

}

}

**Алгоритм поиска в ширину** начинается с определённой вершины, затем исследуются все её соседи на данной глубине и происходит переход к вершинам следующего уровня.

В графах, в отличие от деревьев, могут быть циклы — пути, в которых первая и последняя вершины совпадают. Поэтому необходимо отслеживать посещённые алгоритмом вершины.