**ОАиП ответы на вопросы**

**1. Понятие рекурсии. Условие окончания рекурсивного алгоритма. Целесообразность использования рекурсии. Пример рекурсивного алгоритма.**

Рекурсия — это способ организации вычислительного процесса, при котором функция вызывает сама себя для решения задачи. Важно, чтобы при каждом вызове рекурсивной функции проблема уменьшалась, приближаясь к базовому случаю, который можно решить напрямую без дальнейших вызовов функции.

Условие окончания (или базовый случай) — это условие, при котором рекурсивная функция перестает вызывать саму себя. Это критически важная часть любого рекурсивного алгоритма, так как она предотвращает зацикливание и позволяет функции вернуться из вызовов. Базовый случай обычно проверяет, достигнута ли простейшая форма задачи, которую можно решить без дальнейших рекурсивных вызовов.

Использование рекурсии целесообразно, когда задача может быть естественным образом разделена на подзадачи, схожие с исходной задачей. Это делает код более понятным и лаконичным. Однако, следует учитывать, что рекурсивные алгоритмы могут потреблять много памяти из-за большого числа вложенных вызовов функций, что иногда делает итеративные решения более предпочтительными.

Рассмотрим классический пример — вычисление факториала числа n.

#include <iostream>

int factorial(int n) {

// Базовый случай: если n равно 0 или 1, возвращаем 1

if (n <= 1) {

return 1;

}

// Рекурсивный случай: n умножить на факториал (n-1)

else {

return n \* factorial(n - 1);

}

}

**2. Сортировка массивов. Цель сортировки. Методы сортировки. Критерии оценки эффективности сортировки.**

Сортировка — это процесс упорядочивания элементов массива или списка в определенном порядке, чаще всего в порядке возрастания или убывания. Цель сортировки состоит в упрощении последующих операций поиска, анализа данных и улучшении читаемости и восприятия данных.

Существует множество методов сортировки, каждый из которых имеет свои особенности, преимущества и недостатки. Рассмотрим некоторые из них:

1 Сортировка пузырьком (Bubble Sort). Этот метод заключается в многократном проходе по массиву и обмене соседних элементов, если они стоят в неправильном порядке. Процесс продолжается до тех пор, пока массив не будет полностью отсортирован.

2 Сортировка вставками (Insertion Sort). Метод сортировки вставками предполагает, что массив делится на отсортированную и неотсортированную части. Элементы из неотсортированной части последовательно вставляются в отсортированную часть на правильное место.

3 Сортировка выбором (Selection Sort). Этот метод работает путем поиска наименьшего элемента в массиве и обмена его с первым элементом. Затем процесс повторяется для оставшегося неотсортированного массива.

4 Быстрая сортировка (QuickSort). Быстрая сортировка — это рекурсивный метод, который использует стратегию "разделяй и властвуй". Основная идея заключается в выборе опорного элемента (pivot) и разбиении массива на две части: элементы меньше опорного и элементы больше опорного. Затем каждая часть сортируется рекурсивно.

При оценке эффективности алгоритмов сортировки учитываются следующие критерии:

1 Временная сложность: количество операций, необходимых для сортировки массива. Обычно выражается в терминах O-нотации (например, O(n), O(n^2), O(n log n)).

2 Пространственная сложность: объем дополнительной памяти, необходимой для выполнения сортировки.

3 Стабильность: сохраняет ли алгоритм относительный порядок равных элементов.

4 Адаптивность: насколько хорошо алгоритм справляется с частично отсортированными массивами.

5 Простота реализации: насколько легко реализовать и понять алгоритм.

**3. Сортировка массивов. Простые методы сортировки.**

Простые методы сортировки включают сортировку пузырьком, сортировку вставками и сортировку выбором. Эти методы легко понять и реализовать, но они менее эффективны для больших массивов по сравнению с более сложными алгоритмами сортировки.

Сортировка пузырьком (Bubble Sort). Сортировка пузырьком заключается в многократном проходе по массиву и обмене соседних элементов, если они стоят в неправильном порядке. Этот процесс повторяется до тех пор, пока массив не будет полностью отсортирован.

Алгоритм:

1 Повторяйте проходы по массиву.

2 На каждом проходе сравнивайте соседние элементы.

3 Если элементы находятся в неправильном порядке, меняйте их местами.

4 Повторяйте шаги 1-3, пока массив не будет отсортирован.

Пример кода:

void bubbleSort(int arr[], int n) {

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

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

if (arr[j] > arr[j+1]) {

std::swap(arr[j], arr[j+1]);

}

}

}

}

Сортировка вставками (Insertion Sort). Сортировка вставками предполагает, что массив делится на отсортированную и неотсортированную части. Элементы из неотсортированной части последовательно вставляются в отсортированную часть на правильное место.

Алгоритм:

1 Считайте первый элемент отсортированным.

2 Возьмите следующий элемент и вставьте его в правильное место в отсортированной части массива.

3 Повторяйте шаг 2 для всех элементов неотсортированной части массива.

Пример кода:

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

Сортировка выбором (Selection Sort). Сортировка выбором работает путем поиска наименьшего элемента в массиве и обмена его с первым элементом. Затем процесс повторяется для оставшегося неотсортированного массива.

Алгоритм:

1 Найдите наименьший элемент в массиве.

2 Обменяйте его с первым элементом.

3 Повторяйте шаги 1-2 для оставшейся части массива, начиная со второго элемента.

Пример кода:

void selectionSort(int arr[], int n) {

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

int min\_idx = i;

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

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

std::swap(arr[min\_idx], arr[i]);

}

}

**4. Сортировка массивов. QuickSort. Алгоритм. Достоинства и недостатки метода.**

QuickSort, также известная как быстрая сортировка, — это один из самых эффективных алгоритмов сортировки для больших массивов данных. Алгоритм был разработан Тони Хоаром в 1960 году и использует стратегию "разделяй и властвуй".

Основная идея QuickSort заключается в следующем:

1 Выбирается опорный элемент (pivot) из массива.

2 Массив разбивается на две части: элементы, меньшие опорного, и элементы, большие или равные опорному.

3 QuickSort рекурсивно применяется к каждой части.

Шаги алгоритма:

1 Выбор опорного элемента (pivot): опорный элемент можно выбирать различными способами, например, первым элементом, последним элементом, средним элементом или случайным элементом.

