Учреждение образования

«Белорусский государственный технологический университет»

Факультет информационных технологий

РЕФЕРАТ

Тема: «Карта в C++»

  Выполнил:

Студент 1 курса 10 группы

Жамойдо Артём Игоревич

Преподаватель:

Доцент, кандидат технических наук

Белодед Николай Иванович

2024, Минск

**Содержание**

[Основы карты 1](#_Toc164944392)

[Реализация на базе бинарного дерева поиска 2](#_Toc164944393)

[Балансировка дерева для оптимизации производительности 3](#_Toc164944394)

[Оптимизация операций с картой 4](#_Toc164944395)

[Сравнение с хеш-таблицами 5](#_Toc164944396)

[Заключение 6](#_Toc164944397)

# **Основы карты**

Карта является ключевым элементом в программировании, предоставляя ассоциативное отображение между ключами и значениями. Она играет критическую роль во многих приложениях, от базовых консольных утилит до сложных систем управления данными. Карта позволяет организовать данные в удобной форме, обеспечивая быстрый доступ, модификацию и удаление элементов.

# **Реализация на базе бинарного дерева поиска**

Для реализации карты на языке C++ можно использовать бинарное дерево поиска. Бинарное дерево поиска (BST) представляет собой структуру данных, где каждый узел содержит ключ, значение и указатели на левого и правого потомков. Эта структура обеспечивает эффективное упорядоченное хранение данных. Однако для поддержания высокой производительности необходима балансировка дерева. Без нее дерево может стать несбалансированным, что приведет к увеличению времени выполнения операций.

**Пример кода реализации карты на основе бинарного дерева поиска:**

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  // Определение структуры узла бинарного дерева поиска  struct Node {  int key; // Ключ узла  std::string value; // Значение узла  Node\* left; // Указатель на левого потомка  Node\* right; // Указатель на правого потомка  // Конструктор узла  Node(int k, const std::string& v) : key(k), value(v), left(nullptr), right(nullptr) {} | |
| };  // Класс для работы с бинарным деревом поиска  class BinarySearchTree {  private:  Node\* root; // Корневой узел дерева  // Рекурсивный метод для вставки узла в дерево  Node\* insertRecursive(Node\* current, int key, const std::string& value) {  // Если текущий узел равен nullptr, создаем новый узел и возвращаем его  if (current == nullptr) {  return new Node(key, value);  }  // Если ключ меньше ключа текущего узла, рекурсивно вставляем в левое поддерево  if (key < current->key) {  current->left = insertRecursive(current->left, key, value);  }  // Если ключ больше ключа текущего узла, рекурсивно вставляем в правое поддерево  else if (key > current->key) {  current->right = insertRecursive(current->right, key, value);  }  // Если ключ уже существует, обновляем значение узла  else {  current->value = value;  }  // Возвращаем текущий узел после вставки  return current;  }  // Рекурсивный метод для вывода элементов дерева в порядке возрастания ключей  void printInOrder(Node\* node) {  // Базовый случай: если узел равен nullptr, возвращаемся  if (node != nullptr) {  printInOrder(node->left); // Рекурсивно выводим левое поддерево  std::cout << node->key << ": " << node->value << std::endl; // Выводим текущий узел  printInOrder(node->right); // Рекурсивно выводим правое поддерево  }  }  public:  // Конструктор класса  BinarySearchTree() : root(nullptr) {}  // Метод для вставки элемента в дерево  void insert(int key, const std::string& value) {  root = insertRecursive(root, key, value);  }  // Метод для вывода элементов дерева  void print() {  std::cout << "Элементы дерева:" << std::endl;  printInOrder(root);  }  };  int main() {  setlocale(LC\_ALL, "RU");  // Создание объекта бинарного дерева поиска  BinarySearchTree bst;  // Вставка элементов в дерево  bst.insert(1, "один");  bst.insert(2, "два");  bst.insert(3, "три");  // Вывод элементов дерева  bst.print();  return 0;  } |

|  |
| --- |
|  |

# **Балансировка дерева для оптимизации производительности**

Балансировка бинарного дерева поиска, на котором основана реализация карты, является ключевым фактором для обеспечения высокой производительности. Методы, такие как красно-черные деревья и AVL-деревья, позволяют автоматически поддерживать оптимальную структуру дерева, обеспечивая эффективные операции вставки, удаления и поиска даже для больших объемов данных.

