Учреждение образования

«Белорусский государственный технологический университет»

Факультет информационных технологий

РЕФЕРАТ

Тема: «Обслуживание очереди с приоритетом»

  Выполнил:

Студент 1 курса 10 группы

Жамойдо Артём Игоревич

Преподаватель:

Доцент, кандидат технических наук

Белодед Николай Иванович

2024, Минск

**Содержание**

[Введение в очередь с приоритетом 1](#_Toc164993927)

[Реализация на основе массива 2](#_Toc164993928)

[Реализация на основе кучи 3](#_Toc164993929)

[Сравнение методов реализации 4](#_Toc164993930)

[Практическое применение 5](#_Toc164993931)

[Оптимизация методов 6](#_Toc164993932)

[Заключение и перспективы развития 7](#_Toc164993933)

# **Введение в очередь с приоритетом**

Очередь с приоритетом — это абстрактная структура данных, которая позволяет хранить элементы с ассоциированным с ними приоритетом. В отличие от обычной очереди, где элементы обрабатываются в порядке их поступления, в очереди с приоритетом элементы извлекаются на основе их приоритета.

**Основные характеристики и преимущества**

Очередь с приоритетом предоставляет эффективный способ управления ресурсами, где элементы нужно обработать в определенном порядке. Её преимущества включают быструю вставку и извлечение элементов с высокой степенью организации данных.

**Интересный факт:** Очередь с приоритетом была изначально разработана в 1960-х годах и с тех пор активно применяется во многих областях, начиная от операционных систем и заканчивая алгоритмами искусственного интеллекта.

# **Реализация на основе массива**

Одним из способов реализации очереди с приоритетом является использование массива. В этом случае, элементы хранятся в массиве в порядке их приоритета. При добавлении нового элемента, его место определяется на основе приоритета. При удалении из очереди извлекается элемент с наивысшим приоритетом.

|  |
| --- |
| #include <iostream>  // Класс реализации очереди с приоритетом на основе массива  class PriorityQueueArray {  private:  // Структура для хранения элемента с приоритетом  struct Node {  int priority; // Приоритет элемента  int value; // Значение элемента  };  Node\* data; // Указатель на массив элементов очереди  int capacity; // Емкость массива  int size; // Текущий размер массива  public:  // Конструктор класса  PriorityQueueArray(int capacity) : capacity(capacity), size(0) {  data = new Node[capacity]; // Выделение памяти под массив  }  // Деструктор класса  ~PriorityQueueArray() {  delete[] data; // Освобождение памяти  }  // Метод для добавления элемента в очередь с приоритетом  void push(int priority, int value) {  // Если массив полон, увеличиваем его размер вдвое  if (size == capacity) {  expandCapacity();  }  // Находим правильное место для нового элемента на основе приоритета  int i = size - 1;  while (i >= 0 && data[i].priority < priority) {  data[i + 1] = data[i]; // Сдвигаем элементы для освобождения места под новый элемент  i--;  }  // Вставляем новый элемент на найденное место  data[i + 1] = { priority, value };  size++; // Увеличиваем размер массива  }  // Метод для извлечения элемента с наивысшим приоритетом  int pop() {  int value = data[size - 1].value; // Получаем значение элемента с наивысшим приоритетом  size--; // Уменьшаем размер массива  return value; // Возвращаем значение удаленного элемента  }  private:  // Метод для увеличения емкости массива вдвое  void expandCapacity() {  capacity \*= 2; // Увеличиваем емкость массива вдвое  Node\* newData = new Node[capacity]; // Создаем новый массив большего размера  for (int i = 0; i < size; ++i) {  newData[i] = data[i]; // Копируем элементы из старого массива в новый  }  delete[] data; // Освобождаем память, занятую старым массивом  data = newData; // Переназначаем указатель на новый массив  }  }; |

|  |
| --- |
| int main() {  // Создание объекта очереди с приоритетом на основе массива  PriorityQueueArray queue(3); // Начальная емкость массива = 3  // Добавление элементов в очередь с разными приоритетами  queue.push(3, 10); // Приоритет = 3, значение = 10  queue.push(1, 20); // Приоритет = 1, значение = 20  queue.push(2, 5); // Приоритет = 2, значение = 5  // Извлечение и вывод элементов с приоритетом  std::cout << "PriorityQueueArray:\n";  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 20  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 5  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 10  return 0; // Возврат нуля для завершения программы  } |

|  |
| --- |
|  |

# **Реализация на основе кучи**

Куча — это двоичное дерево, у которого выполняются определенные условия. В контексте очереди с приоритетом, куча позволяет быстро находить и извлекать элемент с наивысшим приоритетом. Добавление элемента в кучу происходит за логарифмическое время, а удаление — также за логарифмическое время.