2 Разделение (Partition): массив разделяется на две части так, чтобы все элементы, меньшие опорного, оказались слева от него, а все элементы, большие или равные опорному, — справа.

3 Рекурсивная сортировка: применение QuickSort к каждой из двух частей массива.

Пример кода:

// Функция для разделения массива на две части

int partition(int arr[], int low, int high) {

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

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

for (int j = low; j <= high - 1; j++) {

// Если текущий элемент меньше или равен опорному

if (arr[j] <= pivot) {

i++; // Увеличиваем индекс меньшего элемента

std::swap(arr[i], arr[j]); // Меняем местами элементы

}

}

std::swap(arr[i + 1], arr[high]); // Меняем опорный элемент на правильное место

return (i + 1);

}

// Функция быстрой сортировки

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high); // Индекс опорного элемента

quickSort(arr, low, pi - 1); // Рекурсивная сортировка элементов до опорного

quickSort(arr, pi + 1, high); // Рекурсивная сортировка элементов после опорного

}

}

Нерекурсивный QuickSort. Для реализации нерекурсивного варианта QuickSort используется явное управление стеком. Это позволяет избежать глубоких рекурсивных вызовов, которые могут привести к переполнению стека вызовов в случае больших массивов.

Шаги алгоритма:

1 Инициализация:

– Создайте пустой стек для хранения границ подмассивов.

– Поместите начальные границы всего массива в стек (левая и правая границы).

2 Итерация. Пока стек не пуст, выполняйте следующие шаги:

– Извлеките верхние границы подмассива из стека.

– Выполните разделение (partition) текущего подмассива:

– Выберите опорный элемент (pivot).

– Переместите элементы, меньшие или равные опорному, влево, а элементы, большие — вправо.

– Определите индекс опорного элемента после разделения.

– Поместите границы новых подмассивов в стек, если они содержат более одного элемента:

– Левый подмассив: от левой границы до индекса опорного элемента минус один.

– Правый подмассив: от индекса опорного элемента плюс один до правой границы.

3 Завершение:

– Когда стек станет пустым, массив будет отсортирован.

Пример кода:

// Функция для разделения массива на две части

int partition(int arr[], int low, int high) {

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

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

for (int j = low; j <= high - 1; j++) {

if (arr[j] <= pivot) {

i++;

std::swap(arr[i], arr[j]);

}

}

std::swap(arr[i + 1], arr[high]);

return (i + 1);

}

// Нерекурсивная версия QuickSort с использованием стека

void quickSortIterative(int arr[], int l, int h) {

std::stack<std::pair<int, int>> stack;

// Инициализируем стек начальными значениями

stack.push(std::make\_pair(l, h));

while (!stack.empty()) {

auto range = stack.top();

stack.pop();

int low = range.first;

int high = range.second;

if (low < high) {

int p = partition(arr, low, high);

// Помещаем левую часть в стек

if (p - 1 > low) {

stack.push(std::make\_pair(low, p - 1));

}

// Помещаем правую часть в стек

if (p + 1 < high) {

stack.push(std::make\_pair(p + 1, high));

}

}

}

}

Достоинства:

1 Высокая производительность: QuickSort обычно имеет временную сложность O(n log n) и работает быстрее большинства других сортировок на больших наборах данных.

2 Эффективность использования памяти: QuickSort является внутренним алгоритмом сортировки, то есть он не требует дополнительной памяти для создания новых массивов, кроме рекурсивного стека.

3 Простота реализации: несмотря на то, что алгоритм рекурсивный, его довольно легко реализовать и понять.

Недостатки:

1 Худший случай O(n^2): в худшем случае QuickSort имеет временную сложность O(n^2), которая возникает, когда массив уже отсортирован или все элементы одинаковы, и выбирается неудачный опорный элемент.

2 Рекурсивный алгоритм: QuickSort использует рекурсию, что может привести к переполнению стека вызовов при сортировке очень больших массивов.

3 Неустойчивость: QuickSort не является стабильной сортировкой, что означает, что он не сохраняет порядок равных элементов.

**5. Поиск в массиве и сортировка массивов. Выбор метода сортировки. Линейный, линейный с барьером и двоичный поиск в одномерном массиве.**

См. вопрос 2.

Поиск в массиве — это процесс нахождения позиции элемента с заданным значением в массиве. Существует несколько алгоритмов поиска, каждый из которых имеет свои особенности и области применения.

Линейный поиск. Линейный поиск — это простой метод поиска, при котором каждый элемент массива последовательно сравнивается с искомым значением.

Алгоритм:

1 Начать с первого элемента массива.

2 Сравнить текущий элемент с искомым значением.

3 Если значения совпадают, вернуть индекс текущего элемента.

4 Если значения не совпадают, перейти к следующему элементу.

5 Повторять шаги 2-4 до тех пор, пока не будет найдено совпадение или не закончится массив.

Пример кода:

int linearSearch(int arr[], int n, int x) {

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

if (arr[i] == x) {

return i;

}

}

return -1; // элемент не найден

}

Линейный поиск с барьером. Линейный поиск с барьером улучшает производительность, добавляя "барьер" (элемент-страж) в конец массива, чтобы избежать проверки выхода за границы массива.

Алгоритм:

1 Сохранить последний элемент массива.

2 Установить искомое значение в последний элемент массива.

3 Выполнить линейный поиск.

4 Восстановить последний элемент массива после поиска.

Пример кода:

int linearSearchWithSentinel(int arr[], int n, int x) {

int last = arr[n - 1];

arr[n - 1] = x;

int i = 0;

while (arr[i] != x) {

i++;

}

arr[n - 1] = last;

if (i < n - 1 || arr[n - 1] == x) {

return i;

}

return -1;

}

Двоичный поиск. Двоичный поиск используется для отсортированных массивов и значительно быстрее линейного поиска. Он работает по принципу "разделяй и властвуй", постоянно деля массив пополам и отбрасывая половину массива, которая не может содержать искомое значение.

Алгоритм:

1 Установить начальный и конечный индексы массива.

2 Найти средний элемент массива.

3 Сравнить средний элемент с искомым значением.

4 Если значения совпадают, вернуть индекс среднего элемента.

5 Если искомое значение меньше среднего элемента, повторить поиск в левой половине массива.

