|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  Калужский филиал  федерального государственного бюджетного  образовательного учреждения высшего образования  ***«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»***  ***(КФ МГТУ им. Н.Э. Баумана)*** |

**ФАКУЛЬТЕТ** ***ИУК «Информатика и управление»***

**КАФЕДРА** \_\_***ИУК4 «Программное обеспечение ЭВМ, информационные технологии»***

**ЛАБОРАТОРНАЯ РАБОТА №3**

**«Реализация очереди с приоритетами»**

**ДИСЦИПЛИНА: «Логика и теория алгоритмов»**

|  |  |  |
| --- | --- | --- |
| Выполнил: студент гр. ИУК4-42Б | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Карельский М.К. )  (Подпись) |
| Проверил: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Амеличев Г.Э. )  (Подпись) |
| Дата сдачи (защиты):  Результаты сдачи (защиты): | | |
|  | - Балльная оценка:  - Оценка: | |

Калуга, 2022

**Цель:** приобрести практические навыки реализации очереди с приоритетами на основе бинарной пирамиды.

**Задачи:** реализовать заданную структуру данных и исследовать вычислительную сложность основных операций с ней.

**Вариант №7**

Невозрастающая очередь с приоритетами

* Приоритет – целое число
* Данные – обобщенный тип

Операции:

* Enqueue
* Dequeue
* Peek
* IncreasePriority

Реализация:

* Бинарная пирамида

**Описание:**

Очереди с приоритетом – разновидность очередей, в которой у каждого элемента есть свой приоритет. Обслуживаются они в соответствии со своим приоритетом. Реализуются с помощью куч.

Куча (heap) – это частный случай двоичного дерева, в котором узлы-родители сравниваются по значениям с узлами-потомками и располагаются соответствующим образом.

**Операции:**

*Heapify: O(log2(n/i))*

Меняем узел местами с наибольшим потомком и повторяем операцию с ним.

void Heapify(int i) // O(log2(n/i))

{

int size = \_heap.size(); // c1

int largest = i; // c2

int l = 2 \* i + 1; // c3

int r = 2 \* i + 2; // c4

if (l < size && \_heap[l].priority > \_heap[largest].priority) // c5

largest = l; // p1 \* c6

if (r < size && \_heap[r].priority > \_heap[largest].priority) // c7

largest = r; // p2 \* c8

if (largest != i) // c9

{

Swap(&\_heap[i], &\_heap[largest]); // c10

Heapify(largest); // c11 \* log2(n/i)

}

}

*HeapifyOnce: O(1)*

Меняем узел местами с наибольшим потомком.

void HeapifyOnce(int i) // O(1)

{

int size = \_heap.size(); // c1

int largest = i; // c2

int l = 2 \* i + 1; // c3

int r = 2 \* i + 2; // c4

if (l < size && \_heap[l].priority > \_heap[largest].priority) // c5

largest = l; // p1 \* c6

if (r < size && \_heap[r].priority > \_heap[largest].priority) // c7

largest = r; // p2 \* c8

Swap(&\_heap[i], &\_heap[largest]); // c9

}

*Enqueue: O(log2(n))*

Добавляем элемент в конец и проводим восходящую сортировку.

void Enqueue(int priority, T data) // O(log2(n))

{

\_heap.push\_back({ priority, data }); // c1

int size = \_heap.size(); // c2

if (size != 1) // c3

for (int i = size / 2 - 1; i >= 0; i = (i + 1) / 2 - 1) // c4 \* log2(n)

HeapifyOnce(i); // c5 \* log2(n)

}

*Dequeue: O(log2(n))*

Меняем корень с последним элементом, удаляем в новой куче последний элемент и проводим нисходящую сортировку от корня.

T Dequeue() // O(log2(n))

{

T data = \_heap[0].data; // c1

int size = \_heap.size(); // c2

Swap(&\_heap[0], &\_heap[size - 1]); // c3

\_heap.pop\_back(); // c4

Heapify(0); // c5 \* log2(n)

return data; // c6

}

*Peek: O(1)*

Возвращаем корень.

T Peek() { return \_heap[0].data; } // O(1)

*IncreasePriority: O(log2(n))*

Повышаем приоритет, проводим восходящую сортировку.

void IncreasePriority(T data, unsigned short increase) // O(log2(n))

{

int size = \_heap.size(); // c1

int i; // c2

for (i = 0; i < size; i++) // c3 \* n

if (data == \_heap[i].data) // c4 \* n

break; // t1 \* c5

\_heap[i].priority += increase; // c6

for (int j = (i + 1) / 2 - 1; j >= 0; j = (j + 1) / 2 - 1) // c7 \* log2(n)

HeapifyOnce(j); // c8 \* log2(n)

}

**Производительность алгоритмов структуры:**

Любой алгоритм работы с данной структурой представляет собой:

* N1 операций Enqueue сложности O(log2(n))
* N2 ≤ N1 операций Dequeue сложности O(log2(n))
* N3 операций Peek сложности O(1)
* N4 операций IncreasePriority сложности O(log2(n))

N1 + N2 + N3 + N4 = N

(N1 + N2 + N4) \* O(log2(n)) + N3 \* O(1) ≤ N \* O(log2(n))