**Пример кода для балансировки бинарного дерева поиска:**

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  // Определение структуры узла бинарного дерева поиска  struct Node {  int key; // Ключ узла  std::string value; // Значение узла  Node\* left; // Указатель на левого потомка  Node\* right; // Указатель на правого потомка  // Конструктор узла  Node(int k, const std::string& v) : key(k), value(v), left(nullptr), right(nullptr) {}  };  // Класс для работы с бинарным деревом поиска  class BinarySearchTree {  private:  Node\* root; // Корневой узел дерева  // Метод для вращения узла влево  Node\* rotateLeft(Node\* node) { | |
| Node\* newRoot = node->right; // Новый корень - правый потомок текущего узла  node->right = newRoot->left; // Левое поддерево нового корня становится правым поддеревом текущего узла  newRoot->left = node; // Текущий узел становится левым потомком нового корня  return newRoot; // Возвращаем новый корень  }  // Метод для вращения узла вправо  Node\* rotateRight(Node\* node) {  Node\* newRoot = node->left; // Новый корень - левый потомок текущего узла  node->left = newRoot->right; // Правое поддерево нового корня становится левым поддеревом текущего узла  newRoot->right = node; // Текущий узел становится правым потомком нового корня  return newRoot; // Возвращаем новый корень  }  // Рекурсивный метод для вставки узла в дерево с балансировкой  Node\* insertRecursive(Node\* current, int key, const std::string& value) {  // Если текущий узел пустой, создаем новый узел  if (current == nullptr) {  return new Node(key, value);  }  // Вставка узла в левое или правое поддерево  if (key < current->key) {  current->left = insertRecursive(current->left, key, value);  }  else if (key > current->key) {  current->right = insertRecursive(current->right, key, value);  }  else {  // Если ключ уже существует, обновляем значение узла  current->value = value;  }  // Вычисление баланса для текущего узла  int balanceFactor = getHeight(current->left) - getHeight(current->right);  // Балансировка дерева  if (balanceFactor > 1) {  if (key < current->left->key) {  // LL вращение  return rotateRight(current);  }  else {  // LR вращение  current->left = rotateLeft(current->left);  return rotateRight(current);  }  }  else if (balanceFactor < -1) {  if (key > current->right->key) {  // RR вращение  return rotateLeft(current);  }  else {  // RL вращение  current->right = rotateRight(current->right);  return rotateLeft(current);  } | | |
| }  return current;  }  // Метод для получения высоты узла в дереве  int getHeight(Node\* node) {  if (node == nullptr) {  return 0;  }  // Рекурсивное вычисление высоты для левого и правого поддеревьев  int leftHeight = getHeight(node->left);  int rightHeight = getHeight(node->right);  // Возвращаем максимальную высоту плюс один для текущего узла  return std::max(leftHeight, rightHeight) + 1;  }  // Рекурсивный метод для вывода дерева в порядке возрастания ключей  void printInOrder(Node\* node) {  if (node != nullptr) {  // Рекурсивный обход в порядке лево-корень-право  printInOrder(node->left);  std::cout << node->key << ": " << node->value << " ";  printInOrder(node->right);  }  }  public:  // Конструктор класса  BinarySearchTree() : root(nullptr) {}  // Метод для вставки элемента в дерево  void insert(int key, const std::string& value) {  root = insertRecursive(root, key, value);  }  // Метод для вывода элементов дерева  void print() {  std::cout << "Элементы дерева (в порядке возрастания ключей):" << std::endl;  printInOrder(root);  std::cout << std::endl;  }  };  int main() {  setlocale(LC\_ALL, "RU");  // Создание объекта бинарного дерева поиска  BinarySearchTree bst;  // Вставка элементов в дерево  bst.insert(1, "один");  bst.insert(2, "два");  bst.insert(3, "три");  // Вывод элементов дерева  bst.print();  return 0;  } | | |

|  |
| --- |
|  |

# **Оптимизация операций с картой**

Эффективность работы с картой напрямую зависит от качества реализации алгоритмов. Оптимизация операций вставки, удаления и поиска включает в себя анализ и выбор подходящих методов, таких как использование кэширования, оптимизация пространственной сложности и адаптация под конкретные требования задачи.

# **Сравнение с хеш-таблицами**

Сравнение карты с хеш-таблицами позволяет оценить их относительные преимущества и недостатки. В отличие от карты, хеш-таблицы предлагают почти постоянное время выполнения операций благодаря хэшированию ключей. Однако у карты есть преимущество в упорядоченном хранении данных, что полезно для диапазонных запросов и обработки отсортированных данных.

**Преимущества карты:**

* **Упорядоченное хранение**: Карта обеспечивает упорядоченное хранение данных, что упрощает работу с отсортированными наборами данных и позволяет выполнять диапазонные запросы.
* **Эффективность операций**: Благодаря структуре бинарного дерева поиска, операции вставки, удаления и поиска имеют логарифмическую сложность, что обеспечивает высокую производительность.

**Недостатки карты:**

* **Сложность реализации**: Реализация карты может быть более сложной из-за потребности в балансировке дерева и управлении узлами.
* **Пространственная сложность**: Дополнительные указатели и структуры данных могут увеличивать использование памяти, особенно для больших наборов данных.

# **Заключение**

Реализация карты на C++ без использования стандартных коллекций представляет собой интересное исследование в области структур данных. Это обеспечивает глубокое понимание алгоритмов и методов оптимизации, необходимых для эффективной работы с данными. Выбор между картой и хеш-таблицами зависит от конкретных требований задачи, объема данных и предпочтений разработчика. Глубокое понимание особенностей и преимуществ каждой структуры данных помогает выбирать наилучший инструмент для решения различных задач.