6 Если искомое значение больше среднего элемента, повторить поиск в правой половине массива.

7 Повторять шаги 2-6 до тех пор, пока не будет найдено совпадение или начальный индекс не превысит конечный индекс.

Пример кода:

int binarySearch(int arr[], int l, int r, int x) {

while (l <= r) {

int m = l + (r - l) / 2;

if (arr[m] == x) {

return m;

}

if (arr[m] < x) {

l = m + 1;

} else {

r = m - 1;

}

}

return -1;

}

Выбор метода сортировки зависит от множества факторов, включая размер массива, распределение данных, требования к устойчивости и ресурсы памяти. Рассмотрим наиболее распространенные методы сортировки и их характеристики, чтобы понять, какой метод лучше подходит для различных ситуаций.

Основные методы сортировки:

1 Сортировка пузырьком (Bubble Sort)

Преимущества:

– Простота реализации.

– Хорошо работает на небольших массивах.

Недостатки:

– Низкая производительность на больших массивах (O(n^2) в худшем случае).

– Неэффективен для отсортированных или почти отсортированных массивов.

Применение: используется для небольших массивов или в учебных целях для объяснения основных принципов сортировки.

2 Сортировка вставками (Insertion Sort)

Преимущества:

– Простота реализации.

– Эффективен для почти отсортированных массивов.

Недостатки:

– Низкая производительность на больших массивах (O(n^2) в худшем случае).

Применение: подходит для небольших массивов и массивов, которые почти отсортированы.

3 Сортировка выбором (Selection Sort)

Преимущества:

– Простота реализации.

– Наименьшее число обменов элементов по сравнению с пузырьком.

Недостатки:

– Низкая производительность на больших массивах (O(n^2) в худшем случае).

Применение: подходит для небольших массивов и массивов, где операции обмена более дорогие, чем операции сравнения.

4 Быстрая сортировка (QuickSort)

Преимущества:

– Высокая производительность для больших массивов (O(n log n) в среднем случае).

– Эффективность использования памяти.

Недостатки:

– Худший случай O(n^2), если массив уже отсортирован или все элементы одинаковы (можно минимизировать с помощью выбора случайного опорного элемента).

– Неустойчивость (не сохраняет порядок равных элементов).

Применение: широко используется для сортировки больших массивов.

Выбор метода сортировки

1 Маленькие массивы (n < 20):

– Сортировка пузырьком

–Сортировка вставками

– Сортировка выбором

2 Средние массивы (20 <= n <= 1000):

– Сортировка вставками, если массив почти отсортирован

– Быстрая сортировка для общих случаев

3 Большие массивы (n > 1000):

– Быстрая сортировка (QuickSort)

4 Требования к устойчивости:

– Сортировка вставками

5 Частично отсортированные массивы:

– Сортировка вставками

6 Операции обмена более дорогие:

– Сортировка выбором

Выбор метода сортировки зависит от конкретных требований задачи, таких как размер массива, его структура и дополнительные требования (например, устойчивость). Быстрая сортировка является одним из наиболее универсальных и эффективных алгоритмов для большинства случаев, но для маленьких массивов или специфических требований могут быть предпочтительны более простые алгоритмы, такие как сортировка пузырьком, вставками или выбором.

**6.** **Понятие списка, стека и очереди. Понятие рекурсивного типа данных.**

Динамическая структура данных — это структура данных, размер и форма которой могут изменяться во время выполнения программы. В отличие от статических структур данных, таких как массивы, где размер фиксирован при создании, динамические структуры могут расти и уменьшаться по мере необходимости. При добавлении новых элементов структуры под них должна выделяться новая память в куче, а при удалении – освобождаться.

Основные характеристики динамических структур данных:

1 Гибкость размера: элементы могут быть добавлены или удалены в любое время, что позволяет эффективно управлять памятью.

2 Связанные элементы: элементы структуры связаны друг с другом с помощью указателей или ссылок, что обеспечивает легкость в перемещении по структуре.

3 Использование динамической памяти: динамические структуры данных используют функции выделения и освобождения памяти (например, malloc и free в C или new и delete в C++).

Список — это динамическая структура данных, в которой элементы хранятся линейно и связаны друг с другом указателями. Списки могут быть однонаправленными (каждый элемент содержит указатель на следующий элемент) и двунаправленными (каждый элемент содержит указатели на предыдущий и следующий элементы).

Стек (Stack) — это структура данных типа LIFO (Last In, First Out), что означает, что последний добавленный элемент будет извлечен первым. Элементы в стек добавляются и извлекаются только с одного конца, который называется вершиной стека. Пример структуры стека:

struct Stack {

int info; // Информационная часть

Stack \*next; // Указатель на следующий элемент

} \*begin; // Указатель на вершину стека

Основные операции со стеком включают добавление элемента (push), удаление элемента (pop) и просмотр стека (peek).

Очередь (Queue) — это структура данных типа FIFO (First In, First Out), что означает, что первый добавленный элемент будет извлечен первым. В очереди элементы добавляются в конец (tail) и извлекаются из начала (head). Пример структуры очереди:

struct Queue {

int info;

Queue \*next;

} \*begin, \*end; // Указатели на начало и конец очереди

Основные операции с очередью включают добавление элемента в конец (enqueue) и удаление элемента из начала (dequeue).

Рекурсивный тип данных — это тип данных, определение которого включает ссылку на сам тип. Это позволяет создавать структуры данных, такие как деревья и списки, где элементы могут содержать указатели на другие элементы того же типа. Пример рекурсивного типа данных — дерево:

struct TreeNode {

int value;

TreeNode \*left;

TreeNode \*right;

};

Рекурсивные типы данных позволяют легко работать с такими структурами, как деревья, и выполнять операции, требующие рекурсивного обхода или обработки, например, поиск, вставка и удаление элементов.

Пример использования однонаправленного списка:

#include <iostream>

struct Node {

int data;

Node\* next;

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node;

newNode->data = data;

newNode->next = head;

head = newNode;

}

void printList(Node\* head) {

Node\* current = head;

while (current != NULL) {

std::cout << current->data << " ";

current = current->next;

}

}

int main() {

Node\* head = NULL;

insert(head, 1);

insert(head, 2);

insert(head, 3);

printList(head);

return 0;

}

Этот код создает однонаправленный список, вставляет в него элементы и выводит их на экран.