Тогда согласно методу усреднения амортизационная стоимость операций

**Результаты:**

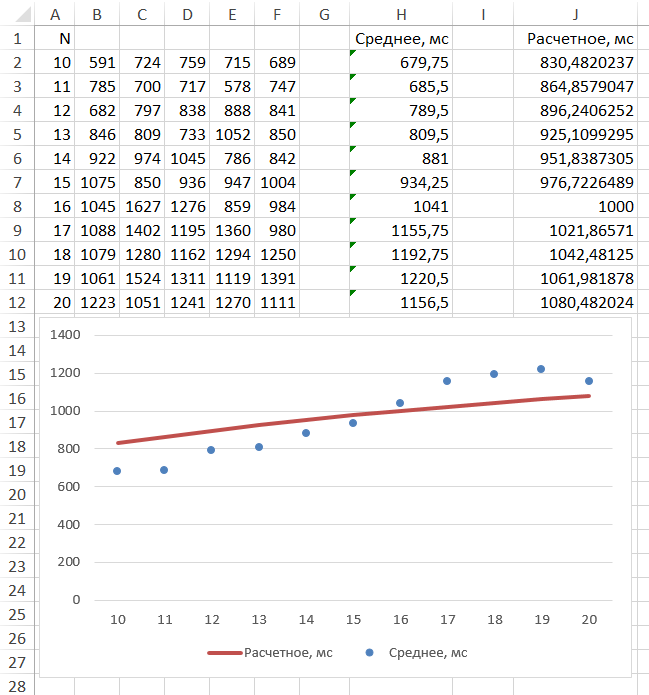


Рис. 1. Время выполнения

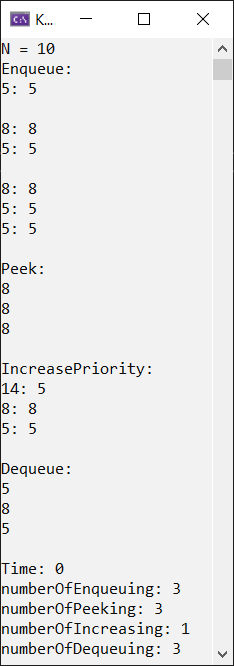


Рис. 2. Результат для N = 10

**Вывод:** в ходе выполнения лабораторной работы были получены практические навыки реализации очереди с приоритетом, кучи, использования метода усреднения амортизационного анализа.

**ПРИЛОЖЕНИЯ**

**Листинг:**

#pragma once

#include <vector>

#include <iostream>

namespace LITA

{

template <typename T>

class PriorityQueue

{

public:

PriorityQueue() : \_heap() {}

void Enqueue(int priority, T data) // O(log2(n))

{

\_heap.push\_back({ priority, data }); // c1

int size = \_heap.size(); // c2

if (size != 1) // c3

for (int i = size / 2 - 1; i >= 0; i = (i + 1) / 2 - 1) // c4 \* log2(n)

HeapifyOnce(i); // c5 \* log2(n)

}

T Dequeue() // O(log2(n))

{

T data = \_heap[0].data; // c1

int size = \_heap.size(); // c2

Swap(&\_heap[0], &\_heap[size - 1]); // c3

\_heap.pop\_back(); // c4

Heapify(0); // c5 \* log2(n)

return data; // c6

}

T Peek() { return \_heap[0].data; } // O(1)

void IncreasePriority(T data, unsigned short increase) // O(log2(n))

{

int size = \_heap.size(); // c1

int i; // c2

for (i = 0; i < size; i++) // c3 \* n

if (data == \_heap[i].data) // c4 \* n

break; // t1 \* c5

\_heap[i].priority += increase; // c6

for (int j = (i + 1) / 2 - 1; j >= 0; j = (j + 1) / 2 - 1) // c7 \* log2(n)

HeapifyOnce(j); // c8 \* log2(n)

}

void PrintArray()

{

for (int i = 0; i < \_heap.size(); ++i)

std::cout << \_heap[i].priority << ": " << \_heap[i].data << "\n";

}

private:

struct Pair

{

int priority{};

T data{};

};

void Swap(Pair\* a, Pair\* b) // O(1)

{

Pair temp = \*b; // c1

\*b = \*a; // c2

\*a = temp; // c3

}

void Heapify(int i) // O(log2(n/i))

{

int size = \_heap.size(); // c1

int largest = i; // c2

int l = 2 \* i + 1; // c3

int r = 2 \* i + 2; // c4

if (l < size && \_heap[l].priority > \_heap[largest].priority) // c5

largest = l; // p1 \* c6

if (r < size && \_heap[r].priority > \_heap[largest].priority) // c7

largest = r; // p2 \* c8

if (largest != i) // c9

{

Swap(&\_heap[i], &\_heap[largest]); // c10

Heapify(largest); // c11 \* log2(n/i)

}

}

void HeapifyOnce(int i) // O(1)

{

int size = \_heap.size(); // c1

int largest = i; // c2

int l = 2 \* i + 1; // c3

int r = 2 \* i + 2; // c4

if (l < size && \_heap[l].priority > \_heap[largest].priority) // c5

largest = l; // p1 \* c6

if (r < size && \_heap[r].priority > \_heap[largest].priority) // c7

largest = r; // p2 \* c8

Swap(&\_heap[i], &\_heap[largest]); // c9

}

std::vector<Pair> \_heap{};

};

}