|  |  |  |
| --- | --- | --- |
| #include <iostream>  // Класс реализации очереди с приоритетом на основе кучи  class PriorityQueueHeap {  private:  // Структура для хранения элемента с приоритетом  struct Node {  int priority; // Приоритет элемента  int value; // Значение элемента  };  // Массив для хранения кучи  Node\* heap; // Указатель на динамический массив  int capacity; // Емкость кучи  int size; // Текущий размер кучи | | |
| // Метод для восстановления свойства кучи при добавлении элемента  void heapifyUp(int index) {  // Продолжаем, пока не достигнем корня или пока текущий узел не станет корнем  while (index != 0 && heap[parent(index)].priority > heap[index].priority) {  // Обмен текущего узла с родительским узлом для восстановления свойства кучи  swapNodes(index, parent(index));  // Переход к родительскому узлу  index = parent(index);  }  }  // Метод для восстановления свойства кучи при удалении элемента  void heapifyDown(int index) {  int minIndex = index;  int leftChild = left(index);  int rightChild = right(index);  // Если левый дочерний элемент существует и его приоритет меньше текущего минимального  if (leftChild < size && heap[leftChild].priority < heap[minIndex].priority)  {  minIndex = leftChild;  }  // Если правый дочерний элемент существует и его приоритет меньше текущего минимального  if (rightChild < size && heap[rightChild].priority < heap[minIndex].priority) {  minIndex = rightChild;  }  // Если текущий индекс не равен минимальному индексу, выполняем обмен и рекурсивно вызываем heapifyDown  if (index != minIndex) {  // Обмен текущего узла с узлом с минимальным приоритетом для восстановления свойства кучи  swapNodes(index, minIndex);  // Рекурсивный вызов для следующего уровня  heapifyDown(minIndex);  }  }  // Методы для вычисления индексов родительского и дочерних узлов  int parent(int i) { return (i - 1) / 2; } // Вычисление индекса родительского узла  int left(int i) { return 2 \* i + 1; } // Вычисление индекса левого дочернего узла  int right(int i) { return 2 \* i + 2; } // Вычисление индекса правого дочернего узла  // Метод для обмена двух узлов в куче  void swapNodes(int i, int j) {  Node temp = heap[i];  heap[i] = heap[j];  heap[j] = temp;  }  public:  // Конструктор класса |
| PriorityQueueHeap(int capacity) : capacity(capacity), size(0) {  heap = new Node[capacity]; // Выделение памяти под массив кучи  }  // Деструктор класса  ~PriorityQueueHeap() {  delete[] heap; // Освобождение памяти, выделенной для массива кучи  }  // Метод для добавления элемента в очередь с приоритетом  void push(int priority, int value) {  Node newNode = { priority, value };  // Добавление нового элемента в конец кучи  heap[size] = newNode;  size++;  // Восстановление свойства кучи после добавления  heapifyUp(size - 1);  }  // Метод для извлечения элемента с наивысшим приоритетом  int pop() {  int value = heap[0].value;  // Замена корневого элемента последним элементом в куче  heap[0] = heap[size - 1];  size--;  // Восстановление свойства кучи после удаления  heapifyDown(0);  return value;  }  };  int main() {  // Создание объекта очереди с приоритетом на основе кучи  PriorityQueueHeap queue(10); // Создание объекта очереди с емкостью 10  // Добавление элементов в очередь с разными приоритетами  queue.push(3, 10); // Приоритет = 3, значение = 10  queue.push(1, 20); // Приоритет = 1, значение = 20  queue.push(2, 5); // Приоритет = 2, значение = 5  // Извлечение и вывод элементов с приоритетом  std::cout << "PriorityQueueHeap:\n";  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 20  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 5  std::cout << queue.pop() << "\n"; // Ожидаемый результат: 10  return 0; // Возврат нуля для завершения программы  } | | | | |

|  |
| --- |
|  |

# **Сравнение методов реализации**

Методы, основанные на куче, обычно предоставляют более эффективные операции добавления и удаления по сравнению с методами на массиве. Куча обеспечивает стабильное время выполнения операций независимо от размера очереди, в то время как массив требует дополнительных усилий для поддержания порядка элементов при каждой операции.

# **Практическое применение**

Очередь с приоритетом часто используется в алгоритмах, таких как алгоритм Дейкстры, для поиска кратчайшего пути или в планировании задач. В медицинских системах она может использоваться для определения срочности обслуживания пациентов или в авиационной индустрии для управления вылетами и посадками. В кибербезопасности очередь с приоритетом может быть использована для обработки угроз в порядке их серьезности.

# **Оптимизация методов**

Для улучшения производительности работы с очередью с приоритетом можно применять различные оптимизационные методы:

**Комбинированный подход**

Эффективность можно повысить, комбинируя методы на основе массива и кучи. Элементы с высоким приоритетом хранятся в куче, элементы с низким приоритетом — в массиве.

**Ленивая перестройка кучи**

Перестройка кучи может быть выполнена только при необходимости, например, при извлечении элемента с наивысшим приоритетом.

**Использование кэша**

Кэширование часто используемых элементов или результатов операций может сократить время доступа к данным и улучшить производительность.

**Параллелизм и многопоточность**

Параллельная обработка операций может увеличить общую пропускную способность системы, распределяя нагрузку между несколькими ядрами процессора.

# **Заключение и перспективы развития**

Очередь с приоритетом — это мощный инструмент для эффективного управления ресурсами и задачами с различными уровнями приоритета. Реализация этой структуры данных может быть адаптирована в зависимости от конкретных требований приложения для достижения оптимальной производительности. Дальнейшие исследования могут включать разработку новых алгоритмов и методов реализации, а также их применение в новых областях, таких как искусственный интеллект, машинное обучение и большие данные. Это может помочь в оптимизации работы систем и улучшении качества обслуживания в различных отраслях.