**7. Понятие списка, стека и очереди. Особенности работы со стеками.**

Про понятие списка, стека и очереди см. вопрос 6.

Рассмотрим основные аспекты работы со стеками.

Основные операции со стеком включают:

1 Формирование стека (push): Добавление нового элемента на вершину стека.

2 Извлечение элемента из стека (pop): Удаление элемента с вершины стека и возврат его значения.

3 Просмотр элементов стека: Перебор и вывод значений элементов стека без их удаления.

4 Освобождение памяти: Удаление всех элементов стека и освобождение занятой памяти.

Простейший вид функции, добавляющей элемент в стек:

Stack\* InStack(Stack \*p, int in) {

Stack \*t = new Stack; // Выделяем память для нового элемента

t->info = in; // Заполняем информационную часть

t->next = p; // Устанавливаем указатель на текущую вершину

return t; // Возвращаем указатель на новую вершину

}

Пример вызова функции для добавления элемента:

begin = InStack(begin, a); // Добавление нового элемента 'a' в стек

Функция извлечения элемента из вершины стека (pop):

Stack\* OutStack(Stack\* p, int \*out) {

Stack \*t = p; // Устанавливаем указатель t на вершину p

\*out = p->info; // Копируем значение информационной части

p = p->next; // Перемещаем вершину на следующий элемент

delete t; // Удаляем бывшую вершину

return p; // Возвращаем новую вершину

}

Функция просмотра элементов стека:

void View(Stack \*p) {

Stack \*t = p;

while (t != NULL) {

std::cout << t->info << std::endl; // Выводим информационную часть

t = t->next; // Переход к следующему элементу

}

}

Функция для удаления всех элементов стека и освобождения памяти:

void Del\_All(Stack \*\*p) {

Stack \*t;

while (\*p != NULL) {

t = \*p;

\*p = (\*p)->next;

delete t; // Освобождение памяти

}

}

Пример программы, демонстрирующей основные операции со стеком:

#include <iostream>

struct Stack {

int info;

Stack \*next;

} \*begin = NULL;

// Функция добавления элемента в стек

Stack\* InStack(Stack \*p, int in) {

Stack \*t = new Stack;

t->info = in;

t->next = p;

return t;

}

// Функция извлечения элемента из стека

Stack\* OutStack(Stack\* p, int \*out) {

Stack \*t = p;

\*out = p->info;

p = p->next;

delete t;

return p;

}

// Функция просмотра стека

void View(Stack \*p) {

Stack \*t = p;

while (t != NULL) {

std::cout << t->info << std::endl;

t = t->next;

}

}

// Функция освобождения памяти

void Del\_All(Stack \*\*p) {

Stack \*t;

while (\*p != NULL) {

t = \*p;

\*p = (\*p)->next;

delete t;

}

}

int main() {

int a;

begin = InStack(begin, 1);

begin = InStack(begin, 2);

begin = InStack(begin, 3);

std::cout << "Stack contents:" << std::endl;

View(begin);

begin = OutStack(begin, &a);

std::cout << "Popped element: " << a << std::endl;

std::cout << "Stack contents after pop:" << std::endl;

View(begin);

Del\_All(&begin);

return 0;

}

Этот код демонстрирует создание стека, добавление элементов, извлечение элемента и освобождение памяти.

**8. Понятие списка, стека и очереди. Особенности работы с однонаправленной очередью.**

Про понятие списка, стека и очереди см. вопрос 6.

Рассмотрим основные аспекты работы с однонаправленной очередью.

Основные характеристики однонаправленной очереди:

1 Принцип FIFO: элементы добавляются в конец и извлекаются из начала очереди.

2 Два указателя: для работы с очередью используются два указателя — на первый элемент (начало очереди) и на последний элемент (конец очереди).

Основные операции с очередью

Формирование очереди:

void Create(Spis1 \*\*b, Spis1 \*\*e, int in) {

Spis1 \*t = new Spis1; // Выделение памяти под новый элемент

t->info = in; // Заполнение информационной части

t->next = NULL; // Следующий элемент пока отсутствует

if (\*b == NULL) { // Если очередь пуста

\*b = \*e = t; // Первый элемент является началом и концом

} else {

(\*e)->next = t; // Присоединение нового элемента к концу очереди

\*e = t; // Новый элемент становится концом очереди

}

}

Пример вызова функции для создания очереди:

Create(&begin, &end, in); // Добавление элемента с информацией 'in'

Извлечение элемента из очереди:

int Dequeue(Spis1 \*\*b, Spis1 \*\*e) {

if (\*b == NULL) {

throw std::underflow\_error("Очередь пуста");

}

Spis1 \*t = \*b; // Указатель на первый элемент

int value = t->info; // Сохранение значения информационной части

\*b = (\*b)->next; // Перемещение начала очереди на следующий элемент

if (\*b == NULL) {

\*e = NULL; // Если очередь пуста, обнуляем указатель на конец

}

delete t; // Освобождение памяти

return value;

}

Пример вызова функции для извлечения элемента:

int value = Dequeue(&begin, &end); // Извлечение первого элемента

Просмотр элементов очереди:

void View(Spis1 \*b) {

Spis1 \*t = b;

while (t != NULL) {

std::cout << t->info << " ";

t = t->next;

}

std::cout << std::endl;

}

Пример вызова функции для просмотра очереди:

View(begin); // Вывод всех элементов очереди

Удаление всех элементов очереди и освобождение памяти:

void Clear(Spis1 \*\*b, Spis1 \*\*e) {

Spis1 \*t;

while (\*b != NULL) {

t = \*b;

\*b = (\*b)->next;

delete t;

}

\*e = NULL;

}

Пример вызова функции для очистки очереди:

Clear(&begin, &end); // Удаление всех элементов очереди

Пример программы, демонстрирующей основные операции с однонаправленной очередью:

#include <iostream>

#include <stdexcept>

struct Spis1 {

int info;

Spis1 \*next;

} \*begin = NULL, \*end = NULL;

void Create(Spis1 \*\*b, Spis1 \*\*e, int in) {

Spis1 \*t = new Spis1;

t->info = in;

t->next = NULL;

if (\*b == NULL) {

\*b = \*e = t;

} else {

(\*e)->next = t;

\*e = t;

}

}

int Dequeue(Spis1 \*\*b, Spis1 \*\*e) {

if (\*b == NULL) {

throw std::underflow\_error("Очередь пуста");

}

Spis1 \*t = \*b;

int value = t->info;

\*b = (\*b)->next;

if (\*b == NULL) {

\*e = NULL;

}

delete t;

return value;

}

void View(Spis1 \*b) {

Spis1 \*t = b;

while (t != NULL) {

std::cout << t->info << " ";

t = t->next;

}

std::cout << std::endl;

}

void Clear(Spis1 \*\*b, Spis1 \*\*e) {

Spis1 \*t;

while (\*b != NULL) {

t = \*b;

\*b = (\*b)->next;

delete t;

}

\*e = NULL;

}

int main() {

Create(&begin, &end, 1);

Create(&begin, &end, 2);

Create(&begin, &end, 3);

std::cout << "Queue contents: ";

View(begin);

int value = Dequeue(&begin, &end);

std::cout << "Dequeued element: " << value << std::endl;

std::cout << "Queue contents after dequeue: ";

View(begin);

Clear(&begin, &end);

std::cout << "Queue cleared." << std::endl;

return 0;

}

Этот код демонстрирует создание однонаправленной очереди, добавление элементов, извлечение элемента и очистку очереди.

**9. Двунаправленные очереди. Объявление. Особенности работы.**

Двунаправленная очередь (двухсвязный список) — это структура данных, где каждый элемент содержит указатели на предыдущий и следующий элементы. Это позволяет легко перемещаться по списку в обе стороны. Рассмотрим основные аспекты работы с двунаправленными очередями.

Объявление структуры двунаправленной очереди. Структура элемента двунаправленной очереди включает информационную часть и два указателя: на предыдущий и следующий элементы:

struct Spis2 {

int info;

Spis2 \*prev, \*next;

} \*begin, \*end;

Рассмотрим некоторые функции.

Формирование первого элемента:

void CreateFirstElement(int value) {

Spis2 \*t = new Spis2; // Захват памяти под элемент

t->info = value; // Заполнение информационной части

t->prev = t->next = NULL; // Инициализация указателей

begin = end = t; // Установка указателей начала и конца

}

Добавление элемента в начало:

void AddToBegin(int value) {

Spis2 \*t = new Spis2; // Выделяем память под новый элемент

t->info = value; // Заполняем информационную часть значением value

t->prev = NULL; // Указатель на предыдущий элемент равен NULL, так как это будет новый первый элемент

t->next = begin; // Указатель на следующий элемент указывает на текущий первый элемент очереди

if (begin != NULL) { // Если очередь не пуста

begin->prev = t; // Устанавливаем указатель на предыдущий элемент у текущего первого элемента

}

begin = t; // Устанавливаем указатель начала очереди на новый элемент

if (end == NULL) { // Если очередь была пуста

end = t; // Устанавливаем указатель конца очереди на новый элемент

}

}

Добавление в конец:

void AddToEnd(int value) {

Spis2 \*t = new Spis2; // Выделяем память под новый элемент

t->info = value; // Заполняем информационную часть значением value

t->next = NULL; // Указатель на следующий элемент равен NULL, так как это будет новый последний элемент

t->prev = end; // Указатель на предыдущий элемент указывает на текущий последний элемент очереди

if (end != NULL) { // Если очередь не пуста

end->next = t; // Устанавливаем указатель на следующий элемент у текущего последнего элемента

}

end = t; // Устанавливаем указатель конца очереди на новый элемент

if (begin == NULL) { // Если очередь была пуста

begin = t; // Устанавливаем указатель начала очереди на новый элемент

}

}

Просмотр с начала:

void ViewFromBegin() {

Spis2 \*t = begin; // Устанавливаем указатель t на начало очереди

while (t != NULL) { // Пока текущий элемент не равен NULL

std::cout << t->info << " "; // Выводим информационную часть текущего элемента

t = t->next; // Переход к следующему элементу

}

std::cout << std::endl; // Печатаем перевод строки

}

Просмотр с конца:

void ViewFromEnd() {

Spis2 \*t = end; // Устанавливаем указатель t на конец очереди

while (t != NULL) { // Пока текущий элемент не равен NULL

std::cout << t->info << " "; // Выводим информационную часть текущего элемента

t = t->prev; // Переход к предыдущему элементу

}

std::cout << std::endl; // Печатаем перевод строки

}

Освобождение памяти:

void ClearQueue() {

Spis2 \*t;

while (begin != NULL) { // Пока начало очереди не равно NULL

t = begin; // Устанавливаем t на текущий первый элемент

begin = begin->next; // Переход к следующему элементу

delete t; // Удаляем текущий первый элемент

}

end = NULL; // Устанавливаем конец очереди в NULL

}

Пример программы для работы с двунаправленной очередью:

#include <iostream>

struct Spis2 {

int info;

Spis2 \*prev, \*next;

} \*begin = NULL, \*end = NULL;

void CreateFirstElement(int value) {

Spis2 \*t = new Spis2;

t->info = value;

t->prev = t->next = NULL;

begin = end = t;

}

void AddToBegin(int value) {

Spis2 \*t = new Spis2;

t->info = value;

t->prev = NULL;

t->next = begin;

if (begin != NULL) {

begin->prev = t;

}

begin = t;

if (end == NULL) {

end = t;

}

}

void AddToEnd(int value) {

Spis2 \*t = new Spis2;

t->info = value;

t->next = NULL;

t->prev = end;

if (end != NULL) {

end->next = t;

}

end = t;

if (begin == NULL) {

begin = t;

}

}

void ViewFromBegin() {

Spis2 \*t = begin;

while (t != NULL) {

std::cout << t->info << " ";

t = t->next;

}

std::cout << std::endl;

}

void ViewFromEnd() {

Spis2 \*t = end;

while (t != NULL) {

std::cout << t->info << " ";

t = t->prev;

}

std::cout << std::endl;

}

void ClearQueue() {

Spis2 \*t;

while (begin != NULL) {

t = begin;

begin = begin->next;

delete t;

}

end = NULL;

}

int main() {

CreateFirstElement(1);

AddToBegin(2);

AddToEnd(3);

AddToEnd(4);

std::cout << "Queue from begin: ";

ViewFromBegin();

std::cout << "Queue from end: ";

ViewFromEnd();

ClearQueue();

return 0;

}

**10.** **Циклические списки. Создание. Особенности работы.**

Циклический список — это разновидность связного списка, где последний элемент указывает на первый, образуя кольцо. Рассмотрим основные аспекты создания и работы с циклическими списками, используя методический материал.

Элемент циклического списка может быть объявлен следующим образом:

struct List {

int data; // Информационная часть элемента

List\* next; // Указатель на следующий элемент

};

Для создания циклического списка нам нужно создать элементы и установить указатель последнего элемента на первый. Рассмотрим основные функции для работы с циклическим списком.

Добавление элемента в циклический список:

void addToEnd(List\*& head, int value) {

List\* newNode = new List; // Выделение памяти под новый элемент

newNode->data = value; // Заполнение информационной части

newNode->next = NULL; // Инициализация указателя на следующий элемент

if (head == NULL) {

// Если список пуст, новый элемент становится головой и указывает на себя

head = newNode; // Установка головы на новый элемент

newNode->next = head; // Новый элемент указывает на голову, образуя кольцо

} else {

// Иначе, ищем последний элемент и вставляем новый элемент

List\* temp = head; // Временный указатель для поиска конца списка

while (temp->next != head) { // Поиск последнего элемента

temp = temp->next; // Переход к следующему элементу

}

temp->next = newNode; // Последний элемент указывает на новый элемент

newNode->next = head; // Новый элемент указывает на голову, завершая кольцо

}

}

Просмотр элементов циклического списка:

void viewList(List\* head) {

if (head == NULL) {

// Если список пуст, выводим сообщение

std::cout << "List is empty." << std::endl;

return; // Выход из функции

}

List\* temp = head; // Устанавливаем временный указатель на голову списка

do {

std::cout << temp->data << " "; // Выводим информационную часть текущего элемента

temp = temp->next; // Переход к следующему элементу

} while (temp != head); // Цикл продолжается до возврата к голове списка

std::cout << std::endl; // Печать перевода строки

}

Пример работы с циклическим списком:

#include <iostream>

// Объявление структуры элемента циклического списка

struct List {

int data; // Информационная часть элемента

List\* next; // Указатель на следующий элемент

};

// Функция добавления элемента в конец циклического списка

void addToEnd(List\*& head, int value) {

List\* newNode = new List; // Выделение памяти под новый элемент

newNode->data = value; // Заполнение информационной части

newNode->next = NULL; // Инициализация указателя на следующий элемент

if (head == NULL) {

// Если список пуст, новый элемент становится головой и указывает на себя

head = newNode; // Установка головы на новый элемент

newNode->next = head; // Новый элемент указывает на голову, образуя кольцо

} else {

// Иначе, ищем последний элемент и вставляем новый элемент

List\* temp = head; // Временный указатель для поиска конца списка

while (temp->next != head) { // Поиск последнего элемента

temp = temp->next; // Переход к следующему элементу

}

temp->next = newNode; // Последний элемент указывает на новый элемент

newNode->next = head; // Новый элемент указывает на голову, завершая кольцо

}

}

// Функция просмотра элементов циклического списка

void viewList(List\* head) {

if (head == NULL) {

// Если список пуст, выводим сообщение

std::cout << "List is empty." << std::endl;

return; // Выход из функции

}

List\* temp = head; // Устанавливаем временный указатель на голову списка

do {

std::cout << temp->data << " "; // Выводим информационную часть текущего элемента

temp = temp->next; // Переход к следующему элементу

} while (temp != head); // Цикл продолжается до возврата к голове списка

std::cout << std::endl; // Печать перевода строки

}

// Основная функция для демонстрации работы с циклическим списком

int main() {

List\* head = NULL; // Инициализация пустого списка

// Добавление элементов

addToEnd(head, 1); // Добавление элемента со значением 1 в конец списка

addToEnd(head, 2); // Добавление элемента со значением 2 в конец списка

addToEnd(head, 3); // Добавление элемента со значением 3 в конец списка

addToEnd(head, 4); // Добавление элемента со значением 4 в конец списка

// Просмотр списка

std::cout << "Circular List: ";

viewList(head); // Просмотр всех элементов списка

return 0; // Завершение программы

}

Особенности работы с циклическими списками:

1 Цикличность: последний элемент списка указывает на первый, образуя кольцо.

3 Просмотр и операции: во избежание бесконечных циклов нужно тщательно проверять условия выхода из циклов.

4 Удаление элементов: требует дополнительных проверок, чтобы сохранить цикличность списка.

5 Применение: циклические списки часто используются в задачах, где требуется повторяющийся доступ к элементам, например, круговые очереди или задачи планирования.

Циклические списки обеспечивают гибкость и удобство в работе с повторяющимися структурами данных. Понимание их особенностей и реализация основных операций позволяет эффективно использовать их в различных приложениях.

**11. Древовидные структуры данных. Объявление. Правила создания.**

Древовидные структуры данных представляют собой иерархическую структуру, где каждый узел имеет ноль или более дочерних узлов. Рассмотрим основные аспекты создания и работы с древовидными структурами данных.

Создание дерева включает несколько ключевых шагов, каждый из которых выполняется в соответствии с определенными правилами. Рассмотрим алгоритм создания бинарного дерева пошагово.

1. Прежде чем приступить к созданию дерева, нужно объявить структуру узла дерева. Узел дерева включает информационную часть (данные) и указатели на левое и правое поддеревья:

struct TreeNode {

int data; // Информационная часть узла

TreeNode\* left; // Указатель на левое поддерево

TreeNode\* right; // Указатель на правое поддерево

};

2. Создание нового узла. При добавлении нового узла в дерево выполняется следующее:

– Выделение памяти: создаем новый узел, выделяя память для него.

– Заполнение данных: присваиваем значению информационной части узла переданное значение.

– Инициализация указателей: устанавливаем указатели на левое и правое поддеревья в NULL, так как новый узел пока не имеет дочерних узлов.

Функция создания нового узла:

TreeNode\* createNode(int value) {

TreeNode\* newNode = new TreeNode();

newNode->data = value;

newNode->left = newNode->right = NULL;

return newNode;

}

3. Вставка узла в дерево

Вставка узла в бинарное дерево выполняется рекурсивно, следуя правилу: узлы со значениями меньше текущего узла добавляются в левое поддерево, а узлы со значениями больше или равными текущему узлу добавляются в правое поддерево.

– Проверка пустого дерева: если дерево пусто (текущий узел NULL), новый узел становится корневым узлом дерева.

– Сравнение значений: сравниваем значение нового узла с текущим узлом.

– Рекурсивная вставка: в зависимости от результата сравнения выполняем рекурсивную вставку в левое или правое поддерево.

Функция вставки узла в дерево:

TreeNode\* insert(TreeNode\* node, int value) {

if (node == NULL) {

return createNode(value); // Если дерево пусто, создаем новый узел

}

if (value < node->data) {

node->left = insert(node->left, value); // Вставка в левое поддерево

} else {

node->right = insert(node->right, value); // Вставка в правое поддерево

}

return node;

}

4. Обход дерева. Для проверки корректности вставки и для выполнения различных операций с деревом, например, поиска, можно использовать обход дерева. Наиболее распространенным является инфиксный обход (inorder traversal), который посещает узлы в порядке: левое поддерево, текущий узел, правое поддерево:

void inorder(TreeNode\* node) {

if (node != NULL) {

inorder(node->left); // Рекурсивный обход левого поддерева

std::cout << node->data << " "; // Обработка текущего узла

inorder(node->right); // Рекурсивный обход правого поддерева

}

}

Пример создания бинарного дерева:

#include <iostream>

// Объявление структуры узла дерева

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

};

// Создание нового узла

TreeNode\* createNode(int value) {

TreeNode\* newNode = new TreeNode();

newNode->data = value;

newNode->left = newNode->right = NULL;

return newNode;

}

// Вставка узла в дерево

TreeNode\* insert(TreeNode\* node, int value) {

if (node == NULL) {

return createNode(value); // Создаем новый узел, если дерево пусто

}

if (value < node->data) {

node->left = insert(node->left, value); // Вставка в левое поддерево

} else {

node->right = insert(node->right, value); // Вставка в правое поддерево

}

return node;

}

// Инфиксный обход дерева

void inorder(TreeNode\* node) {

if (node != NULL) {

inorder(node->left); // Рекурсивный обход левого поддерева

std::cout << node->data << " "; // Обработка текущего узла

inorder(node->right); // Рекурсивный обход правого поддерева

}

}

// Основная функция

int main() {

TreeNode\* root = NULL; // Инициализация пустого дерева

// Вставка элементов в дерево

root = insert(root, 5);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 2);

root = insert(root, 4);

root = insert(root, 6);

root = insert(root, 8);

// Инфиксный обход дерева

std::cout << "Inorder traversal: ";

inorder(root);

std::cout << std::endl;

return 0; // Завершение программы

}

**12.** **Двоичное дерево поиска. Создание дерева. Удаление всего дерева.**

См вопрос 11

Функция для удаления всего дерева:

void deleteTree(TreeNode\* node) {

if (node == NULL) return; // Базовый случай: дерево пусто

deleteTree(node->left); // Рекурсивное удаление левого поддерева

deleteTree(node->right); // Рекурсивное удаление правого поддерева

delete node; // Удаление текущего узла

}

**13.** **Двоичное дерево поиска. Симметричный, прямой и обратный обход дерева.**

Обход дерева — это процесс посещения всех узлов дерева в определенном порядке. В двоичном дереве поиска (BST) используются три основных метода обхода: симметричный (inorder), прямой (preorder) и обратный (postorder). Рассмотрим их подробнее.

Симметричный обход (Inorder Traversal). Симметричный обход посещает узлы в следующем порядке: левое поддерево, текущий узел, правое поддерево. Этот обход обеспечивает получение отсортированной последовательности элементов в BST:

void inorder(TreeNode\* node) {

if (node != NULL) {

inorder(node->left); // Рекурсивный обход левого поддерева

std::cout << node->data << " "; // Обработка текущего узла

inorder(node->right); // Рекурсивный обход правого поддерева

}

}

Прямой обход (Preorder Traversal). Прямой обход посещает узлы в следующем порядке: текущий узел, левое поддерево, правое поддерево. Этот обход полезен для создания копии дерева:

void preorder(TreeNode\* node) {

if (node != NULL) {

std::cout << node->data << " "; // Обработка текущего узла

preorder(node->left); // Рекурсивный обход левого поддерева

preorder(node->right); // Рекурсивный обход правого поддерева

}

}

Обратный обход (Postorder Traversal). Обратный обход посещает узлы в следующем порядке: левое поддерево, правое поддерево, текущий узел. Этот обход полезен для удаления дерева:

void postorder(TreeNode\* node) {

if (node != NULL) {

postorder(node->left); // Рекурсивный обход левого поддерева

postorder(node->right); // Рекурсивный обход правого поддерева

std::cout << node->data << " "; // Обработка текущего узла

}

}

**14. Двоичное дерево поиска. Создание дерева. Поиск максимального, минимального значений.**

См. вопрос 11

Поиск минимального значения. Минимальное значение в BST всегда находится в самом левом узле. Мы можем использовать рекурсию или итерацию для нахождения этого узла:

TreeNode\* findMin(TreeNode\* node) {

while (node->left != NULL) {

node = node->left; // Переход к левому поддереву

}

return node; // Возврат самого левого узла

}

Поиск максимального значения. Максимальное значение в BST всегда находится в самом правом узле. Аналогично, используем рекурсию или итерацию для нахождения этого узла:

TreeNode\* findMax(TreeNode\* node) {

while (node->right != NULL) {

node = node->right; // Переход к правому поддереву

}

return node; // Возврат самого правого узла

}

**15. Алгоритм преобразования выражения из инфиксной в формы в форму обратной польской записи.**

Обратная польская запись (ОПЗ) или постфиксная запись — это форма записи арифметических выражений, в которой операторы следуют за своими операндами. Основное преимущество ОПЗ заключается в том, что выражения можно вычислять без учета приоритетов операторов и скобок.

При преобразовании инфиксного выражения в форму обратной польской записи важно учитывать приоритет операторов, чтобы правильно выстроить порядок операций. В арифметических выражениях операторы имеют разный приоритет, который определяет порядок их выполнения. В таблице ниже перечислены операторы и их приоритеты:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Приоритет** |
| ( | Левая скобка | Самый высокий (особый случай) |
| ) | Правая скобка | Самый высокий (особый случай) |
| \*, / | Умножение, деление | Высокий |
| +, - | Сложение, вычитание | Низкий |

Алгоритм преобразования:

1 Прочитывайте входное инфиксное выражение символ за символом.

2 Если текущий символ является числом (или операндом), добавьте его в выходную строку.

3 Если текущий символ является левой скобкой '(', поместите его в стек.

4 Если текущий символ является правой скобкой ')':

– выталкивайте символы из стека и добавляйте их в выходную строку до тех пор, пока не встретите левую скобку '('.

– удалите левую скобку из стека.

5 Если текущий символ является оператором (+, -, \*, /):

– выталкивайте символы из стека в выходную строку до тех пор, пока в стеке есть операторы с приоритетом большим или равным приоритету текущего оператора.

– поместите текущий оператор в стек.

6 После обработки всех символов входного выражения, выталкивайте оставшиеся операторы из стека в выходную строку.

Этот алгоритм преобразования инфиксного выражения в форму обратной польской записи учитывает приоритет операторов, что позволяет правильно выстроить порядок операций. Это упрощает последующее вычисление выражений и обеспечивает корректное выполнение арифметических операций.

**16. Понятие хеширования. Схемы хеширования.**

Хеширование — это процесс преобразования входных данных произвольной длины в строку фиксированной длины, которая называется хешем, хеш-кодом или хеш-значением. Хеширование широко используется в различных областях информатики, таких как хранение и поиск данных, криптография, контроль целостности данных и другие.

Основные характеристики хеширования:

1 Определенность: для одного и того же входного значения всегда генерируется одинаковый хеш.

2 Быстрота: хеширование должно выполняться быстро даже для больших объемов данных.

3 Распределение: хорошая хеш-функция распределяет входные данные равномерно по всем возможным значениям хеша.

4 Стойкость к коллизиям: коллизия — это ситуация, когда два разных входных значения дают один и тот же хеш. Хеш-функция должна минимизировать вероятность коллизий.

Существует несколько схем хеширования, каждая из которых имеет свои особенности и области применения. Рассмотрим основные из них:

1 Линейная адресация

Функция хеширования:

int hashFunction(int key, int tableSize) {

return key % tableSize;

}

Алгоритм разрешения конфликтов:

void insert(int key, int tableSize, int\* hashTable) {

int index = hashFunction(key, tableSize);

while (hashTable[index] != -1) {

index = (index + 1) % tableSize;

}

hashTable[index] = key;

}

int search(int key, int tableSize, int\* hashTable) {

int index = abs(key % tableSize);

while (hashTable[index] != -1) {

if (hashTable[index] == key) {

return index;

}

index = (index + 1) % tableSize;

}

return -1;

}

2 Квадратичная адресация

Функция хеширования:

int hashFunction(int key, int tableSize) {

return key % tableSize;

}

Алгоритм разрешения конфликтов:

void insert(int key, int tableSize, int\* hashTable) {

int index = hashFunction(key, tableSize);

int i = 0;

while (hashTable[(index + i \* i) % tableSize] != -1) {

i++;

}

hashTable[(index + i \* i) % tableSize] = key;

}

int search(int key, int tableSize, int\* hashTable) {

int index = hashFunction(key, tableSize);

int i = 0;

while (hashTable[(index + i \* i) % tableSize] != -1) {

if (hashTable[(index + i \* i) % tableSize] == key) {

return (index + i \* i) % tableSize;

}

i++;

}

return -1;

}

3 Произвольная адресация

Произвольная адресация использует заранее сгенерированный массив случайных чисел для поиска новой позиции в случае коллизии.

Функция хеширования:

int hashFunction(int key, int tableSize) {

return key % tableSize;

}

Алгоритм разрешения конфликтов:

void insert(int key, int tableSize, int\* hashTable, int\* randomTable) {

int index = hashFunction(key, tableSize);

int i = 0;

while (hashTable[(index + randomTable[i]) % tableSize] != -1) {

i++;

}

hashTable[(index + randomTable[i]) % tableSize] = key;

}

int search(int key, int tableSize, int\* hashTable, int\* randomTable) {

int index = hashFunction(key, tableSize);

int i = 0;

while (hashTable[(index + randomTable[i]) % tableSize] != -1) {

if (hashTable[(index + randomTable[i]) % tableSize] == key) {

return (index + randomTable[i]) % tableSize;

}

i++;

}

return -1;

}

4 Двойное хеширование

Функция хеширования:

int hashFunction1(int key, int tableSize) {

return key % tableSize;

}

int hashFunction2(int key, int tableSize) {

return 1 + (key % (tableSize - 2));

}

void insert(int key, int tableSize, int\* hashTable) {

int index = hashFunction1(key, tableSize);

int i = 0;

while (hashTable[(index + i \* hashFunction2(key, tableSize)) % tableSize] != -1) {

i++;

}

hashTable[(index + i \* hashFunction2(key, tableSize)) % tableSize] = key;

}

int search(int key, int tableSize, int\* hashTable) {

int index = hashFunction1(key, tableSize);

int i = 0;

while (hashTable[(index + i \* hashFunction2(key, tableSize)) % tableSize] != -1) {

if (hashTable[(index + i \* hashFunction2(key, tableSize)) % tableSize] == key) {

return (index + i \* hashFunction2(key, tableSize)) % tableSize;

}

i++;

}

return -1;

}

5 Метод связанного списка

Используется связный список для хранения всех элементов, которые имеют одинаковый хеш-код.

struct Node {

int key;

Node\* next;

};

void insert(Node\*\* hashTable, int key, int tableSize) {

int index = key % tableSize;

Node\* newNode = new Node();

newNode->key = key;

newNode->next = hashTable[index];

hashTable[index] = newNode;

}

Node\* search(Node\*\* hashTable, int key, int tableSize) {

int index = key % tableSize;

Node\* node = hashTable[index];

while (node != nullptr) {

if (node->key == key) {

return node;

}

node = node->next;

}

return nullptr;

}

Хеширование — это мощный инструмент для ускорения поиска данных. Существуют различные схемы хеширования, такие как линейная адресация, квадратичная адресация, произвольная адресация, двойное хеширование и метод цепочек, каждая из которых имеет свои особенности и области применения. Выбор подходящей схемы зависит от конкретных требований к системе, таких как размер данных, допустимая вероятность коллизий и требования к производительности.